PGI® Tools Guide

Parallel Tools for Scientists and Engineers

The Portland Group™
STMicroelectronics
Two Centerpointe Drive, Suite 320
Lake Oswego, OR 97035

Printed in the United States of America

First Printing:Release 5.2, June 2004
Second Printing:Release 6.0, March 2005
Third Printing:Release 6.1, December 2005
Fourth Printing:Release 6.2, August 2006
Fifth Printing: Release 7.0-1, December, 2006
Sixth Printing: Release 7.0-2, February, 2007

Technical support:	http://www.pgroup.com/support/
Sales:	sales@pgroup.com
Web:	http://www.pgroup.com

Contents

Tables

Figures

Preface

This guide describes how to use the PGPROF profiler and PGDBG debugger to tune and debug serial and parallel applications built with The Portland Group (PGI) Fortran, C, and C++ for X86, AMD64 and EM64T processor-based systems. It contains information about how to use the tools, as well as detailed reference information on commands and graphical interfaces.

Intended Audience

This guide is intended for application programmers, scientists and engineers proficient in programming with the Fortran, C, and/or C++ languages. The PGI tools are available on a variety of operating systems for the X86, AMD64, and EM64T hardware platforms. This guide assumes familiarity with basic operating system usage.

Supplementary Documentation

See http://www.pgroup.com/docs.htm for the PGDBG documentation updates. Documentation delivered with PGDBG should be accessible on an installed system by accessing $PGI/docs/index.htm. See http://www.pgroup.com/faq/index.htm for frequently asked PGDBG questions and answers.

Compatibility and Conformance to Standards

The PGI compilers and tools run on a variety of systems. They produce and/or process code that conforms to the ANSI standards for FORTRAN 77, Fortran 95, C, and C++ and includes extensions from MIL-STD-1753, VAX/VMS Fortran, IBM/VS Fortran, SGI Fortran, Cray Fortran, and K&R C. PGF77, PGF90, PGCC ANSI C, and C++ support parallelization extensions based on the OpenMP defacto standard. PGHPF supports data parallel extensions based on the High Performance Fortran (HPF) defacto standard. The PGI Fortran Reference Manual describes Fortran statements and extensions as implemented in the PGI Fortran compilers. PGDBG permits debugging of serial and parallel (multi-threaded, OpenMP and/or MPI) programs compiled with PGI compilers. PGPROF permits profiling of serial and parallel (multi-threaded, OpenMP and/or MPI) programs compiled with PGI compilers.

For further information, refer to the following:

- American National Standard Programming Language FORTRAN, ANSI X3. -1978 (1978).

- ISO/IEC 1539:1991, Information technology – Programming Languages – Fortran, Geneva, 1991 (Fortran 90).
- ISO/IEC 1539:1997, Information technology – Programming Languages – Fortran, Geneva, 1997 (Fortran 95).
- High Performance Fortran Language Specification, Revision 1.0, Rice University, Houston, Texas (1993), http://www.crpc.rice.edu/HPFF.
- High Performance Fortran Language Specification, Revision 2.0, Rice University, Houston, Texas (1997), http://www.crpc.rice.edu/HPFF.
- OpenMP Application Program Interface, Version 2.5, May 2005, http://www.openmp.org.
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).
- American National Standard Programming Language C, ANSI X3.159-1989.
- ISO/IEC 9899:1999, Information technology – Programming Languages – C, Geneva, 1999 (C99).
- HPDF Standard (High Performance Debugging Forum) http://www.ptools.org/hpdf/draft/intro.html

Organization

This manual is organized as follows:

Chapter 1	This chapter describes PGDBG, a symbolic debugger for Fortran, C, C++ and assembly language programs.
	"Definition of Terms" on page 1 through "PGDBG Invocation and Initialization" on page 2 describe how to build a target application for debug and invoke PGDBG.
	"PGDBG Graphical User Interface" on page 5 describes how to use the PGDBG graphical user interface (GUI).

	"PGDBG Command Language" on page 31 through "PGDBG Command Reference" on page 51 provide detailed information about the PGDBG command language, which can be used from the command-line user interface or from the command panel of the graphical user interface.
	"Signals" on page 87 through "Debugging with Core Files" on page 98 give some detail on how PGDBG interacts with signals, how to access registers, language-specific issues, and debugging with core files.
	"Debugging Parallel Programs" on page 100 through "MPI Debugging" on page 136 describe the parallel debugging capabilities of PGDBG and how to use them.
Chapter 2	The PGPROF Profiler chapter describes the PGPROF Profiler. This tool analyzes data generated during execution of specially compiled C, C++, F77, F95, and HPF programs.

Conventions

This guide uses the following conventions:

italic	is used for commands, filenames, directories, arguments, options and for emphasis.
`Constant Width`	is used in examples and for language statements in the text, including assembly language statements.
[item1]	in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.
{ item2 | item 3}	braces indicate that a selection is required. In this case, you must select either item2 or item3.
filename ...	ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.
FORTRAN	Fortran language statements are shown in the text of this guide using a reduced fixed point size.

C/C++	C/C++ language statements are shown in the test of this guide using a reduced fixed point size.

The PGI compilers and tools are supported on both 32-bit and 64-bit variants of the Linux and Windows operating systems on a variety of x86-compatible processors. There are a wide variety of releases and distributions of each of these types of operating systems. The PGI User's Guide defines the following terms with respect to these platforms:

x86	a processor designed to be binary compatible with i386/i486 and previous generation processors from Intel* Corporation.
IA32	an Intel Architecture 32-bit processor designed to be binary compatible with x86 processors, but incorporating new features such as streaming SIMD extensions (SSE) for improved performance.
AMD64	a 64-bit processor from AMD designed to be binary compatible with IA32 processors, and incorporating new features such as additional registers and 64-bit addressing support for improved performance and greatly increased memory range.
EM64T	a 64-bit IA32 processor with Extended Memory 64-bit Technology extensions that are binary compatible with AMD64 processors. This includes the Intel Pentium 4, Intel Xeon, and Intel core 2 processors.
linux86	32-bit Linux operating system running on an x86, AMD64 or EM64T processor-based system, with 32-bit GNU tools, utilities and libraries used by the PGI compilers to assemble and link for 32-bit execution.
linux86-64	64-bit Linux operating system running on an AMD64 or EM64T processor-based system, with 64-bit and 32-bit GNU tools, utilities and libraries used by the PGI compilers to assemble and link for execution in either linux86 or linux86-64 environments. The 32-bit development tools and execution environment under linux86-64 are considered a cross development environment for x86 processor-based applications.
SFU	Services for Unix, a 32-bit-only predecessor of SUA, the Subsystem for Unix Applications. See SUA.
SUA	Subsystem for UNIX-based Applications (SUA) is source-compatibility subsystem for compiling and running custom UNIX-based applications on a computer running 32-bit or 64-bit Windows server-class operating

	system. It provides an operating system for Portable Operating System Interface (POSIX) processes. SUA supports a package of support utilities (including shells and >300 Unix commands), case-sensitive file names, and job control. The subsystem installs separately from the Windows kernel to support UNIX functionality without any emulation.
Win32	any of the 32-bit Microsoft* Windows* Operating Systems (XP/2000/Server 2003) running on an x86, AMD64 or EM64T processor-based system. On these targets, the PGI compiler products include additional tools and libraries needed to build executables for 32-bit Windows systems.
Win64	any of the 64-bit Microsoft* Windows* Operating Systems (XP Professional /Windows Server 2003 x64 Editions) running on an AMD64 or EM64T processor-based system.

Related Publications

The following documents contain additional information related to the X86 architecture and the compilers and tools available from The Portland Group.

- PGI Fortran Reference Manual describes the FORTRAN 77, Fortran 90/95, and HPF statements, data types, input/output format specifiers, and additional reference material related to the use of PGI Fortran compilers.
- System V Application Binary Interface Processor Supplement by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- FORTRAN 95 HANDBOOK, Complete ANSI/ISO Reference (The MIT Press, 1997).
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- The C Programming Language by Kernighan and Ritchie (Prentice Hall).
- C: A Reference Manual by Samuel P. Harbison and Guy L. Steele Jr. (Prentice Hall, 1987).
- The Annotated C++ Reference Manual by Margaret Ellis and Bjarne Stroustrup, AT&T Bell Laboratories, Inc. (Addison-Wesley Publishing Co., 1990)
- PGI User's Guide, PGI Tools Guide, PGI Release Notes, FAQ, Tutorials, http://www.pgroup.com/

- MPI-CH http://www.unix.mcs.anl.gov/mpi/mpich /
- OpenMP http://www.openmp.org/
- Ptools (Parallel Tools Consortium) http://www.ptools.org/
- PAPI (Performance Application Program Interface) http://icl.cs.utk.edu/papi/
- HPDF (High Performance Debugging Forum) Standard http://www.ptools.org/hpdf/draft/intro.html

System Requirements

- Linux or Windows (See http://www.pgroup.com/faq/install.htm for supported releases)
- Intel x86 (and compatible), AMD Athlon or AMD64, or Intel EM64T or Core2 processor

1 The PGDBG Debugger

PGDBG is a symbolic debugger for Fortran, C, C++ and assembly language programs. It provides typical debugger features, such as execution control using breakpoints and single-stepping, as well as examination and modification of application variables, memory locations, and registers. In addition, PGDBG supports debugging of certain types of parallel applications, depending on the operating system on the target machine.

- linux86 and linux86-64:
    - Multi-threaded and OpenMP Linux applications.
    - MPI applications on Linux clusters.
    - Hybrid applications, which use multiple threads or OpenMP as well as multiple MPI processes on Linux clusters.
- Win32 and Win64:
    - Multi-threaded and OpenMP Windows applications.

Multi-threaded and OpenMP applications may be run using more threads than the available number of CPUs, and MPI applications may allocate more than one process to a cluster node. PGDBG supports debugging the listed types of applications regardless of how well the number of threads match the number of CPUs or how well the number of processes match the number of cluster nodes.

Definition of Terms

Host	The system on which PGDBG executes. This will generally be the system where source and executable files reside, and where compilation is performed.
Target	A program being debugged.
Target Machine	The system on which a target runs. This may or may not be the same system as the host.

For an introduction to terminology used to describe parallel debugging, see “Summary of Parallel Debugging Features” on page 100.

Building Applications for Debug

To build an application for debug, compile with the –g option. With this option, the compiler will generate information about the symbols and source files in the program and include it in the executable file. The –g option also sets the compiler optimization to level zero (no optimization) unless you specify optimization options (such as –O, -fast, or –fastsse) on the command line. Optimization options take effect whether they are listed before or after –g on the command line. Programs built with –g and optimization levels higher than –O0 can be debugged, but due to transformations made to the program during optimization, source-level debugging may not be reliable. Machine-level debugging (e.g., accessing registers, viewing assembly code, etc.) will be reliable, even with optimized code. Programs built without –g can be debugged; however, information about types, local variables, arguments and source file line numbers will not be available.

In programs built with both –g and optimization levels higher than –O0, some optimizations may be disabled or otherwise affected by the –g option, possibly changing the program behavior. An alternative option, -gopt, can be used to build programs with full debugging information, but without modifying program optimizations. Unlike –g, the –gopt option does not set optimization to level zero.

To build an application for debug on Windows platforms, applications must be linked with the –g option as well as compiled with -g. This will result in the generation of debug information stored in a '.dwf' file and a '.pdb' file. The PGI compiler driver should always be used to link applications; the linker should never be invoked directly.

PGDBG Invocation and Initialization

PGDBG includes both a command-line interface and a graphical user interface (GUI). Text commands are entered one line at a time through the command-line interface. The GUI interface supports command entry through a point-and-click interface, a view of source and assembly code, a full command-line interface panel, and several other graphical elements and features. "PGDBG Command Language" on page 31 through "PGDBG Command Reference" on page 51 describe in detail how to use the PGDBG command-line interface. "PGDBG Graphical User Interface" on page 5 describes how to use the PGDBG GUI.

Invoking PGDBG

PGDBG is invoked using the pgdbg command as follows:

```
% pgdbg arguments target arg1 arg2 ... argn
```

where *arguments* may be any of the command-line arguments described in "PGDBG Command-Line Options" on page 4. See "Invoking PGDBG for MPI Debugging" on page 103 for instructions on how to debug an MPI program [Linux Only].

The *target* parameter is the name of the program executable file being debugged. The arguments *arg1 arg2 … argn* are the command-line arguments to the target program. Invoking PGDBG as described will start the PGDBG Graphical User Interface (GUI) (see Section "PGDBG Graphical User Interface" on page 5). For users who prefer to use a command-line interface, PGDBG may be invoked with the –text parameter (see "PGDBG Command-Line Options" on page 4 and "PGDBG Command Language" on page 31).

Note that the command shell will interpret any I/O redirection specified on the PGDBG command line. See "Process Control" on page 51 for a description of how to redirect I/O using the run command.

Both 32-bit and 64-bit applications are supported. In general, the PATH is set to the native architecture. If the PATH environment variable is set to use the 32-bit PGI tools, a 64-bit application can be debugged by invoking PGDBG with the –tp option. Conversely, if the PATH environment variable is set to use the 64-bit PGI tools, a 32-bit application can be debugged by invoking PGDBG with the –tp option. See "PGDBG Command-Line Options" on page 4 for details.

Once PGDBG is started, it reads symbol information from the executable file, then loads the application into memory. For large applications this process can take a few moments.

If an initialization file named .pgdbgrc exists in the current directory or in the home directory (as defined by the environment variable HOME), it is opened and PGDBG executes the commands in the file. The initialization file is useful for defining common aliases, setting breakpoints and for other startup commands. If an initialization file is found in the current directory, then the initialization file in the home directory, if there is one, is ignored. However, a script command placed in the initialization file may execute the initialization file in the home directory, or execute PGDBG commands in any other file (for example in the file .dbxinit for users who have an existing dbx debugger initialization file).

After processing the initialization file, PGDBG is ready to process commands. Normally, a session begins by setting one or more breakpoints, using the break, stop or trace commands, and then issuing a run command followed by cont, step, trace or next.

Selecting a Version of Java

The PGDBG graphical user interface (GUI) depends on Java. PGDBG command line mode (pgdbg -text) does not depend on Java. PGDBG requires that the Java Virtual Machine be a specific mimimum version or above. By default, PGDBG will use the version of Java installed with your PGI software; if you chose not to install Java when installing your PGI software, PGDBG will look for Java on your PATH. Both of these can be overriden by setting the PGI_JAVA environment variable to the full path of the Java executable you wish to use. For example, on a Linux system using the bash shell:

```
$ export PGI_JAVA=/home/myuser/myjava/bin/java
```

PGDBG Command-Line Options

The pgdbg command accepts several command line arguments that must appear on the command line before the name of the program being debugged. The valid options are:

-dbx	Start the debugger in dbx mode, which provides a dbx-like debugger command language.
-s *startup*	The default initialization file is ~/.pgdbgrc. The –s option specifies an alternate initialization file *startup*.
-c "*command*"	Execute the debugger *command* command (command must be in double quotes) before executing the commands in the startup file.
-r	Run the debugger without first waiting for a command. If the program being debugged runs successfully, the debugger terminates. Otherwise, the debugger is invoked and stops when an exception occurs.
-text	Run the debugger using a command-line interface (CLI). The default is for the debugger to launch in graphical user interface (GUI) mode.
-tp px, -tp k8-32	Debug a 32-bit program running on under a 64-bit operating system. This option is valid under the 64-bit version of PGDBG only.
-tp p7-64, -tp k8-64	Debug a 64-bit program running under a 64-bit operating system. This option is valid under the 64-bit version of PGDBG only.
–help	Display a list of command-line arguments (this list).
–I <*directory*>	Adds <*directory*> to the list of directories that PGDBG uses to search for source files. This option may be used multiple times to add multiple directories to the search path.

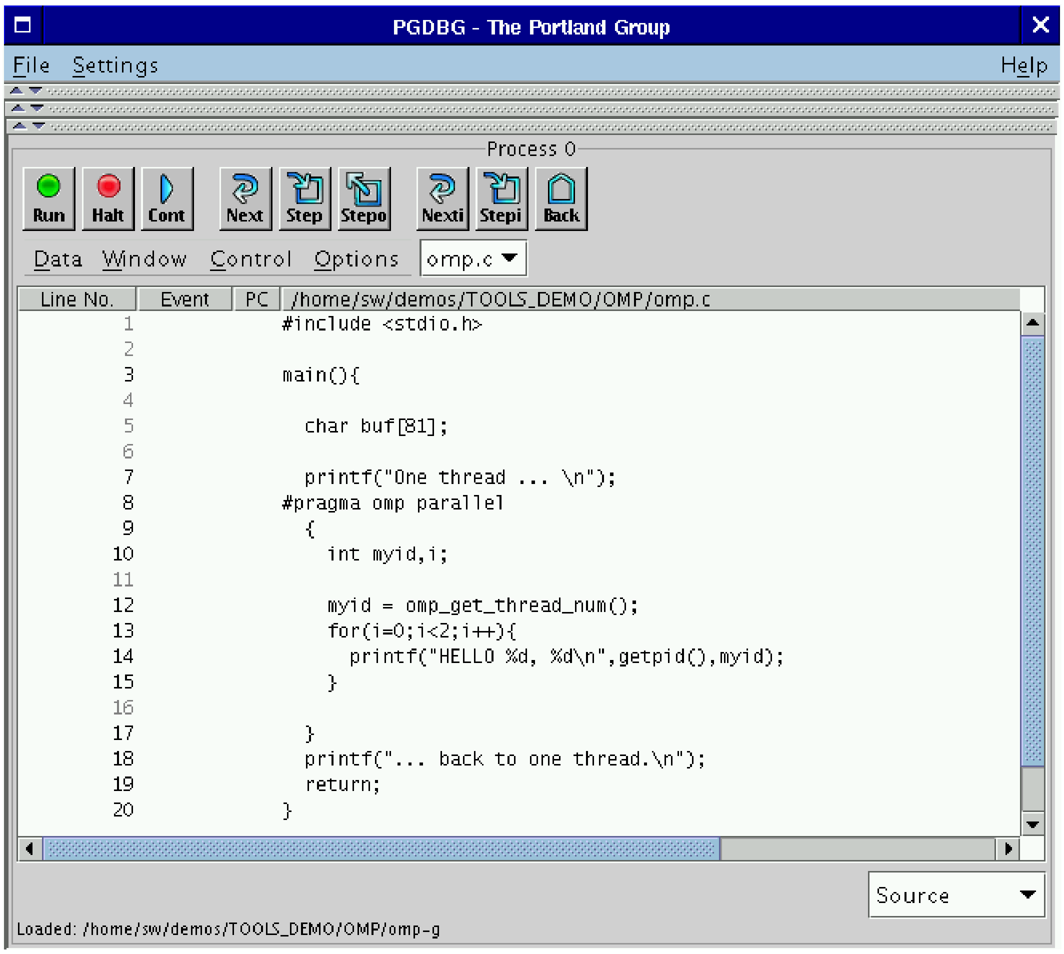
PGDBG Graphical User Interface

The default user interface used by PGDBG is a Graphical User Interface (GUI). There may be minor variations in the appearance of the PGDBG GUI from host to host, depending on the type of display hardware available, the settings for various defaults and the window manager used. Except for differences caused by those factors, the basic interface remains the same across all systems.

Main Window

Figure 1-1 , "Default Appearance of PGDBG GUI", shows the main window of PGDBG GUI when it is invoked for the first time. This window appears when PGDBG starts and remains throughout the debug session. The initial size of the main window is approximately 700 x 600. It can be resized according to the conventions of the window manager. Changes in window size and other settings are saved and used in subsequent invocations of PGDBG. To prevent this, uncheck the Save Settings on Exit item under the Settings menu. See "Main Window Menus" on page 13, for information on the Settings menu.

Figure 1-1: Default Appearance of PGDBG GUI



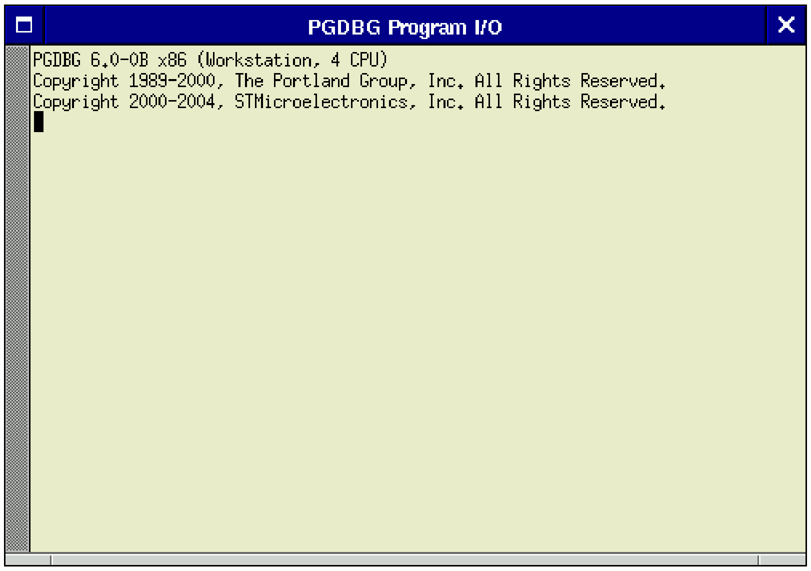
```
#include <stdio.h>

main(){

  char buf[81];

  printf("One thread ... \n");
#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();
    for(i=0;i<2;i++){
      printf("HELLO %d, %d\n",getpid(),myid);
    }

  }
  printf("... back to one thread.\n");
  return;
}
```

Loaded: /home/sw/demos/TOOLS_DEMO/OMP/omp-g

There are three horizontal divider bars (controlled by small up and down arrow icons) at the top of the GUI in Figure 1-1. These dividers hide the following optional control panels: Command Prompt, Focus Panel, and the Process/Thread Grid. Figure 1-3 , "PGDBG GUI with All Control Panels Visible", shows the main window with these controls visible. The GUI will remember which control panels are visible when you exit and will redisplay them when you reopen PGDBG. Below the dividers is the Source Panel described in "Source Panel" on page 11.

A second window named the Program I/O window is displayed when PGDBG is started. Any input or output performed by the target program is entered and/or displayed in this window.

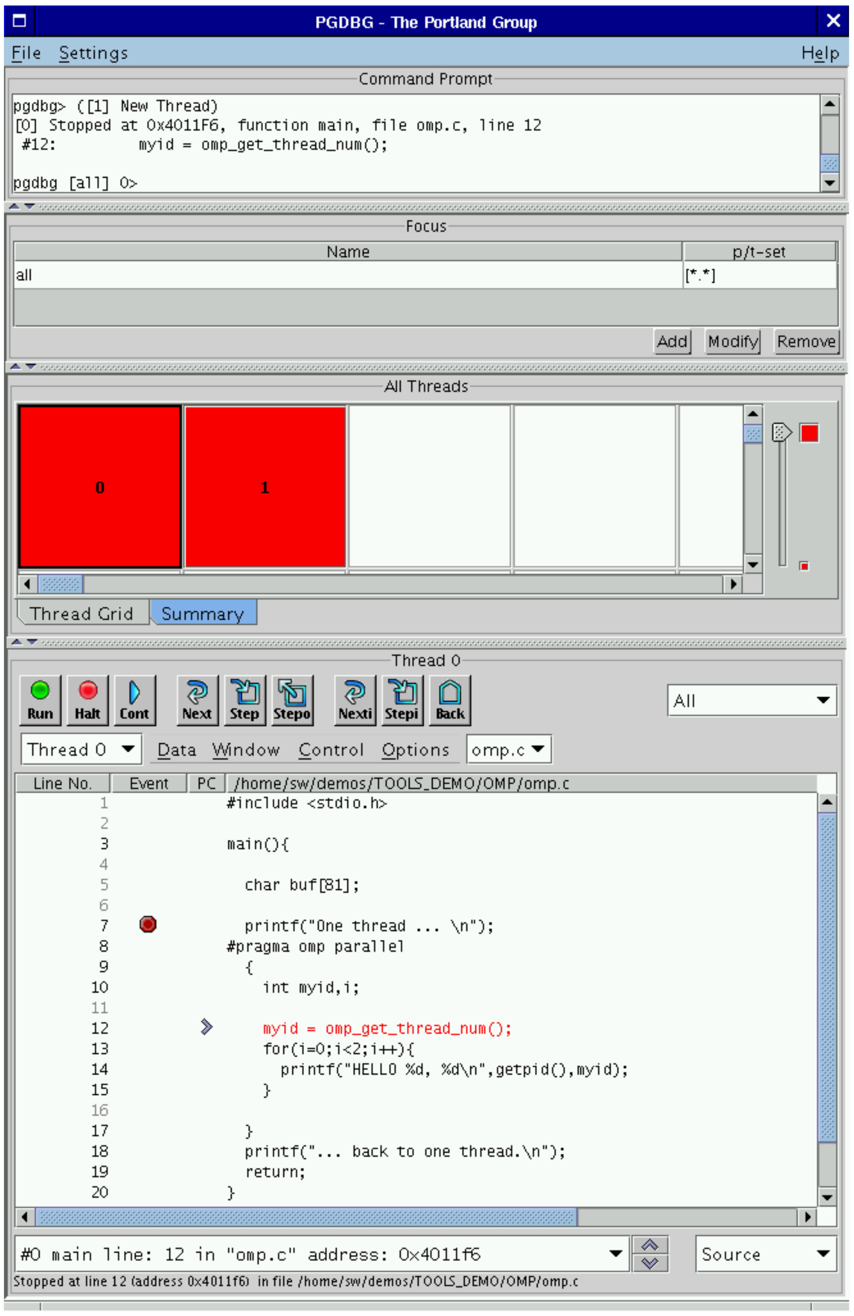
Figure 1-2: PGDBG Program I/O Window



```
PGDBG 6.0-0B x86 (Workstation, 4 CPU)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2004, STMicroelectronics, Inc. All Rights Reserved.
```

On Windows platforms this window is instantiated behind the PGDBG main window in order to maintain input focus in the main window.

Figure 1-3: PGDBG GUI with All Control Panels Visible


PGDBG - The Portland Group
File Settings Help
Command Prompt
Focus
Name p/t-set
all [*.*]
Add Modify Remove
All Threads
0
1
Thread Grid Summary
Thread 0
Run Halt Cont Next Step Stepo Nexti Stepi Back
All
Thread 0 Data Window Control Options omp.c
Line No. Event PC /home/sw/demos/TOOLS_DEMO/OMP/omp.c
Source

```
pgdbg> ([1] New Thread)
[0] Stopped at 0x4011F6, function main, file omp.c, line 12
 #12:         myid = omp_get_thread_num();

pgdbg [all] 0>
```

```
#include <stdio.h>

main(){

  char buf[81];

  printf("One thread ... \n");
#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();
    for(i=0;i<2;i++){
      printf("HELLO %d, %d\n",getpid(),myid);
    }

  }
  printf("... back to one thread.\n");
  return;
}
```

```
#0 main line: 12 in "omp.c" address: 0x4011f6
Stopped at line 12 (address 0x4011f6) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c
```

The components of the main window (from top to bottom as seen in Figure 1-3) are:

- Command Prompt Panel
- Focus Panel
- Process/Thread Grid
- Source Panel

Command Prompt Panel

The Command Prompt Panel provides an interface in which to use the PGDBG command language. Commands entered in this window are executed, and the results are displayed. See "Commands Summary" on page 40, for a list of commands that can be entered in the command prompt panel. The GUI also supports a "free floating" version of this window. To use the "free floating" command prompt window, select the Command Window check box under the Window menu ("Source Panel Menus" on page 15). Users who use only GUI controls may leave this panel hidden.

Focus Panel

The Focus Panel can be used in a parallel debugging session to specify subsets of processes and/or threads known as p/t-sets. P/t-sets allow application of debugger commands to a subset of threads and/or processes. P/t-sets are displayed in the table labeled Focus (Figure 1-3). In Figure 1-3, the Focus table contains one p/t-set called All that represents all processes/threads. P/t-sets are covered in more detail in "p/t-set Notation" on page 110. Within the PGDBG GUI, select a p/t set using a left mouse click on the desired group in the Focus table. The selected group is known as the Current Focus. By default, the Current Focus is set to all processes/threads. Note that this panel has no real use in serial debugging (debugging one single-threaded process).

On Windows platforms, p/t-sets are used only for distinguishing threads.

Process/Thread Grid

The Process/Thread Grid is another component of the interface used for parallel debugging. All active target processes and threads are listed in the Process/Thread Grid. If the target application consists of multiple processes, the grid is labeled Process Grid. If the target application is a single multi-threaded process, the grid is labeled Thread Grid. The colors of each element in the grid represent the state of the corresponding component of the target application; for example, green means running and red means stopped. The colors and their meanings are defined in Table 1-1.

On Windows platforms, the Process/Thread Grid is used only for distinguishing threads.

Table 1-1: Thread State Is Described Using Color

Option	Description
Stopped	Red
Signaled	Blue
Running	Green
Exited	Black
Killed	Black

In the Process/Thread Grid, each element is labeled with a numeric process identifier (see "Process-only Debugging" on page 109) and represents a single process. Each element is a button that can be pushed to select the corresponding process as the Current Process. The Current Process is highlighted with a thick black border.

For single-process/multi-threaded (e.g., OpenMP) targets, the grid is called the Thread Grid. Each element in the thread grid is labeled with a numeric thread identifier (see "Threads-only Debugging" on page 108). As with the process grid, clicking on an element in the thread grid selects that element as the Current Thread, which is highlighted with a thick black border.

For multi-process/multi-threaded (hybrid) targets, the grid is labeled the Process Grid. Selecting a process in the grid will reveal an inner thread grid as shown in Figure 1-4 , "Process Grid with Inner Thread Grid". In Figure 1-4, process 0 has four threads labeled 0.0, 0.1, 0.2, and 0.3; where the integer to the left of the decimal point is the process identifier and the integer to the right of the decimal point is the thread identifier. See "Multilevel Debugging" on page 109 for more information on processes/thread identifiers.

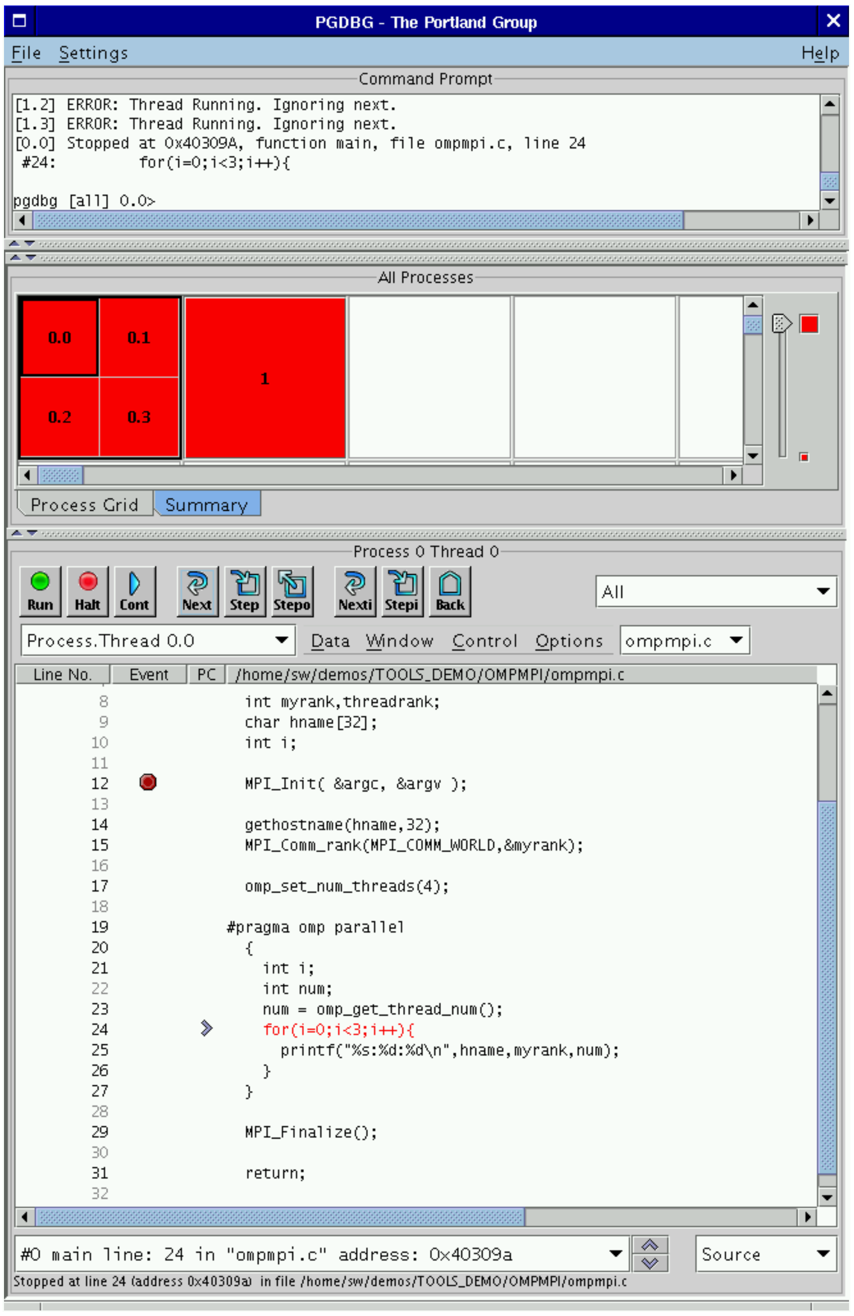
For a text representation of the Process/Thread grid, select the Summary tab under the grid. The text representation is essentially the output of the threads debugger command (see "Process Control" on page 51). When debugging a multi-process or multi-threaded application, the Summary panel will also include a Context Selector (as described in "Source Panel Pop-Up Menus" on page 22). Use the Context Selector to view a summary on a subset of processes/threads. By default, a summary of all the processes/threads displays.

Use the slider to the right of the grid to zoom in and out of the grid. Currently, the grid supports up to 1024 elements. If the slider is not visible, increase the size of the Process/Thread grid's panel.

Source Panel

The Source Panel displays the source code for the current location. The current location is marked by an arrow icon under the PC column. Source line numbers are listed under the Line No. column. Figure 1-4 shows some of the line numbers grayed-out. A grayed-out line number indicates that its respective source line is non-executable. Some examples of non-executable source lines are comments, non-applicable preprocessed code, some routine prologs, and some variable declarations. A line number in a black font represent an executable source line. Breakpoints may be set at any executable source line by clicking the left mouse button under the Event column of the source line. The breakpoints are marked by stop sign icons. An existing breakpoint may be deleted by clicking the left mouse button on the stop sign icon. The source panel is described in greater detail in ."Source Panel" on page 15

Figure 1-4: Process Grid with Inner Thread Grid



```
[1.2] ERROR: Thread Running. Ignoring next.
[1.3] ERROR: Thread Running. Ignoring next.
[0.0] Stopped at 0x40309A, function main, file ompmpi.c, line 24
 #24:         for(i=0;i<3;i++){

pgdbg [all] 0.0>
```

/home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c

```
    int myrank,threadrank;
    char hname[32];
    int i;

    MPI_Init( &argc, &argv );

    gethostname(hname,32);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    omp_set_num_threads(4);

  #pragma omp parallel
    {
      int i;
      int num;
      num = omp_get_thread_num();
      for(i=0;i<3;i++){
        printf("%s:%d:%d\n",hname,myrank,num);
      }
    }

    MPI_Finalize();

    return;
```

#0 main line: 24 in "ompmpi.c" address: 0x40309a

Stopped at line 24 (address 0x40309a) in file /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c

Main Window Menus

The main window includes three menus located at the top of the window: File, Settings, and Help. Below is a summary of each menu in the main window.

- File Menu
  - *Open Target…* – Select this option to begin a new debugging session. After selecting this option, select the program to debug (the target) from the file chooser dialog. The current target is closed and replaced with the target that you selected from the file chooser. Press the Cancel button in the file chooser to abort the operation. See the debug command in "Process Control" on page 51 for more information.
  - *Attach to Target…* – [Linux Only] Select this option to attach to a running process. You can attach to a target running on a local or a remote host. See also the attach command in "Process Control" on page 51.
  - *Detach Target* – [Linux Only] Select this option to end the current debug session. This command does not terminate the target application. See the detach command in "Process Control" on page 51 for more information.
  - *Exit* – End the current debug session and close all the windows.
- Settings Menu
  - *Font…* – This option displays the font chooser dialog box. Use this dialog box to select the font and size used in the Command Prompt Panel, Focus Panel, and Source Panel. The default font is named monospace and the default size is 12.
  - *Show Tool Tips* – Select this check box to enable tool tips. Tool tips are small temporary messages that pop-up when you position the mouse pointer over a component in the GUI. They provide additional information on what a particular component does. Unselect this check box to turn them off.
  - *Restore Factory Settings* – Select this option to restore the GUI to its initial state as shown in Figure 1-1.
  - *Restore Saved Settings* – Select this option to restore the GUI to the state that it was in at the start of the debug session.

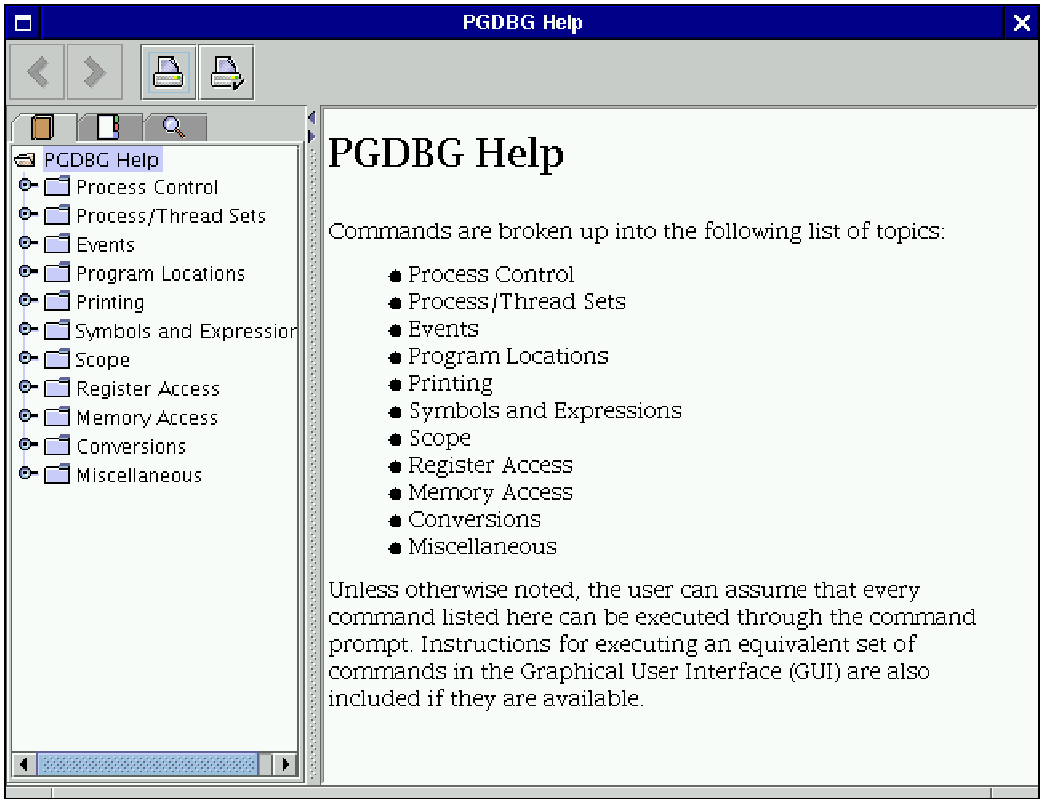
  - *Save Settings on Exit* – By default, the PGDBG will save the state (size and settings) of the GUI when you exit. Uncheck this option to prevent PGDBG from saving the GUI state. This option must be unchecked prior to every exit since PGDBG will always default to saving GUI state. When PGDBG saves state, it stores the size of the main window, the location of the main window on the desktop, the location of each control panel divider, the tool tips preference, the font and size used. The GUI state is not shared across host machines.

- Help Menu

  - *PGDBG Help*… – This option starts up PGDBG's integrated help utility as shown in Figure 1-5. The help utility includes a summary of every PGDBG command. To find a command, use one of the following tabs in the left panel: The "book" tab presents a table of contents, the "index" tab presents an index of commands, and the "magnifying glass" tab presents a search engine. Each help page (displayed on the right) may contain hyperlinks (denoted in underlined blue) to terms referenced elsewhere in the help engine. Use the arrow buttons to navigate between visited pages. Use the printer buttons to print the current help page.

  - *About PGDBG*… – This option displays a dialog box with version and copyright information on PGDBG. It also contains sales and support points of contact.

Figure 1-5: PGDBG Help Utility



PGDBG Help

Commands are broken up into the following list of topics:

- Process Control
- Process/Thread Sets
- Events
- Program Locations
- Printing
- Symbols and Expressions
- Scope
- Register Access
- Memory Access
- Conversions
- Miscellaneous

Unless otherwise noted, the user can assume that every command listed here can be executed through the command prompt. Instructions for executing an equivalent set of commands in the Graphical User Interface (GUI) are also included if they are available.

Source Panel

As described in "Source Panel" on page 11, the source panel is located at the bottom of the GUI; below the Command Prompt, Focus Panel, and Process/Thread Grid. Use the source panel to control the debug session, step through source files, set breakpoints, and browse source code. The source panel descriptions are divided into the following categories: Menus, Buttons, Combo Boxes, Messages, and Events.

Source Panel Menus

The source panel contains the following four menus: Data, Window, Control, and Options. In the descriptions below, keyboard shortcuts will be indicated by keystroke combinations (e.g., Control P) enclosed in parentheses.

Data Menu	The items under this menu are enabled when a data item is selected in the source panel. Selecting and printing data in the source panel is explained in detail in "Source Panel" on page 11. See also "Printing Variables and Expressions" on page 67.

Print	Print the value of the selected item. (Control P).
Print *	Dereference and print the value of the selected item.
String	Treat the selected value as a string and print its value.
Bin	Print the binary value of the selected item.
Oct	Print the octal value of the selected item.
Hex	Print the hex value of the selected item.
Dec	Print the decimal value of the selected item.
Ascii	Print the ASCII value of the selected item.
Addr	Print the address of the selected item.
Type Of	Print data type information for the selected item.

Window Menu	The items under this menu select various subwindows associated with the target application. Subwindows are explained in greater detail in "Source Panel Pop-Up Menus" on page 22.

Registers	Display the registers subwindow. See also the regs command in "Register Access" on page 76.
Stack	Display the stack subwindow. See also the Stack command in "Program Locations" on page 65.
Locals	Display a list of local variables that are currently in scope. See also the names command in "Scope" on page 74.

Custom	Bring up a custom subwindow.
Disassembler	Bring up the PGDBG Disassembler subwindow.
Memory	Bring up the memory dumper subwindow.
Messages	[Linux Only] Display the MPI message queues. See "MPI Message Queues" on page 137 for more information on MPI message queues.
Events	Display a list of currently active breakpoints, watchpoints, etc.
Command Window	When this menu item's check box is selected, the GUI will display a "free floating" version of the command prompt window. See "Commands Summary" on page 40 for a description of each command that can be entered in the command prompt.

Control Menu

The items under this menu control the execution of the target application. Many of the items under this menu have a corresponding button associated with them (see "Source Panel Buttons" on page 20).

Arrive	Return the source pane to the current PC location. See the arrive command in "Program Locations" on page 65 (Control A).
Up	Enter scope of routine up one level in the call stack. See the up command in "Scope" on page 74 (Control U).
Down	Enter scope of routine down one level in the call stack. See the down command in "Scope" on page 74 (Control D).
Run	Run or Rerun the target application. See the run and rerun commands in "Process Control" on page 51 (Control R).

Run Arguments	Opens a dialog box that allows adding to or modifying the target's runtime arguments.
Halt	Halt the running processes or threads. See the halt command in "Process Control" on page 51 (Control H).
Call…	Open a dialog box to request a routine to call. See "Symbols and Expressions" on page 71 for more information on the call command.
Cont	Continue execution from the current location. See the cont command in "Process Control" on page 51 (Control G).
Step	Continue and stop after executing one source line, stepping into called routines. See the step command in "Process Control" on page 51 (Control S).
Next	Continue and stop after executing one source line, stepping over called routines. See the next command in "Process Control" on page 51 (Control N).
Step Out	Continue and stop after returning to the caller of the current routine. See the stepout command in "Process Control" on page 51 (Control O).
Stepi	Continue and stop after executing one machine instruction, stepping into called routines. See the stepi command in "Process Control" on page 51 (Control I).
Nexti	Continue and stop after executing one machine instruction, stepping over called routines. See the nexti command in "Process Control" on page 51 (Control T).

Options Menu

This menu contains additional items that assist in the debug process.

Search Forward…	Select this option to perform a forward keyword search in the source panel (Control F).
Search Backward…	Select this option to perform a backward keyword search in the source panel (Control B).
Search Again	Select this option to repeat the last keyword search that was performed on the source panel (Control E).
Locate Routine…	When this option is selected, PGDBG will query for the name of the routine that you wish to find. If PGDBG has symbol and source information for that routine, it will display the routine in the source panel. See also "Source Panel Pop-Up Menus" on page 22.
Set Breakpoint…	When this option is selected, PGDBG will query for the name of a routine for setting a breakpoint. The GUI will then set a breakpoint at the first executable source line in the specified routine.
Disassemble	Disassemble the data selected in the source panel. See also "Source Panel Pop-Up Menus" on page 22.
Cascade Windows	If one or more subwindows are open, this option can be used to automatically stack subwindows in the upper left-hand corner of the desktop (Control W).
Refresh	Repaint the process/thread grid and source panels (Control L).

Source Panel Buttons

There are nine buttons located above the source panel's menus. Below is a summary of each button.

Run	Same as the Run item under the Control menu.
Halt	Same as the Halt item under the Control menu.
Cont	Same as the Cont item under the Control menu.
Next	Same as the Next item under the Control menu.
Step	Same as the Step item under the Control menu.
Stepo	Same as the Step Out item under the Control menu.
Nexti	Same as the Nexti item under the Control menu.
Stepi	Same as the Stepi item under the Control menu.
Back	Reset the source panel view to the current PC location (denoted by the left arrow icon under the PC column).

Source Panel Combo Boxes

Depending on the state of the debug session, the source panel may contain one or more combo boxes. A combo box is a combination text field and list component. In its closed or default state, it presents a text field of information with a small down arrow icon to its right. When the down arrow icon is selected by a left mouse click, the box opens and presents a list of choices that can be selected.

The source panel, as shown in Figure 1-3, contains five combo boxes labeled All, Thread 0, omp.c, #0 main line: 12 in "omp.c" address: 0x4011f6, and Source. These combo boxes are called the Apply Selector, Context Selector, Source File Selector, Scope Selector, and Display Mode Selector respectively. Below is a description of each combo box.

- Use the Apply Selector to select the set of processes and/or threads on which to operate. Any command entered in the source panel will be applied to this set of processes/threads. These commands include setting breakpoints, selecting items under the Control menu, pressing one of the nine buttons mentioned in "Source Panel Buttons" on page 20, and so on. Depending on whether you are debugging a multi-threaded, multi-process, or multi-process/multi-threaded (hybrid) target, the following options are available:

All	All processes/threads receive commands entered in the source panel (default).
Current Thread	Commands are applied to the current thread ID only.
Current Process	Commands are applied to all threads that are associated with the current process.
Current Process.Thread	Commands are applied to the current thread on the current process only.
Focus	Commands are applied to the focus group selected in the Focus Panel (described in "Main Window" on page 5). Refer to "Process/ Thread Sets" on page 110for more information on this advanced feature.

This combo box is not displayed when debugging a serial program.

- The function of the Context Selector is the same as for the Process/Thread Grid; it is used to change the current Process, Thread, or Process.Thread ID currently being debugged. This combo box is not displayed when debugging a serial program.
- By default, the Source File Selector displays the source file that contains the current target location. It can be used to select another file for viewing in the Source Panel. When this combo box is closed, it displays the name of the source file displayed in the Source Panel. To select a different source file, open the combo box and select a file from the list. If the source file is available, the source file will appear in the Source Panel.
- The Scope Selector displays the scope of the current Program Counter (PC). Open the combo box and select a different scope from the list or use the up and down buttons located on the right of the combo box. The up button is equivalent to the up debugger command and the down button is equivalent to the down debugger command. See "Scope" on page 74 for more information on the up and down commands.
- The Display Mode Selector is used to select three different source display modes: Source, Disassembly, and Mixed. The Source mode shows the source code of the current source file indicated by the File Selector. This is the default display mode if the source file is available. The Disassembly mode shows the machine instructions of the current routine. This is the default display mode if the source file is not available. The Mixed mode shows machine instructions annotated with source code. This mode is available only if the source file is available.

Source Panel Messages

The source panel contains two message areas. The top center indicates the current process/thread ID (e.g., Thread 0 in Figure 1-7) and the bottom left displays status messages (e.g., Stopped at line 12… in Figure 1-7).

Source Panel Events

Breakpoints are displayed under the Event column in the source panel. The stop sign icon denotes a breakpoint. Breakpoints are added through the source panel by clicking the left mouse button on the desired source line under the Event column. Clicking the left mouse button over a stop sign will delete the corresponding breakpoint. Selecting the Events item under the Window menu will display a global list of Events (e.g., breakpoints, watchpoints, etc.).

Source Panel Pop-Up Menus

The PGDBG source panel supports two pop-up menus to provide quick access to commonly used features. One pop-up menu is used to invoke subwindows. It is accessed using a right mouse-click in a blank or vacant area of the source panel. See “Subwindows” on page 24 for more information on invoking subwindows using a pop-up menu.

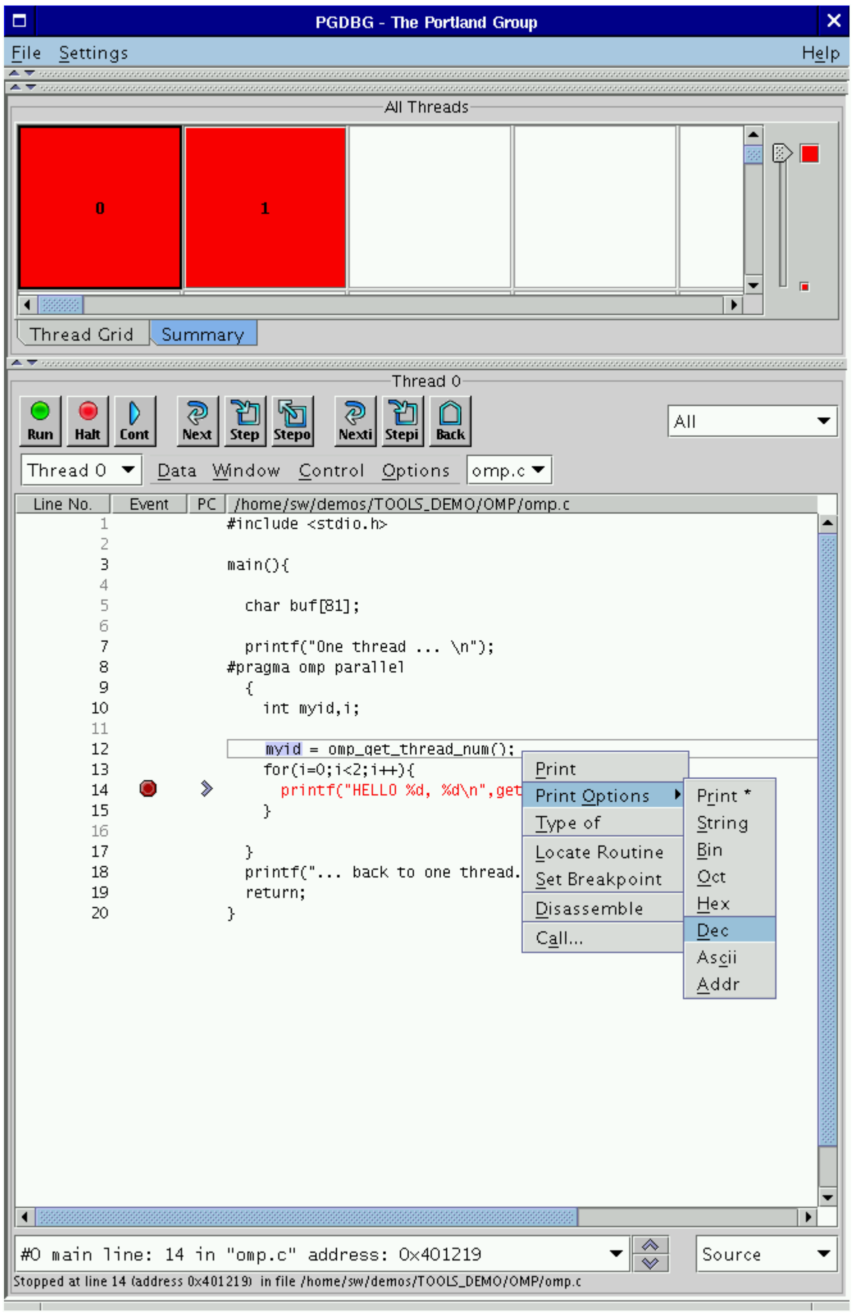
The other pop-up menu is accessed by first highlighting some text in the source panel, then using a right mouse click to bring up the menu. The selections offered by this pop-up menu take the selected text as input.

To select text in the source panel, first click on the line of source containing the text. This will result in the display of a box surrounding the source line. Next, hold down the left mouse button and drag the cursor, or mouse pointer, across the text to be selected. The text should then be highlighted.

Once the text is highlighted, menu selections from the Source Panel menus or from the Source Panel pop-up menu will use the highlighted text as input. In Figure 1-6, the variable myid has been highlighted and the pop-up menu is being used to print its value as a decimal integer. The data type of selected data items may also be displayed using the pop-up menu.

The pop-up menu provides the Disassemble, Call, and Locate selections, which use selected routine names as input. The Disassemble item opens a disassembler subwindow for the selected routine. The Call item can be used to manually call the selected routine. The Locate option displays the source code in which the selected routine is defined. Please see the description for each of these items in “Source Panel Menus” on page 15 for more information.

Figure 1-6: Data Pop-up Menu



```
#include <stdio.h>

main(){

  char buf[81];

  printf("One thread ... \n");
#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();
    for(i=0;i<2;i++){
      printf("HELLO %d, %d\n",get
    }

  }
  printf("... back to one thread.
  return;
}
```

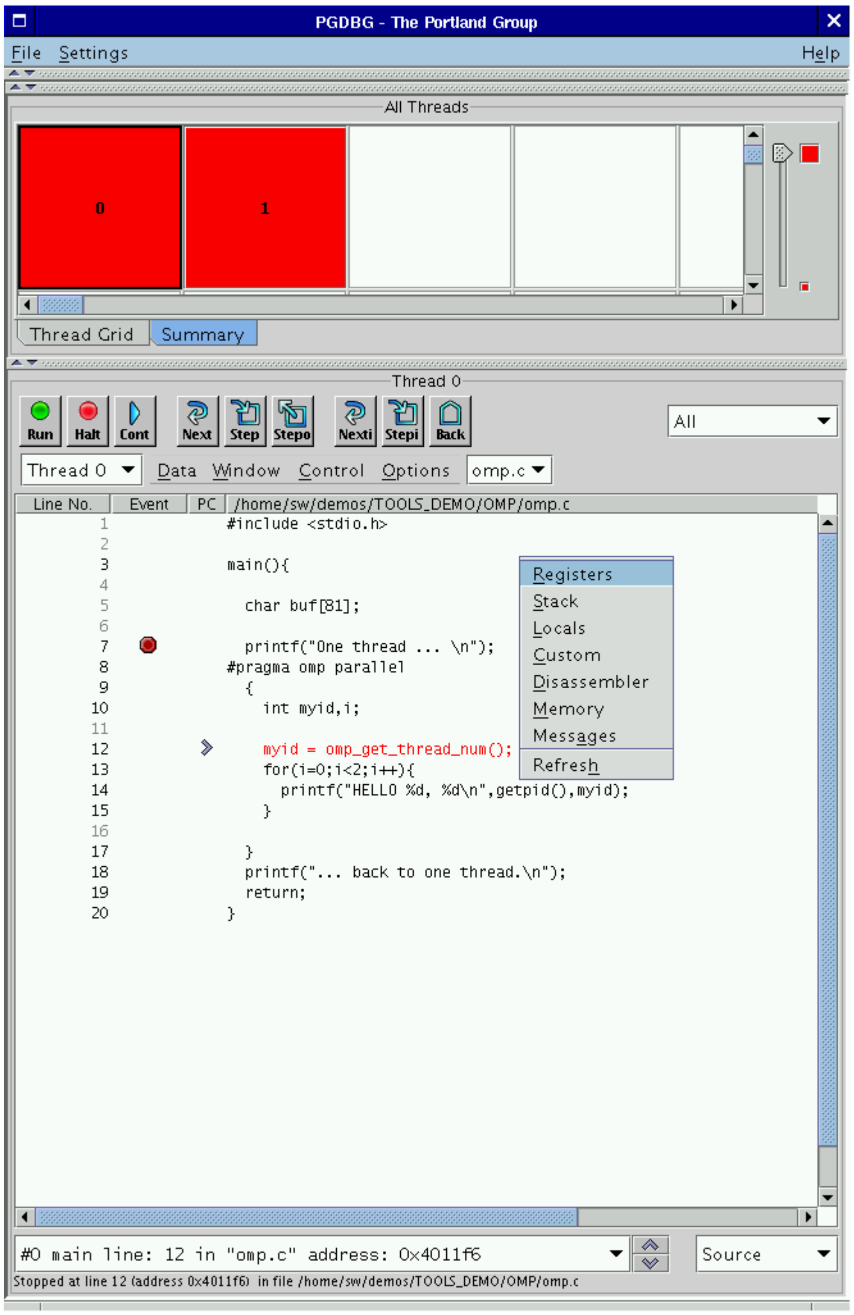
```
#0 main line: 14 in "omp.c" address: 0x401219
Stopped at line 14 (address 0x401219) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c
```

Subwindows

A subwindow is defined as any PGDBG GUI component that is not embedded in the main window described in "Main Window" on page 5. One example of a subwindow is the Program I/O window introduced in Figure 1-2. Other examples of subwindows can be found under the source panel's Window menu. These include the Registers, Stack, Locals, Custom, Disassembler, Memory, Messages, Events, and Command Window subwindows. With the exception of the Command Window, all of these subwindows are controlled by similar mechanisms. The standard subwindow control mechanisms are described in "Standard Subwindow Controls" on page 26. Specific details of other subwindows are described in subsequent sections. See the description of the Window menu, "Source Panel Menus" on page 15 for more information on each subwindow.

The Window menu can be used to to bring up a subwindow. An alternative mechanism is to click the right mouse button over a blank spot in the source panel to invoke a pop-up menu (Figure 1-7), which can be used to select a subwindow. The subwindow that gets displayed is specific to the current process and/or thread. For example, in Figure 1-7, selecting Registers will display the registers for thread 0, which is the current thread.

Figure 1-7: Opening a Subwindow with a Pop-up Menu


PGDBG - The Portland Group
File Settings Help
All Threads
0
1
Thread Grid Summary
Thread 0
Run Halt Cont Next Step Stepo Nexti Stepi Back
All
Thread 0 Data Window Control Options omp.c
Line No. Event PC /home/sw/demos/TOOLS_DEMO/OMP/omp.c
Registers
Stack
Locals
Custom
Disassembler
Memory
Messages
Refresh
Source

```
#include <stdio.h>

main(){

  char buf[81];

  printf("One thread ... \n");
#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();
    for(i=0;i<2;i++){
      printf("HELLO %d, %d\n",getpid(),myid);
    }

  }
  printf("... back to one thread.\n");
  return;
}
```

```
#0 main line: 12 in "omp.c" address: 0x4011f6
Stopped at line 12 (address 0x4011f6) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c
```

Standard Subwindow Controls

The PGDBG graphical user interface supports a number of subwindows for displaying detailed information about the target application state. These subwindows include the memory subwindow, the disassembler subwindow, the registers subwindow, the custom subwindow (used for displaying the output of arbitrary commands), and the messages subwindow (used for displaying MPI state).

Figure 1-8 shows the memory subwindow. This subwindow shows all of the possible controls that are available in a PGDBG subwindow. Not all subwindows will have all of the components shown in this figure. However, nearly all will have the following components: File menu, Options menu, Reset button, Close Button, Update button, and the Lock/Unlock toggle button.

The File menu contains the following items:

Save…	Save the text in this subwindow to a file.
Close	Close the subwindow.

The Options menu contains the following items:

Update	Clear and regenerate the data displayed in the subwindow.
Stop	Interrupt processing. This option comes in handy during long listings that can occur in the Disassembler and Memory subwindows. Control C is a hot key mapped to this menu item.
Reset	Clear the subwindow.

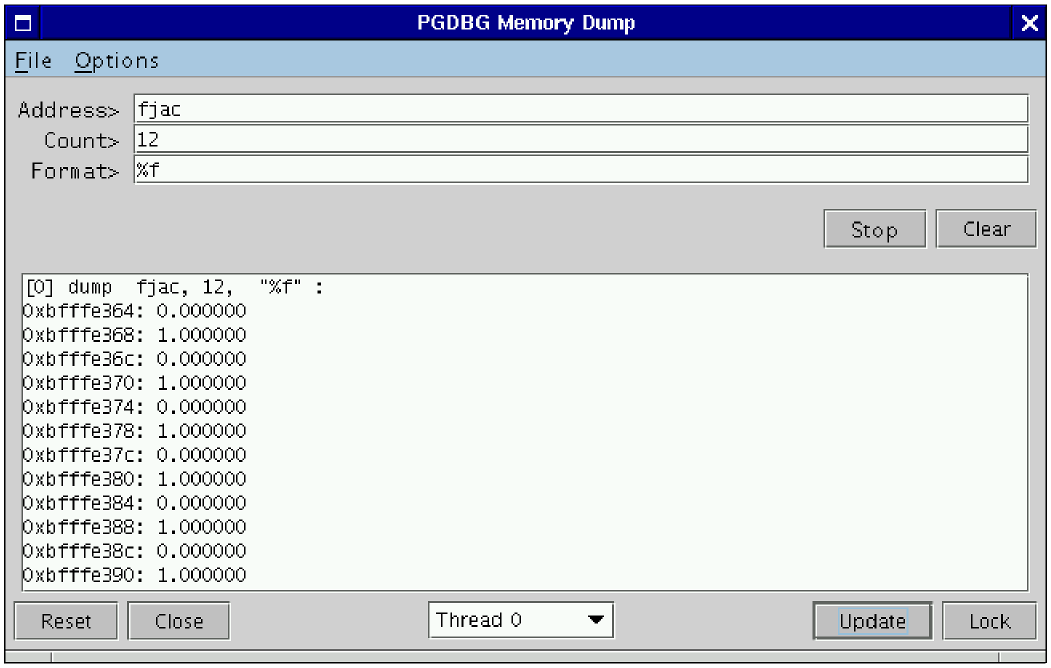
The Reset, Close, and Update buttons are synonymous with their menu item counterparts mentioned above.

The Lock/Unlock button, located in the lower right hand corner of a subwindow, toggles between a lock and an unlock state. Figure 1-8 shows this button in an unlocked state with the button labeled Lock. Figure 1-9 shows this button in a locked state, with the button labeled Unlock. When the Lock/Unlock button is in its unlocked state, subwindows will update themselves whenever a process or thread halts. This can occur after a step, next, or cont command. To preserve the contents of a subwindow, click the left mouse button on the Lock button to lock the display in the subwindow. Figure 1-9 shows an example of a locked subwindow. Note that some of the controls in Figure 1-9 are disabled (grayed-out). After locking a subwindow, PGDBG will disable any controls that affect the display until the subwindow is unlocked. To unlock the subwindow, click the Unlock button. The toggle button will change to Lock and PGDBG will re-enable the other controls.

Besides the subwindow capabilities described above, subwindows may also have one to three input fields. If the subwindow has one or more input fields, then they also contain Stop and Clear buttons. The Stop button is synonymous with the Stop item in the Options menu described above. The Clear button erases the input field(s).

For target applications with more than one process and/or thread, a Context Selector displays in the bottom center as shown in Figure 1-8. The Context Selector can be used to view data specific to a particular process/thread or a subset of process/threads when selecting Focus. Refer to "Process/Thread Sets" on page 110 for more information on Focus.

Figure 1-8: Memory Subwindow



Address> fjac
Count> 12
Format> %f

```
[0] dump  fjac, 12,  "%f" :
0xbfffe364: 0.000000
0xbfffe368: 1.000000
0xbfffe36c: 0.000000
0xbfffe370: 1.000000
0xbfffe374: 0.000000
0xbfffe378: 1.000000
0xbfffe37c: 0.000000
0xbfffe380: 1.000000
0xbfffe384: 0.000000
0xbfffe388: 1.000000
0xbfffe38c: 0.000000
0xbfffe390: 1.000000
```

Memory Subwindow

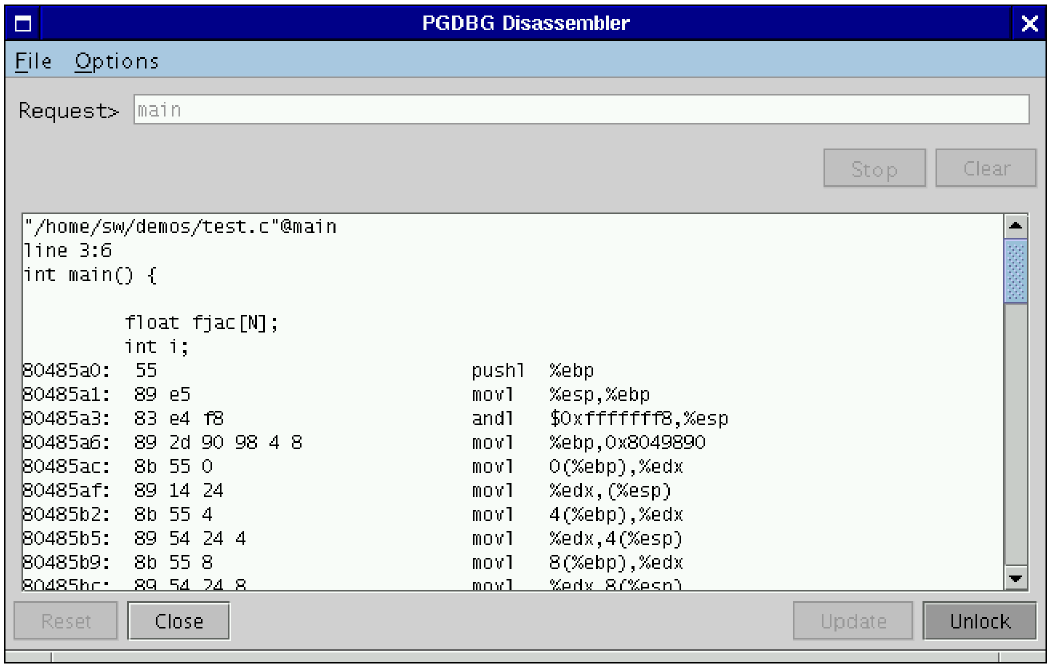
The memory subwindow displays a region of memory using a printf-like format descriptor. In the Memory subwindow, inputs include the starting address in the Address field, the number of items in the Count field, and a printf-like format string in the Format field. See the explanation of the PGDBG dump command ("Memory Access" on page 77) for a description of supported format strings. The Address field will accept a numeric address or a symbolic variable name.

Disassembler Subwindow

Figure 1-9 shows the Disassembler subwindow. Use this subwindow to disassemble a routine (or a text address) specified in the Request> input field. PGDBG will default to the current routine if you specify nothing in the Request> input field. After a request is made to the Disassembler, the GUI will ask if you want to "Display Disassembly in the Source window". Choosing "yes" causes the Disassembler window to disappear and the disassembly to appear in the source panel. Viewing the disassembly in the source panel allows setting breakpoints at the machine instruction level. Choosing "no" will dump the disassembly in the Disassembler subwindow as shown in Figure 1-9.

Specifying a text address (rather than a routine name) in the Request> field will cause PGDBG to disassemble address locations until it runs out of memory or hits an invalid op code. This may cause very large machine language listings. For that case, the subwindow provides a Stop button. Press the Stop button to interrupt long listings that may occur with the Disassembler. Specify a count after the text address to limit the number of instructions dumped to the subwindow. For example, entering 0xabcdef, 16 tells PGDBG to dump up to 16 instructions following address 0xabcdef. The Request> field accepts the same arguments as the disasm command described in "Program Locations" on page 65.

Figure 1-9: Disassembler Subwindow

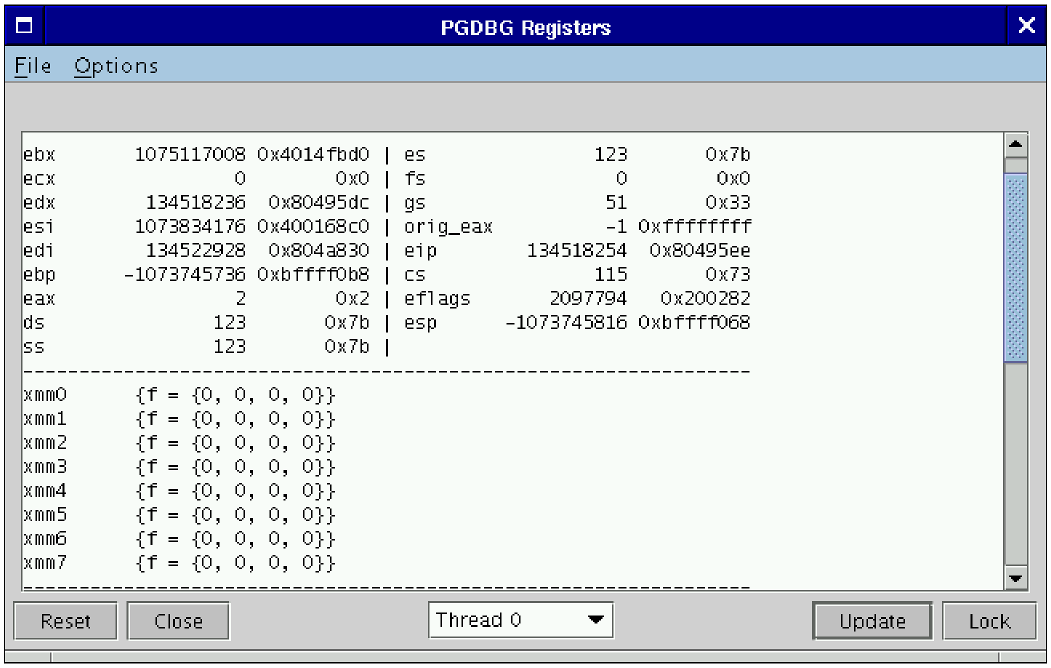


```
"/home/sw/demos/test.c"@main
line 3:6
int main() {

        float fjac[N];
        int i;
80485a0:  55                          pushl  %ebp
80485a1:  89 e5                       movl   %esp,%ebp
80485a3:  83 e4 f8                    andl   $0xfffffff8,%esp
80485a6:  89 2d 90 98 4 8             movl   %ebp,0x8049890
80485ac:  8b 55 0                     movl   0(%ebp),%edx
80485af:  89 14 24                    movl   %edx,(%esp)
80485b2:  8b 55 4                     movl   4(%ebp),%edx
80485b5:  89 54 24 4                  movl   %edx,4(%esp)
80485b9:  8b 55 8                     movl   8(%ebp),%edx
```

Registers Subwindow

Figure 1-10 illustrates the Registers subwindow. As mentioned earlier, view the registers on one or more processes and threads using the Context Selector. The Registers subwindow is essentially a graphical representation of the regs debugger command (see “Register Access” on page 76).

Figure 1-10: Registers Subwindow

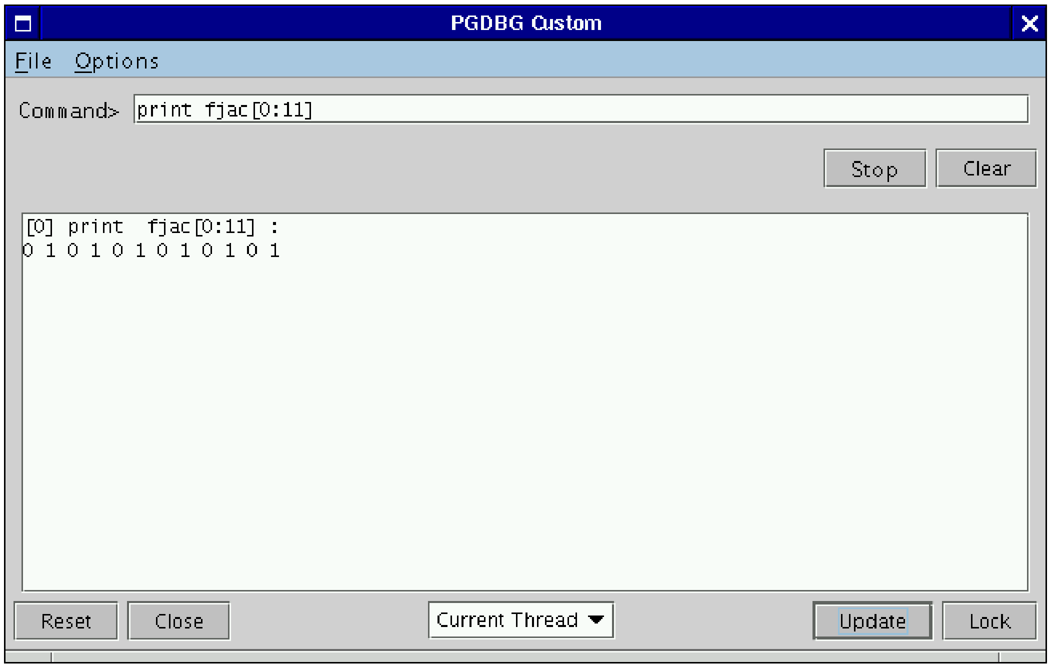


```
ebx       1075117008 0x4014fbd0 | es              123       0x7b
ecx                0        0x0 | fs                0        0x0
edx        134518236  0x80495dc | gs               51       0x33
esi       1073834176 0x400168c0 | orig_eax         -1 0xffffffff
edi        134522928  0x804a830 | eip       134518254  0x80495ee
ebp      -1073745736 0xbffff0b8 | cs              115       0x73
eax                2        0x2 | eflags      2097794   0x200282
ds               123       0x7b | esp     -1073745816 0xbffff068
ss               123       0x7b |
---------------------------------------------------------------
xmm0      {f = {0, 0, 0, 0}}
xmm1      {f = {0, 0, 0, 0}}
xmm2      {f = {0, 0, 0, 0}}
xmm3      {f = {0, 0, 0, 0}}
xmm4      {f = {0, 0, 0, 0}}
xmm5      {f = {0, 0, 0, 0}}
xmm6      {f = {0, 0, 0, 0}}
xmm7      {f = {0, 0, 0, 0}}
---------------------------------------------------------------
```

Custom Subwindow

Figure 1-11 illustrates the Custom subwindow. The Custom subwindow is useful for repeatedly executing a sequence of debugger commands whenever a process/thread halts on a new location or when pressing the Update button. The commands, entered in the edit box labeled "Command>", can be any debugger command mentioned in "Commands Summary" on page 40, including a semicolon-delimited list of commands.

Figure 1-11: Custom Subwindow



```
Command> print fjac[0:11]
```

```
[0] print  fjac[0:11] :
0 1 0 1 0 1 0 1 0 1 0 1
```

Messages Subwindow

The Messages subwindow is used for debugging MPI applications. Refer to “MPI Message Queues” on page 137 for more information on the content and use of this subwindow.

PGDBG Command Language

PGDBG supports a command language that is capable of evaluating complex expressions. The command language can be used by invoking the PGDBG command line interface with the –text option, or in the command prompt panel of the PGDBG graphical user interface. The next three sections of this manual provide information about how to use this command language. See “PGDBG Graphical User Interface” on page 5 for instructions on using the PGDBG GUI.

Commands are entered one line at a time. Lines are delimited by a carriage return. Each line must consist of a command and its arguments, if any. The command language is composed of commands, constants, symbols, locations, expressions, and statements.

Commands are named operations, which take zero or more arguments and perform some action. Commands may also return values that may be used in expressions or as arguments to other commands.

There are two command modes: pgi and dbx. The pgi command mode maintains the original PGDBG command interface. In dbx mode, the debugger uses commands compatible with the familiar dbx debugger. Pgi and dbx commands are available in both command modes, but some command behavior may be slightly different depending on the mode. The mode can be set when PGDBG is invoked by using command line options, or while the debugger is running by using the pgienv command.

Constants

PGDBG supports C language style integer (hex, octal and decimal), floating point, character, and string constants.

Symbols

PGDBG uses the symbolic information contained in the executable object file to create a symbol table for the target program. The symbol table contains symbols to represent source files, subprograms (functions, and subroutines), types (including structure, union, pointer, array, and enumeration types), variables, and arguments. The PGDBG command line interface is case-sensitive with respect to symbol names; a symbol name on the command line must match the name as it appears in the object file.

Scope Rules

Since several symbols in a single application may have the same name, scope rules are used to bind program identifiers to symbols in the symbol table. PGDBG uses the concept of a search scope for looking up identifiers. The search scope represents a routine, a source file, or global scope. When the user enters a name, PGDBG first tries to find the symbol in the search scope. If the symbol is not found, the containing scope, (source file, or global) is searched, and so forth, until either the symbol is located or the global scope is searched and the symbol is not found.

Normally, the search scope will be the same as the current scope, which is the routine where execution is currently stopped. The current scope and the search scope are both set to the current routine each time execution of the target program stops. However, the enter command can be used to change the search scope.

A scope qualifier operator @ allows selection of out-of-scope identifiers. For example, if f is a routine with a local variable i, then:

```
f@i
```

represents the variable i local to f. Identifiers at file scope can be specified using the quoted file name with this operator, for example:

```
"xyz.c"@i
```

represents the variable i defined in file xyz.c.

Register Symbols

In order to provide access to the system registers, PGDBG maintains symbols for them. Register names generally begin with $ to avoid conflicts with program identifiers. Each register symbol has a default type associated with it, and registers are treated like global variables of that type, except that their address may not be taken. See "Register Symbols" on page 89 for a complete list of the register symbols.

Source Code Locations

Some commands must refer to source code locations. Source file names must be enclosed in double quotes. Source lines are indicated by number, and may be qualified by a quoted filename using the scope qualifier operator.

Thus:

```
break 37
```

sets a breakpoint at line 37 of the current source file, and

```
break "xyz.c"@37
```

sets a breakpoint at line 37 of the source file xyz.c.

A range of lines is indicated using the range operator ":". Thus,

```
list 3:13
```

lists lines 3 through 13 of the current file, and

```
list "xyz.c"@3:13
```

lists lines 3 through 13 of the source file xyz.c.

Some commands accept both line numbers and addresses as arguments. In these commands, it is not always obvious whether a numeric constant should be interpreted as a line number or an address. The description for these commands says which interpretation is used. However, PGDBG provides commands to convert from source line to address and vice versa. The line command converts an address to a line, and the addr command converts a line number to an address. For example:

```
{line 37}
```

means "line 37",

```
{addr 0x1000}
```

means "address 0x1000" , and

```
{addr {line 37}}
```

means "the address associated with line 37" , and

```
{line {addr 0x1000}}
```

means "the line associated with address 0x1000".

Lexical Blocks

Line numbers are used to name lexical blocks. The line number of the first instruction contained by a lexical block is used to indicate the start scope of the lexical block.

In the example below, there are two variables named var. One is declared in function main, and the other is declared in the lexical block starting at line 5. The lexical block has the unique name "lex.c"@main@5. The variable var declared in "lex.c"@main@5 has the unique name "lex.c"@main@5@var. The output of the whereis command below shows how these identifiers can be distinguished.

```
lex.c:
1 main()
2 {
3 int var = 0;
4 {
5 int var = 1;
6 printf("var %d\n",var);
7 }
8 printf("var %d\n",var)
9 }


pgdbg> n
Stopped at 0x8048b10, function main, file
/home/demo/pgdbg/ctest/lex.c,
line 6
#6: printf("var %d\n",var);
pgdbg> print var
1
pgdbg> which var
"lex.c"@main@5@var
pgdbg> whereis var
variable: "lex.c"@main@var
variable: "lex.c"@main@5@var
pgdbg> names "lex.c"@main@5
var = 1
```

Statements

Although PGDBG command line input is processed one line at a time, statement constructs allow multiple commands per line, as well as conditional and iterative execution. The statement constructs roughly correspond to the analogous C language constructs. Statements may be of the following forms.

- *Simple Statement:* A command and its arguments. For example:

  ```
  print i
  ```

- *Block Statement:* One or more statements separated by semicolons and enclosed in curly braces. Note: these may only be used as arguments to commands or as part of if or while statements. For example:

  ```
  if(i>1) {print i; step }
  ```

- *If Statement*: The keyword if, followed by a parenthesized expression, followed by a block statement, followed by zero or more else if clauses, and at most one else clause. For example:

  ```
  if(i>j) {print i} else if(i<j) {print
  j} else {print "i==j"}
  ```

- *While Statement:* The keyword while, followed by a parenthesized expression, followed by a block statement. For example:

  ```
  while(i==0) {next}
  ```

Multiple statements may appear on a line separated by a semicolon. For example:

```
break main; break xyz; cont; where
```

sets breakpoints in routines main and xyz, continues, and prints the new current location. Any value returned by the last statement on a line is printed.

Statements can be parallelized across multiple threads of execution. See "Parallel Statements" on page 131 for details.

Events

Breakpoints, watchpoints and other mechanisms used to define the response to certain conditions are collectively called events.

- An event is defined by the conditions under which the event occurs and by the action taken when the event occurs.
- A breakpoint occurs when execution reaches a particular address. The default action for a breakpoint is simply to halt execution and prompt the user for commands.
- A watchpoint occurs when the value of an expression changes.
- A hardware watchpoint occurs when the specified memory location is accessed or modified.

PGDBG supports five basic commands for defining events. Each command takes a required argument and may also take one or more optional arguments. The basic commands are break, watch, hwatch, track and do. The command break takes an argument specifying a breakpoint location. Execution stops when that location is reached. The watch command takes an expression argument. Execution stops and the new value is printed when the value of the expression changes. The hwatch command takes a data address argument (this can be an identifier or variable name). Execution stops when memory at that address is written.

The track command is like watch except that execution continues after the new value is printed. The do command takes a list of commands as an argument. The commands are executed whenever the event occurs.

The five event commands share a common set of optional arguments. The optional arguments provide the ability to make the event definition more specific. They are:

at *line*	Event occurs at indicated line.
at *addr*	Event occurs at indicated address.
in *routine*	Event occurs throughout indicated routine.
if (*condition*)	Event occurs only when condition is true.
do {*commands*}	When event occurs execute commands.

The optional arguments may appear in any order after the required argument and should not be delimited by commas.

For example:

```
watch i at 37 if(y>1)
```

This event definition says to stop and print the value of I whenever line 37 is executed and the value of y is greater than 1.

```
do {print xyz} in f
```

This event definition says that at each line in the routine f print the value of xyz.

```
break func1 if (i==37) do {print
a[37]; stack}
```

This event definition says to print the value of a[37] and do a stack trace when i is equal to 37 in routine func1.

Event commands that do not explicitly define a location will occur at each source line in the program. For example:

```
do {where}
```

prints the current location at the start of each source line, and

```
track a.b
```

prints the value of a.b at the start of each source line if the value has changed.

Events that occur at every line can be useful, but they can make program execution very slow. Restricting an event to a particular address minimizes the impact on program execution speed, and restricting an event that occurs at every line to a single routine causes execution to be slowed only when that routine is executed.

PGDBG supports instruction level versions of several commands (for example breaki, watchi, tracki, and doi). The basic difference in the instruction version is that these commands will interpret integers as addresses rather than line numbers, and events will occur at each instruction rather than at each line.

When multiple events occur at the same location, all event actions will be taken before the prompt for input. Defining event actions that resume execution is allowed but discouraged, since continuing execution may prevent or defer other event actions. For example:

```
break 37 do {continue}
```

```
break 37 do {print i}
```

This creates an ambiguous situation. It's not clear whether i should ever be printed.

Events only occur after the continue and run commands. They are ignored by step, next, call, and other commands.

Identifiers and line numbers in events are bound to the current scope when the event is defined.

For example:

```
break 37
```

sets a breakpoint at line 37 in the current file.

```
track i
```

will track the value of whatever variable i is currently in scope. If i is a local variable then it is wise to add a location modifier (at or in) to restrict the event to a scope where i is defined.

Scope qualifiers can also specify lines or variables that are not currently in scope. Events can be parallelized across multiple threads of execution. See "Parallel Events" on page 129 for details.

Expressions

The debugger supports evaluation of expressions composed of constants, identifiers, commands that return values, and operators. Table 1-2 , "PGDBG Operators" shows the C language operators that are supported. The operator precedence is the same as in the C language.

To use a value returned by a command in an expression, the command and arguments must be enclosed in curly braces. For example:

```
breaki {pc}+8
```

invokes the pc command to compute the current address, adds 8 to it, and sets a breakpoint at that address. Similarly, the following command compares the start address of the current routine with the start address of routine xyz. It prints the value 1 if they are equal and 0 if they are not.

```
print {addr {func}}=={addr
xyz}
```

The @ operator, introduced previously, may be used as a scope qualifier. Its precedence is the same as the C language field selection operators ".", and "->" .

PGDBG recognizes a range operator ":" which indicates array sub-ranges or source line ranges. For example,

```
print a[1:10]
```

prints elements 1 through 10 of the array a, and

```
list 5:10
```

lists source lines 5 through 10, and

```
list "xyz.c"@5:10
```

lists lines 5 through 10 in file xyz.c. The precedence of ':' is between '||' and '='.

The general format for the range operator is [lo : hi : step] where:

lo	is the array or range lower bound for this expression.
hi	is the array or range upper bound for this expression.
step	is the step size between elements.

An expression can be evaluated across many threads of execution by using a prefix p/t-set. See "Current vs. Prefix p/t-set" on page 113 for details.

Table 1-2: PGDBG Operators

Operator	Description	Operator	Description
*	indirection	<=	less than or equal
.	direct field selection	>=	greater than or equal
->	indirect field selection	!=	not equal
[ ]	``C" array index	&&	logical and
()	routine call	||	logical or
&	address of	!	logical not
+	add	|	bitwise or
(type)	cast	&	bitwise and
-	subtract	~	bitwise not
/	divide	^	bitwise exclusive or
*	multiply	<<	left shift
=	assignment	>>	right shift
==	comparison	()	FORTRAN array index
<<	left shift	%	FORTRAN field selector
>>	right shift		

Commands Summary

This section contains a brief summary of the PGDBG debugger commands. For more detailed information on a command, see the section number associated with the command. If you are viewing an online version of this manual, select the hyperlink under the selection category to jump to that section in the manual.

Table 1-3: PGDBG Commands

Name	Arguments	Section
`arri[ve]`		"Program Locations" on page 65
`att[ach]`	<pid> [<exe>] | [<exe> <host>]	"Process Control" on page 51
`ad[dr]`	[n | line | func | var | arg]	"Conversions" on page 79
`al[ias]`	[name [string]]	"Miscellaneous" on page 80
`asc[ii]`	exp [,...exp]	"Printing Variables and Expressions" on page 67
`as[sign]`	var=exp	"Symbols and Expressions" on page 71
`bin`	exp [,...exp]	"Printing Variables and Expressions" on page 67
`b[reak]`	[line | func] [if (condition)] [do {commands}]	"Events" on page 56
`breaki`	[addr | func] [if (condition)] [do {commands}]	"Events" on page 56
`breaks`		"Events" on page 56
`call`	func [(exp,...)]	"Symbols and Expressions" on page 71
`catch`	[number [,number...]]	"Events" on page 56
`cd`	[dir]	"Program Locations" on page 65
`clear`	[all | func | line | addr {addr}]	"Events" on page 56

Name	Arguments	Section
`c[ont]`		“Process Control” on page 51
`cr[ead]`	addr	“Memory Access” on page 77
`de[bug ]`		“Process Control” on page 51
`dec`	exp [,...exp]	“Printing Variables and Expressions” on page 67
`decl[aration]`	name	“Symbols and Expressions” on page 71
`decls`	[func | "sourcefile" | {global}]	“Scope” on page 74
`del[ete]`	event-number | all | 0 | event-number [,.event-number.]	“Events” on page 56
`det[ach`		“Process Control” on page 51
`dir[ectory]`	[pathname]	“Miscellaneous” on page 80
`dis[asm]`	[count | lo:hi | func | addr, count]	“Program Locations” on page 65
`disab[le]`	event-number | all	“Printing Variables and Expressions” on page 67
`display`	exp [,...exp]	“Printing Variables and Expressions” on page 67
`do`	{commands} [at line | in func] [if (condition)]	“Events” on page 56
`doi`	{commands} [at addr | in func] [if (condition)]	“Events” on page 56

Name	Arguments	Section
down		“Scope” on page 74
defset	name [p/t-set]	“Process-Thread Sets” on page 55
dr[ead]	addr	“Memory Access” on page 77
du[mp]	address, count, "format-string"	“Memory Access” on page 77
edit	[filename | func]	“Program Locations” on page 65
enab[le]	event-number | all	“Events” on page 56
en[ter]	func | "sourcefile" | {global}	“Scope” on page 74
entr[y]	func	“Symbols and Expressions” on page 71
fil[e]		“Program Locations” on page 65
files		“Scope” on page 74
focus	[p/t-set]	“Process-Thread Sets” on page 55
fp		“Register Access” on page 76
fr[ead]	addr	“Memory Access” on page 77
func[tion]	[addr | line]	“Conversions” on page 79
glob[al]		“Global Commands” on page 121

Name	Arguments	Section
`halt`	[command]	"Process Control" on page 51
`he[lp]`		"Miscellaneous" on page 80
`hex`	Exp [,...exp]	"Printing Variables and Expressions" on page 67
`hi[story]`	[num]	"Miscellaneous" on page 80
`hwatch`	addr [if (condition)] [do {commands}]	"Events" on page 56
`hwatchb[oth]`	addr [if (condition)] [do {commands}]	"Events" on page 56
`hwatchr[ead]`	addr [if (condition)] [do {commands}]	"Events" on page 56
`ignore`	[number [,number...]]	"Events" on page 56
`ir[ead]`	addr	"Memory Access" on page 77
`language`		"Miscellaneous" on page 80
`lin[e]`	[n | func | addr]	"Conversions" on page 79
`lines`	routine	"Program Locations" on page 65
`lis[t]`	[count | line,count | lo:hi | routine]	"Program Locations" on page 65
`log`	filename	"Miscellaneous" on page 80
`lv[al]`	exp	"Symbols and Expressions" on page 71

Name	Arguments	Section
`mq[dump]`		"Memory Access" on page 77
`names`	[func | "sourcefile" | {global}]	"Scope" on page 74
`n[ext]`	[count]	"Process Control" on page 51
`nexti`	[count]	"Process Control" on page 51
`nop[rint]`	exp	"Miscellaneous" on page 80
`oct`	exp [,...exp]	"Printing Variables and Expressions" on page 67
`pc`		"Register Access" on page 76
`pgienv`	[command]	"Miscellaneous" on page 80
`p[rint]`	exp1 [,...expn]	"Printing Variables and Expressions" on page 67
`printf`	"format_string", expr,...expr	"Printing Variables and Expressions" on page 67
`proc`	[number]	"Process Control" on page 51
`procs`		"Process Control" on page 51
`pwd`		"Program Locations" on page 65
`q[uit]`		"Process Control" on page 51

Name	Arguments	Section
`regs`		"Register Access" on page 76
`rep[eat]`	[first, last] | [first: last:n] | [num] | [-num]	"Miscellaneous" on page 80
`rer[un]`	[arg0 arg1 ... argn] [< inputfile] [> outputfile]	"Process Control" on page 51
`ret[addr]`		"Register Access" on page 76
`ru[n]`	[arg0 arg1 ... argn] [< inputfile] [> outputfile]	"Process Control" on page 51
`rv[al]`	expr	"Symbols and Expressions" on page 71
`sco[pe]`		"Scope" on page 74
`scr[ipt]`	filename	"Miscellaneous" on page 80
`set`	var = ep	"Symbols and Expressions" on page 71
`setenv`	name | name value	"Miscellaneous" on page 80
`sh[ell]`	arg0 [... argn]	"Miscellaneous" on page 80
`siz[eof]`	name	"Symbols and Expressions" on page 71
`sle[ep]`	time	"Miscellaneous" on page 80
`source`	filename	"Miscellaneous" on page 80

Name	Arguments	Section
`sp`		“Register Access” on page 76
`sr[ead]`	addr	“Memory Access” on page 77
`stackd[ump]`	[count]	“Program Locations” on page 65
`stack[trace]`	[count]	“Program Locations” on page 65
`stat[us]`		“Events” on page 56
`s[tep]`	[count] [up]	“Process Control” on page 51
`stepi`	[count] [up]	“Process Control” on page 51
`stepo[ut]`		“Process Control” on page 51
`stop`	[at line | in func] [var] [if (condition)] [do {commands}]	“Events” on page 56
`stopi`	[at addr | in func] [var] [if (condition)] [do {commands}]	“Events” on page 56
`sync`	[func | line]	“Process Control” on page 51
`synci`	[func | addr]	“Process Control” on page 51
`str[ing]`	exp [,...exp]	“Printing Variables and Expressions” on page 67
`thread`	number	“Process Control” on page 51

Name	Arguments	Section
`threads`		"Process Control" on page 51
`track`	expression [at line | in func] [if (condition)] [do {commands}]	"Events" on page 56
`tracki`	expression [at addr | in func] [if (condition)] [do {commands}]	"Events" on page 56
`trace`	[at line | in func] [var| func] [if (condition)] do {commands}	"Events" on page 56
`tracei`	[at addr | in func] [var] [if (condition)] do {commands}	"Events" on page 56
`type`	expr	"Symbols and Expressions" on page 71
`unal[ias]`	name	"Miscellaneous" on page 80
`undefset`	[name | -all]	"Process-Thread Sets" on page 55
`undisplay`	[all | 0 | exp]	"Printing Variables and Expressions" on page 67
`unb[reak]`	line | func | all	"Events" on page 56
`unbreaki`	addr | func | all	"Events" on page 56
`up`		"Scope" on page 74
`use`	[dir]	"Miscellaneous" on page 80
`viewset`	name	"Process-Thread Sets" on page 55

Name	Arguments	Section
`wait`	[any | all | none]	"Process Control" on page 51
`wa[tch]`	expression [at line | in func] [if (condition)] [do {commands}]	"Events" on page 56
`watchi`	expression [at addr | in func] [if(condition)] [do {commands}]	"Events" on page 56
`whatis`	[name]	"Symbols and Expressions" on page 71
`when`	[at line | in func] [if (condition)] do {commands}	"Events" on page 56
`wheni`	[at addr | in func] [if(condition)] do {commands}	"Events" on page 56
`w[here]`	[count]	"Program Locations" on page 65
`whereis`	name	"Symbols and Expressions" on page 71
`whichsets`	[p/t-set]	"Process-Thread Sets" on page 55
`which`	name	"Scope" on page 74
`/`	/ [string] /	"Program Locations" on page 65
`?`	?[string] ?	"Program Locations" on page 65
`!`	History modification	"Miscellaneous" on page 80
`^`	History modification	"Miscellaneous" on page 80

PGDBG Command Reference

This section describes the PGDBG command set in detail.

Notation Used in this Section

Command names may be abbreviated by omitting the portion of the command name enclosed in brackets ([]).. Some commands accept a variety of arguments. Arguments enclosed in brackets([]) are optional. Two or more arguments separated by a vertical line (|) indicate that any one of the arguments is acceptable. An ellipsis (...) indicates an arbitrarily long list of arguments. Other punctuation (commas, quotes, etc.) should be entered as shown. Argument names appear in italics and are chosen to indicate what kind of argument is expected. For example:

```
lis[t] [count | lo:hi | routine | line,count]
```

indicates that the command list may be abbreviated to lis, and that it can be invoked without any arguments or with one of the following: an integer count, a line range, a routine name, or a line and a count.

Process Control

The following commands, together with the breakpoints described in the next section, control the execution of the target program. PGDBG lets you easily group and control multiple threads and processes. See "Process and Thread Control" on page 122 for more details.

attach

```
att[ach] pid [exe] | [exe host]
```

Attach to a running process with process ID pid. If the process is not running on the local host, then specify the absolute path of the executable file exe and the host machine name host. For example, attach 1234 will attempt to attach to a running process whose process ID is 1234 on the local host. On a remote host, you may enter something like attach 1234 /home/demo/a.out myhost. In this example, PGDBG will try to attach to a process ID 1234 called /home/demo/a.out on a host named myhost.

PGDBG will attempt to infer the arguments of the attached target application. If PGDBG fails to infer the argument list, then the program behavior is undefined if the run or rerun command is executed on the attached process. This means that run and rerun should not be used for most attached MPI programs.

The stdio channel of the attached process remains at the terminal from which the program was originally invoked.

cont

```
c[ont]
```

Continue execution from the current location.

debug

```
de[bug] [target [ arg1 _ argn]]
```

Load the specified target program with optional command line arguments.

detach

```
det[ach]
```

Detach from the current running process.

halt

```
halt
```

Halt the running process or thread.

next

```
n[ext] [count]
```

Stop after executing one source line in the current routine. This command steps over called routines. The count argument stops execution only after executing count source lines.

nexti

```
nexti [count]
```

Stop after executing one instruction in the current routine. This command steps over called routines. The count argument stops execution only after executing count instructions.

proc

```
proc [id]
```

Set the current process to the process identified by id. When issued with no argument, proc lists the current program location of the current thread of the current process. See "Multi-Process MPI Debugging" on page 103, for information on how processes are numbered.

procs

```
procs
```

Print the status of all active processes. Each process is listed by its logical process ID.

quit

```
q[uit]
```

Terminate the debugging session.

rerun

```
rer[un]
rer[un] [arg0 arg1 ... argn] [< inputfile ] [ [ > | >& | >> | >>& ] outputfile
]
```

The rerun command is the same as run except if no args are specified, the previously used target arguments are not re-used.

run

```
ru[n]
ru[n] [arg0 arg1 ...argn] [< inputfile ] [ [ > | >& | >> | >>& ] outputfile ]
```

Execute program from the beginning. If arguments arg0, arg1,.. are specified, they are set up as the command line arguments of the program. Otherwise, the arguments for the previous run command are used. Standard input and standard output for the target program can be redirected using < or > and an input or output filename.

step

```
s[tep]
s[tep] count
s[tep] up
```

Stop after executing one source line. This command steps into called routines. The count argument stops execution after executing count source lines. The up argument stops execution after stepping out of the current routine (see stepout). In a parallel region of code, step applies only to the currently active thread.

stepi

```
stepi
stepi count
stepi up
```

Stop after executing one instruction. This command steps into called routines. The count argument stops execution after executing count instructions. The up argument stops the execution after stepping out of the current routine (see stepout). In a parallel region of code, stepi applies only to the currently active thread.

stepout

```
stepo[ut]
```

Stop after returning to the caller of the current routine. This command sets a breakpoint at the current return address, and does a continue. To work correctly, it must be possible to compute the value of the return address. Some routines, particularly terminal (or leaf) routines at higher optimization levels, may not set up a stack frame. Executing stepout from such a routine causes the breakpoint to be set in the caller of the most recent routine that set up a stack frame. This command halts execution immediately upon return to the calling routine.

sync/synci

```
sync
synci
```

Advance the current process/thread to a specific program location; ignoring any user defined events.

thread

```
thread [number]
```

Set the current thread to the thread identified by number; where number is a logical thread id in the current process' active thread list. When issued with no argument, thread lists the current program location of the currently active thread.

threads

```
threads
```

Print the status of all active threads. Threads are grouped by process. Each process is listed by its logical process id. Each thread is listed by its logical thread id.

wait

```
wait [any | all | none]
```

Return the PGDBG prompt only after specific processes or threads stop.

Process-Thread Sets

The following commands deal with defining and managing process thread sets. See "Process/Thread Sets" on page 110, for a detailed discussion of process-thread sets.

defset

```
defset
```

Assign a name to a process/thread set. Define a named set. This set can later be referred to by name. A list of named sets is stored by PGDBG.

focus

```
focus
```

Set the target process/thread set for commands. Subsequent commands will be applied to the members of this set by default.

undefset

```
undefset
```

Remove a previously defined process/thread set from the list of process/thread sets. The debugger-defined p/t-set [all] cannot be removed.

viewset

```
viewset
```

List the members of a process/thread set that currently exist as active threads or list defined p/t-sets.

whichsets

```
whichsets
```

List all defined p/t-sets to which the members of a process/thread set belong.

Events

The following commands deal with defining and managing events. See "Parallel Events" on page 129, for a general discussion of events and the optional arguments.

break

```
b[reak]
b[reak] *line* [if
(*condition*)] [do {*commands*}]
b[reak] *routine* [if
(*condition*)] [do {*commands*}]
```

If no argument is specified, print the current breakpoints. Otherwise, set a breakpoint at the indicated line or routine. If a routine is specified, and the routine was compiled for debugging, then the breakpoint is set at the start of the first statement in the routine (after the routine's prologue code). If the routine was not compiled for debugging, then the breakpoint is set at the first instruction of the routine, prior to any prologue code. This command interprets integer constants as line numbers. To set a breakpoint at an address, use the addr command to convert the constant to an address, or use the breaki command.

When a condition is specified with if, the breakpoint occurs only when the specified condition is true. If do is specified with a command or several commands as an argument, the command or commands are executed when the breakpoint occurs.

The following examples set breakpoints at line 37 in the current file, line 37 in file xyz.c, the first executable line of routine main, address 0xf0400608, the current line, and the current address, respectively.

```
break 37

break "xyz.c"@37

break main

break {addr 0xf0400608}

break {line}

break {pc}
```

More sophisticated examples include:

```
break xyz if(xyz@n > 10)
```

This command stops when routine xyz is entered only if the argument n is greater than 10.

```
break 100 do {print n; stack}
```

This command prints the value of n and performs a stack trace every time line 100 in the current file is reached.

breaki

```
breaki
breaki routine [if (condition)] [do {commands}]
breaki addr [if (condition)] [do {commands}]
```

Set a breakpoint at the indicated address or routine. If a routine is specified, the breakpoint is set at the first address of the routine. This means that when the program stops at this breakpoint the prologue code which sets up the stack frame will not yet have been executed, so values of stack arguments may not yet be correct. Integer constants are interpreted as addresses. To specify a line, use the line command to convert the constant to a line number, or use the break command.

The if and do arguments are interpreted in the same way as for the break command. The next set of examples set breakpoints at address 0xf0400608, line 37 in the current file, line 37 in file xyz.c, the first executable address of routine main, the current line, and the current address, respectively:

```
breaki 0xf0400608
breaki {line 37}
breaki "xyz.c"@37
breaki main
breaki {line}
breaki {pc}
```

Similarly,

```
breaki 0x6480 if(n>3) do {print "n=",
n}
```

stops and prints the new value of n at address 0x6480 only if n is greater than 3.

breaks

```
breaks
```

Display all the existing breakpoints.

catch

```
catch
catch [sig:sig]
catch [sig [, sig...]]
```

With no arguments, print the list of signals being caught. With the sig:sig argument, catch the specified range of signals. With a list, catch signals with the specified number(s). When signals are caught, PGDBG intercepts the signal and does not deliver it to the target application. The target runs as though the signal was never sent.

clear

```
clear
clear all
clear routine
clear line
clear addr {addr}
```

Clear all breakpoints at current location. Clear all breakpoints. Clear all breakpoints from first statement in the specified routine named routine. Clear breakpoints from line number line. Clear breakpoints from the address addr.

delete

```
del[ete] event-number
del[ete] 0
del[ete] all
del[ete] event-number [, event-number...]
```

Delete the event event-number or all events (delete 0 is the same as delete all). Multiple event numbers can be supplied if they are separated by commas.

disable

```
disab[le] event-number
disab[le] all
```

Disable the event event-number or all events. Disabling an event definition suppresses actions associated with the event, but leaves the event defined so that it can be used later.

do

```
do {commands} [if
(condition)]
do {commands} at line [if
(condition)]
do {commands} in routine [if
(condition)]
```

Define a do event. This command is similar to watch except that instead of defining an expression, it defines a list of commands to be executed. Without the optional arguments at or in, the commands are executed at each line in the program. The at argument with a line specifies the commands to be executed each time that line is reached. The in argument with a routine specifies the commands are executed at each line in the routine. The if option has the same meaning as in watch. If a condition is specified, the do commands are executed only when condition is true.

doi

```
doi {commands} [if
(condition)]
doi {commands} at addr [if
(condition)]
doi {commands} in routine [if
(condition)]
```

Define a doi event. This command is similar to watchi except that instead of defining an expression, it defines a list of commands to be executed. If an address (addr) is specified, then the commands are executed each time that the specified address is reached. If a routine (routine) is specified, then the commands are executed at each instruction in the routine. If neither is specified, then the commands are executed at each instruction in the program. The if option has the same meaning as for the do command above.

enable

```
enab[le] event-number | all
```

Enable the indicated event event-number, or all events.

hwatch

```
hwatch addr | var [if
(condition)] [do {commands}]
```

Define a hardware watchpoint. This command uses hardware support to create a watchpoint for a particular address or variable. The event is triggered by hardware when the byte at the given address is written. This command is only supported on systems that provide the necessary hardware and software support. Only one hardware watchpoint can be defined at a time.

When the if option is specified, the event action will only be triggered if the expression is true. When the do option is specified, then the commands will be executed when the event occurs.

hwatchr

```
hwatchr[ead] addr | var [if
(condition)] [do {commands}]
```

Define a hardware read watchpoint. This event is triggered by hardware when the byte at the given address or variable is read. As with hwatch, system hardware and software support must exist for this command to be supported. The if and do options have the same meaning as for the hwatch command.

hwatchb

```
hwatchb[oth] addr | var [if
(condition)] [do {commands}]
```

Define a hardware read/write watchpoint. This event is triggered by hardware when the byte at the given address or variable is either read or written. As with hwatch, system hardware and software support must exist for this command to be supported. The if and do options have the same meaning as for the hwatch command.

ignore

```
ignore
ignore [sig:sig]
ignore [sig [, sig...]]
```

When no arguments are specified, the ignore command will print the list of signals being ignored. With the sig:sig argument it will ignore the specified range of signals, and with a list of signals it will ignore signals with the specified number. When a particular signal number is ignored, signals with that number sent to the target application are not intercepted by PGDBG. They are delivered to the target. See also catch.

status

```
stat[us]
```

Display all the event definitions, including an event number by which the event can be identified.

stop

```
stop varstop at line [if (condition)][do {commands}]
stop in routine [if (condition)][do {commands}]
stop if (condition)
```

Set a breakpoint at the indicated routine or line. Break when the value of the indicated variable var changes. The at keyword and a number specifies a line number. The in keyword and a routine name specifies the first statement of the specified routine. With the if keyword, the debugger stops when the condition condition is true.

stopi

```
stopi var
stopi at address [if (condition)][do {commands}]
stopi in routine [if (condition)][do {commands}]
stopi if (condition)
```

Set a breakpoint at the indicated address or routine. Break when the value of the indicated variable var changes. The at keyword and a number specifies an address to stop at. The in keyword and a routine name specifies the first address of the specified routine to stop at. With the if keyword, the debugger stops when condition is true.

track

```
track expression [at line | in func] [if
(condition)][do {commands}]
```

Define a track event. This command is equivalent to watch except that execution resumes after the new value of the expression is printed.

tracki

```
tracki expression [at addr | in func] [if
(condition)][do {commands}]
```

Define an instruction level track event. This command is equivalent to watchi except that execution resumes after the new value of the expression is printed.

trace

```
trace var [if (condition)][do {commands}]
trace routine [if (condition)][do {commands}]
trace at line [if (condition)][do {commands}]
trace in routine [if (condition)][do {commands}]
```

With the var argument, activate source line tracing when var changes. When a routine is specified, activate source line tracing and trace when in subprogram routine. With the at keyword, activate source line tracing to display the specified line each time it is executed. With in, activate source line tracing when in the specified routine. If *condition* is specified, trace is on only if the condition evaluates to true. The do keyword defines a list of commands to execute at each trace point. Use the command pgienv speed secs to set the time in seconds between trace points. Use the clear command to remove tracing for a line or routine.

tracei

```
tracei var [if (condition)][do {commands}]
tracei routine [if (condition)][do {commands}]
tracei at line [if (condition)][do {commands}]
tracei in routine [if (condition)][do {commands}]
```

With the var argument, activate instruction tracing when var changes. When a routine is specified, activate instruction tracing and trace when in subprogram routine. With the at keyword, activate instruction tracing to display the specified line each time it is executed. With in, activate instruction tracing when in the specified routine. If *condition* is specified, trace is on only if the condition evaluates to true. The do keyword defines a list of commands to execute at each trace point. Use the command pgienv speed secs to set the time in seconds between trace points. Use the clear command to remove tracing for a line or routine.

unbreak

```
unb[reak] line
unb[reak] routine
unb[reak] all
```

Remove a breakpoint from the statement line, the routine routine, or remove all breakpoints.

unbreaki

```
unbreaki addr
unbreaki routine
unbreaki all
```

Remove a breakpoint from the address addr, the routine routine, or remove all breakpoints.

watch

```
wa[tch] expression
wa[tch] expression [if
(condition)][do {commands}]
wa[tch] expression at line [if
(condition)][do {commands}]
wa[tch] expression in routine [if
(condition)][do {commands}]
```

Define a watch event. The given expression is evaluated, and subsequently, each time the value of the expression changes, the program stops and the new value is printed. If a particular line is specified, the expression is only evaluated at that line. If a routine routine is specified, the expression is evaluated at each line in the routine. If no location is specified, the expression will be evaluated at each line in the program. If a condition is specified, the expression is evaluated only when the condition is true. If *commands* are specified, they are executed whenever the expression is evaluated and the value changes.

The watched expression may contain local variables, although this is not recommended unless a routine or address is specified to ensure that the variable will only be evaluated when it is in the current scope.

NOTE

Using watchpoints indiscriminately can dramatically slow program execution.

Using the at and in options speeds up execution by reducing the amount of single-stepping and expression evaluation that must be performed to watch the expression. For example:

```
watch i at 40
```

will barely slow program execution at all, while

```
watch i
```

will slow execution considerably.

watchi

```
watchi expression
watchi expression [if (condition)][do {commands}]
watchi expression at addr [if
(condition)][do {commands}]
watchi expression in routine [if
(condition)][do {commands}]
```

Define an instruction level watch event. This is just like the watch command except that the at option interprets integers as addresses rather than line numbers and the *expression* is evaluated at every instruction rather than at every line.

This command is useful if line number information is limited (i.e. code not compiled '-g' or assembly code). It causes programs to execute more slowly than watch.

when

```
when do {commands} [if
(condition)]
when at line do {commands} [if
(condition)]
when in routine do {commands} [if
(condition)]
```

Execute *commands* at every line in the program, at a specified line in the program or in the specified routine. If the optional condition is specified, *commands* are executed only when the expression evaluates to true.

wheni

```
wheni do {commands} [if
(condition)]
wheni at addr do {commands} [if
(condition)]
wheni in routine do {commands} [if
(condition)]
```

Execute *commands* at each address in the program. If an addr is specified, the commands are executed each time the address is reached. If a routine is specified, the commands are executed at each line in the routine. If the optional *condition* is specified, commands are executed whenever the expression is evaluated true.

Events can be parallelized across multiple threads of execution. See "Parallel Events" on page 129, for details.

Program Locations

This section describes PGDBG program location commands.

arrive

```
arri[ve]
```

Print location information for the current location.

cd

```
cd [dir]
```

Change to the $HOME directory or to the specified directory dir.

disasm

```
dis[asm]
dis[asm] count
dis[asm] lo:hi
dis[asm] routine
dis[asm] addr, count
```

Disassemble memory. If no argument is given, disassemble four instructions starting at the current address. If an integer count is given, disassemble count instructions starting at the current address. If an address range (lo:hi) is given, disassemble the memory in the range. If a routine name is given, disassemble the entire routine. If the routine was compiled for debugging (-g), and source code is available, the source code will be interleaved with the disassembly. If an address and a count are given, disassemble count instructions starting at address addr.

edit

```
edit
edit filenameedit routine
```

If no argument is supplied, edit the current file starting at the current location. With a filename argument, edit the specified file filename. With the func argument, edit the file containing routine routine. This command uses the editor specified by the environment variable $EDITOR.

In the PGDBG GUI, command line editors like vi are launched in the Program I/O Window. On Windows platforms, arguments to the editor may need to be quoted to account for spaces in pathnames.

file

```
file [filename]
```

Change the source file to the file filename and change the scope accordingly. With no argument, print the current file.

lines

```
lines routine
```

Print the lines table for the specified routine.

list

```
lis[t]
lis[t] count
lis[t] line,num
lis[t] lo:hi
lis[t] routine
```

With no argument, list 10 lines centered at the current source line. If a count is given, list count lines centered at the current source line. If a line and count are given, list number lines starting at line number line. In dbx mode, this option lists lines from start to number. If a line range is given, list the indicated source lines in the current source file (this option is not valid in the dbx environment). If a routine name is given, list the source code for the indicated routine.

pwd

```
pwd
```

Print the current working directory.

stacktrace

```
stack[trace] [count]
```

Print a stacktrace. For each active routine print the routine name, source file, line number, current address (if that information is available). This command also prints the names and values of the arguments, if available. If a count is specified, display a maximum of count stack frames.

stackdump

```
stackd[ump] [count]
```

Print a formatted dump of the stack. This command displays a hex dump of the stack frame for each active routine. This command is a machine-level version of the stacktrace command. If a count is specified, display a maximum of count stack frames.

where

```
w[here] [count]
```

Print a stacktrace. For each active routine print the routine name, source file, line number, current address (if that information is available). This command also prints the names and values of the arguments, if available. If a count is specified, display a maximum of count stack frames.

/ (search forward)

```
/ / [string] /
```

Search forward for a string (string) of characters in the current source file. With just /, search for the next occurrence of string in the current source file.

? (search backward)

```
? ?[string] ?
```

Search backward for a string (string) of characters in the current source file. With just ?, search for the previous occurrence of string in the current source file.

Printing Variables and Expressions

This section describes PGDBG commands used for printing and setting variables.

print

```
p[rint] exp1 [,...expn]
```

Evaluate and print one or more expressions. This command is invoked to print the result of each line of command input. Values are printed in a format appropriate to their type. For values of structure type, each field name and value is printed. Character pointers are printed as a hex address followed by the character string.

Character string constants print out literally. For example:

```
pgdbg> print "The value of i is ", i
The value of i is 37
```

The array sub-range operator : prints a range of an array. The following examples print elements 0 through 9 of the array a:

C/C++ example:

```
pgdbg> print a[0:9]
a[0:4]: 0 1 2 3 4
a[5:9]: 5 6 7 8 9
```

FORTRAN example:

```
pgdbg> print a(0:9)
a(0:4): 0 1 2 3 4
a(5:9): 5 6 7 8 9
```

Note that the output is formatted and annotated with index information. PGDBG formats array output into columns. For each row, the first column prints an index expression which summarizes the elements printed in that row. Elements associated with each index expression are then printed in order. This is especially useful when printing slices of large multidimensional arrays.

PGDBG also supports strided array expressions. Below are examples for C/C++ and FORTRAN.

C/C++ example:

```
pgdbg> print a[0:9:2]
a[0:8] 0 2 4 6 8
```

FORTRAN example:

```
pgdbg> print a(0:9:2)
a(0:8): 0 2 4 6 8
```

The print statement may be used to display members of derived types in FORTRAN or structures in C/C++. Below are examples.

```
C/C++ example:

typedef struct tt {
int a[10];
}TT;
TT d = {0,1,2,3,4,5,6,7,8,9};
TT * p = &d;
```

```
pgdbg> print d.a[0:9:2]
d.a[0:8:2]: 0 2 4 6 8

pgdbg> print p->a[0:9:2]
p->a[0:7:2]: 0 2 4 6
p->a[8]: 8
```

FORTRAN example:

```
type tt
integer, dimension(0:9) :: a
end type
type (tt) :: d
data d%a / 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 /
```

```
pgdbg> print d%a(0:9:2)
d%a(0:8:2): 0 2 4 6 8
```

printf

```
printf "format_string", expr,...expr
```

Print expressions in the format indicated by the format string. Behaves like the C library function printf. For example:

```
pgdbg> printf "f[%d]=%G",i,f[i]
f[3]=3.14
```

The pgienv command with the stringlen argument sets the maximum number of characters that will print with a print command. For example, the char declaration below:

```
char *c="a whole bunch of chars over
1000 chars long....";
```

By default, the print c command will only print the first 512 (or stringlen) bytes. Printing of C strings is usually terminated by the terminating null character. This limit is a safeguard against unterminated C strings.

ascii

```
asc[ii] exp [,...exp]
```

Evaluate and print as an ascii character. Control characters are prefixed with the '^' character; that is, 3 prints as ^c. Otherwise, values that can not be printed as characters are printed as integer values prefixed by `\'. For example, 250 prints as \250.

bin

```
bin exp [,...exp]
```

Evaluate and print the expressions. Integer values are printed in binary.

dec

```
dec exp [,...exp]
```

Evaluate and print the expressions. Integer values are printed in decimal.

display

```
display
display exp [,...exp]
```

Without arguments, list the expressions for PGDBG to automatically display at breakpoints. With an argument or several arguments, print expression exp at every breakpoint. See also: undisplay.

hex

```
hex exp [,...exp]
```

Evaluate and print expressions as hexadecimal integers.

oct

```
oct exp [,...exp]
```

Evaluate and print expressions as octal integers.

string

```
str[ing] exp [,...exp]
```

Evaluate and print expressions as null-terminated character strings. This command will print a maximum of 70 characters.

undisplay

```
undisplay 0
undisplay all
undisplay exp [,...exp]
```

Remove all expressions specified by previous display commands. With an argument or several arguments, remove the expression exp from the list of display expressions.

Symbols and Expressions

This section describes the commands that deal with symbols and expressions.

assign

```
as[sign] var = exp
```

Set variable var to the value of expression. The variable var can be any valid identifier accessed properly for the current scope. For example, given a C variable declared 'int * i', the command 'set *i = 9999' could be used to assign the value 9999 to it.

call

```
call routine [(exp,...)]
```

Call the named routine. C argument passing conventions are used. Breakpoints encountered during execution of the routine are ignored. Fortran functions and subroutines can be called, but the argument values will be passed according to C conventions. PGDBG may not always be able to access the return value of a Fortran function if the return value is an array. In the example below, PGDBG calls the routine foo with four arguments:

```
pgdbg> call foo(1,2,3,4)
```

If a signal is caught during execution of the called routine, PGDBG will stop the execution and ask if you want to cancel the call command. For example, suppose a command is issued to call foo as shown above, and for some reason a signal is sent to the process while it is executing the call to foo. In this case, PGDBG will print the following prompt:

```
PGDBG Message: Thread [0] was signalled
while executing a function reachable from the most recent PGDBG
command line call to foo. Would you like to cancel this command
line call? Answering yes will revert the register state of Thread [0] back
to the state it had prior to the last call to foo from the command
line. Answering no will leave Thread [0] stopped
in the call to foo from the command line
Please enter 'y' or 'n' > y
Command line call to foo cancelled
```

Answering yes to this question will return the register state of each thread back to the state they had before invoking the call command. Answering no to this question will leave each thread at the point they were at when the signal occurred.

Note: Answering no to this question and continuing execution of the called routine may produce unpredictable results.

declaration

```
decl[aration] name
```

Print the declaration for the symbol based on its type according to symbol table. The symbol must be a variable, argument, enumeration constant, routine, a structure, union, enum, or typedef tag.

For example, given the C declarations:

```
int i, iar[10];
struct abc {int a; char b[4]; struct
abc *c;}val;
```

the decl command will provide the following output:

```
pgdbg> decl I
int i
pgdbg> decl iar
int iar[10]
pgdbg> decl val
struct abc val

pgdbg> decl abc
struct abc {
 int a;
 char b[4];
 struct abc *c;
};
```

entry

```
entr[y]
entr[y] routine
```

Return the address of the first executable statement in the program or specified routine. This is the first address after the routine's prologue code.

lval

```
lv[al] expr
```

Return the lvalue of the expression expr. The lvalue of an expression is the value it would have if it appeared on the left hand of an assignment statement. Roughly speaking, an lvalue is a location to which a value can be assigned. This may be an address, a stack offset, or a register.

rval

```
rv[al] expr
```

Return the rvalue of the expression expr. The rvalue of an expression is the value it would have if it appeared on the right hand of an assignment statement. The type of the expression may be any scalar, pointer, structure, or function type.

set

```
set var=expression
```

Set variable var to the value of expression. The variable var can be any valid identifier accessed properly for the current scope. For example, given a C variable declared 'int * i', the command 'set *i = 9999' could be used to assign the value 9999 to it.

sizeof

```
siz[eof] name
```

Return the size, in bytes, of the variable type name. If name refers to a routine, sizeof will return the size on bytes of the subprogram.

type

```
type expr
```

Return the type of the expression. The expression may contain structure reference operators (. , and ->), dereference (*), and array index ([ ]) expressions. For example, given the C declarations:

```
int i, iar[10];
struct abc {int a; char b[4]; struct
abc *c;}val;
```

the type command will provide the following output:

```
pgdbg> type I
int
pgdbg> type iar
int [10]
pgdbg> type val
struct abc
pgdbg> type val.a
int

pgdbg> type val.abc->b[2]
char

pgdbg> whatis
whatis name
```

With no arguments, print the declaration for the current routine. With argument name, print the declaration for the symbol name.

Scope

The following commands deal with program scope. See "Scope Rules" on page 32, for a discussion of scope meaning and conventions.

decls

```
declsdecls routine
decls "sourcefile"
decls {global}
```

Print the declarations of all identifiers defined in the indicated scope. If no scope is given, print the declarations for global scope.

down

```
down [number]
```

Enter scope of routine down one level or number levels on the call stack.

enter

```
en[ter]
en[ter] routine
en[ter] "sourcefile"
en[ter] {global}
```

Set the search scope to be the indicated symbol, which may be a routine, source file or global. Using enter with no argument is the same as using enter global.

files

```
files
```

Return the list of known source files used to create the executable file.

global

```
glob[al]
```

Return a symbol representing global scope. This command is useful in combination with the scope operator @ to specify symbols with global scope.

names

```
names
names routine
names "sourcefile"
names {global}
```

Print the names of all identifiers defined in the indicated scope. If no scope is specified, use the search scope.

scope

```
sco[pe]
```

Return a symbol for the search scope. The search scope is set to the current routine each time program execution stops. It may also be set using the enter command. The search scope is always searched first for symbols.

up

```
up [number]
```

Enter scope of routine up one level or number levels on the call stack.

whereis

```
whereis name
```

Print all declarations for name.

which

```
which name
```

Print full scope qualification of symbol name.

Register Access

System registers can be accessed by name. See "Register Symbols" on page 89, for the complete set of registers and how to refer to them in PGDBG. A few commands exist for convenient access to common registers.

fp

```
fp
```

Return the current value of the frame pointer.

pc

```
pc
```

Return the current program address.

regs

```
regs [format]
```

Print a formatted display of the names and values of the integer, float, and double registers. If the format parameter is omitted, then PGDBG will print all of the registers. Otherwise, regs accepts the following optional parameters:

f	Print floats as single precision values (default)
d	Print floats as double precision values
x	Add hexadecimal representation of float values

retaddr

```
ret[addr]
```

Return the current return address.

sp

```
sp
```

Return the current value of the stack pointer.

Memory Access

The following commands display the contents of arbitrary memory locations. Note that for each of these commands, the addr argument may be a variable or identifier.

cread

```
cr[ead] addr
```

Fetch and return an 8-bit signed integer (character) from the specified address.

dread

```
dr[ead] addr
```

Fetch and return a 64 bit double from the specified address.

dump

```
du[mp] address, count,
"format-string"
```

This command dumps the contents of a region of memory. The output is formatted according to a printf-like format descriptor. Starting at the indicated address, values are fetched from memory and displayed according to the format descriptor. This process is repeated count times.

Interpretation of the format descriptor is similar to printf. Format specifiers are preceded by %.

The meaning of the recognized format descriptors is as follows:

```
%d, %D, %o, %O, %x, %X, %u, %U
```

Fetch and print integral values as decimal, octal, hex, or unsigned. Default size is machine dependent. The size of the item read can be modified by either inserting 'h', or 'l' before the format character to indicate half word or long word. For example, if your machine's default size is 32-bit, then %hd represents a 16-bit quantity. Alternatively, a 1, 2, or 4 after the format character can be used to specify the number of bytes to read.

%c

Fetch and print a character.

```
%f, %F, %e, %E, %g, %G
```

Fetch and print a float (lower case) or double (upper case) value using printf f, e, or g format.

```
%s
```

Fetch and print a null terminated string.

`%p<`*`format-chars`*`>`

Interpret the next object as a pointer to an item specified by the following format characters. The pointed-to item is fetched and displayed. Examples:

```
%px
```

Pointer to int. Prints the value of the pointer, the pointed-to address, and the contents of the pointed-to address, which is printed using hexadecimal format.

```
%i
```

Fetch an instruction and disassemble it.

```
%w, %W
```

Display address about to be dumped.

```
%z<n>, %Z<n>, %z<-n>, %Z<-n>
```

Display nothing but advance or decrement current address by n bytes.

```
%a<n>, %A<n>
```

Display nothing but advance current address as needed to align modulo n.

fread

`fr[ead]`*`addr`*

Fetch and print a 32-bit float from the specified address.

iread

```
ir[ead] addr
```

Fetch and print a signed integer from the specified address.

lread

```
lr[ead] addr
```

Fetch and print an address from the specified address.

mqdump

```
mq[dump]
```

Dump MPI message queue information for the current process. Refer to "MPI Message Queues" on page 137, for more information on mqdump.

sread

```
sr[ead]addr
```

Fetch and print a short signed integer from the specified address.

Conversions

The commands in this section are useful for converting between different kinds of values. These commands accept a variety of arguments, and return a value of a particular kind.

addr

```
ad[dr]
ad[dr] n
ad[dr] line
ad[dr] routine
ad[dr] var
ad[dr] arg
```

Create an address conversion under these conditions:

- If an integer is given return an address with the same value.
- If a line is given, return the address corresponding to the start of that line.

- If a routine is given, return the first address of the routine.
- If a variable or argument is given, return the address where that variable or argument is stored.

For example:

```
breaki {line {addr 0x22f0}}
```

function

```
func[tion]
func[tion] addr
func[tion] line
```

Return a routine symbol. If no argument is specified, return the current routine. If an address is given, return the routine containing addr. An integer argument is interpreted as an address. If a line is specified, return the routine containing that line.

line

```
lin[e]
lin[e] nlin[e] routinelin[e] addr
```

Create a source line conversion. If no argument is given, return the current source line. If an integer n is given, return it as a line number. If a routine is given, return the first line of the routine. If an address addr is given, return the line containing that address.

For example, the following command returns the line number of the specified address:

```
line {addr 0x22f0}
```

Miscellaneous

The following commands provide shortcuts, mechanisms for querying, customizing and managing the PGDBG environment, and access to operating system features.

alias

```
al[ias]
al[ias] name
al[ias] name string
```

Create or print aliases. If no arguments are given print all the currently defined aliases. If just a name is given, print the alias for that name. If a name and string are given, make name an alias for string. Subsequently, whenever name is encountered it will be replaced by string. Although string may be an arbitrary string, name must not contain any space characters.

For example:

```
alias xyz print "x= ",x,"y= ",y,"z= ",z;
cont
```

creates an alias for xyz. Now whenever xyz is typed, PGDBG will respond as though the following command was typed:

```
print "x= ",x,"y= ",y,"z= ",z;
cont
```

directory

```
dir[ectory] [pathname]
```

Add the directory pathname to the search path for source files. If no argument is specified, the currently defined directories are printed. This command assists in finding source code that may have been moved or is otherwise not found by the default PGDBG search mechanisms.

For example:

```
dir morestuff
```

adds the directory morestuff to the list of directories to be searched. Now, source files stored in morestuff are accessible to PGDBG.

If the first character in pathname is ~, it will be substituted by $HOME.

help

```
help [command]
```

If no argument is specified, print a brief summary of all the commands. If a command name is specified, print more detailed information about the use of that command.

history

```
history [num]
```

List the most recently executed commands. With the num argument, resize the history list to hold num commands. History allows several characters for command substitution:

!! [modifier]	Execute the previous command
! num [modifier]	Execute command number num
!-num [modifier]	Execute command -num from the most current command
!string [modifier]	Execute the most recent command starting with string
!?string? [modifier]	Execute the most recent command containing string
^	Quick history command substitution ^old^new^<modifier> this is equivalent to !:s/old/new/

The history modifiers may be:

:s/old/new/	Substitute the value new for the value old.
:p	Print but do not execute the command.

The command pgienv history off tells the debugger not to display the history record number. The command pgienv history on tells the debugger to display the history record number.

language

```
language
```

Print the name of the language of the current file.

log

```
log filename
```

Keep a log of all commands entered by the user and store it in the named file. This command may be used in conjunction with the script command to record and replay debug sessions.

noprint

```
nop[rint] exp
```

Evaluate the expression but do not print the result.

pgienv

```
pgienv [command]
```

Define the debugger environment. With no arguments, display the debugger settings.

help pgienv	Provide help on pgienv
pgienv	Display the debugger settings
pgienv dbx on	Set the debugger to use dbx style commands
pgienv dbx off	Set the debugger to use pgi style commands
pgienv history on	Display the `history' record number with prompt
pgienv history off	Do NOT display the `history' number with prompt
pgienv exe none	Ignore executable's symbolic debug information
pgienv exe symtab	Digest executable's native symbol table (typeless)
pgienv exe demand	Digest executable's symbolic debug information incrementally on command
pgienv exe force	Digest executable's symbolic debug information when executable is loaded
pgienv solibs none	Ignore symbolic debug information from shared libraries
pgienv solibs symtab	Digest native symbol table (typeless) from each shared library
pgienv solibs demand	Digest symbolic debug information from shared libraries incrementally on demand

pgienv solibs force	Digest symbolic debug information from each shared library at load time
pgienv mode serial	Single thread of execution (implicit use of p/t-sets)
pgienv mode thread	Debug multiple threads (condensed p/t-set syntax)
pgienv mode process	Debug multiple processes (condensed p/t-set syntax)
pgienv mode multilevel	Debug multiple processes and multiple threads
pgienv omp [on|off]	Enable/Disable the PGDBG OpenMP event handler. This option is disabled by default. The PGDBG OpenMP event handler, when enabled, sets breakpoints at the beginning and end of each parallel region. Breakpoints are also set at each thread synchronization point. The handler coordinates threads across parallel constructs to maintain source level debugging. This option, when enabled, may significantly slow down program performance. Enabling this option is recommended for localized debugging of a particular parallel region only.
pgienv prompt <name>	Set the command line prompt to <name>
pgienv promptlen <num>	Set maximum size of p/t-set portion of prompt
pgienv speed <secs>	Set the time in seconds <secs> between trace points
pgienv stringlen <num>	Set the maximum # of chars printed for `char *'s
pgienv termwidth <num>	Set the character width of the display terminal.
pgienv logfile <name>	Close logfile (if any) and open new logfile <name>
pgienv threadstop sync	When one thread stops, the rest are halted in place
pgienv threadstop async	Threads stop independently (asynchronously)

pgienv procstop sync	When one process stops, the rest are halted in place
pgienv procstop async	Processes stop independently (asynchronously)
pgienv threadstopcon-fig auto	For each process, debugger sets thread stopping mode to 'sync' in serial regions, and 'async' in parallel regions
pgienv threadstopcon-fig user	Thread stopping mode is user defined and remains unchanged by the debugger.
pgienv procstopconfig auto	Not currently used.
pgienv procstopconfig user	Process stop mode is user defined and remains unchanged by the debugger.
pgienv threadwait none	Prompt available immediately; no wait for running threads
pgienv threadwait any	Prompt available when at least a single thread stops
pgienv threadwait all	Prompt available only after all threads have stopped
pgienv procwait none	Prompt available immediately; no wait for running processes
pgienv procwait any	Prompt available when at least a single process stops
pgienv procwait all	Prompt available only after all processes have stopped
pgienv threadwaitcon-fig auto	For each process, the debugger will set the thread wait mode to ‘all’ in serial regions and ‘none’ in parallel regions. (default)
pgienv threadwaitcon-fig user	The thread wait mode is user defined and will remain unchanged by the debugger.

pgienv verbose <bit-mask>	Choose which debug status messages to report. Accepts an integer valued bit mask of the following values:
	• 0x1 - Standard messaging (default). Report status information on current process/thread only.
	• 0x2 - Thread messaging. Report status information on all threads of (current) processes.
	• 0x4 - Process messaging. Report status information on all processes.
	• 0x8 - OpenMP messaging (default). Report OpenMP events.
	• 0x10 - Parallel messaging (default). Report parallel events.
	• 0x20 - Symbolic debug information. Report any errors encountered while processing symbolic debug information (e.g. STABS, DWARF). Pass 0x0 to disable all messages.
	• Pass 0x0 to disable all messages.

repeat

```
rep[eat] [first, last]
rep[eat] [first,:last:n]
rep[eat] [num ]
rep[eat] [-num ]
```

Repeat the execution of one or more previous history list commands. With the num argument, re-execute the command number num, or with -num, the last num commands. With the first and last arguments, re-execute commands number first to last (optionally n times).

script

```
scr[ipt] filename
```

Open the indicated file and execute the contents as though they were entered as commands. If you use ~ before the filename, it is expanded to the value of the environment variable HOME.

setenv

```
setenv name
setenv name value
```

Print value of environment variable name. With a specified value, set name to value.

shell

```
shell [arg0, arg1,... argn]
```

Fork a shell (defined by $SHELL) and give it the indicated arguments (the default shell is sh). If no arguments are specified, an interactive shell is invoked, and executes until a "^D" is entered.

sleep

```
sle[ep] [time]
```

Pause for time seconds. If no time is specified, pause for one second.

source

```
sou[rce] filename
```

Open the indicated file and execute the contents as though they were entered as commands. If you use ~ before the filename, it is expanded to the value of $HOME.

unalias

```
unal[ias] name
```

Remove the alias definition for name, if one exists.

use

```
use [dir]
```

Print the current list of directories or add dir to the list of directories to search. If the first character in pathname is ~, it will be substituted with the value of $HOME.

Signals

PGDBG intercepts all signals sent to any of the threads in a multi-threaded program and passes them on according to that signal's disposition as maintained by PGDBG (see the catch and ignore commands), except for signals that cannot be intercepted or signals used internally by PGDBG.

Control-C

If the target application is not running, control-C can be used to interrupt long-running PGDBG commands. For example, a command requesting disassembly of thousands of instructions might run for a long time, and it can be interrupted by control-C. In such cases the target application is not affected.

If the target application is running, entering control-C at the PGDBG command prompt will halt execution of the target. This is useful in cases where the target "hangs" due to an infinite loop or deadlock,

Sending a SIGINT (control-C) to a program while it is in the middle of initializing its threads (calling omp_set_num_threads(), or entering a parallel region) may kill some of the threads if the signal is sent before each thread is fully initialized. Avoid sending SIGINT in these situations. Note that when the number of threads employed by a program is large, thread initialization may take a while.

Sending SIGINT (control-C) to a running MPI program is not recommended. See "MPI Listener Processes" on page 139, for details. Use the PGDBG halt command as an alternative to sending SIGINT to a running program. The PGDBG command prompt must be available in order to issue a halt command. The PGDBG command prompt is available while threads are running if pgienv threadwait none is set.

Signals Used Internally by PGDBG

SIGTRAP and SIGSTOP are used by Linux for communication of application events to PGDBG. Management of these signals is internal to PGDBG. Changing the disposition of these signals in PGDBG (via catch and ignore)will result in undefined behavior.

Signals Used by Linux Libraries

Some Linux thread libraries use SIGRT1 and SIGRT3 to communicate among threads internally. Other Linux thread libraries, on systems that do not have support for real-time signals in the kernel, use SIGUSR1 and SIGUSR2. Changing the disposition of these signals in PGDBG (via catch and ignore) will result in undefined behavior.

Target applications built for sample-based profiling (compiled with '-pg') generate numerous SIGPROF signals. Although SIGPROF can be handled by PGDBG, debugging of applications built for sample-based profiling is not recommended.

Register Symbols

This section describes the register symbols defined for X86 processors and EM64T/AMD64 processors operating in compatibility or legacy mode.

X86 Register Symbols

This section describes the X86 register symbols.

Table 1-4: General Registers

Name	Type	Description
$edi	unsigned	General purpose
$esi	unsigned	General purpose
$eax	unsigned	General purpose
$ebx	unsigned	General purpose
$ecx	unsigned	General purpose
$edx	unsigned	General purpose

Table 1-5: x87 Floating-Point Stack Registers

Name	Type	Description
$d0 - $d7	80-bit IEEE	Floating-point

Table 1-6: Segment Registers

Name	Type	Description
$gs	16-bit unsigned	Segment register
$fs	16-bit unsigned	Segment register
$es	16-bit unsigned	Segment register
$ds	16-bit unsigned	Segment register
$ss	16-bit unsigned	Segment register
$cs	16-bit unsigned	Segment register

Table 1-7: Special Purpose Registers

Name	Type	Description
$ebp	32-bit unsigned	Frame pointer
$efl	32-bit unsigned	Flags register
$eip	32-bit unsigned	Instruction pointer
$esp	32-bit unsigned	Privileged-mode stack pointer
$uesp	32-bit unsigned	User-mode stack pointer

AMD64/EM64T Register Symbols

This section describes the register symbols defined for AMD64/EM64T processors operating in 64-bit mode.

Table 1-8: General Registers

Name	Type	Description
$r8 - $r15	64-bit unsigned	General purpose
$rdi	64-bit unsigned	General purpose
$rsi	64-bit unsigned	General purpose
$rax	64-bit unsigned	General purpose
$rbx	64-bit unsigned	General purpose
$rcx	64-bit unsigned	General purpose
$rdx	64-bit unsigned	General purpose

Table 1-9: Floating-Point Registers

Name	Type	Description
$d0 - $d7	80-bit IEEE	Floating-point

Table 1-10: Segment Registers

Name	Type	Description
$gs	16-bit unsigned	Segment register
$fs	16-bit unsigned	Segment register
$es	16-bit unsigned	Segment register
$ds	16-bit unsigned	Segment register
$ss	16-bit unsigned	Segment register
$cs	16-bit unsigned	Segment register

Table 1-11: Special Purpose Registers

Name	Type	Description
$ebp	64-bit unsigned	Frame pointer
$rip	64-bit unsigned	Instruction pointer
$rsp	64-bit unsigned	Stack pointer
$eflags	64-bit unsigned	Flags register

Table 1-12: SSE Registers

Name	Type	Description
$mxcsr	64-bit unsigned	SIMD floating-point control
$xmm0 - $xmm15	Packed 4x32-bit IEEE Packed 2x64-bit IEEE	SSE floating-point registers

SSE Register Symbols

On AMD64/EM64T, Pentium III, and compatible processors, an additional set of SSE (Streaming SIMD Enhancements) registers and a SIMD floating-point control and status register are available.

Each SSE register may contain four IEEE 754 compliant 32-bit single-precision floating-point values. The PGDBG regs command reports these values individually in both hexadecimal and floating-point format. PGDBG provides syntax to refer to these values individually, as members of a range, or all together. There is no support for SSE2 or packed integers.

The component values of each SSE register can be accessed using the same syntax that is used for array subscripting. Pictorially, the SSE registers can be thought of as follows:

Bits: 127 96 95 65 63 32 31 0

$xmm0(3)	$xmm0(2)	$xmm0(1)	$xmm0(0)
$xmm1(3)	$xmm1(2)	$xmm1(1)	$xmm1(0)
$xmm7(3)	$xmm7(2)	$xmm7(1)	$xmm7(0)

To access a $xmm0(3), the 32-bit single-precision floating point value that occupies bits 96 – 127 of SSE register 0, use the following PGDBG command:

```
pgdbg> print $xmm0(3)
```

To set $xmm2(0) to the value of $xmm3(2), use the following PGDBG command:

```
pgdbg> set $xmm2(3) = $xmm3(2)
```

SSE registers can be subscripted with range expressions to specify runs of consecutive component values, and access an SSE register as a whole. For example, the following are legal PGDBG commands:

```
pgdbg> set $xmm0(0:1) = $xmm1(2:3)
pgdbg> set $xmm6 = 1.0/3.0
```

The first command above initializes elements 0 and 1 of $xmm0 to the values in elements 2 and 3 respectively in $xmm1. The second command above initializes all four elements of $xmm6 to the constant 1.0/3.0 evaluated as a 32-bit floating-point constant.

In most cases, PGDBG detects when the target environment supports the SSE registers. In the the event PGDBG does not allow access to SSE registers on a system that should have them, set the PGDBG_SSE environment variable to `on' to enable SSE support.

Debugging Fortran

Fortran Types

PGDBG displays Fortran type declarations using Fortran type names. The only exception is Fortran character types, which are treated as arrays of the C type char.

Arrays

Fortran array subscripts and ranges are accessed using the Fortran language syntax convention, denoting subscripts with parentheses and ranges with colons.

PGI compilers for the linux86-64 platform (AMD64 or Intel EM64T) support large arrays (arrays with an aggregate size greater than 2GB). Large array support is enabled by compiling with '–mcmodel=medium –Mlarge_arrays'. PGDBG provides full support for large arrays and large subscripts.

PGDBG supports arrays with non-default lower bounds. Access to such arrays uses the same subscripts that are used in the target application.

PGDBG also supports adjustable arrays. Access to adjustable arrays may use the same subscripting that is used in the target application.

Operators

In general, PGDBG uses C language style operators in expressions. The Fortran array index selector "()" and the Fortran field selector "%" for derived types are supported. However, .eq., .ne., and so forth are not supported. The analogous C operators ==, !=, etc. must be used instead. Note that the precedence of operators matches the C language, which may in some cases be different than for Fortran. See Table 1-2 for a complete list of operators and their definition.

Name of the Main Routine

If a PROGRAM statement is used, the name of the main routine is the name in the program statement. Otherwise, the name of the main routine is __unnamed_. A routine symbol named _MAIN_ is defined with start address equal to the start of the main routine. As a result,

```
break MAIN
```

can always be used to set a breakpoint at the start of the main routine.

Fortran Common Blocks

Each subprogram that defines a common block will have a local static variable symbol to define the common. The address of the variable will be the address of the common block. The type of the variable will be a locally defined structure type with fields defined for each element of the common block. The name of the variable will be the common block name, if the common block has a name, or _BLNK_ otherwise.

For each member of the common block, a local static variable is declared which represents the common block variable. Thus given declarations:

```
common /xyz/ integer a, real b
```

then the entire common block can be printed out using,

```
print xyz
```

Individual elements can be accessed by name. For example:,

```
print a, b
```

Nested Subroutines

To reference a nested subroutine qualify its name with the name of its enclosing routine using the scoping operator @.

For example:

```
subroutine subtest (ndim)
integer(4), intent(in) :: ndim
integer, dimension(ndim) :: ijk
call subsubtest ()
contains
 subroutine subsubtest ()
 integer :: I
 i=9
 ijk(1) = 1
 end subroutine subsubtest
 subroutine subsubtest2 ()
 ijk(1) = 1
 end subroutine subsubtest2
end subroutine subtest
program testscope
integer(4), parameter :: ndim = 4
call subtest (ndim)
end program testscope
```

```
pgdbg> break subtest@subsubtest
breakpoint set at: subsubtest line: 8 in "ex.f90" address: 0x80494091
pgdbg> names subtest@subsubtest
i = 0
pgdbg> decls subtest@subsubtest
arguments:
variables:
integer*4 i;
pgdbg> whereis subsubtest
function: "ex.f90"@subtest@subsubtest
```

Fortran 90 Modules

To access a member mm of a Fortran 90 module M, qualify mm with M using the scoping operator @. If the current scope is M, the qualification can be omitted.

For example:

```
module M
implicit none
real mm
contains
subroutine stub
print *,mm
end subroutine stub
end module M

program test
use M
implicit none
call stub()
print *,mm
end program test

pgdbg> Stopped at 0x80494e3, function MAIN, file M.f90,
line 13
#13: call stub()
pgdbg> which mm
"M.f90"@m@mm
pgdbg> print "M.f90"@m@mm
0
pgdbg> names m
mm = 0
stub = "M.f90"@m@stub
pgdbg> decls m
real*4 mm;
subroutine stub();
pgdbg> print m@mm
0
pgdbg> break stub
breakpoint set at: stub line:6 in "M.f90" address: 0x8049446 1
pgdbg> c
Stopped at 0x8049446, function stub, file M.f90, line 6
```

```
 #6: print *,mm
pgdbg> print mm
0
pgdbg>
```

Debugging C++

Calling C++ Instance Methods

To use the call command to call a C++ instance method, the object must be explicitly passed as the first parameter to the call. For example, given the following definition of class Person and the appropriate implementation of its methods:

```
class Person {
public:
char name[10];
Person(char * name);
void print();
};
main(){
Person * pierre;
pierre = new Person("Pierre");
pierre->print();
}
```

The instance method print on object Pierre is called as follows:

```
pgdbg> call Person::print(pierre)
```

Notice that pierre must be explicitly passed into the method (it is the this pointer), and the class name must also be specified.

Debugging with Core Files

PGDBG supports debugging of core files on the linux86 and linux86-64 platforms. To invoke PGDBG for core file debugging, use the following options:

$ pgdbg –core coreFileName programName

Core files are generated when a fatal exception occurs in an application. The shell environment in which the application runs must be set up to allow core file creation. On many systems, the default user ulimit does not allow core file creation. Check the ulimit as follows:

For sh/bash users:

```
$ ulimit -c
```

For csh/tcsh users:

```
% limit coredumpsize
```

If the core file size limit is zero or something too small for the application, it can be set to unlimited as follows:

For sh/bash users:

```
$ ulimit -c unlimited
```

For csh/tcsh users:

```
% limit coredumpsize unlimited
```

See the Linux shell documentation for more details. Some versions of Linux provide system-wide limits on core file creation.

Core files (or core dumps) are generated when a program encounters an exception or fault. For example, one common exception is the segmentation violation, which can be caused by referencing an invalid memory address. The memory and register states of the program are written into a core file so that they can be examined by a debugger.

The core file is normally written into the current directory of the faulting application. It is usually named core or core.pid where pid is the process ID of the faulting thread. If the shell environment is set correctly and a core file is not generated in the expected location, the system core dump policy may require configuration by a system administrator.

Different versions of Linux handle core dumping slightly differently. The state of all process threads are written to the core file in most modern implementations of Linux. In some new versions of Linux, if more than one thread faults, then each thread's state is written to separate core files using the core.pid file naming convention mentioned above. In older versions of Linux, only one faulting thread is written to the core file.

If a program uses dynamically shared objects (i.e., shared libraries named lib*.so), as most programs on Linux do, then accurate core file debugging requires that the program be debugged on the system where the core file was created. Otherwise, slight differences in the version of a shared library or the dynamic linker can cause erroneous information to be presented by the debugger. Sometimes a core file can be debugged successfully on a different system, particularly on more modern linux systems, but you should take care when attempting this.

PGDBG supports all non-control commands when debugging core files. It will perform any command that does not cause the program to run. Any command that causes the program to run will generate an error message in PGDBG. Depending on the type of core file created, PGDBG may provide the status of multiple threads. PGDBG does not support multi-process core file debugging.

Debugging Parallel Programs

This section gives an overview of how to use PGDBG to debug parallel applications. It provides some important definitions and background information on how PGDBG represents processes and threads.

Summary of Parallel Debugging Features

PGDBG is a parallel application debugger capable of debugging multi-process MPI applications, multi-thread and OpenMP applications, and hybrid multi-thread/multi-process applications that use MPI to communicate between multi-threaded or OpenMP processes. On Windows platforms, only OpenMP/ multi-thread debugging is supported.

OpenMP and Multi-thread Support

PGDBG provides full control of threads in parallel regions. Commands can be applied to all threads, a single thread, or a group of threads. Thread identification in PGDBG uses the native thread numbering scheme for OpenMP applications; for other types of multi-threaded applications thread numbering is arbitrary. OpenMP PRIVATE data can be accessed accurately for each thread. PGDBG provides understandable status displays regarding per-thread state and location.

Advanced features provide for configurable thread stop modes and wait modes, allowing debugger operation that is concurrent with application execution.

MPI and Multi-Process Support

PGDBG supports debugging of multi-process MPI applications, whether running on a single system or distributed on multiple systems. MPI applications can be started under debugger control using the mpirun command, or PGDBG can attach to a running, distributed MPI application. In either case all processes are automatically brought under debugger control. Process identification uses the MPI rank within COMMWORLD.

Graphical Presentation of Threads and Processes

PGDBG graphical user interface components that provide support for parallelism are described in detail in "PGDBG Graphical User Interface" on page 5.

Basic Process and Thread Naming

Because PGDBG can debug multi-threaded applications, multi-process applications, and hybrid multi-threaded/multi-process applications (only multi-thread on Windows platforms), it provides a convention for uniquely identifying each thread in each process. This section gives a brief overview of this naming convention and how it is used in order to provide adequate background for the subsequent sections. A more detailed discussion of this convention, including advanced techniques for applying it, is provided in "Thread and Process Grouping and Naming" on page 107.

PGDBG identifies threads in an OpenMP application using the OpenMP thread IDs. Otherwise, PGDBG assigns arbitrary IDs to threads, starting at zero and incrementing in order of thread creation.

PGDBG identifies processes in an MPI application using MPI rank (in communicator COMMWORLD). Otherwise, PGDBG assigns arbitrary IDs to processes; starting at zero and incrementing in order of process creation. Process IDs are unique across all active processes.

In a multi-threaded/multi-process application, each thread can be uniquely identified across all processes by prefixing its thread ID with the process ID of its parent process. For example, thread 1.4 identifies the thread with ID 4 in the process with ID 1.

An OpenMP application (single-process) logically runs as a collection of threads with a single process, process 0, as the parent process. In this context, a thread is uniquely identified by its thread ID. The process ID prefix is implicit and optional. See "Threads-only Debugging" on page 108.

An MPI program logically runs as a collection of processes, each made up of a single thread of execution. Thread 0 is implicit to each MPI process. A process ID uniquely identifies a particular process, and thread ID is implicit and optional. See "Process-only Debugging" on page 109.

A hybrid, or multilevel, MPI/OpenMP program requires the use of both process and thread IDs to uniquely identify a particular thread. See "Multilevel Debugging" on page 109.

A serial program runs as a single thread of execution, thread 0, belonging to a single process, process 0. The use of thread IDs and process IDs is unnecessary but optional.

Multi-Thread and OpenMP Debugging

PGDBG automatically attaches to new threads as they are created during program execution. PGDBG reports when a new thread is created and the thread ID of the new thread is printed.

```
([1] New Thread)
```

The system ID of the freshly created thread is available through the threads command. The procs command can be used to display information about the parent process.

PGDBG maintains a conceptual current thread. The current thread is chosen by using the thread command when the debugger is operating in text mode (invoked with the -text option), or by clicking in the thread grid when the GUI interface is in use (the default). A subset of PGDBG commands known as thread-level commands, when executed, apply only to the current thread. See "Thread Level Commands" on page 120, for more information.

The threads command lists all threads currently employed by an active program. The threads command displays each thread's unique thread ID, system ID (Linux process ID), execution state (running, stopped, signaled, exited, or killed), signal information and reason for stopping, and the current location (if stopped or signaled). An arrow (=>) indicates the current thread. The process ID of the parent is printed in the top left corner. The thread command changes the current thread.

```
pgdbg [all] 2> thread 3
pgdbg [all] 3> threads
0 ID PID STATE SIGNAL LOCATION
=> 3 18399 Stopped SIGTRAP main line: 31 in "omp.c" address:
0x80490ab
 2 18398 Stopped SIGTRAP main line: 32 in "omp.c" address: 0x80490cf
 1 18397 Stopped SIGTRAP main line: 31 in "omp.c" address: 0x80490ab
 0 18395 Stopped SIGTRAP f line: 5 in "omp.c" address: 0x8048fa0
```

In the GUI, thread state is represented by a color in the process/thread grid.

Table 1-13: Thread State Is Described Using Color

Thread State	Description	Color
Stopped	If all threads are stopped at break-points, or where directed to stop by PGDBG	Red
Signaled	If at least one thread is stopped due to delivery of a signal	Blue
Running	If at least one thread is running	Green
Exited or Killed	If all threads have been killed or exited	Black

Multi-Process MPI Debugging

When installed as part of the PGI Cluster Development Kit (CDK) on Linux platforms, PGDBG supports multi-process MPI debugging. The PGI CDK contains versions of MPICH and MPICH2 pre-configured to support debugging cluster applications with PGDBG. Non-CDK MPI software must be configured to support PGDBG; see http://www.pgroup.com/support/faq.htm for more information.

Invoking PGDBG for MPI Debugging

The command used to start MPI debugging under MPICH using the PGDBG GUI is:

```
% mpirun -np nprocs -dbg=pgdbg executable [ arg1,...argn ]
```

The command used to start MPI debugging under MPICH2 using the PGDBG GUI is:

```
% mpiexec -np nprocs -pgi executable [ arg1,...argn ]
```

Note that for MPICH2, the mpdboot command must have been run, as with any other MPICH2 application.

The command used to start MPI debugging using PGDBG in TEXT mode is the same, except that the DISPLAY environment variable must be undefined in the shell that is invoking mpirun:

For sh/bash users:

```
$ unset DISPLAY
```

For csh/tcsh users:

```
% unsetenv DISPLAY
```

In either case, PGDBG must be installed and found in the PATH.

When an MPI debug session begins, PGDBG will stop the program at the first executable statement in the program. Execution does not need to be started using the run command as it does with serial or multi-threaded programs. Execution is started using one of the other control commands, such as cont, next, or step.

Using PGDBG for MPI Debugging

The initial MPI process is run locally; 'local' describes the host on which PGDBG is running. PGDBG automatically attaches to new MPI processes as they are created by the running MPI application. PGDBG displays an informational message as it attaches to the freshly created processes.

```
([1] New Process)
```

The MPI global rank is printed with the message. The procs command can be used to list the host and the PID of each process by rank. The current process is indicated by an arrow (=>). The proc command can be used to change the current process by process ID.

```
pgdbg [all] 0.0> proc 1; procs
Process 1: Thread 0 Stopped at 0x804a0e2, function main, file mpi.c,
line 30
 #30: aft=time(&aft);
 ID IPID STATE THREADS HOST
 0 24765 Stopped 1 local
 => 1 17890 Stopped 1 red2.wil.st.com
```

The execution state of a process is described in terms of the execution state of its component threads. See Table 1-13 for a description of how thread state is represented in the GUI.

The PGDBG command prompt displays the current process and the current thread. In the above example, the current process was changed to process 1 by the proc 1 command and the current thread of process 1 is 0; this is written as 1.0:

```
 pgdbg [all] 1.0>
```

See "Process and Thread Control" on page 122, for a complete description of the prompt format.

The following rules apply during a PGDBG debug session:

- PGDBG maintains a conceptual current process and current thread.
- Each active process has a thread set of size >=1.
- The current thread is a member of the thread set of the current process.

Certain commands, when executed, apply only to the current process or the current thread. See "Process Level Commands" on page 119, and "Thread Level Commands" on page 120, for more information.

A license file distributed with PGDBG restricts PGDBG to debugging a total of 64 threads. PGI Workstation, Server, and CDK (Cluster Development Kit) license files may further restrict the number of threads that PGDBG is eligible to debug. PGDBG will use the PGI Workstation, Server, or CDK license files to determine the number of threads it is allowed to debug.

With its 64-thread limit, PGDBG is capable of debugging a 16-node cluster with 4 CPUs on each node or a 32-node cluster with 2 CPUs on each node or any combination of threads that add up to 64.

MPICH Support

PGDBG supports redirecting stdin, stdout, and stderr with the following MPICH switches:

Table 1-14: MPICH Support

Command	Output
`-stdout <file>`	Redirect standard output to <file>
`-stdin <file>`	Redirect standard input from <file>
`-stderr <file>`	Redirect standard error to <file>

PGDBG also provides support for the following MPICH switches:

Command	Output
`-nolocal`	PGDBG runs locally, but no MPI processes run locally
`-all-local`	PGDBG runs locally, all MPI processes run locally

For information about how to configure an arbitrary installation of MPICH to use PGDBG, see the PGDBG online FAQ at http://www.pgroup.com/support/faq.htm.

When PGDBG is invoked via mpirun the following PGDBG command line arguments are not accessible. A workaround is listed for each.

Argument	Workaround
`-dbx`	Include 'pgienv dbx on' in .pgdbgrc file
`-s startup`	Use .pgdbgrc default script file and the script command
`-c "command"`	Use .pgdbgrc default script file and the script command

Argument	Workaround
`-text`	Clear your DISPLAY environment variable before invoking mpirun
`-t <target>`	Add to the beginning of the PATH environment variable a path to the appropriate PGDBG

LAM-MPI Support

The Portland Group Cluster Development Kit (CDK) includes an implementation of MPICH. PGDBG is configured to automatically integrate with MPICH. PGDBG can also be used with LAM-MPI, but some configuration is required. For more information, see the online FAQ at http://www.pgroup.com/support/faq.htm.

Thread and Process Grouping and Naming

This section describes how to name a single thread, how to group threads and processes into sets, and how to apply PGDBG commands to groups of processes and threads.

PGDBG Debug Modes

PGDBG can operate in four debug modes. The mode determines a short form for uniquely naming threads and processes. The debug mode is set automatically or by the pgienv command.

Table 1-15: PGDBG Debug Modes

Debug Mode	Program Characterization
Serial	A single thread of execution
Threads-only	A single process, multiple threads of execution
Process-only	Multiple processes, each process made up of a single thread of execution [Linux Only]
Multilevel	Multiple processes, at least one process employing multiple threads of execution [Linux Only]

PGDBG initially operates in serial mode reflecting a single thread of execution. Thread IDs can be ignored in serial debug mode since there is only a single thread of execution.

The PGDBG prompt displays the ID of the current thread according to the current debug mode. See "The PGDBG Command Prompt" on page 128, for a description of the PGDBG prompt.

The pgienv command is used to change debug modes manually.

```
pgienv mode [serial|thread|process|multilevel]
```

The debug mode can be changed at any time during a debug session.

Threads-only Debugging

Enter threads-only mode to debug a program with a single multi-threaded process. As a convenience the process ID portion can be omitted. PGDBG automatically enters threads-only debug mode from serial debug mode when it detects and attaches to new threads.

Example 1-1: Thread IDs in Threads-only Debug Mode

`1`	Thread 1 of process 0 (*.1)
`*`	All threads of process 0 (*. *)

`0.7`	Thread 7 of process 0 (multilevel names are valid in threads-only mode)

In threads-only debug mode, status and error messages are prefixed with thread IDs depending on context.

Process-only Debugging

[Linux Only] Enter process-only mode to debug an application consisting of single-threaded processes. As a convenience, the thread ID portion can be omitted. PGDBG automatically enters process-only debug mode from serial debug mode when the target program returns from MPI_Init.

Example 1-2: Process IDs in process-only debug mode

`0`	All threads of process 0 (0.*)
`*`	All threads of all processes (*.*)
`1.0`	Thread 0 of process 1 (multilevel names are valid in process-only mode)

In process-only debug mode, status and error messages are prefixed with process IDs depending on context.

Multilevel Debugging

[Linux Only] The name of a thread in multilevel debug mode is the thread ID prefixed with its parent process ID. This forms a unique name for each thread across all processes. This naming scheme is valid in all debug modes. PGDBG changes automatically to multilevel debug mode from process-only debug mode or threads-only debug mode when at least one MPI process creates multiple threads.

Example 1-3: Thread IDs in multilevel debug mode

`0.1`	Thread 1 of process 0
`0.*`	All threads of process 0

*	All threads of all processes

In multilevel debug, mode status and error messages are prefixed with process/thread IDs depending on context.

Process/Thread Sets

A process/thread set (p/t-set) is used to restrict a debugger command to apply to just a particular set of threads. A p/t-set is a set of threads drawn from all threads of all processes in the target program. Use p/t-set notation (described in "p/t-set Notation" on page 110) to define a p/t-set.

In the following sections, frequent reference is made to three named p/t-sets:

- The target p/t-set is the set of processes and threads to which a debugger command is applied. The target p/t-set is initially defined by the debugger to be the set [all] which describes all threads of all processes.
- A prefix p/t-set is defined when p/t-set notation is used to prefix a debugger command. For the prefixed command, the target p/t-set is the prefix p/t-set.
- The current p/t-set is the p/t set currently set in the PGDBG environment. The current p/t-set can be defined using the focus command. The current p/t set is used as the target p/t-set unless a prefix p/t-set overrides it.

p/t-set Notation

The following set of rules describes how to use and construct p/t-sets:

Use a prefix p/t-set with a simple command:

```
 [p/t-set prefix] command parm0,
parm1, ...
```

Use a prefix p/t-set with a compound command:

```
 [p/t-set prefix] simple-command [;
simple-command ...]
```

p/t-id:

```
 {integer|*}.{integer|*}
```

`p/t-id` optional notation when process-only or threads-only debugging is in effect (see the pgienv command):

```
{integer|*}
```

p/t-range:

```
p/t-id:p/t-id
```

p/t-list:

```
{p/t-id|p/t-range} [, {p/t-id|p/t-range} ...]
```

p/t-set:

```
[[!]{p/t-list|set-name}]
```

Example 1-4: p/t-sets in Threads-only Debug Mode

`[0,4:6]`	Threads 0,4,5, and 6
`[*]`	All threads
`[*.1]`	Thread 1. Multilevel notation is valid in threads-only mode
`[*.*]`	All threads

Example 1-5: p/t-sets in Process-only Debug Mode

`[0,2:3]`	Processes 0, 2, and 3 (equivalent to [0.*,2:3.*])
`[*]`	All processes (equivalent to [*.*])
`[0]`	Process 0 (equivalent to [0.*])
`[*.0]`	Process 0. Multilevel syntax is valid in process-only mode.

`[0:2.*]`	Processes 0, 1, and 2. Multilevel syntax is valid in process-only debug mode.

Example 1-6: p/t-sets in Multilevel Debug Mode

`[0.1,0.3,0.5]`	Thread 1,3, and 5 of process 0
`[0.*]`	All threads of process 0
	Thread 1,2, and 3 of process 1
[1:2.1]	Thread 1 of processes 1 and 2
[clients]	All threads defined by named set clients
[1]	Incomplete; invalid in multilevel debug mode

Dynamic vs. Static p/t-sets

The defset command can be used to define both dynamic and static p/t-sets. The members of a dynamic p/t-set are those active threads described by the p/t-set at the time that p/t-set is used. A p/t-set is dynamic by default. Threads and processes are created and destroyed as the target program runs and, therefore, membership in a dynamic set varies as the target program executes.

Example 1-7: Defining a Dynamic p/t-set

defset clients [*.1:3]	Defines a named set clients whose members are threads 1, 2, and 3 of all processes that are currently active when clients is used. Membership in clients changes as processes are created and destroyed.

The members of a static p/t-set are those threads described by the p/t-set at the time that p/t-set is defined. Use a ! to specify a static set. Membership in a static set is fixed at definition time.

Example 1-8: Defining a Static p/t-set

`defset clients [!*.1:3]`	Defines a named set clients whose members are threads 1, 2, and 3 of those processes that are currently active at the time of the definition.

p/t-sets defined with defset are not mode dependent and are valid in any debug mode.

Current vs. Prefix p/t-set

The current p/t-set is set by the focus command. The current p/t-set is described by the debugger prompt (depending on debug mode; see "The PGDBG Command Prompt" on page 128, for a description of the PGDBG prompt). A p/t-set can be used to prefix a command to override the current p/t-set. The prefix p/t-set becomes the target p/t-set for the command. The target p/t-set defines the set of threads that will be affected by a command.

- In the following command line, the target p/t-set is the current p/t-set:

```
pgdbg [all] 0.0> cont
Continue all threads in all processes
```

- In contrast, a prefix p/t-set is used in the following command so the the target p/t-set is the prefix p/t-set (note the prefix p/t-set in bold:

```
pgdbg [all] 0.0> [0.1:2] cont
Continue threads 1 and 2 of process 0 only
```

In both of the above examples, the current p/t-set is the debugger-defined set [all]. In the first case, [all] is the target p/t-set. In the second case, the prefix p/t-set overrides [all] and becomes the target p/t-set. The continue command is applied to all active threads in the target p/t-set. Using a prefix p/t-set does not change the current p/t-set.

p/t-set Commands

The following commands can be used to collect threads into logical groups.

- defset and undefset can be used to manage a list of named p/t-sets.
- focus is used to set the current p/t-set.

- viewset is used to view the active members described by a particular p/t-set, or to list all the defined p/t-sets.
- whichsets is used to describe the p/t-sets to which a particular process/thread belongs.

Table 1-16: p/t-set Commands

Command	Description
`defset`	Define a named p/t-set. This set can later be referred to by name. A list of named sets is stored by PGDBG.
`focus`	Set the target process/thread set for commands. Subsequent commands will be applied to the members of this set by default.
`undefset`	Undefine a previously defined process/thread set. The set is removed from the list. The debugger-defined p/t-set [all] cannot be removed.
`viewset`	List the members of a process/thread set that currently exist as active threads, or list all the defined p/t-sets.
`whichsets`	List all defined p/t-sets to which the members of a process/thread set belongs.

Examples of the p/t-set commands in the previous table follow.

Use defset to define the p/t-set initial is to contain only thread 0:

```
pgdbg [all] 0> defset initial [0]
"initial" [0] : [0]
```

Change the current p/t-set to initial using the focus command:

```
pgdbg [all] 0> focus [initial]
[initial] : [0]
[0]
```

Advance the thread. Because the code is not using a prefix p/t-set, the target p/t-set is the current p/t-set, which is initial:

```
pgdbg [initial] 0> next
```

The whichsets command shows that thread 0 is a member of two defined p/t-sets:

```
pgdbg [initial] 0> whichsets [initial]
Thread 0 belongs to:
all
initial
```

The viewset command displays all threads that are active and are members of defined p/t-sets:

```
pgdbg [initial] 0> viewset
"all" [*.*] : [0.0,0.1,0.2,0.3]
"initial" [0] : [0]
```

The focus command can be used to set the current p/t-set back to [all]:

```
pgdbg [initial] 0> focus [all]
[all] : [0.0,0.1,0.2,0.3]
[*.*]
```

The undefset command undefines the initial p/t-set:

```
pgdbg [all] 0> undefset initial
p/t-set name "initial" deleted.
```

The examples above illustrate how to manage named p/t-sets using the command-line interface. A similar capability is available in the PGDBG GUI. "Focus Panel" on page 9, contains information about the Focus Panel. The Focus Panel, shown in Figure 1-3, contains a table labeled Focus with two columns: a Name column and a p/t-set column. The entries in this table are p/t-sets exactly like the p/t-sets used in the command line interface.

To create a p/t set in the Focus Panel , left-click the Add button. This opens a dialog box similar to the one in Table 1-12. Enter the name of the p/t-set in the Focus Name text field and the p/t-set in the p/t-set text field. Click the left mouse button on the OK button to add the p/t-set. The new p/t-set will appear in the Focus Table. Clicking the Cancel button or closing the dialog box will abort the operation. The Clear button will clear the Focus Name and p/t-set text fields

To select a p/t-set, click the left mouse button on the desired p/t-set in the table. The selected p/t-set is also known as the Current Focus. PGDBG will apply all commands entered in the Source Panel to the Current Focus when you choose Focus in the Apply Selector ("Source Panel Combo Boxes" on page 20). Current Focus can also be used in a GUI subwindow. Choose Current Focus in a subwindow's Context Selector ("Subwindows" on page 24) to display data for the Current Focus only.

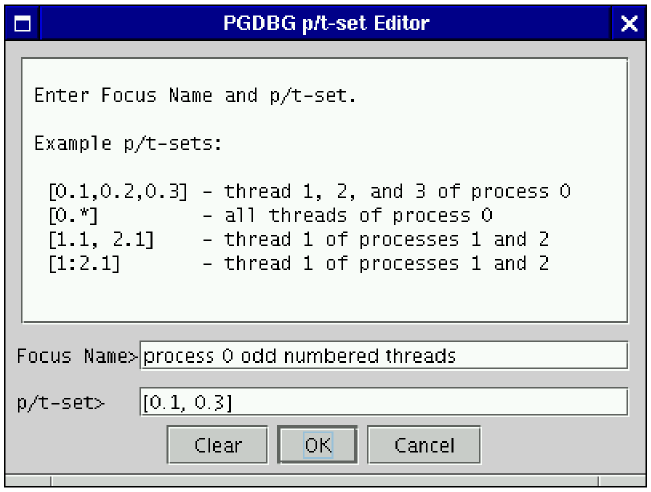
To modify an existing p/t-set, select the desired group in the Focus Table and left-click the Modify button. A dialog box similar to that in Figure 1-12 will appear, except that the Focus Name and p/t-set text fields will contain the selected group's name and p/t-set respectively. You can edit the information in these text fields and click OK to save the changes.

To remove an existing p/t-set, select the desired item in the Focus Table and left-click the Remove button. PGDBG will display a dialog box asking for confirmation of the request for removal of the selected p/t-set. Left-click the Yes button to confirm or the No button to cancel the operation.

It should be noted that p/t-sets defined in the Focus Panel of the PGDBG GUI are only used by the Apply and Context Selectors in the GUI. They do not affect focus in the Command Prompt Panel. Conversely, focus changes made in the Command Prompt Panel affect only the Command Prompt Panel and not the rest of the PGDBG GUI.

For example, in Figure 1-12 there is a p/t-set named "process 0 odd numbered threads". The p/t-set is [0.1, 0.3] which indicates threads 1 and 3 in process 0. Table 1-13 shows this p/t-set in the Focus Table. We also chose Focus in the Apply Selector. Any command issued in the Source Panel is applied to the Current Focus, or thread 1 and 3 on process 0 only. All other threads will remain idle until either the All p/t-set is selected in the Focus Panel or All is selected in the Apply Selector. Note that "process 0 odd numbered threads" is not available in the Command Prompt Panel.

Figure 1-12: Focus Group Dialog Box

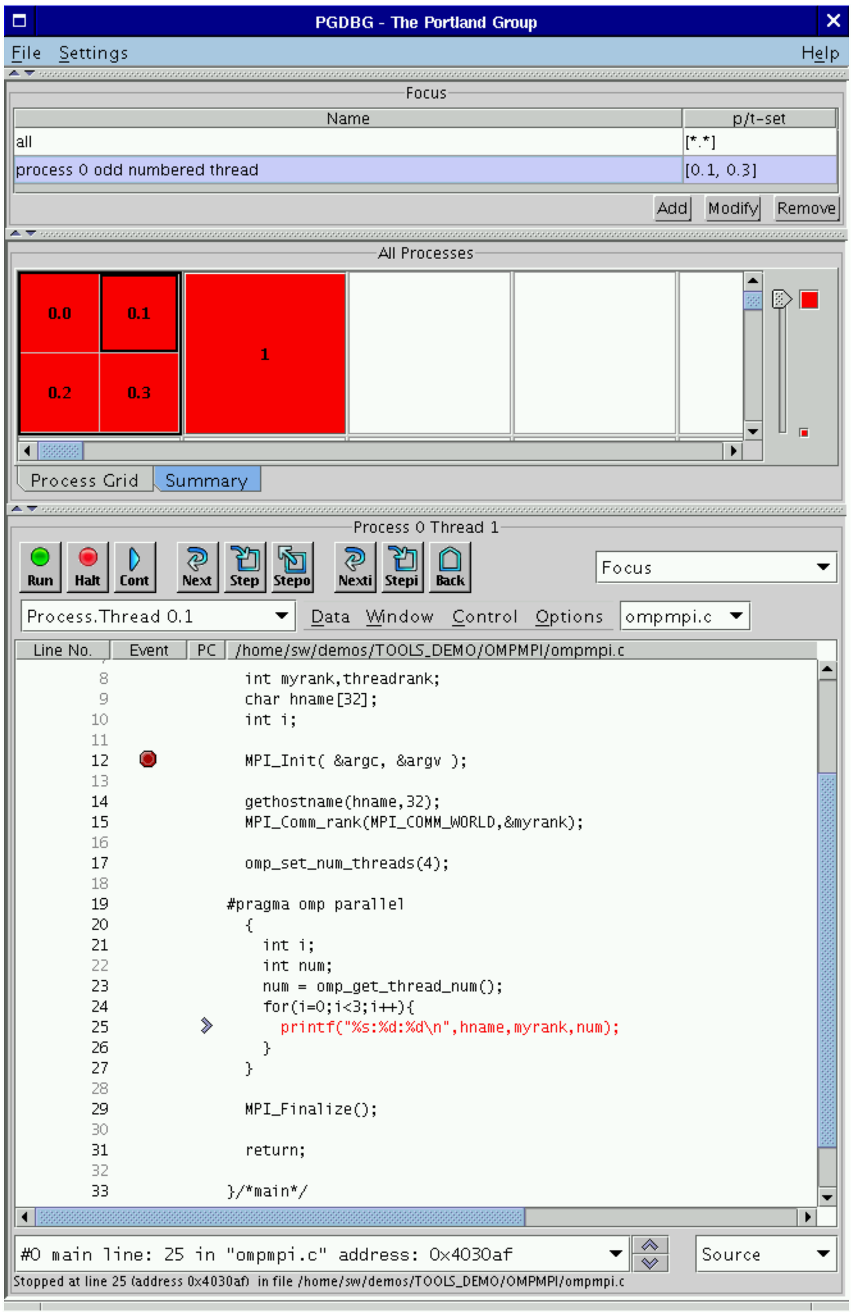


```
Enter Focus Name and p/t-set.

Example p/t-sets:

 [0.1,0.2,0.3] - thread 1, 2, and 3 of process 0
 [0.*]         - all threads of process 0
 [1.1, 2.1]    - thread 1 of processes 1 and 2
 [1:2.1]       - thread 1 of processes 1 and 2
```

Focus Name> process 0 odd numbered threads

p/t-set> [0.1, 0.3]

Figure 1-13: Focus in the GUI



Name	p/t-set
all	[*.*]
process 0 odd numbered thread	[0.1, 0.3]

Process.Thread 0.1

Line No. | Event | PC | /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c

```
    int myrank,threadrank;
    char hname[32];
    int i;

    MPI_Init( &argc, &argv );

    gethostname(hname,32);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    omp_set_num_threads(4);

  #pragma omp parallel
    {
      int i;
      int num;
      num = omp_get_thread_num();
      for(i=0;i<3;i++){
        printf("%s:%d:%d\n",hname,myrank,num);
      }
    }

    MPI_Finalize();

    return;

  }/*main*/
```

```
#0 main line: 25 in "ompmpi.c" address: 0x4030af
Stopped at line 25 (address 0x4030af) in file /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c
```

Command Set

For the purpose of parallel debugging, the PGDBG command set is divided into three disjoint subsets according to how each command reacts to the current p/t-set. Process level and thread level commands can be parallelized. Global commands cannot be parallelized.

Table 1-17: PGDBG Parallel Commands

Commands	Action
Process Level Commands	Parallel by current p/t-set or prefix p/t-set [Linux Only]
Thread Level Commands	Parallel by prefix p/t-set. Ignores current p/t-set
Global Commands	Non-parallel commands

Process Level Commands

The process level commands are the PGDBG control commands.

The PGDBG control commands apply to the active members of the current p/t-set by default. A prefix set can be used to override the current p/t-set. The target p/t-set is the prefix p/t-set if present.

```
cont        nexti       stepout      synci
halt        step        sync         wait
next        stepi
```

Apply the next command to threads 1 and 2 of process 0:

```
pgdbg [all] 0.0> focus [0.1:2]
pgdbg [0.1:2] 0.0> next
```

Apply the next command to thread 3 of process 0 using a prefix p/t-set:

```
pgdbg [all] 0.0> [0.3] n
```

Thread Level Commands

The following commands are not concerned with the current p/t-set. When no p/t-set prefix is used, these commands execute in the context of the current thread of the current process by default. That is, thread level commands ignore the current p/t-set. Thread level commands can be applied to multiple threads by using a prefix p/t-set. When a prefix p/t-set is used, the commands in this section are executed in the context of each active thread described by the prefix p/t-set. The target p/t-set is the prefix p/t-set if present, or the current thread (not the current p/t-set) if no prefix p/t set exists. The thread level commands are:

`addr`	`dump`	`noprint`	`sp`
`ascii`	`entry`	`oct`	`sread`
`assign`	`fp`	`pc`	`stack`
`bin`	`fread`	`pf`	`stackdump`
`break*`	`func`	`print`	`string`
`cread`	`hex`	`regs`	`watch`
`dec`	`hwatch`	`retaddr`	`watchi`
`decl`	`iread`	`rval`	`whatis`
`disasm`	`line`	`scope`	`where`
`do`	`lines`	`set`	`track`
`doi`	`lval`	`sizeof`	`tracki`
`dread`			

* breakpoints and variants: (stop, stopi, break, breaki) if no prefix p/t-set is specified, [all] is used (overriding current p/t-set).

The following occurs when a prefix p/t-set is used:

- The threads described by the prefix are sorted per process by thread ID in increasing order.
- The processes are sorted by process ID in increasing order, and duplicates are removed.
- The command is then applied to the threads in the resulting list in order.

Without a prefix p/t-set, the print command executes in the context of the current thread of the current process, thread 0.0, printing rank 0:

```
pgdbg [all] 0.0> print myrank
0
```

With a prefix p/t-set, the thread members of the prefix are sorted and duplicates are removed. The print command iterates over the resulting list:

```
pgdbg [all] 0.0> [2:3.*,1:2.*] print
myrank
[1.0] print myrank:
1
[2.0] print myrank:
2
[2.1] print myrank:
2
[2.2] print myrank:
2
[3.0] print myrank:
3
[3.2] print myrank:
3
[3.1] print myrank:
3
```

Global Commands

The rest of the PGDBG commands ignore threads and processes, or are defined globally for all threads across all processes. The current p/t-set and prefix p/t-set (if any) are ignored.

The following is a list of commands that are defined globally.

?	disable	pgienv	status
/	display	proc	thread
alias	edit	procs	threads
arrive	enable	pwd	unalias
breaks	files	quit	unbreak
call	focus	repeat	undefset

`fatch`	`funcs`	`rerun`	`use`
`cd`	`help`	`run`	`viewset`
`debug`	`history`	`script`	`wait`
`defset`	`ignore`	`shell`	`whereis`
`delete`	`log`	`source`	`whichsets`
`directory`			

Process and Thread Control

PGDBG supports thread and process control (e.g. step, next, cont ...) everywhere in the program. Threads and processes can be advanced in groups anywhere in the program. Recall that multi-process MPI debugging is supported only on Linux platforms.

The PGDBG control commands are:

`cont`	`nexti`	`stepout`	`synci`
`halt`	`step`	`sync`	`wait`
`next`	`stepi`		

To describe those threads to be advanced, set the current p/t-set or use a prefix p/t-set.

A thread inherits the control operation of the current thread when it is created. If the current thread single-steps over an _mp_init call (found at the beginning of every OpenMP parallel region) using the next command, then all threads created by _mp_init will step into the parallel region as if by the next command.

A process inherits the control operation of the current process when it is created. So if the current process returns from a call to MPI_Init under the control of a cont command, the new process will do the same.

Configurable Stop Mode

PGDBG supports configuration of how threads and processes stop in relation to one another. PGDBG defines two new pgienv environment variables, threadstop and procstop, for this purpose. PGDBG defines two stop modes, synchronous (sync) and asynchronous (async).

Table 1-18: PGDBG Stop Modes

Command	Result
`sync`	Synchronous stop mode; when one thread stops at a breakpoing (event), all other threads are stopped soon after.
`async`	Asynchronous stop mode; each thread runs independently of the other threads. One thread stopping does not affect the behavior of another.

Thread stop mode is set using the pgienv command as follows:

```
pgienv threadstop [sync|async]
```

Process stop mode is set using the pgienv command as follows:

```
pgienv procstop [sync|async]
```

PGDBG defines the default to be asynchronous for both thread and process stop modes. When debugging an OpenMP program, PGDBG automatically enters synchronous thread stop mode in serial regions, and asynchronous thread stop mode in parallel regions.

The pgienv environment variables threadstopconfig and procstopconfig can be set to automatic (auto) or user defined (user) to enable or disable this behavior:

```
pgienv threadstopconfig [auto|user]
pgienv procstopconfig [auto|user]
```

Selecting the user-defined stop mode prevents the debugger from changing stop modes automatically. Automatic stop configuration is the default for both threads and processes.

Configurable Wait Mode

Wait mode describes when PGDBG will accept the next command. The wait mode is defined in terms of the execution state of the program. Wait mode describes to the debugger which threads/processes must be stopped before it will accept the next command. In certain situations, it is desirable to be able to enter commands while the program is running and not stopped at an event. The PGDBG prompt will not appear until all processes/threads are stopped. However, a prompt may be available before all processes/

threads have stopped. Pressing <enter> at the command line will bring up a prompt if it is available. The availability of the prompt is determined by the current wait mode and any pending wait commands (described below).

PGDBG accepts a compound statement at each prompt. Each compound statement is a sequence of semicolon-separated commands, which are processed immediately in order. The wait mode describes when to accept the next compound statement. PGDBG supports three wait modes, which can be applied to processes and/or threads.

Table 1-19: PGDBG Wait Modes

Command	Result
`all`	The prompt is available only after all threads have stopped since the last control command.
`any`	The prompt is available only after at least one thread has stopped since the last control command.
`none`	The prompt is available immediately after a control command is issued.

- Thread wait mode describes which threads PGDBG waits for before accepting new commands.
- Process wait mode describes which processes PGDBG waits for before accepting new commands.

Thread wait mode is set using the pgienv command as follows:

```
pgienv threadwait [any|all|none]
```

Process wait mode is set using the pgienv command as follows:

```
pgienv procwait [any|all|none]
```

If process wait mode is set to none, then thread wait mode is ignored.

In TEXT mode, PGDBG defaults to:

```
threadwait all
procwait any
```

If the target program goes MPI parallel, then procwait is changed to none automatically by PGDBG.

If the target program goes thread parallel, then threadwait is changed to none automatically by PGDBG. The pgienv environment variable threadwaitconfig can be set to automatic (auto) or user defined (user) to enable or disable this behavior.

```
pgienv threadstopconfig [auto|user]
```

Selecting the user defined wait mode prevents the debugger from changing wait modes automatically. Automatic wait mode is the default thread wait mode.

PGDBG defaults to the following in GUI mode:

```
threadwait none
procwait none
```

Setting the wait mode may be necessary when invoking the debugger using the -s (script file) option in GUI mode (to ensure that the necessary threads are stopped before the next command is processed).

PGDBG also provides a wait command that can be used to insert explicit wait points in a command stream. Wait uses the target p/t-set by default, which can be set to wait for any combination of processes/threads. The wait command can be used to insert wait points between the commands of a compound command.

The threadwait and procwait pgienv variables can be used to configure the behavior of wait (see pgienv usage in "Configurable Wait Mode" on page 123).

The following table describes the behavior of wait. In the example in the table:

- S is the target p/t-set
- P is the set of all processes described by S and p is a single process
- T is the set of all threads described by S and t is a single thread

Table 1-20: PGDBG Wait Behavior

Command	threadwait	procwait	Wait Set
wait	all	all	Wait for T
wait	all	any none	Wait for all threads in at least one p in P
wait	any none	all	Wait for T
wait	any none	any none	Wait for all t in T for at least one p in P
wait all	all	all	Wait for T
wait all	all	any none	Wait for all threads of at least one p in P
wait all	any none	all	Wait for T
wait all	any none	any none	Wait for all t in T for at least one p in P
wait any	all	all	Wait for at least one thread for each process p in P
wait any	all	any none	Wait for at least one t in T
wait any	any none	all	Wait for at least one thread in T for each process p in P
wait any	any none	any none	Wait for at least one t in T

Command	threadwait	procwait	Wait Set
wait none	all any none	all any none	Wait for no threads

Status Messages

PGDBG can produce a variety of status messages during a debug session. This feature can be useful in text mode in the absence of the graphical aids provided by the GUI. Use the pgienv command to enable or disable the types of status messages produced by setting the verbose environment variable to an integer-valued bit mask using pgienv:

```
pgienv verbose <bitmask>
```

The values for the bit mask listed in the following table control the type of status messages desired.

Table 1-21: PGDBG Status Messages

Value	Type	Information
0x1	Standard	Report status information on current process/thread only. A message is printed when the current thread stops and when threads and processes are created and destroyed. Standard messaging is the default and cannot be disabled.
0x2	Thread	Report status information on all threads of current processes. A message is reported each time a thread stops. If process messaging is also enabled, then a message is reported for each thread across all processes. Otherwise, messages are reported for threads of the current process only.
0x4	Process	Report status information on all processes. A message is reported each time a process stops. If thread messaging is also enabled, then a message is reported for each thread across all processes. Otherwise, messages are reported for the current thread only of each process.
0x8	SMP	Report SMP events. A message is printed when a process enters or exits a parallel region, or when the threads synchronize. The PGDBG OpenMP handler must be enabled.
0x16	Parallel	Report process-parallel events (default).
0x32	Symbolic debug information	Report any errors encountered while processing symbolic debug information (e.g. ELF, DWARF2).

The PGDBG Command Prompt

The PGDBG command prompt reflects the current debug mode (see "PGDBG Debug Modes" on page 107).

In serial debug mode, the PGDBG prompt looks like this:

```
pgdbg>
```

In threads-only debug mode, PGDBG displays the current p/t-set in square brackets followed by the ID of the current thread:

```
pgdbg [all] 0>
Current thread is 0
```

In process-only debug mode, PGDBG displays the current p/t-set in square brackets followed by the ID of the current process:

```
pgdbg [all] 0>
Current process is 0
```

In multilevel debug mode, PGDBG displays the current p/t-set in square brackets followed by the ID of the current thread prefixed by the id of its parent process:

```
pgdbg [all] 1.0>
Current thread 1.0
```

The pgienv promptlen variable can be set to control the number of characters devoted to printing the current p/t-set at the prompt.

Parallel Events

This section describes how to use a p/t-set to define an event across multiple threads and processes. Events, such as breakpoints and watchpoints, are user-defined events. User-defined events are thread-level commands (see "Thread Level Commands" on page 120, for details).

Breakpoints, by default, are set across all threads of all processes. A prefix p/t-set can be used to set breakpoints on specific processes and threads. For example:

```
i) pgdbg [all] 0> b 15
ii) pgdbg [all] 0> [all] b 15
iii) pgdbg [all] 0> [0.1:3] b
15
```

(i) and (ii) are equivalent. (iii) sets a breakpoint only in threads 1,2,3 of process 0.

By default, all other user events are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads. For example:

```
i) pgdbg [all] 0> watch glob
ii) pgdbg [all] 0> [*] watch
glob
```

(i) sets a data breakpoint for glob on thread 0 only. (ii) sets a watchpoint for glob on all threads that are currently active.

When a process or thread is created, it inherits all of the breakpoints defined for the parent process or thread. All other events must be defined explicitly after the process or thread is created. All processes must be stopped to add, enable, or disable a user event.

Events may contain if and do clauses. For example:

```
pgdbg [all] 0> [*] break
func if (glob!=0) do {set f = 0}
```

The breakpoint will fire only if glob is non-zero. The do clause is executed if the breakpoint fires. The if and do clauses execute in the context of a single thread. The conditional in the if clause and the body of the do execute in the context of a single thread, the thread that triggered the event. The conditional definition as above can be restated as follows:

```
[0] if (glob!=0) {[0] set
f = 0}
[1] if (glob!=0) {[1] set
f = 0}
...
```

When thread 1 hits func, glob is evaluated in the context of thread 1. If glob evaluates to non-zero, f is bound in the context of thread 1 and its value is set to 0.

Control commands can be used in do clauses, however they only apply to the current thread and are only well defined as the last command in the do clause. For example:

```
pgdbg [all] 0> [*] break
func if (glob!=0) do {set f = 0; c}
```

If the wait command appears in a do clause, the current thread is added to the wait set of the current process. For example:

```
pgdbg [all] 0> [*] break
func if (glob!=0) do {set f = 0; c; wait}
```

if conditionals and do bodies cannot be parallelized with prefix p/t-sets. For example, an illegal command would be:

```
pgdbg [all] 0> break func if (glob!=0)
do {[*] set f = 0} ILLEGAL
```

This is illegal. The body of a do statement cannot be parallelized.

Parallel Statements

This section describes how to use a p/t-set to define a statement across multiple threads and processes.

Parallel Compound/Block Statements

Each command in a compound statement is executed in order. The target p/t-set is applied to all statements in a compound statement. The following two examples (i) and (ii) are equivalent:

```
i) pgdbg [all] 0>[*] break
main; cont; wait; print f@11@i
ii) pgdbg [all] 0>[*] break
main; [*]cont; [*]wait; [*]print
f@11@i
```

Use the wait command if subsequent commands require threads to be stopped as the print command above does.

The threadwait and procwait environment variables do not affect how commands within a compound statement are processed. These pgienv environment variables describe to PGDBG under what conditions (runstate of program) it should accept the next (compound) statement.

Parallel If, Else Statements

A prefix p/t-set can be used to parallelize an if statement. An if statement executes in the context of the current thread by default. The following example:

```
pgdbg [all] 0> [*] if
(i==1) {break func; c; wait} else {sync
func2}
```

is equivalent to the following pseudo-code:

```
 for the subset of [*] where
(i==1)
 break func; c; wait;
 for the subset of [*] where (i!=1)
 sync func2
```

Parallel While Statements

A prefix p/t-set can be used to parallelize a while statement. A while statement executes in the context of the current thread by default. The following example:

```
pgdbg [all] 0> [*] while
(i<10) {n; wait; print i}
```

is equivalent to the following pseudo-code:

```
 loop:
 if the subset of [*] is the empty set
 goto done
 for the subset of [*] where (i<10)
 [s]n; [s]wait; [s]print
i;
 goto loop
 done:
```

The while statement terminates when either the subset of the target p/t-set matching the while condition is the empty set, or a return statement is executed in the body of the while.

Return Statements

The return statement is defined only in serial context since it cannot return multiple values. When return is used in a parallel statement, it returns the last value evaluated.

OpenMP Debugging

An attempt is made by PGDBG to preserve line level debugging and to help make debugging OpenMP programs more intuitive. PGDBG preserves line level debugging across OpenMP threads in the following situations:

- Entrance to parallel region
- Exit from parallel region
- Synchronization points of nested parallel regions
- Critical and exclusive sections
- Parallel sections

When directives or pragmas that imply complex parallel operations are encountered in the execution of an OpenMP application, PGDBG treats those directives as a single source line.

Serial vs. Parallel Regions

The initial thread is the thread with OpenMP ID 0. Conceptually, the initial thread is the only thread that can be effectively debugged in a serial region of code. All threads may be debugged in a parallel region of code. When the initial thread is in a serial region, the non-initial threads are waiting to be assigned to do some work in the next parallel region. All threads enter the next parallel region only when the first thread has entered the parallel region.

PGDBG source level debugging operations (next, step,...) are not useful for debugging non-initial threads in serial regions, since these threads are idle, executing low-level code that is not compiled to include source line information. Non-initial threads in serial regions may be debugged using assembly-level debugging operations, but it is not recommended.

To ease debugging in serial and parallel regions of an OpenMP program, PGDBG automatically configures both the thread wait mode and the thread stop mode of the debug session.

Upon entering a serial region, PGDBG automatically changes the thread stop mode to synchronous stop mode and the thread wait mode to all. This allows easy control of all threads together in serial regions. For example, a next command, applied to all threads in a serial region, will complete successfully when the initial thread hits the next source line.

Upon entering a parallel region, PGDBG automatically changes the thread stop mode to asynchronous stop mode and the threadwait mode to none. This allows control of each thread independently. For example, a next command, applied to all threads in a parallel region, will not complete successfully until all threads hit their next source line. With the thread wait mode set to none, use the halt command on threads that hit barrier points.

To disable the automatic configuration of the thread wait and thread stop modes, see the threadstopconfig and threadwaitconfig options of the pgienv command (“Miscellaneous” on page 80).

The configuration of the thread wait and stop modes, as described above, occurs automatically for OpenMP programs only. When debugging a Linuxthread program, the threadstop and threadwait configuration options should be set using the pgienv command (“Miscellaneous” on page 80).

The PGDBG OpenMP Event Handler

The OpenMP event handler is deprecated as of PGDBG release 5.2.

PGDBG provides optional explicit support for OpenMP events. OpenMP events are points in a well-defined OpenMP program where the behavior of one thread depends on the location of another thread. For example, a thread may continue after another thread reaches a barrier point. The PGDBG OpenMP event handler is disabled by default. It can be enabled using the pgienv omp environment variable as shown below:

```
pgienv omp [on|off]
```

The PGDBG OpenMP event handler sets breakpoints before a parallel region, after a parallel region, and at each thread synchronization point. Using the OpenMP event handler causes a noticeable slowdown in performance of the program as it runs with the debugger.

Debugging OpenMP Private Data

PGDBG supports debugging of OpenMP private data for all supported languages as of release 6.0. When an object is declared private in the context of an OpenMP parallel region, it essentially means that each thread team will have its own copy of the object. This capability is shown in the following Fortran and C/C++ examples, where the loop index variable i is private by default.

FORTRAN example:

```
 program omp_private_data
 integer array(8)
 call omp_set_num_threads(2)
!$OMP PARALLEL DO
 do i=1,8
 array(i) = i
 enddo
!$OMP END PARALLEL DO
 print *, array
 end
```

C/C++ example:

```
#include <omp.h>
int main ()
{
 int i;
 int array[8];
 omp_set_num_threads(2);
#pragma omp parallel
 {
```

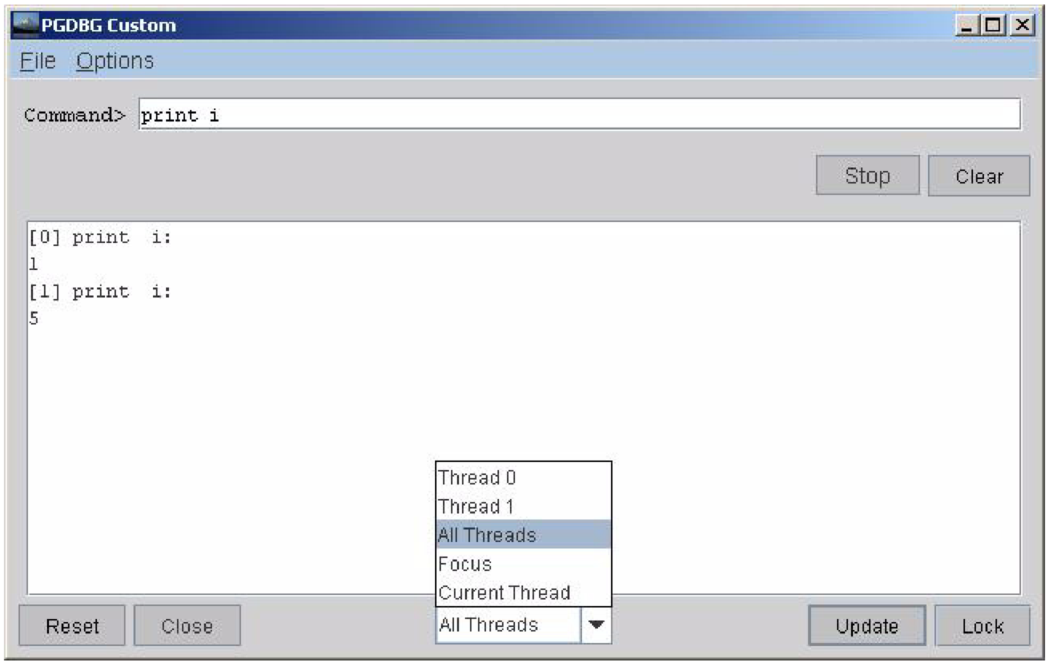
```
#pragma omp for
 for (i = 0; i < 8; ++i)
 array[i] = i;
 }
 for (i = 0; i < 8; ++i)
 printf("array[%d] = %d\n",
i, array[i]);
}
```

Display of OpenMP private data when the above examples are built with a PGI compiler (6.0 or higher) and displayed by PGDBG (6.0 or higher) is as follows:

```
pgdbg [all] 0> [*] print
i
[0] print i:
1
[1] print i:
5
```

The example specifies [*] for the p/t-set to execute the print command on all threads. Table 1-14 shows the values for i in the PGDBG GUI using a Custom Window. Note that All Threads is selected in the Context Selector to display the value on both threads.

Figure 1-14: OpenMP Private Data in PGDBG GUI



Command> print i

```
[0] print  i:
1
[1] print  i:
5
```

MPI Debugging

MPI debugging is supported on Linux platforms.

Process Control

PGDBG is capable of debugging parallel-distributed MPI programs and hybrid distributed multi-threaded applications. PGDBG is invoked via MPIRUN and automatically attaches to each MPI process as it is created. See "Multi-Process MPI Debugging" on page 103, to get started.

Here are some things to consider when debugging an MPI program:

- Use p/t-sets to focus on a set of processes. Be mindful of process dependencies.
- In order for a process to receive a message, the sender must be allowed to run.
- Process synchronization points, such as MPI_Barrier, will not return until all processes have hit the sync point.

- MPI_Finalize will not return for Process 0 until Process 1..n-1 exit.

A control command (cont, step, ...) can be applied to a stopped process while other processes are running. A control command applied to a running process is applied to the stopped threads of that process and is ignored by its running threads. Those threads held by the OpenMP event handler will also ignore the control command in most situations.

PGDBG automatically switches to process wait mode none as soon as it attaches to its first MPI process. See the pgienv command and "Configurable Wait Mode" on page 123, for details.

Use the run command to rerun an MPI program. The rerun command is not useful for debugging MPI programs since MPIRUN passes arguments to the program that must be included. After MPI debugging is shut down, PGDBG will clean up all of its MPI processes.

Process Synchronization

Use the PGDBG sync command to synchronize a set of processes to a particular point in the program. The following command runs all processes to MPI_Finalize:

```
pgdbg [all] 0.0> sync MPI_Finalize
```

The following command runs all threads of process 0 and process 1 to MPI_Finalize:

```
pgdbg [all] 0.0> [0:1.*] sync
MPI_Finalize
```

A synchronize command will only successfully sync the target processes if the sync address is well defined for each member of the target process set, and all process dependencies are satisfied. If these conditions are not met, for example, a member could wait forever for a message. The debugger cannot predict if a text address is in the path of an executing process.

MPI Message Queues

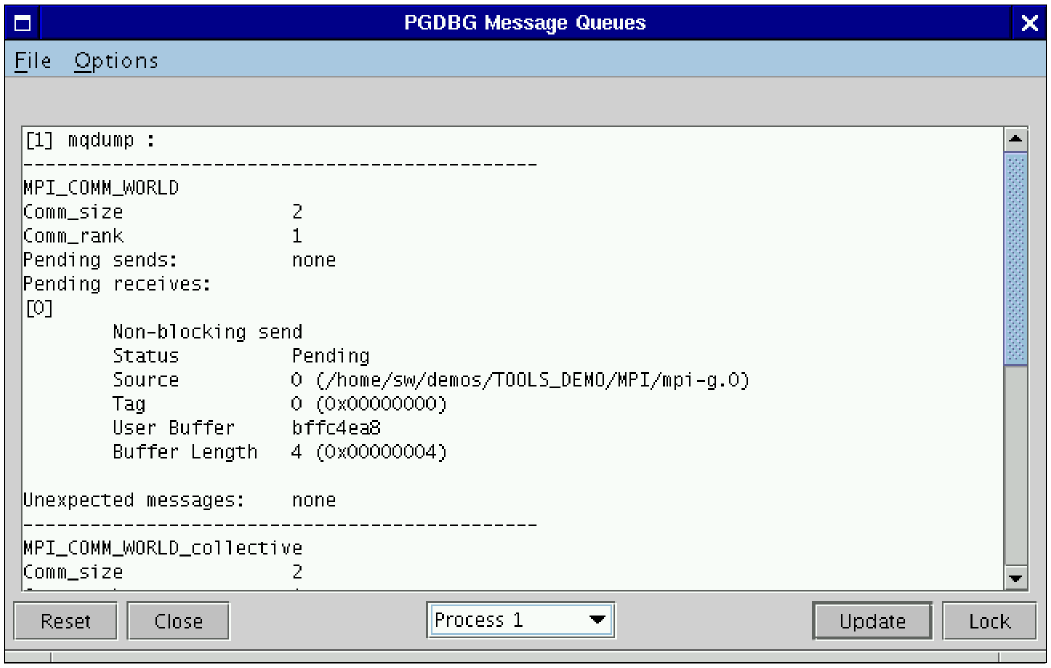
PGDBG can dump the MPI message queues through the mqdump command ("Memory Access" on page 77). In the PGDBG GUI, the message queues can be viewed by selecting the Messages item under the Windows menu. This command can also have a p/t-set prefix to specify a subset of processes and/or threads. When using the GUI, a subwindow is displayed with the message queue output as shown in Figure 1-15 (the PGDBG text debugger produces the same output). Within the subwindow, you can select which process/threads to display with the Context Selector combo box located at the bottom of the subwindow (e.g., Process 1 in Figure 1-15).

The message queue dump is only available for MPI application debugging using the PGI CDK licensed version of PGDBG. The following error message may display if you invoke mqdump:

```
ERROR: MPI Message Queue library not found. Try setting
'PGDBG_MQS_LIB_OVERRIDE' environment
variable.
```

If this message is displayed by a CDK-licensed version of PGDBG, then the PGDBG_MQS_LIB_OVERRIDE environment variable should be set to the absolute path of libtvmpich.so or compatible library. This library is normally located in $PGI/lib.

Figure 1-15: Messages Subwindow



```
[1] mqdump :
----------------------------------------------
MPI_COMM_WORLD
Comm_size               2
Comm_rank               1
Pending sends:          none
Pending receives:
[0]
        Non-blocking send
        Status          Pending
        Source          0 (/home/sw/demos/TOOLS_DEMO/MPI/mpi-g.0)
        Tag             0 (0x00000000)
        User Buffer     bffc4ea8
        Buffer Length   4 (0x00000004)

Unexpected messages:    none
----------------------------------------------
MPI_COMM_WORLD_collective
Comm_size               2
```

MPI Groups

PGDBG identifies each process by its COMMWORLD rank. In general, PGDBG currently ignores MPI groups.

MPI Listener Processes

Entering Control-C (^C) from the PGDBG command line can be used to halt all running processes. This is not the preferred method, however, to use while debugging an MPI program. Entering ^C at the command line sends a SIGINT signal to the debugger's children. This signal is never received by the MPI processes listed by the procs command (i.e., the initial and attached processes); SIGINT is intercepted in each case by PGDBG. However, PGDBG does not attach to the MPI listener processes paired with each MPI process. These listener processes will receive a ^C from the command line, which will kill these processes and result in undefined program behavior.

For this reason, PGDBG automatically switches to process wait mode none (pgienv procwait none) as soon as it attaches to its first MPI process. Setting 'pgienv procwait none' allows commands to be entered while there are running processes, which allows the use of the halt command to stop running processes without the use of ^C.

Note: halt cannot interrupt a wait command. ^C must be used for this. In MPI debugging, wait should be used with care.

SSH and RSH

By default, PGDBG uses rsh for communication between remote PGDBG components. PGDBG can also use ssh for secure environments. The environment variable PGRSH, should be set to ssh or rsh, to indicate the desired communication method.

2 The PGPROF Profiler

This chapter describes the PGPROF profiler. The profiler is a tool that analyzes data generated during execution of specially compiled C, C++, F77, F95 and HPF programs. The PGPROF profiler displays information about which routines and lines were executed, how often they were executed, and how much of the total time they consumed.

The PGPROF profiler can also be used to profile multi-process HPF or MPI programs, multi-threaded programs (e.g., OpenMP or programs compiled with –Mconcur, etc.), or hybrid multi-process programs employing multiple processes with multiple threads in each process. Profile data from multi-process and multi-threaded applications can be examined on combined views or on a process-by-process basis. This information can be used to identify communication patterns or the portions of a program that will benefit the most from performance tuning.

Introduction

Profiling is a three-step process:

Compilation	Compile with additional options that may cause special profiling calls to be inserted in the code, generate debugging information that can be used to correlate instruction addresses with source code line numbers and, in most cases, add data collection libraries to be linked in.
Execution	Unless the OProfile interface is used, the profiled program is invoked normally but collects call counts and timing data during execution. When the profile is collected via the OProfile interface, the timing and event data is collected via the OProfile daemon, which must be started and stopped before and after the program is run, respectively. See "Profiling with Hardware Event Counters using PGPROF -collect." on page 146 for more details.
Analysis	The PGPROF tool interprets the pgprof.out file to display the profile data and associated source files. The profiler supports routine level, line level and data collection modes. The next section provides definitions for these data collection modes.

Definition of Terms

Basic Block	At optimization levels above 0, code is broken into basic blocks, which are groups of sequential statements with only one entry and one exit.
Check Box	A check box is a GUI component consisting of a square or box icon that can be selected by left mouse clicking inside the square. The check box has a selected and an unselected state. In its selected state, a check mark will appear inside the box. The box is empty in its unselected state.
Combo Box	A combo box is a GUI component consisting of a text field and a list of text items. In its closed or default state, it presents a text field of information with a small down arrow icon to its right. When the down arrow icon is selected by a left mouse-click, the box opens and presents a list of choices.
CPU Time	The amount of time the CPU is computing on behalf of a process, not waiting for input/output or running other programs.
Dialog Box	A dialog box is a GUI component that displays information in a graphical box. It may also request some input from the user. After reading and/or entering some information, the user can click on the OK button to acknowledge the message and/or accept their input.
Elapsed Time	Total time to complete a task including disk accesses, memory accesses, input/output activities and operating system overhead.
Function Level Profiling	Call counts and execution times are collected on a per-routine (e.g., subroutine, subprogram, function, etc.) basis.
GUI	Stands for Graphical User Interface. A set of windows, and associated menus, buttons, scrollbars, etc., that can be used to control the profiler and display the profile data.
Hardware Counters and Events	These are various performance monitors that allow the user to track specific hardware behavior in their program. Some examples of hardware counters include: Instruction Counts, CPU Cycle Counts, Floating Point Operations, Cache Misses, Memory Reads, and so on.
Host	The system on which the PGPROF profiler executes. This will generally be the system where source and executable files reside, and where compilation is performed.

Instruction Level Profiling	Execution counts and times are collected at the machine instruction level.
Line Level Profiling	Execution counts and times are collected for source lines within a called routine.
Radio Button	A radio button is a GUI component consisting of a circle icon that can be selected by left mouse clicking inside the circle. The radio button has a selected and an unselected state. In its selected state, the circle is filled in with a solid color, usually black. The circle is empty or unfilled when the button is in its unselected state.
Routine Level Profiling	Call counts and execution times are collected on a per-routine (e.g., subroutine, subprogram, function, etc.) basis.
Target Machine	The system on which a profiled program runs. This may or may not be the same system as the host.
Virtual Timer	A statistical method for collecting time information by directly reading a timer which is being incremented at a known rate on a processor by processor basis.
Wall-Clock Time	Total time to complete a task including disk accesses, memory accesses, input/output activities and operating system overhead.

Compilation

The following list shows compiler options that cause profile data collection calls to be inserted and libraries to be linked in the executable file:

–Mprof=dwarf	Add limited DWARF symbol information for viewing source line information when profiling using the OProfile interface or with other third party profilers.
–Mprof=func	Insert calls to produce a pgprof.out file for routine level data (routine entry/exit profiling).
–qp	Same as –Mprof=func.
–Mprof=lines	Insert calls to produce a pgprof.out file which contains both routine and line level data.
–ql	Same as –Mprof=lines.

–pg	[Linux Only] Enable gprof-style (sample-based) profiling. Running an executable compiled with this option will produce a gmon.out profile file which contains routine, line, and instruction level profiling data.
–qp	[Linux Only] Same as –pg.
–Mprof=time	[Linux Only] Same as –pg except a pgprof.out file is produced rather than a gmon.out file.
–Mprof=hwcts	[Linux Only] Produce an instruction level profile using hardware counters via the PAPI interface. Compiling and linking with this option produces an executable that generates a pgprof.out file which contains function (routine), line, and instruction level profiling data. See "Profiling with Hardware Event Counters using PAPI" on page 148 for more information on profiling with hardware counters.
–Mprof=mpi	[Linux Only] Link with an MPI profile library which intercepts MPI calls in order to record message sizes and to count message sends and receives. This switch can be used in addition to the other types of profiling available (e.g., –Mprof=mpi,time; –Mprof=mpi,hwcts; –Mprof=mpi,func; –Mprof=mpi,lines).
–Mprof=cost	[ST100 Only] Insert calls to produce a pgprof.out file for routine level cost profiling. See the definition of COST in "Profile Data" on page 153 for more information.

NOTE

Not all profiler options are available in every compiler. Please consult the appropriate compiler user guide for a complete list of profiling options. A list of available profiling options can also be generated with the compiler's –help option.

PGI supports three methods of profiling: Sample based profiling via a program library [Linux Only], sample based profiling via the OProfile interface [Linux Only], and profiling through instrumentation of user code. Compiling with -pg, -Mprof=time, and -Mprof=hwcts switches enables generating sampled based profiling via a program library. See "Profiling with Hardware Event Counters using PGPROF - collect." on page 146 for details on generating sample based profiles via the OProfile interface. Sample based profiling may provide more accurate timings than instrumentation (e.g., –Mprof=[lines|func]) because it can be less intrusive. It may have some limitations on particular systems; check the PGPROF release notes for more information. Instrumentation of user code allows computing the coverage accomplished during execution of an application (see Coverage in "Profile Data" on page 153). There are also differences in how instrumentation and sample based profiling measure time (see "Measuring Time" on page 152).

When working with sample based profiles, it is important that PGPROF know the name of the executable. By default, PGPROF will assume that your executable is called a.out. To indicate a different executable, use the –exe command line argument or the Set Executable… option under the File menu in the GUI. See "Profiler Invocation and Initialization" on page 150 for more information on changing the executable name.

Program Execution

Once a program is compiled for profiling, it must be executed to produce profile data. The profiled program is invoked normally, but while running, it collects call counts and/or time data. When the program terminates, a profile data file is generated. Depending on the profiling method used, this data file is called pgprof.out or gmon.out. The following system environment variables can be set to change the way profiling is performed:

- GMON_ARCS – Use this environment variable to set the maximum number of arcs (caller/callee pairs). The default is 4096. This option only applies to gprof style profiling (e.g., programs compiled with the –pg option).
- PGPROF_DEPTH – Use this environment variable to change the maximum routine call depth for PGPROF profiled programs. The default is 4096 and is applied to programs compiled -Mprof=func, –Mprof=lines, –Mprof=mpi, –Mprof=mpi,hwcts, or –Mprof=mpi,time.
- PGPROF_EVENTS – Use this environment variable to specify hardware (event) counters from which to collect data. This variable is applied to programs compiled with the –Mprof=hwcts or -Mprof=hwcts,mpi options. The use of hardware (event) counters is discussed in further detail in "Profiling with Hardware Event Counters using PAPI" on page 148.
- PGPROF_NAME – Use this environment variable to change the name of the output file intended for PGPROF. The default is pgprof.out. This option is only applied to programs compiled with the -Mprof=[func | hwcts | lines | mpi | time] options. If a program is compiled with the –pg option, then the output file is always called gmon.out.

Profiling MPI Programs

MPI profiling is available only on Linux platforms.

To profile an MPI program, use mpirun or mpiexec to execute a program that was compiled and linked with the –Mprof=mpi switch. A separate data file is generated for each non-initial MPI process. The pgprof.out file acts as the "root" profile data file. It contains profile information on the initial MPI process and points to the separate data files for the rest of the processes involved in the profiling run.

Profiling Multi-threaded Programs

Profiling of multi-threaded programs (e.g., OpenMP, programs compiled with –Mconcur, etc.) has different results depending on the profiling method used. If a program is compiled with –Mprof=hwcts, –Mprof=lines, –Mprof=func, or –Mprof=mpi,[hwcts | lines | func], then each thread gets profiled. If a program is compiled with –pg or –Mprof=time, then only the master thread gets profiled.

The type of data presented by each profiling method varies. Profiles generated with –Mprof=func, -Mprof=lines, or –Mprof=mpi,[func | lines] measure elapsed or wall-clock time for each thread. Profiles generated with –Mprof=hwcts or –Mprof=mpi,hwcts collect hardware counter data for each thread. Profiles generated with –pg or –Mprof=time collect total CPU time for the master thread only. Elapsed time is not available for –Mprof=hwcts, –Mprof=mpi,hwcts, –pg, or –Mprof=time.

Profiling with Hardware Event Counters (Linux Only)

On Linux platforms, PGI performance tools provide support for capturing information about low-level processor behavior (e.g. cache misses) and correlating it with source or assembly code using PGPROF. Two methods of data collection are supported: execution of the program under the control of PGPROF using the the -collect option, and building the program with the -Mprof=hwcts compiler option and executing it independently. Collection of profile data using pgprof -collect may be done on any linux86 or linux86_64 system where Oprofile is installed. Profiling by compiling with the -Mprof=hwcts option is only available on linux86_64 systems where PAPI has been installed. OProfile is included as an install-time option with most Linux distributions; it may also be downloaded from oprofile.sourceforge.net. PAPI is available for download from http://icl.cs.utk.edu/papi/.

Profiling with Hardware Event Counters using PGPROF -collect.

PGPROF can also be used to display time-based and hardware event-based profiles generated via the OProfile package, which is available on most current Linux distributions.

Unlike profiling with the PAPI interface, no special link time options are needed to enable profiling, though compiling with -Mprof=dwarf, -g, or -gopt allows viewing profiles with source code annotations under PGPROF.

To simplify generating profiles, invoke pgprof with the -collect option. PGPROF will execute the program, collect profile data and generate a pgprof.out file, viewable with PGPROF.

Default event specification options are provided to handle standard profiling situations. For example:

```
pgprof -collect -dcache program arg ...
```

With the options shown above, pgprof monitors various causes of data cache miss. By default a one millisecond time-based profile is produced. See the output of "pgprof -help" for more usage information.

Note that pgprof -collect can invoke a script instead of a program. This is useful if you want to produce an aggregated profile of several invocations of the program using different data sets. In this situation, use the -exe option, which allows the data collection phase to determine which program is being profiled.

```
pgprof -collect -dcache -exe program sh run_script
```

In this situation, if you neglect to specify the -exe option, you can generate the pgprof.out file by executing the following command before another profiling run is started:

```
optopgprof
program
```

The driver script pgprof contains more detailed documentation on its usage.

When using PGPROF -collect, you control the OProfile kernel driver and the sample collection daemon via opcontrol. This requires root privileges for management operations. Thus, invocations to opcontrol, which are performed when pgprof is called with the -collect option, are executed via the sudo commmand. One technique that requires minimal updates to the /etc/sudoers files is to assume that all users in a group are allowed to execute opcontrol with group privileges. For example, you could make the following changes to /etc/sudoers:

```
# User
alias specification
User_Alias SW = %sw
...
SW ALL=NOPASSWD: /usr/bin/opcontrol
```

The lines above permit all members of the group ’sw’ to run opcontrol with root privileges.

Note that pgprof -collect will shutdown the oprofile daemon when interrupted. However, if the script is terminated with SIGKILL, you must execute the following:

```
pgprof -collect -shutdown
```

This is important because if the oprofile daemon is left running, disk space on the root file system will eventually be exhausted.

Since OProfile provides only system wide profiling, when you invoke pgprof with the -collect option pgprof provides a locking mechanism that allows only one invocation to be active at a time. Note that this locking mechanism is external to OProfile and does not prevent other profile runs from invoking opcontrol through other mechanisms, but it would is straightforward to incorporate pgprof's locking mechanism in other OProfile-based profiling scripts.

Profiling with Hardware Event Counters using PAPI

To use PAPI-style profiling, PAPI must be installed. Installation of PAPI requires rebuilding the Linux kernel. PGI compiler and tools releases are built with the version of PAPI that is current at the time of the PGI release. Normally, the profiling support code for -Mprof=hwcts supports profiling against that current version and the previous version of PAPI (though a warning message is generated if the previous version is used).

To bypass the version check, set the environment variable PGPROF_PAPI_VER to m.n where where m and n respectively are the major and minor numbers associated with your PAPI library.

To profile using hardware counters, compile with the option –Mprof=hwcts. This option adds the PAPI and PGI profiling libraries to the application's linker command. By default, this will use the PAPI_TOT_CYC counter to profile total CPU cycles executed by the application. PGPROF will convert the cycle counts into CPU time (seconds). The PGPROF_EVENTS environment variable can be set to specify up to four counters to use. The format for the PGPROF_EVENTS variable follows:

```
event0[.over][:event1[.over]] _
```

The event field is the name of the event or hardware counter and the optional over field specifies the overflow value. The overflow value is the number of events to be counted before collecting profile information. Overflow provides some control on the sampling rate of the profiling mechanism. The default overflow is 1000000.

To determine which hardware counters are available on the system, compile and run the following simple program. This program uses the PAPI and PGI libraries to dump the available hardware counters to standard output.

```
int main(int argc, char *argv[]) {
__pgevents();
exit(0);
}
```

Save the code in the previous example in a file called pgevents.c and compile it as follows:

```
pgcc pgeventc.c -o pgevents -lpgnod_prof_papi
-lpapi
```

To display the available events, run the newly created program called pgevents. The pgevents utility shows the format of the PGPROF_EVENTS environment variable, the list of PAPI preset events, and the list of native (or processor specific) events. Below is an example of specifying 4 events with the PGPROF_EVENTS environment variable (using tcsh or csh shells):

```
% setenv PGPROF_EVENTS \
PAPI_TOT_CYC.1593262939:PAPI_FP_OPS:PAPI_L1_DCM:PAPI_L2_ICM
```

Specify the events above using the sh or bash shells in the following manner:

```
$ set PGPROF_EVENTS=\
PAPI_TOT_CYC.1593262939:PAPI_FP_OPS:PAPI_L1_DCM:PAPI_L2_ICM
$ export PGPROF_EVENTS
```

If PGPROF_EVENTS is not defined, then the profiling mechanism will count CPU cycles (PAPI_TOT_CYC event) by default.

The following example shows a partial output from pgevents:

```
Selecting Events
Hardware Information
cpus/node - 4
nodes - 1
total cpus - 4
vendor - AuthenticAMD
model - AMD K8 Revision C
speed 1593.262939mhz
event counters 4
Preset Events
PAPI_L1_DCM - Level 1 data cache misses
PAPI_L1_ICM - Level 1 instruction cache misses
PAPI_L2_DCM - Level 2 data cache misses
PAPI_L2_ICM - Level 2 instruction cache misses
PAPI_L1_TCM - Level 1 cache misses
```

```
PAPI_L2_TCM - Level 2 cache misses
...
PAPI_TOT_CYC - Total Cycles
...
Native Events
FP_ADD_PIPE - Dispatched FPU ops - Revision B
and later revisions - Add pipe ops excluding junk ops.
FP_MULT_PIPE - Dispatched FPU ops - Revision B
and later revisions - Multiply pipe ops excluding junk ops.
...
CPU_CLK_UNHALTED - Cycles processor is running
(not in HLT or STPCLK state)
```

Profiler Invocation and Initialization

The PGPROF profiler is used to analyze profile data produced during the execution phase as described in "Introduction" on page 141.

The PGPROF profiler is invoked as follows:

```
% pgprof [options] [datafile]
```

If invoked without any options or arguments, the PGPROF profiler attempts to open a data file named pgprof.out, and assumes that application source files are in the current directory. The program executable name, as specified when the program was run, is usually stored in the profile data file. If all program-related activity occurs in a single directory, the PGPROF profiler needs no options.

Selecting a Version of Java

PGPROF (both GUI and command line) depends on Java. PGPROF requires that the Java Virtual Machine be a specific mimimum version or above. By default, PGPROF will use the version of Java installed with your PGI software; if you chose not to install Java when installing your PGI software, PGPROF will look for Java on your PATH. Both of these can be overriden by setting the PGI_JAVA environment variable to the full path of the Java executable you wish to use. For example, on a Linux system using the bash shell:

```
$ export
PGI_JAVA=/home/myuser/myjava/bin/java
```

Command Line Options

If present, PGPROF options are interpreted as follows:

datafile	A single datafile name may be specified on the command line. For profiled MPI applications, the specified datafile should be that of the initial MPI process. Access to the profile data for all MPI processes is available in that case, and data may be filtered to allow inspection of the data from a subset of the processes.
–s	Use the PGPROF Command Line Interface (CLI).
–text	Same as -s.
–scale “files(s)”	Compare scalability of datafile with one or more files. A list of files may be specified by enclosing the list within quotes and separating each filename with a space. For example:
-scale “one.out two.out”	This example will compare the profiles one.out and two.out with datafile (or pgprof.out by default). If only one file is specified quotes are not required.

For sample based profiles (e.g., gmon.out) specified with this option, PGPROF assumes that all profile data was generated by the same executable. For information on how to specify multiple executables in a sample-based scalability comparison, see the Scalability Comparison… item in the description of the File menu (“File Menu” on page 166).

–I srcdir

Specify the source file search path. The PGPROF profiler will always look for a program source file in the current directory first. If it does not find the source file in the current directory, it will consult the search path specified in srcdir. The srcdir argument is a string containing one or more directories separated by a path separator. The path separator is platform dependent; on Linux/Solaris it is a colon (:) and on Windows it is a semicolon (;). Directories in the path will then be searched in order from left-to-right. When a directory with a filename that matches a source file is found, that directory is used. Below is an example for Linux/Solaris:

```
-I ../src:STEPS
```

In the example above, the profiler first looks for source files in the current directory, then in the ../src directory, followed by the STEPS directory. The following is the same example for Windows:

```
-I ..\src;STEPS
```

	See the Set Source Directory… item in the description of the File menu ("File Menu" on page 166) for more information.
–exe filename	Set the executable to filename (default is a.out).
–o filename	Same as –exe.
–title string	Set the title of the application to string (GUI only).
–V	Print version information.
–help	Prints a list of available command line arguments.
–usage	Same as –help.
-dt (number)	Set the time multiply factor (default is 1.0). This is used to calibrate the times reported by PGPROF. The profiler will display a time multiplied by the specified number. This option works for all profiling mechanisms that measure time (e.g., –Mprof=time, – pg, –Mprof=lines, –Mprof=func, –Mprof=mpi.[time | lines | func]).

Measuring Time

The profiling mechanism will collect total CPU time for programs compiled with –Mprof=time, –pg, and –Mprof=mpi,time (see also "Profiling Multi-threaded Programs" on page 146). For programs compiled with –Mprof=hwcts or –Mprof=mpi,hwcts, no timings are collected. For programs compiled to count CPU cycles with –Mprof=hwcts or –Mprof=mpi,hwcts, PGPROF automatically converts CPU cycles into CPU time.

Programs compiled with –Mprof=lines, –Mprof=func, or –Mprof=mpi,[time | lines | func] employ a virtual timer for measuring the elapsed time of each running process/thread. This data collection method employs a single timer that starts at zero (0) and is incremented at a fixed rate while the active program is being profiled. For multiprocessor programs, there is a timer on each processor, and the profiler's summary data (minimum, maximum and per processor) is based on each processor's time executing in a function. How the timer is incremented and at what frequency depends on the target machine. The timer is read from within the data collection functions and is used to accumulate COST and TIME values for each line, function, and the total execution time. The line level data is based on source lines; however, in some cases, there may be multiple statements on a line and the profiler will show data for each statement.

NOTE

Due to the timing mechanism used by the profiler to gather data, information provided for longer running functions will be more accurate than for functions that only execute for a shorter time relative to the overhead of the individual timer calls. Refer to "Caveats (Precision of Profiling Results)" on page 154 for more information about profiler accuracy.

Profile Data

The following statistics are collected and may be displayed by the PGPROF profiler.

BYTES	For HPF and MPI profiles only. This is the number of message bytes sent and received.
BYTES RECEIVED	For HPF and MPI profiles only. This is the number of bytes received in a data transfer.
BYTES SENT	For HPF and MPI profiles only. This is the number of bytes sent.
CALLS	The number of times a function is called.
COST	[ST100 only] The sum of the differences between the timer value entering and exiting a function. This includes time spent on behalf of the current function in all children whether profiled or not. PGPROF can provide cost information when you compile your program with the –Mprof=cost or –Mprof=lines options (see "Compilation" on page 143).
COUNT	The number of times a line or function is executed.
COVERAGE	This is the percentage of lines in a function that were executed at least once.
LINE NUMBER	For line mode, this is the line number for that line. For function mode, this is the line number of the first line of the function.
MESSAGES	For HPF and MPI profiles only. This is the number of messages sent and received by the function or line.
RECEIVES	For HPF and MPI profiles only. This is the number of messages received by the function or line.
SENDS	For HPF and MPI profiles only. This is the number of messages sent by the function or line.

STMT ON LINE	For programs with multiple statements on a line, data is collected and displayed for each statement individually.
TIME	The time spent only within the function or executing the line. The TIME does not include time spent in functions called from this function or line. TIME may be displayed in seconds or as a percent of the total time.
TIME PER CALL	This is the TIME for a function divided by the CALLS to that function. TIME PER CALL is displayed in milliseconds.

Caveats (Precision of Profiling Results)

Accuracy of Performance Data

The collection of performance data will always introduce some overhead, or intrusion, that can affect the behavior of the application being monitored. How this overhead affects the accuracy of the performance data depends on the performance monitoring method chosen, system software and hardware attributes, and the idiosyncracies of the profiled application. Although the PGPROF implementation attempts to minimize intrusion and maximize accuracy, it would be unwise to assume the data is beyond question.

Clock Granularity

Many target machines provide a clock resolution of only 20 to 100 ticks per second. Under these circumstances, a function must consume at least a few seconds of CPU time to generate meaningful line level times.

Souce Code Correlation

At higher optimization levels, and especially with highly vectorized code, significant code reorganization may occur within functions. The PGPROF profiler allows line profiling at any optimization level. In some cases, the correlation between source and data may at times appear inconsistent. Compiling at a lower optimization level or examining the assembly language source may be necessary to interpret the data in these cases.

Overhead of -Mprof=lines

The profiling mode enabled by the compiler switch -Mprof=lines adds real time instrumentation calls to compiler generated code. This profiling mode incurs a significant overhead due to the need to generate a timestamp for each line executed. Currently on Linux, the gettimeofday system call is invoked to generate these timestamps. On modern x86 and x64 processors, the Linux kernel can implement gettimeofday via a virtual system call mechanism which allows the timestamps to be generated in less than .1 microseconds. When the timestamp is generated by the power management timer via the normal system call mechanism, a call to gettimeofday can take between 1 to 2 microseconds. The Linux kernel will use the virtual system call implementation on single processor systems, or on Multi-Processor (MP) systems where it is known that the rdtsc instructions generates timestamps via a synchronized clock (more modern processors). Unless the virtual system call mode is used for gettimeofday, extremely long run times will result for code compiled with the -Mprof=lines option.

The overhead of the gettimeofday system call can be determined by running the following program on an unloaded system:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <sys/time.h>
void warmup(int usecs)
{
 struct timeval tv;
 long long t0, t, dt;
 volatile int i;
 gettimeofday(&tv, NULL);
 t0 = tv.tv_sec * 1000000 + tv.tv_usec;
 for ( ; ; ) {
 for (i = 0; i < 1000; i++)
 ;
 gettimeofday(&tv, NULL);
 t = tv.tv_sec * 1000000 + tv.tv_usec;
 dt = t - t0;
 if (dt > usecs)
 break;
 }
}
int main(int argc, char **argv)
{
 int cnt = 20, i, j;
 int rcnt = 100;
 double *vgs;
 double *vgd, gsum, gave;
 struct timeval tv;
 assert((vgs = calloc(sizeof vgs[0],
cnt + 1)) != NULL);
 assert((vgd = calloc(sizeof vgd[0],
cnt)) != NULL);
 /* Warmup system to jiggle powersaved out
of low frequency state */ warmup(1000000);
 for (i = 0; ; i++) {
 gettimeofday(&tv, NULL);
 vgs[i] = (double)tv.tv_sec + (double)tv.tv_usec/1000000;
 if (i == cnt)
 break;
 for (j = 0; j < rcnt; j++)
 gettimeofday(&tv, NULL);
 }
 gsum = 0;
 for (i = 0; i < cnt; i++) {
 vgd[i] = (vgs[i+1] -
vgs[i]) / rcnt;
 gsum += vgd[i];
 }
 gave = gsum / cnt;
 printf("gettimeofday average (usec) = %f\n",
gave * 1000000);
 return 0;
}
```

Graphical User Interface

The PGPROF Graphical User Interface (GUI) is invoked using the command pgprof. This section describes how to use the profiler with the GUI on systems where it is supported. There may be minor variations in the GUI from host to host, depending on the type of monitor available, the settings for various defaults and the window manager used. Some monitors do not support the color features available with the PGPROF GUI. The basic interface across all systems remains the same, as described in this chapter, with the exception of the differences tied to the display characteristics and the window manager used.

There are two major advantages provided by the PGPROF GUI.

Source Interaction	The PGPROF GUI will display the program source of any routine for which the profiler has information, whether or not line level profile data is available. To display the source code of a routine, select the routine name. Since interpreting profile data usually involves correlating the program

source and the data, the source interaction provided by the GUI greatly reduces the time spent interpreting data. The GUI allows you to easily compare data on a per processor/thread basis, and identify problem areas of code based on processor/thread execution time differences for routines or lines.

Graphical Display of Data It is often difficult to visualize the relationships between the various percentages and execution counts. The GUI displays bar graphs that graphically represent these relationships to help locate the 'hot spots' in the target program.

The PGPROF GUI Layout

On startup, PGPROF, the profiler attempts to load the profile datafile specified on the command line (or the default pgprof.out). If no file is found, a file chooser dialog box is displayed. Choose a profile datafile from the list or select Cancel.

When a profile datafile is opened, PGPROF populates the following areas in the GUI, shown from top to bottom in "Profiler Window" on page 160:

- Profile Summary – Below the "File…Settings…Help" menu bar is the profile summary area. This information displays the label Profiled followed by: executable name, date last modified, the amount of time consumed by the executable and the number of processes (if the application being profiling has more than one process).
- Profile Entry Combo Box – Below the Profile Summary is the Profile Entry Combo Box. The profile entry (profile datafile, source file, subprogram,…) displayed in this box is known as the current profile entry. This entry corresponds to the data highlighted in the profile tables described below. The current entry can be changed by entering a new entry or selecting an entry from the combo box. Left-click on the down-arrow icon to show a list of previously viewed entries available for selection. See "Profile Navigation" on page 162 for more information on profile entries.
- Navigation Buttons – Use the left and right arrow buttons, located on the left of the Profile Entry Combo Box, to navigate between previously viewed profile entries.
- Select Combo Box – This combo box is located to the right of the Profile Entry Combo Box. Open the Select Combo Box to refine the criteria for displaying profile entries in the tables mentioned below. By default, the selection is set to All profile entries.

- Top Left Table – The Top Left Table, located below the Navigation Buttons, displays the static profile entry information. This includes filenames, routine names, and line numbers of the profile entries. When viewing line level information, this table will also show the source code if the source files are available. If this table has more than one entry in it, then a column labeled View displays. See the description on the Bottom Table for more information.

- Top Right Table – The Top Right Table displays profile summary information for each profile entry. To change what is displayed, select the Processes or View menus, discussed in "Processes Menu" on page 173 and "View Menu" on page 174, respectively. To view profile information at the line level, compiled with –Mprof=lines or –pg, then in the routine level view, double click the left mouse button to view its line level profile information.

- Bottom Table – The Bottom Table displays detailed profile information for the current profile entry. For a multi-process application, this table contains a profile entry for each application process. For a multi-threaded (or multi-process/multi-threaded) application, PGPROF offers the option to view process and/or thread level profile information. A Process/Thread Selector (combo box) will appear in the lower right hand corner when profiling multi-threaded programs. Use this combo box to toggle between thread, process, or process/thread profile information. Figure 2-3 , "Figure 2-3: PGPROF with Visible Process/Thread Selector", shows the Process/Thread Selector in its opened state. Three choices are available: Processes, Threads, Process.Threads.

  The heading in the leftmost column will be Process(es) by default. When profiling a multi-threaded application, the heading in the leftmost column will reflect whatever is selected in the Process/Thread Selector. When the leftmost column is displaying processes or threads, each entry will contain integers that represent process/thread IDs. When the leftmost heading is displaying processes and threads (denoted Process(es).Threads in the column heading), each entry is a floating-point number of the form (Process_ID).(Thread_ID). Following the process/thread ID, the filename, routine name, or line number display enclosed in parentheses. This provides additional ownership information of the process/thread. It also acts as a minor sort key. See the discussion on Sort, "Sort Menu" on page 178, for more information.

  This table will display process/thread information for the current profile entry by default. To view other entries, use the View check boxes in the Top Left Table to select other entries. The View check boxes are shown in Figure 2-11 , "Source Lines with Multiple Profile Entries", in "View Menu" on page 174. These support easy comparison of more than one process/thread in the Bottom Table. When you Print the tables to a file or a printer, an entry with a checked View box gets printed with each profile entry. Again, this allows for easy comparison of more than one process/thread. See the Print option, under the File menu, in "File Menu" on page 166 for more information on printing.

- Histogram – Located at the bottom of the GUI is a tabbed pane with two tabs labeled View and Histogram. When the Histogram tab is selected, the GUI displays a histogram of one or more profiled data items. The data items that are displayed are the same data items selected in the View menu (see "View Menu" on page 174). Each row is labeled with the data in the histogram. Each column is a profile entry. The bars are sorted in the order specified in the Sort menu (see "Sort Menu" on page 178). Left-clicking on a bar displays information for the corresponding profile item in the Top Left and Right tables. Double-clicking the left mouse button on a bar will drill down into the profile on that item (see "Profile Navigation" on page 162). Selected bars are highlighted in blue. The histogram is illustrated in Figure 2-2 , "Figure 2-2: Profiler Window with Visible Histogram".

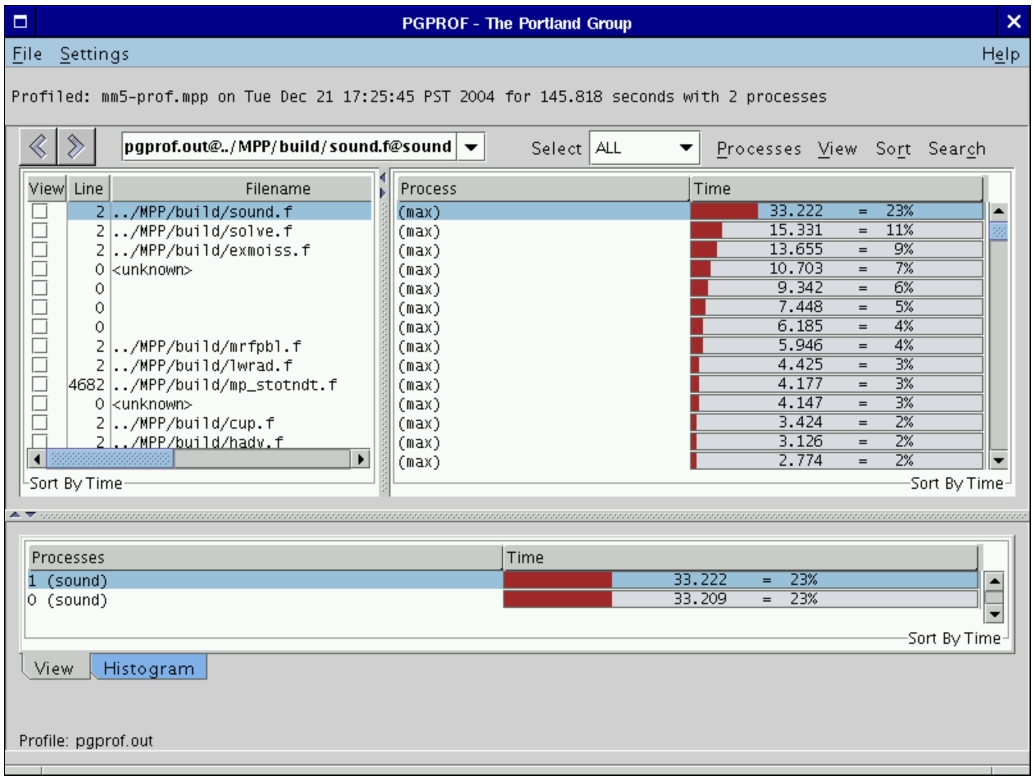
- Profile Name – The Profile Name area is located in the lower left hand corner of the GUI. It is preceded with the keyword Profile: This area displays the profile filename.

GUI Customization

Figure 2-1 , "Profiler Window", shows how the PGPROF GUI appears when launched for the first time. The default dimensions of the GUI are 800 x 600. It can be resized according to the conventions of the window manager. The width of the Top Left and Right tables can be adjusted using the grey vertical divider located between the two tables. The height of the Top Left, Right, and Bottom tables can be adjusted using the grey horizontal divider. Both of these dividers can be dragged in the direction shown by arrow icons located on each divider. Left-click on these arrow icons can be used to quickly "snap" the display in either direction.

After customizing of the display, PGPROF will save the size of the main window and the location of each divider for subsequent PGPROF sessions. To prevent saving these settings on exit from PGPROF, uncheck the Save Settings on Exit item under the Settings menu. The Settings menu is described in more detail in "Settings Menu" on page 168.

Figure 2-1: Profiler Window


PGPROF - The Portland Group
File
Settings
Help
pgprof.out@../MPP/build/sound.f@sound
Select
ALL
Processes
View
Sort
Search
View
Histogram

```
Profiled: mm5-prof.mpp on Tue Dec 21 17:25:45 PST 2004 for 145.818 seconds with 2 processes
```

View	Line	Filename
☐	2	../MPP/build/sound.f
☐	2	../MPP/build/solve.f
☐	2	../MPP/build/exmoiss.f
☐	0	<unknown>
☐	0	
☐	0	
☐	0	
☐	2	../MPP/build/mrfpbl.f
☐	2	../MPP/build/lwrad.f
☐	4682	../MPP/build/mp_stotndt.f
☐	0	<unknown>
☐	2	../MPP/build/cup.f
☐	2	../MPP/build/hadv.f

Sort By Time

Process	Time
(max)	33.222 = 23%
(max)	15.331 = 11%
(max)	13.655 = 9%
(max)	10.703 = 7%
(max)	9.342 = 6%
(max)	7.448 = 5%
(max)	6.185 = 4%
(max)	5.946 = 4%
(max)	4.425 = 3%
(max)	4.177 = 3%
(max)	4.147 = 3%
(max)	3.424 = 2%
(max)	3.126 = 2%
(max)	2.774 = 2%

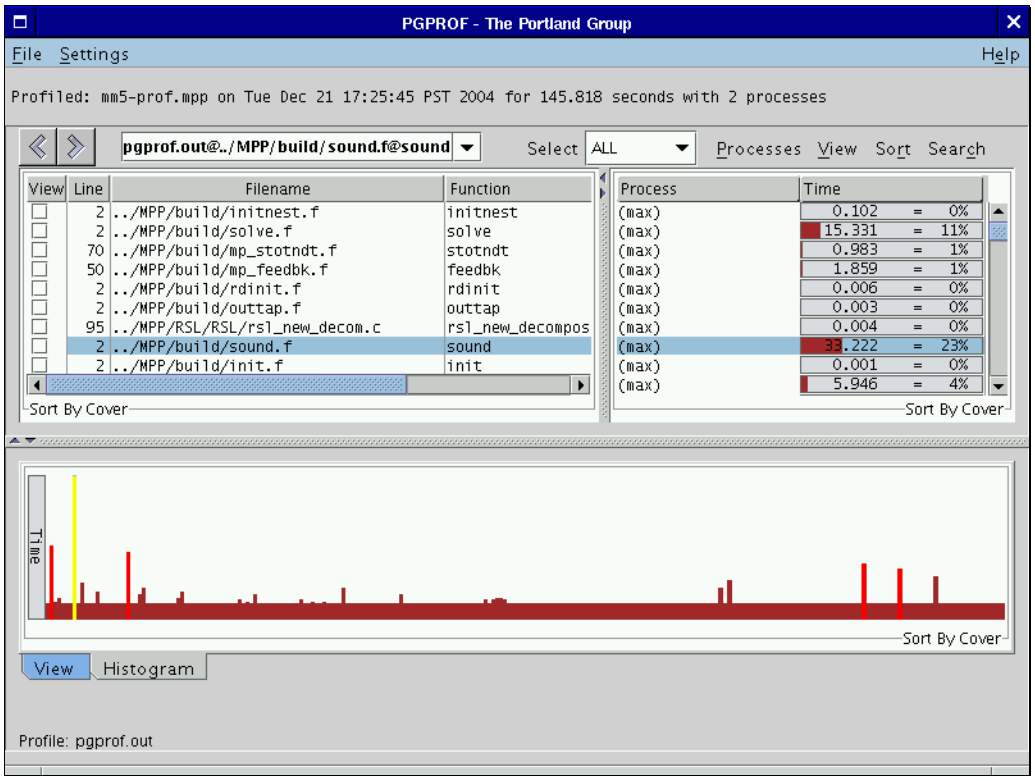
Sort By Time

Processes	Time
1 (sound)	33.222 = 23%
0 (sound)	33.209 = 23%

Sort By Time

Profile: pgprof.out

Figure 2-2: Figure 2-2: Profiler Window with Visible Histogram



```
Profiled: mm5-prof.mpp on Tue Dec 21 17:25:45 PST 2004 for 145.818 seconds with 2 processes
```

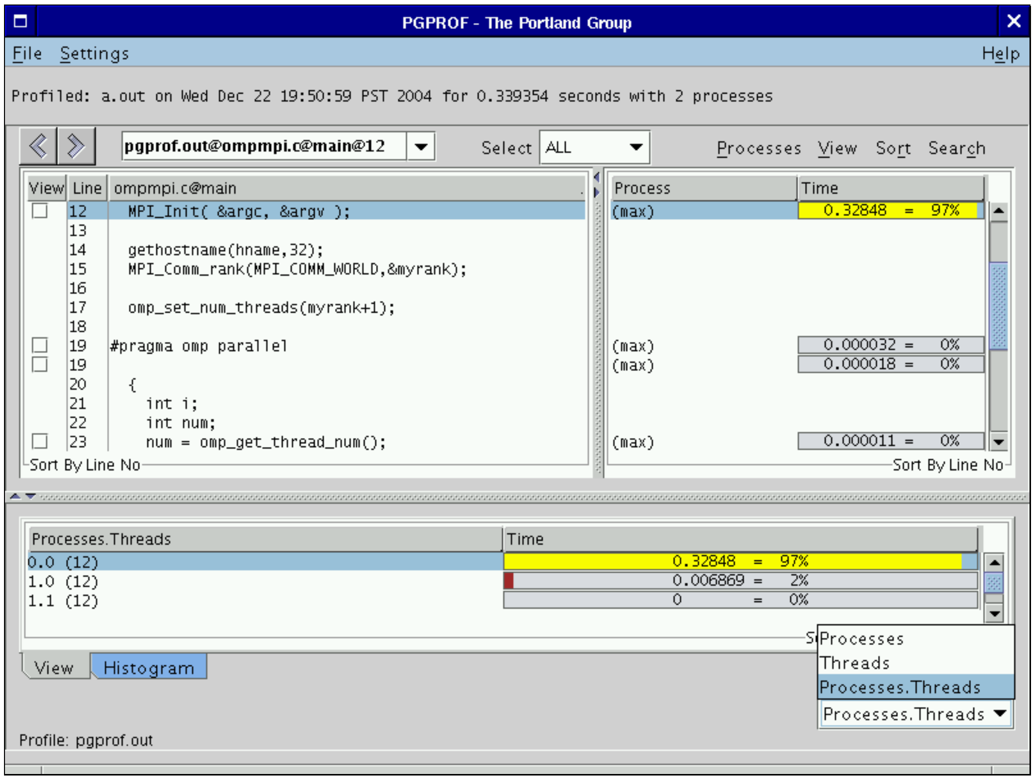
pgprof.out@../MPP/build/sound.f@sound

View	Line	Filename	Function
☐	2	../MPP/build/initnest.f	initnest
☐	2	../MPP/build/solve.f	solve
☐	70	../MPP/build/mp_stotndt.f	stotndt
☐	50	../MPP/build/mp_feedbk.f	feedbk
☐	2	../MPP/build/rdinit.f	rdinit
☐	2	../MPP/build/outtap.f	outtap
☐	95	../MPP/RSL/RSL/rsl_new_decom.c	rsl_new_decompos
☐	2	../MPP/build/sound.f	sound
☐	2	../MPP/build/init.f	init

Process	Time
(max)	0.102 = 0%
(max)	15.331 = 11%
(max)	0.983 = 1%
(max)	1.859 = 1%
(max)	0.006 = 0%
(max)	0.003 = 0%
(max)	0.004 = 0%
(max)	33.222 = 23%
(max)	0.001 = 0%
(max)	5.946 = 4%

Profile: pgprof.out

Figure 2-3: Figure 2-3: PGPROF with Visible Process/Thread Selector


PGPROF - The Portland Group
File Settings Help
pgprof.out@ompmpi.c@main@12
Select ALL
Processes View Sort Search
View Histogram

```
Profiled: a.out on Wed Dec 22 19:50:59 PST 2004 for 0.339354 seconds with 2 processes
```

View	Line	ompmpi.c@main
☐	12	MPI_Init(&argc, &argv);
	13	
	14	gethostname(hname,32);
	15	MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
	16	
	17	omp_set_num_threads(myrank+1);
	18	
☐	19	#pragma omp parallel
☐	19	
	20	{
	21	int i;
	22	int num;
☐	23	num = omp_get_thread_num();

Sort By Line No

Process	Time
(max)	0.32848 = 97%
(max)	0.000032 = 0%
(max)	0.000018 = 0%
(max)	0.000011 = 0%

Sort By Line No

Processes.Threads	Time
0.0 (12)	0.32848 = 97%
1.0 (12)	0.006869 = 2%
1.1 (12)	0 = 0%

Processes
Threads
Processes.Threads
Processes.Threads

Profile: pgprof.out

Profile Navigation

The PGPROF GUI is modeled after a web browser. The current profile entry can be thought of as an address, similar to a web page address (e.g. URL). This address is displayed in the Profile Entry Combo Box; introduced in "The PGPROF GUI Layout" on page 157. The address format follows:

```
(profile)[@sourceFile[@routine[@lineNumber[@textAddress]]]]
```

The only required component of the address is the profile datafile (e.g., pgprof.out, gmon.out, etc.). Each additional component is separated by an '@'. For example, Figure 2-4 , "Example Routine Level Profile" shows a profile of an application with a single routine called main. When a profile is initially displayed, the first entry in the Top Left and Right tables is selected (highlighted) by default. The Profile

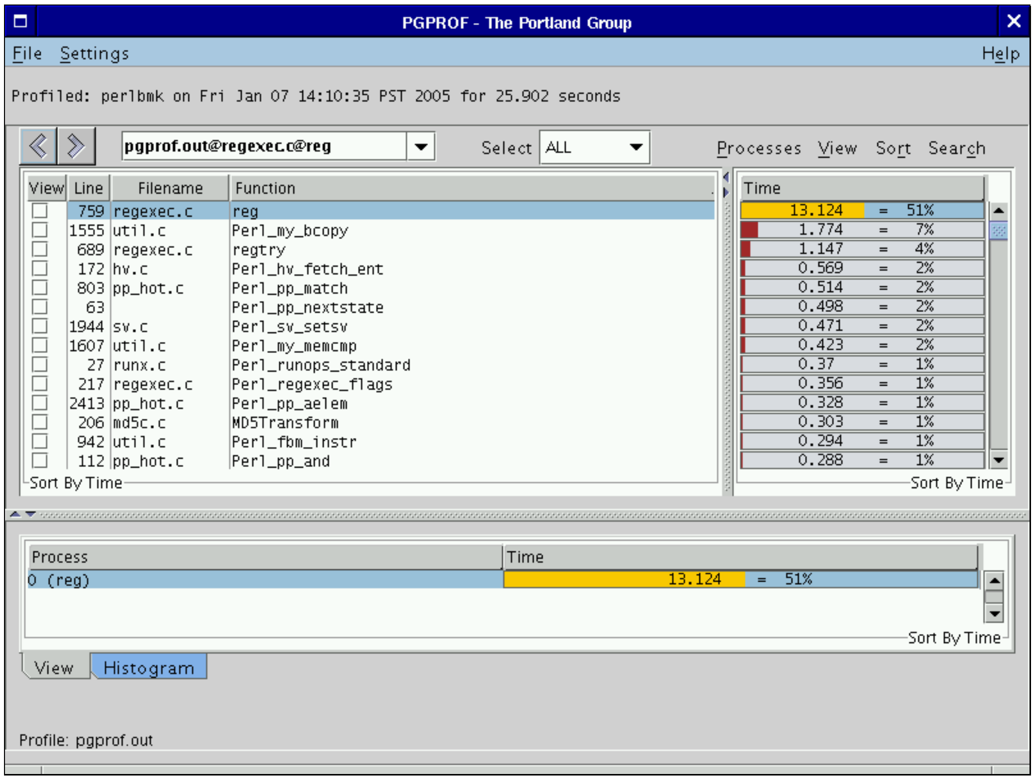
Entry Combo Box reflects the selected entry by displaying its address. In this case, the Profile Entry Combo Box contents are: pgprof.out@regexec.c@reg. This indicates that the current profile entry is a routine named reg located in file regexec.c.

A different address can be entered in the Profile Entry Combo Box using the above address format or by choosing a previously viewed profile entry in the combo box. Click on the down arrow in the combo box to choose from a list of previously viewed profile entries. As described in "The PGPROF GUI Layout" on page 157, previously viewed profile entries can be selected with the Profile Entry Combo Box or with the Navigation Buttons.

The current profile entry is highlighted in the Top Left, Right, and Histogram tables. To change the current profile entry, left-click on a new entry in the Right Table or Histogram. This may also be done by clicking on an entry in the Left Table, but there must be a corresponding entry in the Right Table. Double clicking the left mouse button on a profile entry will drill down into the selected profile entry. The example in Figure 2-4 , "Example Routine Level Profile" assumes that the program was compiled with –Mprof=time and the current profile entry is pgprof.out@regexec.c@reg. Double clicking on the highlighted entry in the Right Table causes PGPROF to display reg's line level information. Figure 2-5 , "Example Line Level Profile", shows this example after double clicking on main. Double clicking on line 759 causes PGPROF to display the instruction level profile shown in Figure 2-6 , "Example Instruction Level Profile".

Drilling down works at higher levels of profiling too. For example, if the current profile entry is pgprof.out, then double clicking on pgprof.out displays a list of profiled files and their profile information. Double clicking on a file from this list moves to the routine level profiling information for that file, etc.

Figure 2-4: Example Routine Level Profile



Profiled: perlbmk on Fri Jan 07 14:10:35 PST 2005 for 25.902 seconds

pgprof.out@regexec.c@reg

Select ALL

View	Line	Filename	Function	Time
☐	759	regexec.c	reg	13.124 = 51%
☐	1555	util.c	Perl_my_bcopy	1.774 = 7%
☐	689	regexec.c	regtry	1.147 = 4%
☐	172	hv.c	Perl_hv_fetch_ent	0.569 = 2%
☐	803	pp_hot.c	Perl_pp_match	0.514 = 2%
☐	63		Perl_pp_nextstate	0.498 = 2%
☐	1944	sv.c	Perl_sv_setsv	0.471 = 2%
☐	1607	util.c	Perl_my_memcmp	0.423 = 2%
☐	27	runx.c	Perl_runops_standard	0.37 = 1%
☐	217	regexec.c	Perl_regexec_flags	0.356 = 1%
☐	2413	pp_hot.c	Perl_pp_aelem	0.328 = 1%
☐	206	md5c.c	MD5Transform	0.303 = 1%
☐	942	util.c	Perl_fbm_instr	0.294 = 1%
☐	112	pp_hot.c	Perl_pp_and	0.288 = 1%

Sort By Time

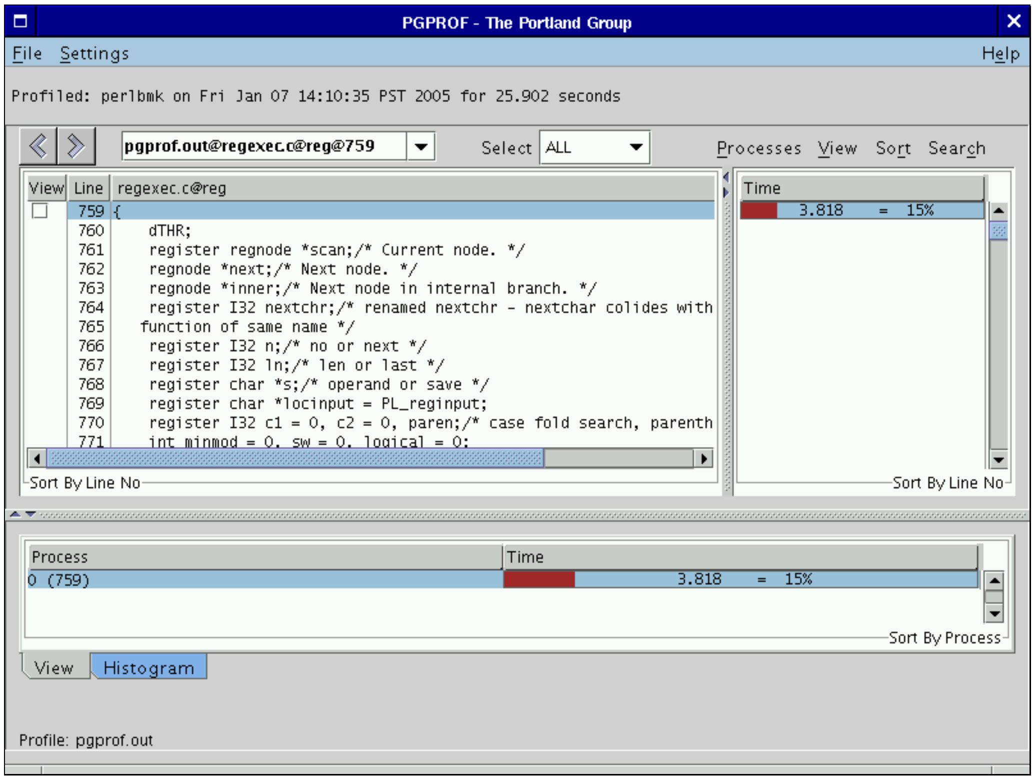
Sort By Time

Process	Time
0 (reg)	13.124 = 51%

Sort By Time

Profile: pgprof.out

Figure 2-5: Example Line Level Profile



Profiled: perlbmk on Fri Jan 07 14:10:35 PST 2005 for 25.902 seconds

pgprof.out@regexec.c@reg@759

```
759 {
760     dTHR;
761     register regnode *scan;/* Current node. */
762     regnode *next;/* Next node. */
763     regnode *inner;/* Next node in internal branch. */
764     register I32 nextchr;/* renamed nextchr - nextchar colides with
765    function of same name */
766     register I32 n;/* no or next */
767     register I32 ln;/* len or last */
768     register char *s;/* operand or save */
769     register char *locinput = PL_reginput;
770     register I32 c1 = 0, c2 = 0, paren;/* case fold search, parenth
771     int minmod = 0, sw = 0, logical = 0;
```

Time
3.818 = 15%

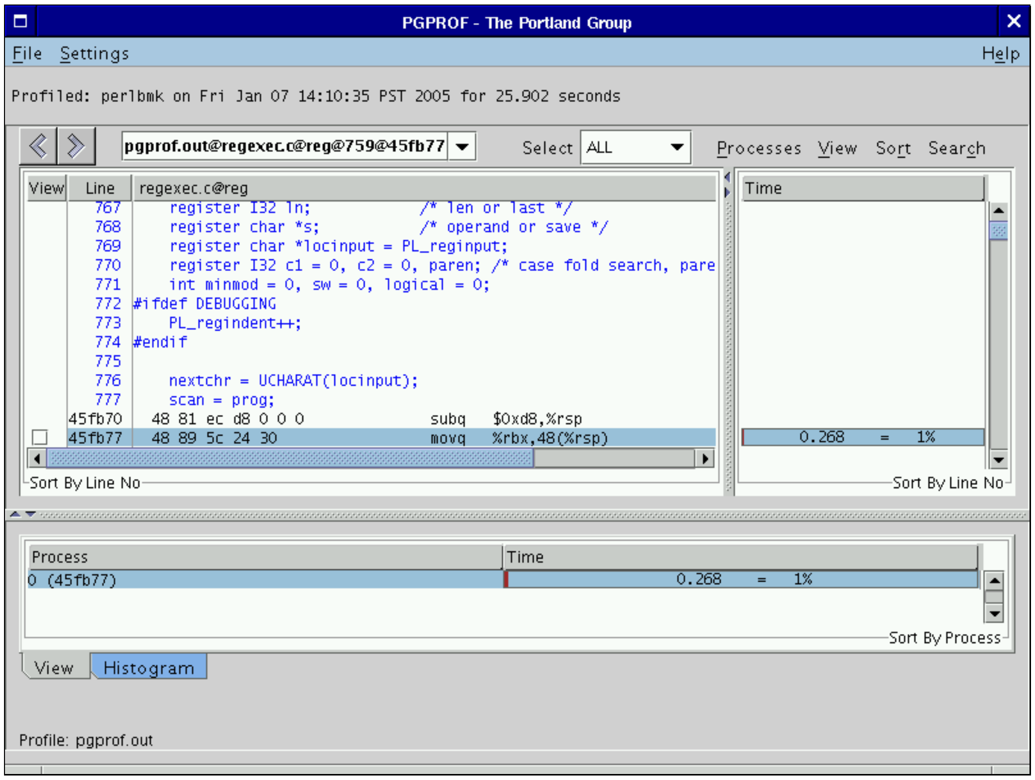
Sort By Line No

Process	Time
0 (759)	3.818 = 15%

Sort By Process

Profile: pgprof.out

Figure 2-6: Example Instruction Level Profile



```
Profiled: perlbmk on Fri Jan 07 14:10:35 PST 2005 for 25.902 seconds

pgprof.out@regexec.c@reg@759@45fb77

View  Line    regexec.c@reg                                                  Time
      767        register I32 ln;           /* len or last */
      768        register char *s;          /* operand or save */
      769        register char *locinput = PL_reginput;
      770        register I32 c1 = 0, c2 = 0, paren; /* case fold search, pare
      771        int minmod = 0, sw = 0, logical = 0;
      772   #ifdef DEBUGGING
      773        PL_regindent++;
      774   #endif
      775
      776        nextchr = UCHARAT(locinput);
      777        scan = prog;
      45fb70   48 81 ec d8 0 0 0            subq    $0xd8,%rsp
      45fb77   48 89 5c 24 30               movq    %rbx,48(%rsp)        0.268   =   1%
Sort By Line No                                                          Sort By Line No

Process                       Time
0 (45fb77)                    0.268   =   1%
                                                                         Sort By Process

Profile: pgprof.out
```

PGPROF Menus

As shown in Figures 2-1 through 2-4, there are five menus in the GUI: File, Settings, Help, Processes, View, and Sort. “File Menu” on page 166 through “Sort Menu” on page 178 describe each menu in detail. Keyboard shortcuts, when available, are listed next to menu items.

File Menu

The File menu contains the following items:

- New Window (control N) – Select this option to create a copy of the current profiler window on your screen.

- Open Profile… – Select this option to open another profile. After selecting this menu item, locate a profile data file in a file chooser dialog box. Select the new file in the dialog by double clicking it using the left mouse button. A new profile window will appear with the selected profile. Note: The name of the profile's executable must be set before opening a sample based profile (e.g., gmon.out). See the Set Executable… option below for more details.

- Set Executable… – Select this option to select the executable to be analyzed. Selection of this menu item launches a file selection dialog in which to locate the profiled executable. Select the executable by double clicking the left mouse button on it. When working with sample based profiles (e.g., gmon.out), the executable chosen must match the executable that generated the profile. By default, the profiler assumes that the executable is called a.out.

- Set Source Directory… – Select this option to set the location of the profiled executable's source files. The profiler displays a text field in a dialog box. Enter one or more directories in this text field. Each directory is separated by a path separator. The path separator is platform dependent. On Linux/Unix it is a colon (:), on Windows it is a semicolon (;). These directories act as a search path when the profiler cannot find a source file in the current directory. On Linux, for example:

```
../src:STEPS
```

- After entering the string above into the dialog box, the profiler will first search for source files in the current directory, then in the ../src directory, and finally in the STEPS directory. The directory can also be set through the –I command line option described in "Profiler Invocation and Initialization" on page 150. The same example for Windows follows:

```
..\src;STEPS
```

- Scalability Comparison… – Select this option to open another profile for scalability comparison. Follow the same directions for the Open Profile… option described above. The new profile will contain a Scale column in its Top Right table. You can also open one or more profiles for scalability comparison through the –scale command line option explained in "Profiler Invocation and Initialization" on page 150. See also "Scalability Comparison" on page 182 for more information on scalability.

- Print… – The print option is used to make a hard copy of the current profile data. The profiler will combine the data in all three profile tables and send the output to a printer. A printer dialog box will appear. A printer may be selected from the Print Service Name combo box. Click on the Print To File check box to send the output to a file. Other print options may be available. However, they are dependent on the specific printer and the Java Runtime Environment (JRE).

- Print to File… – Same as Print… option except the output goes to a file. After selecting this menu item, a save file dialog box will appear. Enter or choose an output file in the dialog box. Click Cancel to abort the print operation.

Settings Menu

The Settings menu contains the following items:

Bar Chart Colors… – This menu option will open a color chooser dialog box and a bar chart preview panel ("Bar Chart Color Dialog Box" on page 171). There are four bar chart colors based on the percentage filled and three bar chart attributes. The Filled Text Color attribute represents the text color inside the filled portion of the bar chart. The Unfilled Text Color attribute represents the text color outside the filled portion of the bar chart. The Background Color attribute represents the color of the unfilled portion of the bar chart. Table 2-1 , "Default Bar Chart Colors" lists the default colors.

To modify a bar chart or attribute color, click on its radio button. Next, choose a color from the Swatches, HSB, or RGB pane. Press the left mouse button on the OK button to accept the changes and close the dialog box. Click Reset to reset the selected bar chart or attribute to its previously selected color. Closing the window will also accept the changes. PGPROF will save color selections for subsequent runs unless the Save Settings on Exit box is unchecked (see discussion on this option below).

Font… – This menu option opens the Font Chooser dialog box (Figure 2-8 , "Font Chooser Dialog Box"). A new font may be chohsen from a list of fonts in this dialog's top combo box. A new font size may also be chosen from a list of sizes in this dialog's bottom combo box. The font is previewed in the Sample Text pane to the left. The font does not change until the OK button is selected. Click Cancel or close the dialog box to abort any changes. The default font is monospace size 12.

Show Tool Tips – Select this check box to enable tool tips. Tool tips are small temporary messages that pop-up when the mouse pointer is positioned over a component in the GUI. They provide additional information on what a particular component does. Unselect this check box to turn tool tips off.

Restore Factory Settings…– Use this option to restore the default look and feel of the GUI to the original settings. The PGPROF GUI will appear similar to the example in Figure 2-1 , "Profiler Window" after selecting this option.

Restore Saved Settings… – Use this option to restore the look and feel of the GUI to the previously saved settings. See the Save Settings on Exit option for more information.

Save Settings on Exit – When this check box is enabled, PGPROF will save the current look and feel settings on exit. These settings include the size of the main window, position of the horizontal/vertical dividers, the bar chart colors, the selected font, the tools tips preference, and the options selected in the View menu. When the PGPROF GUI is started again on the same host machine, these saved settings are used. To prevent saving these settings on exit, uncheck this box. Unchecking this box disables the saving of settings only for the current session.

Table 2-1: Default Bar Chart Colors

Bar Chart Style/Attribute	Default Color
1-25%	Brown
26-50%	Red
51%-75%	Orange
76%-100%	Yellow
Filled Text Color	Black
Unfilled Text Color	Black
Background Color	Grey

Figure 2-7: Bar Chart Color Dialog Box

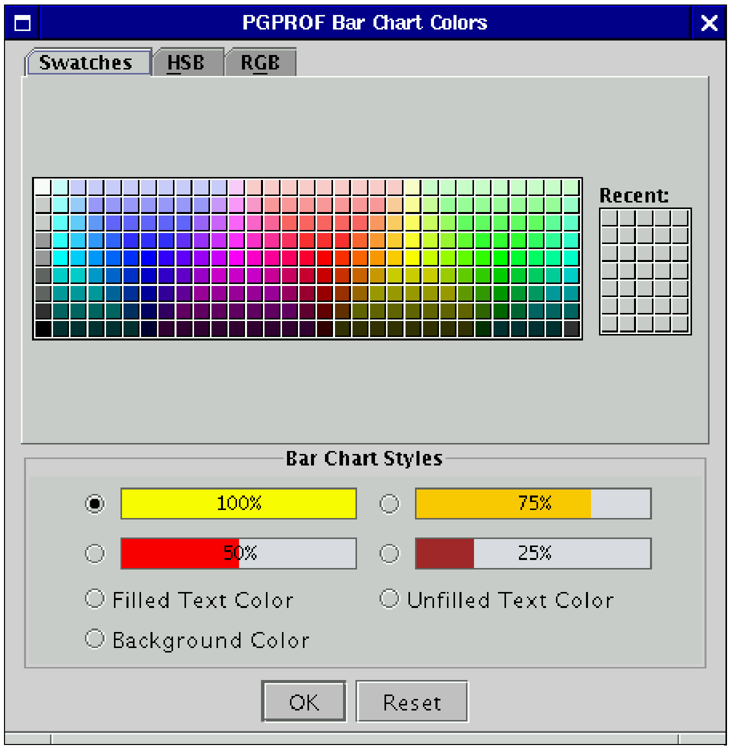
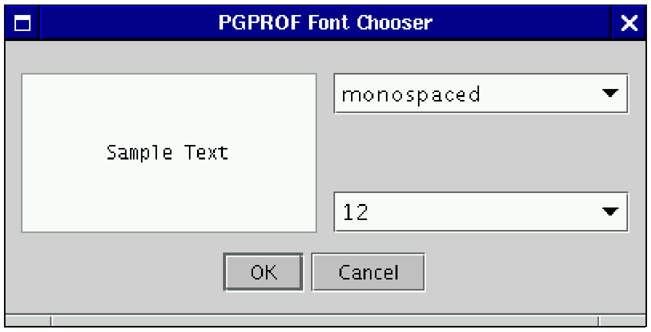


Figure 2-8: Font Chooser Dialog Box



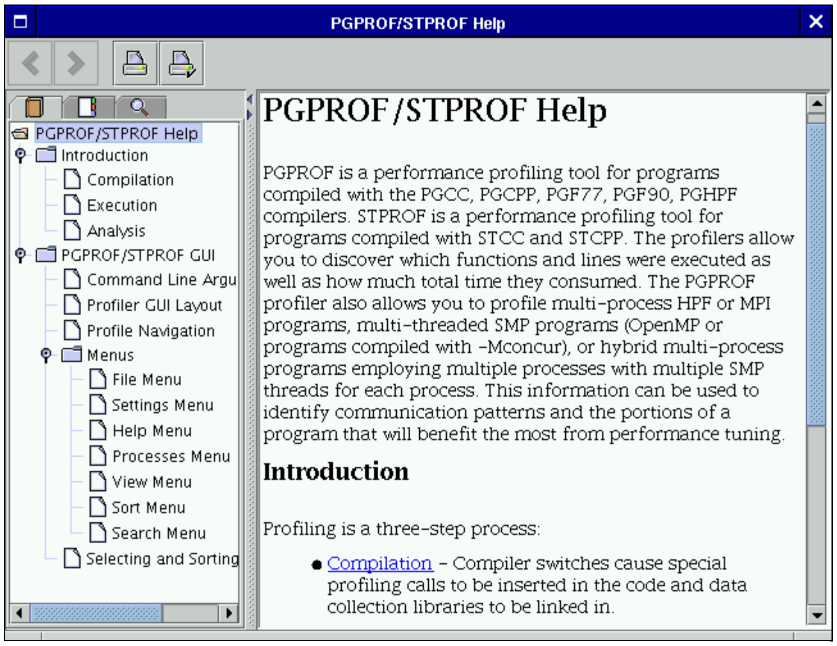
Help Menu

The Help menu contains the following items:

PGPROF Help… – This option invokes PGPROF's integrated help utility as shown in Figure 2-9 , "PGPROF Help". The help utility includes an abridged version of this manual. To find a help topic, use one of the follow tabs in the left panel: The "book" tab presents a table of contents, the "index" tab presents an index of commands, and the "magnifying glass" tab presents a search engine. Each help page (displayed on the right) may contain hyperlinks (denoted in underlined blue) to terms referenced elsewhere in the help engine. Use the arrow buttons to navigate between visited pages. Use the printer buttons to print the current help page.

About PGPROF… – This option opens a dialog box with version and contact information for PGPROF.

Figure 2-9: PGPROF Help



PGPROF/STPROF Help

PGPROF is a performance profiling tool for programs compiled with the PGCC, PGCPP, PGF77, PGF90, PGHPF compilers. STPROF is a performance profiling tool for programs compiled with STCC and STCPP. The profilers allow you to discover which functions and lines were executed as well as how much total time they consumed. The PGPROF profiler also allows you to profile multi-process HPF or MPI programs, multi-threaded SMP programs (OpenMP or programs compiled with -Mconcur), or hybrid multi-process programs employing multiple processes with multiple SMP threads for each process. This information can be used to identify communication patterns and the portions of a program that will benefit the most from performance tuning.

Introduction

Profiling is a three-step process:

- Compilation - Compiler switches cause special profiling calls to be inserted in the code and data collection libraries to be linked in.

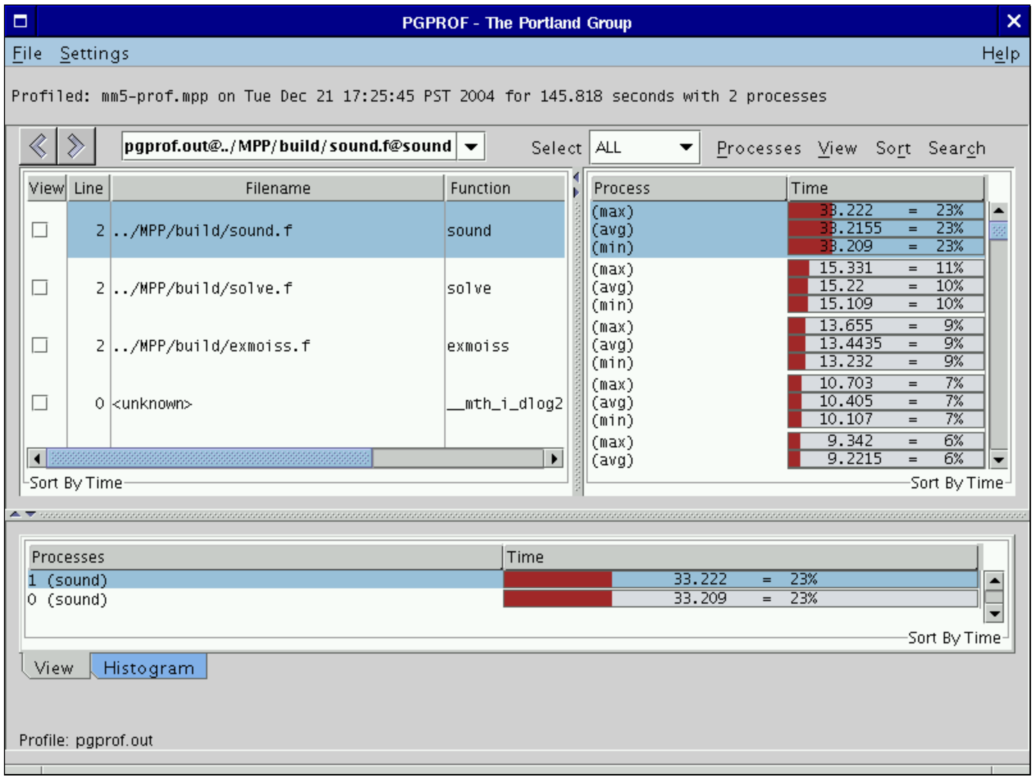
Processes Menu

The Processes menu is enabled for multi-process programs only. This menu contains three check boxes: Min, Max, and Avg. They represent the minimum process value, maximum process value, and average process value respectively. By default Max, is selected.

When Max is selected, the highest value for any profile data in the Top Right Table is reported. For example, when reporting Time, the longest time for each profile entry gets reported when Max is selected. When the Min process is selected, the lowest value for any profile data is reported in the Right Table. AVG reports the average value between all of the processes. Any, all, or none of these check boxes may be selected. When no check boxes are selected, the Top, Left and Right Tables are empty. If the Process check box under the View menu is selected, then each row of data in the Right Table is labeled max, avg, and min respectively.

Figure 2-10 , “PGPROF with Max, Avg, Min rows”, illustrates max, avg, and min with the Process check box enabled.

Figure 2-10: PGPROF with Max, Avg, Min rows


PGPROF - The Portland Group
File
Settings
Help
pgprof.out@../MPP/build/sound.f@sound
Select
ALL
Processes
View
Sort
Search
View
Histogram

Profiled: mm5-prof.mpp on Tue Dec 21 17:25:45 PST 2004 for 145.818 seconds with 2 processes

View	Line	Filename	Function
☐	2	../MPP/build/sound.f	sound
☐	2	../MPP/build/solve.f	solve
☐	2	../MPP/build/exmoiss.f	exmoiss
☐	0	<unknown>	__mth_i_dlog2

Sort By Time

Process	Time
(max)	33.222 = 23%
(avg)	33.2155 = 23%
(min)	33.209 = 23%
(max)	15.331 = 11%
(avg)	15.22 = 10%
(min)	15.109 = 10%
(max)	13.655 = 9%
(avg)	13.4435 = 9%
(min)	13.232 = 9%
(max)	10.703 = 7%
(avg)	10.405 = 7%
(min)	10.107 = 7%
(max)	9.342 = 6%
(avg)	9.2215 = 6%

Sort By Time

Processes	Time
1 (sound)	33.222 = 23%
0 (sound)	33.209 = 23%

Sort By Time

Profile: pgprof.out

View Menu

Use the View menu to select which columns of data to view in the Top Left, Top Right, and Bottom tables. This selection also affects the way that tables are printed to a file and a printer (see Print in "File Menu" on page 166).

The following lists View menu items and their definition. Note that not all items may be available for a given profile.

- Count – Enables the Count column in the Top Right and Bottom tables. Count is associated with the number of times this profile entry has been visited during execution of the program. For function level profiling, Count is the number of times the routine was called. For line level profiling, Count is the number of times a profiled source line was executed.

- Time – Enables the Time column in the Top Right and Bottom tables. The time column displays the time spent in a routine (for function level profiling) or at a profiled source line (for line level profiling).
- Cost – Enables the Cost column in the Top Right and Bottom tables. Cost is defined as the execution time spent in this routine and all of the routines that it called. The column will contain all zeros if cost information is not available for a given profile.
- Coverage – Enables the Cover column in the Top Right and Bottom tables. Coverage is defined as the number of lines in a profile entry that were executed. By definition, a profiled source line will always have a coverage of 1. A routine's coverage is equal to the sum of all its source line coverages. Coverage is only available for line level profiling. The column will contain all zeros if coverage information is not available for a given profile.
- Messages – Enables the message count columns in the Top Right and Bottom tables. Use this menu item to display total MPI messages sent and received for each profile entry. This menu item contains Message Sends and Message Receives submenus for separately displaying the sends and receives in the Top Right and Bottom tables. The message count columns will contain all zeros if no messages were sent or received for a given profile.
- Bytes – Same as Messages except message byte totals are displayed instead of counts.
- Scalability – Enables the Scale column in the Top Right table. Scalability is used to measure the linear speed-up or slow-down of two profiles. This menu contains two check boxes: Metric and Bar Chart. When Metric is selected, the raw Scalability value is displayed. When Bar Chart is selected, a graphical representation of the metric is displayed. Scalability is discussed in "Scalability Comparison" on page 182.
- Processes…(control P) – This menu item is enabled when profiling an application with more than one process. Use the Processes menu item to select individual processes for viewing in the Bottom table. When this item is selected, a dialog box will appear with a text field. Individual processes or a range of processes can be entered for viewing in this text field. Individual processes must be separated with a comma.

A range of processes must be entered in the form: [start]-[end]; where start represents the first process of the range and end represents the last process of the range. For example:

```
0,2-16,31
```

This tells the profiler to display information for process 0, process 2 through 16, and process 31. These changes remain active until they are changed again or the profiler session is terminated. Leave the text field blank to view all of the processes in the Bottom table.

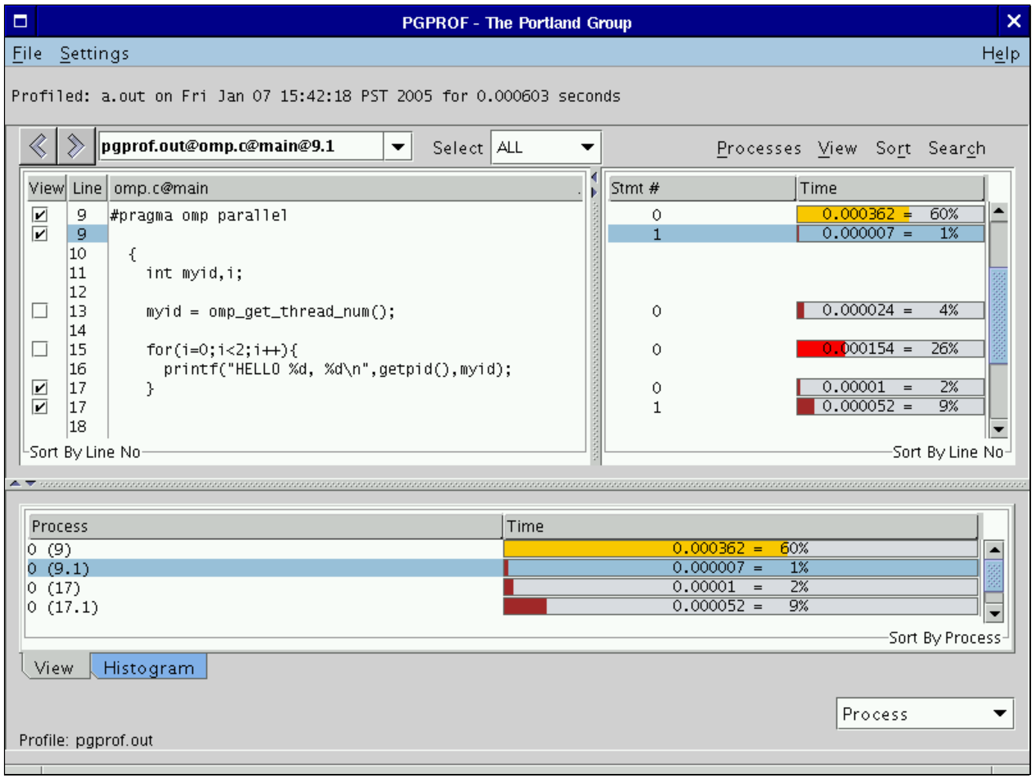
- Threads… (control T) – Same as Processes... except it selects the threads rather than the processes viewed in the Bottom table.
- Filename – Enables the Filename column in the Top Left table.
- Line Number – Enables the Line column in the Top Left table.
- Name – Enables the Function (routine) name column in the Top Left table when viewing function level profiling.
- Source – Enables the Source column in the Top Left table when viewing line level profiles. If the source code is available, this column will display the source lines for the current routine. Otherwise, this column will be blank.
- Statement Number – Enables the Stmt # column in the Top Right table. Sometimes more than one statement is profiled for a given source line number. One example of this is a "C" for statement. The profiler will assign a separate row for each substatement in the Top Left and Right tables. In line level profiling, duplicate line numbers display in the Line column. Each substatement is assigned a statement number starting at 0. Any substatement numbered one or higher will have a '.' and their statement number tacked onto the end of their profile address. For example, in Figure 2-11 , "Source Lines with Multiple Profile Entries", source lines 9 and 17 both have multiple profile entries. As shown in the Profile Entry Combo Box, the second entry for line 9 has the following address:

  ```
  pgprof.out@omp.c@main@9.1
  ```

- This line numbering convention is also reflected in the Bottom table of Figure 2-11 , "Source Lines with Multiple Profile Entries", where the line number is enclosed in parentheses.
- Process – This menu option is enabled when more than one process was profiled for a given application. When this check box is selected, a column labeled Process is displayed in the Top Right table. The values for the Process column depend on whatever was enabled in the Processes menu discussed in "Processes Menu" on page 173.

- Event1 – Event4 – If hardware event counters are supported on the profiled system, then up to four unique events can be displayed in the Top Right and Bottom tables. In this case, menu items for each counter will be enabled, with names corresponding to each particular event. Each event can exist for some or all of the executing threads in the profiled application.

Figure 2-11: Source Lines with Multiple Profile Entries



Profiled: a.out on Fri Jan 07 15:42:18 PST 2005 for 0.000603 seconds

View	Line	omp.c@main	Stmt #	Time
✓	9	`#pragma omp parallel`	0	0.000362 = 60%
✓	9		1	0.000007 = 1%
	10	`{`		
	11	`int myid,i;`		
	12			
☐	13	`myid = omp_get_thread_num();`	0	0.000024 = 4%
	14			
☐	15	`for(i=0;i<2;i++){`	0	0.000154 = 26%
	16	`printf("HELLO %d, %d\n",getpid(),myid);`		
✓	17	`}`	0	0.00001 = 2%
✓	17		1	0.000052 = 9%
	18			

Process	Time
0 (9)	0.000362 = 60%
0 (9.1)	0.000007 = 1%
0 (17)	0.00001 = 2%
0 (17.1)	0.000052 = 9%

Profile: pgprof.out

The submenus Count, Time, Cost, Coverage, Messages, Bytes, and Event1 through Event4 contain three check boxes for selecting how the data is presented in each column. The first check box enables a raw number to be displayed. The second check box enables a percentage. The third check box is a bar chart.

When a percentage is selected, a percentage based on the global value of the selected statistic displays. For example, in Figure 2-11 , "Source Lines with Multiple Profile Entries", line 13 consumed 0.000579 seconds, or 42% of the total time of 0.001391 seconds.

When the bar chart is selected, a graphical representation of the percentage is displayed. The colors are based on this percentage. For a list of default colors and their respective percentages, see the Bar Chart Colors option under the Settings menu ("Settings Menu" on page 168).

Sort Menu

The sort menu can be used to alter the order in which profile entries appear in the Top Left, Top Right, and Bottom tables. The current sort order is displayed at the bottom of each table. In Figure 2-11 , "Source Lines with Multiple Profile Entries", the tables have a "Sort by" clause followed with "Line No" or "Process". This indicates the sort order is by source line number or by process number respectively. In PGPROF, the default sort order is by Time for function level profiling and by Line No (source line number) for line level profiling. The sort is performed in descending order, from highest to lowest value, except when sorting by filename, function name, or line number. Filename, function name, and line number sorting is performed in ascending order; lowest to highest value. Sorting is explained in greater detail in "Selecting Profile Data" on page 179.

Search Menu

The search menu can be used to perform a text search within the Top Left table. The search menu contains the following items:

- Forward Search… (control F)
- Backward Search… (control B)
- Search Again (control G)
- Clear Search (control Q)

The PGPROF GUI displays a dialog box when you invoke the Forward Search… or Backward Search… menu items. The dialog box will prompt for the text to be located. Once the text is entered and the OK button selected, PGPROF will search for the text in the Top Left table. Select Cancel to abort the search. If Forward Search was selected, PGPROF will scroll forward to the next occurrence of the text entered in the dialog box. If Backward Search was selected, PGPROF will scroll backwards to the first previous occurrence of the text in the Top Left table. Top Left table columns that contain matching text are displayed in red. To repeat a search, select the Search Again menu item. To clear the search and turn the color of all matching text back to black, select the Clear Search menu item.

Selecting and Sorting Profile Data

Selecting and sorting affects what profile data is displayed and how it is displayed in PGPROF's Top Left, Top Right, and Bottom tables. The Sort menu, explained in "Sort Menu" on page 178, can be used to change the sort order. The sort order can also be changed by left-clicking a column heading in the Top Left, Top Right, and Bottom tables. The Select Combo Box, introduced in "Sort Menu" on page 178, may be used to select which profile entries are displayed based on certain criteria.

Selecting Profile Data

By default, PGPROF selects all profile entries for display in the Top Left and Right tables. To change the selection criteria, left mouse click on the Select Combo Box next to the Select label.

The following options are available:

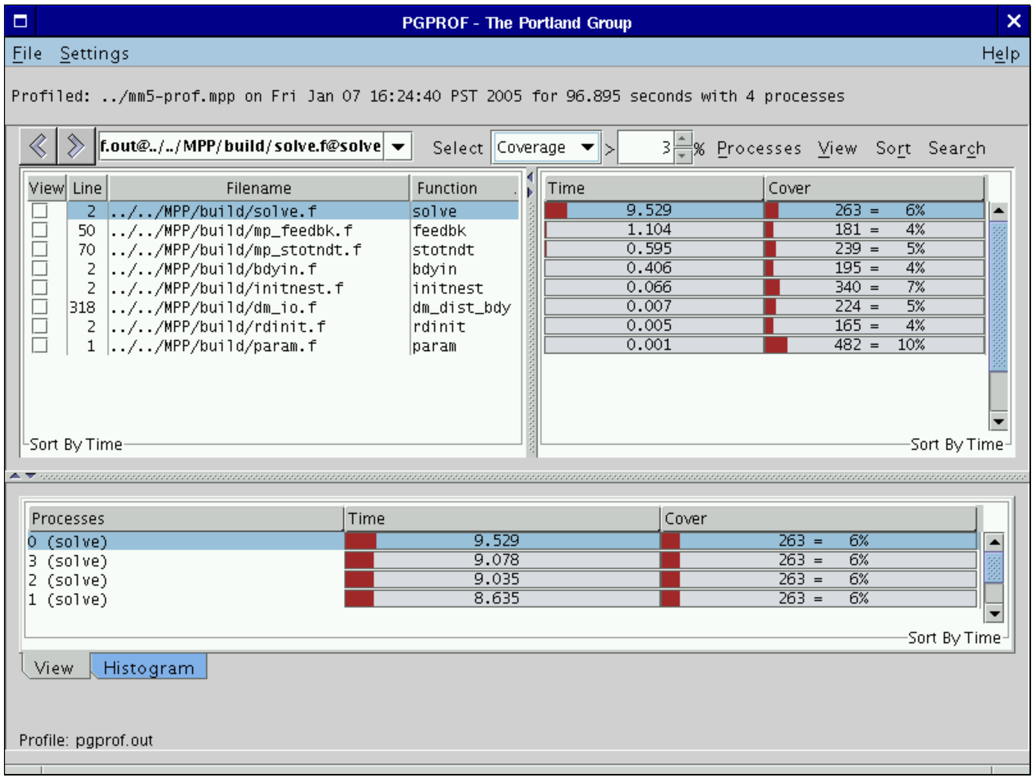
- All – Default. Display all profile entries.
- Coverage – Select entries based on Coverage. When Coverage is selected, an additional text field will appear with up and down arrow keys. Use the up and down arrow keys to increase the minimum coverage a profile entry needs before PGPROF will display it. The desired minimum may be entered directly into the text field. The value in the text field represents a percentage. Profile entries with coverage that exceed the input percentage are displayed in the tables. In Figure 2-12 , "Selecting Profile Entries with Coverage Greater Than 3%", the example shows selecting all routines that have coverage greater than 3% of the coverage for the entire program.
- Count – Select entries based on a count criteria. This is the same as Coverage except this selects the minimum count required for each profile entry. Profile entries with counts greater than the entered count value are displayed in the tables.
- Profiled – Select all entries in the Top Left table that have a corresponding entry in the Top Right table. See the discussion below for more information.
- Time – Same as Coverage except the criteria is based on percent of Time a profile entry consumes rather than Coverage.
- Unprofiled – Select all entries in the Top Left table that do not have a corresponding entry in the Top Right table. See the discussion below for more information.

NOTE

For applications compiled with –Mprof=hwcts or –Mprof=mpi,hwcts, a hardware event may be selected from the list above as well.

When Profiled is selected , profile entries that have a corresponding entry in both the Top Left and Right tables are selected. A profile entry may be listed in the Top Left table but not in the Top Right table. In this case, the entry is an Unprofiled entry. A Profiled entry is a point in the program in which profile information was collected. Depending on the profiling method used, this could be at the end of a basic block (e.g., –Mprof=[func | line], –Mprof=mpi,[func | lines] instrumented profiles) or when the profiling mechanism saved its state (e.g., –pg, –Mprof=time, –Mprof=hwcts, –Mprof=mpi,[time, hwcts] sample based profiles).

Figure 2-12: Selecting Profile Entries with Coverage Greater Than 3%


PGPROF - The Portland Group
File Settings Help
f.out@../../MPP/build/solve.f@solve
Select Coverage > 3 % Processes View Sort Search
View Histogram

```
Profiled: ../mm5-prof.mpp on Fri Jan 07 16:24:40 PST 2005 for 96.895 seconds with 4 processes
```

View	Line	Filename	Function	Time	Cover
☐	2	../../MPP/build/solve.f	solve	9.529	263 = 6%
☐	50	../../MPP/build/mp_feedbk.f	feedbk	1.104	181 = 4%
☐	70	../../MPP/build/mp_stotndt.f	stotndt	0.595	239 = 5%
☐	2	../../MPP/build/bdyin.f	bdyin	0.406	195 = 4%
☐	2	../../MPP/build/initnest.f	initnest	0.066	340 = 7%
☐	318	../../MPP/build/dm_io.f	dm_dist_bdy	0.007	224 = 5%
☐	2	../../MPP/build/rdinit.f	rdinit	0.005	165 = 4%
☐	1	../../MPP/build/param.f	param	0.001	482 = 10%

Sort By Time   Sort By Time

Processes	Time	Cover
0 (solve)	9.529	263 = 6%
3 (solve)	9.078	263 = 6%
2 (solve)	9.035	263 = 6%
1 (solve)	8.635	263 = 6%

Sort By Time

Profile: pgprof.out

Sorting Profile Data

The current sort order is displayed at the bottom of each table. For example, the message Sort By Time is present at the bottom of each table in Figure 2-12 , "Selecting Profile Entries with Coverage Greater Than 3%". The Bottom table will display one of the following messages when sorting by Filename, Name, or Line Number:

- Sort By Process
- Sort By Processes
- Sort By Threads
- Sort By Process.Threads

- Sort By Processes.Threads

If one of these messages appears in the Bottom table, then the profiler is treating the process/thread number as the major sort key and the Filename, Name, or Line Number as the minor sort key. This can be used to easily compare two different profile entries with the same process/thread number. Use the check boxes under the View column in the Top Left table to compare more than one profile entry in the Bottom table. This is demonstrated in Figure 2-11 , "Source Lines with Multiple Profile Entries".

Scalability Comparison

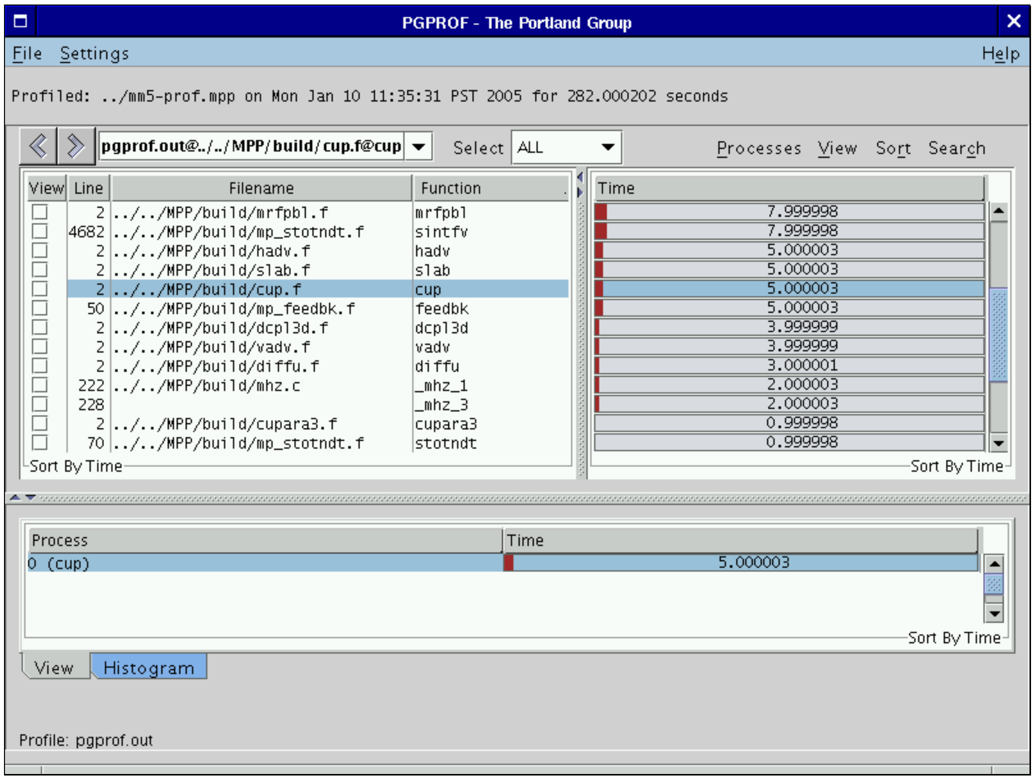
PGPROF has a Scalability Comparison feature that can be used to measure linear speed-up or slow-down between multiple executions of an application. Scalability between executions can be measured with a varying number of processes or threads. To use scalability comparison, first generate two or more profiles for a given application. For best results, compare profiles from the same application using the same input data. Also, scalability comparison works best for serial or multi-process (MPI) programs. To measure scalability for a multi-threaded program, profiling with –Mprof=func or –Mprof=lines is recommended. See "Profiling Multi-threaded Programs" on page 146 and "Measuring Time" on page 152 for more information on multi-threaded profiling.

The number of processes and/or threads used in each execution can be different. After generating two or more profiles, load one of them into PGPROF. Select the Scalability Comparison item under the File menu and choose another profile for comparison ("File Menu" on page 166). A new profiler window will appear with a column called Scale in its Top Right table ("View Menu" on page 174).

Figure 2-13 , "Profile of an Application Run with 1 Process" shows a profile of an application that was run with one process. Figure 2-14 , "Profile with Visible Scale Column", shows a profile of the same application run with two processes. The profile in Figure 2-14 , "Profile with Visible Scale Column", also has a Scale column in its Top Right table. Each profile entry that has timing information has a Scale value. The scale value measures the linear speed-up or slow-down for these entries across profiles. A scale value of zero (or one for serial/multi-threaded programs) indicates no change in the execution time between the two runs. A positive value means the time improved by that scaled factor. A negative value means that the time slowed down by that scaled factor.

Bar charts in the Scale column show positive values with bars extending from left to right and negative values with bars extending from right to left (Figure 2-14 , "Profile with Visible Scale Column"). If there is a question mark ('?') in the Scale column, then PGPROF is unable to perform the scalability comparison for this profile entry. This may happen if the two profiles do not share the same executable or input data.

Figure 2-13: Profile of an Application Run with 1 Process


PGPROF - The Portland Group
File Settings Help
pgprof.out@../../MPP/build/cup.f@cup
Select ALL
Processes View Sort Search
View Histogram

```
Profiled: ../mm5-prof.mpp on Mon Jan 10 11:35:31 PST 2005 for 282.000202 seconds
```

View	Line	Filename	Function	Time
☐	2	../../MPP/build/mrfpbl.f	mrfpbl	7.999998
☐	4682	../../MPP/build/mp_stotndt.f	sintfv	7.999998
☐	2	../../MPP/build/hadv.f	hadv	5.000003
☐	2	../../MPP/build/slab.f	slab	5.000003
☐	2	../../MPP/build/cup.f	cup	5.000003
☐	50	../../MPP/build/mp_feedbk.f	feedbk	5.000003
☐	2	../../MPP/build/dcpl3d.f	dcpl3d	3.999999
☐	2	../../MPP/build/vadv.f	vadv	3.999999
☐	2	../../MPP/build/diffu.f	diffu	3.000001
☐	222	../../MPP/build/mhz.c	_mhz_1	2.000003
☐	228		_mhz_3	2.000003
☐	2	../../MPP/build/cupara3.f	cupara3	0.999998
☐	70	../../MPP/build/mp_stotndt.f	stotndt	0.999998

Sort By Time

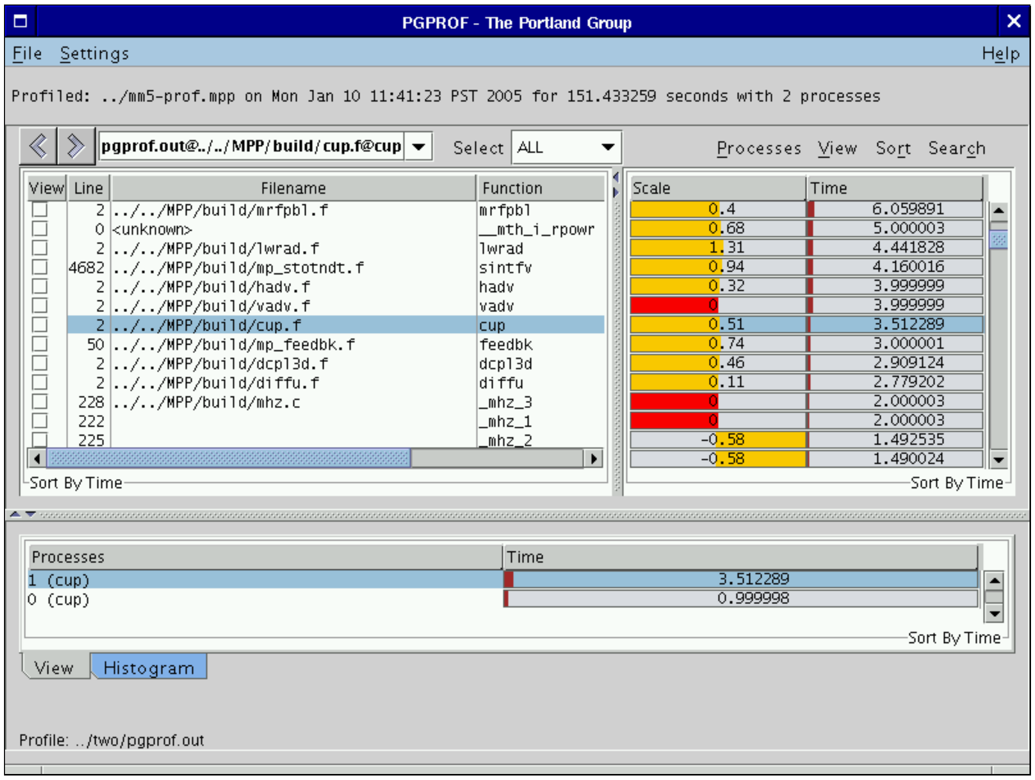
Sort By Time

Process	Time
0 (cup)	5.000003

Sort By Time

Profile: pgprof.out

Figure 2-14: Profile with Visible Scale Column



Profiled: ../mm5-prof.mpp on Mon Jan 10 11:41:23 PST 2005 for 151.433259 seconds with 2 processes

pgprof.out@../../MPP/build/cup.f@cup

View	Line	Filename	Function	Scale	Time
☐	2	../../MPP/build/mrfpbl.f	mrfpbl	0.4	6.059891
☐	0	<unknown>	__mth_i_rpowr	0.68	5.000003
☐	2	../../MPP/build/lwrad.f	lwrad	1.31	4.441828
☐	4682	../../MPP/build/mp_stotndt.f	sintfv	0.94	4.160016
☐	2	../../MPP/build/hadv.f	hadv	0.32	3.999999
☐	2	../../MPP/build/vadv.f	vadv	0	3.999999
☐	2	../../MPP/build/cup.f	cup	0.51	3.512289
☐	50	../../MPP/build/mp_feedbk.f	feedbk	0.74	3.000001
☐	2	../../MPP/build/dcpl3d.f	dcpl3d	0.46	2.909124
☐	2	../../MPP/build/diffu.f	diffu	0.11	2.779202
☐	228	../../MPP/build/mhz.c	_mhz_3	0	2.000003
☐	222		_mhz_1	0	2.000003
☐	225		_mhz_2	-0.58	1.492535
				-0.58	1.490024

Sort By Time   Sort By Time

Processes	Time
1 (cup)	3.512289
0 (cup)	0.999998

Sort By Time

Profile: ../two/pgprof.out

PGPROF uses the two formulas shown below for computing scalability. Formula 2-1 computes scalability for multiprocess programs and Formula 2-2 computes scalability for serial and multi-threaded programs.

In Formula 2-1, a scalability value greater than zero indicates some degree of speed-up. A scalability value of one indicates perfect linear speed-up. Anything greater than one, indicates super speed-up. Similar negative values indicate linear slow-down and super slow-down respectively. A value of zero indicates that no change in execution time occurred between the two runs.

Formula 2-1: Scalability for Multiprocess Programs

```
P1 = number of processes used in first run of
application
P2 = number of processes used in second run of application
where P2 > P1
Time1 = Execution time using P1 processes
Time2 = Execution time using P2 processes
Scalability = log( Time1 ÷ Time2) ÷ log( P2 ÷ P1 )
```

In Formula 2-2, a scalability value greater than zero indicates some degree of speed-up. A scalability value equal to the ratio (T2 / T1) indicates perfect linear speed-up. Anything greater than one, indicates super speed-up. Similar negative values indicate linear slow-down and super slow-down respectively. A value of one indicates that no change in execution time occurred between the two runs.

Formula 2-2: Scalability for Serial and Multi-threaded Programs

```
T1 = number of threads used in first run of application
T2 = number of threads used in second run of application
where T2 ≥ T1
Time1 = Execution time using T1 threads
Time2 = Execution time using T2 threads
Scalability = Time2 ÷ Time1
```

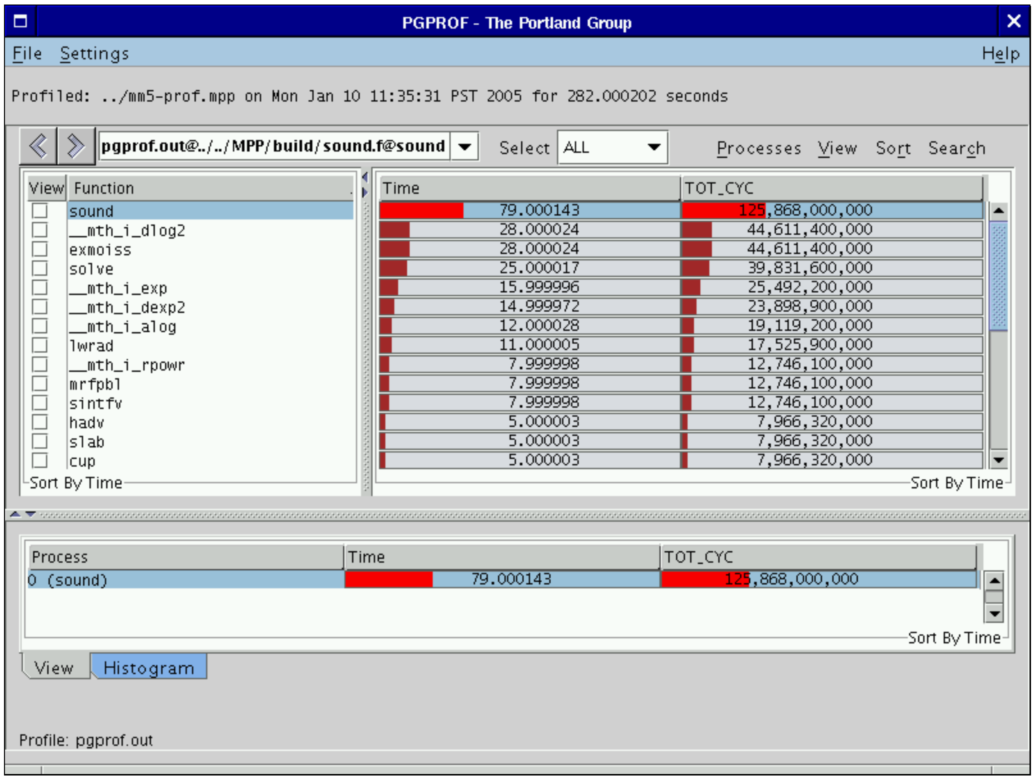
Viewing Profiles with Hardware Event Counters

If you executed your program under the control of pgprof -collect or if you compiled your program with the -Mprof=hwcts or the -Mprof=mpi,hwcts option, then you can profile up to four event counters and view them in PGPROF (use -Mprof=dwarf or -Mprof=mpi,dwarf option when generating profiles via the OProfile interface). See “Profiling with Hardware Event Counters (Linux Only)” on page 146 for more details.

Figure 2-15 , “Profile with Hardware Event Counter”, shows a profile of one event counter called TOT_CYC, which counts the number of CPU cycles the program consumed. This event is enabled by default or by adding PAPI_TOT_CYC to your PGPROF_EVENTS environment variable (“Profiling with Hardware Event Counters (Linux Only)” on page 146). Each entry under the Time column represents CPU time computed by its corresponding TOT_CYC entry. PGPROF will not report any time for hardware counter profiles unless one of the hardware events is PAPI_TOT_CYC.

Each event can be toggled for viewing and sorting under the View (“View Menu” on page 174) and Sort (“Sort Menu” on page 178) menus respectively. Hardware event criteria may also be selected under the Select combo box (“Selecting Profile Data” on page 179).

Figure 2-15: Profile with Hardware Event Counter



PGPROF - The Portland Group

File Settings Help

Profiled: ../mm5-prof.mpp on Mon Jan 10 11:35:31 PST 2005 for 282.000202 seconds

pgprof.out@../../MPP/build/sound.f@sound Select ALL Processes View Sort Search

View	Function	Time	TOT_CYC
☐	sound	79.000143	125,868,000,000
☐	__mth_i_dlog2	28.000024	44,611,400,000
☐	exmoiss	28.000024	44,611,400,000
☐	solve	25.000017	39,831,600,000
☐	__mth_i_exp	15.999996	25,492,200,000
☐	__mth_i_dexp2	14.999972	23,898,900,000
☐	__mth_i_alog	12.000028	19,119,200,000
☐	lwrad	11.000005	17,525,900,000
☐	__mth_i_rpowr	7.999998	12,746,100,000
☐	mrfpbl	7.999998	12,746,100,000
☐	sintfv	7.999998	12,746,100,000
☐	hadv	5.000003	7,966,320,000
☐	slab	5.000003	7,966,320,000
☐	cup	5.000003	7,966,320,000

Sort By Time Sort By Time

Process	Time	TOT_CYC
0 (sound)	79.000143	125,868,000,000

Sort By Time

View Histogram

Profile: pgprof.out

Command Language

The user interface for non-GUI (Win32) versions of the PGPROF profiler is a simple command language. This command language is available in GUI versions of the profiler using the –s or –text option. The language is composed of commands and arguments separated by white space. A pgprof> prompt is issued unless input is being redirected.

Command Usage

This section describes the profiler's command set. Command names are printed in bold and may be abbreviated as indicated. Arguments enclosed by brackets ('['']') are optional. Separating two or more arguments by '|' indicates that any one is acceptable. Argument names in italics are chosen to indicate what kind of argument is expected. Argument names that are not in italics are keywords and should be entered as they appear.

display

```
d[isplay] [display options] | all | none
```

Specify display information. This includes information on minimum values, maximum values, average values, or per processor/thread data. Below is a list of possible display options:

[no]calls [no]cover [no]time [no]timecall [no]cost [no]proc [no]thread [no]msgs [no]msgs_sent [no]msgs_recv [no]bytes [no]bytes_sent [no]name [no]file [no]line [no]lineno [no]visits [no]scale [no]stmtno

help

```
he[lp] [command]
```

Provide brief command synopsis. If the command argument is present only information for that command will be displayed. The character "?" may be used as an alias for help.

history

```
h[istory] [ size ]
```

Display the history list, which stores previous commands in a manner similar to that available with csh or dbx. The optional size argument specifies the number of lines to store in the history list.

lines

```
l[ines] function [[>] filename]
```

Print (display) the line level data together with the source for the specified function. If the filename argument is present, the output will be placed in the named file. The '>' means redirect output, and is optional.

asm

```
a[sm] routine [[>] filename]
```

Print (display) the instruction and line level data together with the source and assembly for the specified routine. If the filename argument is present, the output will be placed in the named file. The '>' means redirect output, and is optional. This command is only available on platforms that support instruction level profiling.

load

```
lo[ad] [ datafile]
```

Load a new dataset. With no arguments reloads the current dataset. A single argument is interpreted as a new data file. With two arguments, the first is interpreted as the program and the second as the data file.

merge

```
m[erge] datafile
```

Merge the profile data from the named datafile into the current loaded dataset. The datafile must be in standard pgprof.out format, and must have been generated by the same executable file as the original dataset (no datafiles are modified.)

process

```
pro[cess] processor_num
```

For multi-process profiles, specify the processor number of the data to display.

print

```
p[rint] [[>] filename]
```

Print (display) the currently selected function data. If the filename argument is present, the output will be placed in the named file. The '>' means redirect output, and is optional.

quit

```
q[uit]
```

Exit the profiler.

select

```
sel[ect] calls | timecall | time | cost | cover | all [[>] cutoff]
```

Display data for a selected subset of the functions. This command is used to set the selection key and establish a cutoff percentage or value. The cutoff value must be a positive integer, and for time related fields is interpreted as a percentage. The '>' means greater than, and is optional. The default is all.

sort

```
so[rt] [by] [max | avg | min | proc | thread] calls | cover | timecall | time
| cost | name | msgs | msgs_sent | msgs_recv | bytes | bytes_sent | bytes_recv
| visits | file
```

Function level data is displayed as a sorted list. This command establishes the basis for sorting. The default is max time.

srcdir

```
src[dir] directory
```

Set the source file search path.

stat

```
s[tat] [no]min|[no]avg|[no]max|[no]proc|[no]thread|[no]all]
```

Set which process fields to display (or not to display when using the arguments beginning with “no”).

thread

```
th[read] thread_num
```

Specify a thread for a multi-threaded process profile.

times

```
t[imes] raw | pct
```

Specify whether time-related values should be displayed as raw numbers or as percentages. The default is pct.

! (history)

```
!!
```

Repeat previous command.

```
! num
```

Repeat previous command numbered num in the history list.

```
!-num
```

Repeat the num-th previous command numbered num in the history list.

```
! string
```

Repeat most recent command starting with string from the history list.

Index