PGI® Tools Guide

Parallel Tools for Scientists and Engineers

The Portland Group®
STMicroelectronics
Two Centerpointe Drive
Lake Oswego, OR 97035

PGI® Tools Guide

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
Seventh Printing: Release 7.1, October, 2007
Eighth Printing: Release 7.2 May, 2008
Ninth Printing: Release 8.0 November, 2008
Tenth Printing: Release 9.0 June, 2009
Eleventh Printing: Release 2010, version10.0, November, 2009
Twelfth Printing: Release 2010, version10.4, April, 2010

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

ID: 10841215

Contents

Figures

Tables

Examples

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 Intel 64 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 Intel 64 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 docs/index.htm in the PGI installation directory. Typically the value of the environment variable PGI is set to the PGI installation directory. 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 PGCPP 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:

Part I: The PGDBG Debugger contains these thirteen chapters that describe PGDBG, a symbolic debugger for Fortran, C, C++ and assembly language programs.

Chapter 1, "*Getting Started with the PGDBG Debugger*"
contains information on how to start using the debugger, including a description of how to build a target application for debug and how to invoke PGDBG.

Chapter 2, "*The PGDBG Graphical User Interface*"
describes how to use the PGDBG graphical user interface (GUI).

Chapter 3, "*PGDBG Command Line Options*"
describes the PGDBG command-line options.

Chapter 4, "*PGDBG Command Language*"
provides 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.

Chapter 5, "*PGDBG Command Summary*"
provides a brief summary table of the PGDBG debugger commands, providing a brief description of the command as well as information about the category of command use.

Chapter 6, "*PGDBG Assembly-Level Debugging*"
contains information on assembly-level debugging; basic debugger operations, commands, and features that are useful for debugging assembly code; and how to access registers.

Chapter 7, "*PGDBG Source-Level Debugging*"
contains information on language-specific issues related to source debugging.

Chapter 8, "*PGDBG Platform-Specific Features*"
contains platform-specific information as it relates to debugging.

Chapter 9, "*PGDBG Parallel Debugging Overview*"
contains an overview of the parallel debugging capabilities of PGDBG.

Chapter 10, "*PGDBG - Parallel Debugging with OpenMP*"
describes the parallel debugging capabilities of PGDBG and how to use them with OpenMP.

Chapter 11, "*PGDBG Parallel Debugging with MPI*"
describes the parallel debugging capabilities of PGDBG and how to use them with MPI.

Chapter 12, "*PGDBG Parallel Debugging of Hybrid Applications*"
describes the parallel debugging capabilities of PGDBG and how to use them with hybrid applications.

Chapter 13, "*PGDBG Command Reference*"
provides reference information about each of the PGDBG commands, separated by area of use.

Part II: The PGPROF Profiler contains these ten chapters that describe the PGPROF Profiler, a tool for analyzing the performance characteristics of C, C++, F77, and F95 programs.

Chapter 14, "*Getting Started with the PGPROF Profiler*"
contains information on how to start using the profiler, including a description of the profiling process, information specific to certain how to profile MPI and OpenMP programs and how to profile with hardware event counters.

Chapter 15, "*Using PGPROF*"
describes how to use the PGPROF graphical user interface (GUI).

Chapter 16, "*Compiler Options for Profiling*"
describes the compiler options available for profiling.

Chapter 17, "*PGPROF Command Line Options*"
describes the PGPROF command-line options and provides sample invocations with descriptions.

Chapter 18, "*PGPROF Environment Variables*"
contains information on environment variables that you can set to control the way profiling is performed in PGPROF.

Chapter 19, "*PGPROF Data and Precision*"
contains descriptions of the profiling mechanism that measures time, how statistics are collected, and the precision of the profiling results.

Chapter 20, "*PGPROF Reference*"
provides reference information about the PGPROF graphical user interface, including information about the menus, the toolbars, and the subwindows.

Chapter 21, "*The PGPROF Command Line Interface*"
provides information about the PGPROF command language, giving reference information about each of the PGPROF commands, separated by area of use.

Chapter 22, "*pgcollect Reference*"
: provides reference information about the **pgcollect** command.

Conventions

This guide uses the following conventions:

italic
: is used for emphasis.

`Constant Width`
: is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
: is used for commands.

[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.

Terminology

If there are terms in this guide with which you are unfamiliar, PGI provides a glossary of terms which you can access at *www.pgroup.com/support/definitions.htm*

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/
- PAPI (Performance Application Program Interface) http://icl.cs.utk.edu/papi/

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 64 or Core2 processor
- Intel x86 (and compatible), AMD Athlon or AMD64, or Intel 64 or Core2 processor

Part I. PGDBG Debugger

Part I of the *PGI Tools Guide* is about the PGDBG debugger. The information in this part describes PGDBG, a symbolic debugger for Fortran, C, C++, and assembly language programs.

- Chapter 1, “Getting Started with the PGDBG Debugger,” starting on page 1, contains information on how to start using the debugger, including a description of how to build a target application for debug, and how to invoke PGDBG.
- Chapter 2, “The PGDBG Graphical User Interface,” starting on page 5, describes how to use the PGDBG graphical user interface (GUI).
- Chapter 3, “PGDBG Command Line Options,” starting on page 25, describes the PGDBG command-line options and how they are interpreted.
- Chapter 4, “PGDBG Command Language,” starting on page 27, provides 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.
- Chapter 5, “PGDBG Command Summary,” starting on page 37, provides a brief, alphabetical, summary table of the PGDBG debugger commands. The table includes a brief description of the command as well as information about the category of command use.
- Chapter 6, “PGDBG Assembly-Level Debugging,” starting on page 49, contains information on machine-level debugging; basic debugger operations, commands, and features that are useful for debugging assembly code, as well as information on how to access registers.
- Chapter 7, “PGDBG Source-Level Debugging,” starting on page 55, contains information on language-specific issues related to source debugging.
- Chapter 8, “PGDBG Platform-Specific Features,” starting on page 61, contains platform-specific information as it relates to debugging.
- Chapter 9, “PGDBG Parallel Debugging Overview,” starting on page 65, contains an overview of the parallel debugging capabilities of PGDBG.
- Chapter 10, “PGDBG - Parallel Debugging with OpenMP,” starting on page 83, describes how to use the parallel debugging capabilities of PGDBG with OpenMP.
- Chapter 11, “PGDBG Parallel Debugging with MPI,” starting on page 87, describes how to use the parallel debugging capabilities of PGDBG with MPI.
- Chapter 12, “PGDBG Parallel Debugging of Hybrid Applications,” starting on page 95, describes how to use the parallel debugging capabilities of PGDBG in multilevel debug mode on hybrid applications.
- Chapter 13, “PGDBG Command Reference,” starting on page 97, provides reference information about each of the PGDBG commands, grouping the commands by category of use.

Chapter 1. Getting Started with the PGDBG Debugger

PGDBG is a symbolic debugger for Fortran, C, C++ and assembly language programs. It provides debugger features, such as execution control using breakpoints, single-stepping, and examination and modification of application variables, memory locations, and registers.

PGDBG supports debugging of certain types of parallel applications:

- Multi-threaded and OpenMP applications.
- MPI applications
- Hybrid applications, which use multiple threads or OpenMP as well as multiple MPI processes on Linux clusters.

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

Throughout Part I of this manual we use the term *host* to refer to the system on which PGDBG executes, *target*, to refer to the program being debugged, and *target machine* to refer to the system on which the target runs. For more detailed definitions of these terms, refer to the PGI glossary of terms which you can access at *www.pgroup.com/support/definitions.htm*.

For an introduction to terminology used to describe parallel debugging, refer to Chapter 9, "*PGDBG Parallel Debugging Overview*".

Building Applications for Debug

To build an application for debug, compile with the `-g` option. With this option, the compiler generates information about the symbols and source files in the program and includes it in the executable file. The option `-g` 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.

Debugging Optimized Code

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. Assembly-level debugging (e.g., accessing registers, viewing assembly code, etc.) is reliable, even with optimized code. Programs built without –g can be debugged; however, information about types, local variables, arguments and source file line numbers are not available. For more information on assembly-level debugging, refer to Chapter 6, "*PGDBG Assembly-Level Debugging*".

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.

Building for Debug on Windows

To build an application for debug on Windows platforms, applications must be linked with the `-g` option as well as compiled with`-g`. This process results 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; except for special circumstances, the linker should not 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*" and "*PGDBG Command Reference*" describe in detail how to use the PGDBG command-line interface. "*The PGDBG Graphical User Interface*" 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 Chapter 3, "*PGDBG Command Line Options*". See Chapter 11, "*PGDBG Parallel Debugging with MPI*" for instructions on how to debug an MPI program [Linux and Windows CCS 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 starts the PGDBG Graphical User Interface (GUI) (See "PGDBG Graphical User Interface," on page 3). For users who prefer to use a command-line interface, PGDBG may be invoked with the –text parameter, as described in Chapter 3, "*PGDBG Command Line Options*" and "PGDBG Command Language," on page 3.

Note

The command shell interprets any I/O redirection specified on the PGDBG command line. Refer to "Process Control," on page 87 for a description of how to redirect I/O using the **run** command.

PGDBG supports both 32-bit and 64-bit applications. If the PATH environment variable is set to use the 32-bit PGI tools, a 64-bit application can be debugged by invoking PGDBG normally. PGDBG automatically determines the architecture of the target program and configures itself for 64-bit debugging. Conversely, you can debug a 32-bit application by invoking PGDBG with the PATH set to invoke the 64-bit PGI tools. For more details, refer to Chapter 3, "*PGDBG Command Line Options*".

Initializing PGDBG

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. An initialization file is useful for defining common aliases, setting breakpoints, and for other startup commands.

If an initialization file named .pgdbgrc exists in the current directory or in the home directory, defined by the environment variable HOME, PGDBG opens this file and executes the commands in it.

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.

Starting a Session

Once PGDBG is invoked and the initialization file is processed, 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`.

Using Command Line Options

PGDBG can interpret command-line options. For information on these options and how they are interpreted, refer to Chapter 3, "*PGDBG Command Line Options*".

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. For more information on the PGDBG GUI, refer to "*The PGDBG Graphical User Interface*".

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. For more information on the command language, refer to "*PGDBG Command Language*" and "*PGDBG Command Reference*".

Troubleshooting

If you are having trouble during invocation or the initialization process, use the following sections for tips on what might be causing your problem.

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 minimum version or above. By default, PGDBG uses the version of Java installed with your PGI software; if you chose not to install Java when installing your PGI software, PGDBG looks for Java on your PATH. Both of these can be overridden 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, use this command to specify the location of Java:

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

Chapter 2. The 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 2.1, "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 910x850. 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. For information on the Settings menu, refer to "Main Window Menus," on page 11.

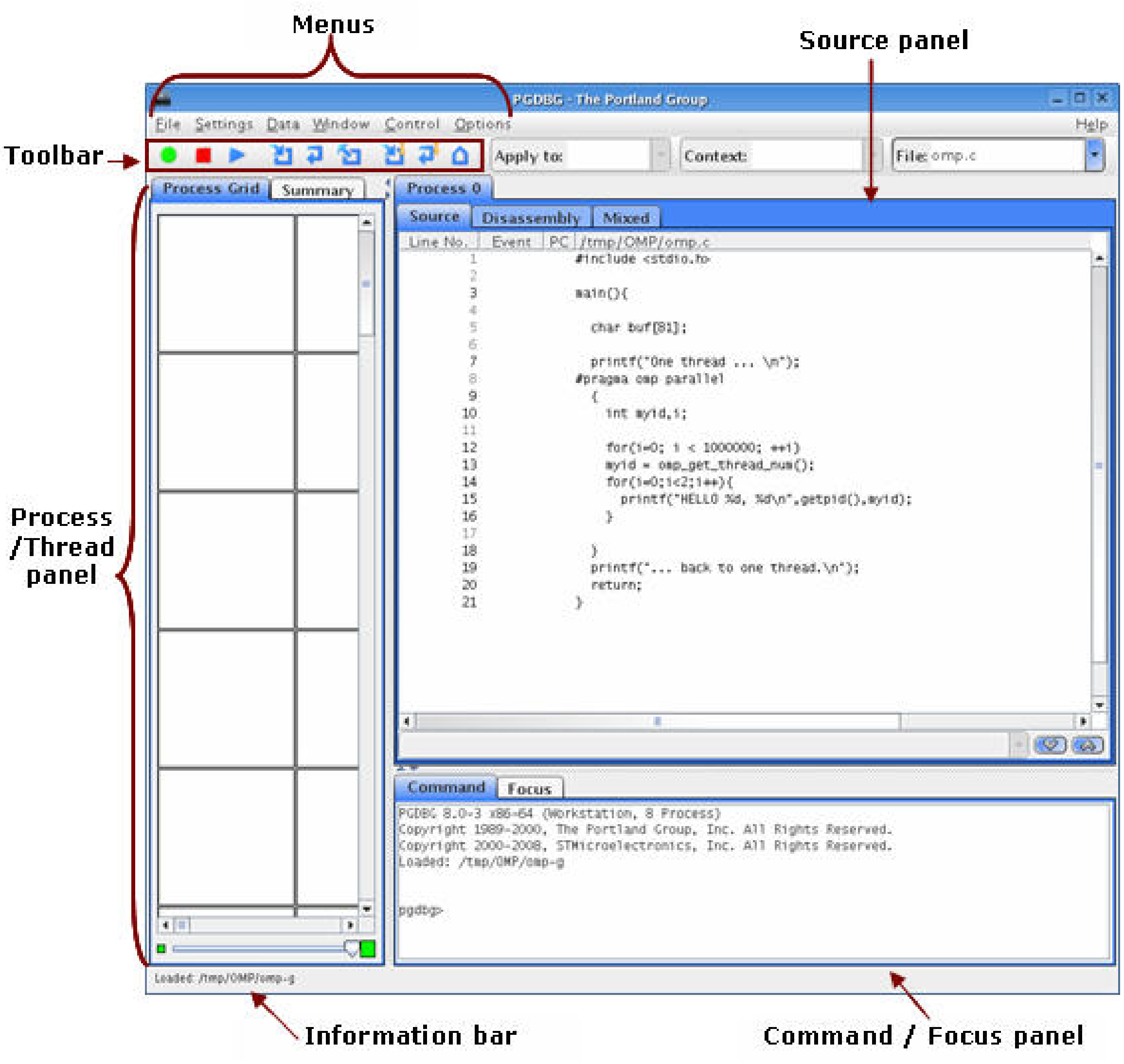
As illustrated in Figure 2.1, the main PGDBG GUI window consists of three panels: Source panel, Thread/ Process panel, and the Command/Focus panel. You can control the size of these windows by moving the horizontal divider bars or by clicking on the small up/down or left/right arrow icons.

The GUI remembers which control panels are visible when you exit and redisplays them when you reopen PGDBG.

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.When you first start PGDBG, this window includes information on the version of PGDBG that you are running as well as copyright information.

On Windows platforms, to maintain input focus in the main window, the Program I/O window is instantiated behind the DBG main window.

Figure 2.1. Default Appearance of PGDBG GUI



```
#include <stdio.h>

main(){

  char buf[81];

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

    for(i=0; i < 1000000; ++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;
}
```

```
PGDBG 8.0-3 x86-64 (Workstation, 8 Process)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2008, STMicroelectronics, Inc. All Rights Reserved.
Loaded: /tmp/OMP/omp-g


pgdbg>
```

PGDBG Main Window Components

Figure 2.1 illustrates the major components of the main PGDBG window, which includes these three panels:

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

The window also contains menus, a toolbar, and three combo boxes that are labeled in Figure 2.2.

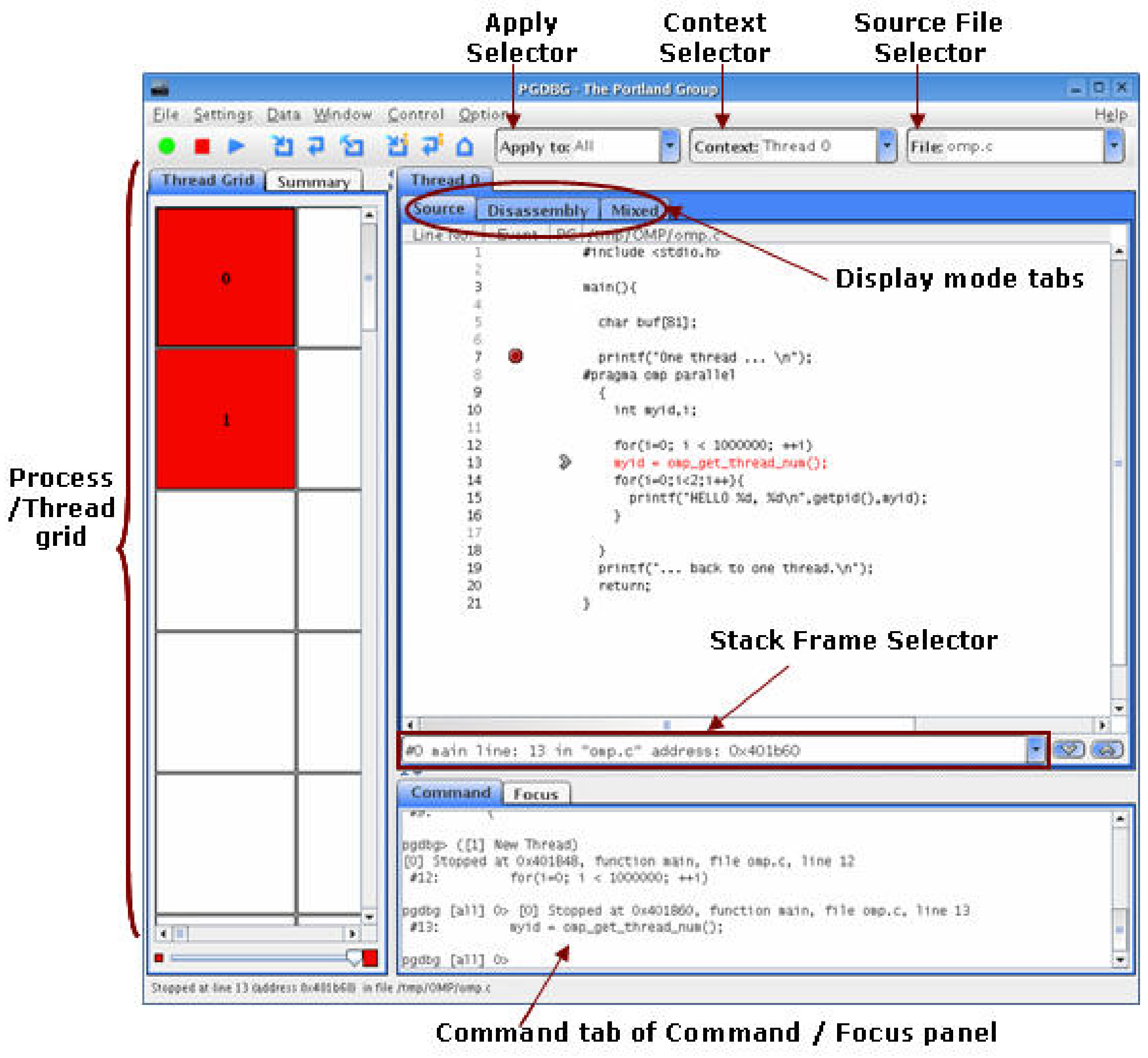
- Apply selector
- Context selector
- Source File selector

For more information on the menus, refer to “Main Window Menus,” on page 11.

For more information on the toolbar, refer to "Toolbar," on page 16.

For more information on the combo boxes, refer to "Main Window Combo Boxes," on page 17.

Figure 2.2. PGDBG GUI Components



Apply to: All

Context: Thread 0

File: omp.c

```
#include <stdio.h>

main(){

  char buf[81];

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

    for(i=0; i < 1000000; ++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: 13 in "omp.c" address: 0x401b60
```

```
pgdbg> ([1] New Thread)
[0] Stopped at 0x401848, function main, file omp.c, line 12
 #12:         for(i=0; i < 1000000; ++i)

pgdbg [all] 0> [0] Stopped at 0x401860, function main, file omp.c, line 13
 #13:         myid = omp_get_thread_num();

pgdbg [all] 0>
```

Stopped at line 13 address 0x401b60 in file /tmp/OMP/omp.c

Let us begin by looking at how you use the panels.

Command/Focus Panel

The Command/Focus panel has two tabs: the Command tab and the Focus tab.

Command tab

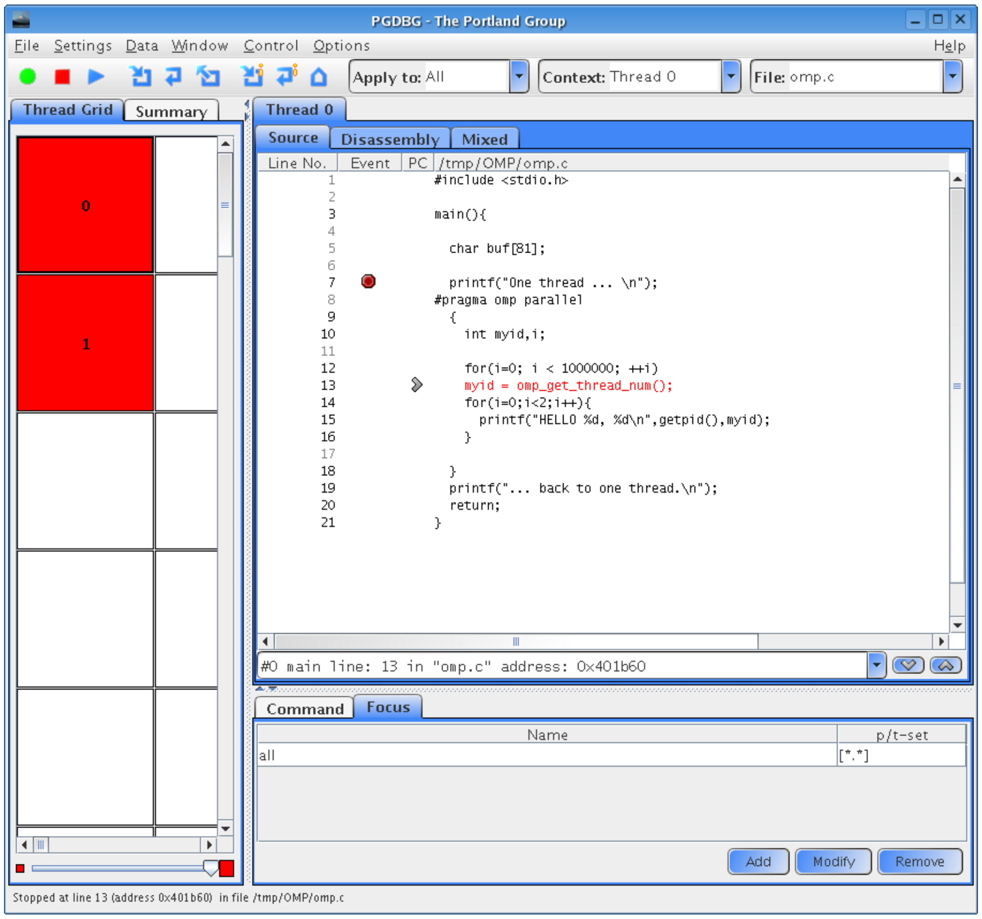
Selecting the Command tab displays the Command panel and provides an interface in which to use the PGDBG command language. Commands entered in this panel are executed, and the results are displayed. For a list of commands that you can enter, refer to "Command Summary," on page 38.

The GUI also supports a "free-floating" version of the Command panel, that is, a version that is not anchored to a particular location on the screen. To use the free-floating command prompt window, select the Command Window check box under the Window menu, as described in "Window Menu," on page 13.

Focus tab

Selecting the Focus tab in a parallel debugging session provides access to the Focus panel in which you can 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. Figure 2.3 displays p/t-sets in the table labeled Focus. In this illustration the Focus table contains one p/t-set called `All` that represents all processes/threads. For more information on p/t-sets, refer to "p/t-set Notation," on page 68.

Figure 2.3. PGDBG GUI Main Window with Focus Tab Selected



```
#include <stdio.h>

main(){

  char buf[81];

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

    for(i=0; i < 1000000; ++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: 13 in "omp.c" address: 0x401b60
```

Name	p/t-set
all	[*.*]

```
Stopped at line 13 (address 0x401b60)  in file /tmp/OMP/omp.c
```

Within the PGDBG GUI, to select a p/t set, use 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.

For non-MPI applications, p/t-sets are used only for distinguishing threads.

Note

The Focus tab has no use in serial debugging, debugging one single-threaded process.

Process/Thread Panel

The Process/Thread panel is another component of the interface used for parallel debugging. This panel has two tabs: the Process/Thread Grid tab and the Summary tab.

Use the slider at the bottom of the grid to zoom in and out on the grid. If the slider is not visible, increase the size of the Process/Thread grid's panel.

Process/Thread Grid tab

When you select the Process/Thread Grid tab, all active target processes and threads are represented as a grid.

- If the target application consists of multiple processes, the tab label is *Process Grid*.
- If the target application is a single multi-threaded process, the tab label is *Thread Grid*.

Note

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

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. Table 2.1 lists the colors and their meanings.

Table 2.1. Colors Describing Thread State

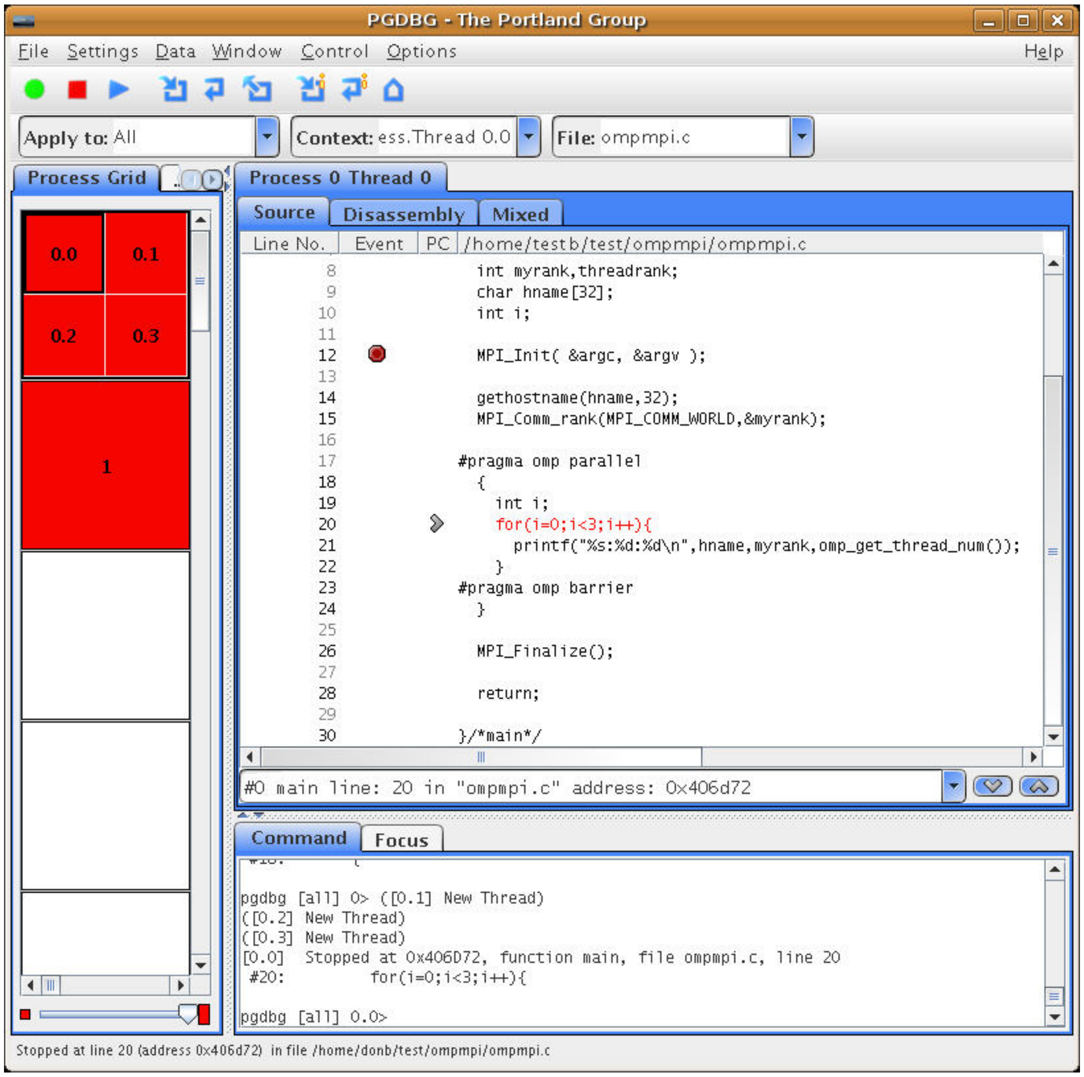
Option	Description
Stopped	Red
Signaled	Blue
Running	Green
Terminated	Black

In the Process/Thread Grid, each element is labeled with a numeric process identifier, as described in "Process-only Debugging," on page 67, 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, as described in "Threads-only Debugging," on page 67. 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 reveals an inner thread grid as illustrated in Figure 2.4.

Figure 2.4. Process Grid with Inner Thread Grid



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

    MPI_Init( &argc, &argv );

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

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

    MPI_Finalize();

    return;

  }/*main*/
```

```
#0 main line: 20 in "ompmpi.c" address: 0x406d72
```

```
pgdbg [all] 0> ([0.1] New Thread)
([0.2] New Thread)
([0.3] New Thread)
[0.0]  Stopped at 0x406D72, function main, file ompmpi.c, line 20
 #20:          for(i=0;i<3;i++){

pgdbg [all] 0.0>
```

```
Stopped at line 20 (address 0x406d72) in file /home/donb/test/ompmpi/ompmpi.c
```

In Figure 2.4 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 67 for more information on processes/thread identifiers.

Summary tab

When you select the Summary tab of the Process/Thread panel, you see a textual representation of the Process/Thread grid. This text representation is essentially the output of the `threads` debugger command, discussed in “Process Control,” on page 87. By default, a summary of all the processes/threads displays.

Source Panel

The Source Panel displays the source code for the current location. You use the source panel to control the debug session, step through source files, set breakpoints, and browse source code.

Source Panel Components

The Source panel has a number of visual aids that allow you to know more about the execution of your code.

- The tab-like section of the panel is a label that defines the process or thread currently being debugged.
- The Display Mode tabs allow you to choose the format for displaying the code: source, assembly language, or mixed - a mix of source and assembly - as described in the following section: "Display Mode tabs".
- The arrow icon under the PC column marks the current location.
- The Line No. column lists source line numbers.
  - Figure 2.3 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 prologues, and some variable declarations.
  - A line number in a black font represents an executable source line.
- The Event column indicates where specific events, such as breakpoints or watchpoints exist.

  For example, Figure 2.2 shows a breakpoint at line 7, indicated by the stop sign icon. Breakpoints may be set at any executable source line by clicking the left mouse button under the Event column of the source line. An existing breakpoint may be deleted by clicking the left mouse button on the stop sign icon. For more information on events, refer to "Source Panel Events," on page 18.
- The Stack Selector, at the bottom of the panel, as illustrated in Figure 2.3, 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. For more information on these commands, refer to the **up** and **down** commands in "Scope," on page 116.

Display Mode tabs

You use the Display Mode Selector to select three different source display modes: Source, Disassembly, and Mixed.

- **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.
- **Disassembly** mode shows the assembly-level instructions of the current routine. This is the default display mode if the source file is not available.
- **Mixed** mode shows assembly-level instructions annotated with source code. This mode is available only if the source file is available.

Main Window Menus

The main window menu bar includes seven menus: File, Settings, Data, Window, Control, Options and Help. Below is a summary of each menu in the main window. Some descriptions contain keyboard shortcuts that are indicated by keystroke combinations (e.g., Control P) enclosed in parentheses.

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. For more information, see the command "**debug**," on page 98.

Attach to Target…

Select this option to attach to a running process. You can attach to a target running on a local or a remote host. For more information, refer to the command" "**attach**," on page 98.

Detach Target

Select this option to end the current debug session. This command does not terminate the target application. For more information, refer to the command" "**detach**," on page 98.

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 the functionality of the component. Deselect this check box to turn them off.

Restore Factory Settings

Select this option to restore the GUI to its initial state illustrated in Figure 2.1, "Default Appearance of PGDBG GUI," on page 6.

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 saves 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 always defaults 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.

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 Pop-Up Menus," on page 18. For more information on printing, also refer to "Printing Variables and Expressions," on page 110.

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 hexadecimal 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 18.

Registers
: Display the registers subwindow. For more information, refer to the command: “**regs**,” on page 117.

Stack
: Display the stack subwindow. For more information, refer to the command” “**stacktrace**,” on page 110.

Locals
: Display a list of local variables that are currently in scope. For more information, refer to the command “**names**,” on page 116.

Custom
: Bring up a custom subwindow.

Disassembler
: Bring up the PGDBG Disassembler subwindow.

Memory
: Bring up the memory dumper subwindow.

Messages

[MPI Debugging Only] Display the MPI message queues. For more information on MPI message queues, refer to "MPI Message Queues," on page 88.

Events

Display a list of currently active breakpoints, watchpoints, etc.

Command Window

When this menu item's check box is selected, the GUI displays a "free-floating" version of the command prompt window. See Chapter 13, "*PGDBG Command Reference*" 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, described in "Toolbar," on page 16. When a control key is available, it is shown in parenthesis.

Arrive

Display the current program location in the Source panel. For more information, refer to the command: "**arrive**," on page 108.

Up

Enter the scope of routine up one level in the call stack. For more information, refer to the command: "**up**," on page 117 (Control U).

Down

Enter the scope of routine down one level in the call stack. For more information, refer to the command" "**down**," on page 116 (Control D).

Run

Run or Rerun the target application. For more information, refer to the commands "**run**," on page 99 and "**rerun**," on page 99 (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. For more information, refer to the command: "**halt**," on page 99 (Control H).

Call…

Open a dialog box to request a routine to call. For more information, refer to the command "**call**," on page 113.

Cont

Continue execution from the current location. For more information, refer to the command: "**cont**," on page 98 (Control G).

Step

Continue and stop after executing one source line, stepping into called routines. For more information, refer to the command: "**step**," on page 100 (Control S).

Next
: Continue and stop after executing one source line, stepping over called routines. For more information, refer to the command: "**next**," on page 99 (Control N).

Step Out
: Continue and stop after returning to the caller of the current routine. For more information, refer to the command: "**stepout**," on page 100 (Control O).

Stepi
: Continue and stop after executing one assembly-level instruction, stepping into called routines. For more information, refer to the command: "**stepi**," on page 100 (Control I).

Nexti
: Continue and stop after executing one assembly-level instruction, stepping over called routines. For more information, refer to the command: "**nexti**," on page 99 (Control T).

Options Menu

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

Search Forward...
: Select this option to perform a forward string search in the currently displayed source file (Control F).

Search Backward...
: Select this option to perform a backward string search in the currently displayed source file (Control B).

Search Again
: Select this option to repeat the last search that was performed on the source panel (Control E).

Locate Routine...
: When this option is selected, PGDBG queries for the name of the routine that you wish to find. If PGDBG has symbol and source information for that routine, it displays the routine in the source panel. Refer to "Source Panel Pop-Up Menus," on page 18.

Set Breakpoint...
: When this option is selected, PGDBG queries for the name of a routine on which to set a breakpoint. The GUI then sets a breakpoint at the first executable source line in the specified routine.

Disassemble
: Disassemble the data selected in the source panel. For more information, refer to the command: "**disasm**," on page 108.

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).

Help Menu

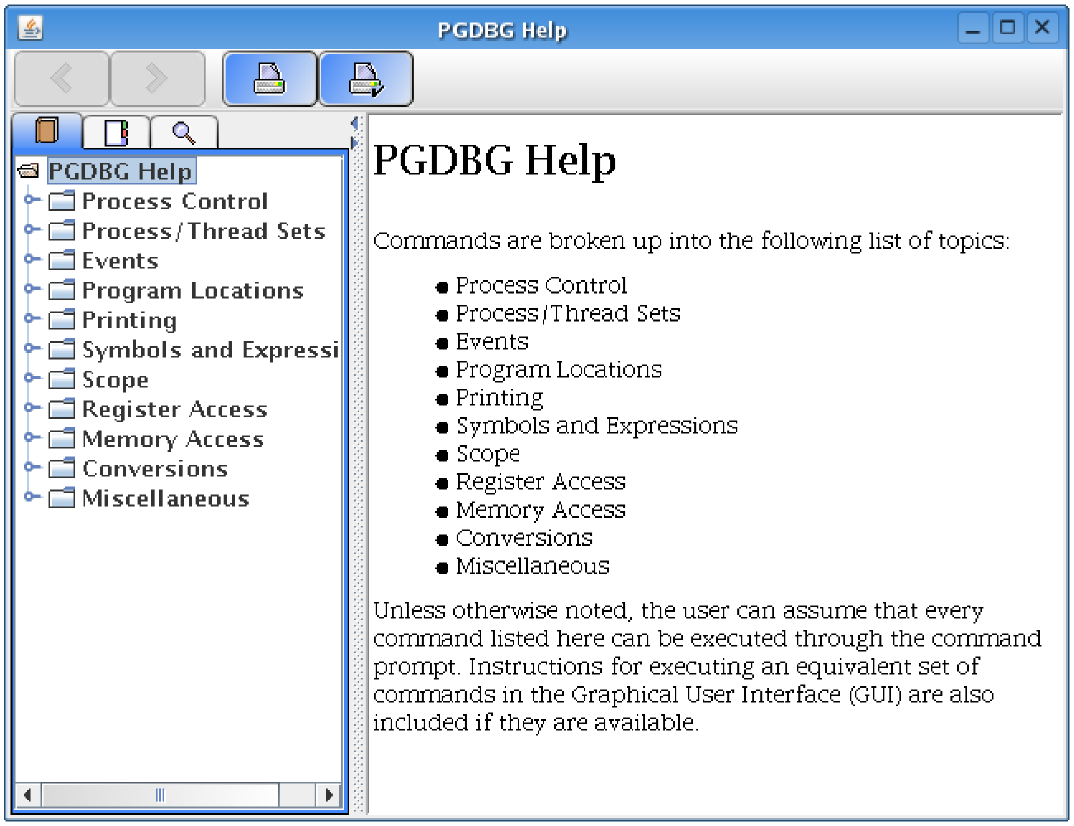
PGDBG Help…

This option starts up PGDBG's integrated help utility as illustrated in Figure 2.5. 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 2.5. PGDBG Help Utility



PGDBG Help

Commands are broken up into the following list of topics:

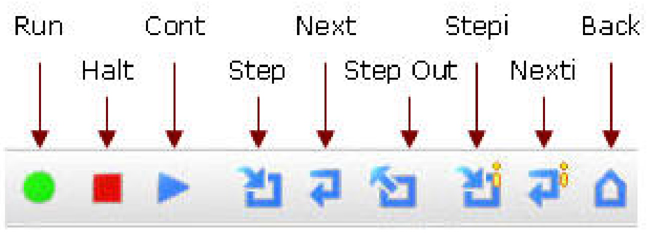
- Process Control
- Process/Thread Sets
- Events
- Program Locations
- Printing
- Symbols and Expressions
- Scope
- Register Access
- Memory Access
- Conversions
- Miscellaneous

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.

Toolbar

There are nine buttons located on the main window toolbar. Figure 2.6 illustrates the toolbar.

Figure 2.6. PGDBG Toolbar



Except for the Back button, these buttons have a corresponding menu item on the Control menu, described in "Control Menu". The functionality from the toolbar is exactly the same as selecting the menu item.

The Back button resets the source panel view to the current PC location, denoted by the left arrow icon under the PC column.

Main Window Combo Boxes

As illustrated in Figure 2.3 the main window contains three 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.

In Figure 2.3 the combo boxes contain the values: All, Thread 0, omp.c, These combo boxes, described in the following paragraphs, are named: Apply Selector, Context Selector, and Source File Selector.

- Use the **Apply Selector** to select the set of processes and/or threads on which to operate. Any command entered in the source panel is 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"Toolbar," on page 16, and so on. Depending on whether you are debugging a multi-threaded, multi-process, or multi-process/multi-threaded (hybrid) target, the following options may be 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
  : *MPI only.* Commands are applied to the current thread in the current process only.

  Focus
  : Commands are applied to the focus group selected in the Focus Panel, described in "Command/Focus Panel," on page 7). Refer to "Process/Thread Sets," on page 68 for 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 - to select the current Process, Thread, or Process.Thread. The current Process, Thread, or Proceess.Thread controls the display in the source panel; it can also be used as a selection for control and display operations. This combo box is not accessible 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 appears in the Source Panel.

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 2.7) and the bottom left displays status messages (e.g., Stopped at line 12… in Figure 2.7).

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 deletes the corresponding breakpoint. Selecting the Events item under the Window menu displays a global list of Events, such as breakpoints, watchpoints, and so on.

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 20 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.

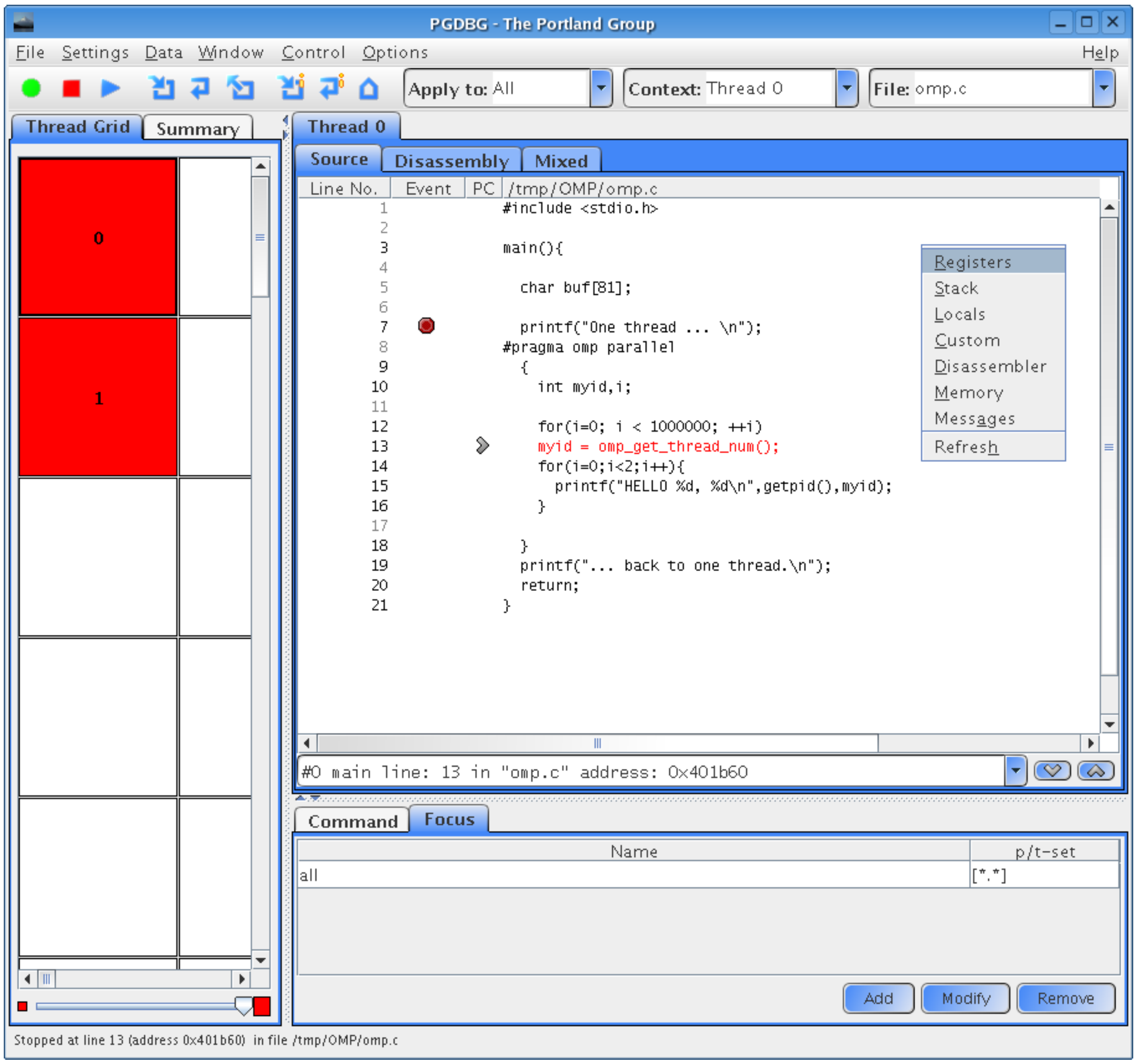
To select text in the source panel:

1. Click on the line of source containing the text.

   This action results in the display of a box surrounding the source line.
2. 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.

Figure 2.7. Opening a Subwindow with a Pop-up Menu



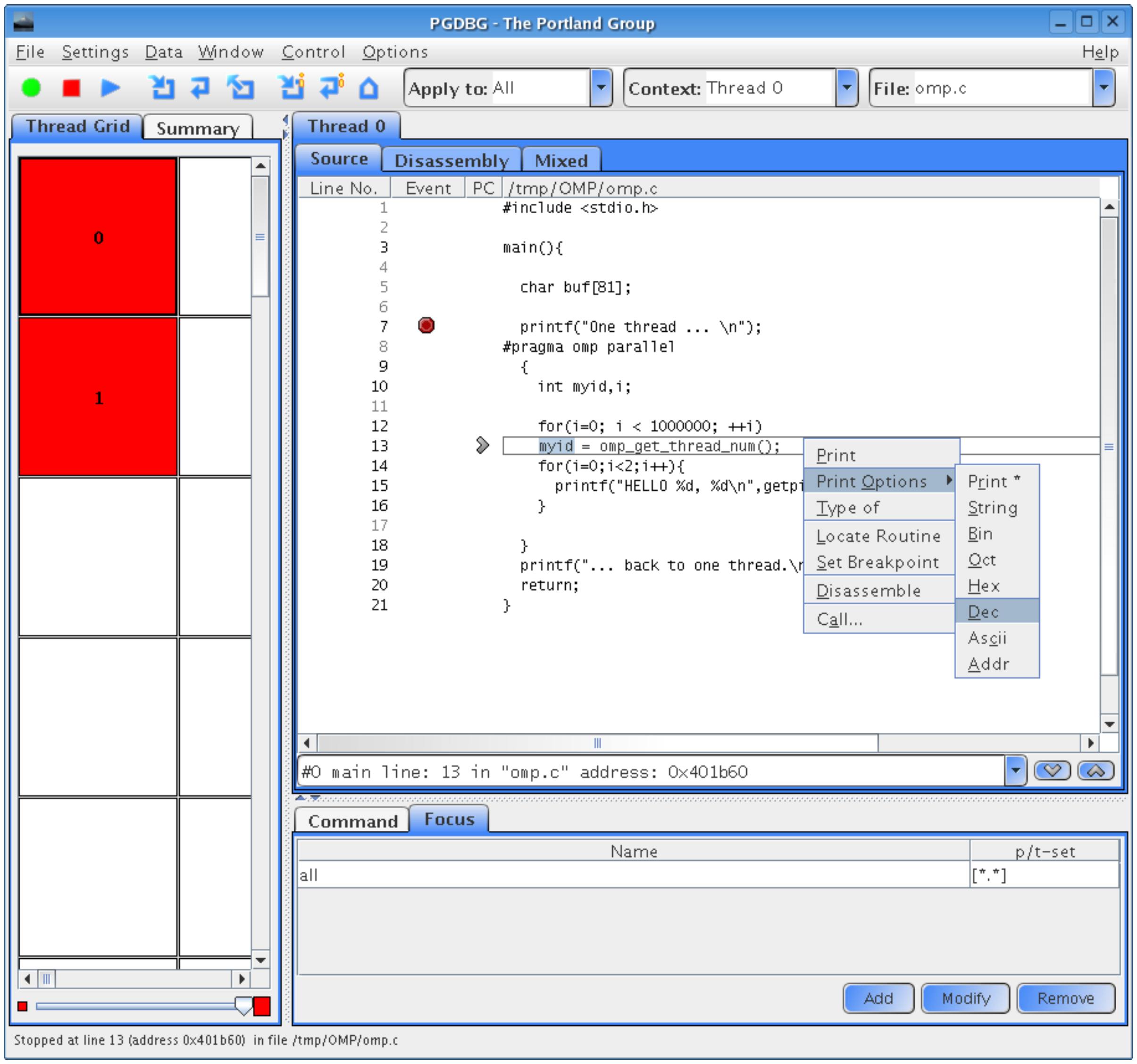
```
#include <stdio.h>

main(){

  char buf[81];

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

    for(i=0; i < 1000000; ++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: 13 in "omp.c" address: 0x401b60
```

```
Stopped at line 13 (address 0x401b60)  in file /tmp/OMP/omp.c
```

Once the text is highlighted, menu selections from the Source Panel menus or from the Source Panel pop-up menu use the highlighted text as input. In Figure 2.8 the variable `myid` is highlighted and the pop-up menu is 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.

Figure 2.8. Data Pop-up Menu



```
#include <stdio.h>

main(){

  char buf[81];

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

    for(i=0; i < 1000000; ++i)
    myid = omp_get_thread_num();
    for(i=0;i<2;i++){
      printf("HELLO %d, %d\n",getpi
    }

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

```
#0 main line: 13 in "omp.c" address: 0x401b60
```

```
Stopped at line 13 (address 0x401b60) in file /tmp/OMP/omp.c
```

The pop-up menu shown in Figure 2.8 provides the Disassemble, Call, and Locate Routine 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 Routine** option displays the source code in which the selected routine is defined.

For more information on each of these selections, refer to "Main Window Menus," on page 11.

Subwindows

PGDBG provides some features that are subwindows, that is, windows that are not connected to the main frame of the PGDBG GUI. One example of a subwindow is the Program I/O window that is displayed at startup. 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, as shown in Figure 2.7, "Opening a Subwindow with a Pop-up Menu," on page 19. 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 21. Specific details of other subwindows are described in subsequent sections. See the description of the Window menu, "Main Window Menus," on page 11 for more information on each subwindow.

The Window menu can be used to bring up a subwindow. You can invoke the same menu by clicking the right mouse button over a blank spot in the source panel. Subwindows are specific to the current process and/or thread. For example, in Figure 2.7, selecting Registers displays the registers subwindow for thread 0, the current thread.

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 that is used for displaying the output of arbitrary commands, and the messages subwindow that is used for displaying the MPI state.

Figure 2.9 shows the Memory subwindow. This subwindow shows all of the possible controls that are available in a PGDBG subwindow. Not all subwindows have all of the components shown in this figure. However, nearly all 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.

The contents of subwindows are generally updated whenever a process or thread stops, such as after a **step**, **next**, **cont**, or **halt** command. You can control updating by using the Lock/Unlock button. Whenever a subwindow is "Locked", its contents are not updated. Figure 2.10 shows a subwindow in the "Locked" state, with the Lock/Unlock button labeled "Unlock". Clicking this button toggles the state of the subwindow to

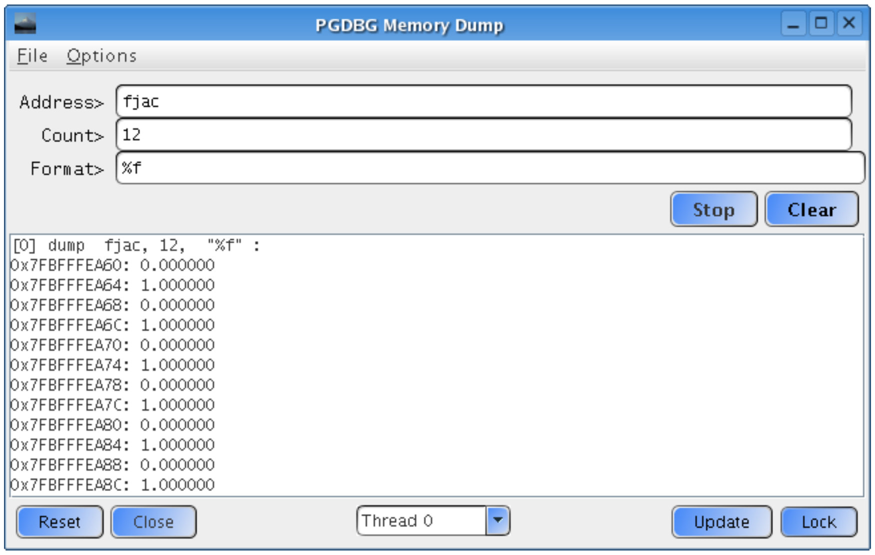
"Unlocked", with the button labeled "Lock", as shown in Figure 2.9. In the "Unlocked" state, the subwindow contents are updated whenever processes or threads stop.

In addition to the subwindow capabilities previously described, subwindows may also have one to three input fields. If the subwindow has one or more input fields, then it also contains 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 illustrated in Figure 2.9. You can use the Context Selector 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 68 for more information on Focus.

Memory Subwindow

Figure 2.9. Memory Subwindow



Address> fjac
Count> 12
Format> %f

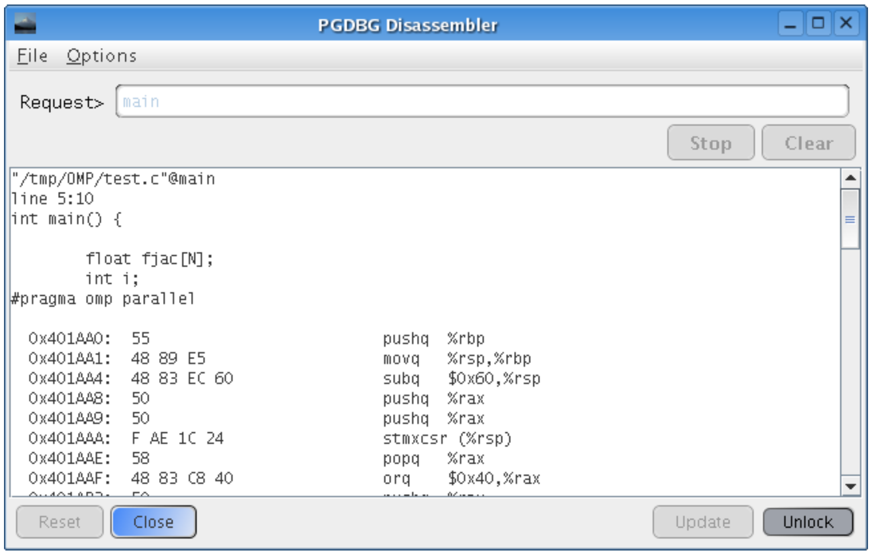
```
[0] dump  fjac, 12,  "%f" :
0x7FBFFFEA60: 0.000000
0x7FBFFFEA64: 1.000000
0x7FBFFFEA68: 0.000000
0x7FBFFFEA6C: 1.000000
0x7FBFFFEA70: 0.000000
0x7FBFFFEA74: 1.000000
0x7FBFFFEA78: 0.000000
0x7FBFFFEA7C: 1.000000
0x7FBFFFEA80: 0.000000
0x7FBFFFEA84: 1.000000
0x7FBFFFEA88: 0.000000
0x7FBFFFEA8C: 1.000000
```

The memory subwindow displays a region of memory using a format descriptor like that of the printf routine from the Standard C Library. 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 118) for a description of supported format strings. The Address field accepts a numeric address or a symbolic variable name.

Disassembler Subwindow

Figure 2.10 shows the Disassembler subwindow. Use this subwindow to disassemble a routine (or a text address) specified in the Request> input field. PGDBG defaults to the current routine if you specify nothing in the Request> input field. After a request is made to the Disassembler, the GUI asks 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 assembly-level instruction level. Choosing "no" dumps the disassembly in the Disassembler subwindow illustrated in Figure 2.10.

Figure 2.10. Disassembler Subwindow



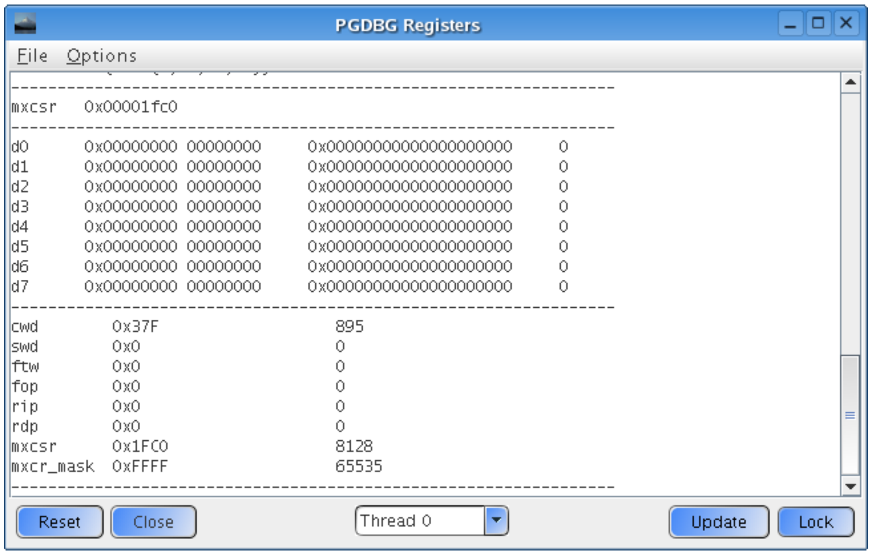
```
"/tmp/OMP/test.c"@main
line 5:10
int main() {

        float fjac[N];
        int i;
#pragma omp parallel

  0x401AA0:  55                          pushq  %rbp
  0x401AA1:  48 89 E5                    movq   %rsp,%rbp
  0x401AA4:  48 83 EC 60                 subq   $0x60,%rsp
  0x401AA8:  50                          pushq  %rax
  0x401AA9:  50                          pushq  %rax
  0x401AAA:  F AE 1C 24                  stmxcsr (%rsp)
  0x401AAE:  58                          popq   %rax
  0x401AAF:  48 83 C8 40                 orq    $0x40,%rax
```

Specifying a text address, rather than a routine name, in the Request> field causes 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 108.

Registers Subwindow

Figure 2.11 illustrates the Registers subwindow. You may view the registers on one or more processes and threads using the Context Selector. The Registers subwindow is essentially a representation of the **regs** debugger command, described in "Register Access," on page 117.

Figure 2.11. Registers Subwindow

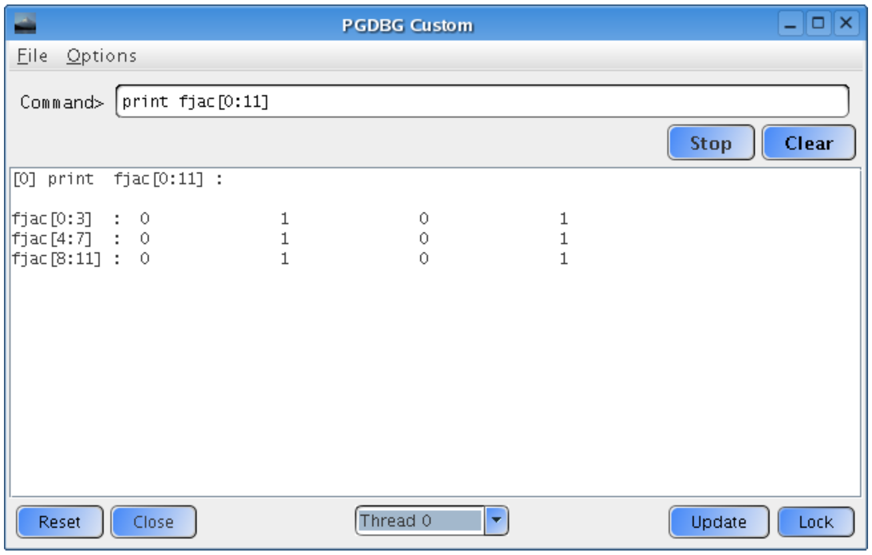


```
------------------------------------------------------------------
mxcsr   0x00001fc0
------------------------------------------------------------------
d0      0x00000000 00000000     0x0000000000000000000     0
d1      0x00000000 00000000     0x0000000000000000000     0
d2      0x00000000 00000000     0x0000000000000000000     0
d3      0x00000000 00000000     0x0000000000000000000     0
d4      0x00000000 00000000     0x0000000000000000000     0
d5      0x00000000 00000000     0x0000000000000000000     0
d6      0x00000000 00000000     0x0000000000000000000     0
d7      0x00000000 00000000     0x0000000000000000000     0
------------------------------------------------------------------
cwd        0x37F                   895
swd        0x0                     0
ftw        0x0                     0
fop        0x0                     0
rip        0x0                     0
rdp        0x0                     0
mxcsr      0x1FC0                  8128
mxcr_mask  0xFFFF                  65535
------------------------------------------------------------------
```

Custom Subwindow

Figure 2.12 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 "*PGDBG Command Language*", including a semicolon-delimited list of commands.

Figure 2.12. Custom Subwindow



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

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

fjac[0:3]  :  0            1            0            1
fjac[4:7]  :  0            1            0            1
fjac[8:11] :  0            1            0            1
```

Messages Subwindow

You use the Messages subwindow for debugging MPI applications. Refer to "MPI Message Queues," on page 88 for more information on the content and use of this subwindow.

Chapter 3. PGDBG Command Line Options

As we stated in Chapter 1, "*Getting Started with the PGDBG Debugger*", PGDBG can interpret command-line options when present on the command line. This chapter describes these options and how they are interpreted.

Command-Line Options Syntax

The pgdbg command accepts several command-line options.

These options must appear on the command line before the name of the program being debugged.

Command-Line Options

The valid PGDBG options are these:

-dbx
: Start the debugger in dbx mode, which provides a dbx-like debugger command language.

-s `startup`
: Specify an alternate initialization file `startup`.

  The default initialization file is `~/.pgdbgrc`.

-c "`command`"
: Execute the debugger `command` command, where the 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.

-mpi
: Debug an MPI application (except for MPICH-1).

-text

Run the debugger using a command-line interface (CLI). The default is for the debugger to launch in graphical user interface (GUI) mode.

–help

Display a list of command-line arguments (this list).

–I *<directory>*

Add *<directory>* to the list of directories that PGDBG uses to search for source files. You can use this option multiple times to add multiple directories to the search path.

Chapter 4. PGDBG Command Language

PGDBG supports a command language that is capable of evaluating complex expressions. The command language is composed of commands, constants, symbols, locations, expressions, and statements.

You can use the command language by invoking the PGDBG command-line interface with the `-text` option, or in the command prompt panel of the PGDBG graphical user interface, as described in "*The PGDBG Graphical User Interface*".

Command Overview

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.

Command Syntax

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.
- You can place multiple commands on a single line by using the semi-colon (;) as a delimiter.

Command Modes

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 is 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, you can use the **enter** command 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

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 51 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. Further, a range of lines is indicated using the range operator ":". Here are some examples:

`break 37`	sets a breakpoint at line 37 of the current source file.
`break "xyz.c"@37`	sets a breakpoint at line 37 of the source file xyz.c.
`list 3:13`	lists lines 3 through 13 of the current file.
`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. Here are some examples:

`line 37`	means “line 37”
`addr 0x1000`	means "address 0x1000"
`addr {line 37}`	means "the address associated with line 37"
`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 following example, 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 that follows 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. The following example sets breakpoints in routines main and xyz, continues, and prints the new current location.

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

However, since the **where** command does not wait until the target application has halted, this statement displays the stack at some arbitrary execution point in the program. To control when the stack is printed, insert a **wait** command, as shown in this example:

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

Note

Any value returned by the last statement on a line is printed.

Statements can be parallelized across multiple threads of execution. For more information, refer to "Parallel Statements," on page 81.

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.

Event Commands

PGDBG supports six 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**, **trace**, **track**, and **do**.

Event Command Descriptions

- The **break** command 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, which can be either an identifier or a variable name. Execution stops when memory at that address is written.
- The **trace** command activates source line tracing, as specified by the arguments you supply.
- 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.

Event Command Arguments

The six 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.

Event Command Examples

Here are some event definition examples:

`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 Command Action

It is useful to know when events take place.

- Event commands that do not explicitly define a location occur at each source line in the program. Here are some examples:

`do {where}`	prints the current location at the start of each source line.
`trace a.b`	prints the value of a.b each time the value has changed.
`track a.b`	prints the value of a.b at the start of each source line if the value has changed.

Note

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, such as **breaki**, **watchi**, **tracei**, **tracki**, and **doi**. The basic difference in the instruction-level version is that these commands interpret integers as addresses rather than line numbers, and events occur at each instruction rather than at each line.
- When multiple events occur at the same location, all event actions are 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, the following syntax creates an ambiguous situation:

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

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

  With this sequence, it is 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, the following command sets a breakpoint at line 37 in the current file.

```
break 37
```

The following command tracks the value of whatever variable `i` is currently in scope.

```
track i
```

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 80 for details.

Expressions

The debugger supports evaluation of expressions composed of constants, identifiers, commands that return values, and operators.

The following rules apply:

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

  For example, the following command invokes the **pc** command to compute the current address, adds 8 to it, and sets a breakpoint at that address.

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

  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. The precedence of ':' is between '||' and '='.

  Here are a few examples that use the range operator:

`print a[1:10]`	prints elements 1 through 10 of the array a.
`list 5:10`	lists source lines 5 through 10.
`list "xyz.c"@5:10`	lists lines 5 through 10 in file xyz.c.

  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. For more details, refer to "Current vs. Prefix p/t-set," on page 70.

Table 4.1, “PGDBG Operators” shows the C language operators that PGDBG supports. The PGDBG operator precedence is the same as in the C language.

Table 4.1. PGDBG Operators

Operator	Description
*	indirection
.	direct field selection
->	indirect field selection
[ ]	``C’’ array index
()	routine call
&	address of
+	add
(type)	cast
-	subtract
/	divide
*	multiply
=	assignment
==	comparison
<<	left shift
>>	right shift

Operator	Description
<=	less than or equal
>=	greater than or equal
!=	not equal
&&	logical and
||	logical or
!	logical not
|	bitwise or
&	bitwise and
~	bitwise not
^	bitwise exclusive or
<<	left shift
>>	right shift
()	FORTRAN array index
%	FORTRAN field selector

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 halts 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, by 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 MPICH-1 program is not recommended. See “MPI Listener Processes,” on page 89, 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.

As described in “Invoking PGDBG for MPI Debugging,” on page 90, when debugging an MPI job via the following command, PGDBG spawns the job in a manner that prevents console-generated interrupts from directly reaching the MPI job launcher or any of the MPI processes.

```
pgdbg -mpi...
```

In this case, typing Control-C only interrupts PGDBG, leaving the MPI processes running. When PGDBG's thread wait mode is not set to none, you can halt the MPI job after using Control-C by entering PGDBG's halt command, even if no PGDBG prompt is generated.

Chapter 5. PGDBG Command Summary

This chapter contains a brief summary of the PGDBG debugger commands. For a detailed description of each command, grouped by category of use, refer to Chapter 13, “*PGDBG Command Reference*”.

If you are viewing an online version of this manual, you can select the hyperlink under the selection category to jump to that section in the manual.

Notation Used in Command Sections

The command sections that follow use these conventions for the command names and arguments, when the command accepts one.

- Command names may be abbreviated by omitting the portion of the command name enclosed in brackets ([]).
- Argument names are chosen to indicate what kind of argument is expected.
- 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.

For example, the following syntax 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 arguments: an integer count, a line range, a routine name, or a line and a count.

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

Command Summary

Table 5.1. PGDBG Commands

Name	Arguments	Category
ad[dr]	[n | line n | routine | var | arg]	“Conversions,” on page 120
	Creates an address conversion under certain conditions.	
al[ias]	[name [string]]	“Miscellaneous,” on page 121
	Create or print aliases.	
arri[ve]		“Program Locations,” on page 108
	Print location information for the current location.	
asc[ii]	exp [,...exp]	“Printing Variables and Expressions,” on page 110
	Evaluate and print as an ascii character.	
as[sign]	var=exp	“Symbols and Expressions,” on page 113
	Set variable `var` to the value of the expression `exp`.	
att[ach]	pid [exe [host]]	“Process Control,” on page 98
	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.	
bin	exp [,...exp]	“Printing Variables and Expressions,” on page 110
	Evaluate and print the expressions. Integer values are printed in binary.	
b[reak]	[line | routine] [if (condition)] [do {commands}]	“Events,” on page 102
	When arguments are specified, sets a breakpoint at the indicated line or routine. When no arguments are specified, prints the current breakpoints.	
breaki	[addr | routine] [if (condition)] [do {commands}]	“Events,” on page 102
	When arguments are specified, sets a breakpoint at the indicated address or routine. When no arguments are specified, prints the current breakpoints.	
breaks		“Events,” on page 102
	Displays all the existing breakpoints	
call	routine [(exp,...)]	“Symbols and Expressions,” on page 113
	Call the named routine.	

Name	Arguments	Category
catch	[number [,number...]]	"Events," on page 102
	With arguments, catches the signals and runs target as though signal was not sent. With no arguments, prints the list of signals being caught.	
cd	[dir]	"Program Locations," on page 108
	Change to the $HOME directory or to the specified directory dir.	
clear	[all | routine | line | addr {addr}]	"Events," on page 102
	With arguments, clears the indicated breakpoints. When no arguments are specified, this command clears all breakpoints at the current location.	
c[ont]		"Process Control," on page 98
	Continue execution from the current location.	
cr[ead]	addr	"Memory Access," on page 118
	Fetch and return an 8-bit signed integer (character) from the specified address.	
de[bug]	[target [arg1 _ argn]]	"Process Control," on page 98
	Load the specified target program with optional command-line arguments.	
dec	exp [,...exp]	"Printing Variables and Expressions," on page 110
	Evaluate and print the expressions. Integer values are printed in decimal	
decl[aration]	name	"Symbols and Expressions," on page 113
	Print the declaration for the symbol based on its type according to symbol table.	
decls	[routine | "sourcefile" | {global}]	"Scope," on page 116
	Print the declarations of all identifiers defined in the indicated scope. If no scope is given, print the declarations for global scope	
defset	name [p/t-set]	"Process-Thread Sets," on page 101
	Assign a name to a process/thread set. Define a named set.	
del[ete]	event-number | all | 0 | event-number [,.event-number.]	"Events," on page 102
	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.	
det[ach]		"Process Control," on page 98
	Detach from the current running process.	
dir[ectory]	[pathname]	"Miscellaneous," on page 121
	Add the directory pathname to the search path for source files. If no argument is specified, the currently defined directories are printed.	

Name	Arguments	Category
dis[asm]	[count | lo:hi | routine | addr, count]	"Program Locations," on page 108
	Disassemble memory. If no argument is given, disassemble four instructions starting at the current address.	
disab[le]	event-number | all	"Printing Variables and Expressions," on page 110
	With arguments, disables the event `event-number` or all events. When no arguments are specified, prints both enabled and disabled events.	
display	exp [,...exp]	"Printing Variables and Expressions," on page 110
	With an argument or several arguments, print expression exp at every breakpoint. Without arguments, list the expressions for PGDBG to automatically display at breakpoints.	
do	{commands} [at line | in routine] [if (condition)]	"Events," on page 102
	Define a do event. Without the optional arguments at or in, the commands are executed at each line in the program.	
doi	{commands} [at addr | in routine] [if (condition)]	"Events," on page 102
	Define a doi event. If neither the at or in argument is specified, then the commands are executed at each instruction in the program.	
down		"Scope," on page 116
	Enter scope of routine down one level or number levels on the call stack.	
dr[ead]	addr	"Memory Access," on page 118
	Fetch and return a 64 bit double from the specified address.	
du[mp]	address, count, "format-string"	"Memory Access," on page 118
	Dumps the contents of a region of memory. The output is formatted according to a printf-like format descriptor.	
edit	[filename | routine]	"Program Locations," on page 108
	Edit the specified file or file containing the routine. If no argument is supplied, edit the current file starting at the current location.	
enab[le]	event-number | all	"Events," on page 102
	With arguments, this command enables the event `event-number` or all events. When no arguments are specified, prints both enabled and disabled events.	
en[ter]	routine | "sourcefile" | {global}	"Scope," on page 116
	Set the search scope to be the indicated symbol, which may be a routine, source file or global. Using no argument is the same as using enter global	

Name	Arguments	Category
entr[y]	routine	“Symbols and Expressions,” on page 113
	Return the address of the first executable statement in the program or specified routine.	
fil[e]		“Program Locations,” on page 108
	Change the source file to the file filename and change the scope accordingly. With no argument, print the current file.	
files		“Scope,” on page 116
	Return the list of known source files used to create the executable file	
focus	[p/t-set]	“Process-Thread Sets,” on page 101
	Set the target process/thread set for commands. Subsequent commands are applied to the members of this set by default.	
fp		“Register Access,” on page 117
	Return the current value of the frame pointer.	
fr[ead]	addr	“Memory Access,” on page 118
	Fetch and print a 32-bit float from the specified address.	
func[tion]	[addr | line]	“Conversions,” on page 120
	Return a routine symbol. If no argument is specified, return the current routine.	
glob[al]		“Global Commands,” on page 75
	Return a symbol representing global scope.	
halt	[command]	“Process Control,” on page 98
	Halt the running process or thread.	
he[lp]		“Miscellaneous,” on page 121
	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.	
hex	Exp [,...exp]	“Printing Variables and Expressions,” on page 110
	Evaluate and print expressions as hexadecimal integers.	
hi[story]	[num]	“Miscellaneous,” on page 121
	List the most recently executed commands. With the num argument, resize the history list to hold num commands.	
hwatch	addr | var [if (condition)] [do {commands}]	“Events,” on page 102
	Define a hardware watchpoint.	

Name	Arguments	Category
hwatchb[oth]	addr | var [if (condition)] [do {commands}]	"Events," on page 102
	Define a hardware read/write watchpoint.	
hwatchr[ead]	addr | var [if (condition)] [do {commands}]	"Events," on page 102
	Define a hardware read watchpoint.	
ignore	[number [,number...]]	"Events," on page 102
	Ignores the specified signals and does not deliver them to the target. When no arguments are specified, prints the list of signals being ignored.	
ir[ead]	addr	"Memory Access," on page 118
	Fetch and print a signed integer from the specified address.	
language		"Miscellaneous," on page 121
	Print the name of the language of the current file.	
lin[e]	[n | routine | addr]	"Conversions," on page 120
	Create a source line conversion. If no argument is given, return the current source line.	
lines	routine	"Program Locations," on page 108
	Print the lines table for the specified routine.	
lis[t]	[count | line,count | lo:hi | routine]	"Program Locations," on page 108
	With no argument, list 10 lines centered at the current source line. If an argument is specified, lists lines based on information requested.	
lr[ead]	addr	"Memory Access," on page 118
	Fetch and print an address from the specified address.	
log	filename	"Miscellaneous," on page 121
	Keep a log of all commands entered by the user and store it in the named file.	
lv[al]	exp	"Symbols and Expressions," on page 113
	Return the lvalue of the expression expr.	
mq[dump]		"Memory Access," on page 118
	Dump MPI message queue information for the current process.	
names	[routine | "sourcefile" | {global}]	"Scope," on page 116
	Print the names of all identifiers defined in the indicated scope. If no scope is specified, use the search scope.	

Name	Arguments	Category
n[ext]	[count]	"Process Control," on page 98
	Stop after executing one or count source line(s) in the current routine.	
nexti	[count]	"Process Control," on page 98
	Stop after executing one or count instruction(s) in the current routine.	
nop[rint]	exp	"Miscellaneous," on page 121
	Evaluate the expression but do not print the result.	
oct	exp [,...exp]	"Printing Variables and Expressions," on page 110
	Evaluate and print expressions as octal integers.	
pc		"Register Access," on page 117
	Return the current program address.	
pgienv	[command]	"Miscellaneous," on page 121
	Define the debugger environment. With no arguments, display the debugger settings.	
p[rint]	exp1 [,...expn]	"Printing Variables and Expressions," on page 110
	Evaluate and print one or more expressions.	
printf	"format_string", expr,...expr	"Printing Variables and Expressions," on page 110
	Print expressions in the format indicated by the format string.	
proc	[id]	"Process Control," on page 98
	Set the current process to the process identified by id. When issued with no argument, proc lists the location of the current thread of the current process in the current program.	
procs		"Process Control," on page 98
	Print the status of all active processes, listing each process by its logical process ID.	
pwd		"Program Locations," on page 108
	Print the current working directory.	
q[uit]		"Process Control," on page 98
	Terminate the debugging session.	
regs	[x | f | d]	"Register Access," on page 117
	Print a formatted display of the names and values of the integer, float, and double registers. If the format parameter is omitted, then PGDBG prints all of the registers.	
rep[eat]	[first, last] | [first: last:n] | [num] | [-num]	"Miscellaneous," on page 121
	Repeat the execution of one or more previous history list commands.	

Name	Arguments	Category
rer[un]	[arg0 arg1 ... argn] [< inputfile] [[> | >& | >> | >>&] outputfile]	“Process Control,” on page 98
	Like the **run** command, except if no args are specified, the previously used target arguments are not re-used.	
ret[addr]		“Register Access,” on page 117
	Return the current return address.	
ru[n]	[arg0 arg1 ... argn] [< inputfile] [> outputfile]	“Process Control,” on page 98
	Execute program from the beginning. If arguments arg0, arg1, and so on are specified, they are set up as the command-line arguments of the program.	
rv[al]	expr	“Symbols and Expressions,” on page 113
	Return the rvalue of the expression expr.	
sco[pe]		“Scope,” on page 116
	Return a symbol for the search scope.	
scr[ipt]	filename	“Miscellaneous,” on page 121
	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.	
set	var = ep	“Symbols and Expressions,” on page 113
	Set variable var to the value of expression.	
setenv	name | name value	“Miscellaneous,” on page 121
	Print value of environment variable name. With a specified value, set name to value.	
sh[ell]	arg0 [... argn]	“Miscellaneous,” on page 121
	Fork a shell (defined by $SHELL) and give it the indicated arguments (the default shell is sh). Without arguments, invokes an interactive shell, and executes until a "^D" is entered.	
siz[eof]	name	“Symbols and Expressions,” on page 113
	Return the size, in bytes, of the variable type name; or, if the name refers to a routine, returns the size in bytes of the subprogram.	
sle[ep]	time	“Miscellaneous,” on page 121
	Pause for time seconds. If no time is specified, pause for one second	
source	filename	“Miscellaneous,” on page 121
	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.	

Name	Arguments	Category
sp		"Register Access," on page 117
	Return the current stack pointer address.	
sr[ead]	addr	"Memory Access," on page 118
	Fetch and print a short signed integer from the specified address	
stackd[ump]	[count]	"Program Locations," on page 108
	Print a formatted dump of the stack. This command displays a hex dump of the stack frame for each active routine.	
stack[trace]	[count]	"Program Locations," on page 108
	Print a stacktrace. For each active routine print the routine name, source file, line number, current address, provided that information is available.	
stat[us]		"Events," on page 102
	Display all the event definitions, including an event number by which the event can be identified.	
s[tep]	[count | up]	"Process Control," on page 98
	Step into the current routine and stop after executing one or count source line(s). If the up argument is specified, stops execution after stepping out of the current routine.	
stepi	[count | up]	"Process Control," on page 98
	Step into the current routine and stop after executing one or count source line(s). If the up argument is specified, stops execution after stepping out of the current routine.	
stepo[ut]		"Process Control," on page 98
	Stop after returning to the caller of the current routine.	
stop	[at line | in routine] [var] [if (condition)] [do {commands}]	"Events," on page 102
	Set a breakpoint at the indicated routine or line. Break when the value of the indicated variable var changes.	
stopi	[at addr | in routine] [var] [if (condition)] [do {commands}]	"Events," on page 102
	Set a breakpoint at the indicated address or routine. Break when the value of the indicated variable var changes.	
sync	[routine | line]	"Process Control," on page 98
	Advance the current process/thread to a specific program location, ignoring any user-defined events.	
synci	[routine | addr]	"Process Control," on page 98
	Advance the current process/thread to a specific program location, ignoring any user-defined events.	

Name	**Arguments**	**Category**
str[ing]	exp [,...exp]	"Printing Variables and Expressions," on page 110
	Evaluate and print expressions as null-terminated character strings, up to a maximum of 70 characters.	
thread	number	"Process Control," on page 98
	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		"Process Control," on page 98
	Prints the status of all active threads, grouped by process.	
trace	[at line | in routine] [var | routine] [if (condition)] do {commands}	"Events," on page 102
	Activates source line tracing as specified by the arguments supplied.	
tracei	[at addr | in routine] [var] [if (condition)] do {commands}	"Events," on page 102
	Activates instruction tracing as specified by the arguments supplied.	
track	expression [at line | in routine] [if (condition)] [do {commands}]	"Events," on page 102
	Define a track event.	
tracki	expression [at addr | in routine] [if (condition)] [do {commands}]	"Events," on page 102
	Define an assembly-level track event.	
type	expr	"Symbols and Expressions," on page 113
	Return the type of the expression.	
unal[ias]	name	"Miscellaneous," on page 121
	Remove the alias definition for name, if one exists.	
undefset	[name | -all]	"Process-Thread Sets," on page 101
	Remove a previously defined process/thread set from the list of process/thread sets.	
undisplay	[all | 0 | exp]	"Printing Variables and Expressions," on page 110
	Remove all expressions specified by previous display commands. With an argument or several arguments, remove the expression exp from the list of display expressions.	

Name	Arguments	Category
unb[reak]	line | routine | all	"Events," on page 102
	Remove a breakpoint from the statement line, the routine `routine`, or remove all breakpoints.	
unbreaki	addr | routine | all	"Events," on page 102
	Remove a breakpoint from the address addr, the routine `routine`, or remove all breakpoints.	
up		"Scope," on page 116
	Enter scope of routine up one level or number levels on the call stack.	
use	[dir]	"Miscellaneous," on page 121
	Print the current list of directories or add dir to the list of directories to search. If the first character in pathname is ~, the value of $HOME is substituted for this character.	
viewset	name	"Process-Thread Sets," on page 101
	List the members of a process/thread set that currently exist as active threads or list defined p/t-sets.	
wait	[any | all | none]	"Process Control," on page 98
	Inserts explicit wait points in a command stream.	
wa[tch]	expression [at line | in routine] [if (condition)] [do {commands}]	"Events," on page 102
	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.	
watchi	expression [at addr | in routine] [if(condition)] [do {commands}]	"Events," on page 102
	Define an assembly-level watch event.	
whatis	[name]	"Symbols and Expressions," on page 113
	With no arguments, prints the declaration for the current routine. With argument name, prints the declaration for the symbol name.	
when	[at line | in routine] [if (condition)] do {commands}	"Events," on page 102
	Execute `commands` at every line in the program, at a specified line in the program or in the specified routine.	
wheni	[at addr | in routine] [if(condition)] do {commands}	"Events," on page 102
	Execute `commands` at each address in the program. If an addr is specified, the commands are executed each time the address is reached.	

Name	Arguments	Category
w[here]	[count]	“Program Locations,” on page 108
	Print a stacktrace. For each active routine print the routine name, routine arguments, source file, line number, current address, provided that information is available.	
whereis	name	“Symbols and Expressions,” on page 113
	Print all declarations for name.	
which	name	“Scope,” on page 116
	Print full scope qualification of symbol name.	
whichsets	[p/t-set]	“Process-Thread Sets,” on page 101
	List all defined p/t-sets to which the members of a process/thread set belong.	
/	/ [string] /	“Program Locations,” on page 108
	Search forward for a string (`string`) of characters in the current source file	
?	?[string] ?	“Program Locations,” on page 108
	Search backward for a string (`string`) of characters in the current source file.	
!	History modification	“Miscellaneous,” on page 121
	Executes a command from the command history list. The command executed depends on the information that follows the !.	
^	History modification	“Miscellaneous,” on page 121
	Quick history command substitution ^old^new^<modifier> this is equivalent to !:s/old/new/	

Chapter 6. PGDBG Assembly-Level Debugging

This section provides information about PGDBG assembly-level debugging, including an overview and what to expect if you are using assembly-level debugging or if you did not compile your program for debugging.

Assembly-Level Debugging Overview

PGDBG does not require that the program under debug be compiled with debugging information, using `-g`. It can debug code that is lacking debug information, but because it is missing information about symbols and line numbers, it can only access the program at the assembly level.

As described in "Building Applications for Debug," on page 1, the most information is available when the program is compiled using `-g` or `-gopt` with no optimization. When a program is compiled at higher levels of optimization, less information about source-level symbols and line numbers is available, even if the program was compiled with `-g` or `-gopt`. In such cases, if you want to find the source of a problem without rebuilding the program, you may need to debug at the assembly level.

If a program has been "stripped" of all symbols, either by the linker or a separate utility, then debugging is at the assembly level. PGDBG is only able to examine or control the program in terms of memory addresses and registers.

Assembly-Level Debugging on Microsoft Windows Systems

When applications are built without `-g` on Windows systems, the resulting binary, the `.exe` file, does not contain any symbol information. Microsoft stores symbol information in a program database, a `.pdb` file. To generate a `.pdb` file using the PGI compiler drivers, you must use `-g` during the link step. You can do this even if you did not use `-g` during the compile step. Having this `.pdb` file available provides PGDBG with enough symbol information to map addresses to routine names.

Assembly-Level Debugging with Fortran

To refer to Fortran symbol names when debugging at the assembly level, you must translate the names to use the naming convention that matches the calling convention in use by the compiler. For code compiled by the

PGI compilers, in most cases this means translating to lower case and appending an underbar. For example, a routine that appears in the source code as "`VADD`" would be referred to in the debugger as "`vadd_`".

On 32-bit Windows systems there are alternative calling conventions. The one described above matches the convention used when the compiler is invoked with `-Munix`' For details of other 32-bit Windows calling conventions, refer to the *PGI User's Guide*.

Note

Name translation is only necessary for assembly-level debugging. When debugging at the source level, you may refer to symbol names as they appear in the source.

A special symbol, `MAIN_`, is created by PGI Fortran to refer to the main program. PGI Fortran generates this special symbol whether or not there is a PROGRAM statement. One way to run to the beginning of a Fortran program is to set a breakpoint on MAIN_, then run.

Assembly-Level Debugging with C++

C++ symbol names are "mangled" names. For the names of C++ methods, the names are modified to include not only the name as it appears in the source code, but information about the enclosing class hierarchy, argument and return types, and other information. The names are long and arcane. At the source level these names are translated by PGDBG to the names as they appear in the source. At the assembly level, these names are in the mangled form. Translation is not easy and not recommended. If you have no other alternative, you can find information about name mangling in the *PGI User's Guide*.

Assembly-Level Debugging Using the PGDBG GUI

This section describes some basic operations for assembly-level debugging using the PGDBG GUI. When you invoke PGDBG using the following command, you are presented with a dialog telling you that PGDBG "`Can't find main function compiled -g`". No source is displayed.

```
pgdbg myprog
```

To get into the program, you can select the Options | Set Breakpoint... menu option. To stop at program entry, for example, in Fortran you could enter `MAIN_` in response to the dialog query; in C or C++ you could enter `main`.

Assembly-Level PGDBG Menu Options

PGDBG menu options that are useful in assembly-level debugging:

Window | Registers
: Display all the registers

Window | Memory
: Display memory locations

Window | Stack
: Display a stack traceback

Window | Disassembly
: Display disassembly

If disassembly is not automatically displayed in the code panel when the program stops at a breakpoint or after a single step, use Window | Disassembly and enter the name of the routine of interest. Answer "`Yes`" when asked if you want to display disassembly in the source window.

Assembly-Level Debugging Using the PGDBG Command-line Interface

This section describes some basic operations for assembly-level debugging using the PGDBG command-line interface. When you invoke PGDBG using this command:

```
  pgdbg -text myprog
```

you are presented with a message telling you that PGDBG `"Can't find main function compiled -g"`.

To get into the program, you can set a breakpoint at a named routine. To stop at program entry, for example, in Fortran you could use

```
   break MAIN_
```

and in C/C++ you could use

```
   break main
```

Some useful commands for assembly-level debugging using the PGDBG command-line interface include:

run
: run the program from the beginning

cont
: continue program execution from the current point

nexti
: single-step one instruction, stepping over calls

stepi
: single-step one instruction, stepping into calls

breaki
: set a breakpoint at a given address

regs
: display the registers

print $<regname>
: display the value of the specified register

  For more information on register names, refer to "Register Symbols," on page 51.

dump
: dump memory locations

stacktrace
: display a stack traceback

stackdump
: display a traceback/dump of stack frame

Register Symbols

This section describes the register symbols defined for X86 processors and Intel 64/AMD64 processors operating in compatibility or legacy mode.

X86 Register Symbols

This section describes the X86 register symbols.

Table 6.1. 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 6.2. x87 Floating-Point Stack Registers

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

Table 6.3. 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 6.4. 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 6.5. 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 6.6. Floating-Point Registers

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

Table 6.7. 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 6.8. 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 6.9. 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:

127 96	95 64	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.

Chapter 7. PGDBG Source-Level Debugging

This chapter describes source-level debugging, including debugging Fortran and Debugging C++.

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 64) support large arrays (arrays with an aggregate size greater than 2GB). You can enable large array support by compiling using these options: `–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 and supports the Fortran array index selector "()" and the Fortran field selector "%" for derived types. However, `.eq.`, `.ne.`, and so forth are not supported. You must use the analogous C operators ==, !=, and so on, instead.

Note

The precedence of operators matches the C language, which may in some cases be different than for Fortran.

See Table 5.1, “PGDBG Commands” 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. You can always use the following command to set a breakpoint at the start of the main routine.

```
break MAIN
```

Common Blocks

Each subprogram that defines a common block has a local static variable symbol to define the common. The address of the variable is the address of the common block. The type of the variable is a locally-defined structure type with fields defined for each element of the common block. The name of the variable is 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/ a, b
integer a
integer 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
```

Internal Procedures

To unambiguously reference an internal procedure, qualify its name with the name of its host using the scoping operator @.

For example:

```
subroutine sub1 ()
    call internal_proc ()
    contains
    subroutine internal_proc ()
        print *, "internal_proc in sub1"
    end subroutine internal_proc
end subroutine

subroutine sub2 ()
    call internal_proc ()
    contains
    subroutine internal_proc ()
        print *, "internal_proc in sub2"
    end subroutine internal_proc
end subroutine
program main
    call sub1 ()
    call sub2 ()
end program

pgdbg> whereis internal_proc
function:       "/path/ip.f90"@sub1@internal_proc
function:       "/path/ip.f90"@sub2@internal_proc

pgdbg> break sub1@internal_proc
(1)breakpoint set at: internal_proc line: "ip.f90"@5 address: 0x401E3C 1
pgdbg> break sub2@internal_proc
(2)breakpoint set at: internal_proc line: "ip.f90"@13 address: 0x401EEC 2
```

Modules

A member of a Fortran 90 module can be accessed during debugging.

```
module mod
    integer iMod
end module
subroutine useMod()
    use mod
    iMod = 1000
end subroutine
program main
    call useMod()
end program
```

- If the module is in the current scope, no qualification is required to access the module's members.

```
pgdbg> b useMod
(1)breakpoint set at: usemod line: "modv.f90"@7 address: 0x401CC4
1

Breakpoint at 0x401CC4, function usemod, file modv.f90, line 7
 #7:         iMod = 1000

pgdbg> p iMod
0
```

- If the module is not in the current scope, use the scoping operator @ to qualify the member's name.

```
Breakpoint at 0x401CF0, function main, file modv.f90, line 11
 #11:         call useMod()

pgdbg> p iMod
"iMod" is not defined in the current scope

pgdbg> p mod@iMod
0
```

Module Procedures

A module procedure is a subroutine contained within a module. A module procedure itself can contain internal procedures. The scoping operator @ can be used when working with these types of subprograms to prevent ambiguity.

```
module mod
    contains
    subroutine mod_proc1()
        call internal_proc()
        contains
        subroutine internal_proc()
            print *, "internal_proc in mod_proc1"
        end subroutine
    end subroutine
    subroutine mod_proc2()
        call internal_proc()
        contains
        subroutine internal_proc()
            print *, "internal_proc in mod_proc2"
        end subroutine
    end subroutine
end module
```

```
program main
    use mod
    call mod_proc1
    call mod_proc2
end program
```

```
pgdbg> whereis internal_proc
function:       "/path/modp.f90"@mod@mod_proc1@internal_proc
function:       "/path/modp.f90"@mod@mod_proc2@internal_proc

pgdbg> break mod@mod_proc1@internal_proc
(1)breakpoint set at: internal_proc line: "modp.f90"@7 address: 0x401E3C
1
pgdbg> break mod@mod_proc2@internal_proc
(2)breakpoint set at: internal_proc line: "modp.f90"@14 address: 0x401EEC
2
```

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, suppose you were given the following definition of class Person and the appropriate implementation of its methods:

```
class Person
{
    public:
    char name[10];
    Person(char * inName);
    void print();
};

int main ()
{
    Person * pierre;
    pierre = new Person("Pierre");
    pierre->print();
    return 0;
}
```

Call the instance method **print** on object `pierre` as follows:

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

Notice that `pierre` must be explicitly passed into the method because it is the *this* pointer. You can also specify the class name to remove ambiguity.

Chapter 8. PGDBG Platform-Specific Features

This chapter describes the PGDBG features that are specific to particular platforms, such as pathname conventions, debugging with core files, and signals.

Pathname Conventions

PGDBG uses the forward slash character (/) as the path component separator on all platforms. The backslash (\) is used as the escape character in the PGDBG command language.

On Windows systems, a drive letter specifier may be used whenever specifying a full path, but the forward slash separator convention is still in effect. For example, to add the Windows pathname "C:\Temp\src" to the list of searched source directories, use the command:

```
pgdbg> dir C:/Temp/src
```

To set a breakpoint at line 10 of the source file specified by the relative path `sub1\main.c`, use this command:

```
pgdbg> break "sub1/main.c":10
```

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 (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 shell environment in which the application runs must be set up to allow core file creation. On many systems, the default user setting `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 previously described. 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.

When debugging core files, PGDBG:

- Supports all non-control commands.
- Performs any command that does not cause the program to run.
- Generates an error message in PGDBG for any command that causes the program to run.
- May provide the status of multiple threads, depending on the type of core file created.

PGDBG does not support multi-process core file debugging.

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.

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) 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) 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.

Chapter 9. PGDBG Parallel Debugging Overview

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

Overview of Parallel Debugging Capability

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.

For specific information on Multi-thread and OpenMP debugging, refer to Chapter 10, "*PGDBG - Parallel Debugging with OpenMP*".

For specific information on Multi-process MPI debugging, refer to Chapter 11, "*PGDBG Parallel Debugging with MPI*".

Graphical Presentation of Threads and Processes

PGDBG graphical user interface components that provide support for parallelism are described in detail in "*The PGDBG Graphical User Interface*".

Basic Process and Thread Naming

Because PGDBG can debug multi-threaded applications, multi-process applications, and hybrid multi-threaded/multi-process applications, 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 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 66.

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 MPI_COMM_WORLD). 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. For more information on debugging threads, refer to "Threads-only Debugging," on page 67.

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. For more information on process debugging, refer to "Process-only Debugging," on page 67.

A hybrid, or multilevel, MPI/OpenMP program requires the use of both process and thread IDs to uniquely identify a particular thread. For more information on multilevel debugging, refer to "Multilevel Debugging," on page 67.

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 allowed but unnecessary.

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 9.1. 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
Multilevel	Multiple processes, at least one process employing multiple threads of execution

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. For a description of the PGDBG prompt, refer to "The PGDBG Command Prompt," on page 79.

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

To change debug mode manually, use the **pgienv** command.

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

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 9.1. 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

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 9.2. 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

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 9.3. 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

You use a process/thread set (p/t-set) 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 68) to define a p/t-set.

Named p/t-sets

In the following sections, you will notice frequent references 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. You can use the **focus** command to define the current p/t-set. Unless a prefix p/t-set overrides it, the current p/t set is used as the target p/t-set.

p/t-set Notation

The following rules describe 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|*}
```

Use *p/t-id* optional notation when process-only or threads-only debugging is in effect. For more information, refer to the **pgienv** command.

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 9.4. 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 9.5. 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 9.6. 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
`[1.1:3]`	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 the p/t-set is used. By default, a p/t-set is dynamic. 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 9.7. 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.

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

Example 9.8. 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.

Note

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 and depends on debug mode. For a description of the PGDBG prompt, refer to "The PGDBG Command Prompt," on page 79. You can use a p/t-set to prefix a command that overrides 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 that the target p/t-set is the prefix p/t-set, shown in this example 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. Also, using a prefix p/t-set does not change the current p/t-set.

p/t-set Commands

You can use the following commands to collect threads and processes into logical groups.

- Use **defset** and **undefset** to manage a list of named p/t-sets.
- Use **focus** to set the current p/t-set.
- Use **viewset** to view the active members described by a particular p/t-set, or to list all the defined p/t-sets.
- Use **whichsets** to describe the p/t-sets to which a particular process/thread belongs.

Table 9.2. 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 are 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 to contain only thread 0:

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

Use the **focus** command to change the current p/t-set to initial:

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

Advance the thread. Currently the code is not using a prefix p/t-set, and 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]
```

You can use the **focus** command 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 previous examples illustrate how to manage named p/t-sets using the command-line interface. A similar capability is available in the PGDBG GUI. "Command/Focus Panel," on page 7 contains information about the Focus Panel. This panel, shown in Figure 2.3, "PGDBG GUI Main Window with Focus Tab Selected ," on page 8, 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.

Operations on p/t set

You can create, select, modify and remove p/t sets.

Create a p/t set

To create a p/t set in the Focus Panel:

1. Click the Add button. This opens a dialog box similar to the one in Figure 9.1.
2. Enter the name of the p/t-set in the Focus Name text field and enter the p/t-set in the p/t-set text field.
3. Click the left mouse button on the OK button to add the p/t-set.

The new p/t-set appears in the Focus Table. Clicking the Cancel button or closing the dialog box aborts the operation. The Clear button clears the Focus Name and p/t-set text fields.

Select a p/t set

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 applies all commands entered in the Source Panel to the Current Focus when you choose Focus in the Apply Selector (“Main Window Combo Boxes,” on page 17). Current Focus can also be used in a GUI subwindow. Choose Current Focus in a subwindow’s Context Selector (“Subwindows,” on page 20) to display data for the Current Focus only.

Modify a p/t set

To modify an existing p/t-set, select the desired group in the Focus Table and click the Modify button. A dialog box similar to that in Figure 9.1 appears, except that the Focus Name and p/t-set text fields 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.

Remove a p/t set

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

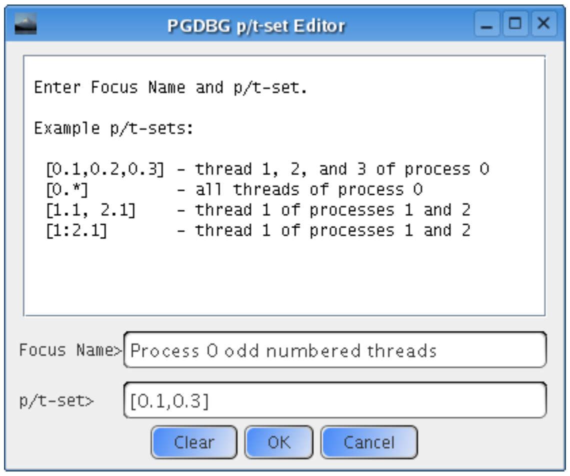
p/t set Usage

Note

p/t-sets defined in the Focus Panel of the PGDBG GUI are only used by the Apply and View 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 9.1 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.

Figure 9.1. Focus Group Dialog Box



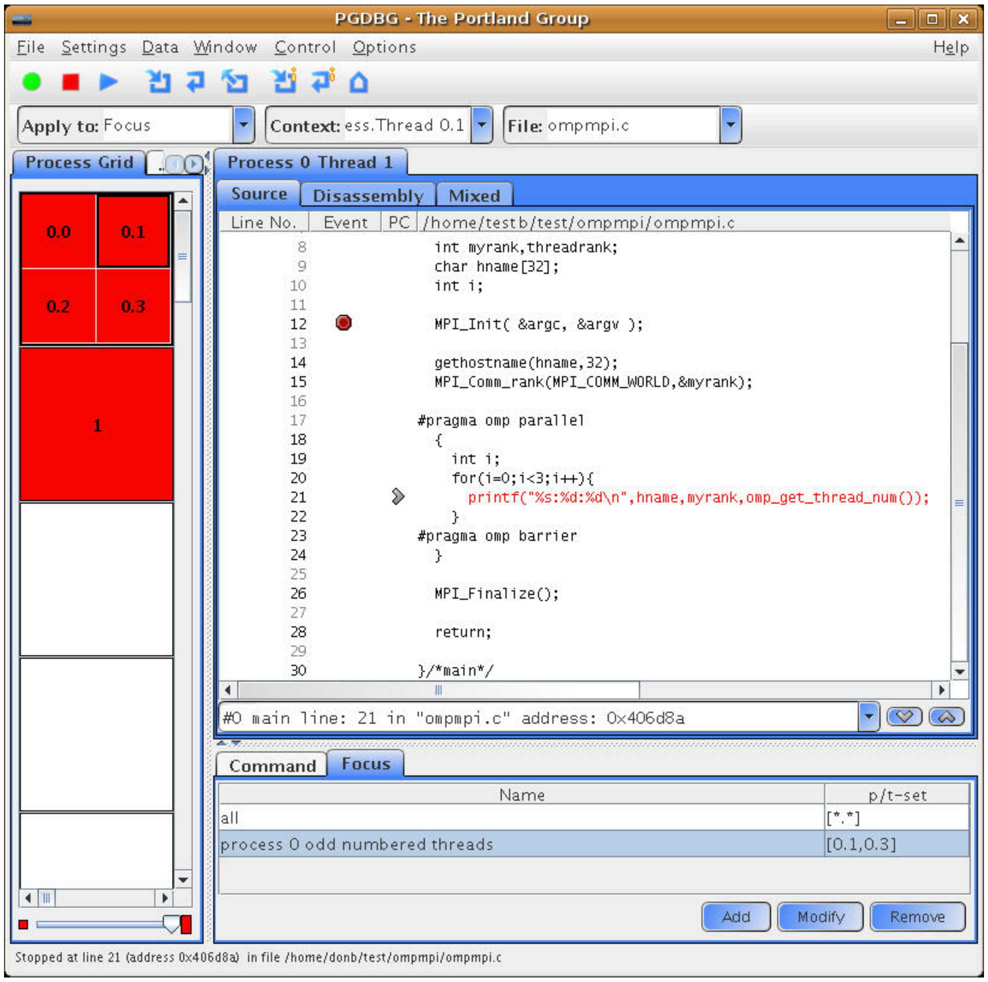
```
Enter Focus Name and p/t-set.

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 9.2 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 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 9.2. Focus in the GUI



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

    MPI_Init( &argc, &argv );

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

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

    MPI_Finalize();

    return;

  }/*main*/
```

```
#0 main line: 21 in "ompmpi.c" address: 0x406d8a
```

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

```
Stopped at line 21 (address 0x406d8a) in file /home/donb/test/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 9.3. PGDBG Parallel Commands

Commands	Action
Process Level Commands	Parallel by current p/t-set or prefix p/t-set
Thread Level Commands	Parallel by prefix p/t-set only; current p/t-set is ignored.
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	next	step	stepout	synci
halt	nexti	stepi	sync	wait

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

* 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 actions occur 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.

?	/	alias	arrive	breaks
call	catch	cd	debug	defset
delete	directory	disable	display	edit
enable	files	focus	funcs	help
history	ignore	log	pgienv	proc
procs	pwd	quit	repeat	rerun
run	script	shell	source	status
thread	threads	unalias	unbreak	undefset
use	viewset	wait	whereis	whichsets

Process and Thread Control

PGDBG supports thread and process control everywhere in the program. Threads and processes can be advanced in groups anywhere in the program.

The PGDBG control commands are:

cont	next	step	stepout	synci
halt	nexti	stepi	sync	wait

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` 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 does the same.

Configurable Stop Mode

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

Table 9.4. PGDBG Stop Modes

Command	Result
sync	Synchronous stop mode; when one thread stops at a breakpoint (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 accepts 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 does 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 brings up a prompt if it is available. The availability of the prompt is determined by the current wait mode and any pending **wait** commands.

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 9.5. 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.

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

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

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

  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. You can use the **wait** command 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. For more information, refer to pgienv usage in "Configurable Wait Mode," on page 76.

Table 9.6, "PGDBG Wait Behavior" describes the behavior of wait.

Suppose `S` is the target p/t-set. In the table,

- `P` is the set of all processes described by S.
- `p` is a single process.
- `T` is the set of all threads described by S.
- `t` is a single thread.

Table 9.6. PGDBG Wait Behavior

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

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 9.7. 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, as described in "PGDBG Debug Modes," on page 66.

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, described in "Thread Level Commands," on page 74.

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 fires 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, the following command is illegal:

```
pgdbg [all] 0> break func if (glob!=0) do {[*] set f = 0} 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 that executes for 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 in the example 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
endif
for the subset [s] of [*] where (i<10)
[s]n; [s]wait; [s]print i;
endfor
goto loop
```

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.

Chapter 10. PGDBG - Parallel Debugging with OpenMP

This chapter provides information on how to use PGDBG to debug OpenMP applications. Before reading this chapter, review the information in Chapter 9, “*PGDBG Parallel Debugging Overview*”.

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.

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. You can use the **procs** command 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 74, 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 (OS 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 **threads** command does not change 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 10.1. Thread State Is Described Using Color

Thread State	Description	Color
Stopped	If all threads are stopped at breakpoints, or were 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

Debugging OpenMP Private Data

PGDBG supports debugging of OpenMP private data for all supported languages. When an object is declared private in the context of an OpenMP parallel region, it essentially means that each thread team has 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 and displayed by PGDBG is as follows:

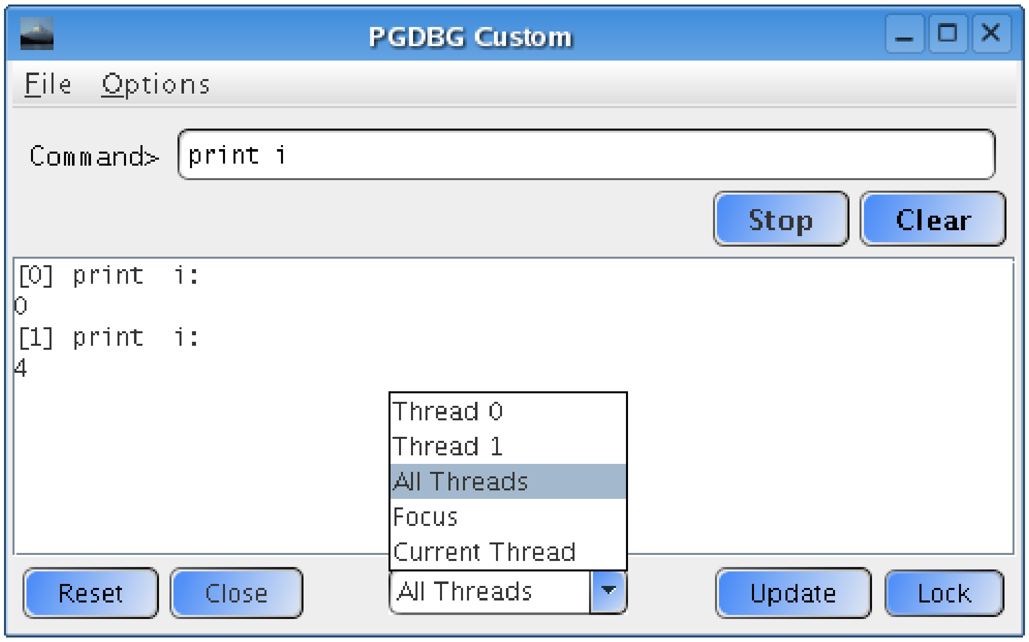
```
pgdbg [all] 0> [*] print i
[0] print i:
1
[1] print i:
5
```

The example specifies [*] for the p/t-set to execute the **print** command on all threads. Figure 10.1 shows the values for i in the PGDBG GUI using a Custom Window.

Note

All Threads is selected in the Context Selector to display the value on both threads.

Figure 10.1. OpenMP Private Data in PGDBG GUI



```
[0] print  i:
0
[1] print  i:
4
```

Note

All Threads is selected in the Context Selector to display the value on both threads.

Chapter 11. PGDBG Parallel Debugging with MPI

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

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 MPI_COMM_WORLD.

MPI debugging is supported on Linux and Windows 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 90 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.
- For a running process to receive a message, the sending process must be allowed to run.
- Process synchronization points, such as MPI_Barrier, do not return until all processes have hit the sync point.

- MPI_Finalize acts as an implicit barrier, though on MPICH-1 process 0 returns while Processes 1 through n-1 exit.

You can apply a control command, such as cont or step, 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 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 76 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 cleans 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 only successfully syncs 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, described in "Memory Access," on page 118. 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 illustrated in Figure 11.1 (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 11.1).

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
or set via the PGDBG command: pgienv mqslib <path>.
```

If this message is displayed, then the PGDBG_MQS_LIB_OVERRIDE environment variable should be set to the absolute path of libtvmpich.so or another shared object that is compatible with the version of MPI being used.

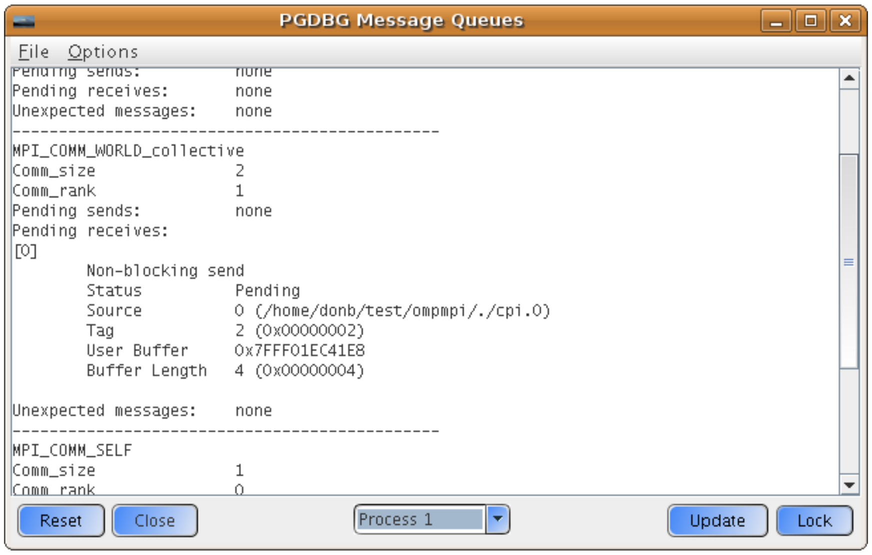
Note

The default path can be overwritten via the mqslib variant of the pgienv PGDBG command.

Note

mqdump is currently not supported on Windows.

Figure 11.1. Messages Subwindow



```
Pending sends:          none
Pending receives:       none
Unexpected messages:    none
----------------------------------------------
MPI_COMM_WORLD_collective
Comm_size               2
Comm_rank               1
Pending sends:          none
Pending receives:
[0]
        Non-blocking send
        Status          Pending
        Source          0 (/home/donb/test/ompmpi/./cpi.0)
        Tag             2 (0x00000002)
        User Buffer     0x7FFF01EC41E8
        Buffer Length   4 (0x00000004)

Unexpected messages:    none
----------------------------------------------
MPI_COMM_SELF
Comm_size               1
Comm_rank               0
```

MPI Groups

PGDBG identifies each process by its MPI_COMM_WORLD 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 MPICH-1 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 receive a ^C from the command line, which kills these processes and results 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.

If you use SSH as the mechanism for launching the remote components of PGDBG, you may want to do some additional configuration. The default configuration of ssh can result in a password prompt for each remote cluster node on which the debugger runs. You should check with your network administrator to make sure that you comply with your local security policies when configuring ssh. The following set of steps provide one way to configure SSH to eliminate this prompt.

```
$ ssh-keygen -t dsa
$ eval `ssh-agent -s`
$ ssh-add
<make sure that $HOME is not group-writable>
$ cd $HOME/.ssh
$ cp id_dsa.pub authorized_keys
```

Then for each cluster node you use in debugging, use:

```
$ ssh <host>
```

A few things that are important related to this example are these:

- The ssh-keygen command prompts for a passphrase. that is used to authenticate to the ssh-agent during future sessions. The passphrase can be anything you choose.
- Once you answer the prompts to make the initial connection, subsequent connections should not require further prompting.
- This example uses 'ssh-agent -s', which is correct for the sh or bash shells. For csh shells, use 'ssh-agent -c'.

After logging out and logging back in, the ssh-agent must be restarted and reauthorized. For example, in a bash shell, this is accomplished as follows:

```
$ eval `ssh-agent -s`
 $ ssh-add
```

You must enter the passphrase that was initially given to ssh-add to authenticate to the ssh-agent.

For further information, consult your SSH documentation.

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, MPICH2, and MVAPICH 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-1 using the PGDBG GUI is:

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

For TEXT mode debugging, be certain that the DISPLAY variable is undefined in the shell that is invoking mpirun. If this variable is set, you can undefine it by using one of the following commands:

For sh/bash users, use this command:

```
$ unset DISPLAY
```

For csh/tcsh users, use this command:

```
% unsetenv DISPLAY
```

To launch and debug an MPICH-2, MVAPICH, MSMPI, or HPMPI job, use this command:

```
% pgdbg [-text] -mpi[:<launcher>] <mpiexec_args> [ -program_args arg1,...argn ]
```

The default setting for `<launcher>` in `-mpi:<launcher>` is `mpiexec`.

Note

If the path for `<launcher>` is not part of the PATH environment variable, then you must specify the full path to the `<launcher>` command.

```
% pgdbg [-text] -mpi:mpiexec] <mpiexec_args> [ -program_args arg1,...argn ]
```

- For HPMPI, `<launcher>` is `mpirun`, so the need to instead specify `-mpi:mpirun`
- For MVAPICH, `<launcher>` is `mpirun_rsh`, so you specify `-mpi:mpirun_rsh`

When debugging an MPI job by invoking PGDBG with the -mpi option, each process is stopped before the first assembly instruction in the program. Continuing execution using `step` or `next` is not appropriate; instead, use the `cont` command.

Another way to invoke PGDBG for debugging an MPI job applies only to MPICH-2, though this invocation currently does not support TEXT mode:

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

Note

You cannot restart an MPI application from within PGDBG. You must exit the debugger and start a new debug session.

Newer versions of the Linux kernel supports a security feature that allows shared objects to be loaded at randomized addresses. Older versions of PGDBG assumed that a shared object loaded by an MPI application would be loaded at the same address for each process of an MPI job; thus incorrect mapping of symbols in shared objects could occur when this mode is enabled.

PGDBG now supports debugging of MPI jobs running on Linux kernels when this address randomization mode is enabled. However, when this mode is enabled, the current implementation of PGDBG does not share symbol table information associated with shared objects that are loaded by each process of an MPI job, which increases memory usage by PGDBG. Therefore, PGI recommends that this kernel mode be disabled on Linux clusters where PGDBG is used to debug MPI applications.

You can disable randomization mode by executing the following command as root on each node of the clusters:

```
sysctl -w kernel.randomize_va_space=0
```

Note

PGDBG emits a warning whenever it detects that it is being invoked on a multi-process MPI job when this kernel mode is enabled.

Using PGDBG for MPI Debugging

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. You can use the **procs** command to list the host and the PID of each process by rank. The current process is indicated by an arrow (=>). You can use the **proc** command 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 10.1, “Thread State Is Described Using Color,” on page 84 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 75 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 73 and “Thread Level Commands,” on page 74 for more information.

A license file distributed with PGDBG restricts the total number of MPI processes that can be debugged. There are internal limits on the number of threads per process that can be debugged.

Debugging Support for MPICH-1

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

Table 11.1. 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.
`-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.

Chapter 12. PGDBG Parallel Debugging of Hybrid Applications

PGDBG is a parallel application debugger capable of debugging multi-process MPI applications, multi-thread and OpenMP applications, as well as 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. This section concentrates on parallel debugging of hybrid 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 supported 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.

PGDBG Multilevel Debug Mode

As described in "PGDBG Debug Modes," on page 66, 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.

For multilevel debugging, you use the **pgienv** command to set the debug mode to multilevel.

```
pgienv mode multilevel
```

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 12.1. 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. Further, 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
```

For more information on p/t sets, refer to “Process/Thread Sets,” on page 68.

Chapter 13. PGDBG Command Reference

This chapter describes the PGDBG command set in detail, grouping the commands by these categories:

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

For an alphabetical listing of all the commands, with a brief description of each, refer to "Command Summary," on page 38 in "*PGDBG Command Summary*".

Notation Used in Command Sections

The command sections that follow use these conventions for the command names and arguments, when the command accepts one.

- Command names may be abbreviated by omitting the portion of the command name enclosed in brackets ([]).
- Argument names are italicized.
- Argument names are chosen to indicate what kind of argument is expected.
- 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, such as commas and quotes, must be entered as shown.

Example 13.1. Syntax examples

Example 1:

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

This syntax 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.

Example 2:

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

This syntax indicates that the command **attach** may be abbreviated to att, and, when invoked, must have a process ID argument, *pid*. Optionally you can specify an executable file, *exe*, or both an executable file and a host name, *host*.

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 "Basic Process and Thread Naming," on page 65 for more details.

attach

```
att[ach] pid [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. For example, `attach 1234` attempts 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 tries to attach to a process ID 1234 called `/home/demo/a.out` on a host named `myhost`.

PGDBG attempts 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 [command]
```

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 location of the current thread of the current process in the current program. For information on how processes are numbered, refer to "Multi-Process MPI Debugging," on page 90.

procs

```
procs
```

Print the status of all active processes, listing each process by its logical process ID.

quit

```
q[uit]
```

Terminate the debugging session.

rerun

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

Execute program from the beginning. If arguments arg0, arg1, and so on 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] [count | count]
```

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).

stepi

```
stepi [count | 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).

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

```
sy[nc] line | func
```

Advance to the specified source location, either the specified line or the first line in the specified function, ignoring any user-defined events.

synci

```
synci  addr | func
```

Advance to the specified address, or to the first address in the specified function, 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 68, for a detailed discussion of process-thread sets.

defset

```
defset name [p/t-set]
```

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  [p/t-set]
```

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

undefset

```
undefset [name | -all]
```

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 [name]
```

List the active members of the named process/thread set. If no process/thread set is given, list the active members of all defined process/thread sets.

whichsets

```
whichsets [p/t-set]
```

List all defined p/t-sets to which the members of a process/thread set belong. If no process/thread set is specified, the target process/thread set is used.

Events

The following commands deal with defining and managing events. See "Parallel Events," on page 80, 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}]
```

When no arguments are specified, the **break** command prints 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 table provides examples of using **break** to set breakpoints at various locations.

This break command...	Sets breakpoints...
`break 37`	at line `37` in the current file
`break "xyz.c"@37`	at line `37` in the file `xyz.c`
`break main`	at the first executable line of routine main
`break {addr 0xf0400608}`	at address `0xf0400608`
`break {line}`	at the current line
`break {pc}`	at the current address,

The following more sophisticated command stops when routine `xyz` is entered only if the argument `n` is greater than 10.

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

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

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

breaki

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

When no arguments are specified, the **breaki** command prints the current breakpoints. Otherwise, this command sets 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. T

The following table provides examples of setting breakpoints using **breaki**.

This break command...	Sets breakpoints...
`breaki 0xf0400608`	at address `0xf0400608`
`breaki {line 37}`	at line `37` in the current file
`breaki "xyz.c"@37`	at line `37` in the file `xyz.c`
`breaki main`	at the first executable address of routine main
`breaki {line}`	at the current line
`breaki {pc}`	at the current address,

In the following slightly more complex example, when `n` is greater than 3, the following command stops and prints the new value of n at address 0x6480:

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

breaks

```
breaks
```

Display all the existing breakpoints.

catch

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

When no arguments are specified, the **catch** command prints the list of signals being caught. With the sig:sig argument, this command catches 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 [ all | routine|  line | {addr addr}]
```

Clear breakpoints. With `all` argument, clear all breakpoints. With a `routine` argument, clear all breakpoints from the first statement in the specified routine, `routine`. With a line number argument, clear all breakpoints from the specified line number in the current source file. With an address argument, clear breakpoints from the specified address `addr`.

When no arguments are specified, the **clear** command clears all breakpoints at the current location.

delete

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

Delete the event `event-number` or all events. **delete 0** is the same as **delete all.** To specify multiple event numbers, separate the even numbers by commas.

When no arguments are given, the delete command lists all defined events by event number.

disable

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

When no arguments are specified, the **disable** command prints both enabled and disabled events.

With arguments, this command disables 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 the 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 an address nor a routine 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, described in the previous section.

enable

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

When no arguments are specified, the **enable** command prints both enabled and disabled events.

With arguments, this command enables the 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.

Note

Only one hardware watchpoint can be defined at a time.

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

hwatchread

```
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.

hwatchboth

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

When no arguments are specified, the **ignore** command prints the list of signals being ignored. With the sig:sig argument this command ignores the specified range of signals. With a list of signals the command ignores 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; rather, the signals are delivered to the target.

For information on intercepting signals, refer to “**catch**,” on page 103.

status

```
stat[us]
```

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

stop

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

Break when the value of the indicated variable `var` changes. Set a breakpoint at the indicated routine or line. 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 is true.

stopi

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

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

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 addr [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.

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

unbreak

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

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

unbreaki

```
unbreaki addr | routine | 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 is 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 is only 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
```

barely slows program execution at all, while

```
watch i
```

slows 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 assembly-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 80, 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] [ count | lo:hi | routine | 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 is interleaved with the disassembly. If an address and a count are given, disassemble count instructions starting at address addr.

edit

```
edit
edit filename
edit routine
```

Edit a file.

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] [ count | line,num |  lo:hi |  routine[,num] ]
```

Lists source code.

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. If a number is specified with routine, list the first num lines of 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 an assembly-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. The primary print commands are print and printf, described at the beginning of this section. The remainder of the commands are alternate commands that provide similar functionality to the print and printf commands.

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 using a comma-separated list. For example:

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

Prints this:

```
"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 1:

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

FORTRAN example 1:

```
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 2:

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

FORTRAN example 2:

```
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 3:

```
 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 3:

```
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 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 only prints 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; for example, 3 prints as `^c`. Otherwise, values that cannot 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 [ 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. For more information, refer to the **undisplay** command.

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 prints a maximum of 70 characters.

undisplay

```
undisplay 0 | all |  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 the expression `exp`. The variable var can be any valid identifier accessed properly for the current scope. For example, given a C variable declared ‘int * i’, you can use the following command to assign the value 9999 to it.

```
as *i = 9999
```

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 are 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 stops the execution and asks 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 prints 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 returns the register state of each thread back to the state they had before invoking the **call** command. Answering no to this question leaves 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 provides 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] [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` returns the size in 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 provides 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", for a discussion of scope meaning and conventions.

decls

```
decls [routine | "sourcefile" |  {global} ]
```

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

down

```
down [number]
```

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

enter

```
en[ter] [routine | "sourcefile" |  {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 [routine | "sourcefile" |  {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 51, 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 prints 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 address of the pointer, the value of the pointer, 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. For more information on mqdum, refer to "MPI Message Queues," on page 88.

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]  [n | {line n} | routine | var |  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] [[addr...] | [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] [ n | routine |  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]  [ 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 both a name and string are given, make name an alias for string. Subsequently, whenever name is encountered it is replaced by string.

Although string may be an arbitrary string, name must not contain any space characters.

For example, the following statement creates an alias for xyz.

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

Now whenever xyz is typed, PGDBG responds 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, the following statement adds the directory `morestuff` to the list of directories to be searched.

```
dir morestuff
```

Now, source files stored in `morestuff` are accessible to PGDBG.

If the first character in pathname is ~, then $HOME replaces that character.

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.

Table 13.1. pgienv Commands

Use this command...	To do this...
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

Use this command...	To do this...
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 threadstopconfig auto	For each process, debugger sets thread stopping mode to 'sync' in serial regions, and 'async' in parallel regions
pgienv threadstopconfig 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 threadwaitconfig auto	For each process, the debugger sets the thread wait mode to ‘all’ in serial regions and ‘none’ in parallel regions. (default)
pgienv threadwaitconfig user	The thread wait mode is user-defined and remains unchanged by the debugger.
pgienv mqslib default	Determine MPI message queue debug library by inspecting executable.
pgienv mqslib <path>	Determine MPI message queue debug library to <path>.

Use this command...	To do this...
pgienv verbose <bitmask>	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 |  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 ~, then the value of $HOME replaces that character.

Part II. PGPROF Profiler

In Part I you learned how to use the PGI debugger, PGDBG. Part II of the *PGI Tools Guide* is about the PGPROF profiler. The information in this part describes PGPROF, 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 execution time they consumed.

- Chapter 14, “Getting Started with the PGPROF Profiler,” starting on page 127, contains information on how to start using the profiler, including a description of the profiling process, as well as how to invoke and initialize the profiler.
- Chapter 15, “Using PGPROF ,” starting on page 133, describes how to choose a profiling method, build your program, and execute it to collect profile data.
- Chapter 16, “Compiler Options for Profiling,” starting on page 155, describes the compiler options used for profiling, and howthey are interpreted.
- Chapter 17, “PGPROF Command Line Options for Profiling,” starting on page 161, describes PGPROF command-line options used for profiling, and provides sample invocations and startup commands.
- Chapter 18, “PGPROF Environment Variables,” starting on page 165, contains information on environment variables that you can set to control the way profiling is performed in PGPROF.
- Chapter 19, “PGPROF Data and Precision,” starting on page 167, contains descriptions of the profiling mechanism that measures time, how statistics are collected, and the precision of the profiling results.
- Chapter 20, “PGPROF Reference,” starting on page 171, provides reference information about each of the features of the PGPROF performance profiler.
- Chapter 21, “The PGPROF Command Line Interface,” starting on page 191, describes the PGPROF profiler command line interface, providing both a summary table and then more details about the commands. The table includes the command name, the arguments for the command, as well as a brief description of the command.
- Chapter 22, “pgcollect Reference,” starting on page 197, describes the PGPROF command line options and how to use them to configure and control collection of application performance data.

Chapter 14. Getting Started with the PGPROF Profiler

This chapter describes the PGPROF profiler. PGPROF provides a way to visualize and diagnose the performance of the components of your program. Using tables and graphs, PGPROF associates execution time with the source code and instructions of your program, allowing you to see where and how execution time is spent. Through resource utilization data and compiler feedback information, PGPROF also provides features for helping you to understand why certain parts of your program have high execution times.

You can also use the PGPROF profiler to profile parallel programs, including multiprocess MPI programs, multi-threaded programs such as OpenMP programs, or a combination of both. PGPROF provides views of the performance data for analysis of MPI communication, multiprocess and multi-thread load balancing, and scalability.

Using the Common Compiler Feedback Format (CCFF), PGI compilers save information about how your program was optimized, or why a particular optimization was not made. PGPROF can extract this information and associate it with source code and other performance data, allowing you to view all of this information simultaneously. PGPROF also supports a *feedbackonly* mode, which allows you to browse Compiler Feedback in the absence of a performance profile.

Each performance profile depends on the resources of the system where it is run. PGPROF provides a summary of the processor(s) and operating system(s) used by the application during any given performance experiment.

Basic Profiling

Performance profiling can be considered a two-stage process.

- In the first stage, you collect performance data when your application runs using typical input.
- In the second stage, you analyze the performance data using PGPROF.

There are a variety of ways to collect performance data from your application. For basic execution-time profiling, we recommend that you use the **pgcollect** tool, which has several attributes that make it a good choice:

- You don't have to recompile or relink your application.
- Data collection overhead is low.
- It is simple to use.
- It supports multi-threaded programs.
- It supports shared objects, DLLs, and dynamic libraries.

Note

In release 2010, PGI's performance data collection tools are in transition. To verify if PGI supports **pgcollect** on your platform, refer to the release notes for your PGI software or run the following command:

```
pgcollect -help
```

If **pgcollect** is not supported in your PGI installation, check *www.pgroup.com* for updates.

To profile your application named 'myprog', you execute the following commands:

```
    $ pgcollect myprog
    $ pgprof -exe myprog
```

The information available to you when you analyze your application's performance can be significantly enhanced if you compile and link your program using the `–Minfo=ccff` option. This option saves information about the compilation of your program, compiler feedback, for use by PGPROF. For more information on compiler feedback, refer to "Compiler Feedback," on page 139.

For a more complete analysis, our command execution might look similar to this:

```
    $ pgf90 -fast -Minfo=ccff -o myprog myprog.90
    $ pgcollect myprog
    $ pgprof -exe myprog
```

Methods of Collecting Performance Data

PGI provides a number of methods for collecting performance data in addition to the basic **pgcollect** method described in the previous section. Some of these have advantages or capabilities not found in the basic **pgcollect** method. We divide these methods into two categories: instrumentation-based profiling and sample-based profiling.

Instrumentation-based Profiling

Instrumentation-based profiling is one way to measure time spent executing the functions or source lines of your program. The compiler inserts timer calls at key points in your program and does the bookkeeping necessary to track the execution time and execution counts for routines and source lines. This method is available on all platforms on which PGI compilers are supported.

Instrumentation-based profiling:

- Provides exact call counts.

- Provides exact line/block execution counts.
- Reports time attributable to only the code in a routine.
- Reports time attributable to the code in a routine and all the routines it called.

This method requires that you recompile and relink your program using one of these compiler options:

- Use `-Mprof=func` for routine-level profiling.

  Routine-level profiling can be useful in identifying which portions of code to analyze with line-level profiling.
- Use `-Mprof=lines` for source line-level profiling.

  The overhead of using line-level profiling can be high, so it is more suited for fine-grained analysis of small pieces of code, rather than for analysis of large, long-running applications.

Sample-based Profiling

Sample-based profiling uses statistical methods to determine the execution time and resource utilization of the routines, source lines, and assembly instructions of the program. Sample-based profiling is less intrusive than instrumentation-based profiling, so profiling runs take much less time. Further, in some cases it is not necessary to rebuild the program.

Note

The basic **pgcollect** method described earlier in "Basic Profiling" is a time-based sampling method. **Pgcollect** also supports event-based profiling on linux86-64.

The following sections describe both time-based and event-based sampling. For information on the differences in how instrumentation- and sample- based profiling measure time, refer to "Measuring Time," on page 167.

Time-based Sampling

With time-based sampling the program's current instruction address (program counter) is read, and tracked, at statistically significant intervals. Instruction addresses where a lot of time is spent during execution are read numerous times. The profiler can map these addresses to source lines and/or functions in your program, providing an easy way to navigate from the function where the most time is spent, to the line, to the assembly instruction.

You can build your program using the -Mprof=time compiler option for time-based sampling of single-threaded Linux programs. When using -Mprof=time, you are required only to re-link your program. However, unless you compile with -Minfo=ccff, compiler feedback will not be available.

- As described previously in "Basic Profiling", we recommend using **pgcollect** for time-based profiling. Building with `-Minfo=ccff` is not required, but will improve the profile information available by including compiler feedback.
- You can build your program using the `-Mprof=time` compiler option for time-based sampling of single-threaded Linux programs. When using `-Mprof=time`, you are required only to re-link your program. However, compiler feedback is available only if you compile with `-Minfo=ccff`.

Event-based Sampling

As well as reading the program's instruction address, event-based sampling uses various methods to read and track the values of selected hardware counters. These counters track processor events such as data cache misses and floating point operations. You can use this information to help determine not just that time is being spent in a particular block of code, but why so much time is spent there. If there is bottleneck related to a particular resource, such as the L2 data cache, these counters can help you discover where the bottleneck is occurring.

Event-based sampling requires that certain system software be co-installed with the PGI software on the Linux system. Either the Linux kernel must have been built to support PAPI (the Performance API), or the kernel must have a performance tool named *OProfile* installed and operational.

OProfile is a performance profiling utility for Linux systems. It runs in the background collecting information at a low overhead and providing profiles of code based on processor hardware events. When installed, **pgcollect** collects this type of performance data for analysis with PGPROF. For more information on OProfile, see http://oprofile.sourceforge.net/.

PAPI, Performance Application Programming Interface, provides the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to collect low level performance metrics, such as instruction counts, clock cycles, and cache misses, of computer systems running UNIX/Linux operating systems.

- Use `-Mprof=hwcts` for event-based sampling with PAPI
- Run your program using the **pgcollect** command for event-based sampling with OProfile.

Note

MPI performance data may be collected in concert with all of these methods except **pgcollect** sampling.

Choose Profile Method

Use the following guidelines to decide which performance data collection method to use:

- A good starting point for any performance analysis is to use time-based sampling with **pgcollect**, as described in "Basic Profiling," on page 127.
- If you are profiling an MPI application on Linux, build your application using `-Mprof=time,<mpi>`, where <mpi> is the supported MPI distribution that you are using, for example, mpich1.
- If your MPI application also uses OpenMP or multiple threads per process and you want to determine where the majority of time is spent, build with `-Mprof=func,<mpi>`. Then build that portion of the program with `-Mprof=lines,<mpi>` to isolate the performance problem.
- If you want exact execution counts, build with `-Mprof=func` or `-Mprof=lines`.
- On Linux86-64 platforms on which either PAPI or OProfile is installed, once you have collected a time-based profile using either instrumentation- or sampling-based profiling, consider further examining the resource utilization of those portions of code where the most time is spent. Do this with event-based

sampling, using either the `-Mprof=hwcts` compiler option with PAPI or the **pgcollect** command with event-based sampling options , as described in Chapter 22, “*pgcollect Reference*”.

Collect Performance Data

To obtain the performance data required for PGPROF, you must run your program.

- If you use any method other than the **pgcollect** command to collect data, run your program normally using a representative input data set.
- If you use the **pgcollect** command to collect data, refer to “Basic Profiling,” on page 127 for information of how to execute a profiling run of your program. For specific details on **pgcollect**, refer to Chapter 22, “*pgcollect Reference*”

Profiling Output File

In all profiling methods, once the program's profiling run is complete, a file named `pgprof.out` is written to the program's working directory. This file contains the performance data used by the PGPROF profiler to analyze the program's performance.

Using System Environment Variables

You can use system environment variables to change the way profiling is performed. For more information on these variables, refer to Chapter 18, “*PGPROF Environment Variables*”.

Profiling MPI and Multi-threaded Programs

MPI profiling is available only on Linux and Windows HPC Server. To learn more about profiling MPI programs, refer to “Profiling MPI Programs”.

To learn more about profiling multi-threaded programs, refer to “Profiling Multi-threaded Programs”.

Profiling with Hardware Event Counters

You can also profile using hardware event counters. For more specific information on this type of profiling, refer to “Profiling Resource Utilization with Hardware Event Counters,” on page 146.

Profiler Invocation and Initialization

The PGPROF profiler is invoked as follows:

```
% pgprof.exe [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, 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.

Probably the most common method to invoke the profiler is this:

```
% pgprof -exe <execname>
```

When you use this command to launch PGPROF:

- If a `pgprof.out` file exists in the current directory, PGPROF tries to open it and use <execname>. Further, the GUI is populated according to profile data, if valid.
- If no `pgprof.out` file exists in the current directory, the GUI is not populated and no dialog appears. Further, when the user selects the menu `File | New Profiling Session...`, then the Text Field for `Executable` is set with <*execname*> in the dialog.

For information on all available profiler options and how they are interpreted, refer to Chapter 16, "*Compiler Options for Profiling*". For information on the command line options for the Profiler, refer to Chapter 17, "*PGPROF Command Line Options*". For sample launch commands; refer to "Profiler Invocation and Startup," on page 162.

Application Tuning

So how do you make your program faster? Tuning your program ranges from simple to complex.

- In the simple case, you may be able to easily tune the application and improve performance dramatically simply by adding a compiler option when you build. The Compiler Feedback and System Information tabs in the PGPROF user interface contain information that can help identify these situations.
- In a slightly more challenging scenario, you may need to restructure part of your code to allow the compiler to optimize it more effectively. For instance, the Compiler Feedback for a given loop may provide a hint to remove a call from the loop. If the call can be moved out of the loop or inlined, the loop might be vectorized by the next compile.
- More difficult cases involve memory alignment and algorithm restructuring. These issues are beyond the scope of this manual.

Troubleshooting

If you are having trouble during invocation or the initialization process, use the following sections for tips on what might be causing your problem.

Selecting a Version of Java

PGPROF (both GUI and command line) depends on Java. PGPROF requires that the Java Virtual Machine be a specific minimum version or above. For the MAC OS X, PGPROF uses the JAVA that comes with the system. For all other systems, by default, PGPROF uses the version of Java installed with your PGI software. If you chose not to install Java when installing your PGI software, PGPROF looks for Java on your PATH. These default Java executables can be overridden 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, use this command to specify the location of Java:

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

Slow Network

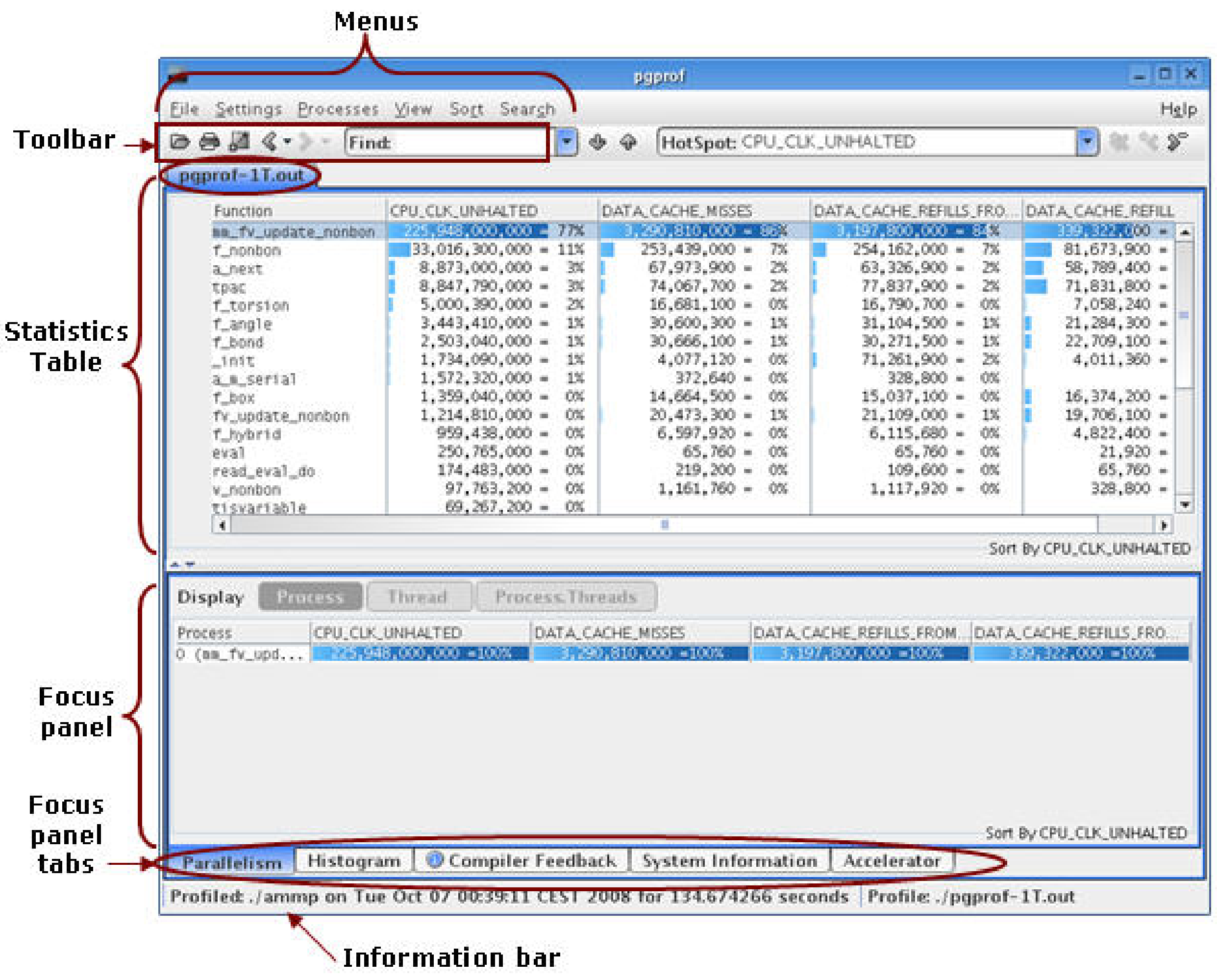
If you are viewing a profile across a slow network connection, of a connection that does not support remote display of Java GUI, consider using the PGPROF command-line interface, described in Chapter 21, "*The PGPROF Command Line Interface*," on page 191.

Chapter 15. Using PGPROF

In Chapter 14, “*Getting Started with the PGPROF Profiler*” you learned how to choose a profiling method, build your program, and execute it to collect profile data. This chapter provides a more detailed description of how to use the features of PGPROF, in particular:

- Profile navigation
- HotSpot navigation
- Sorting profile data
- Compiler Feedback
- Profiling parallel programs, including multi-threaded and MPI programs
- Scalability comparison
- Profiling resource utilization with hardware event counters
- Profiling accelerator programs

Figure 15.1. PGPROF Overview


Menus
Toolbar
Statistics Table
Focus panel
Focus panel tabs
Information bar
pgprof
File Settings Processes View Sort Search Help
Find:
HotSpot: CPU_CLK_UNHALTED
pgprof-1T.out
Display Process Thread Process.Threads
Parallelism Histogram Compiler Feedback System Information Accelerator

Function	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FRO...	DATA_CACHE_REFILL
ma_fv_update_nonbon	225,948,000,000 = 77%	3,290,810,000 = 86%	3,197,800,000 = 84%	339,322,000 =
f_nonbon	33,016,300,000 = 11%	253,439,000 = 7%	254,162,000 = 7%	81,673,900 =
a_next	8,873,000,000 = 3%	67,973,900 = 2%	63,326,900 = 2%	58,789,400 =
tpac	8,847,790,000 = 3%	74,067,700 = 2%	77,837,900 = 2%	71,831,800 =
f_torsion	5,000,390,000 = 2%	16,681,100 = 0%	16,790,700 = 0%	7,058,240 =
f_angle	3,443,410,000 = 1%	30,600,300 = 1%	31,104,500 = 1%	21,284,300 =
f_bond	2,503,040,000 = 1%	30,666,100 = 1%	30,271,500 = 1%	22,709,100 =
_init	1,734,090,000 = 1%	4,077,120 = 0%	71,261,900 = 2%	4,011,360 =
a_m_serial	1,572,320,000 = 1%	372,640 = 0%	328,800 = 0%	
f_box	1,359,040,000 = 0%	14,664,500 = 0%	15,037,100 = 0%	16,374,200 =
fv_update_nonbon	1,214,810,000 = 0%	20,473,300 = 1%	21,109,000 = 1%	19,706,100 =
f_hybrid	959,438,000 = 0%	6,597,920 = 0%	6,115,680 = 0%	4,822,400 =
eval	250,765,000 = 0%	65,760 = 0%	65,760 = 0%	21,920 =
read_eval_do	174,483,000 = 0%	219,200 = 0%	109,600 = 0%	65,760 =
v_nonbon	97,763,200 = 0%	1,161,760 = 0%	1,117,920 = 0%	328,800 =
tisvariable	69,267,200 = 0%			

Sort By CPU_CLK_UNHALTED

Process	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FROM...	DATA_CACHE_REFILLS_FRO...
0 (ma_fv_upd...	225,948,000,000 =100%	3,290,810,000 =100%	3,197,800,000 =100%	339,322,000 =100%

Sort By CPU_CLK_UNHALTED

Profiled: ./ammp on Tue Oct 07 00:39:11 CEST 2008 for 134.674266 seconds | Profile: ./pgprof-1T.out

PGPROF Tabs and Icons Overview

Before we describe how to navigate within PGPROF, it is useful to have some common terminology for the tabs and icons that you see within the application.

Closeable and Non-closeable Tabs

PGPROF displays both closeable and non-closeable tabs. For example, when you first invoke PGPROF, you see a window with a non-closeable tab. Then, to access profiling data specific to a given function, you double-click on it and a *closeable* tab opens with source code and profiling statistics for that function. this closeable tab navigation approach provides a way for you to easily view a variety of information quickly.

PGPROF Common Icons

Table 15.1 provides a summary of the common icons you see during profile navigation.

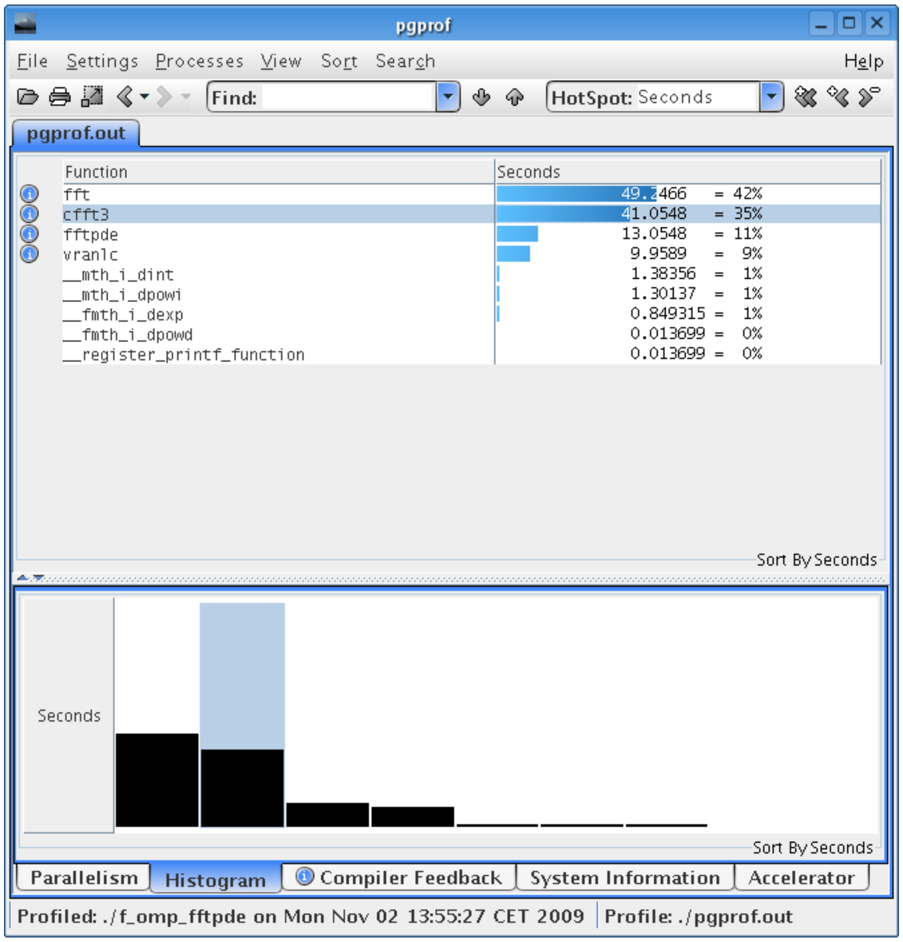
Table 15.1. PGPROF Icon Summary

Click this icon...	to...
⊕	Display the corresponding assembly code for this line.
⊖	Hide the corresponding assembly code for this line.
✕	Close the tab on which it is displayed.
ⓘ	Display the CCFF, common compiler feedback format, information for this line.

Profile Navigation

When you first invoke PGPROF, it displays top-level profiling information in a non-closeable tab, as illustrated in Figure 15.2.

Figure 15.2. PGPROF Initial View



```
Function                         Seconds
fft                              49.2466  = 42%
cfft3                            41.0548  = 35%
fftpde                           13.0548  = 11%
vranlc                            9.9589  =  9%
__mth_i_dint                      1.38356 =  1%
__mth_i_dpowi                     1.30137 =  1%
__fmth_i_dexp                     0.849315 = 1%
__fmth_i_dpowd                    0.013699 = 0%
__register_printf_function        0.013699 = 0%
```

Profiled: ./f_omp_fftpde on Mon Nov 02 13:55:27 CET 2009 | Profile: ./pgprof.out

This tab shows the Statistics Table containing a routine list in the Function column and performance data associated with each routine in the Time column. This list is sorted by the Time value, assuming there is such a value in the profile data. Time values may include hardware counters associated with time, such as CPU_CLK_UNHALTED.

To access profiling data specific to a given function, you double-click on it and a *closeable* tab opens with source code and profiling statistics for that function.

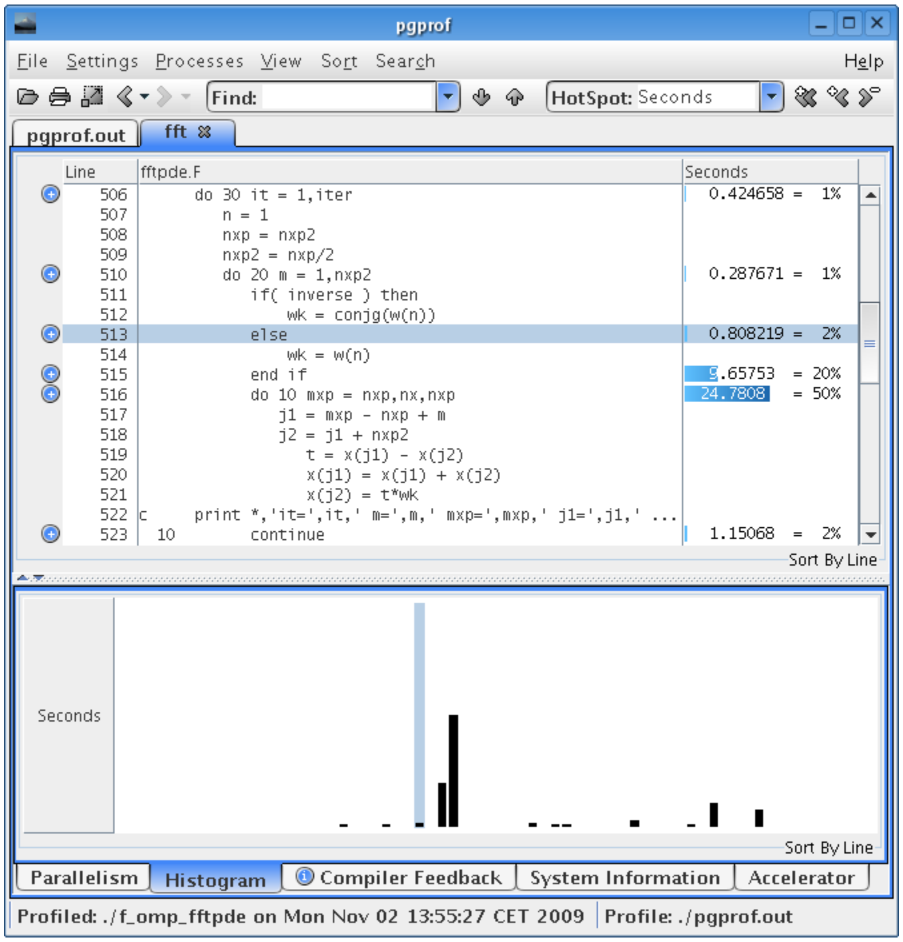
By default, PGI compilers include enough symbol information in executables to allow PGPROF to display performance data at the source line level as well as at the routine level. However, if you compiled with the option `–Mnodwarf`, or if you built your program using another compiler, you may only be able to access the routine-level view.

- To zoom in to the line level for a particular routine, double-click the function name.

  This action opens a tab that displays profiling data specific to the given function. The tab label is the function name followed by an **x**. You use the **x** to close the tab when you no longer want to view that information.

  In this tab, PGPROF displays the source code for that routine, together with the performance data for each line. For example, is you double-click on the function `fft`, PGPROF displays a new tab labelled `fft x` that contains the source code for that function, as illustrated in Figure 15.3.

Figure 15.3. Source Code View



```
Line  fftpde.F                                                Seconds
 506        do 30 it = 1,iter                                  0.424658 =  1%
 507           n = 1
 508           nxp = nxp2
 509           nxp2 = nxp/2
 510           do 20 m = 1,nxp2                                0.287671 =  1%
 511              if( inverse ) then
 512                 wk = conjg(w(n))
 513              else                                         0.808219 =  2%
 514                 wk = w(n)
 515              end if                                       9.65753  = 20%
 516              do 10 mxp = nxp,nx,nxp                       24.7808  = 50%
 517                 j1 = mxp - nxp + m
 518                 j2 = j1 + nxp2
 519                    t = x(j1) - x(j2)
 520                    x(j1) = x(j1) + x(j2)
 521                    x(j2) = t*wk
 522 c      print *,'it=',it,' m=',m,' mxp=',mxp,' j1=',j1,' ...
 523    10        continue                                     1.15068  =  2%
```

Profiled: ./f_omp_fftpde on Mon Nov 02 13:55:27 CET 2009 | Profile: ./pgprof.out

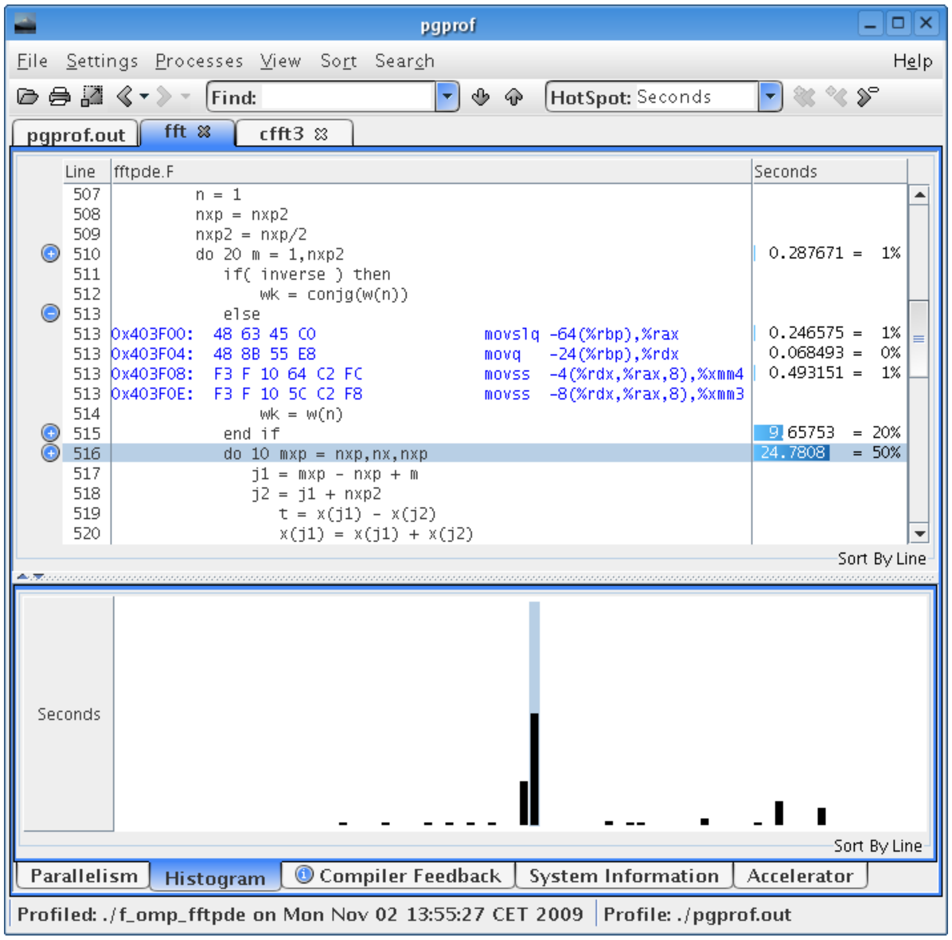
Because your program is probably optimized, you may notice that performance data is only shown for a subset of the source lines. For example, a multi-line loop may only have line-level data for the first line of the loop.

In the optimization process, the compiler may significantly reorder the assembly instructions used to implement the loop, making it impossible to associate any given instruction with a line in the loop. However, it is possible to associate all of a loop's instructions with that loop, so all of the performance data for the loop is associated with a single "line". For example, in Figure 15.3, "Source Code View," on page 136, line 516 contains the information for the entire **do** loop.

- To zoom in to the assembly level for a particular source line, click the plus symbol (+) in the row of the Statistics Table containing that source line.

  PGPROF displays the routine with assembly code interspersed with the source lines with which that assembly code is associated, as Figure 15.4 illustrates for the **else** portion of the code in the function `fft`.

Figure 15.4. Assembly Level View



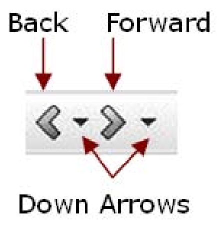
Line	fftpde.F		Seconds
507	`n = 1`		
508	`nxp = nxp2`		
509	`nxp2 = nxp/2`		
510	`do 20 m = 1,nxp2`		0.287671 = 1%
511	`if( inverse ) then`		
512	`wk = conjg(w(n))`		
513	`else`		
513	0x403F00: 48 63 45 C0	movslq -64(%rbp),%rax	0.246575 = 1%
513	0x403F04: 48 8B 55 E8	movq -24(%rbp),%rdx	0.068493 = 0%
513	0x403F08: F3 F 10 64 C2 FC	movss -4(%rdx,%rax,8),%xmm4	0.493151 = 1%
513	0x403F0E: F3 F 10 5C C2 F8	movss -8(%rdx,%rax,8),%xmm3	
514	`wk = w(n)`		
515	`end if`		9.65753 = 20%
516	`do 10 mxp = nxp,nx,nxp`		24.7808 = 50%
517	`j1 = mxp - nxp + m`		
518	`j2 = j1 + nxp2`		
519	`t = x(j1) - x(j2)`		
520	`x(j1) = x(j1) + x(j2)`		

Profiled: ./f_omp_fftpde on Mon Nov 02 13:55:27 CET 2009 | Profile: ./pgprof.out

PGPROF displays performance data associated with a specific assembly instruction in the row of the Statistics Table containing that instruction.

- To return to a previous view, use the Back button ("<") in the Toolbar, just below the Menus.

Figure 15.5. View Navigation Buttons



The Back and Forward buttons work much like those found in web browsers, moving to previous and next views, respectively.

- To select and jump to a specific view, use the *down arrow* on each of the Forward and Back buttons.

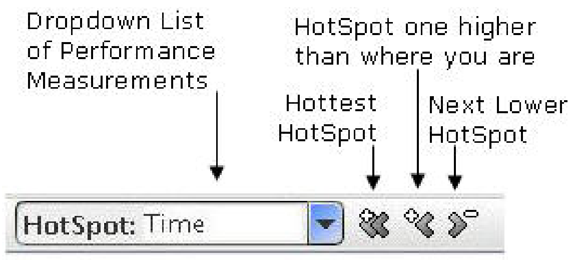
Note

You can have multiple function views open at a time, as illustrated in Figure 15.4, where tabs for both functions `fft` and `cfft3` are displayed.

HotSpot Navigation

In addition to the HotSpot navigation controls on the toolbar, illustrated in Figure 15.6, you can find the performance-critical parts of your program using the Histogram tab which shows clickable bar graphs of the performance data plotted against the address range of the program.

Figure 15.6. HotSpot Navigation Controls


Dropdown List of Performance Measurements
HotSpot one higher than where you are
Hottest HotSpot
Next Lower HotSpot
HotSpot: Time

- The HotSpot navigation controls in the Toolbar are usually the quickest way to locate a hot spot. By *hot spot* we mean a program location that has a high value for some performance measurement such as Time, Count, and so on.

  To locate the hotspot, select the desired performance measurement in the HotSpot drop-down menu in the Toolbar, then click on the "Hottest" button ("<<+") to select the highest value for that measurement in the current view.

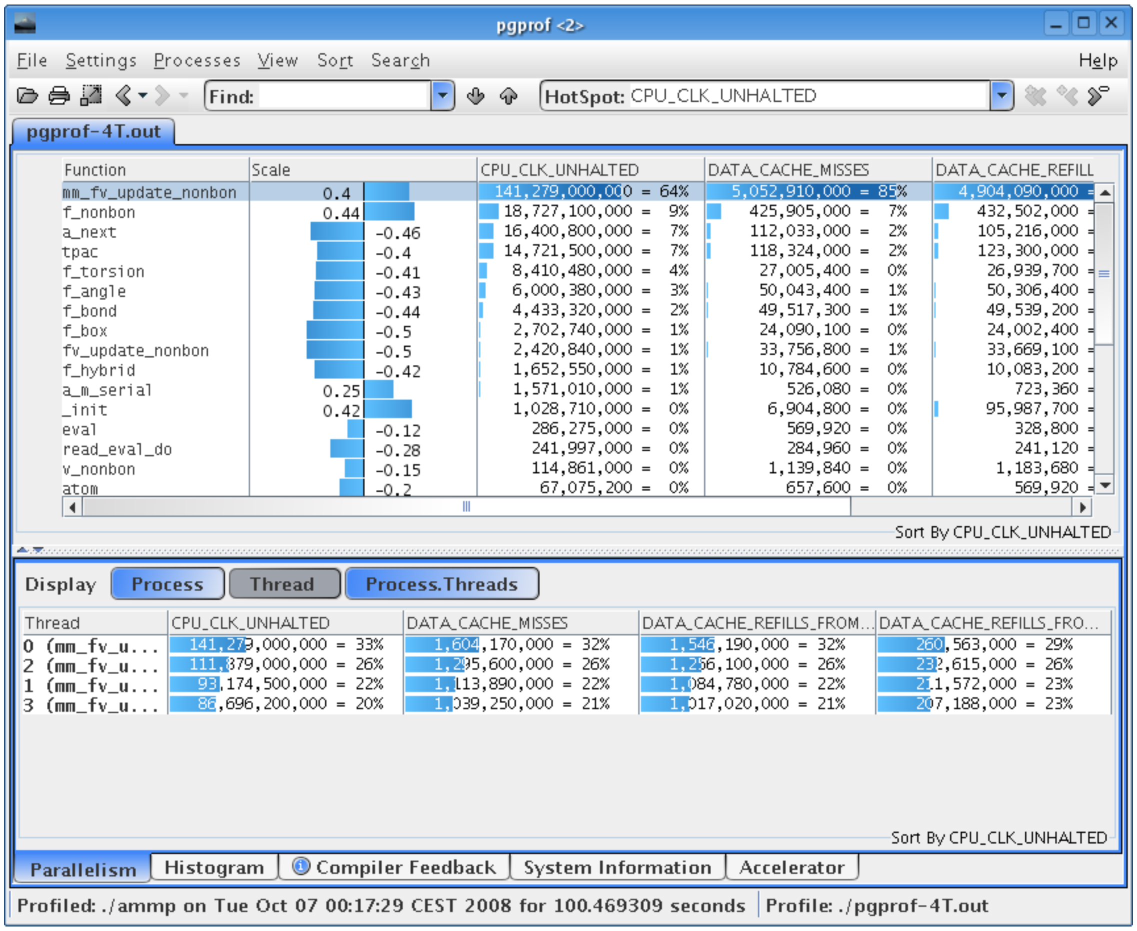
- To find a HotSpot using the Histogram, click on the Histogram tab. In the histogram for the measurement you are interested in, click on the tallest bar. The corresponding row in the Statistics Table will be selected.

Sorting Profile Data

PGPROF maintains a consistent sort order for the Statistics Table and the Histogram tab. Changing the sort order for any one of these changes it for all of them. The sort order can be changed by using the Sort Menu, as described in "Sort Menu," on page 177 or by clicking the column header in the Statistics Table or the row header in the Histogram tab.

The current sort order, such as sorting by the CPU Clock time, is displayed at the bottom of each table. For example, the message `Sort By CPU_CLK_UNHALTED` is present at the bottom of both tables in Figure 15.7.

Figure 15.7. Sort View


pgprof <2>
File Settings Processes View Sort Search Help
Find:
HotSpot: CPU_CLK_UNHALTED
pgprof-4T.out

Function	Scale	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILL
mm_fv_update_nonbon	0.4	141,279,000,000 = 64%	5,052,910,000 = 85%	4,904,090,000 =
f_nonbon	0.44	18,727,100,000 = 9%	425,905,000 = 7%	432,502,000 =
a_next	-0.46	16,400,800,000 = 7%	112,033,000 = 2%	105,216,000 =
tpac	-0.4	14,721,500,000 = 7%	118,324,000 = 2%	123,300,000 =
f_torsion	-0.41	8,410,480,000 = 4%	27,005,400 = 0%	26,939,700 =
f_angle	-0.43	6,000,380,000 = 3%	50,043,400 = 1%	50,306,400 =
f_bond	-0.44	4,433,320,000 = 2%	49,517,300 = 1%	49,539,200 =
f_box	-0.5	2,702,740,000 = 1%	24,090,100 = 0%	24,002,400 =
fv_update_nonbon	-0.5	2,420,840,000 = 1%	33,756,800 = 1%	33,669,100 =
f_hybrid	-0.42	1,652,550,000 = 1%	10,784,600 = 0%	10,083,200 =
a_m_serial	0.25	1,571,010,000 = 1%	526,080 = 0%	723,360 =
_init	0.42	1,028,710,000 = 0%	6,904,800 = 0%	95,987,700 =
eval	-0.12	286,275,000 = 0%	569,920 = 0%	328,800 =
read_eval_do	-0.28	241,997,000 = 0%	284,960 = 0%	241,120 =
v_nonbon	-0.15	114,861,000 = 0%	1,139,840 = 0%	1,183,680 =
atom	-0.2	67,075,200 = 0%	657,600 = 0%	569,920 =

Sort By CPU_CLK_UNHALTED

Thread	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FROM...	DATA_CACHE_REFILLS_FRO...
0 (mm_fv_u...	141,279,000,000 = 33%	1,604,170,000 = 32%	1,546,190,000 = 32%	260,563,000 = 29%
2 (mm_fv_u...	111,879,000,000 = 26%	1,295,600,000 = 26%	1,256,100,000 = 26%	232,615,000 = 26%
1 (mm_fv_u...	93,174,500,000 = 22%	1,113,890,000 = 22%	1,084,780,000 = 22%	211,572,000 = 23%
3 (mm_fv_u...	86,696,200,000 = 20%	1,039,250,000 = 21%	1,017,020,000 = 21%	207,188,000 = 23%

Sort By CPU_CLK_UNHALTED

Profiled: ./ammp on Tue Oct 07 00:17:29 CEST 2008 for 100.469309 seconds | Profile: ./pgprof-4T.out

Compiler Feedback

The PGI compilers generate a special kind of information that is saved inside the executable file so that it is available to tools, such as PGPROF, to help with program analysis. A compiler discovers a lot about a program during the build process. Most compilers use such information for compilation, then discard it. However, when the `-Mprof` or `-Minfo=ccff` options are used, the PGI compilers save this information in the object and executable files using the *Common Compiler Feedback Format*, or CCFF.

Feedback messages provide information about what the compiler did in optimizing the code, as well as describe obstacles to optimization. Most feedback messages have associated explanations or hints that explain what the message means in more detail. Further, these messages sometimes provide suggestions for improving the performance of the program.

●	The informational icon indicates that CCFF information is available.

In PGPROF you can access Compiler Feedback using the **Compiler Feedback tab** in the Focus Panel. Messages are categorized according to the type of information that they contain.

For more information on the Compiler Feedback tab, refer to "PGPROF Focus Panel," on page 185. For more information on the Common Compiler Feedback Format (CCFF), refer to the website:`www.pgroup.com/ccff/`

Special Feedback Messages

There are some Compiler Feedback messages that deserve some explanation, specifically intensity messages and messages for inlined routines.

Intensity Messages

Computational intensity has been defined as the number of arithmetic operations performed per memory transfer.[1]The key idea is this: a high compute intensity value means that the time spent on data transfer is low compared to the time spent on arithmetic; a low compute intensity value suggests that memory traffic involving data transfer may dominate the overall time used by the computer.

The PGI Compiler emphasizes floating point operations, if they are present, to calculate the compute intensity ratio within a particular loop. If floating point operations are not present, the PGI compiler uses integer operations.

In some cases it is necessary to build programs using profile-guided optimization by building with `–Mpfi` or`–Mpfo`, as described in the section *Profile-Feedback Optimization using -Mpfi/Mpfo* in the Optimizing and Parallelizing chapter of the PGI User's Guide. Profile-guided optimization can often determine loop counts and other information needed to calculate the Compute Intensity for a given statement or loop.

Messages for Inlined Routines

Inlined functions are identified by CCFF messages. These Compiler Feedback messages for routines that have been inlined are associated with the source line where the routine is called. Further, these messages are prefixed with the routine and line number, and are indented to show the level of inlining. Currently there is not a way to view the source code of that inlined instance of the routine.

Profiling Parallel Programs

You can use PGPROF to analyze the performance of parallel programs, including multi-threaded and OpenMP programs, multi-process MPI programs, and programs that are a combination of the two. PGPROF also provides a Scalability Analysis feature that allows you to compare two profiling runs, and thus determine how well different parts of your program scale as the number of threads or processes changes.

[1]R.W. Hockney and C. R. Jesshope, Parallel Computers 2: Architecture, Programming and Algorithms 1988

Profiling Multi-threaded Programs

Multi-threaded programs that you can profile using PGPROF include OpenMP programs that are built with `–mp`, auto-parallelized programs that are built with `–Mconcur`, and programs that use native thread libraries such as pthreads.

Collecting Data from Multi-Threaded Programs

Some methods of performance data collection work better with multi-threaded programs than others. As always, the recommended approach is to use **pgcollect**, initially with time-based sampling, optionally followed by event-based sampling. Building with `–Minfo=ccff` is always a good idea when using **pgcollect**. Alternatively, building with one of the compiler options `–Mprof=lines` or `–Mprof=hwcts` creates a program that collects accurate multi-threaded performance profiles.

Alternatively, on Linux systems that support OProfile, described in“Event-based Sampling,” on page 130, you can build the program using `–Mprof=dwarf –Minfo=ccff`, and collect data using the **pgcollect** command.

The `–Mprof=func` option works with multi-threaded programs. However, routines that contain one or more parallel regions appear to be run on a single thread; because when the parallelism is not active, the data collection is at the entry and exit of the routine.

The `–Mprof=time` and `–pg` options generate programs that only collect data on a single thread.

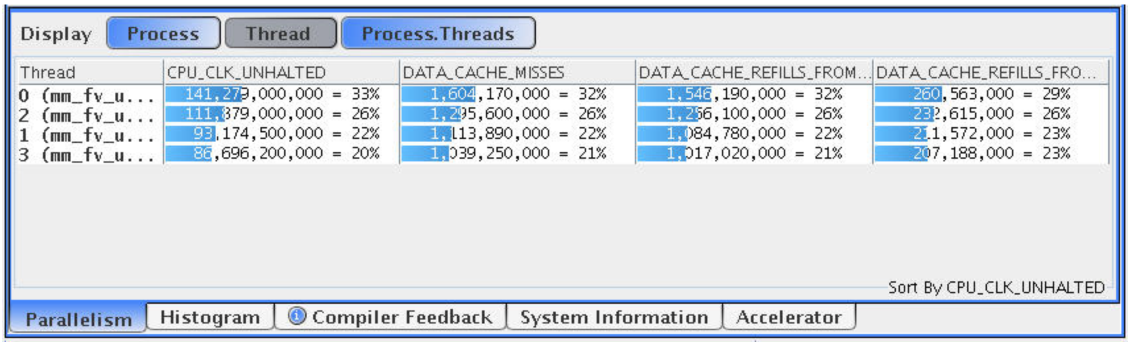
To collect data, run your program normally. Upon successful termination, a `pgprof.out` file is created.

Analyzing the Performance of Multi-Threaded Programs

The display of profile data for a multi-threaded program differs from that of a single-threaded program in a couple of ways, as illustrated in Figure 15.7, “Sort View ,” on page 139.

- In the Statistics Table, the data shown is the maximum value for any single thread in the process.
- The Parallelism tab shows the thread-specific performance data for the row selected in the Statistics Table, whether the Statistics Table is in the routine-level, line-level, or assembly-level view.

  You can use thread-specific data to determine how well-balanced the application is. Ideally, each thread would spend exactly the same amount of time on a given part of the program. If there are large disparities in the time spent by the various threads, this points to a load imbalance, where some threads are left idle while other threads are working. In this case, the resources of the system are not being used with 100% efficiency.



Thread	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FROM...	DATA_CACHE_REFILLS_FRO...
0 (mm_fv_u...	141,279,000,000 = 33%	1,604,170,000 = 32%	1,546,190,000 = 32%	260,563,000 = 29%
2 (mm_fv_u...	111,879,000,000 = 26%	1,295,600,000 = 26%	1,256,100,000 = 26%	232,615,000 = 26%
1 (mm_fv_u...	93,174,500,000 = 22%	1,113,890,000 = 22%	1,084,780,000 = 22%	211,572,000 = 23%
3 (mm_fv_u...	86,696,200,000 = 20%	1,039,250,000 = 21%	1,017,020,000 = 21%	207,188,000 = 23%

For example, in the program illustrated here, we can see that thread 0 spent 33% of the time in the routine, while thread 3 spent only 20% of the time there. Performance might improve if the work could be distributed more evenly.

Profiling MPI Programs

You can profile MPI programs using PGI compilers and tools by building with the `-Mprof` option, which allows you to specify the implementation of MPI you want to use - MPICH-1 (using the included version of MPICH-1), HP-MPI for Linux, MPICH-2, or MVAPICH. For the latest list of supported MPI implementations and MPI profiling options, refer to the PGI Release Notes.

For example, to build for time-based profiling with MPICH-1 profiling on Linux, use the following command:

```
$ pgf90 -fast -Mprof=time,mpich1 myprog.f90
```

PGPROF MPI profiling collects counts of the number of messages and bytes sent and received. You can then use this information to analyze a program's message passing behavior.

OpenMPI Profiling

PGI provides performance profiling of MPI message passing support for OpenMPI applications on Linux and Mac OS X. On Apple systems, no special configuration is necessary. On Linux systems you must configure the OpenMPI installation to work with the PGI profiling system.

Once your system has been configured for OpenMPI profiling, you can follow these steps to build your program, run it, and invoke the profile.

Note

For information on how to build and install PGI-built OpenMPI and how to configure OpenMPI for PGI profiling, refer to the PGI Workstation Installation Guide.

On Linux

1. **Build** your program using the OpenMPI compiler wrappers (mpicc, mpic++, mpif77, and/or mpif90) with one of the PGI profiling options.

   #### Note

   When you build with `-Mprof=time|lines|func|hwcts`, MPI profiling is included automatically.

2. **Run** your program as you normally would. One or more files named `pgprof.out` is created in your working directory.

3. **Invoke** the profiler to see the results of your profiling run.

   ```
   pgprof -exe your_program
   ```

On Apple OS X

Note

On Apple OS X, there is no need to install or configure OpenMPI. PGI Workstation for Apple includes a pre-configured version of OpenMPI.

1. **Build** your program using the OpenMPI compiler wrappers (mpicc, mpic++, mpif77, and/or mpif90) with one of the PGI profiling options.

   #### Note

   When you build with `-Mprof=lines|func`, MPI profiling is included automatically.

2. **Run** your program as you normally would. One or more files named `pgprof.out` is created in your working directory.
3. **Invoke** the profiler to see the results of your profiling run.

```
pgprof -exe your_program
```

Collecting Data from MPI Programs

PGI provides performance profiling of MPI message passing support for OpenMPI applications on Linux and Mac OS X. On Apple systems, no special configuration is necessary. On Linux systems you must configure the OpenMPI installation to work with the PGI profiling system.

To collect MPI performance data, you must build your program using one of the MPI suboptions to the option `-Mprof`. These suboptions include:

- `-Mprof=mpich1` (Linux only)
- `-Mprof=mpich2` (Linux only)
- `-Mprof=mvapich1` (Linux only)
- `-Mprof=hpmpi` (Linux only)
- `-Mprof=msmpi` (Windows only)

Important

The MPI profiling options cannot be used alone. They may only be used in concert with another suboption of `-Mprof`, such as lines, func, time, and hwcts.

For example, to do time-based sampling of an MPICH-2 Fortran program on Linux, you might build with the following command:

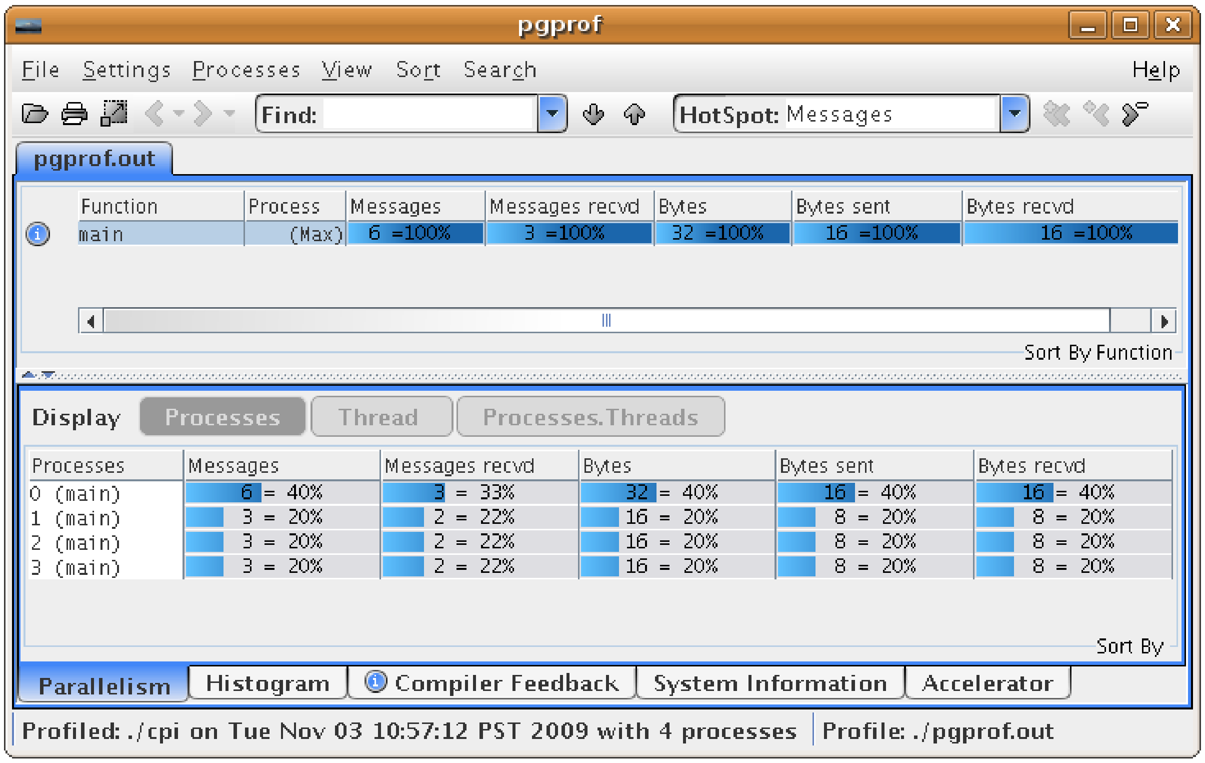
```
   pgf95 -fast -Mprof=time,mpich2 -o myprog myprog.f90
```

To collect MPI profile data, run your program normally. On successful program termination, one profile data file is created for each MPI process. The master profile data file is named `pgprof.out`. The other files have names similar to `pgprof.out`, but they are numbered.

Analyzing the Performance of MPI Programs

Figure 15.8 illustrates an MPI profile.

Figure 15.8. Sample MPI Profile



Function	Process	Messages	Messages recvd	Bytes	Bytes sent	Bytes recvd
main	(Max)	6 =100%	3 =100%	32 =100%	16 =100%	16 =100%

Processes	Messages	Messages recvd	Bytes	Bytes sent	Bytes recvd
0 (main)	6 = 40%	3 = 33%	32 = 40%	16 = 40%	16 = 40%
1 (main)	3 = 20%	2 = 22%	16 = 20%	8 = 20%	8 = 20%
2 (main)	3 = 20%	2 = 22%	16 = 20%	8 = 20%	8 = 20%
3 (main)	3 = 20%	2 = 22%	16 = 20%	8 = 20%	8 = 20%

Profiled: ./cpi on Tue Nov 03 10:57:12 PST 2009 with 4 processes | Profile: ./pgprof.out

For MPI profiles, the Statistics Table displays an extra column, titled *Process*. The values in this column denote whether the other values shown in the Statistics Table are the max, min, or avg values. You can use the Processes menu, described in "Processes Menu," on page 175, to customize the display of max, min, or avg values.

The Parallelism tab for MPI programs is used in the same way that it is used for multi-threaded programs, as described in "Analyzing the Performance of Multi-Threaded Programs," on page 141.

You can use the send and receive counts for messages, and the byte counts to identify potential communication bottlenecks, and use the process-specific data to find load imbalances.

Scalability Comparison

PGPROF provides a Scalability Comparison feature that measures changes in the program's performance between multiple executions of an application. Generally this information is used to measure the performance of the program when it is run 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.

Scalability is computed using the maximum time spent in each thread/process. Depending on how you profiled your program, this measurement may be displayed in the Statistics Table in a column with one of these heading titles:

Time	if you used `-Mprof=func`, `-Mprof=lines`, or `-Mprof=time`
CPU_CLK_UNHALTED	if you used **pgcollect**
TOT_CYC	if you used `-Mprof=hwcts`

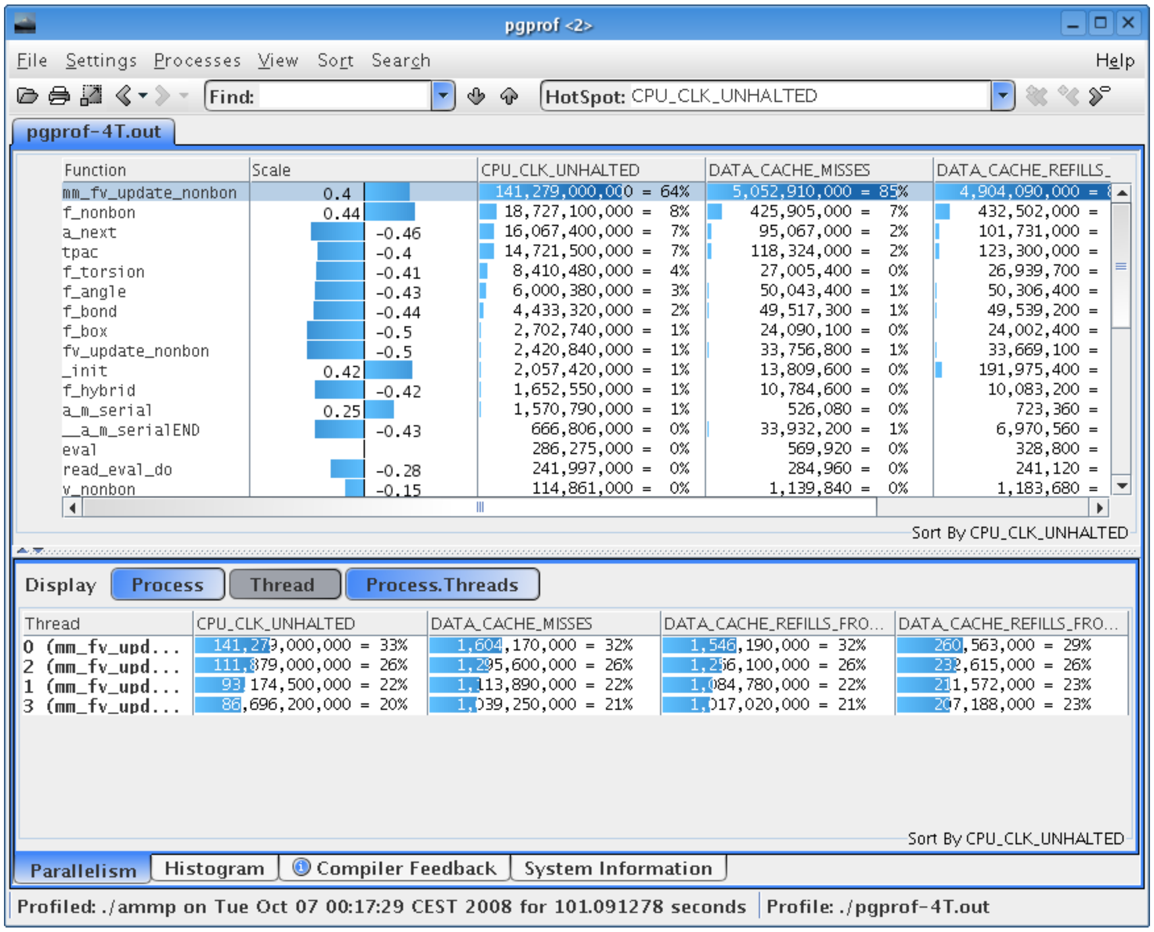
Important

Profiling multi-process MPI programs with the **pgcollect** command is not supported.

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, described in“File Menu,” on page 172, or click the Scalability comparison button in the Toolbar. Choose a second profile for comparison. A new instance of PGPROF appears, with a column named `Scale` in the Statistics Table.

Figure 15.9 shows the profile of a run that used four threads with Scalability Comparison to the same program run with a single thread.

Figure 15.9. Profile of an Application Run with 4 Threads



HotSpot: CPU_CLK_UNHALTED

Function	Scale	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_
mm_fv_update_nonbon	0.4	141,279,000,000 = 64%	5,052,910,000 = 85%	4,904,090,000 =
f_nonbon	0.44	18,727,100,000 = 8%	425,905,000 = 7%	432,502,000 =
a_next	-0.46	16,067,400,000 = 7%	95,067,000 = 2%	101,731,000 =
tpac	-0.4	14,721,500,000 = 7%	118,324,000 = 2%	123,300,000 =
f_torsion	-0.41	8,410,480,000 = 4%	27,005,400 = 0%	26,939,700 =
f_angle	-0.43	6,000,380,000 = 3%	50,043,400 = 1%	50,306,400 =
f_bond	-0.44	4,433,320,000 = 2%	49,517,300 = 1%	49,539,200 =
f_box	-0.5	2,702,740,000 = 1%	24,090,100 = 0%	24,002,400 =
fv_update_nonbon	-0.5	2,420,840,000 = 1%	33,756,800 = 1%	33,669,100 =
_init	0.42	2,057,420,000 = 1%	13,809,600 = 0%	191,975,400 =
f_hybrid	-0.42	1,652,550,000 = 1%	10,784,600 = 0%	10,083,200 =
a_m_serial	0.25	1,570,790,000 = 1%	526,080 = 0%	723,360 =
__a_m_serialEND	-0.43	666,806,000 = 0%	33,932,200 = 1%	6,970,560 =
eval		286,275,000 = 0%	569,920 = 0%	328,800 =
read_eval_do	-0.28	241,997,000 = 0%	284,960 = 0%	241,120 =
v_nonbon	-0.15	114,861,000 = 0%	1,139,840 = 0%	1,183,680 =

Sort By CPU_CLK_UNHALTED

Thread	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FRO...	DATA_CACHE_REFILLS_FRO...
0 (mm_fv_upd...	141,279,000,000 = 33%	1,604,170,000 = 32%	1,546,190,000 = 32%	260,563,000 = 29%
2 (mm_fv_upd...	111,879,000,000 = 26%	1,295,600,000 = 26%	1,256,100,000 = 26%	232,615,000 = 26%
1 (mm_fv_upd...	93,174,500,000 = 22%	1,113,890,000 = 22%	1,084,780,000 = 22%	211,572,000 = 23%
3 (mm_fv_upd...	86,696,200,000 = 20%	1,039,250,000 = 21%	1,017,020,000 = 21%	207,188,000 = 23%

Sort By CPU_CLK_UNHALTED

Profiled: ./ammp on Tue Oct 07 00:17:29 CEST 2008 for 101.091278 seconds | Profile: ./pgprof-4T.out

Each profile entry that has timing information has a Scale value. The scale value measures how well these parts of the program scaled, or improved their performance as a result of parallelism.

- A scale value of zero indicates no change in the execution time between the two runs.
- A scale value of one means that part of the program achieved perfect scalability. For example, if a routine had a Time value of 100 seconds with one thread, and 25 seconds with four threads, it would have a Scale value of one.
- A negative value is the relative slowdown without taking the number of threads or processes into account. If a routine takes 20% more time to execute using four threads than it took using one thread, the Scale value is -0.2.
- A question mark ('?') in the Scale column indicates that PGPROF is unable to perform the scalability comparison for this profile entry. For example, scalability comparison may not be possible if the two profiles do not share the same executable or input data.

Profiling Resource Utilization with Hardware Event Counters

Important

Profiling with hardware counters is available only on Linux.

Modern x86 and x64 processors provide low-level hardware counters that can be used to track the resource utilization of a program. Tracking this information can be useful in tuning program performance because it allows you to go beyond just knowing where the program is spending the most time and examine why it is spending time there.

Linux systems do not provide hardware counter support by default. These systems must either have the OProfile package installed or have a kernel patch that enables PAPI, as described in “Profiling with Hardware Event Counters using PAPI,” on page 147

Profiling with Hardware Event Counters (Linux Only)

PGPROF supports two methods of hardware counter data collection:

- Execution of the program under the control of the **pgcollect** command.

  Collection of profile data using **pgcollect** may be done on any linux86 or linux86_64 system where Oprofile is installed. OProfile is included as an install-time option with most Linux distributions; it may also be downloaded from *http://oprofile.sourceforge.net/*.

- Building the program with the `-Mprof=hwcts` compiler option and executing it independently.

  Profiling by compiling with the `-Mprof=hwcts` option is only available on linux86_64 systems where PAPI has been installed. PAPI is available for download from *http://icl.cs.utk.edu/papi/*.

Profiling with Hardware Event Counters using pgcollect

You can use PGPROF to display time-based and hardware event-based profiles generated via the OProfile package, which is available on most current Linux distributions.

No special build options are required to enable event-based profiling with **pgcollect**, although building with the option `–Minfo=ccff` may provide useful compiler feedback.

For specific information on using PGPROF with hardware event counters, refer to Chapter 22, "*pgcollect Reference*," on page 197.

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 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 option uses the PAPI_TOT_CYC counter to profile total CPU cycles executed by the application. PGPROF converts 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 is this:

```
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.

Determine Available Hardware Counters

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);
}
```

This program uses the PAPI and PGI libraries to dump the available hardware counters to standard output.

Display Available Events

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.

The following example, which uses tcsh or csh shell, specifies four events with the PGPROF_EVENTS environment variable:

```
% setenv PGPROF_EVENTS PAPI_TOT_CYC.1593262939:PAPI_FP_OPS:PAPI_L1_DCM:PAPI_L2_ICM
```

To specify the same four events using the sh or bash shell, use this command:

```
$ 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, by default, the profiling mechanism counts CPU cycles (PAPI_TOT_CYC event).

The following example shows a partial output from pgevents:

Example 15.1. 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)
```

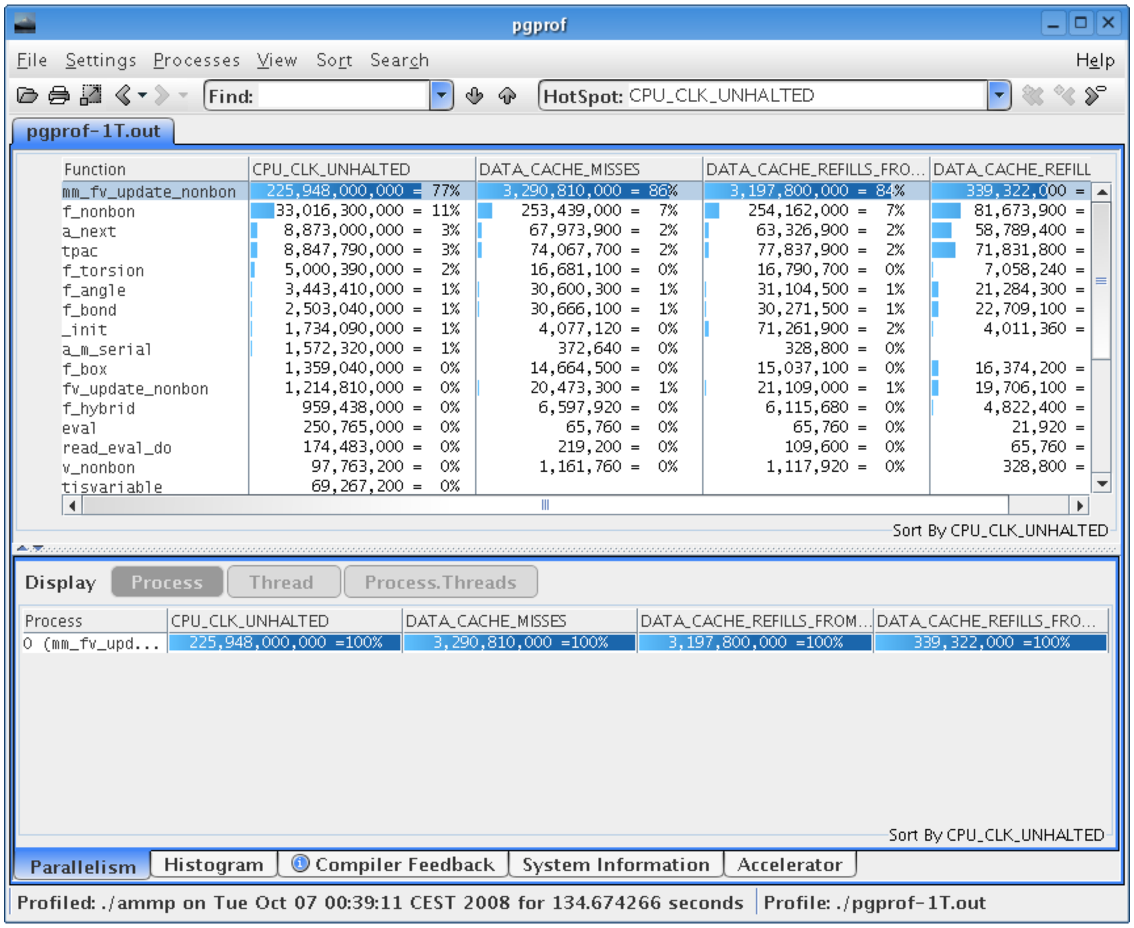
Analyzing Event Counter Profiles

If you compiled your program with the `–Mprof=hwcts` option or executed your program under the control of **pgcollect**, then you can profile up to four event counters and view them in PGPROF.

Figure 15.10 shows a profile of four event counters: CPU_CLK_UNHALTED, DATA_CACHE_MISSES, DATA_CACHE_REFILLS _FROM_L2, DATA_CACHE_REFILLS _FROM_SYSTEM.

In this example, the routine using the most time is also getting many cache misses. Investigating the memory access behavior in that routine, and looking at the Compiler Feedback, may offer some clues for improving its performance.

Figure 15.10. Profile with Hardware Event Counter


pgprof
File Settings Processes View Sort Search Help
Find:
HotSpot: CPU_CLK_UNHALTED
pgprof-1T.out

Function	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FRO...	DATA_CACHE_REFILL
mm_fv_update_nonbon	225,948,000,000 = 77%	3,290,810,000 = 86%	3,197,800,000 = 84%	339,322,000 =
f_nonbon	33,016,300,000 = 11%	253,439,000 = 7%	254,162,000 = 7%	81,673,900 =
a_next	8,873,000,000 = 3%	67,973,900 = 2%	63,326,900 = 2%	58,789,400 =
tpac	8,847,790,000 = 3%	74,067,700 = 2%	77,837,900 = 2%	71,831,800 =
f_torsion	5,000,390,000 = 2%	16,681,100 = 0%	16,790,700 = 0%	7,058,240 =
f_angle	3,443,410,000 = 1%	30,600,300 = 1%	31,104,500 = 1%	21,284,300 =
f_bond	2,503,040,000 = 1%	30,666,100 = 1%	30,271,500 = 1%	22,709,100 =
_init	1,734,090,000 = 1%	4,077,120 = 0%	71,261,900 = 2%	4,011,360 =
a_m_serial	1,572,320,000 = 1%	372,640 = 0%	328,800 = 0%	
f_box	1,359,040,000 = 0%	14,664,500 = 0%	15,037,100 = 0%	16,374,200 =
fv_update_nonbon	1,214,810,000 = 0%	20,473,300 = 1%	21,109,000 = 1%	19,706,100 =
f_hybrid	959,438,000 = 0%	6,597,920 = 0%	6,115,680 = 0%	4,822,400 =
eval	250,765,000 = 0%	65,760 = 0%	65,760 = 0%	21,920 =
read_eval_do	174,483,000 = 0%	219,200 = 0%	109,600 = 0%	65,760 =
v_nonbon	97,763,200 = 0%	1,161,760 = 0%	1,117,920 = 0%	328,800 =
tisvariable	69,267,200 = 0%			

Sort By CPU_CLK_UNHALTED

Process	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FROM...	DATA_CACHE_REFILLS_FRO...
0 (mm_fv_upd...	225,948,000,000 =100%	3,290,810,000 =100%	3,197,800,000 =100%	339,322,000 =100%

Sort By CPU_CLK_UNHALTED

Parallelism Histogram Compiler Feedback System Information Accelerator

Profiled: ./ammp on Tue Oct 07 00:39:11 CEST 2008 for 134.674266 seconds | Profile: ./pgprof-1T.out

Profiling Accelerator Programs

You can use PGPROF to analyze the performance of accelerator programs. Accelerator performance data is included in the profile, `pgprof.out`, when an accelerator program is run using **pgcollect**. The profiling procedure is the same as for host-only programs, except that PGPROF provides an Accelerator tab that allows you to review profiling information provided by the accelerator. You do not need to build with any special options to collect accelerator performance data.

Here is an example of the commands you might use in a simple accelerator profiling session:

```
$ pgfortran -ta=nvidia -o myprog myprog.f90
$ pgcollect -time ./myprog
$ pgprof -exe ./myprog
```

Note

You can build your program to print accelerator performance data to the standard output by using the `time` suboption to the target accelerator option `-ta'`. For example, you can use this command:

```
$  pgfortran -ta=nvidia,time myprog.f90
```

The `time` suboption has no effect on `pgcollect` or PGPROF profiling.

For more information on using PGI compilers to build programs for accelerators and on related terminology, refer to Chapter 7, *"Using an Accelerator,"* in the *PGI User's Guide*.

For more information on **pgcollect**, refer to Chapter 22, *"pgcollect Reference"*.

Analyzing Accelerator Performance Data

This section provides a description of how to examine accelerator performance data using PGPROF.

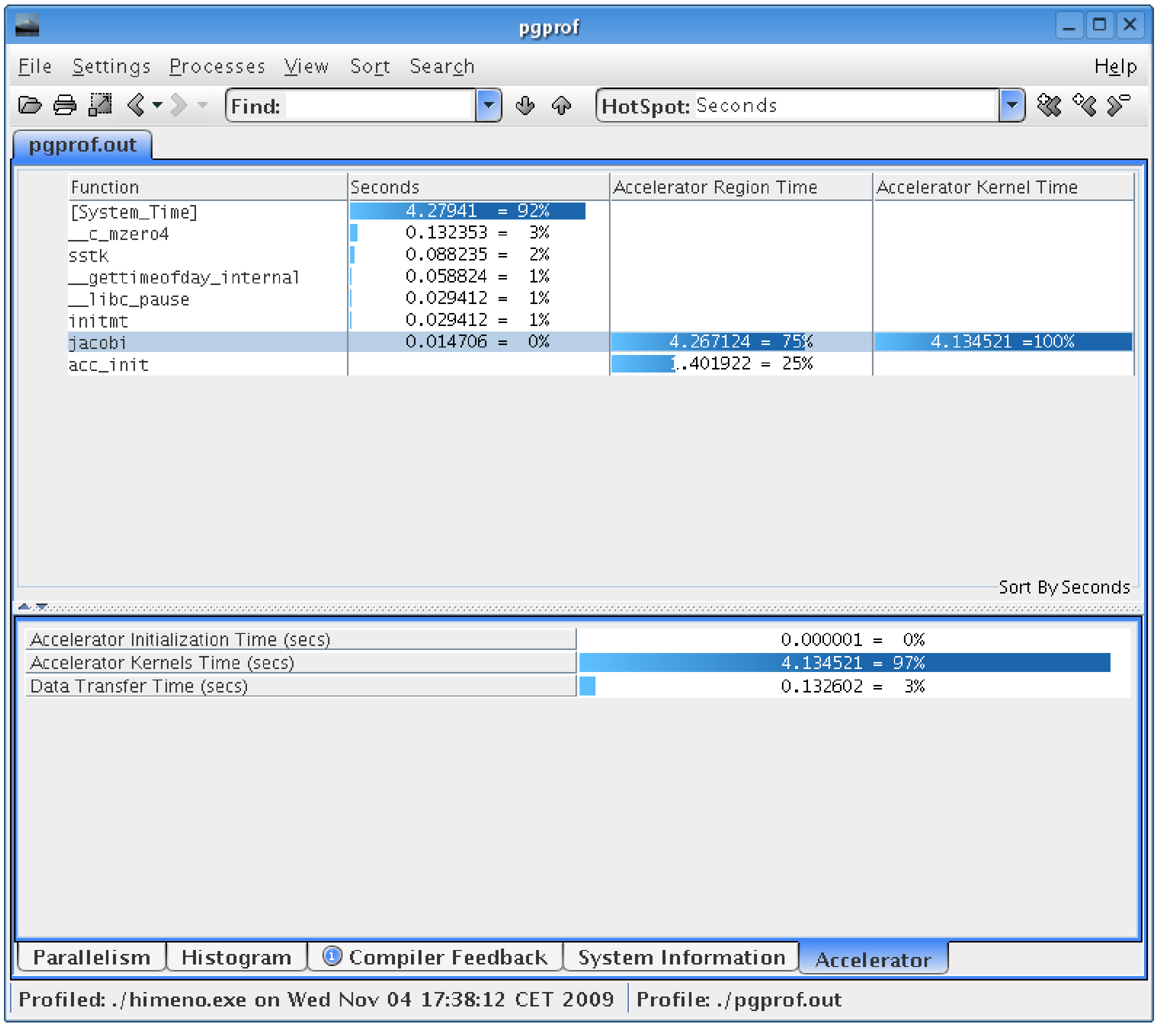
Note

A comprehensive guide to tuning accelerator programs is beyond the scope of this manual.

Function-Level Analysis

When you invoke PGPROF on the profile of an accelerator program, the initial view displays a function list showing host times in the Seconds column and accelerator times in the Accelerator Region Time column and Accelerator Kernel Time column. Figure 15.11 illustrates a routine-level view with the routine `jacobi` selected and the Accelerator tab chosen in the Focus Panel.

Figure 15.11. Accelerator Performance Data for Routine-Level Profiling Example


pgprof
File Settings Processes View Sort Search Help
Find:
HotSpot: Seconds
pgprof.out
Sort By Seconds
Parallelism Histogram Compiler Feedback System Information Accelerator

Function	Seconds	Accelerator Region Time	Accelerator Kernel Time
[System_Time]	4.27941 = 92%		
__c_mzero4	0.132353 = 3%		
sstk	0.088235 = 2%		
__gettimeofday_internal	0.058824 = 1%		
__libc_pause	0.029412 = 1%		
initmt	0.029412 = 1%		
jacobi	0.014706 = 0%	4.267124 = 75%	4.134521 =100%
acc_init		1.401922 = 25%	

Accelerator Initialization Time (secs)	0.000001 = 0%
Accelerator Kernels Time (secs)	4.134521 = 97%
Data Transfer Time (secs)	0.132602 = 3%

Profiled: ./himeno.exe on Wed Nov 04 17:38:12 CET 2009 | Profile: ./pgprof.out

One of the first things to look at in tuning an accelerator program is whether the Data Transfer Time is large relative to the Accelerator Kernels Time. In the example illustrated in Figure 15.11, the Accelerator Kernels Time of 4.134521 seconds is much larger than the Data Transfer Time of 0.132602 seconds, so we have efficient use of the accelerator.

If data transfer time is taking a significant portion of the total time, you would want to investigate if transfer time could be reduced using *data regions*, described in Chapter 7, *Using an Accelerator*, of the PGI User's Guide.

If data transfer time is relatively high and you have already considered data regions, you might want to examine the Compiler Feedback. You must compile with`-Minfo=ccff` to be able to do this. Check if the compiler is generating `copyin/copyout` operations that use slices of your arrays,. If so, you may want to override the compiler to `copyin/copyout` the entire array.

For more information on compiler feedback, refer to "Compiler Feedback," on page 139.

Region-Level Analysis

As with host-only profiles, you can drill down to the source code level by double-clicking on the routine name in the Function column. For an accelerator program, the display centers on the accelerator region directive for the longest-executing region. The Accelerator tab shows a breakdown of timing statistics for the region and the accelerator kernels it contains.

Note

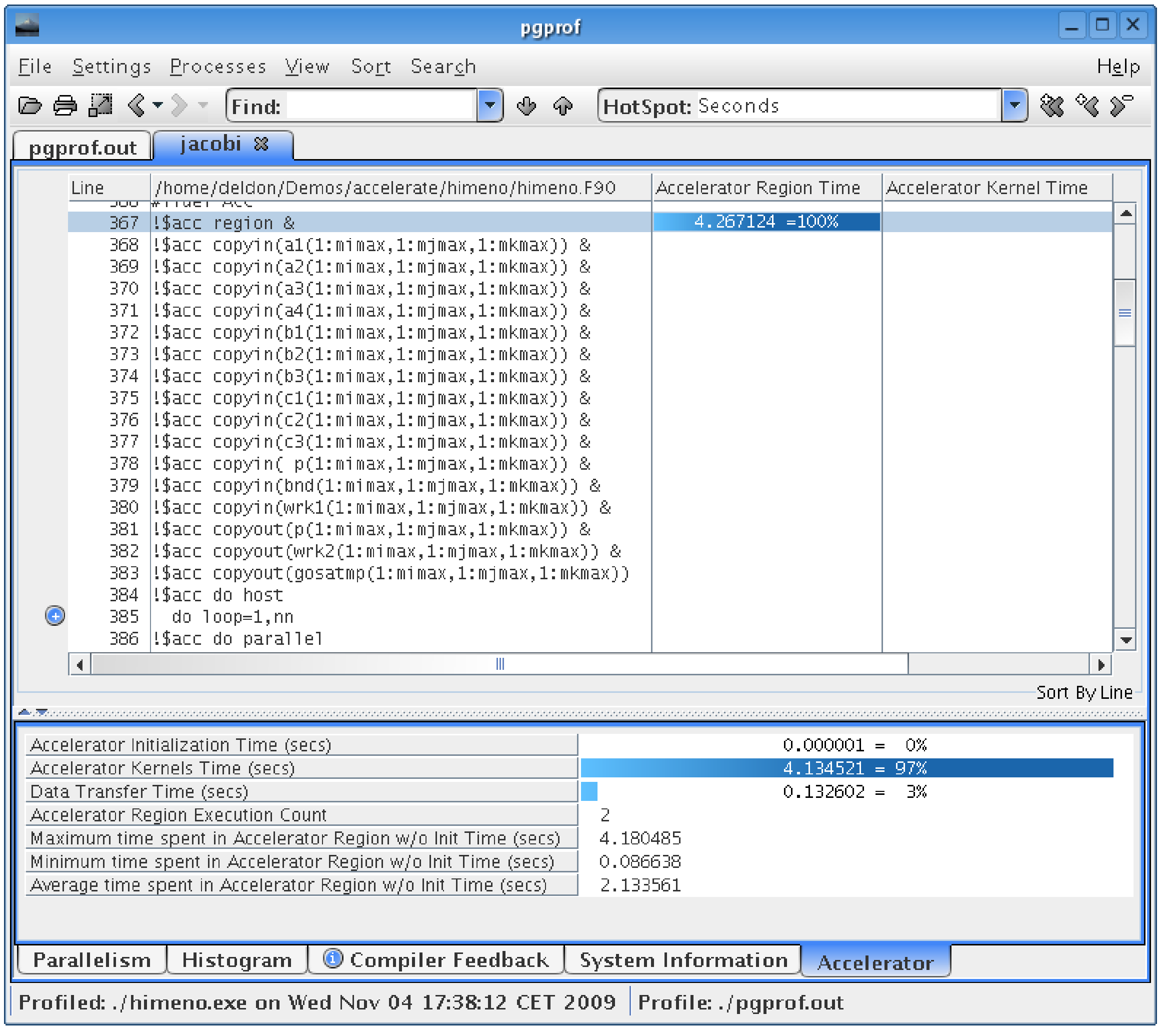
A routine can contain more than one accelerator region.

Figure 15.12 shows an example of a source-level view with an accelerator region directive selected.

Note

In this illustration, if you want to see the Seconds column, you could scroll to the right in the Statistics Table.

Figure 15.12. Source-Level Profiling for an Accelerator Region



Line	/home/deldon/Demos/accelerate/himeno/himeno.F90	Accelerator Region Time	Accelerator Kernel Time
367	!$acc region &	4.267124 =100%	
368	!$acc copyin(a1(1:mimax,1:mjmax,1:mkmax)) &		
369	!$acc copyin(a2(1:mimax,1:mjmax,1:mkmax)) &		
370	!$acc copyin(a3(1:mimax,1:mjmax,1:mkmax)) &		
371	!$acc copyin(a4(1:mimax,1:mjmax,1:mkmax)) &		
372	!$acc copyin(b1(1:mimax,1:mjmax,1:mkmax)) &		
373	!$acc copyin(b2(1:mimax,1:mjmax,1:mkmax)) &		
374	!$acc copyin(b3(1:mimax,1:mjmax,1:mkmax)) &		
375	!$acc copyin(c1(1:mimax,1:mjmax,1:mkmax)) &		
376	!$acc copyin(c2(1:mimax,1:mjmax,1:mkmax)) &		
377	!$acc copyin(c3(1:mimax,1:mjmax,1:mkmax)) &		
378	!$acc copyin(p(1:mimax,1:mjmax,1:mkmax)) &		
379	!$acc copyin(bnd(1:mimax,1:mjmax,1:mkmax)) &		
380	!$acc copyin(wrk1(1:mimax,1:mjmax,1:mkmax)) &		
381	!$acc copyout(p(1:mimax,1:mjmax,1:mkmax)) &		
382	!$acc copyout(wrk2(1:mimax,1:mjmax,1:mkmax)) &		
383	!$acc copyout(gosatmp(1:mimax,1:mjmax,1:mkmax))		
384	!$acc do host		
385	do loop=1,nn		
386	!$acc do parallel		

Sort By Line

Accelerator Initialization Time (secs)	0.000001 = 0%
Accelerator Kernels Time (secs)	4.134521 = 97%
Data Transfer Time (secs)	0.132602 = 3%
Accelerator Region Execution Count	2
Maximum time spent in Accelerator Region w/o Init Time (secs)	4.180485
Minimum time spent in Accelerator Region w/o Init Time (secs)	0.086638
Average time spent in Accelerator Region w/o Init Time (secs)	2.133561

Profiled: ./himeno.exe on Wed Nov 04 17:38:12 CET 2009 | Profile: ./pgprof.out

Kernel-Level Analysis

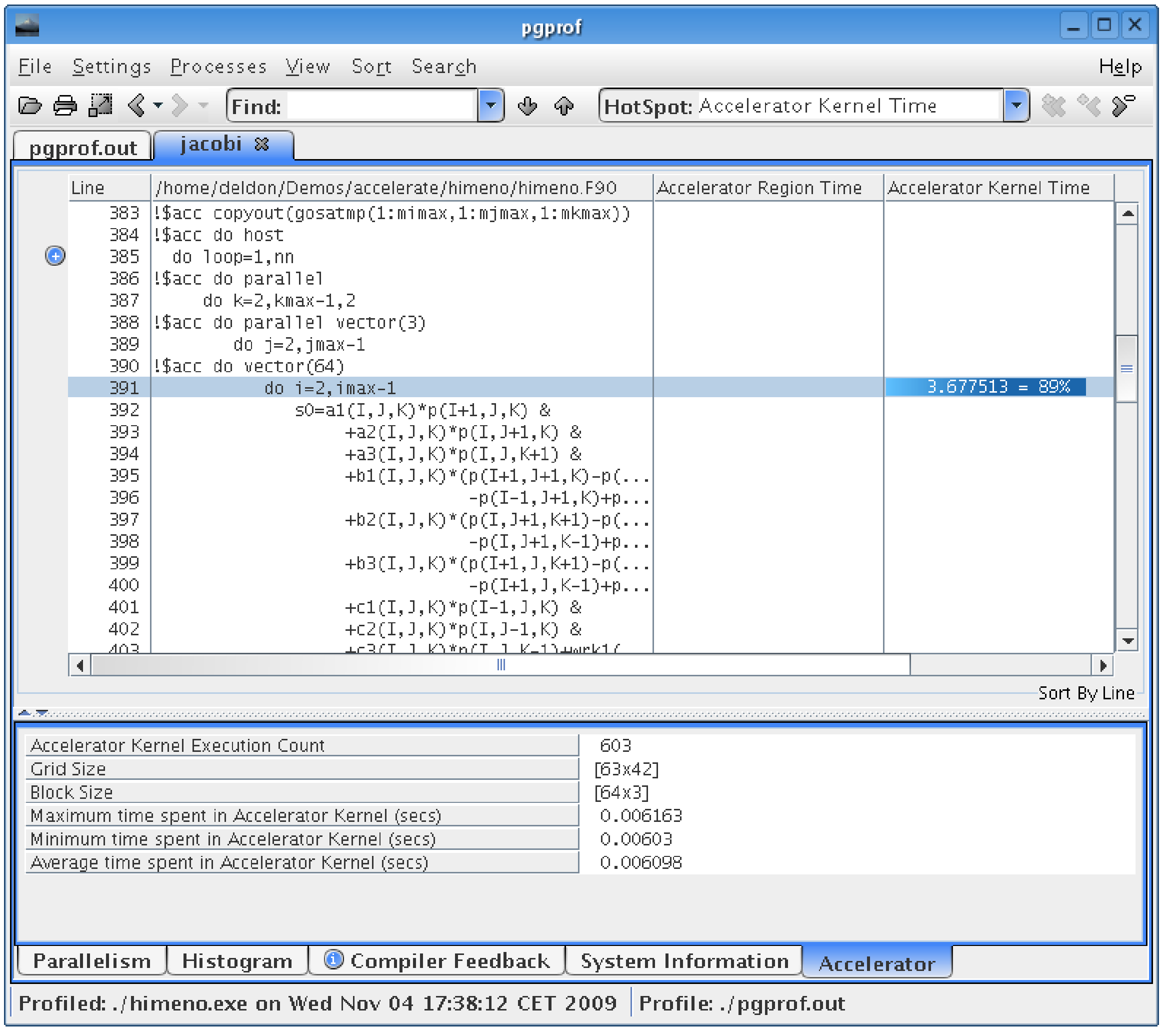
Since an accelerator region can contain multiple distinct kernels, you may want to examine performance data for an individual kernel. You do this by selecting the first source line of the kernel.

In the source-level view, the first line of a kernel has data listed in the Accelerator Kernel Time column.

To navigate to the longest-executing kernel:

1. Select Accelerator Kernel Time in the HotSpot selector in the upper-right portion of the user interface.
2. Click the double left arrow (<<) located next to the HotSpot selector.

Figure 15.13. Source-Level Profiling for an Accelerator Kernel



Line	/home/deldon/Demos/accelerate/himeno/himeno.F90	Accelerator Region Time	Accelerator Kernel Time
383	`!$acc copyout(gosatmp(1:mimax,1:mjmax,1:mkmax))`		
384	`!$acc do host`		
385	`  do loop=1,nn`		
386	`!$acc do parallel`		
387	`     do k=2,kmax-1,2`		
388	`!$acc do parallel vector(3)`		
389	`        do j=2,jmax-1`		
390	`!$acc do vector(64)`		
391	`           do i=2,imax-1`		3.677513 = 89%
392	`              s0=a1(I,J,K)*p(I+1,J,K) &`		
393	`                   +a2(I,J,K)*p(I,J+1,K) &`		
394	`                   +a3(I,J,K)*p(I,J,K+1) &`		
395	`                   +b1(I,J,K)*(p(I+1,J+1,K)-p(...`		
396	`                               -p(I-1,J+1,K)+p...`		
397	`                   +b2(I,J,K)*(p(I,J+1,K+1)-p(...`		
398	`                               -p(I,J+1,K-1)+p...`		
399	`                   +b3(I,J,K)*(p(I+1,J,K+1)-p(...`		
400	`                               -p(I+1,J,K-1)+p...`		
401	`                   +c1(I,J,K)*p(I-1,J,K) &`		
402	`                   +c2(I,J,K)*p(I,J-1,K) &`		

Accelerator Kernel Execution Count	603
Grid Size	[63x42]
Block Size	[64x3]
Maximum time spent in Accelerator Kernel (secs)	0.006163
Minimum time spent in Accelerator Kernel (secs)	0.00603
Average time spent in Accelerator Kernel (secs)	0.006098

Profiled: ./himeno.exe on Wed Nov 04 17:38:12 CET 2009 | Profile: ./pgprof.out

In this illustration the selected line in the main Statistics Table has a value only in the Accelerator Kernel Time. The Accelerator tab displays all the details for the Accelerator Kernel performance data.

Note

To see the Seconds column, you would scroll to the right in the Statistics Table.

For more information on tuning accelerator programs, refer to Chapter 7, *Using an Accelerator*, of the PGI User's Guide.

Chapter 16. Compiler Options for Profiling

This chapter describes the PGI compiler options that are used to control profiling and how they are interpreted.

-Mprof Syntax

You can use the following compiler options to control data collection. Most of these options are related to `-Mprof`, for which the syntax is:

```
-Mprof[=option[,option,...]]
```

You use `-Mprof` to set performance profiling options. Use of these options causes the resulting executable to create a performance profile that can be viewed and analyzed with the PGPROF performance profiler.

Profiling Compilation Options

In the descriptions that follow, PGI-style profiling implies compiler-generated source instrumentation. MPICH-style profiling implies the use of instrumented wrappers for MPI library routines.

–Minfo=ccff
: Generate compiler feedback information and store it in object and executable files for later access by performance tools. Use `–Minfo=ccff` when collecting performance data using **pgcollect**. All `–Mprof` options except `–Mprof=dwarf` imply `–Minfo=ccff`.

–Mprof=dwarf
: Generate a subset of DWARF symbol information adequate for viewing source line information with most performance profilers.

: In the PGI compilers `–Mprof=dwarf` is on by default. You can use the `–Mnodwarf` option to disable it. Source-level information is not available if you profile a program built with`–Mnodwarf`.

–Mprof=func
: Perform routine-level instrumentation-based profiling.

–Mprof=hpmpi
: [Linux only] Use the profiled HPMPI communication library. Implies `–Mmpi=hpmpi`.

–Mprof=hwcts

[linux86_64 Only - PAPI must be installed] Generate a profile using event-based sampling of hardware counters via the PAPI interface. Compiling and linking with this option produces an executable that generates a `pgprof.out` file which contains routine, line, and assembly-level profiling data. See "Profiling with Hardware Event Counters using PAPI," on page 147 for more information on profiling with hardware counters.

–Mprof=lines

Perform PGI-style line-level profiling.

–Mprof=mpich1

Perform MPICH-style profiling for MPICH-1. Implies `–Mmpi=mpich1`.

–Mprof=mpich2

Perform MPICH-style profiling for MPICH-2. Implies `–Mmpi=mpich2`.

–Mprof=msmpi

[Microsoft HPC Server only] Perform MPICH-style profiling for Microsoft MSMPI. Implies option `–Mmpi=msmpi`.

For `–Mprof=msmpi` to work, the `CCP_SDK` environment variable must be set. This variable is typically set when the Microsoft HPC Server SDK is installed.

–Mprof=mvapich1

[Linux only] Perform MPICH-style profiling for MVAPICH-1. Implies `–Mmpi=mvapich1`.

–Mprof=time

[Linux only] Generate a profile using time-based assembly-level statistical sampling. This is equivalent to `-pg` except the profile is saved in a file named `pgprof.out` rather than in `gmon.out`.

–pg

[Linux Only] Enable gprof-style (sample-based) profiling. Running an executable compiled with this option produces a `gmon.out` profile file which contains routine, line, and assembly-level profiling data.

Configuration Files for OpenMPI Profiling

For OpenMPI profiling on Linux, the OpenMPI installation must be properly built and configured to work with the PGI profiling system. The Installation Guide contains complete instructions on how to build the OpenMPI software distribution with PGI compilers and install it. It also includes the information required to modify some configuration files, which we refer to as compiler wrapper data files. This section provides information about these wrappers.

Once these compiler wrapper data files are modified, your system is ready for OpenMPI profiling, as described in "OpenMPI Profiling," on page 142.

Compiler Wrapper Data Files

The compiler wrapper data files are located in the `/share/openmpi` directory of your OpenMPI installation. Sample compiler wrapper data files located in your PGI '`/etc`' directory are available for you to direct modifications of the wrapper data files generated when you built OpenMPI.

The wrapper file names are:

```
mpicc-wrapper-data.txt
mpic++-wrapper-data.txt
mpif77-wrapper-data.txt
mpif90-wrapper-data.txt
```

A sample wrapper file includes a block of data similar to the following:

Note

The lines in bold are ones that must be modified to configure your OpenMPI installation for PGI profiling.

```
compiler_args=
project=Open MPI
project_short=OMPI
version=1.2.8
language=C
compiler_env=CC
compiler_flags_env=CFLAGS
compiler=pgcc
extra_includes=
preprocessor_flags=-D_REENTRANT
compiler_flags=
linker_flags=
libs=-lmpi -lopen-rte -lopen-pal
-lrt -ldl -Wl,--export-dynamic -lnsl -lutil -lpthread -ldl
required_file=
includedir=${includedir}
libdir=${libdir}
```

Configure OpenMPI for PGI Profiling

To configure OpenMPI for PGI profiling, you edit the compiler wrapper data files.

Note

The lines that you modify are in bold in the sample wrapper data file in the previous section.

Important

Before you begin, make backup copies of your original wrapper data files.

Make these modifications:

1. Add the line `compiler_args=` before any other configuration lines.
2. Copy the entire data block in the sample file *twice*.

   You need a data block for each of these compiler options:

   - `–Mprof=func|lines`
   - `–Mprof=time|hwcts` (hwcts is linux86-64 only)
3. In the second data block, modify the `compiler_args=` and the `compiler_flags=` lines. The PGI profiling options are shown just to the right of the equal sign. The compiler flags you select immediately follow the equal sign, with a space between each flag.

Your lines should look similar to these:

```
          compiler_args=-Mprof=func;-Mprof=lines
          ...
          compiler_flags=
         ...
```

4. In the third data block, modify the `compiler_args=` and the `compiler_flags=` lines. The PGI profiling options are shown just to the right of the equal sign. The compiler flags in this data block should include: -W0, -profile, lines at the beginning of the list of flags you select.

   Your lines should look similar to these:

```
          compiler_args=-Mprof=time;-Mprof=hwcts
          ...
          compiler_flags=-W0,-profile,lines
          ...
```

5. In both the second and third data blocks, modify the `libs=` line so that `-lpgnod_prof_openmpi` comes just before `-lmpi`.

   Note

   Do not modify any other lib values.

   The new `libs=` line looks similar to this:

```
libs=-lpgnod_prof_openmpi -lmpi -lopen-rte  -lopen-pal -lrt
 -ldl -Wl,--export-dynamic -lnsl -lutil -lpthread -ldl
```

Modified Compiler Wrapper Data File Sample

When you complete your modifications, your new wrapper data file has three data blocks that look similar to these. The lines you modified are in bold.

```
compiler_args=
project=Open MPI
project_short=OMPI
version=1.2.8
language=C
compiler_env=CC
compiler_flags_env=CFLAGS
compiler=pgcc
extra_includes=
preprocessor_flags=-D_REENTRANT
compiler_flags=
linker_flags=
libs=-lmpi -lopen-rte -lopen-pal -lrt -ldl -Wl,--export-dynamic -lnsl -lutil
 -lpthread -ldl
required_file=
includedir=${includedir}
libdir=${libdir}

compiler_args=-Mprof=func;-Mprof=lines
project=Open MPI
project_short=OMPI
version=1.2.8
language=C
```

```
compiler_env=CC
compiler_flags_env=CFLAGS
compiler=pgcc
extra_includes=
preprocessor_flags=-D_REENTRANT
compiler_flags=
linker_flags=
libs=-lpgnod_prof_openmpi
-lmpi -lopen-rte -lopen-pal -lrt  -ldl -Wl,--export-dynamic -lnsl
-lutil -lpthread -ldl
required_file=
includedir=${includedir}
libdir=${libdir}

compiler_args=-Mprof=time;-Mprof=hwcts
project=Open MPI
project_short=OMPI
version=1.2.8
language=C
compiler_env=CC
compiler_flags_env=CFLAGS
compiler=pgcc
extra_includes=
preprocessor_flags=-D_REENTRANT
compiler_flags=-W0,-profile,lines
linker_flags=
libs=-lpgnod_prof_openmpi
-lmpi -lopen-rte -lopen-pal -lrt  -ldl -Wl,--export-dynamic -lnsl
-lutil -lpthread -ldl
required_file=
includedir=${includedir}
libdir=${libdir}
```

Chapter 17. PGPROF Command Line Options

This chapter describes the PGPROF command-line options and how they are interpreted.As we stated in Chapter 14, "*Getting Started with the PGPROF Profiler*", PGPROF can interpret command-line options when present on the command line.

Command Line Option Descriptions

The following list describes the options and how PGPROF interprets them.

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.

  The default datafile name is `pgprof.out`. If no datafile argument is used, PGPROF attempts to use `pgprof.out` in the current directory.

–exe <filename>
: Set the executable to filename. The default filename is `a.out`.

–feedbackonly (Linux only)
: Only browse source code and Compiler Feedback information. Do not load any performance data from profile runs.

–help
: Prints a list of available command-line arguments.

–I <srcpath>
: Specify the source file search path. The PGPROF profiler always looks for a program source file in the current directory first. If it does not find the source file in the current directory, it consults the search path specified in `srcpath`. The `srcpath` argument is a string containing one or more directories separated by a path separator. The path separator is platform dependent: on Linux and MAC OS X, it is a colon (:), and on Windows it is a semicolon (;). Directories in the path are then 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 and MAC OS X.

```
-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
```

For more information, see the New Profiling Session… item in the description of the "File Menu," on page 172.

–jarg, arg1[, arg2,..., argn]

Pass specified arguments, separated by commas, to java. For example, the following option passes the argument `-Xmx256m` to java.

```
-jarg, -Xmx256m
```

This option is provided for troubleshooting purposes and is expected to rarely be used. If you do use this option, be certain not to forget the comma between the option and the first argument.

–scale "file(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 compares 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," on page 172.

–text

Use the PGPROF Command-Line Interface (CLI).

–V

Print version information.

Profiler Invocation and Startup

Let's take a look at some common ways to invoke the profiler, describing what each launch command means.

% pgprof

- If a `pgprof.out` file exists in the current directory, PGPROF tries to open it.
  - If an executable name can be determined from the `pgprof.out` file, the GUI is populated according to profile data, if valid.
  - If an executable name can NOT be determined from the `pgprof.out` file, then a dialog is opened on top of the main window with following message:

```
Can't determine executable for file 'pgprof.out'Please use 'File |
New Profiling Session...' menu to specify one
```

- If no `pgprof.out` file exists in the current directory, the GUI is not populated and no dialog appears.

% pgprof -exe <execname>

- If a `pgprof.out` file exists in the current directory, PGPROF tries to open it and use <execname>. Further, the GUI is populated according to profile data, if valid.
- If no `pgprof.out` file exists in the current directory, the GUI is not populated and no dialog appears. Further, when the user selects the menu `File | New Profiling Session...`, then the Text Field for `Executable` is set with <*execname*> in the dialog.

% pgprof -exe <execname> <profilename>

PGPROF tries to open the profile <*profilename*> using <*execname*> for the executable name. Further, the GUI is populated according to profile data, if valid.

% pgprof -feedbackonly

- If a `a.out` file exists in the current directory, PGPROF tries to open it.Further, if `a.out` is an executable with valid DWARF/ELF/CCFF info, then PGPROF populates the GUI. You are then in *feedback-only mode*.
- If no `a.out` file exists in the current directory, the GUI is not populated and no dialog appears. Further, when the user selects the menu `File | New Profiling Session...`, then the `Feedback only` checkbox is selected in the dialog.

% pgprof -exe <execname> -feedbackonly

PGPROF tries to open the executable <*execname*>. Further, if the executable <*execname*> is valid with DWARF/ELF/CCFF info, then PGPROF populates the GUI. You are then in *feedback-only mode*.

% pgprof -exe <execname> -feedbackonly <profilename>

Note

<profilename> is ignored without warning

PGPROF tries to open the executable <*execname*>. Further, if the executable is valid with DWARF/ELF/CCFF info, then PGPROF populates the GUI. You are then in *feedback-only mode*.

Chapter 18. PGPROF Environment Variables

This chapter describes the system environment variables that you can set to change the way profiling is performed.

System Environment Variables

As you learned in "Basic Profiling," on page 127, a profiled program 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.

You can set the following system environment variables 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, this is, 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 with any of the following options: `–Mprof=func`, `–Mprof=lines`, `–Mprof=hwcts`, or `–Mprof=time`.

- PGPROF_EVENTS – Use this environment variable to specify hardware (event) counters from which to collect data.

  This variable is applied to programs compiled either with the `–Mprof=hwcts` option or executed with the **pgcollect** command. The use of hardware (event) counters is discussed in further detail in "Profiling Resource Utilization with Hardware Event Counters," on page 146.

- 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 any of the following options: `–Mprof=[func | hwcts | lines | MPI | time]`. If a program is compiled with the `–pg` option, then the output file is always called `gmon.out`.

Chapter 19. PGPROF Data and Precision

This chapter contains descriptions of the profiling mechanism that measures time, how statistics are collected, and the precision of the profiling results.

Measuring Time

The sample-based profiling mechanism collects total CPU time for programs that are compiled with options `-pg` and `-Mprof=time`, as described in "Sample-based Profiling," on page 129. The profiling mechanism collects cycle counts for programs compiled with `-Mprof=hwcts` or run under the control of **pgcollect**. PGPROF automatically converts CPU cycles into CPU time.

Programs compiled for instrumentation-based profiling with `-Mprof=lines` or `-Mprof=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 shows data for each statement.

NOTE

For instrumentation-based profiling, information provided for longer running functions are more accurate than for functions that only execute for a short time relative to the overhead of the individual timer calls. Refer to "Caveats (Precision of Profiling Results)," on page 168 for more information about profiler accuracy.

Profile Data

The following statistics are collected and may be displayed by the PGPROF profiler.

BYTES
: For MPI profiles only. This is the number of message bytes sent and received.

BYTES RECEIVED
: For MPI profiles only. This is the number of bytes received in a data transfer.

BYTES SENT
: For MPI profiles only. This is the number of bytes sent.

CALLS
: The number of times a function is called.

COST
: 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 either the `–Mprof=cost` or the `–Mprof=lines` option. For more information, refer to"Basic Profiling," on page 127.

COUNT
: The number of times a line or function is executed.

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. PGPROF sometimes generates multiple statements for a single source line; thus multiple profiling entries might appear for a single source line. To distinguish them, PGPROF uses the notation: *lineNo.statementNo*

MESSAGES
: For MPI profiles only. This is the number of messages sent and received by the function or line.

RECEIVES
: For MPI profiles only. This is the number of messages received by the function or line.

SENDS
: For MPI profiles only. This is the number of messages sent by the function or line.

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.

Caveats (Precision of Profiling Results)

Accuracy of Performance Data

The collection of performance data always introduces 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 idiosyncrasies 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 routine must consume at least a few seconds of CPU time to generate meaningful line level times.

Source 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 help you interpret the data in these cases.

Chapter 20. PGPROF Reference

This chapter provides a reference guide to the features of the PGPROF performance profiler.

For information about how to invoke PGPROF, refer to "Profiler Invocation and Initialization," on page 131.

For information about using the PGPROF text-based command-line interface, refer to Chapter 16, "*Compiler Options for Profiling*".

For information about how to choose a profiling method, build your program, and execute it to collect profile data, refer to Chapter 14, "*Getting Started with the PGPROF Profiler*".

PGPROF User Interface Overview

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 user interface, as illustrated and labeled in Figure 20.1.

Menu Bar
: Contains these menus: File, Settings, Processes, View, Sort, Search and Help.

Toolbar
: Provides navigation shortcuts and controls for frequently performed operations.

Statistics Table
: Displays profile summary information for each profile entry. Information can be displayed at up to three levels - routine, line, or assembly - depending on the type of profile data collected, how the program was built, and whether the PGPROF source file search path has been set to include the program source directories. The initial view is the routine level view.

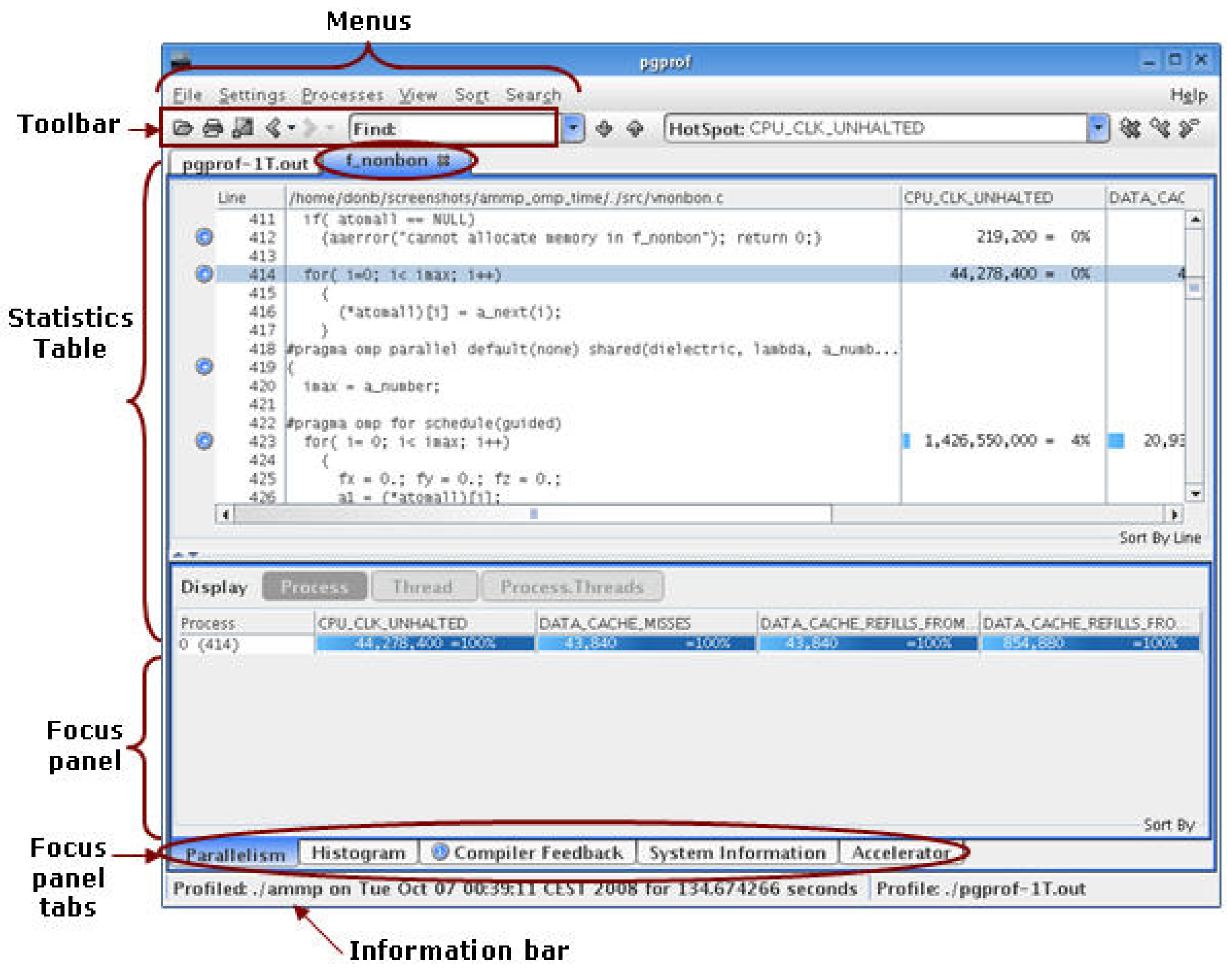
Focus Panel
: Consists of four tabbed panes labeled Parallelism, Histogram, Compiler Feedback, and System Information.

Information Bar

Displays the profile summary information such as the name of the executable, the time and date of the profile run, execution time, number of processes, if more than one, and the datafile name.

The following sections describe each of these components in more detail.

Figure 20.1. PGPROF User Interface



```
Line  /home/donb/screenshots/ammp_omp_time/./src/vnonbon.c                CPU_CLK_UNHALTED      DATA_CAC
411     if( atomall == NULL)
412       {aaerror("cannot allocate memory in f_nonbon"); return 0;}      219,200 =  0%
413
414     for( i=0; i< imax; i++)                                          44,278,400 =  0%
415       {
416         (*atomall)[i] = a_next(i);
417       }
418   #pragma omp parallel default(none) shared(dielectric, lambda, a_numb...
419   {
420     imax = a_number;
421
422   #pragma omp for schedule(guided)
423     for( i= 0; i< imax; i++)                                      1,426,550,000 =  4%      20,93
424       {
425         fx = 0.; fy = 0.; fz = 0.;
426         a1 = (*atomall)[i];
```

Process	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FROM...	DATA_CACHE_REFILLS_FRO...
0 (414)	44,278,400 =100%	43,840 =100%	43,840 =100%	854,880 =100%

Profiled: ./ammp on Tue Oct 07 00:39:11 CEST 2008 for 134.674266 seconds | Profile: ./pgprof-1T.out

PGPROF Menus

There are six menus in the GUI: File, Settings, Processes, View, Sort, and Help. This section describes each menu in detail. Keyboard shortcuts, when available, are listed next to menu items.

File Menu

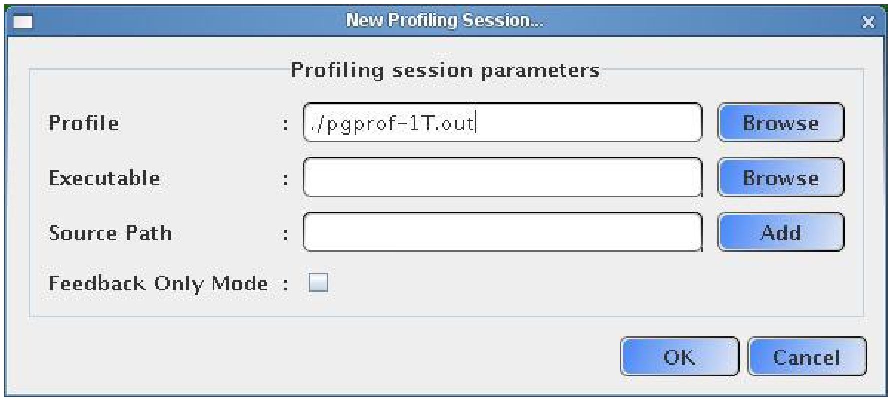
The File menu contains the following items:

- **New Window** (control N) – Select this option to create a copy of the current profiler window on your screen.
- **New Profile Session…** – Select this option to begin analyzing a different profile. A dialog box like the one in Figure 20.2 appears, requesting information about the profile data file (default `pgprof.out`), the

executable file, and the location of the source files. A new profile session is started using the information specified in the dialog box.

If Source Path is the only parameter change from current session parameters, then current session uses new Source Path to search for sources.

Figure 20.2. New Profile Session dialog box



- **Scalability Comparison…** – Select this option to open another profile for scalability comparison. Like you did for the New Profile Session… option described above, provide information about the profile data file, the executable file, and the location of the source files. Notice that the new profile contains a Scale column in its Statistics table.

  ### Note

  Another method to open profiles for scalability comparison is by using the `-scale` command-line option explained in "Profiler Invocation and Initialization," on page 131.

  For more information on scalability, refer to "Scalability Comparison," on page 144.

- **Print…** – Select this option to make a hard copy of the current profile data. The profiler processes data from the Statistics table and sends the output to a printer. A printer dialog box appears.

  You can select a printer from the Print Service Name combo box. Click 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…** – Option, output is not sent to printer, but is formatted as an editable text file. After selecting this menu item, a save file dialog box appears. Enter or choose an output file in the dialog box. Click Cancel to abort the print operation.
- **Close…** – Select this option to close the current profiling session.
- **Exit…** – Select this option to end the profiling session and exit the profiler.

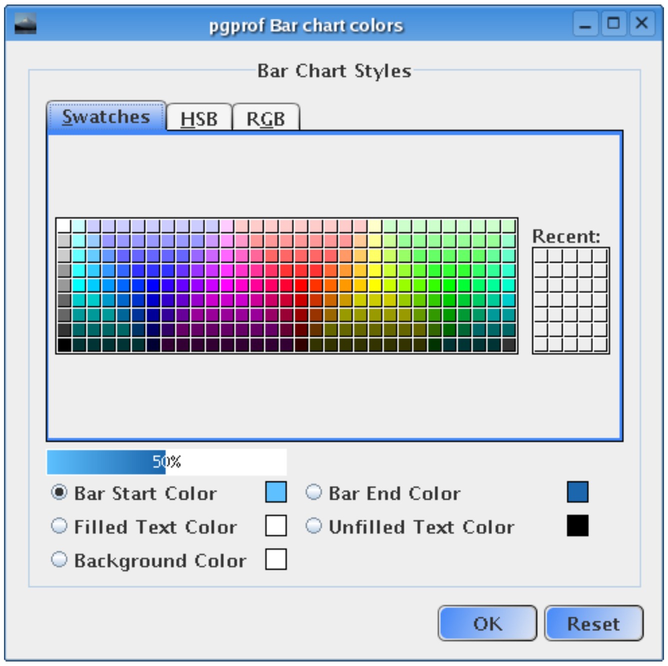
Settings Menu

Use the Settings menu to change the look and feel of the PGPROF user interface, such as fonts and chart colors. This menu contains the following items:

- **Bar Chart Colors…** – This menu option opens a color chooser dialog box and a bar chart preview panel.

  Figure 20.3 illustrates the bar chart bar colors, and the three bar chart attributes.

Figure 20.3. Bar Chart Color Dialog Box



- The bar chart bars are 'gradient filled', meaning that the color of the bar gradually transitions from the Bar Start Color to the Bar End Color. To have solid colored bars without gradient fill, simply set both of these colors to the same color.
- 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.
- The Reset button allows you to reset the selected bar chart or attribute to its previously selected color.
- The OK button accepts your changes and closes the dialog box.

Note

Closing the dialog box is the same as choosing OK.

To modify a bar chart or attribute color:

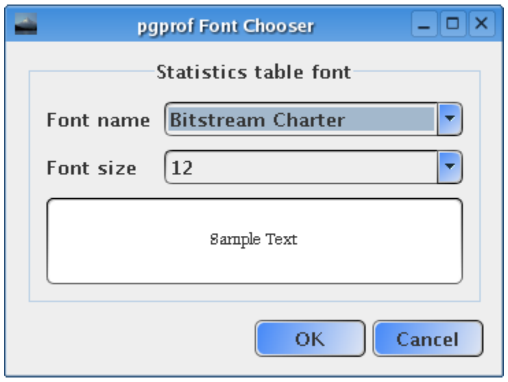
1. Click the radio button.
2. Choose a color from the Swatches, HSB, or RGB pane.
3. Click the OK button to accept the changes and close the dialog box.

PGPROF saves color selections for subsequent runs unless the Save Settings on Exit box is unchecked, as described later in this section.

- **Font...** – This menu option opens the Font Chooser dialog box illustrated in Figure 20.4.

  You may choose a new font from the list of fonts in this dialog's top combo box. You may also choose a new font size from the list of sizes in this dialog's bottom combo box. As you change the font, you can preview the changes in the Sample Text pane.

Figure 20.4. Font Chooser Dialog Box



  To change the font you must click the OK button is selected.

  Click Cancel or close the dialog box to abort any changes.

- **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, such as a button, in the user interface. Tool tips provide a summary or hint about what a particular component does. Deselect this check box to turn tool tips off.
- **Restore Factory Settings**...- Use this option to restore the default look and feel of the user interface to the original settings.
- **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 selected, PGPROF saves the current look and feel settings on exit. These settings include the size of the main window, position of the horizontal dividers, the bar chart colors, the selected font, the tool tips preference, and the options selected in the View menu. When PGPROF is started again on the same host machine, these saved settings are used. To prevent saving these settings on exit, uncheck this check box.

Processes Menu

Use the Processes menu to report process values. This 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. Selecting more than one check box is allowed. When only one check box is selected, it can't be deselected, thus guaranteeing at least one criteria to be displayed.

- When **Max** is selected, the highest value for any profile data in the Statistics Table is reported. For example, when reporting Time, the longest time for each profile entry gets reported when Max is selected.
- When **Min** 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.

Note

If the Process check box under the View menu is selected, then each row of data in the Statistics Table is labeled max, avg, and min respectively.

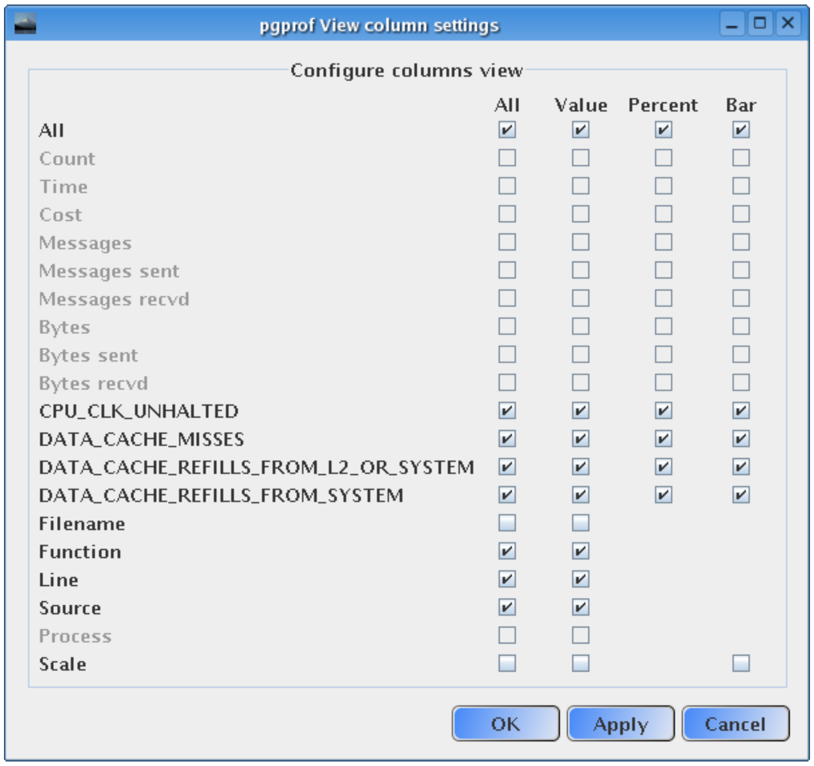
View Menu

Use the View menu to select which columns of data to view in the Statistics Table and Focus Panel tables. This selection also affects the way that tables are printed to a file and a printer, as described in the Print selection of "File Menu," on page 172.

The View menu contains the following items:

- **Configure…** - Invokes a dialog box that allows you to select the columns of the Statistics Table to be displayed, and how the data in the columns should be displayed. As illustrated in Figure 20.5, an example of this dialog box, your choices are: Value, Percent, Bar, or All.

Figure 20.5. View | Configure Dialog Box



Configure columns view

	All	Value	Percent	Bar
All	☑	☑	☑	☑
Count	☐	☐	☐	☐
Time	☐	☐	☐	☐
Cost	☐	☐	☐	☐
Messages	☐	☐	☐	☐
Messages sent	☐	☐	☐	☐
Messages recvd	☐	☐	☐	☐
Bytes	☐	☐	☐	☐
Bytes sent	☐	☐	☐	☐
Bytes recvd	☐	☐	☐	☐
CPU_CLK_UNHALTED	☑	☑	☑	☑
DATA_CACHE_MISSES	☑	☑	☑	☑
DATA_CACHE_REFILLS_FROM_L2_OR_SYSTEM	☑	☑	☑	☑
DATA_CACHE_REFILLS_FROM_SYSTEM	☑	☑	☑	☑
Filename	☐	☐		
Function	☑	☑		
Line	☑	☑		
Source	☑	☑		
Process	☐	☐		
Scale	☐	☐		☐

- **Processes…** (control P) - Allows you to select individual processes for viewing in the Focus Panel table. This menu item is enabled only when profiling an application with more than one process.

  When this item is selected, a dialog box appears with a text field. In this text field, you can enter individual processes or a range of processes for viewing. 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, the following entry tells the profiler to display information for process 0, processes 2 through 16, and process 31. These changes remain active until they are changed again or until the profiler session is terminated.

  ```
  0,2-16,31
  ```

  This entry tells the profiler to display information for process 0, processes 2 through 16, and process 31. These changes remain active until they are changed again or until the profiler session is terminated.

  To view all of the processes in the View tab, leave the text field blank.

- **Threads…** (control T) - Allows you to select individual threads for viewing in the Focus Panel table. This menu item is enabled only when profiling an application with more than one thread.

  When this item is selected, a dialog box appears with a text field. In this text field you can enter individual threads or a range of threads for viewing. Individual threads must be separated with a comma. A range of threads must be entered in the form: [start]-[end]; where start represents the first thread of the range and end represents the last thread of the range. For example:

  ```
  0,2-16,31
  ```

  This entry tells the profiler to display information for thread 0, threads 2 through 16, and thread 31. These changes remain active until they are changed again or until the profiler session is terminated.

  To view all of the threads in the View tab, leave the text field blank.

Sort Menu

Use the sort menu to alter the order in which profile entries appear in the Statistics Table, the Focus Panel | Parallelism and in the Focus Panel | Histogram. The current sort order is displayed at the bottom of each table.

In Figure 20.1, “PGPROF User Interface”, 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.

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 “Sorting Profile Data,” on page 139.

Search Menu

Use the search menu to perform a text search within the Source table.

The search menu contains the following items:

- Forward Search… (control F)
- Backward Search… (control B)
- Search Again (control G)
- Clear Search (control Q)

PGPROF displays a dialog box when you invoke the Forward Search… or Backward Search… menu items. The dialog box prompts for the text to be located. Once the text is entered and the OK button selected, PGPROF searches for the text in the function list, source code, or assembly code displayed in the Statistics Table. Matching text is displayed in red.

- To scroll forward to the next occurrence of the text entered in the dialog box, select the Forward Search... menu item.
- To scroll backwards to the first previous occurrence of the text in the Source table, select the Backward Search... menu item.
- 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.
- To abort the search, select Cancel.

Note

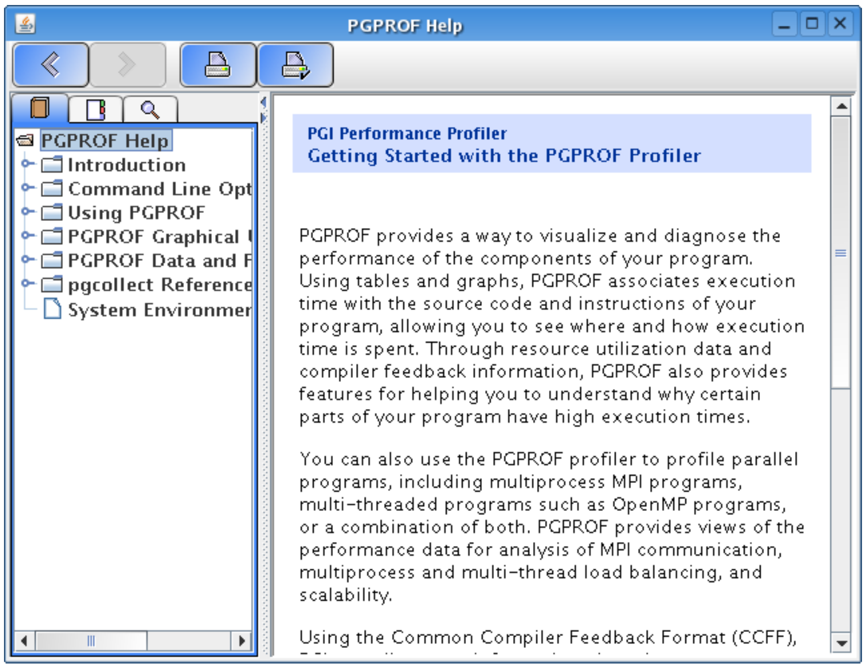
You can also use the Find: box in the toolbar to invoke the PGPROF search facility.

Help Menu

The Help menu contains the following items:

- PGPROF Help… – This option invokes PGPROF's integrated help utility illustrated in Figure 20.6, "PGPROF Help". The help utility includes an abridged version of this manual. To find a help topic, use one of the tabs in the left panel:
  - The *book* tab presents a table of contents.
  - The *index* tab presents an index of commands.
  - The *magnifying glass* tab presents a search engine.

Figure 20.6. PGPROF Help



PGI Performance Profiler
Getting Started with the PGPROF Profiler

PGPROF provides a way to visualize and diagnose the performance of the components of your program. Using tables and graphs, PGPROF associates execution time with the source code and instructions of your program, allowing you to see where and how execution time is spent. Through resource utilization data and compiler feedback information, PGPROF also provides features for helping you to understand why certain parts of your program have high execution times.

You can also use the PGPROF profiler to profile parallel programs, including multiprocess MPI programs, multi-threaded programs such as OpenMP programs, or a combination of both. PGPROF provides views of the performance data for analysis of MPI communication, multiprocess and multi-thread load balancing, and scalability.

Using the Common Compiler Feedback Format (CCFF),

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.

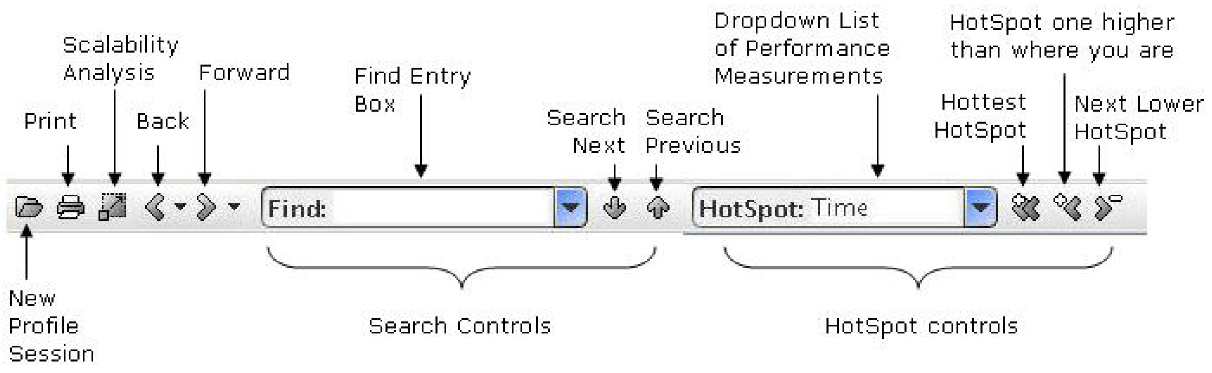
Use the printer buttons to print the current help page.

- About PGPROF… – This option opens a dialog box with version and contact information for PGPROF.

PGPROF Toolbar

As illustrated in Figure 20.7, the PGPROF toolbar provides navigation shortcuts and controls for frequently performed operations.

Figure 20.7. PGPROF Toolbar



The toolbar includes these buttons and controls:

- **New Profile Session** button - clicking this button is the same as selecting File | New Profile Session… from the menu bar.
- **Print** button - clicking this button is the same as selecting File | Print… from the menu bar.
- **Scalability Analysis** button - clicking this button is the same as selecting File | Scalability Comparison… from the menu bar.
- *Forward* and **Back** buttons - click these buttons to navigate forward and back to previous and subsequent views, respectively.

  Use the down-arrow to display the full list of views, and to select a view to jump to. These lists use a notation to describe the profile views as follows:

  ```
  profile_data_file@source_file@routine@line@address
  ```

  The address field is omitted for line-level views, and both the line and address fields are omitted for routine-level views. For example, the following item in a list would describe a view that uses profile data from `pgprof.out`, and is displaying line 370 in the routine named `solver` in source file `main.f`.

  ```
  pgprof.out@main.f@solver@370
  ```

- **Search** controls - use these to locate information. The controls include:
  - A text box labeled *Find:*. Entering a search string here and hitting `Enter` is the same as using the dialog box invoked from the Search | Forward Search… menu bar item.
  - Two buttons labeled with down and up arrows, respectively. These buttons provide Search Next and Search Previous operations, similar to Search | Search Again. Search Next searches for the next occurrence of the last search string below the current location, and Search Previous searches for the next occurrence above.
- **HotSpot Navigation** controls - use these to navigate to the most significant measurements taken in the profiling run. The controls include:
  - A drop-down menu labeled *HotSpot:*, which you use to select the specific performance measurement of interest.
  - Three navigation buttons, containing *Forward* and *Back* icons with associated plus (+) and minus (-) signs.

    When the profile is first displayed, the Statistics Table selects the row for the routine with the highest measured Time as though you had clicked on that row. To navigate to the row with the next-highest Time, you click on the button labeled with the *Forward* icon and the minus (-) sign, denoting the next Time HotSpot lower than the current one. Once you have navigated to this second HotSpot, the *Back* HotSpot buttons are activated, allowing you to navigate to the hottest HotSpot using the "<<" button, or to the next higher Time, using the "<" button.

    You can use the HotSpot drop-down menu to change the measurement used to identify the HotSpots. The default selection in the HotSpot menu is *Time*, assuming that Time is one of the available measurements. You can click on the down-arrow in the drop-down menu to select any other metric listed in the menu, then click the "Hottest" button to navigate to the row showing the routine with the highest measured value for that metric.

PGPROF Statistics Table

This section describes the PGPROF Statistics Table. The Statistics Table displays an overview of the performance data, and correlates it with the associated source code or assembly instructions. This is where you should start when analyzing performance data with PGPROF.

The Statistics Table displays information at up to three levels, depending on the type of profile data collected, how the program was built, and whether the PGPROF source file search path has been set to include the program source directories.

Performance Data Views

The Statistics Table allows you to zoom in and out on the components of your program by providing several views: the routine-level view, the line-level view, and the assembly-level view.

- The initial view when you invoke PGPROF is the routine-level view.
- To navigate to the line level from the function level, click on the Statistics Table row corresponding to the function of interest.
- To navigate to the assembly code level from the line level, click the assembly code icon, the plus (+) symbol, on the Statistics Table row that corresponds to the source line of interest.

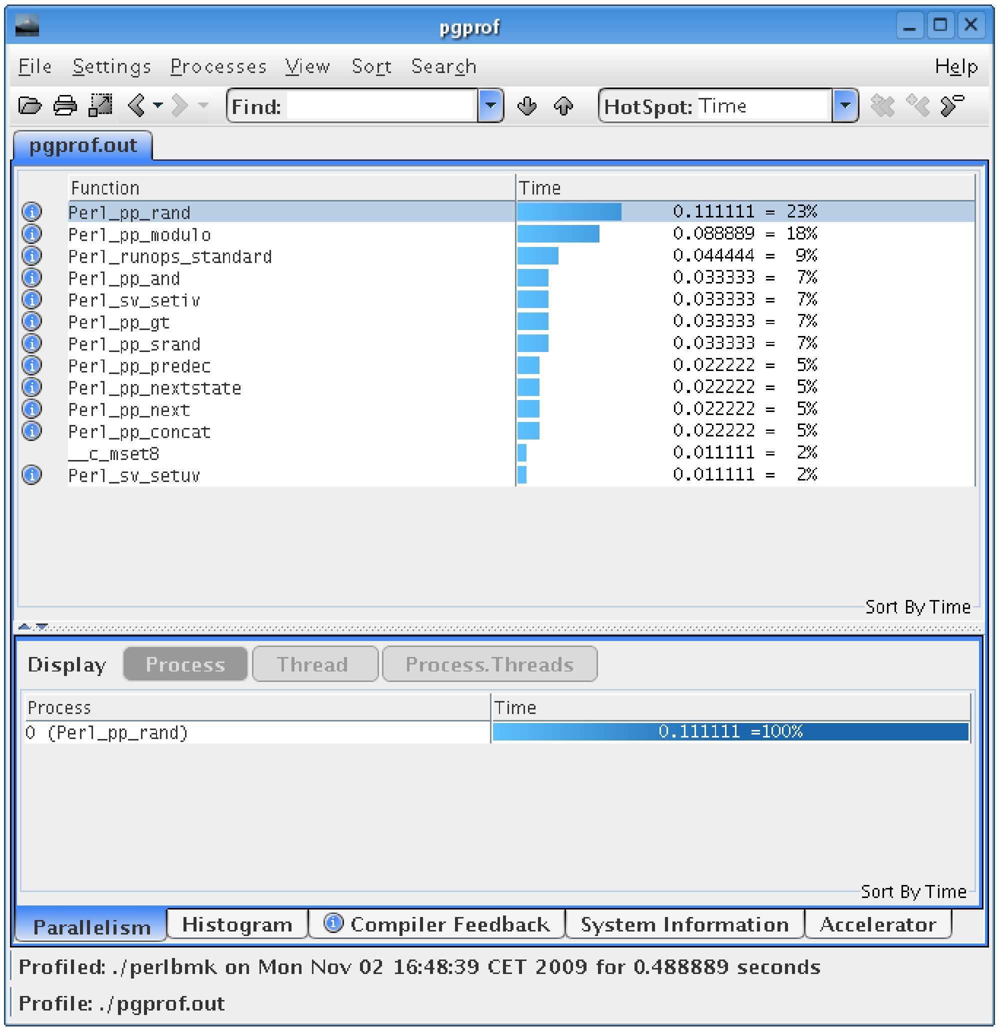
Note

You can use the View | Configure… menu option to configure the data shown in the Statistics Table.

Routine-level view

The routine-level view shows a list of the functions or subprograms in your application, with the performance data for that routine in the same row of the table. In addition, if there is any compiler feedback information for the routine, a round button containing the letter 'i' is at the far left of the row. Clicking that button populates the Compiler Feedback tab with the compiler feedback relating to that routine.

Figure 20.8. Routine-level View

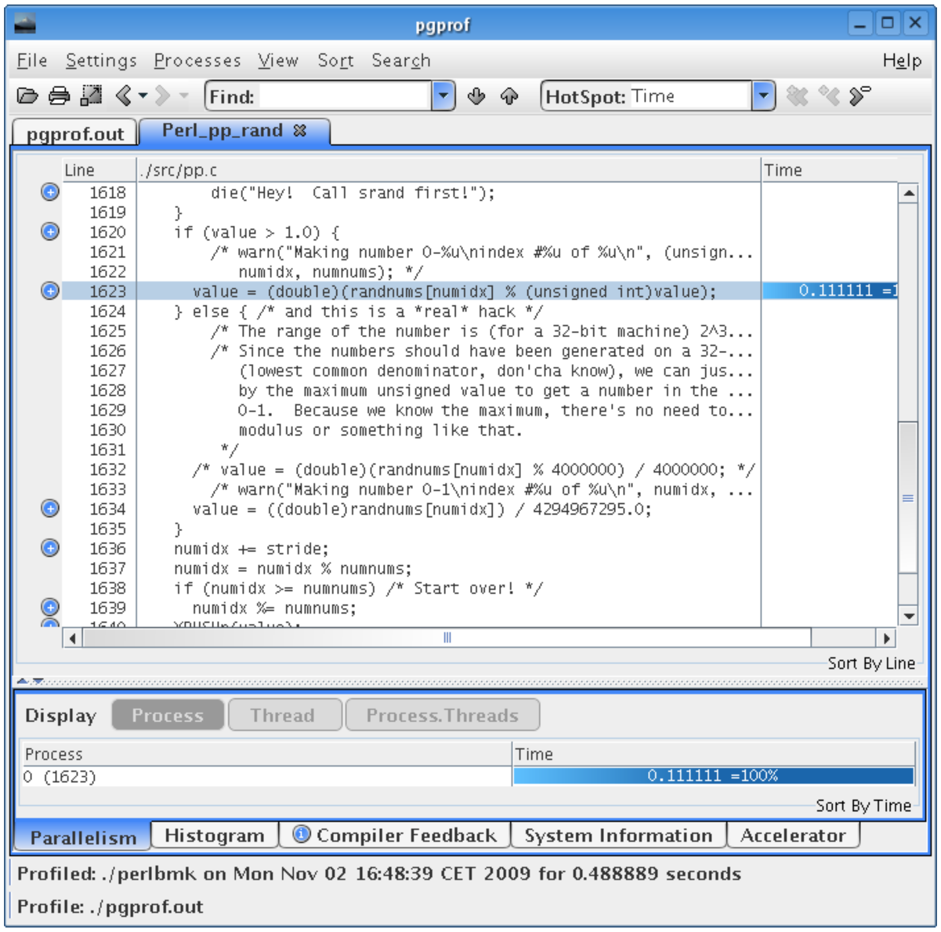

pgprof
File Settings Processes View Sort Search Help
Find:
HotSpot: Time
pgprof.out
Display Process Thread Process.Threads
Parallelism Histogram Compiler Feedback System Information Accelerator

```
Function                Time
Perl_pp_rand            0.111111 = 23%
Perl_pp_modulo          0.088889 = 18%
Perl_runops_standard    0.044444 =  9%
Perl_pp_and             0.033333 =  7%
Perl_sv_setiv           0.033333 =  7%
Perl_pp_gt              0.033333 =  7%
Perl_pp_srand           0.033333 =  7%
Perl_pp_predec          0.022222 =  5%
Perl_pp_nextstate       0.022222 =  5%
Perl_pp_next            0.022222 =  5%
Perl_pp_concat          0.022222 =  5%
__c_mset8               0.011111 =  2%
Perl_sv_setuv           0.011111 =  2%
                                    Sort By Time

Process                 Time
0 (Perl_pp_rand)        0.111111 =100%
                                    Sort By Time

Profiled: ./perlbmk on Mon Nov 02 16:48:39 CET 2009 for 0.488889 seconds
Profile: ./pgprof.out
```

Line-level View

You access the line-level view of a routine, illustrated in Figure 20.9, “Line-level View”, by clicking that routine's row in the routine-level view. PGPROF opens a new tab showing the line-level information for the routine. The tab label is the routine name and the tab contains an x which allows you to close the tab when you are done viewing the source code. The Statistics Table in the new tab shows the source code for the selected function, with performance data and Compiler Feedback buttons as with the routine-level view.

Figure 20.9. Line-level View



```
Line  ./src/pp.c                                                               Time
1618          die("Hey!  Call srand first!");
1619      }
1620      if (value > 1.0) {
1621          /* warn("Making number 0-%u\nindex #%u of %u\n", (unsign...
1622             numidx, numnums); */
1623        value = (double)(randnums[numidx] % (unsigned int)value);         0.111111 =1
1624      } else { /* and this is a *real* hack */
1625          /* The range of the number is (for a 32-bit machine) 2^3...
1626          /* Since the numbers should have been generated on a 32-...
1627             (lowest common denominator, don'cha know), we can jus...
1628             by the maximum unsigned value to get a number in the ...
1629             0-1.  Because we know the maximum, there's no need to...
1630             modulus or something like that.
1631           */
1632        /* value = (double)(randnums[numidx] % 4000000) / 4000000; */
1633          /* warn("Making number 0-1\nindex #%u of %u\n", numidx, ...
1634        value = ((double)randnums[numidx]) / 4294967295.0;
1635      }
1636      numidx += stride;
1637      numidx = numidx % numnums;
1638      if (numidx >= numnums) /* Start over! */
1639        numidx %= numnums;
```

```
Process                 Time
0 (1623)                0.111111 =100%
```

Profiled: ./perlbmk on Mon Nov 02 16:48:39 CET 2009 for 0.488889 seconds

Profile: ./pgprof.out

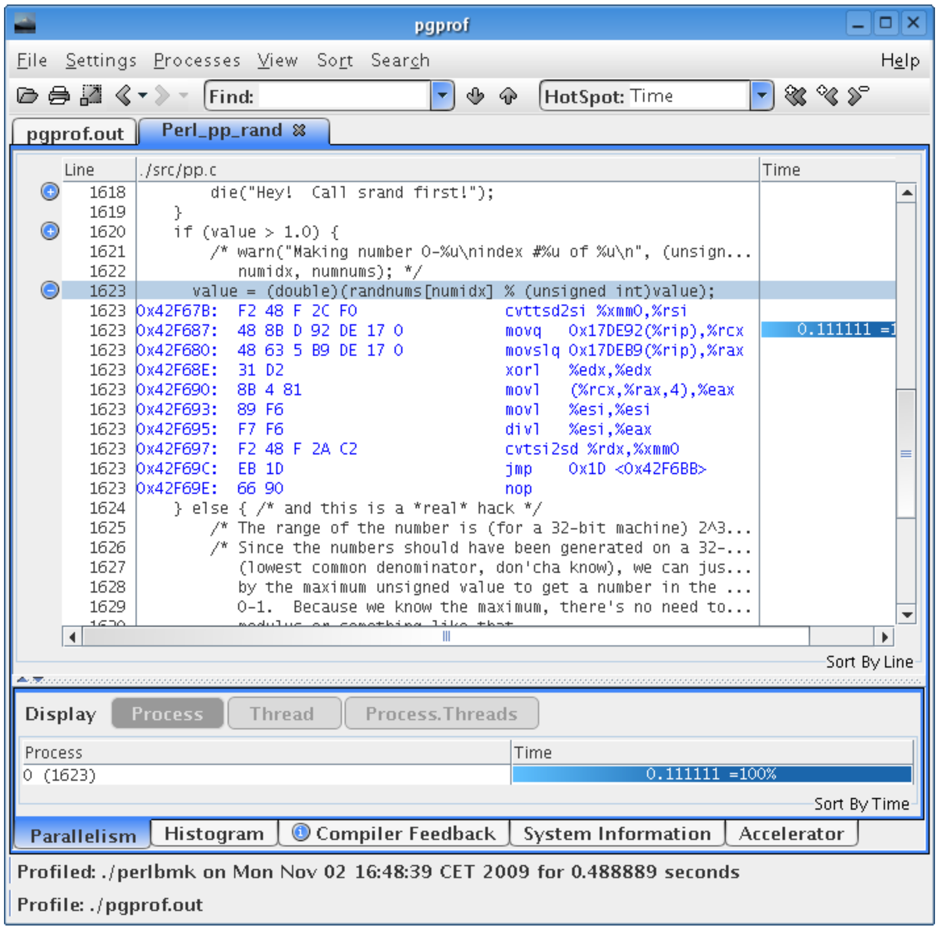
Assembly-level View

You access the assembly-level view of a source line or routine, illustrated in Figure 20.10, “Assembly-level View”, by clicking the assembly code icon, the plus (+) symbol, on the Statistics Table row that corresponds to the row of interest in the line-level view. The table changes to show the assembly code, interspersed with the source lines that were compiled to generate the code.

Note

You can use the View | Configure… menu option to configure the data shown in the Statistics Table.

Figure 20.10. Assembly-level View



```
Line  ./src/pp.c                                                                  Time
1618         die("Hey!  Call srand first!");
1619     }
1620     if (value > 1.0) {
1621         /* warn("Making number 0-%u\nindex #%u of %u\n", (unsign...
1622            numidx, numnums); */
1623       value = (double)(randnums[numidx] % (unsigned int)value);
1623 0x42F67B:  F2 48 F 2C F0              cvttsd2si %xmm0,%rsi
1623 0x42F687:  48 8B D 92 DE 17 0         movq   0x17DE92(%rip),%rcx     0.111111 =1
1623 0x42F680:  48 63 5 B9 DE 17 0         movslq 0x17DEB9(%rip),%rax
1623 0x42F68E:  31 D2                      xorl   %edx,%edx
1623 0x42F690:  8B 4 81                    movl   (%rcx,%rax,4),%eax
1623 0x42F693:  89 F6                      movl   %esi,%esi
1623 0x42F695:  F7 F6                      divl   %esi,%eax
1623 0x42F697:  F2 48 F 2A C2              cvtsi2sd %rdx,%xmm0
1623 0x42F69C:  EB 1D                      jmp    0x1D <0x42F6BB>
1623 0x42F69E:  66 90                      nop
1624     } else { /* and this is a *real* hack */
1625         /* The range of the number is (for a 32-bit machine) 2^3...
1626         /* Since the numbers should have been generated on a 32-...
1627            (lowest common denominator, don'cha know), we can jus...
1628            by the maximum unsigned value to get a number in the ...
1629            0-1.  Because we know the maximum, there's no need to...
```

```
Process                         Time
0 (1623)                        0.111111 =100%
```

Profiled: ./perlbmk on Mon Nov 02 16:48:39 CET 2009 for 0.488889 seconds

Profile: ./pgprof.out

Source Code Line Numbering

In the optimization process, the compiler may reorder the assembly instructions such that they can no longer be associated with a single line. Therefore, for optimized code, a *source line* may actually be a code block consisting of multiple source lines. This occurrence is common, and expected, and should not interfere with the tuning process when using PGPROF.

PGPROF sometimes shows multiple rows in the Statistics Table for a single source line. The line numbers for such lines are shown in the Statistics Table using the notation

```
          line.statement
```

There are several situations where this line numbering can occur:

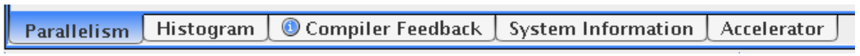
- When there is more than one statement in a source line, as in a C/C++ program where one line contains multiple statements, separated by semicolons (;).
- When the compiler generates multiple alternative implementations of a loop. The compiler may create alternate versions to handle differences in the data and how it is stored in memory.
- When there is a complicated or conditional loop setup.

For these cases, it is generally safe to sum the times and counts of all the lines. However, take care, particularly with call counts, not to double-count measurements.

PGPROF Focus Panel

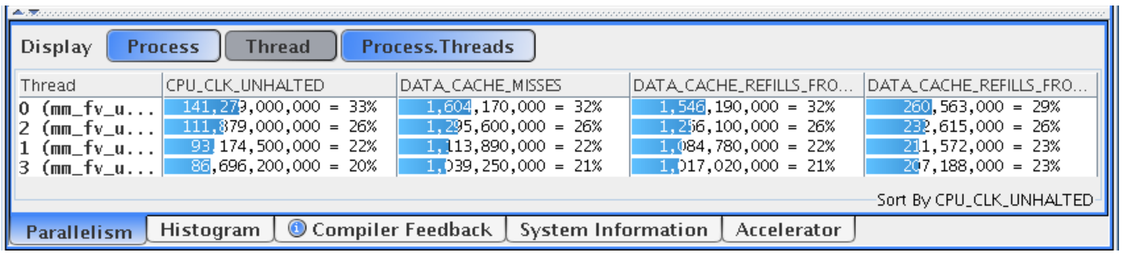
The Focus Panel consists of a number of tabs that allow you to select the focus of your attention as you view the profile data.

Figure 20.11. Focus Panel Tabs



- **Parallelism tab** - Displays a table with detailed profile information for the current profile entry.

Figure 20.12. Parallelism Tab of Focus Panel



Thread	CPU_CLK_UNHALTED	DATA_CACHE_MISSES	DATA_CACHE_REFILLS_FRO...	DATA_CACHE_REFILLS_FRO...
0 (mm_fv_u...	141,279,000,000 = 33%	1,604,170,000 = 32%	1,546,190,000 = 32%	260,563,000 = 29%
2 (mm_fv_u...	111,879,000,000 = 26%	1,295,600,000 = 26%	1,256,100,000 = 26%	232,615,000 = 26%
1 (mm_fv_u...	93,174,500,000 = 22%	1,113,890,000 = 22%	1,084,780,000 = 22%	211,572,000 = 23%
3 (mm_fv_u...	86,696,200,000 = 20%	1,039,250,000 = 21%	1,017,020,000 = 21%	207,188,000 = 23%

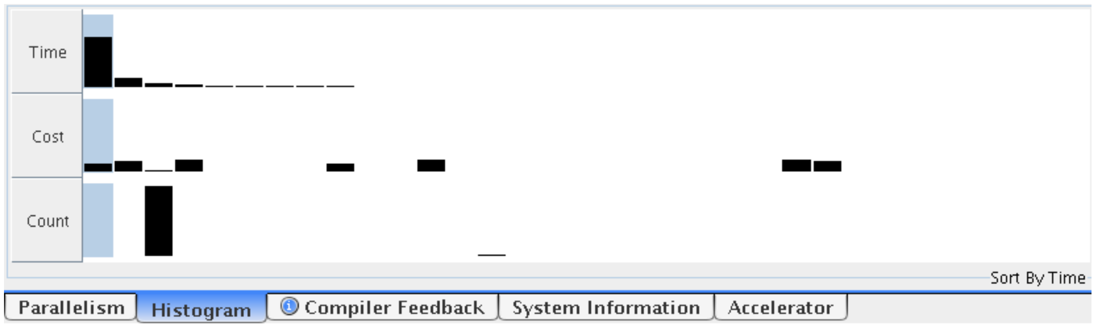
Sort By CPU_CLK_UNHALTED

- 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, the Parallelism table provides the option to view process- and/or thread-level profile information. Three buttons labeled Process, Thread, and Process.Threads are available for you to use to toggle between these views when such views are available. We refer to these buttons as the *Process/Thread Selector* buttons.

The default heading for the leftmost column is *Process(es)*. When profiling a multi-threaded application, the heading in the leftmost column reflects whatever is selected by the Process/Thread Selector buttons. When the leftmost column is displaying processes or threads, denoted `Process(es).Threads` in the column heading, each entry is displayed using the notation (Process_ID).(Thread_ID). Following the process/thread ID, the filename, routine name, or line number display is displayed, enclosed in parentheses. This display provides additional ownership information of the process/thread, as well as acting as a secondary key for sorting. For more information on sorting, refer to "Sort Menu," on page 177.

- **Histogram tab** – Displays a histogram of one or more profiled data items. The performance measurements displayed are the same as those selected in the View menu, described in "View Menu," on page 176.

Figure 20.13. Histogram Tab of Focus Panel

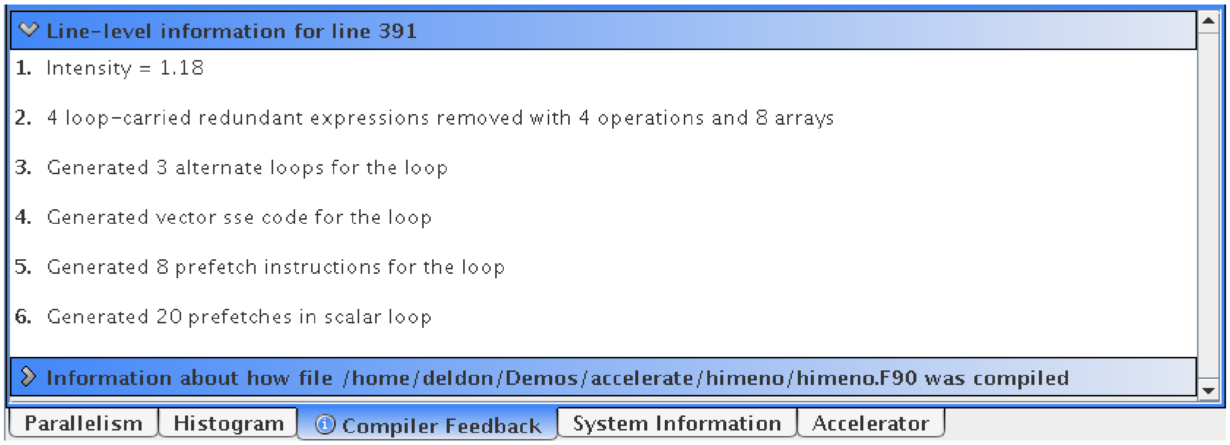


- Each bar graph corresponds to one of the performance measurements.
- Each vertical bar corresponds to a profile entry.
- The bars are sorted in the order specified in the Sort menu, described in "Sort Menu," on page 177. Further, the sort order of the Statistics table is reflected in the sort order of the histogram.
- Clicking on a bar displays information for the corresponding profile item in the Statistics Table.
- Double-clicking on a bar drills down into the profile for the portion of the program corresponding to the bar.
- Selected bars are highlighted in blue.

- **Compiler Feedback tab** - Displays information provided by the compiler regarding the characteristics of a selected piece of the program, including optimization and parallelization information, obstacles to optimization or parallelization, and hints about how to improve the performance of that portion of the code. Such information is available at the line level and the function list level.

  To access the information, click on an *info* button displayed at the far left of the Statistics Table. If any information is available, round, blue buttons, containing a lower-case 'i', are displayed.

Figure 20.14. Compiler Feedback Tab of Focus Panel



Line-level information for line 391

1. Intensity = 1.18
2. 4 loop-carried redundant expressions removed with 4 operations and 8 arrays
3. Generated 3 alternate loops for the loop
4. Generated vector sse code for the loop
5. Generated 8 prefetch instructions for the loop
6. Generated 20 prefetches in scalar loop

Information about how file /home/deldon/Demos/accelerate/himeno/himeno.F90 was compiled

The information is separated into these categories of information about:

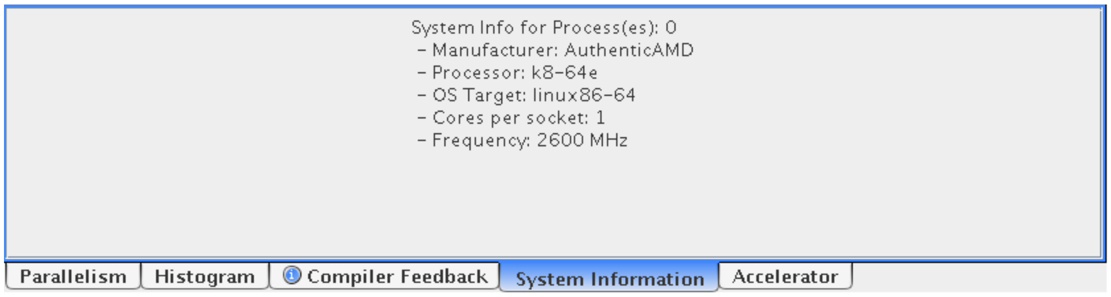
- A source line
- Routines referenced inside another routine
- Variables referenced inside a routine
- How a file was compiled

Each category is represented by a wide bar that functions like a button. Clicking the bar expands the display to show the information in that category. If no information is available in a given category, that category is not listed.

This information is only available if the program was compiled and also linked using either the `–Mprof` or the `–Minfo=ccff` option. In some cases it is necessary to build programs using profile-guided optimization by building with `–Mpfi` or`–Mpfo`, as described in the section *Profile-Feedback Optimization using -Mpfi/Mpfo* in the Optimizing and Parallelizing chapter of the PGI User's Guide. Profile-guided optimization can often determine loop counts and other information needed to calculate the Compute Intensity for a given statement or loop.

- **System Information tab** - Displays a panel containing information about the system on which the profile run was executed. If the profile run was executed on multiple systems, there may be information for multiple systems. As illustrated in Figure 20.15, the information can include:

Figure 20.15. System Information Tab of Focus Panel



```
System Info for Process(es): 0
 - Manufacturer: AuthenticAMD
 - Processor: k8-64e
 - OS Target: linux86-64
 - Cores per socket: 1
 - Frequency: 2600 MHz
```

- *Process(es)* - the process number(s), or MPI rank(s), of the processes that executed on the specified system in the profiling run.
- *Manufacturer* - the processor manufacturer
- *Processor* - the cpu architecture on which the profiling run was executed, specified using the architecture name used with the PGI compilers' '-tp' option. See the PGI User's Guide for more information.
- *OS Target* - the operating system platform that the executable was built for. Note that although the processor may be a 64-bit processor, the executable may target a 32-bit platform.
- *Total cores* - the total number of processor cores on the system
- *Cores per socket* - the number of processor cores per CPU. If the total cores value is 16, and the cores per socket value is 4, then you know you are using a quad-core processor.
- *Frequency* - the processor frequency

- **Accelerator tab** - Displays profiling information provided by the accelerator. This information is available in the file `pgprof.out` and is collected by using **pgcollect** on an executable binary compiled for an accelerator target. For more information on **pgcollect**, refer to Chapter 22, "*pgcollect Reference*".

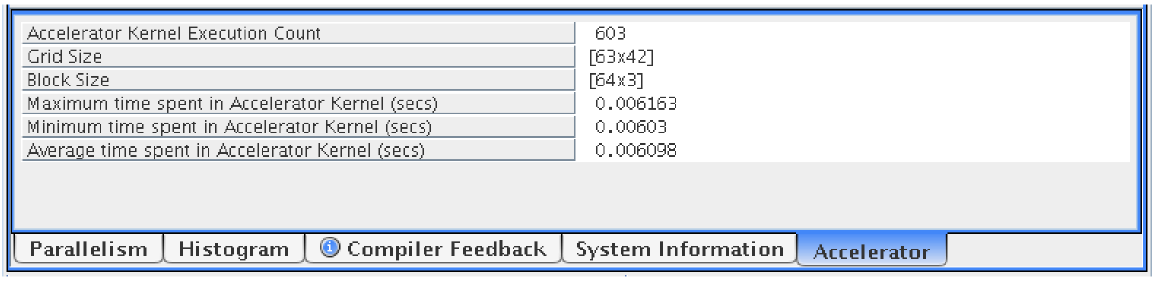
  The profiling information is relative either to an Accelerator Region or to an Accelerator Kernel.

  Accelerator Region
  : An accelerator region is a region of code that has been executed on the accelerator device. An accelerator region might transfer data between the host and the accelerator device. Further, an accelerator region can be split into several accelerator kernels.

  Accelerator Kernel
  : An accelerator kernel is a compute intensive, highly parallel portion of code executed on an accelerator device. Each compiler-generated kernel is code executed by a block of threads mapped into a grid of blocks.

  Figure 20.16 illustrates one possible display for the Accelerator tab, one that is relative to the Accelerator Kernel:

Figure 20.16. Accelerator Tab of Focus Panel



Accelerator Kernel Execution Count	603
Grid Size	[63x42]
Block Size	[64x3]
Maximum time spent in Accelerator Kernel (secs)	0.006163
Minimum time spent in Accelerator Kernel (secs)	0.00603
Average time spent in Accelerator Kernel (secs)	0.006098

PGPROF displays two Accelerator events in the Statistic table:

- *Accelerator Region Time* - the time, in seconds, spent in the Accelerator region
- *Accelerator Kernel Time* - the time, in seconds, spent in the Accelerator kernel.

When a user selects a line for which one of these events is non-zero, the table in the Accelerator tab contains details about that event. The information displayed depends on the selection.

If a user selects a line in which both events are non-zero, then the Accelerator tab displays only Accelerator Initialization Time, Accelerator Region Time, and Accelerator Kernel Time.

Accelerator Region Timing Information

When you select a non-zero Accelerator Region Timing item, you see the following information in the Accelerator tab:

- *Accelerator Initialization Time* - the time, in seconds, spent in accelerator initialization for the selected region.
- *Accelerator Kernel Time* - the time, in seconds, spent in compute kernel(s) for the selected region.

- *Data Transfer Time* - the time, in seconds, spent in data transfer between host and accelerator memory.
- *Accelerator Execution Count* - execution count for the selected region.
- *Maximum time spent in accelerator region (w/o init)* - the maximum time, in seconds, spent in a single execution of selected region.
- *Minimum time spent in accelerator region (w/o init)* - the minimum time, in seconds, spent in a single execution of selected region.
- *Average time spent in accelerator region (w/o init)* - the average time, in seconds, spent per execution of selected region.

Note

The table does not contain values that are not relevant, such as zero values or values that cannot be computed. For example, in a routine-level profile, a routine can execute multiple accelerator regions. In this instance, only time spent in Initialization, in the Region, and in the Kernel can be accurately computed so other values are not displayed in the Accelerator tab.

Accelerator Kernel Timing Information

When you select a non-zero Accelerator Kernel Timing item, you see the following information in the Accelerator tab:

- *Kernel Execution Count* - execution count for the selected kernel.
- *Grid Size* - the size, in 1D [X] or 2D [XxY], of the grid used to execute blocks of threads for the selected kernel.
- *Block Size* - the size, in 1D [X], 2D [XxY] or 2D [XxYxZ], of the grid used to execute blocks of threads for the selected kernel.
- *Maximum time spent in accelerator kernel* - the maximum time, in seconds, spent in a single execution of selected kernel.
- *Minimum time spent in accelerator kernel* - the minimum time, in seconds, spent in a single execution of selected kernel.
- *Average time spent in accelerator kernel* - the average time, in seconds, spent per execution of selected kernel.

Note

When there are multiple invocations of the same kernel in which the grid-size and/or block-size changes, the size information displayed in the Accelerator tab is expressed as a range. For example, if the same kernel could be executed with a 2D-block of size [2,64] and a 2D-block of size [4,32], then the size displayed in Accelerator tab is the range: [2-4, 32-64].

Chapter 21. The PGPROF Command Line Interface

The user interface for non-GUI (Win32) versions of the PGPROF profiler is a simple command language. This command language is available in the profiler through the `-text` option. The language is composed of commands and arguments separated by white space. A **pgprof>** prompt is issued unless input is being redirected.

This chapter describes the PGPROF profiler command line interface, providing both a summary and then more details about the commands.

Command Description Syntax

This chapter 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.

PGPROF Command Summary

Table 21.1 summarizes the commands for use in the non-GUI version of PGPROF, providing the applicable arguments and a brief description of the use of each command. The next section provides more details about each command.

Table 21.1. PGPROF Commands

Name	Arguments	Usage
d[isplay]	[display options] | all | none	Specify display information.
he[lp]	[command]	Provide brief command synopsis.
h[istory]	[size]	Display the history list, which stores previous commands in a manner similar to that available with csh or dbx.
l[ines]	function [[>] filename]	Print (display) the line level data together with the source for the specified function.
a[sm]	routine [[>] filename]	Print (display) the instruction and line level data together with the source and assembly for the specified routine.
lo[ad]	[datafile]	Load a new dataset. With no arguments reloads the current dataset.
m[erge]	datafile	Merge the profile data from the named datafile into the current loaded dataset.
pro[cess]	processor_num	For multi-process profiles, specify the processor number of the data to display.
p[rint]	[[>] filename]	Print (display) the currently selected function data.
q[uit]		Exit the profiler.
sel[ect]	calls | timecall | time | cost | cover | all [[>] cutoff]	Display data for a selected subset of the functions.
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.
src[dir]	directory	Set the source file search path.
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")
th[read]	thread_num	Specify a thread for a multi-threaded process profile.
t[imes]	raw | pct	Specify whether time-related values should be displayed as raw numbers or as percentages. The default is pct.
! (history)	! | num | -num | string	Repeat recent commands

Command Reference

This section provides more details about the commands in the previous Command Summary Table.

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 is placed in the named file. The '>' means redirect output, and is optional. This command is only available on platforms that support assembly-level profiling.

ccff

```
c[cff] file[@function] [line_number]
```

Print compiler feedback for the specified file, function, or source line. PGI compilers can produce information in the Common Compiler Feedback Format (CCFF) that provides details about the compiler's analysis and optimization of your program. Often this information can illuminate ways in which to further optimize a program.

The CCFF information is produced by default when using the `–Mprof'` compiler option, but if you are profiling with the **pgcollect** command, you must build your program with the '`–Minfo=ccff`' compiler option to produce this information.

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 is 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 is placed in the named file. The '>' means redirect output, and is optional.

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 is 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.

Chapter 22. pgcollect Reference

The **pgcollect** command is a development tool used to control collection of performance data for analysis using the pgprof performance profiler. This chapter describes how to use **pgcollect**.

pgcollect Overview

pgcollect runs the specified program with the supplied arguments. While the program runs, pgcollect gathers performance statistics. When the program exits, the data that is gathered is written to a file. You can then use this file in the PGPROF performance profiler to analyze and tune the performance of the program.

The **pgcollect** command supports two distinct methods of performance data collection:

Time-based sampling
: Creates a time-based profile that correlates execution time to code, showing the amount of time spent in each routine, each source line, and each assembly instruction in the program. For more information on time-based profiling, refer to "Time-based Sampling," on page 129.

Event-based sampling
: Supported only on liinux86-64 systems, creates an event-based profile that correlates hardware events to program source code. In this method, **pgcollect** uses hardware event counters supported by the processor to gather resource utilization data, such as cache misses.

Note

This method requires co-installation of the open source performance tool `OProfile`.

For more information on event-based profiles, refer to "Event-based Sampling," on page 130.

Both forms of the **pgcollect** command gather performance data that can be correlated to individual threads, including OpenMP threads, as well as to shared objects, dynamic libraries, and DLLs.

For current availability of **pgcollect** and **pgcollect** features on a given platform, refer to the PGI Release Notes.

Invoke pgcollect

The command you use to invoke **pgcollect** depends on the type of profile you wish to create.

Use the following command to invoke **pgcollect** for time-based sampling:

```
pgcollect [-time] program [program_args]
```

Use the following command to invoke **pgcollect** for event-based sampling available on Linux86-64:

```
pgcollect [<event_options>] program_or_script [program_or_script_args]
```

`program` or `program_or_script` are either the filename of the program to be profiled, or the name of a script that invokes the program. When applicable, you can provide arguments for the specified program or script: *program_args* or *program_or_script_args*.

The following sections describe the **pgcollect** command-line options in more detail.

Build for pgcollect

If your program was built with PGI compilers, you do not need to use any special options to use **pgcollect**. However, if your programs are built using the `-Minfo=ccff` option, then PGPROF can correlate compiler feedback and optimization hints with the source code and performance data.

If you built your program using a non-PGI compiler, consider building with debugging information so you can view source-level performance data. Be aware, however, that building with debugging information may change the performance of your program.

General Options

This section describes options that apply to all forms of the **pgcollect** command. For options specific to controlling time-based or event-based profiling, refer to "Time-Based Profiling Options" and "Event-based Profiling Options" respectively.

-V
: Display the version of pgcollect being run.

-help
: Show **pgcollect** usage and switches.

Time-Based Profiling

Time-based profiling runs the program using time-based sampling. This form of **pgcollect** uses operating system facilities for sampling the program counter at 10-millisecond intervals.

Time-Based Profiling Options

-time
: Provide time-based sampling only. The sampling interval is 10 milliseconds. This option is the default.

When using **pgcollect** for time-based sampling, you can have multiple instances of **pgcollect** running simultaneously, but doing so is not recommended, since this will probably skew your performance results.

Event-Based Profiling

You can use the **pgcollect** command on linux86-64 to drive an OProfile session. Event-based profiling provides several predefined data collection options that gather data from commonly used counters.

For event-based sampling, the only required argument is the `program_or_script`, which is either the filename of the program to be profiled, or the name of a script that invokes the program. Using a script can be 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.

When applicable, you can provide arguments for the specified program or script.

Since OProfile provides only system-wide profiling, when you invoke **pgcollect** it provides a locking mechanism that allows only one invocation to be active at a time.

Note

The **pgcollect** locking mechanism is external to OProfile and does not prevent other profile runs from invoking **opcontrol** through other mechanisms.

Root Privileges Requirement

When using **pgcollect** for event-based profiling, you control the OProfile kernel driver and the sample collection daemon via the OProfile command **opcontrol**. This control requires root privileges for management operations. Thus, invocations to **opcontrol**, performed by **pgcollect**, are executed via the **sudo** command.

When using **pgcollect**, you control the OProfile kernel driver and the sample collection daemon via the OProfile command **opcontrol**. This control requires root privileges for management operations. Thus, invocations to opcontrol, which are performed when **pgcollect** is used, are executed via the **sudo** command.

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` to permit all members of the group 'sw' to run **opcontrol** with root privileges.

```
# User alias specification
User_Alias SW = %sw
     ...
SW ALL=NOPASSWD: /usr/bin/opcontrol
```

Interrupted Profile Runs

pgcollect shuts down the OProfile daemon when interrupted. However, if the script is terminated with SIGKILL, you must execute the following:

```
pgprof -collect -shutdown
```

Executing this command is important because if the OProfile daemon is left running, disk space on the root file system is eventually exhausted.

Event-based Profiling Options

-check-events
: Do not execute a profiling run, just check the event settings specified on the command line.

-exe <exename>
: Specify the program to be profiled. You only need to use `-exe` when the file argument is a script that invokes the program.

-list-events
: List profiling events supported by the system.

-shutdown
: Shut down the profiling interface. You only need to use this option in rare cases when a profiling run was interrupted and OProfile was not shut down properly.

Predefined Performance Data Collection Options

-allcache
: Profile instruction, data, and branch cache misses

-dcache
: Profile various sources of data cache misses

-imisses
: Profile instruction cache-related misses.

-hwtime <millisecs>
: Provide time-based sampling only. Specify the sampling interval in milliseconds.

User-Defined Performance Data Collection Options

-es-function <name>
: Set profile events via a shell function.

-event <spec>
: Manually add an event profile specification. An event profile specification is an **opcontrol '--event'** argument; that is, the event profile specification provided on the command line is appended to '--event=' and passed as an argument to **opcontrol**.

-post-function <name>
: Execute a shell function after profiling is complete.

Defining Custom Event Specifications

The `pgcollect '-event=EVENTSPEC'` options are accumulated and used to specify events to be measured. For more information about these events, refer to the **opcontrol** man page.

x64 processors provide numerous event counters that measure the usage of a variety of processor resources. Not all processors support the same set of counters. To see which counters are supported on a given system, use the following command:

```
 pgcollect -list-events
```

The output of this command also provides information on event masks (the hex value in the event specification) and minimum overflow values.

Here are two examples of shell functions providing event specifications to pgcollect. These functions would be implemented in a `.pgoprun` file:

Example 22.1. Custom Event Example 1

This function specifies the events needed to calculate cycles per instruction (CPU_CLK_UNHALTED / RETIRED_INSTRUCTIONS). The fewer cycles used per instruction, the more efficient a program is.

```
cpi_data () {
    event[${#event[@]}]=--event=CPU_CLK_UNHALTED:500000:0x00:0:1
    event[${#event[@]}]=--event=RETIRED_INSTRUCTIONS:500000:0x00:0:1
}
```

To use these events, invoke **pgcollect** with the following arguments:

```
-es-function cpi_data
```

Example 22.2. Custom Event Example 2

Example 2

This function specifies events needed to determine memory bandwidth:

```
mem_bw_data () {
    event[${#event[@]}]=--event=CPU_CLK_UNHALTED:500000:0x00:0:1
    event[${#event[@]}]=--event=SYSTEM_READ_RESPONSES:500000:0x07:0:1
    event[${#event[@]}]=--event=QUADWORD_WRITE_TRANSFERS:500000:0x00:0:1
    event[${#event[@]}]=--event=DRAM_ACCESSES:500000:0x07}:0:1
}
```

To use these events, invoke **pgcollect** with the following arguments:

```
    -es-function mem_bw_data
```

Index