PGI Tools Guide

Parallel Tools
for Scientists and Engineers

The Portland Group Compiler Technology
STMicroelectronics
9150 SW Pioneer Court, Suite H
Wilsonville, OR 97070

PGI Tools Guide
Copyright © 2004 STMicroelectronics, Inc.
All rights reserved.
Printed in the United States of America

Part Number:	2040-990-888-0603
Printing:	Release 5.2, June 2004
Technical support:	trs@pgroup.com
Sales:	sales@pgroup.com
Web:	http://www.pgroup.com

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Preface

This guide describes how to use The Portland Group Compiler Technology (PGI) Fortran, *C*, and *C*++ debugger and profiler tools. In particular, these include the *PGPROF* profiler, and the *PGDBG* debugger. You can use the PGI compilers and tools to debug and profile serial (uni-processor) and parallel (multi-processor) applications for X86 and AMD64 processor-based systems.

Intended Audience

This guide is intended for scientists and engineers using the PGI debugging and profiling tools. To use these tools, you should be aware of the role of high-level languages (e.g., Fortran, *C*, *C*++) and assembly-language in the software development process and should have some level of understanding of programming. The PGI tools are available on a variety of operating systems for the X86 and AMD64 hardware platforms. You need to be familiar with the basic commands available on your system.

Finally, your system needs to be running a properly installed and configured version of the compilers and tools. For information on installing PGI compilers and tools, refer to the installation instructions.

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 9x, *C*, and *C*++ and includes extensions from MIL-STD-1753, VAX/VMS Fortran, IBM/VS Fortran, SGI Fortran, Cray Fortran, and K&R *C*. *PGF77*, *PGF90, PGCC* ANSI *C,* and *C++* support parallelization extensions based on the OpenMP defacto standard. *PGHPF* supports data parallel extensions based on the High Performance Fortran (HPF) defacto standard. The PGI Fortran reference manuals describe Fortran statements and extensions as implemented in the PGI Fortran compilers. *PGDBG* permits debugging of serial and parallel (OpenMP and/or MPI) programs compiled with PGI compilers. *PGPROF* permits profiling of serial and parallel (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).
- *American National Standard Programming Language FORTRAN*, ANSI X3. -1991 (1991).
- *International Language Standard ISO Standard 1539-199 (E).*
- *Fortran 90 Handbook*, Intertext-McGraw Hill, New York, NY, 1992.
- *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 Fortran Application Program Interface*, Version 1.1, November 1999, http://www.openmp.org.
- *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998, 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.

- HPDF Standard (High Performance Debugging Forum)
  http://www.ptools.org/hpdf/draft/intro.html

Organization

This manual is divided into the following chapters:

Chapter 1

The PGDBG Debugger describes the *PGDBG* symbolic debugger. *PGDBG* is a symbolic debugger for Fortran, *C*, *C*++ and assembly language programs. Sections 1.1 through 1.13 describe *PGDBG* invocation, commands, signals, debugging Fortran and C++ using *PGDBG*, the *PGDBG* graphical user interface, and *PGDBG* parallel debugging capabilities.

1.14 Debugging Parallel Programs with PGDBG describes how to invoke the debugger for thread-parallel (SMP) debugging and for process-parallel (MPI) debugging.

1.15 Thread-parallel and Process-parallel Debugging 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.

1.16 OpenMP Debugging describes some debug situations within the context of a single process composed of many OpenMP threads.

1.17 MPI Debugging describes how *PGDBG* is used to debug parallel-distributed MPI programs and hybrid distributed SMP programs.

Chapter 2

The PGPROF Profiler describes the *PGPROF* Profiler. This tool analyzes data generated during execution of specially compiled *C*, *C*++, F77, F9x and HPF programs.

Conventions

This guide uses the following conventions:

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

Related Publications

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

- *PGF77 Reference User Manual* describes the FORTRAN 77 statements, data types, input/output format specifiers, and additional reference material.
- *PGHPF Reference Manual* describes the HPF statements, data types, input/output format specifiers, and additional reference material.
- *System V Application Binary Interface Processor Supplement* by AT&T UNIX System Laboratories, Inc. (Prentice Hall, Inc.).
- *FORTRAN 90 HANDBOOK,* Complete ANSI/ISO Reference (McGraw-Hill, 1992).
- *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 5.2 Release Notes*, FAQ, Tutorials
  http://www.pgroup.com/docs.htm
- MPI-CH
  http://www.netlib.org/
- OpenMP
  http://www.openmp.org/
- Ptools (Parallel Tools Consortium)
  http://www.ptools.org/
- HPDF (High Performance Debugging Forum) Standard
  http://www.ptools.org/hpdf/draft/intro.html

System Requirements

- PGI CDK 5.2, or WS 5.2
- Linux (See http://www.pgroup.com/faq/install.htm for supported releases)
- Intel X86 (and compatible), AMD Athlon, AMD64 processors

Chapter 1

The *PGDBG* Debugger

This chapter describes the *PGDBG* symbolic debugger. *PGDBG* is a symbolic debugger for Fortran, *C*, *C++* and assembly language programs. It allows you to control the execution of programs using breakpoints and single-stepping, and lets you check the state of a program by examining variables, memory locations, and registers. The following are *PGDBG* capabilities.

- Provides the capability to debug SMP Linux programs.
- Provides the capability to debug MPI programs on Linux clusters.
- Provides the capability to debug hybrid SMP/MPI programs on Linux clusters where each node contains multiple CPUs sharing memory but where each node has a separate memory from all other nodes.

1.1 Definition of Terms

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

For an introduction to terminology used to describe parallel debugging, see *Section 1.14.1 Processes and Threads*.

1.1.1 Compiler Options for Debugging

Use the *–g* compiler command line option to build programs for debugging. This option causes information about the symbols and source files in the program to be included in the executable file. The –g option also sets the optimization to level zero unless you specify *–O* on the command line. Programs built with –g and optimization levels higher than –O0 can be debugged, but due to transformations made to the program during optimization, source-level debugging may not be reliable. Machine-level debugging (e.g., accessing registers, viewing assembly code, etc.) will be reliable, even with optimized code. Programs built without *–g* can be debugged; however, information about types, local variables, arguments and source file line numbers is not available unless you specify *–g*.

When the *-g* compiler command line option is used, PGI compilers emit DWARF Version 2 debug information by default. To emit DWARF Version 1 debug information, specify the *-Mdwarf1* option with the *-g* option at the compiler command line.

1.2 Invocation and Initialization

PGDBG is invoked using the `pgdbg` command as follows:

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

where `arguments` may be any of the command-line arguments described in the following section, *Command-line Arguments*. See *1.14.4.1 Invoking PGDBG: MPI Debugging* for how to debug an MPI program.

The `program` or *debugee* is the name of the target program being debugged. The arguments `arg1 arg2 … argn` are the command-line arguments to the target program. After initiating a debug session using the commands above, *PGDBG* will invoke its Graphical User Interface (GUI) by default (*Section 1.12 PGDBG GRAPHICAL USER INTERFACE*). You can also invoke a command-line interface with the `-text` command-line argument (*Section 1.3 Command-Line Arguments*). After starting *PGDBG,* the debugger begins by creating a symbol table for the program. Then the program is loaded into memory.

If an initialization file named `.pgdbgrc` exists in the current directory or in the home directory, it is opened and *PGDBG* executes the commands in the file. The initialization file is useful for defining common aliases, setting breakpoints and for other startup commands. If an initialization file is found in the current directory, then the initialization file in the home directory, if there is one, is ignored. However, a *script* command placed in the initialization file may execute the initialization file in the home directory, or execute *PGDBG* commands in any other file (for example in the file `.dbxinit` if you have a *dbx* debugger initialization file set up).

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

1.3 Command-Line Arguments

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

–dbx	Start the debugger in *dbx* mode.
––program_args	*PGDBG* passes all arguments following this command line option to the program being debugged if an executable is included on the command line. This command-line argument should appear after the name of the executable (e.g., pgdbg a.out –program_args 0 1 2 3).
–s *startup*	The default initialization file is ~/.pgdbgrc. The *–s* option specifies an alternate initialization file *startup*.
–c "*command*"	Execute the debugger command *command* (*command* must be in double quotes) before executing the commands in the startup file.
–r	Run the debugger without first waiting for a command. If the program being debugged runs successfully, the debugger terminates. Otherwise, the debugger is invoked and stops when the trap occurs.
–text	Run the debugger using a command-line interface (CLI). The default is for the debugger to launch in graphical user interface (GUI) mode.
–tp k8-32	Debug a program running on an X86 target machine. This option is only necessary if the default *PGDBG*, determined by the PATH environment variable, is not capable of debugging X86 targeted programs (AMD Opteron™ only).
–tp k8-64	Debug a program running on an AMD64 target machine. This option is only necessary if the default *PGDBG*, determined by the PATH environment variable, is not capable of debugging AMD64 targeted programs (AMD Opteron™ only).
-motif	Use the older (Motif) version of the Graphical User Interface (GUI) (Not available on every platform).
-help	Display a list of command-line arguments (this list).

-I <directory>	Adds <directory> to the list of directories that *PGDBG* uses to search for source files.

1.4 Command Language

There are two methods for telling *PGDBG* what to do. You can type a command through a command-line interface or invoke a command through a Graphical User Interface (GUI). In this section we will describe how to instruct *PGDBG* through a command-line interface. We introduce the GUI in *Section 1.12 PGDBG GRAPHICAL USER INTERFACE*.

In order to instruct *PGDBG* through its command-line interface, you will need to learn the *PGDBG Command Language.* Commands are entered in the command-line interface one line at a time. Lines are delimited with a carriage return (invoked by the **Enter** key on most systems). After pressing **Enter**, *PGDBG* will process the line. Each line must begin with the name of a command and its arguments, if any. The command language is composed of commands, constants, symbols, locations, expressions, and statements.

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

There are two command modes: *pgi* and *dbx*. The *pgi* command mode maintains the original *PGDBG* command interface. In *dbx* mode, the debugger uses commands with a syntax compatible with the familiar *dbx* debugger. Both command sets are available in both command modes, however some commands have a slightly different syntax depending on the mode. The *pgienv* command allows you to change modes while running the debugger.

1.4.1 Constants

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

1.4.2 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. Symbol names are case-sensitive and must match the name as it appears in the object file.

The compilers add an underscore character, "_", to the beginning of each external identifier. On UNIX systems, the PGI Fortran compilers also add an underscore to the end of each external identifier. Therefore, if *PGDBG* is unable to locate a symbol as entered, it prepends an underscore and tries again. If that fails, it adds an underscore to the end of the name and tries again. If that fails, the leading underscore is stripped and the search is repeated. For example, if `cfunc` and `ffunc` are *C* and Fortran routines, respectively, then the names for the symbols in the object file are `_cfunc` and `_ffunc_`. *PGDBG* will accept `cfunc`, and `_cfunc` as names for `_cfunc`, and will accept `ffunc`, `_ffunc`, and `_ffunc_` as names for `_ffunc_`. Note however, that due to case-sensitivity, `FFUNC`, `_FFUNC`, etc. are not accepted as names for `_ffunc_`.

1.4.3 Scope Rules

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

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

1.4.4 Register Symbols

In order to provide access to the system registers, *PGDBG* builds symbols for them. Register names generally begin with `$` to avoid conflicts with program identifiers. Each register symbol has a type associated with it, and registers are treated like global variables of that type, except that their address may not be taken. See *Section 1.10 Commands Summary* for a complete list of the register symbols.

1.4.5 Source Code Locations

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

Thus:

```
break 37
```

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

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

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

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

```
list 3:13
```

lists lines 3 through 13 of the current file, and

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

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

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

```
{line 37}
```

means "line 37",

```
{addr 0x1000}
```

means "address 0x1000" , and

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

means "the address associated with line 37" , and

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

means "the line associated with address `0x1000`".

1.4.6 Lexical Blocks

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

Below variable ***var*** 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***.

For Example:

```
lex.c:
main()
{
    int var = 0;
    {
        int var = 1;
        printf("var %d\n",var);
    }
    printf("var %d\n",var)
}
```

```
pgdbg> n
Stopped at 0x8048b10, function main, file
/home/pete/pgdbg/bugs/workon3/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
```

1.4.7 Statements

Although input is processed one line at a time, statement constructs allow multiple commands per line, and 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 by separating them with a semicolon. For example:

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

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

Statements can be parallelized across multiple threads of execution. See *Section 1.15.17 Parallel Statements* for details.

1.4.8 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.

The default action is to print the new value of the expression, and prompt the user for commands. By adding a location, or a condition, the event can be limited to a particular address or routine, or may occur only when the condition is true. The action to be taken when an event occurs can be defined by specifying a command list.

PGDBG supports four 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*, *track* and *do*. The command *break* takes an argument specifying a breakpoint location. Execution stops when that location is reached. The *watch* command takes an expression argument. Execution stops and the new value is printed when the value of the expression changes.

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

The optional arguments bring flexibility to the event definition. They are:

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

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

For example:

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

This event definition says that whenever execution is at line 37, and the value of `i` has changed since the last time execution was at line 37, and `y` is greater than 1, stop and print the new value of `i`.

```
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 that each time the routine `func1` is entered and `i` is equal to 37, then the value of `a[37]` should be printed, and a stack trace should be performed.

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

```
do {where}
```

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

```
track a.b
```

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

Events that occur at every line can be useful, but to perform them requires single-stepping the target program (this may slow execution considerably). Restricting an event to a particular address causes minimal impact on program execution speed, while restricting an event to a single routine causes execution to be slowed only when that routine is executed.

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

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

```
break 37 do {continue}

break 37 do {print i}
```

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

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

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

For example:

```
break 37
```

sets a breakpoint at line 37 in the current file.

```
track i
```

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

Scope qualifiers can also specify lines or variables that are not currently in scope. Events can be parallelized across multiple threads of execution. See *Section 1.15.16 Parallel Events* for details.

1.4.9 Expressions

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

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

```
break {pc}+8
```

invokes the pc command to compute the current address, adds 8 to it, and sets a breakpoint at that address. Similarly, the following command compares the start address of the current routine, with the start address of routine `xyz`, and prints the value 1, if they are equal and 0 otherwise.

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

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

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

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

prints elements 1 through 10 of the array `a,` and

```
list 5:10
```

lists source lines 5 through 10, and

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

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

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

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

An expression can be evaluated across many threads of execution by using a prefix p/t-set. See *Section 1.15.8 Current vs. Prefix P/t-set* for details.

Table 1-1: *PGDBG* Operators

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

1.5 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, ignore* commands).

If a thread runs into a busy loop or if the program runs into deadlock, control-C over the debugging command line to interrupt the threads. This causes SIGINT to be sent to all threads. By default *PGDBG* does not relay SIGINT to any of the threads, so in most cases program behavior is not affected.

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

Sending SIGINT (control-C) to a running MPI program is not recommended. See *Section 1.17.5 MPI Listener Processes* for details. Use the *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 is available while threads are running if *pgienv threadwait none* is set.

1.5.1 Signals Used Internally by *PGDBG*

SIGTRAP indicates a breakpoint has been hit. A message is displayed whenever a thread hits a breakpoint. SIGSTOP is used internally by *PGDBG*. Its use is mostly invisible to the user. Changing the disposition of these signals in *PGDBG* will result in undefined behavior.

Reserved Signals: On Linux, the thread library uses SIGRT1, SIGRT3 to communicate among threads internally. In the absence of real-time signals in the kernel, SIGUSR1, SIGUSR2 are used. Changing the disposition of these signals in *PGDBG* will result in undefined behavior.

1.6 Debugging Fortran

In order to create symbolic information for debugging, invoke your PGI Fortran compiler with the *–g* option. Fortran type declarations are printed using Fortran type names, not *C* type names. The only exception is Fortran character types, which are treated as arrays of *C* characters.

1.6.1 Arrays

Large arrays (e.g., arrays with an aggregate size greater than 2GB), arrays with lower dimensions, and adjustable arrays are all supported. Fortran array elements and ranges should be accessed using parentheses, rather than square brackets.

1.6.2 Operators

Only those operators that exist in the *C* language may be used in expressions. In particular .eq., .ne., and so forth are not supported. The analogous *C* operators ==, !=, etc. must be used instead. Note that the precedence of operators matches the *C* language, which may in some cases be different than for Fortran.

1.6.3 Name of Main Routine

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

```
break MAIN
```

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

1.6.4 Fortran Common Blocks

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

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

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

then the entire common block can be printed out using,

```
print xyz
```

and the individual elements can be accessed by name as in,

```
print a, b
```

1.6.5 Nested Subroutines

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

For example:

```
subroutine subtest (ndim)
   integer(4), intent(in) :: ndim
   integer, dimension(ndim) :: ijk
   call subsubtest ()
   contains
```

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

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

1.6.6 Fortran 90 Modules

To access a member ***mm*** of a Fortran 90 module ***M*** you must qualify ***mm***

with ***M*** using the scoping operator **@**. If the current scope is ***M*** the qualification can be omitted.

For example:

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

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

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

1.7 Debugging *C++*

In order to create symbolic information for debugging, invoke your PGI *C++* compiler with the *–g* option.

Calling C++ Instance Methods

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

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

main(){
     Person * pierre;
     pierre =  new Person("Pierre");
     pierre.print();
    }
```

To call the instance method `print` on object `pierre`, use the following syntax:

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

Notice that `pierre` is explicitly passed into the method, and the class name must also be specified.

1.8 Core Files

PGDBG 5.2 does not currently support core file debugging.

1.9 *PGDBG* Commands

This section describes the *PGDBG* command set in detail. *Section 1.9 PGDBG Commands* presents a table of all the debugger commands, with a summary of their syntax.

1.9.1 Commands

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

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

indicates that the command *list* may be abbreviated to *lis*, and that it will accept either no argument, an integer count, a line range, a routine name, or a line and a count.

1.9.1.1 Process Control

The following commands, together with the breakpoints described in the next section, let you control the execution of the target program. *PGDBG* lets you easily group and control multiple threads and processes. See *Section 1.15.11 Process and Thread Control* for more details.

att[ach] <pid>[<exe>] | [<exe> <host>]

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

c[ont]

Continue execution from the current location. This command may also be used to begin execution of the program at the beginning of the session.

de[bug]

Print the name and arguments of the program being debugged.

det[ach]

Detach from the current running process.

halt

Halt the running process or thread.

n[ext] [*count*]

Stop after executing one source line in the current routine. This command steps over called routines. The *count* argument stops execution after executing *count* source lines. In a parallel region of code, *next* applies only to the currently active thread.

nexti [*count*]

Stop after executing one instruction in the current routine. This command steps over called routines. The *count* argument stops execution after executing *count* instructions. In a parallel region of code, *nexti* applies only to the currently active thread.

proc [*number*]

Set the current thread to number. When issued with no argument, proc lists the current program location of the current thread of the current process. See *Section 1.14.4 Process-Parallel Debugging* for how processes are numbered.

procs

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

q[uit]

Terminate the debugging session.

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

Works like *run* except if no *args* are specified, none are used.

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

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

s[tep]
s[tep] *count*
s[tep] up

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

stepi
stepi *count*
stepi up

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

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 stops immediately upon return to the calling routine. This means that the current location may not be the start of a source line because multiple routine calls may occur on a single source line, and you might want to stop after the first call. If you want to step out of the current routine and continue to the start of the next source line, simply follow *stepout* with *next*. In a parallel region of code, *stepout* applies only to the currently active thread.

sync
synci

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

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

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

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

1.9.1.2 Process-Thread Sets

The following commands deal with defining and managing process thread sets. See *Section 1.15.9 P/t-set Commands* for a general discussion of process-thread sets.

defset

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

focus

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

undefset

'undefine' a previously defined process/thread set. The set is removed from the list. The debugger-defined p/t-set `[all]` cannot be removed.

viewset

List the members of a process/thread set that currently exist as active threads.

whichsets

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

1.9.1.3 Events

The following commands deal with defining and managing events. See *Section 1.4.8 Events* for a general discussion of events, and the optional arguments.

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

If no argument is specified, print the current breakpoints. Otherwise, set a breakpoint at the indicated line or routine. If a routine is specified, and the routine was compiled for debugging, then the breakpoint is set at the start of the first statement in the routine, that is, 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* evaluates true. If *do* is specified with a *command* or several *commands* as an argument, the command or commands are executed when the breakpoint occurs.

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

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

More sophisticated examples include:

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

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

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

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

breaki
breaki *func* [if (*condition*)] [do {*commands*}]
breaki *addr* [if (*condition*)] [do {*commands*}]

Set a breakpoint at the indicated address or routine. If a routine is specified, the breakpoint is set at the first address of the routine. This means that when the program stops at this breakpoint the prologue code which sets up the stack frame will not yet have been executed, and hence, values of stack arguments will not 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 as in the *break* command. The next examples set breakpoints at address `0xf0400608`, line 37 in the current file, line 37 in file `xyz.c`, the first executable address of routine `main`, the current line, and the current address, respectively:

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

Similarly,

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

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

breaks

Display all the existing breakpoints.

catch
catch [*sig:sig*]
catch [*sig* [, *sig*...]]

With no arguments, print the list of signals being caught. With the **:** argument, catch the specified range of signals. With a list, trap signals with the specified number.

clear
clear all
clear *func*
clear *line*
clear addr {*addr*}

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

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

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

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

Disable the indicated 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 {*commands*} [if (*condition*)]
do {*commands*} at *line* [if (*condition*)]
do {*commands*} in *func* [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 func specifies the commands are executed at each line in the routine. The *if* option has the same meaning as in *watch*. If a condition is specified, the *do* commands are executed only when *condition* is true.

doi {*commands*} [if (*condition*)]
doi {*commands*} at *addr* [if (*condition*)]
doi {*commands*} in *func* [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 (*func*) is specified, then the commands are executed at each line in the routine. If neither is specified, then the commands are executed at each address in the program. The *if* option has the same meaning as in *do* above.

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

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

hwatch *addr* [if (*condition*)] [do {*commands*}]

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

If an *if* option is specified, the event will cause no action unless the expression is true. If a *do* option is specified, then the commands will be executed when the event occurs.

hwatchr[ead] *addr* [if (*condition*)] [do {*commands*}]

Define a hardware read watchpoint. This event is triggered by hardware when the byte at the given address 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 *hwatch*.

hwatchb[oth] *addr* [if (*condition*)] [do {*commands*}]

Define a hardware read/write watchpoint. This event is triggered by hardware when the byte at the given address 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 *hwatch*.

ignore
ignore[*sig:sig*]
ignore [*sig* [, *sig*...]]

With no arguments, print the list of signals being ignored. With the **:** argument, ignore the specified range of signals. With a list, ignore signals with the specified number.

stat[us]

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

stop *var*
stop at *line* [if (*condition*)][do {*commands*}]
stop in *func* [if (*condition*)][do {*commands*}]
stop if (*condition*)

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

stopi *var*
stopi at *address* [if (*condition*)][do {*commands*}]
stopi in *func* [if (*condition*)][do {*commands*}]
stopi if (*condition*)

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

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

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

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

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

trace *var* [if (*condition*)][do {*commands*}]
trace *func* [if (*condition*)][do {*commands*}]
trace at *line* [if (*condition*)][do {*commands*}]
trace in *func* [if (*condition*)][do {*commands*}]

Activate source line tracing when var changes. Activate source line tracing and trace when in routine func. With *at*, activate source line tracing to display the specified line each time it is executed. With **in**, activate source line tracing to display the specified each source line 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 *var* [if (*condition*)][do {*commands*}]
tracei *func* [if (*condition*)][do {*commands*}]
tracei at *addr* [if (*condition*)][do {*commands*}]
tracei in *func* [if (*condition*)][do {*commands*}]

Activate instruction tracing when var changes. Activate instruction tracing when in routine func. With *at*, activate tracing to display the specified line each time it is executed. With the *in* keyword, display instructions while in the specified routine. 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.

unb[reak] *line*
unb[reak] *func*
unb[reak] all

Remove a breakpoint from the statement line. Remove a breakpoint from the routine *func*. Remove all breakpoints.

unbreaki *addr*
unbreaki *func*
unbreaki all

Remove a breakpoint from the address *addr*. Remove a breakpoint from the routine *func*. Remove all breakpoints.

wa[tch] *expression*
wa[tch] *expression* [if (*condition*)][do {*commands*}]
wa[tch] *expression* at *line* [if (*condition*)][do {*commands*}]
wa[tch] *expression* in *func* [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 *func* is specified, the expression is evaluated at each line in the routine. If no location is specified, the expression will be evaluated at each line in the program. If a *condition* is specified, the expression is evaluated only when the condition is true. If commands are specified, they are executed whenever the expression is evaluated and the value changes.

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

***Note:** Using watchpoints indiscriminately can dramatically slow program execution.*

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

```
watch i at 40
```

will barely slow program execution at all, while

```
watch i
```

will slow execution considerably.

watchi *expression*
watchi *expression* [if(*condition*)][do {*commands*}]
watchi *expression* at *addr* [if(*condition*)][do {*commands*}]
watchi *expression* in *func* [if(*condition*)][do {*commands*}]

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

This command is useful if line number information is limited. It causes programs to execute more slowly than *watch*.

when do {*commands*} [if (*condition*)]
when at *line* do {*commands*} [if (*condition*)]
when in *func* do {*commands*} [if (*condition*)]

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

wheni do {*commands*} [if (*condition*)]
wheni at *addr* do {*commands*} [if (*condition*)]
wheni in *func* 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 *func* 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 *Section 1.15.16 Parallel Events* for details.

1.9.1.4 Program Locations

This section describes PGDBG program locations commands.

arri[ve]

Print location information and update GUI markers for the current location.

cd [*dir*]

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

dis[asm]
dis[asm] *count*
dis[asm] *lo*:*hi*
dis[asm] *func*
dis[asm] *addr, count*

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

edit
edit *filename*
edit *func*

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 *func*. This command uses the editor specified by the environment variable $EDITOR.

file [*filename*]

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

lines *routine*

Print the lines table for the specified routine.

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

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

pwd

Print the current working directory.

stack[trace] [*count*]

Print a stacktrace. For each live routine print the routine name, source file, line number, current address. 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.

stackd[ump] [*count*]

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

w[here] [*count*]

Print the address, routine, source file and line number for the current location. If *count* is specified, print a maximum of *count* live routines on the stack.

/
/ [*string*] /

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

?
?[*string*] ?

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

1.9.1.5 Printing and Setting Variables

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

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 character string.

Character string constants print out literally. For example:

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

The array sub-range operator **:** prints a range of an array. The following example prints elements 0 through 10 of the array `a`:

```
        print a[0:10]
```

printf "*format_string*", *expr*,...*expr*

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

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

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

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

A `print c` command will only print the first 512 (or *stringlen*) bytes. Normally, the printing occurs until a `NULL` is reached, but without some limit, the printing may never end.

asc[ii] *exp* [,...*exp*]

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

bin *exp* [,...*exp*]

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

dec *exp* [,...*exp*]

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

display
display *exp* [,...*exp*]

Without arguments, list the expressions set to display at breakpoints. With an argument or several arguments, print expression *exp* at every breakpoint. See the description for *undisplay*.

hex *exp* [,...*exp*]

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

oct *exp* [,...*exp*]

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

set *var=expression*

Set variable *var* to the value of *expression*.

str[ing] *exp* [,...*exp*]

For each expression, evaluate, treat the result as a character pointer, and fetch and print a null terminated string from that address. This command will fetch a maximum of 70 characters.

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

Remove all expressions being printed at breakpoints. With an argument or several arguments, remove the expression *exp* from the list of display expressions.

1.9.1.6 Symbols and Expressions

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

as[sign] *var* = *exp*

Assign the value of the expression *exp* to the specified variable *var*.

call *func* [(*exp*,...)]

Call the named routine. *C* argument passing conventions are used. Breakpoints encountered during execution of the routine are ignored. If a signal is caught during execution of the routine, execution will stop, but continued execution may produce unpredictable results. Fortran functions and subroutines can be called, but the argument values will be passed according to *C* conventions. *PGDBG* may not always be able to access the return value of a Fortran function if the return value is an array.

decl[aration] *name*

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

For example, given declarations:

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

The commands,

```
decl I
    decl iar
    decl val
    decl abc
```

will respectively print out as

```
int i

int iar[10]

struct abc val

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

entr[y]
entr[y] *func*

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.

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.

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.

siz[eof] *name*

Return the size, in bytes, of the variable type *name*.

type *expr*

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

```
type I
type iar
type val
type val.a
type val.abc->b[2]
```

produce the following output:

```
int

int [10]

struct abc

int

char
```

whatis
whatis *name*

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

1.9.1.7 Scope

The following commands deal with program scope. See *Section 1.4.3 Scope Rules* for a discussion of scope meaning and conventions.

decls
decls *func*
decls "*sourcefile*"
decls {global}

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

down [*number*]

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

en[ter]
en[ter] *func*
en[ter] "*sourcefile*"
en[ter] {global}

Set the search scope to be the indicated symbol, which may be a routine, source file or global. If no scope is specified, use the search scope. The default *enter* with no argument is *enter global*.

files

Return the list of the files that make up the object file.

glob[al]

Return a symbol representing global scope. This command can also be used with the scope operator @ to specify symbols at global scope.

names
names *func*
names "*sourcefile*"
names {global}

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

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

whereis *name*

Print all declarations for *name*.

which *name*

Print full scope qualification of symbol *name*.

1.9.1.8 Register Access

System registers can be accessed by name. See *Section 1.4.4 Register Symbols* for the complete set of registers. A few commands exist to access common registers.

fp

Return the current value of the frame pointer.

pc

Return the current program address.

regs [format]

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

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

ret[addr]

Return the current return address.

sp

Return the current value of the stack pointer.

1.9.1.9 Memory Access

The following commands display the contents of arbitrary memory locations.

cr[ead]*addr*

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

dr[ead]*addr*

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

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

This command dumps a region of memory 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 bits or long bits. 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 hex int.

`%ps`
pointer to string.

`%pps`
pointer to pointer to string.

`%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*.

fr[ead]*addr*

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

ir[ead] *addr*

Fetch and return a signed integer from the specified address.

lr[ead] *addr*

Fetch and return an address from the specified address.

mq[dump]

Dump message queue information for current process. Refer to *Section 1.17.3 MPI Message Queues* for more information on *mqdump*.

sr[ead]*addr*

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

1.9.1.10 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.

ad[dr]
ad[dr] *n*
ad[dr] *line*
ad[dr] *func*
ad[dr] *var*
ad[dr] *arg*

Create an address conversion under these conditions:

- If an integer is given return an address with the same value.
- If a line is given, return the address corresponding to the start of that line.
- If a routine is given, return the first address of the routine.
- If a variable or argument is given, return the address where that variable or argument is stored.

For example:

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

func[tion]
func[tion] *addr*
func[tion] *line*

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

lin[e]
lin[e] *n*
lin[e] *func*
lin[e] *addr*

Create a source line conversion. If no argument is given, return the current source line. If an integer *n* is given, return it as a line number. If a routine *func* 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}
```

1.9.1.11 Miscellaneous

The following commands make using the debugger easier.

al[ias]
al[ias] *name*
al[ias] *name string*

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

For example:

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

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

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

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 exists so that you can debug programs even when some or all of the program source files are in a directory other than your current directory. For example:

```
dir morestuff
```

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

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

help [*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 [*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

Print the name of the language of the current file.

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.

nop[rint] *exp*

Evaluate the expression but do not print the result.

pgienv [*command*]

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

help *pgienv*	Provide help on pgienv
[*pgi*]env	Define the debugger environment
pgienv	Display the debugger settings
pgienv dbx on	Set the debugger to use *dbx* style commands
pgienv dbx off	Set the debugger to use *pgi* style commands
pgienv history on	Display the `history' record number with prompt
pgienv history off	Do NOT display the `history' number with prompt
pgienv exe none	Ignore executable's symbolic debug information
pgienv exe symtab	Digest executable's native symbol table (typeless)
pgienv exe demand	Digest executable's symbolic debug information incrementally on command
pgienv exe force	Digest executable's symbolic debug information when executable is loaded
pgienv solibs none	Ignore symbolic debug information from shared libraries
pgienv solibs symtab	Digest native symbol table (typeless) from each shared library
pgienv solibs demand	Digest symbolic debug information from shared libraries incrementally on demand
pgienv solibs force	Digest symbolic debug information from each shared library at load time
pgienv mode serial	Single thread of execution (implicit use of p/t-sets)
pgienv mode thread	Debug multiple threads (condensed p/t-set syntax)
pgienv mode process	Debug multiple processes (condensed p/t-set syntax)
pgienv mode multilevel	Debug multiple processes and multiple threads
pgienv omp [on|off]	Enable/Disable the *PGDBG* OpenMP event handler. This option is disabled by default. The *PGDBG* OpenMP event handler, when enabled, sets breakpoints at the beginning and end of each parallel region. Breakpoints are also set at each thread synchronization point. The handler coordinates threads across parallel constructs to maintain source level debugging. This option, when enabled, may significantly slow down program performance. Enabling this option is recommended for localized debugging of a particular parallel region only.
pgienv prompt <*name*>	Set the command line prompt to <name>
pgienv promptlen <*num*>	Set maximum size of p/t-set portion of prompt
pgienv speed <*secs*>	Set the time in seconds <secs> between trace points
pgienv stringlen <*num*>	Set the maximum # of chars printed for `char *'s

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

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).

scr[ipt] *filename*

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

setenv *name*
setenv *name value*

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

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.

sle[ep] [*time*]

Pause for *time* seconds or one second if no time is specified.

sou[rce] *filename*

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

unal[ias] *name*

Remove the alias definition for name, if one exists.

use [*dir*]

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

1.10 Commands Summary

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

Table 1-2: *PGDBG* Commands

Name	Arguments	Section
arri[ve]		1.9.1.4 Program Locations
att[ach]	<pid> [<exe>] | [<exe> <host>]	1.9.1.1 Process Control
ad[dr]	[*n* | *line* | *func* | *var* | *arg*]	1.9.1.10 Conversions
al[ias]	[*name* [*string*]]	1.9.1.11 Miscellaneous
asc[ii]	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
as[sign]	*var=exp*	1.9.1.6 Symbols and Expressions
bin	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
b[reak]	[*line* | *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
breaki	[*addr* | *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
breaks		1.9.1.3 Events
call	*func* [(*exp*,...)]	1.9.1.6 Symbols and Expressions
catch	[*number* [,*number*...]]	1.9.1.3 Events

Name	Arguments	Section
cd	[*dir*]	1.9.1.4 Program Locations
clear	[all | *func* | *line* | addr {*addr*}]	1.9.1.3 Events
c[ont]		1.9.1.1 Process Control
cr[ead]	*addr*	1.9.1.9 Memory Access
de[bug]		1.9.1.1 Process Control
dec	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
decl[aration]	*name*	1.9.1.6 Symbols and Expressions
decls	[*func* | "*sourcefile*" | {global}]	1.9.1.7 Scope
del[ete]	*event-number* | all | 0 | *event-number* [,.*event-number.*]	1.9.1.3 Events
det[ach		1.9.1.1 Process Control
dir[ectory]	[*pathname*]	1.9.1.11 Miscellaneous
dis[asm]	[*count* | *lo*:*hi* | *func* | *addr*, *count*]	1.9.1.4 Program Locations
disab[le]	*event-number* | all	1.9.1.3 Events
display	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
do	{*commands*} [at *line* | in *func*] [if (*condition*)]	1.9.1.3 Events
doi	{*commands*} [at *addr* | in *func*] [if (*condition*)]	1.9.1.3 Events
down		1.9.1.7 Scope

Name	Arguments	Section
defset	*name* [p/t-set]	1.9.1.2 Process-Thread Sets
dr[ead]	*addr*	1.9.1.9 Memory Access
du[mp]	*address*, *count*, "format-string"	1.9.1.9 Memory Access
edit	[*filename* | *func*]	1.9.1.4 Program Locations
enab[le]	*event-number* | all	1.9.1.3 Events
en[ter]	*func* | "*sourcefile*" | {global}	1.9.1.7 Scope
entr[y]	*func*	1.9.1.6 Symbols and Expressions
fil[e]		1.9.1.4 Program Locations
files		1.9.1.7 Scope
focus	[p/t-set]	1.9.1.2 Process-Thread Sets
fp		1.9.1.8 Register Access
fr[ead]	*addr*	1.9.1.9 Memory Access
func[tion]	[*addr* | *line*]	1.9.1.10 Conversions
glob[al]		1.15.10.3 Global Commands
halt	[*command*]	1.9.1.1 Process Control
he[lp]		1.9.1.11 Miscellaneous
hex	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
hi[story]	[*num*]	1.9.1.11 Miscellaneous

Name	Arguments	Section
hwatch	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
hwatchb[oth]	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
hwatchr[ead]	*addr* [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
ignore	[*number* [,*number...*]]	1.9.1.3 Events
ir[ead]	*addr*	1.9.1.9 Memory Access
language		1.9.1.11 Miscellaneous
lin[e]	[*n* | *func* | *addr*]	1.9.1.10 Conversions
lines	*routine*	1.9.1.4 Program Locations
lis[t]	[*count* | *line,count* | *lo*:*hi* | *routine*]	1.9.1.4 Program Locations
log	*filename*	1.9.1.11 Miscellaneous
lv[al]	*exp*	1.9.1.6 Symbols and Expressions
mq[dump]		1.9.1.9 Memory Access
names	[*func* | "*sourcefile*" | {global}]	1.9.1.7 Scope
n[ext]	[*count*]	1.9.1.1 Process Control
nexti	[*count*]	1.9.1.1 Process Control
nop[rint]	*exp*	1.9.1.11 Miscellaneous
oct	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
pc		1.9.1.8 Register Access
pgienv	[*command*]	1.9.1.11 Miscellaneous

Name	Arguments	Section
p[rint]	*exp1* [,...*expn*]	1.9.1.5 Printing and Setting Variables
printf	"*format_string*", *expr,...expr*	1.9.1.5 Printing and Setting Variables
proc	[*number*]	1.9.1.1 Process Control
procs		1.9.1.1 Process Control
pwd		1.9.1.4 Program Locations
q[uit]		1.9.1.1 Process Control
regs		1.9.1.8 Register Access
rep[eat]	[*first*, *last*] | [*first*: *last*:*n*] | [*num*] | [-*num*]	1.9.1.11 Miscellaneous
rer[un]	[*arg0 arg1* ... *argn*] [< *inputfile*] [> *outputfile*]	1.9.1.1 Process Control
ret[addr]		1.9.1.8 Register Access
ru[n]	[*arg0 arg1* ... *argn*] [< *inputfile*] [> *outputfile*]	1.9.1.1 Process Control
rv[al]	*expr*	1.9.1.6 Symbols and Expressions
sco[pe]		1.9.1.7 Scope
scr[ipt]	*filename*	1.9.1.11 Miscellaneous
set	*var* = *ep*	1.9.1.6 Symbols and Expressions
setenv	*name* | *name value*	1.9.1.11 Miscellaneous
sh[ell]	*arg0* [... *argn*]	1.9.1.11 Miscellaneous

Name	Arguments	Section
siz[eof]	*name*	1.9.1.6 Symbols and Expressions
sle[ep]	*time*	1.9.1.11 Miscellaneous
source	*filename*	1.9.1.11 Miscellaneous
sp		1.9.1.8 Register Access
sr[ead]	*addr*	1.9.1.9 Memory Access
stackd[ump]	[*count*]	1.9.1.4 Program Locations
stack[trace]	[*count*]	1.9.1.4 Program Locations
stat[us]		1.9.1.3 Events
s[tep]	[*count*] [up]	1.9.1.1 Process Control
stepi	[*count*] [up]	1.9.1.1 Process Control
stepo[ut]		1.9.1.1 Process Control
stop	[at *line* | in *func*] [*var*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
stopi	[at *addr* | in *func*] [*var*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
sync	[*func* | *line*]	1.9.1.1 Process Control
synci	[*func* | *addr*]	1.9.1.1 Process Control
str[ing]	*exp* [,...*exp*]	1.9.1.5 Printing and Setting Variables
thread	*number*	1.9.1.1 Process Control
threads		1.9.1.1 Process Control

Name	Arguments	Section
track	*expression* [at *line* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
tracki	*expression* [at *addr* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
trace	[at *line* | in *func*] [*var*| *func*] [if (*condition*)] do {*commands*}	1.9.1.3 Events
tracei	[at *addr* | in *func*] [*var*] [if (*condition*)] do {*commands*}	1.9.1.3 Events
type	*expr*	1.9.1.6 Symbols and Expressions
unal[ias]	*name*	1.9.1.11 Miscellaneous
undefset	[*name* | *-all*]	1.9.1.2 Process-Thread Sets
undisplay	[all | 0 | *exp*]	1.9.1.5 Printing and Setting Variables
unb[reak]	*line* | *func* | all	1.9.1.3 Events
unbreaki	*addr* | *func* | all	1.9.1.3 Events
up		1.9.1.7 Scope
use	[*dir*]	1.9.1.11 Miscellaneous
viewset	*name*	1.9.1.2 Process-Thread Sets
wait	[any | all | none]	1.9.1.1 Process Control
wa[tch]	*expression* [at *line* | in *func*] [if (*condition*)] [do {*commands*}]	1.9.1.3 Events
watchi	*expression* [at *addr* | in *func*] [if(*condition*)] [do {*commands*}]	1.9.1.3 Events

Name	Arguments	Section
whatis	[*name*]	1.9.1.6 Symbols and Expressions
when	[at *line* | in *func*] [if (*condition*)] do {*commands*}	1.9.1.3 Events
wheni	[at *addr* | in *func*] [if(*condition*)] do {*commands*}	1.9.1.3 Events
w[here]	[*count*]	1.9.1.4 Program Locations
whereis	*name*	1.9.1.7 Scope
whichsets	[p/t-set]	1.9.1.2 Process-Thread Sets
which	*name*	1.9.1.7 Scope
/	/ [*string*] /	1.9.1.4 Program Locations
?	**?**[*string*] **?**	1.9.1.4 Program Locations
!	History modification	1.9.1.11 Miscellaneous
^	History modification	1.9.1.11 Miscellaneous

1.11 Register Symbols

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

1.11.1 X86 Register Symbols

This section describes the X86 register symbols.

Table 1-3: General Registers

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

Table 1-4: x87 Floating-Point Stack Registers

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

Table 1-5: Segment Registers

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

Table 1-6: 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

1.11.2 AMD64 Register Symbols

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

Table 1-7: General Registers

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

Table 1-8: Floating-Point Registers

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

Table 1-9: Segment Registers

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

Table 1-10: Special Purpose Registers

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

Table 1-11: 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

1.11.3 SSE Register Symbols

On AMD64, 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 contains 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.

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

```
Bits:    127        96 95        65   63         32 31          0
```

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

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

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

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

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

You can also subscript SSE registers 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 linux86 environment, set the `PGDBG_SSE` environment variable to `` `on' `` to enable SSE support.

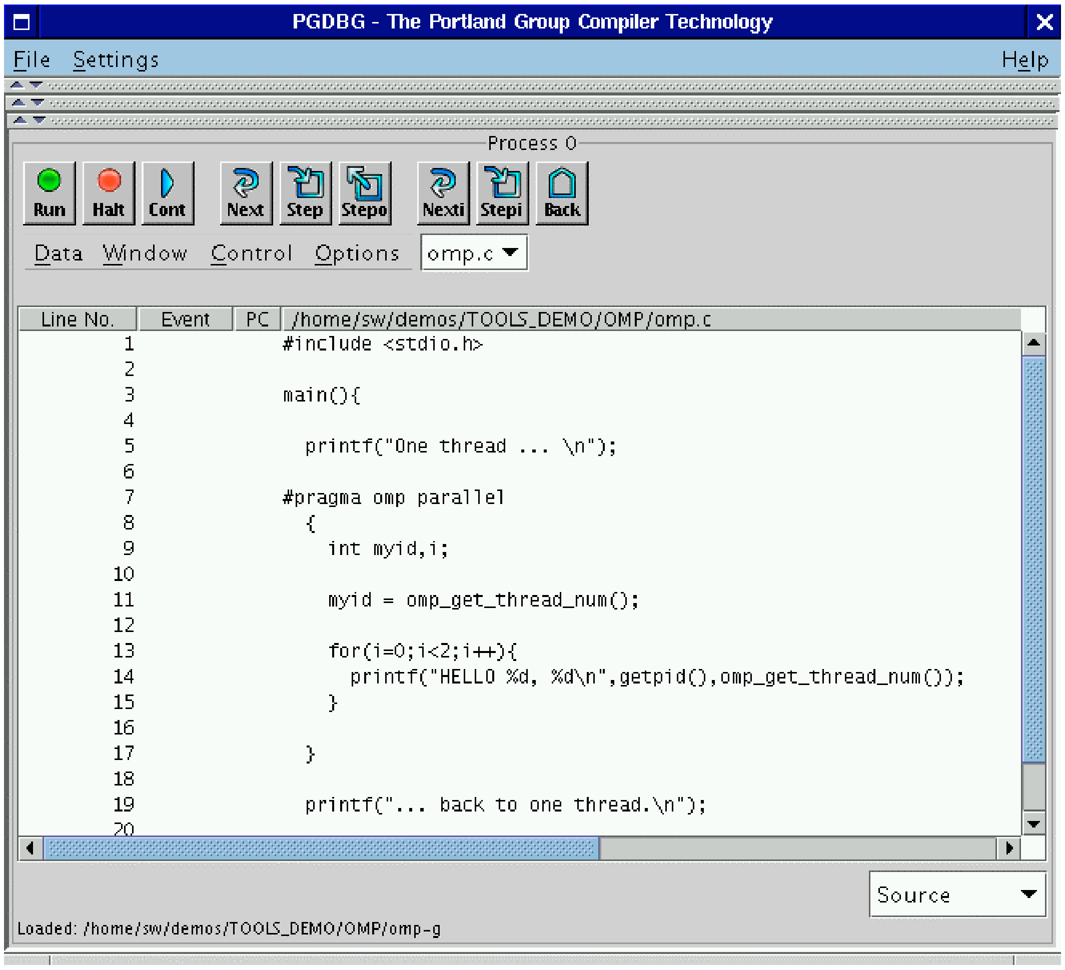
1.12 PGDBG Graphical User Interface

The *PGDBG* Graphical User Interface (GUI) is invoked on UNIX systems by default using the command `pgdbg`. The GUI runs as a separate process and communicates with `pgdbg`. There may be minor variations in the GUI from host to host, depending on the type of monitor available, the settings for various defaults and the window manager used. The basic interface across all systems remains the same with the exception of the differences tied to the display characteristics and the window manager used.

1.12.1 Main Window

Figure 1-1 shows the main window of *PGDBG* when you start the GUI 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 `800 x 600`. It can be resized in whatever fashion that your window manager supports. After resizing the main window, the GUI will remember the size that you choose the next time you bring up *PGDBG*. If you do not wish to save these settings when you exit *PGDBG,* then uncheck the *Save Settings on Exit* item under the *Settings* menu. We will discuss the *Settings* menu in *Section 1.12.1.5 Main Window Menus.*

Figure 1-1: Default Appearance of *PGDBG* GUI



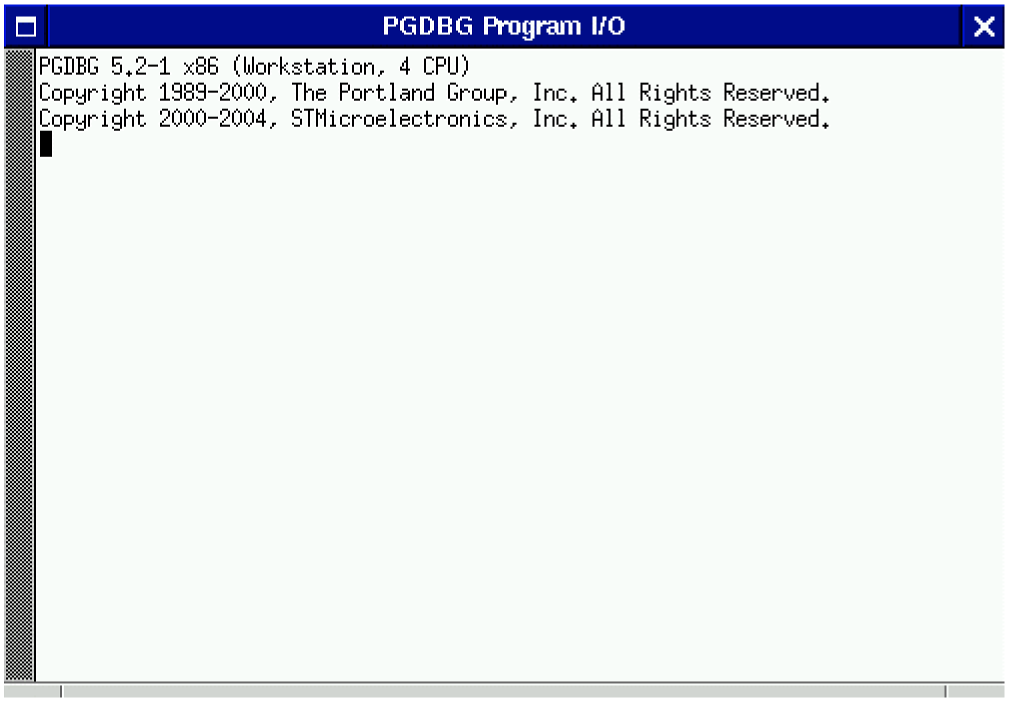
```
#include <stdio.h>

main(){

  printf("One thread ... \n");

#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();

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

  }

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

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

There are three horizontal divider bars (denoted with up and down arrow icons) at the top of the GUI in Figure 1-1. These dividers hide the following optional control panels: *Command Prompt, Focus Panel*, and the *Process/Thread Grid*. Figure 1-3 shows the main window with these controls visible. The GUI will remember which control panels are visible when you exit and will redisplay them when you reopen *PGDBG*. Below the dividers is the *Source Panel* described in *Section* 1.12.2 Source Panel.

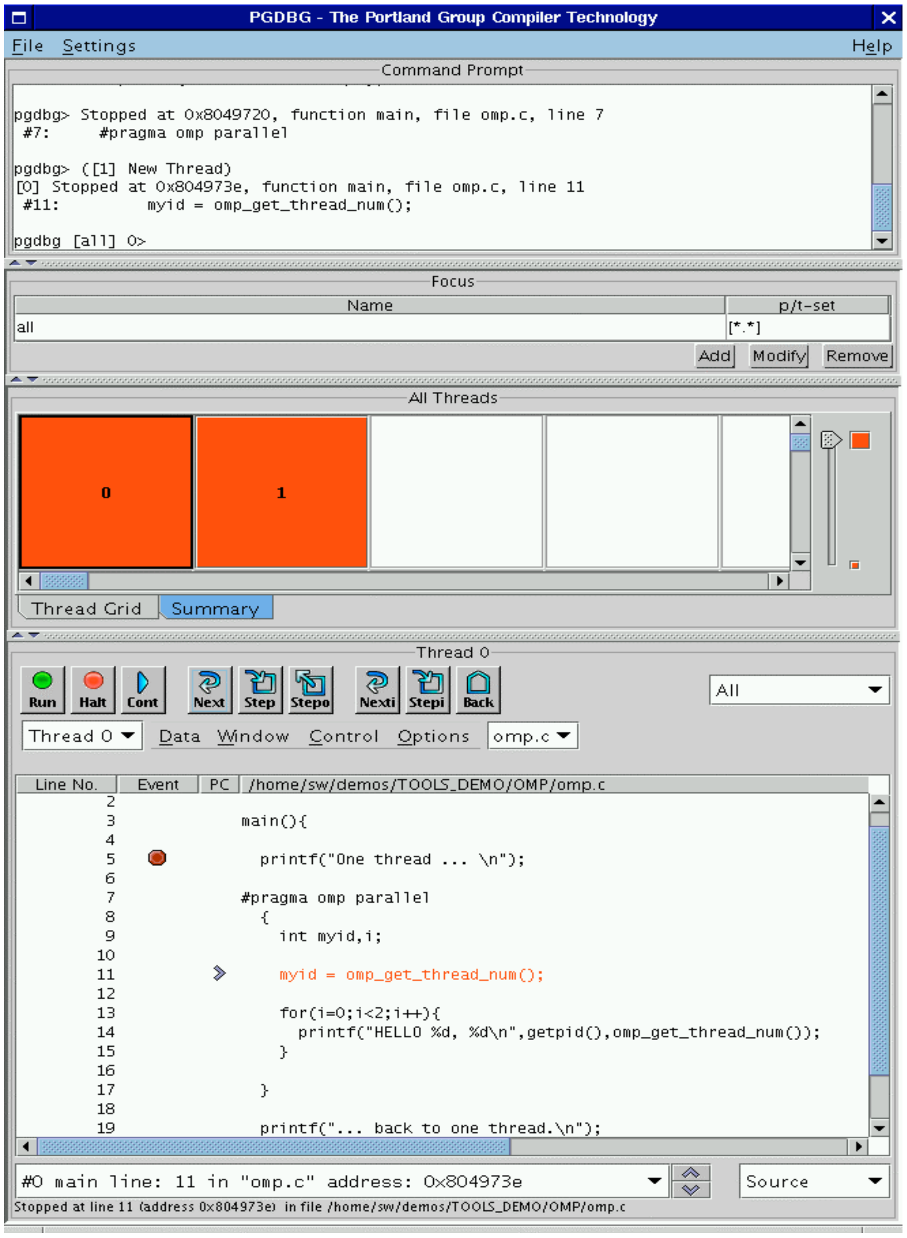
Besides the main window, a separate *Program I/*O window, similar to the one in Figure 1-2, is displayed when you first start up *PGDBG*. Any input or output performed by the debuggee is entered and/or displayed in this window.

Figure 1-2: *PGDBG* Program I/O Window



```
PGDBG 5.2-1 x86 (Workstation, 4 CPU)
Copyright 1989-2000, The Portland Group, Inc. All Rights Reserved.
Copyright 2000-2004, STMicroelectronics, Inc. All Rights Reserved.
```

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



```
pgdbg> Stopped at 0x8049720, function main, file omp.c, line 7
 #7:     #pragma omp parallel

pgdbg> ([1] New Thread)
[0] Stopped at 0x804973e, function main, file omp.c, line 11
 #11:          myid = omp_get_thread_num();

pgdbg [all] 0>
```

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

```
/home/sw/demos/TOOLS_DEMO/OMP/omp.c

main(){

  printf("One thread ... \n");

#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();

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

  }

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

```
#0 main line: 11 in "omp.c" address: 0x804973e
Stopped at line 11 (address 0x804973e) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c
```

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

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

1.12.1.1 Command Prompt

The *Command Prompt* supports a dialog with the debugger. Commands entered in this window are executed, and the results are displayed. See *Section 1.10 Commands Summary* for a list of commands that you can enter in the command prompt. The GUI also supports a "free floating" version of this window. To use the "free floating" command prompt, select the *Command Window* check box under the *Window* menu (*Section 1.12.2.1 Source Panel Menus*). If you are going to only use GUI controls, then you can keep this panel hidden.

1.12.1.2 Focus Panel

The *Focus Panel* is used to specify subsets of processes and/or threads known as *focus groups. Focus groups* allow you to apply debugger commands to a subset of threads and/or processes. *Focus Groups* are displayed in the table labeled *Focus* (Figure 1-3). In Figure 1-3, the *Focus* table contains one focus group called *All* that represents all processes/threads. We will revisit *focus groups* in *Section 1.15.6 P/t-set NotationProcess/Thread Sets.* In the meantime, just keep in mind that you can select a *focus group* by left mouse clicking 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. If you are debugging serial programs, then you can keep this panel hidden.

1.12.1.3 Process/Thread Grid

The *PGDBG* GUI lists all active processes/threads in the *Process/Thread Grid*. If you are debugging a multiprocess application, then this control is known as the *Process Grid.* If you are debugging a multithreaded (and single process) application, then the grid is known as a *Thread Grid.* Colors of each element in the grid represent a state. These colors and their meaning are defined in Tables 1-12 and 1-13 (located in S*ection 1.14.3 Graphical Features*).

For the *Process Grid*, each element is labeled with a numeric process identifier (see S*ection 1.15.3 Process-only debugging)* and represents a single process. Each element is a button that can be pushed to select a particular process as the *Current Process*. The *Current Process* is highlighted with a black border.

If you are debugging a multithreaded (e.g., OpenMP, etc.) program, then this control is called a *Thread Grid*. Each element in the thread grid is labeled with a numeric thread identifier (see S*ection 1.15.2 Threads-only debugging*). Similar to processes, clicking on an element in the thread grid selects that element as the *Current Thread.* The *Current Thread* is highlighted with a black border.

If you are debugging a multiprocess/multithreaded (hybrid) program, then selecting a process in the grid will reveal an inner thread grid as shown in Figure 1-4. In Figure 1-4, process 0 has four threads labeled 0.0, 0.1, 0.2, and 0.3; where the integer to the left of the decimal is the process identifier and the integer to the right of the decimal is the thread identifier. See *Section 1.15.4 Multilevel debugging* for more information on processes/thread identifiers.

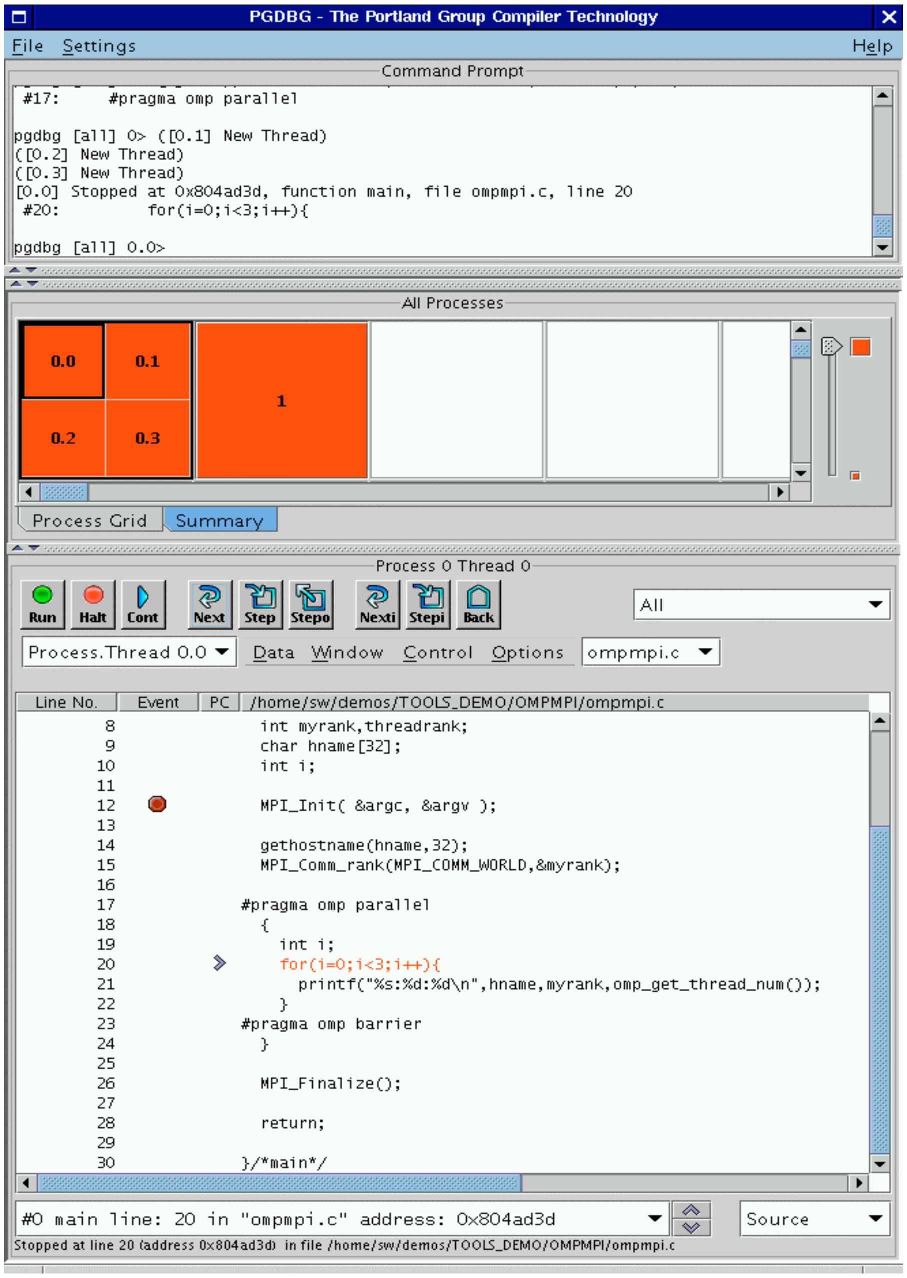
For a text dump of the *Process/Thread* grid, select the *Summary* tab under the grid. The text dump is essentially a graphical version of the *threads* debugger command (see *Section 1.9.1 CommandsProcess Control)*. When debugging a multiprocess or multithreaded application, the *Summary* panel will also include a *Context Selector* (as described in S*ection 1.12.3 Subwindows)*. Use the *Context Selector* to view a summary on a subset of processes/threads. By default, you will see a summary of all the processes/threads.

Use the slider to the right of the grid to zoom in and out of the grid. Currently, the grid supports up to 1024 elements. If you are debugging serial programs, then you can keep the *Process/Thread Grid* hidden.

1.12.1.4 Source Panel

The *Source Panel* displays the source code for the current location. The current location is marked by an arrow icon under the *PC* column. Breakpoints may be set at any source line by clicking the left mouse button under the *Event* column of the source line. The breakpoints are marked by stop sign icons. An existing breakpoint may be cleared by clicking the left mouse button on the stop sign icon. The source panel is described in greater detail in *Section 1.12.2 Source Panel.*

Figure 1-4: Process Grid with Inner Thread Grid



```
 #17:     #pragma omp parallel

pgdbg [all] 0> ([0.1] New Thread)
([0.2] New Thread)
([0.3] New Thread)
[0.0] Stopped at 0x804ad3d, function main, file ompmpi.c, line 20
 #20:         for(i=0;i<3;i++){

pgdbg [all] 0.0>
```

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

```
 8          int myrank,threadrank;
 9          char hname[32];
10          int i;
11
12          MPI_Init( &argc, &argv );
13
14          gethostname(hname,32);
15          MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
16
17        #pragma omp parallel
18          {
19            int i;
20            for(i=0;i<3;i++){
21              printf("%s:%d:%d\n",hname,myrank,omp_get_thread_num());
22            }
23        #pragma omp barrier
24          }
25
26          MPI_Finalize();
27
28          return;
29
30        }/*main*/
```

```
#0 main line: 20 in "ompmpi.c" address: 0x804ad3d
Stopped at line 20 (address 0x804ad3d) in file /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c
```

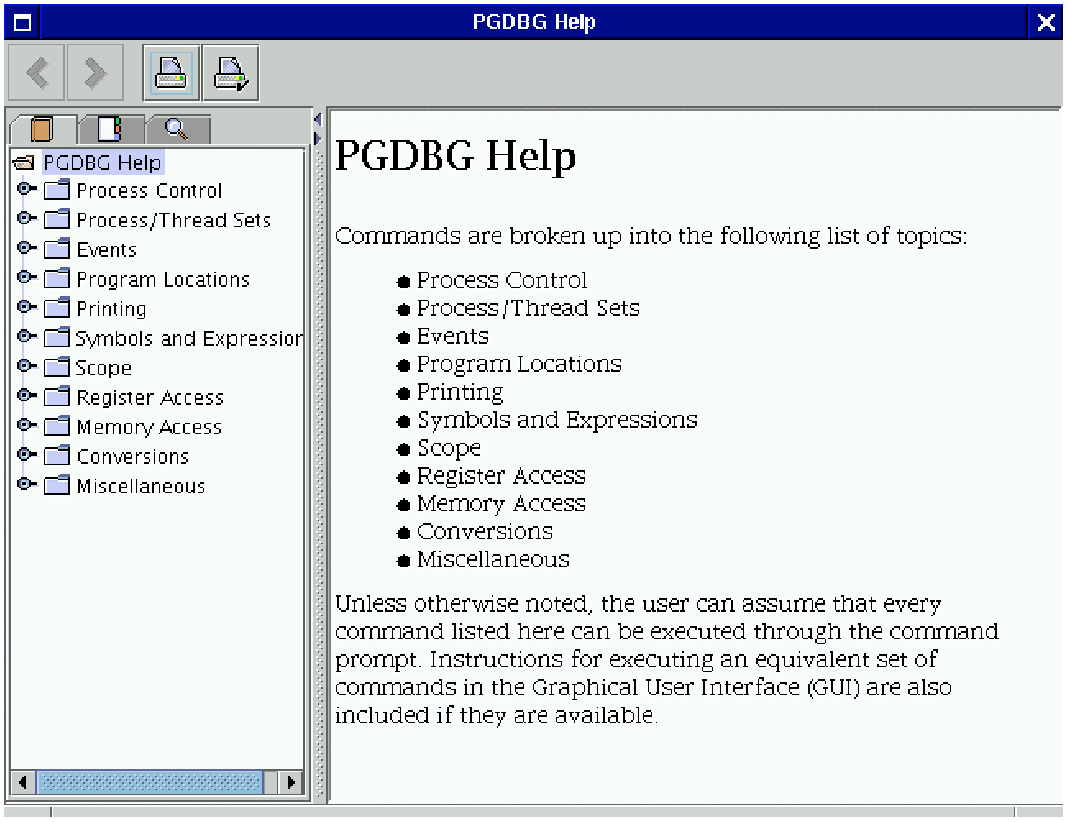
1.12.1.5 Main Window Menus

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

- **File Menu**
  - *Open Debugee…* - Select this option to begin a new debugging session. After selecting this option, select the program to debug (the *debugee*) from the file chooser dialog. The current debuggee is closed and replaced with the debugee that you selected from the file chooser. Press the *Cancel* button in the file chooser to abort the operation. Also see the *debug* command in *Section 1.9.1.1 Process Control.*
  - *Attach to Debugee…* - Select this option to attach to a running process. You can attach to a debugee running on a local or a remote host. See also the *attach* command in *Section 1.9.1.1 Process Control.*
  - *Detach Debugee* – Select this option to end the current debug session. See also the *detach* command in *Section 1.9.1.1 Process Control.*
  - *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 its size used in the *Command Prompt, Focus Panel,* and *Source Panel*. The default font is called *monospace* and the default size is 12.
  - *Show Tool Tips* – Select this check box to enable tool tips. Tool tips are small temporary messages that pop-up when you position the mouse pointer over a component in the GUI. They provide additional information on what a particular component does. Unselect this check box to turn them off.
  - *Restore Factory Settings* – Select this option to restore the GUI back to its initial state as shown in Figure 1-1.
  - *Restore Saved Settings* – Select this option to restore the GUI back to the state that it was in at the start of the debug session.
  - *Save Settings on Exit* – By default, the GUI will remember the state it was in when you exit. Unselect this check box if you do not want the GUI to remember state. You must unselect this option every time that you want the GUI to not remember state. When the GUI saves state, it stores the size of the main window, the location of the main window on your desktop, the location of each control panel divider, your tool tips preference, the font and size used. The GUI state is not shared across host machines.

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

Figure 1-5: *PGDBG* Help Utility



PGDBG Help

Commands are broken up into the following list of topics:

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

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

1.12.2 Source Panel

As mentioned in S*ection 1.12.1.4 Source Panel,* the source panel is located at the bottom of the GUI; below the *Command Prompt, Focus Panel,* and *Process/Thread Grid*. Use the source panel to control the debug session, step through source files and set breakpoints. To describe the source panel, we will divide each component into the following categories: *Menus, Buttons, Combo Boxes, Messages,* and *Events.*

1.12.2.1 Source Panel Menus

The source panel contains the following four menus: *Data, Window, Control, and Options.* A keystroke (e.g., Control P) enclosed in parentheses indicates a keyboard short cut for that menu option.

- **Data Menu** – The items under this menu are enabled when you select data in the source panel. Selecting and printing data in the source panel is explained in detail in *Section 1.12.1.4 Source Panel*. See also *Section 1.9.1.5 Printing and Setting Variables.*
    - *Print* – Print the value of the selected item. (Control P).
    - *Print* * - Dereference and print the value of the selected item.
    - *String* – Treat the selected value as a string and print its value.
    - *Bin* – Print the binary value of the selected item.
    - *Oct* – Print the octal value of the selected item.
    - *Hex* – Print the hex value of the selected item.
    - *Dec* – Print the decimal value of the selected item.
    - *Ascii* – Print the ASCII value of the selected item.
    - *Addr* – Print the address of the selected item.
- **Window Menu** – The items under this menu select various subwindows associated with the debugee. Subwindows are explained in greater detail in *Section 1.12.3 Subwindows.*
    - *Registers* – Display the registers subwindow. See also the *regs* command in *Section 1.9.1.8 Register Access.*
    - *Stack* – Display the stack subwindow. See also the *Stack* command in S*ection 1.9.1.4 Program Locations.*

  - *Locals* – Display a list of local variables that are currently in scope. See also the *names* command in S*ection 1.9.1.7 Scope.*
  - *Custom* – Bring up a custom subwindow.
  - *Disassembler* – Bring up the *PGDBG* Disassembler subwindow.
  - *Memory* – Bring up the memory dumper subwindow.
  - *Messages* – Display the MPI message queues. See *Section 1.17.3 MPI Message Queues* for more information on MPI message queues.
  - *Events* – Display a list of currently active break points, watch points, etc.
  - *Command Window* – When you select this menu item's check box, the GUI will display a "free floating" version of the command prompt window (*Section 1.12.1 Command Prompt*). See also *Section 1.10 Commands Summary* for a description of each command that you can enter in the command prompt.

- **Control Menu** – The items under this menu control the execution of the debugee. Many of the items under this menu have a corresponding button associated with them (see *Section 1.12.2.2 Source Panel* Buttons*).*
  - Arrive – Return the source pane to the current PC location. See also the *arrive* command in *Section 1.9.1.4 Program Locations* (Control A).
  - Up – Enter scope of routine up one level in the call stack. See also the *up* command in *Section 1.9.1.7 Scope* (Control U).
  - Down – Enter scope of routine down one level in the call stack. See also the *down* command in *Section 1.9.1.7 Scope* (Control D).
  - Run – Run or Rerun the Debugee. See also the *run* and *rerun* commands in *Section 1.9.1.1 Process Control* (Control R).
  - Run Arguments - Opens a dialog box that allows you to add or modify the debugee's runtime arguments.
  - Halt – Halt the running processes or threads. See also the *halt* command in *Section 1.9.1.1 Process Control* (Control H).
  - Call… - Open a dialog box to request a routine to call. See *Section 1.9.1.6 Symbols and Expressions* for more information on the *call* command.
  - Cont – Continue execution from the current location. See also the *cont* command in *Section 1.9.1.1 Process Control* (Control G).

- *Step* – Continue and stop after executing one source line. See also the *step* command in *Section 1.9.1.1 Process Control* (Control S).
- *Next* – This command is similar to *Step* except it steps over called routines. See also the *next* command in *Section 1.9.1.1 Process Control* (Control N).
- *Step Out* – Continue and stop after returning to the caller of the current routine. See also the *stepout* command in *Section 1.9.1.1 Process Control* (Control O).
- *Stepi* – Continue and stop after executing one machine instruction. See also the *stepi* command in *Section 1.9.1.1 Process Control* (Control I).
- *Nexti* – This command is similar to *Stepi* except it steps over called routines. See also the *nexti* command in *Section 1.9.1.1 Process Control* (Control T).

- ***Options Menu*** – This menu contains additional items that assist in the debug process.
  - *Search Forward…* - Select this option to perform a forward keyword search in the source panel (Control F).
  - *Search Backward…* **-** Select this option to perform a backward keyword search in the source panel (Control B).
  - *Search Again…* **-** Select this option to repeat the last keyword search that was performed on the source panel (Control E).
  - *Locate Routine…* - When you select this option, the GUI will ask you to enter the name of the routine that you wish to find. If *PGDBG* has debug information on that routine, it will display the routine in the source panel. See also S*ection 1.12.4 Selecting and Printing Data.*
  - *Disassemble* – Disassemble the data selected in the source panel. See also S*ection 1.12.4 Selecting and Printing Data*.
  - *Cascade Windows* – If you have one or more subwindows open, then you can use this option to automatically stack your subwindows in the upper left-hand corner of your desktop (Control W).
  - *Refresh* – Repaint the process/thread grid and source panels (Control L).

1.12.2.2 Source Panel Buttons

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

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

1.12.2.3 Source Panel Combo Boxes

Besides buttons and menus, the source panel also contains one or more combo boxes. A combo box is a combination text field and list component. In its *closed* or default state, it presents a text field of information with a small down arrow icon to its right. When the down arrow icon is left mouse clicked, the box *opens* and presents a list of choices for you to select.

The source panel, as shown in Figure 1-3, contains five combo boxes labeled *All, Thread 0, omp.c, #0 main line: 11 in "omp.c" address: 0x804973e,* and *Source*. These combo boxes are called the *Apply Selector*, *Context Selector, Source File Selector, Scope Selector,* and *Display Mode Selector* respectively. Below is a description of each combo box.

- Use the *Apply Selector* to select the set of processes and/or threads on which you wish to operate. Any command that you enter in the source panel will be applied to this set of processes/threads. These commands include setting break points, selecting items under the *Control* menu, pressing one of the nine buttons mentioned in *Section 1.12.2.2 Buttons*, etc. Depending on whether you are debugging a multithreaded, multiprocess, or multiprocess/multithreaded (hybrid) program the following possibilities are available:
    - *All* – All processes/threads receive commands entered in the source panel (default).

- o *Current Thread* – Commands are applied to the current thread ID only.
- o *Current Process* – Commands are applied to all threads that are associated with the current process.
- o *Current Process.Thread* – Commands are applied to the current thread on the current process only.
- o *Focus* – Commands are applied to the focus group selected in the *Focus Panel* (described in *Section 1.12.1 Main Window).* Refer to *Section 1.15.5 Process/Thread Sets* for more information on this advanced feature.

If you are debugging a serial program, then this combo box is not displayed.

Similar to the *Process/Thread* Grid, the *Context Selector* allows you to change the current *Process*, *Thread*, or *Process.Thread* ID that you are currently debugging. If you are debugging a serial program, then this combo box is not displayed.

The *Source File Selector* displays the current file being debugged and allows you to select and view a different file 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 its combo box and select a file from the list. If the source file is available, the source file will appear in the *Source Panel.*

The *Scope Selector* displays the current *scope* of the current Program Counter (PC) that you are debugging. You can open its combo box and select a different scope from the list or use the up and down buttons located on the right of the combo box. The *up* button is equivalent to the *up* debugger command and the *down* button is equivalent to the *down* debugger command. See *Section 1.9.1.7 Scope* for more information on the *up* and *down* commands.

The *Display Mode Selector* allows you to select three different source display modes: *Source, Disassembly,* and *Mixed.* The *Source* mode shows the source code of the current source file indicated by the *File Selector.* This is the default display mode if the source file is available. The *Disassembly* mode shows the machine instructions of the current routine that you are debugging. This is the default display mode if the source file is not available. The *Mixed* mode shows machine instructions annotated with source code. This mode is available only if the source file is available.

1.12.2.4 Source Panel Messages

The source panel contains two message areas. The top center indicates the current process/thread ID (e.g., *Thread 0* in Figure 1-6) and the bottom left displays status messages (e.g., *Stopped at line 11…* in Figure 1-6).

1.12.2.5 Source Panel Events

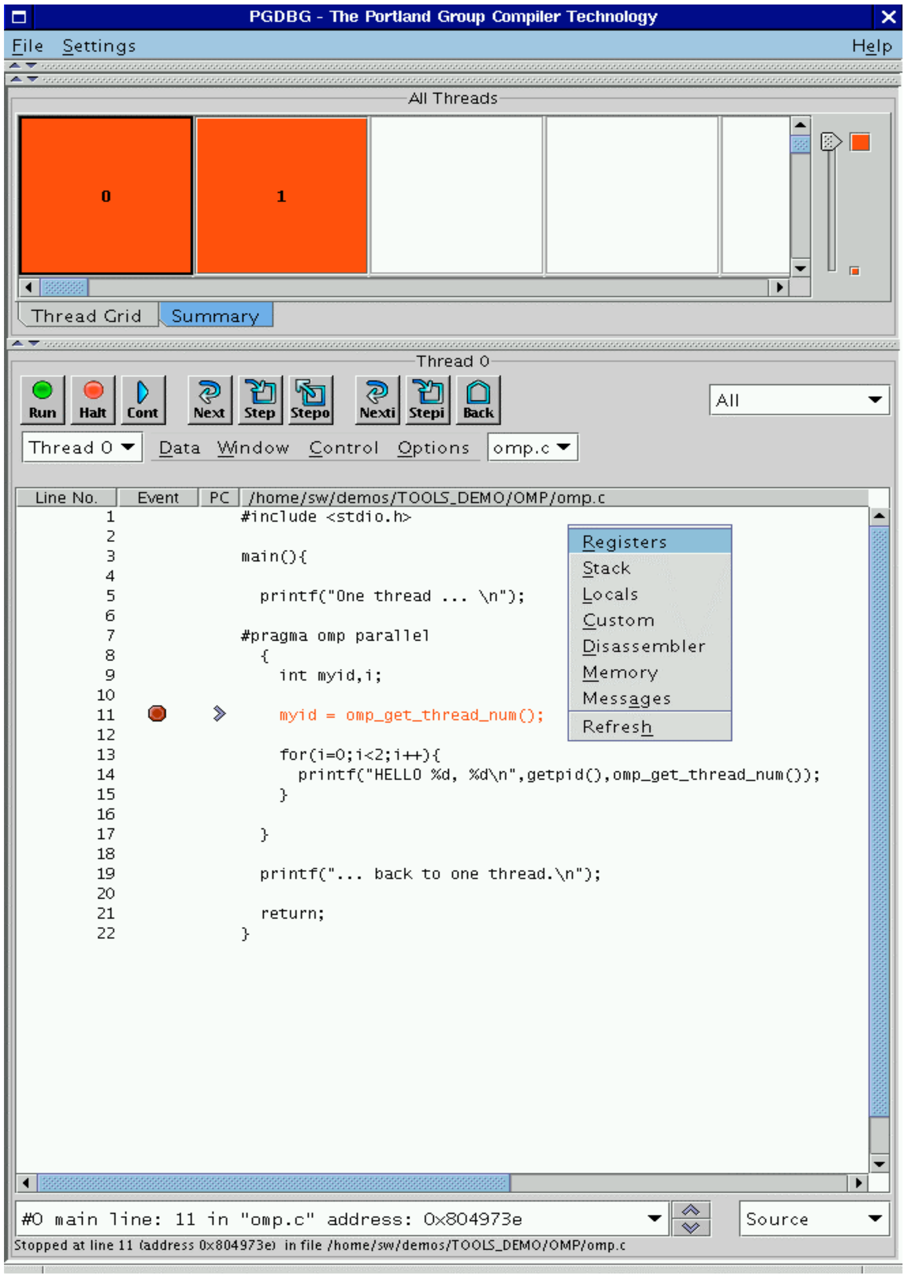
Events, such as breakpoints, are displayed under the *Event* column in the source panel. Currently, the source panel can only display breakpoint events. A stop sign icon denotes a breakpoint. Breakpoints are added through the source panel by left mouse clicking on the desired source line under the *Event* column. Left mouse click a stop sign to clear its breakpoint. Select the *Events* item under the *Window* menu to view a global list of *Events* (e.g., breakpoints, watch points, etc.).

1.12.3 Subwindows

A subwindow is defined as any *PGDBG* GUI component that is not embedded in the main window described in *Section 1.12.1 Main Window*. One example of a subwindow is the *Program I/O* window introduced in Figure 1-2. Other examples of subwindows can be found under the source panel's *Window* menu. These include the *Registers, Stack, Locals, Custom, Disassembler, Memory, Messages, Events,* and *Command Window* subwindows. With the exception of the *Command Window*, all of these subwindows have similar capabilities. Therefore, we will describe only a sampling of these subwindows here. See the description of the *Window* menu, S*ection 1.12.2.1 Source Panel Menus,* for more information on each subwindow.

Besides using the *Window* menu to bring up a subwindow, you can right mouse click on a blank spot in the source panel to bring up a pop-up menu (Figure 1-6). Use this pop-up menu to select a subwindow. The subwindow that gets displayed is specific to the current process and/or thread. For example, in Figure 1-6, selecting *Registers* will display the registers for thread 0, which is the current thread.

Figure 1-6: Opening a Subwindow with a Pop-up Menu



/home/sw/demos/TOOLS_DEMO/OMP/omp.c

```
#include <stdio.h>

main(){

  printf("One thread ... \n");

#pragma omp parallel
  {
    int myid,i;

    myid = omp_get_thread_num();

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

  }

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

  return;
}
```

#0 main line: 11 in "omp.c" address: 0x804973e

Stopped at line 11 (address 0x804973e) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c

1.12.3.1 Memory Subwindow

Figure 1-7 shows the memory subwindow. The memory subwindow displays a region of memory in a *printf*-like format descriptor. It is essentially a graphical interface to the debugger *dump* command (*Section 1.9.1.9 Memory Access)*.

This subwindow shows all of the possible controls that you can expect in a subwindow. Not all subwindows will have all of the components shown in this figure. However, nearly all will have the following components: *File* menu, *Options* menu, *Reset* button, *Close* Button, *Update* button, and the *Lock/Unlock* toggle button.

The *File* menu contains the following items:

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

The *Options* menu contains the following items:

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

The *Reset*, *Close,* and *Update* buttons are synonymous with their menu item counterparts mentioned above. The *Lock/Unlock* toggle button, on the other hand, has its own unique purpose. The *Lock/Unlock* button, located in the lower right hand corner of a subwindow, toggles between a lock and an unlock state. Figure 1-7 shows this button in an unlocked state. Note that the button is labeled *Lock.* Figure 1-8 shows this button in a locked state. Note that the button is labeled *Unlock.*

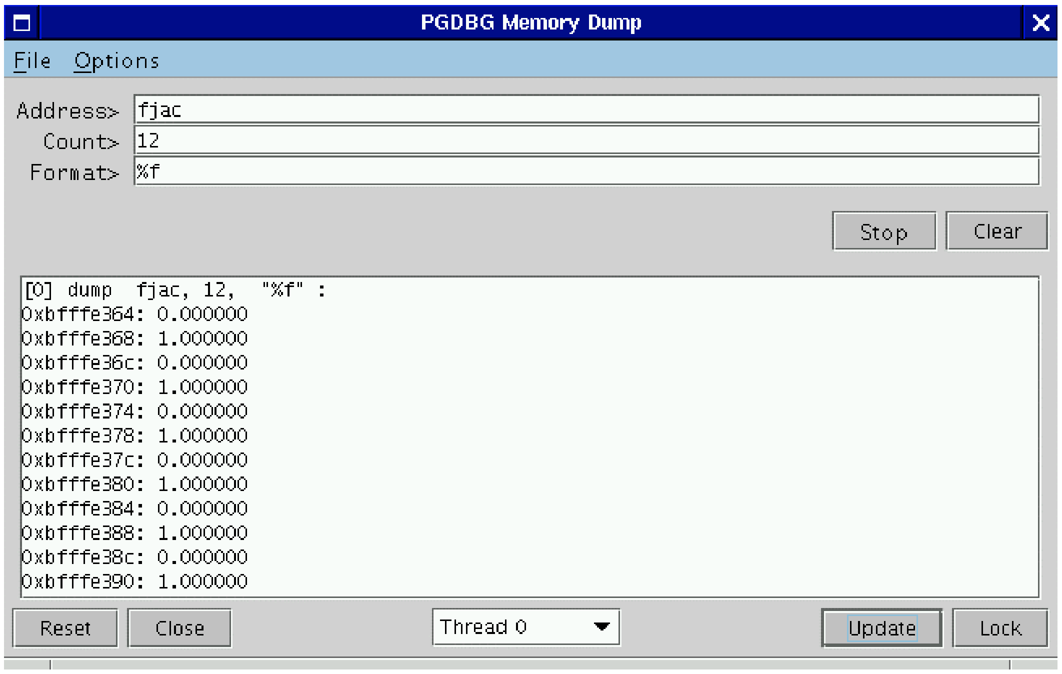
When the *Lock/Unlock* button is in its unlocked state, subwindows will update themselves whenever a process or thread halts. This can occur after a *step, next,* or *cont* command. If you want to preserve whatever is in a subwindow, left mouse click the *Lock* button to lock the display in the subwindow. Figure 1-8 shows an example of a locked subwindow. Note that some of the controls in Figure 1-8 are disabled (greyed out). After locking a subwindow, the GUI will disable any controls that affect the display until you unlock the subwindow. To unlock the subwindow, click the *Unlock* button. The toggle button will now say *Lock* and the GUI will re-enable the other controls.

Besides the previously mentioned memory subwindow capabilities, subwindows may also have one to three input fields. In the *Memory* subwindow, the user enters 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.

If the subwindow has one or more input fields, then there is also a *Stop* and *Clear* button. The *Stop* button is synonymous with the *Stop* menu item mentioned above. The C*lear* button erases the input field(s).

If you are debugging a program with more than one process and/or thread, you will also get a *Context Selector* in the bottom center as shown in Figure 1-7. With the *Context Selector* you can view data specific to a particular process/thread or a subset of process/threads when you select *Focus***.** Refer to *Section 1.15.5 Process/Thread Sets* for more information on *Focus*.

Figure 1-7: Memory Subwindow



```
Address> fjac
  Count> 12
 Format> %f
```

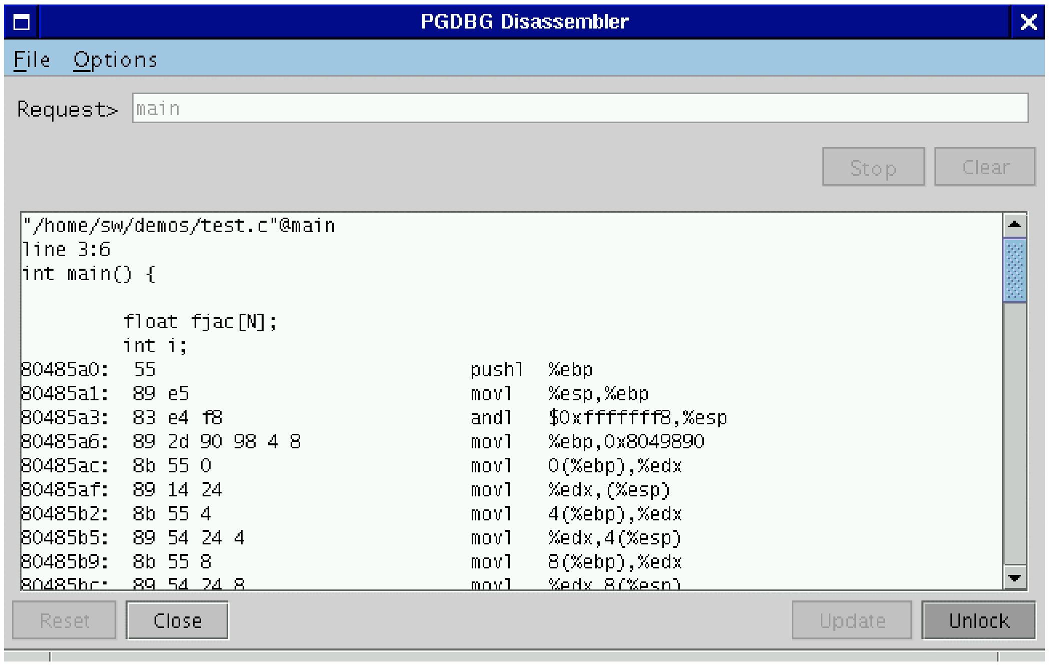
```
[0] dump  fjac, 12,  "%f" :
0xbfffe364: 0.000000
0xbfffe368: 1.000000
0xbfffe36c: 0.000000
0xbfffe370: 1.000000
0xbfffe374: 0.000000
0xbfffe378: 1.000000
0xbfffe37c: 0.000000
0xbfffe380: 1.000000
0xbfffe384: 0.000000
0xbfffe388: 1.000000
0xbfffe38c: 0.000000
0xbfffe390: 1.000000
```

1.12.3.2 Disassembler Subwindow

Figure 1-8 shows the *Disassembler* subwindow. Use this subwindow to disassemble a routine (or a text address) specified in the *Request*> input field. *PGDBG* will default to the current routine if you specify nothing in the *Request*> input field. After a request is made to the *Disassembler*, the GUI will ask if you want to "Display Disassembly in the Source window". Choosing "yes" causes the *Disassembler* window to disappear and the disassembly to appear in the source panel. By viewing the disassembly in the source panel, you can set breakpoints at the machine instruction level. Choosing "no" will dump the disassembly in the *Disassembler* subwindow as shown in Figure 1-8.

Sometimes you may not know the name of the routine you want to disassemble. You may only have a text address. Specifying just a text address in the *Request*> field will cause *PGDBG* to disassemble address locations until it runs out of memory or hits an invalid op code. This may cause very large machine language listings. For that case, the subwindow provides a *Stop* button. You can hit the *Stop* button to interrupt long listings that may occur with the *Disassembler.* You can also 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 locations following address 0xabcdef. The *Request*> field can take the same arguments as the *disasm* command described in S*ection 1.9.1.4 Program Locations*.

Figure 1-8: Disassembler Subwindow

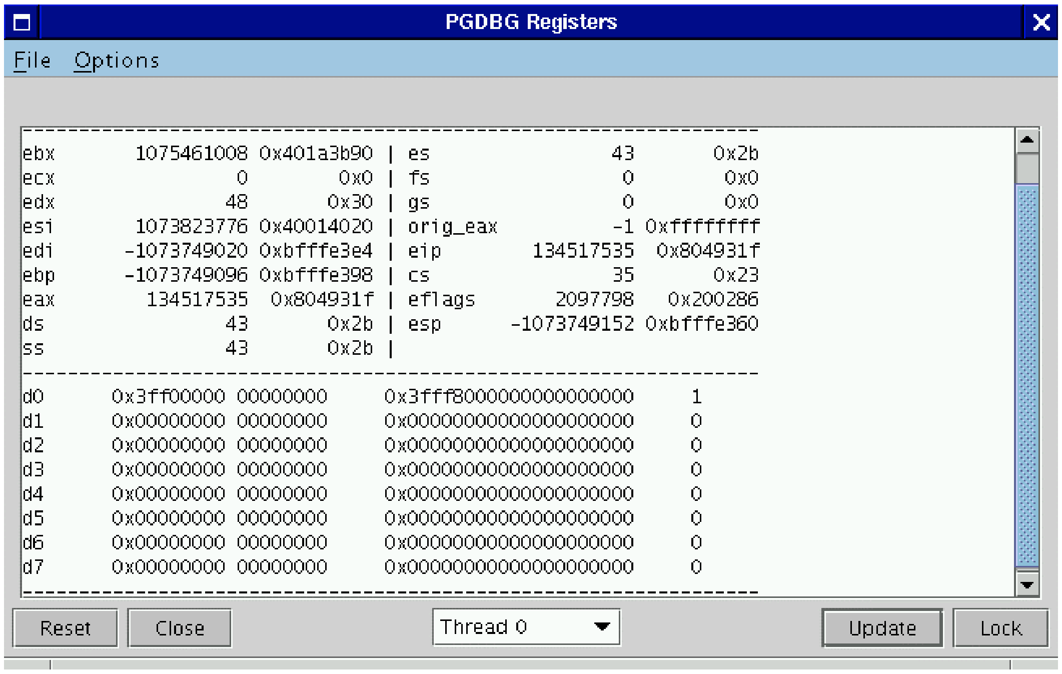


```
"/home/sw/demos/test.c"@main
line 3:6
int main() {

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

1.12.3.3 Registers Subwindow

Figure 1-9 illustrates the *Registers* subwindow. As mentioned earlier, you can view the registers on one or more processes and threads using the *Context Selector*. The *Registers* subwindow is essentially a graphical representation of the *regs* debugger command (*see Section 1.9.1.8 Register Access*).

Figure 1-9: Registers Subwindow

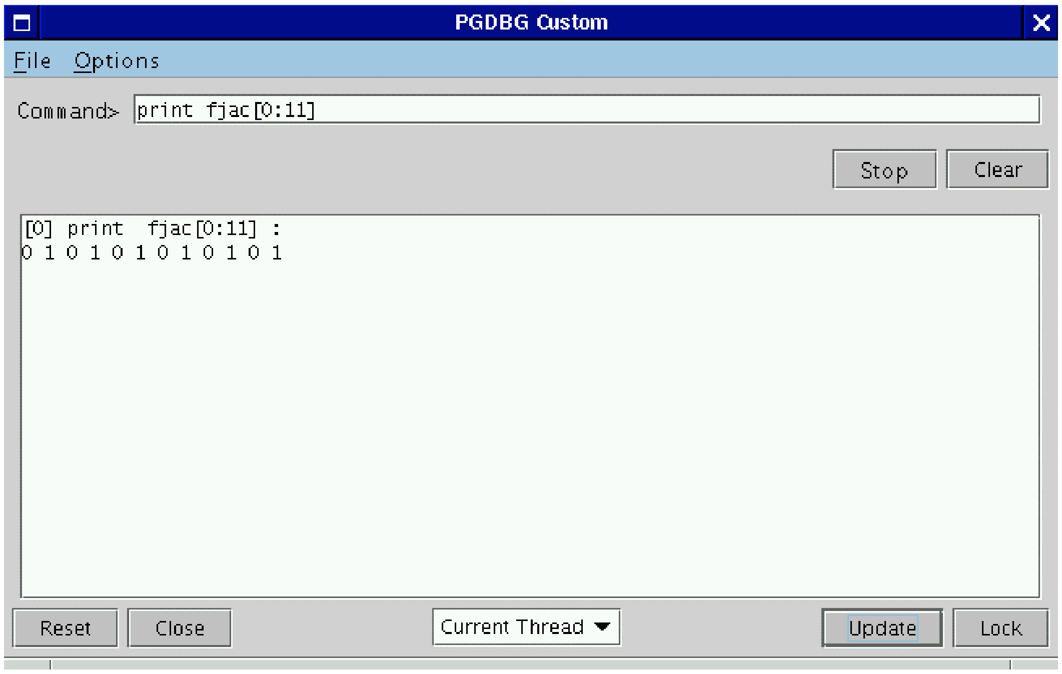


```
-------------------------------------------------------------------
ebx       1075461008 0x401a3b90 | es                 43       0x2b
ecx                0        0x0 | fs                  0        0x0
edx               48       0x30 | gs                  0        0x0
esi       1073823776 0x40014020 | orig_eax           -1 0xffffffff
edi      -1073749020 0xbfffe3e4 | eip        134517535  0x804931f
ebp      -1073749096 0xbfffe398 | cs                 35       0x23
eax        134517535  0x804931f | eflags        2097798   0x200286
ds                43       0x2b | esp      -1073749152 0xbfffe360
ss                43       0x2b |
-------------------------------------------------------------------
d0       0x3ff00000 00000000     0x3fff8000000000000000     1
d1       0x00000000 00000000     0x00000000000000000000     0
d2       0x00000000 00000000     0x00000000000000000000     0
d3       0x00000000 00000000     0x00000000000000000000     0
d4       0x00000000 00000000     0x00000000000000000000     0
d5       0x00000000 00000000     0x00000000000000000000     0
d6       0x00000000 00000000     0x00000000000000000000     0
d7       0x00000000 00000000     0x00000000000000000000     0
-------------------------------------------------------------------
```

1.12.3.4 Custom Subwindow

Figure 1-10 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 you press the *Update* button. The commands, entered in the control panel, can be any debugger command mentioned in S*ection 1.10 Commands Summary.*

Figure 1-10: Custom Subwindow



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

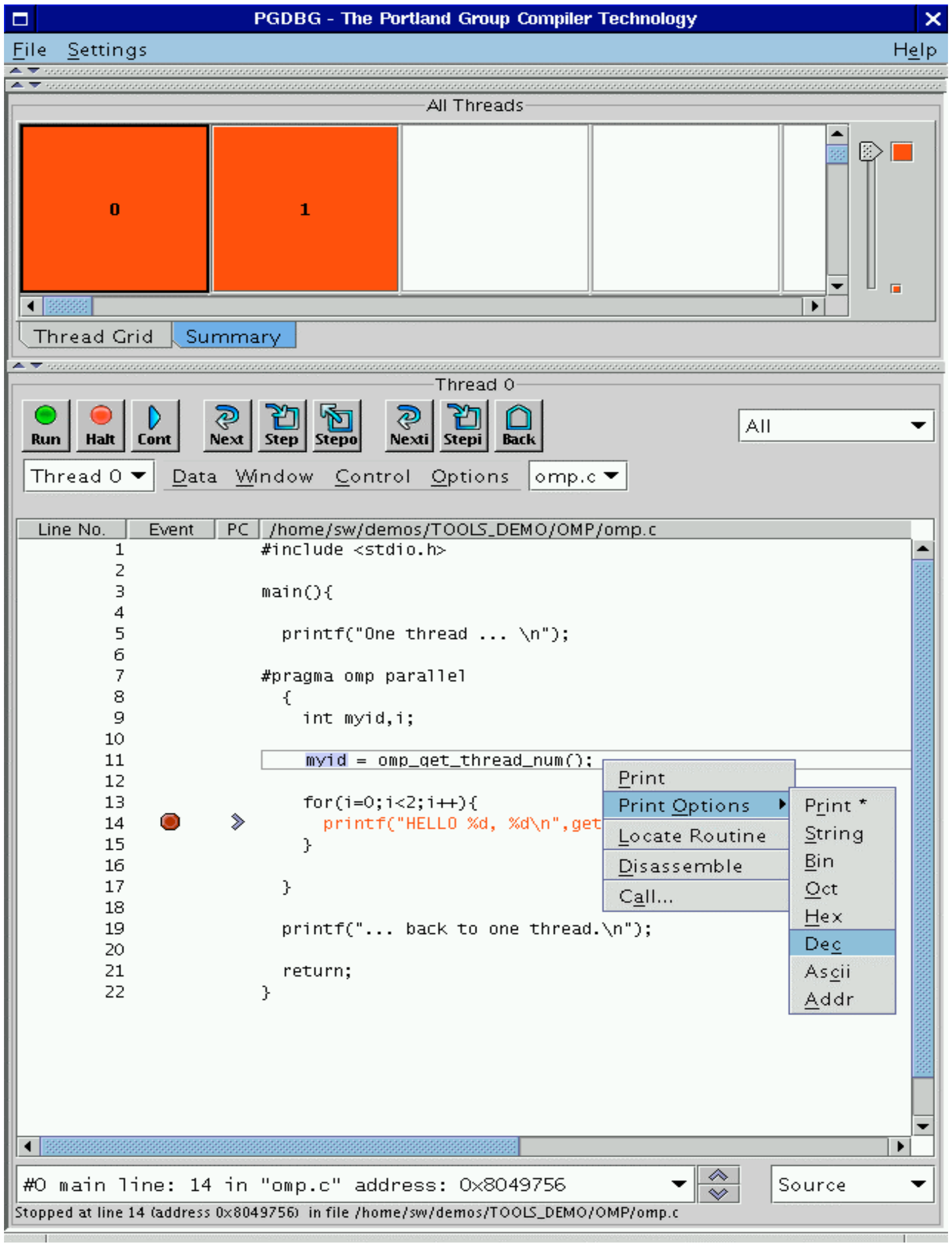
```
[0] print  fjac[0:11] :
0 1 0 1 0 1 0 1 0 1 0 1
```

1.12.4 Selecting and Printing Data

We introduced data printing in S*ection 1.12.2.1 Source Panel Menus* when we described the *Data* menu. To print data, left mouse click the text in the source panel that you wish to print. A text field displays in the source panel around the text on which you clicked. While holding down the left mouse button, drag the mouse pointer over the text that you want to print. The GUI will highlight this text. When you are done highlighting, release the left mouse button and select the desired print option from the *Data* menu or press the right mouse button for a pop-up menu. The pop-up menu is demonstrated in Figure 1-11.

Besides printing, you can also select *Disassemble, Call,* and *Locate* from the *Options* or pop-up menu*.* All three of these items assume that your selected text is a routine. The *Disassemble* item will open a disassembler subwindow with your selected routine. The *Call* item can be used to manually call the selected routine. The *Locate* option will take you to the location in the source code where the selected routine is defined. Please see the description for each of these items under the *Options* menu, S*ection 1.12.2.1 Source Panel Menus*, for more information.

Figure 1-11: Data Pop-up Menu



/home/sw/demos/TOOLS_DEMO/OMP/omp.c

```
 1   #include <stdio.h>
 2
 3   main(){
 4
 5     printf("One thread ... \n");
 6
 7   #pragma omp parallel
 8     {
 9       int myid,i;
10
11       myid = omp_get_thread_num();
12
13       for(i=0;i<2;i++){
14         printf("HELLO %d, %d\n",get
15       }
16
17     }
18
19     printf("... back to one thread.\n");
20
21     return;
22   }
```

```
#0 main line: 14 in "omp.c" address: 0x8049756
```

Stopped at line 14 (address 0x8049756) in file /home/sw/demos/TOOLS_DEMO/OMP/omp.c

1.12.3.5 Messages Subwindow

Figure 1-14 (in *section 1.17.3*) illustrates the *Messages* subwindow. Refer to S*ection 1.17.3 MPI Message Queues* for more information on the content of this subwindow.

1.13 *PGDBG*: Parallel Debug Capabilities

This section describes the parallel debug capabilities of *PGDBG*. *PGDBG* is a parallel application debugger capable of debugging distributed-memory MPI programs, thread-parallel shared-memory parallel (SMP) OpenMP and Linuxthreads/pthreads programs, and hybrid shared/distributed programs that use MPI to communicate between thread-parallel SMP processes.

See http://www.pgroup.com/docs.htm for the most recent documentation. This material is also available in $PGI/docs/index.htm. See http://www.pgroup.com/faq/index.htm for an online FAQ.

1.13.1 OpenMP and Linuxthread Support

- Full thread control in parallel regions
- Thread grouping
- Threads presented by their OpenMP logical thread number
- Line level debugging preserved when thread
    - Enters a parallel region
    - Enters a serial region
    - Hits an OpenMP barrier
    - Hits an OpenMP synchronize statement
    - Enters an OpenMP sections program section
- Informative messages regarding thread state and location

1.13.2 MPI Support

- Automatic process detection and attach
- Informative messages regarding process state and location

- Process grouping
- Processes presented by their global rank in COMMWORLDx

1.13.3 Process & Thread Control

- Concise control of groups of processes/threads
- Thread and process synchronization
- Configurable thread and process stop mode
- Configurable wait mode
- Serial, process-only, threads-only, and multilevel debug modes

1.13.4 Graphical Presentation of Threads and Processes

- Process grid
- Thread grid
- Graphical grouping logic
- Color depiction of whole program execution state
- *Summary* panel (selected by the *Summary* tab in the process/thread grid). Lists each thread by its logical CPU ID. Displays for each thread its state and stop location. Threads are grouped by parent process. See *Section 1.12.1 Main Window* for more information on the *Summary* panel.
- Process/Thread grid. Displays each process/thread as a color-coded button in a grid. Click on a grid element to refresh the GUI in the scope of that process. Each grid element is numbered with the process/thread's logical ID. See *Section 1.12.1 Main Window* for more information on the process/thread grid.
- Zoom in and out of the Process/Thread Grid to easily focus on a subset of processes/threads. The zoom slider bar is located on the right of the process/thread grid (*Section 1.12.1 Main Window*).
- Control all, some, or one process/thread at a time. Form subsets of processes/threads using *focus groups*. See *Section1.12.1.2 Focus Panel* for more information on the *Focus Panel.*

1.14 Debugging Parallel Programs with *PGDBG*

This section describes how to invoke the debugger for thread-parallel (SMP) debugging and for process-parallel (MPI) debugging. It provides some important definitions and background information on how *PGDBG* represents processes and threads.

1.14.1 Processes and Threads

An active process is made up of one or more active threads of execution. In the context of a process-parallel program, a process is an MPI process composed of one thread of execution. In the context of a thread-parallel program, a thread is an OpenMP or Linux Pthread SMP thread. *PGDBG* is capable of debugging hybrid process-parallel/thread-parallel programs where the program employs multiple SMP processes.

When debugging an OpenMP program, *PGDBG* identifies threads using their OpenMP IDs. Otherwise, *PGDBG* assigns arbitrary IDs to threads; starting at zero and incrementing in order of thread creation.

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

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 having thread ID 4 and the parent process having process ID 1.

An OpenMP program (thread-parallel only) logically runs as a collection of threads with a single process, process 0, as the parent process. In this context, a thread is uniquely identified by its thread ID. The process ID prefix is implicit and optional. See *Section 1.15.2 Threads-only debugging.*

An MPI program (non-SMP) logically runs as a collection of processes, each made up of a single thread of execution. Thread 0 is implicit to each MPI process. A Process ID uniquely identifies a particular process, and thread ID is implicit and optional. See *Section 1.15.3 Process-only debugging.*

A hybrid, or *multilevel* MPI/OpenMP program, requires the use of both process and thread IDs to uniquely identify a particular thread. See *Section 1.15.4 Multilevel debugging.*

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

1.14.2 Thread-Parallel Debugging

PGDBG automatically attaches to new threads as they are created during program execution. *PGDBG* describes when a new thread is created; the thread ID of each new thread is printed.

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

The system ID of the freshly created thread is available through using the *threads* command. Use the *procs* command to display information about the parent process.

During a debug session, at any one time, *PGDBG* operates in the context of a single thread, the 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). See *Section 1.15.10.2 Thread Level Commands.*

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

```
pgdbg [all] 2> thread 3

pgdbg [all] 3> threads

0   ID PID    STATE      SIGNAL      LOCATION

=> 3  18399  Stopped    SIGTRAP     main line: 31 in "omp.c" address:
    0x80490ab

   2  18398  Stopped    SIGTRAP     main line: 32 in "omp.c" address:
    0x80490cf

   1  18397  Stopped    SIGTRAP     main line: 31 in "omp.c" address:
    0x80490ab

   0  18395  Stopped    SIGTRAP     f line: 5 in "omp.c" address:
    0x8048fa0
```

1.14.2.1 Invoking *PGDBG*: OpenMP, Linux Pthread Debugging

Use the following to invoke *PGDBG*, OpenMP, Linux Pthread debugging using text or GUI mode:

```
GUI mode:

   %pgdbg <executable> <args>,...<args>


TEXT mode:

   %pgdbg -text <executable> <args>,...<args>
```

1.14.3 Graphical Features

The *PGDBG* Graphical User Interface (GUI) lists all active threads in a thread grid. Each element of the thread grid is labeled with a thread ID and represents a single thread. Each element is a button that can be pushed to select a particular thread as the current thread. The *PGDBG* GUI displays the program context of the current thread. Figure 1-3 shows the thread grid in the GUI with two active threads.

Each button in the thread grid is color coded to depict the execution state of the underlying thread.

Table 1-12: Thread State is Described using Color

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

1.14.4 Process-Parallel Debugging

PGDBG automatically attaches to new MPI processes as they are created by a running MPI program. *PGDBG* must be invoked via the MPIRUN script. Use the MPIRUN *-dbg* option to specify which debugger to use. To choose *PGDBG*, use *-dbg=pgdbg* before the executable name (this is not a program argument). *PGDBG* must be installed on your system and your PGI environment variable set appropriately, and added to your PATH.

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. Use the *procs* command to list the host and the PID of each process by rank. The current process is marked with an arrow. To change the current process by process ID, use the *proc* command.

```
    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


    pgdbg [all] 1.0>
```

The prompt displays the current process and the current thread. The current process above has been changed to process 1, and the current thread of process 1 is 0. This is written as 1.0. See *Section 1.15.15 The PGDBG Command Prompt* for a complete description of the prompt format.

The following rules apply during a *PGDBG* debug session:

- At any one time, *PGDBG* operates in the context of a single process, the current process.
- Each active process has a thread set of size >=1.
- The current thread is a member of the thread set of the current process.

A license file distributed with *PGDBG* that restricts *PGDBG* to debugging a total of 64 threads. *Workstation* and *CDK* license files may further restrict the number of threads that *PGDBG* is eligible to debug. *PGDBG* will use the *Workstation* or *CDK* license files to determine the number of threads it is able to debug.

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

Use the *proc* command to change the current process. Those *PGDBG* commands that refer to program scope execute off of the current scope of the current thread by default. The current thread must be stopped in order to read from its memory space. See *Section 1.15.10.2 Thread Level Commands* for a description and list of these context sensitive commands.

To list all active processes, use the *procs* command. The *procs* command lists all active processes by process ID (MPI rank where applicable). Listed for each process: the system ID of the initial thread, process execution state, number of active threads, and host name. The initial process is run locally; 'local' describes the host the debugger is running on. The execution state of a process is described in terms of the execution state of its component threads:

Table 1-13: Process state is described using color

Process state	Description	Color
Stopped	If all threads are stopped at breakpoints, or where directed to stop by *PGDBG*	Red
Signaled	If at least one thread is stopped on an interesting signal (as described by *catch*)	Blue
Running	If at least one thread is running	Green
Exited or Killed	If all threads have been killed or exited	Black

1.14.4.1 Invoking *PGDBG*: MPI Debugging

To debug an MPI program, *PGDBG* is invoked via MPIRUN. MPIRUN sets a breakpoint at `main` and starts the program running under the control of *PGDBG*. When the initial process hits `main` no other MPI processes are active. The non-initial MPI processes are created when the process calls `MPI_Init`.

A Fortran MPI program stops at `main` initially instead of `MAIN`. You must *step* into `MAIN`.

GUI mode:

```
%mpirun -np 4 -dbg=pgdbg <executable> <args>,...<args>
```

TEXT mode:

```
%unsetenv DISPLAY

%mpirun -np 4 -dbg=pgdbg <executable> <args>,...<args>
```

An MPI debug session starts with the initial process stopped at `main`. Set a breakpoint at a program location after the return of `MPI_Init` to stop all processes there. If debugging Fortran, *step* into the `MAIN` program.

1.14.4.2 MPI-CH Support

PGDBG supports redirecting `stdin`, `stdout`, and `stderr` with the following MPI-CH switches:

Table 1-14: MPI-CH Support

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

PGDBG also provides support for the following MPI-CH switches:

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

If you are using your own version of MPI-CH, see our online FAQ for how to integrate the MPIRUN scripts with *PGDBG*.

When *PGDBG* is invoked via MPIRUN the following *PGDBG* command line arguments are not accessible. A possible 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*.

1.14.4.3 LAM-MPI Support

The CDK comes with MPI-CH. *PGDBG* is configured to automatically work with MPI-CH. *PGDBG* also works with LAM-MPI, but not automatically. For more information, see the online FAQ at http://www.pgroup.com/faq/index.htm.

1.15 Thread-parallel and Process-parallel Debugging

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.

1.15.1 *PGDBG* Debug Modes

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

Table 1-15: The *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 starts out 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.

If *PGDBG* is licensed as a *Workstation* product, it operates in Threads-only mode by default (however multilevel notation is always valid).

If *PGDBG* is licensed as a *CDK* product, it operates in process-only mode by default.

The *PGDBG* prompt displays the current thread according to the current debug mode. See *Section 1.15.15 The PGDBG Command Prompt* for a description of the *PGDBG* prompt.

The *pgienv* command is used to change debug modes manually.

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

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

1.15.2 Threads-only debugging

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

Example 1-1: Thread IDs in threads-only debug mode

`1`	Thread 1 of all processes (*.1)
`*`	All threads of all processes (*. *)
`0.7`	Thread 7 of process 0 (Multilevel thread names valid in threads-only debug mode)

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

1.15.3 Process-only debugging

Enter process-only mode to debug a program with non-SMP nodes. As a convenience, the thread ID portion can be omitted. *PGDBG* automatically enters process-only debug mode from serial debug mode when the target program returns from `MPI_Init`.

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

`0`	All threads of process 0 (0.*)
`*`	All threads of all processes (*.*)
`1.0`	Thread 0 of process 1 (Multilevel thread names are valid in this mode)

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

1.15.4 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 spawns SMP threads.

Example 1-3: Thread IDs in multilevel debug mode

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

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

1.15.5 Process/Thread Sets

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

The *current p/t-set* can be set using the *focus* command, which establishes the default p/t-set for cases where no p/t-set prefix is specified. This begins as the debugger-defined set `[all]`, which describes all threads of all processes.

P/t-set notation can be used to prefix a debugger command. This overrides the current p/t-set defining the target threads to be those threads described by the *prefix p/t-set*.

The *target p/t-set* is defined then to be the prefix p/t-set if present, it is the current p/t-set otherwise.

- Use *defset* to define a named or user-defined p/t-set.
- Use *viewset* and *whichsets* to inspect the active members described by a particular p/t-set.

The target p/t-set determines which threads are affected by a *PGDBG* command. If you are using the *PGDBG* Graphical User Interface (GUI) (*Section 1.12 PGDBG GRAPHICAL USER INTERFACE*), then you can define a *focus group* in the GUI's *Focus* panel as an alternative to the *defset* command (*Section 1.12.1.2 Focus Panel*). We will discuss this further in *Section 1.15.9 P/t-set Commands.*

1.15.6 P/t-set Notation

The following set of rules describes how to use and construct process/thread sets (*p/t-sets*).

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

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

p/t-id :
    {integer|*}.{integer|*}
```

Optional notation when processes-only debugging or threads-only debugging is in effect (see the *pgienv* command).

{integer|*}

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

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

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

Example 1-4: P/t-sets in threads-only debug

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

Example 1-5: P/t-sets in process-only debug

`[0,2:3]`	Processes 0, 2, and 3 (equivalent to [0.*,2:3.*])
`[*]`	All processes (equivalent to [*.*])
`[0]`	Process 0 (equivalent to [0.*])
`[*.0]`	Process 0. Multilevel syntax is valid in process-only mode.
`[0:2.*]`	Processes 0, 1, and 2. Multilevel syntax is valid in process-only debug mode.

Example 1-6: P/t-sets in multilevel debug mode

`[0.1,0.3,0.5]`	Thread 1,3, and 5 of process 0
`[0.*]`	All threads of process 0
`[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

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

1.15.7 Dynamic vs. Static P/t-sets

The members of a *dynamic p/t-set* are those active threads described by the p/t-set at the time that p/t-set is used. A p/t-set is dynamic by default. Threads and processes are created and destroyed as the target program runs. Membership in a dynamic set varies as the target program runs.

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

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

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

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

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

1.15.8 Current vs. Prefix P/t-set

The current p/t-set is set by the *focus* command. The current p/t-set is described by the debugger prompt (depending on debug mode). A p/t-set can be used to prefix a command to override the current p/t-set. The prefix p/t-set becomes the target p/t-set for the command. The target p/t-set defines the set of threads that will be affected by a command. See *Section 1.15.15 The PGDBG Command Prompt* for a description of the *PGDBG* prompt.

- The target p/t-set is the current p/t-set:

  ```
  pgdbg [all] 0.0> cont
  ```

  ```
  Continue all threads in all processes
  ```

- The target p/t-set is the prefix p/t-set:

  ```
  pgdbg [all] 0.0> [0.1:2] cont
  ```

  ```
  Continue threads 1 and 2 of process 0 only
  ```

Above, the current p/t-set is the debugger-defined set `[all]` in both cases. In the first case, `[all]` is the target p/t-set. In the second case, the prefix set overrides `[all]` as the target p/t-set. The *continue* command is applied to all active threads in the target p/t-set. Using a prefix p/t-set does not change the current p/t-set.

1.15.9 P/t-set Commands

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

- *defset* and *undefset* can be used to manage a list of named p/t-sets.
- *focus* is used to set the current p/t-set.
- *viewset* is used to view the active members described by a particular p/t-set.
- *whichsets* is used to describe the p/t-sets to which a particular process/thread belongs.

Table 1-16: P/t-set commands

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

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


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

The p/t-set `initial` is defined to contain only thread 0. We focus on `initial` and advance the thread. *Focus* sets the current p/t-set. Because we are not using a prefix p/t-set, the target p/t-set is the current p/t-set which is `initial`.

The *whichsets* command above shows us that thread 0 is a member of two defined p/t-sets. The *viewset* command displays all threads that are active and are members of defined p/t-sets. The `'pgienv verbose'` command can be used to turn on verbose messaging, displaying the stop location of each thread as it stops.

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

```
all
initial

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

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

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

The examples above illustrate how to manage named *p/t-sets* in the command-line interface. A similar capability is available in the *PGDBG GUI.* In *Section 1.12.2 Focus Panel,* we introduced the *Focus Panel*. The *Focus Panel,* shown in Figure 1-3, contains a table labeled *Focus* with two columns: a *Name* column and a *p/t-set* column. The entries in this table are called *focus groups*. Like the named *p/t-sets* created with the *defset* command, *focus groups* are named *p/t-sets* in the GUI.

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

To select a *focus group,* click the left mouse button on the desired *focus group* in the table. The selected *focus group* is also known as the *Current Focus*. The GUI will apply all commands entered in its *Source Panel* to the *Current Focus* when you choose *Focus* in the *Apply Selector* (*Section 1.12.2.3 Source Panel Combo Boxes*). *Current Focus* can also be used in a GUI subwindow. Choose *Current Focus* in a subwindow's *Context Selector* (*Section 1.12.3 Subwindows*) to display data for the *Current Focus* only.

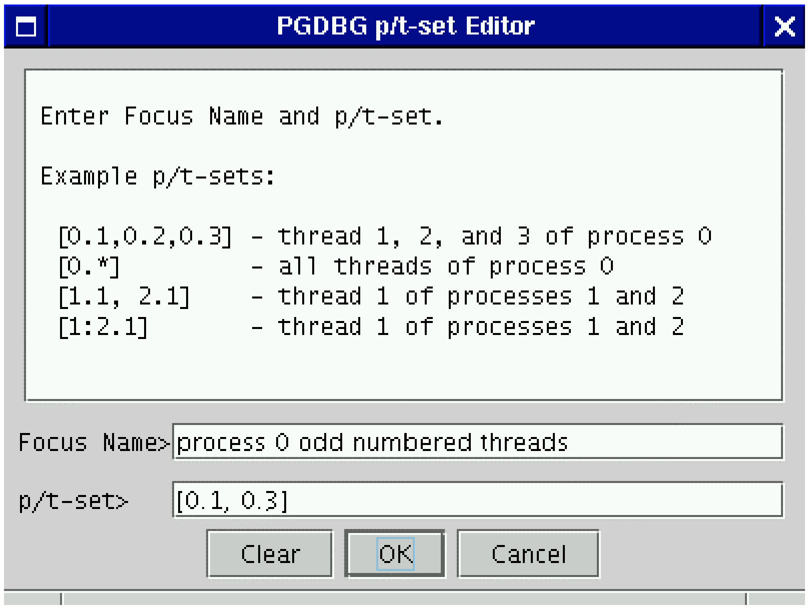
To modify an existing *focus group*, select the desired group in the *Focus Table* and left mouse click the *Modify* button. You will see a similar dialog box as Figure 1-12 except the *Focus Name* and *p/t-set* text fields will contain the selected group's name and *p/t-set* respectively. You can edit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

To remove an existing *focus group,* select the desired group in the *Focus Table* and left mouse click the *Remove* button. The GUI will display a dialog box asking you to confirm removal of the selected *focus group.* Left mouse click the *Yes* button to confirm, or click the *No* button to cancel the operation.

It should be noted that *focus groups* are only used by the *Apply* and *Context Selectors* in the GUI. They do not affect *focus* in the command-line interface. If you are using the *command prompt* in the GUI, then you will still need to use the *defset* and *focus* commands to change focus within the *command prompt*. The focus changes made in the *command prompt* only affect the *command prompt* and not the rest of the GUI.

As an example, let us return to Figure 1-12. Here we created a *focus group* called "process 0 odd numbered threads". The *p/t-set* associated with this group is `[ 0.1, 0.3 ]` which indicates threads 1 and 3 on process 0. In Figure 1-13, we selected this *focus group* in the *Focus Table.* We also chose *Focus* in the *Apply Selector*. Any command issued in the *Source Panel* gets applied to the *Current Focus,* or thread 1 and 3 on process 0 only. All other threads will remain idle until we either select the *All focus group* or choose *All* in the *Apply Selector.*

Figure 1-12: Focus Group Dialog Box

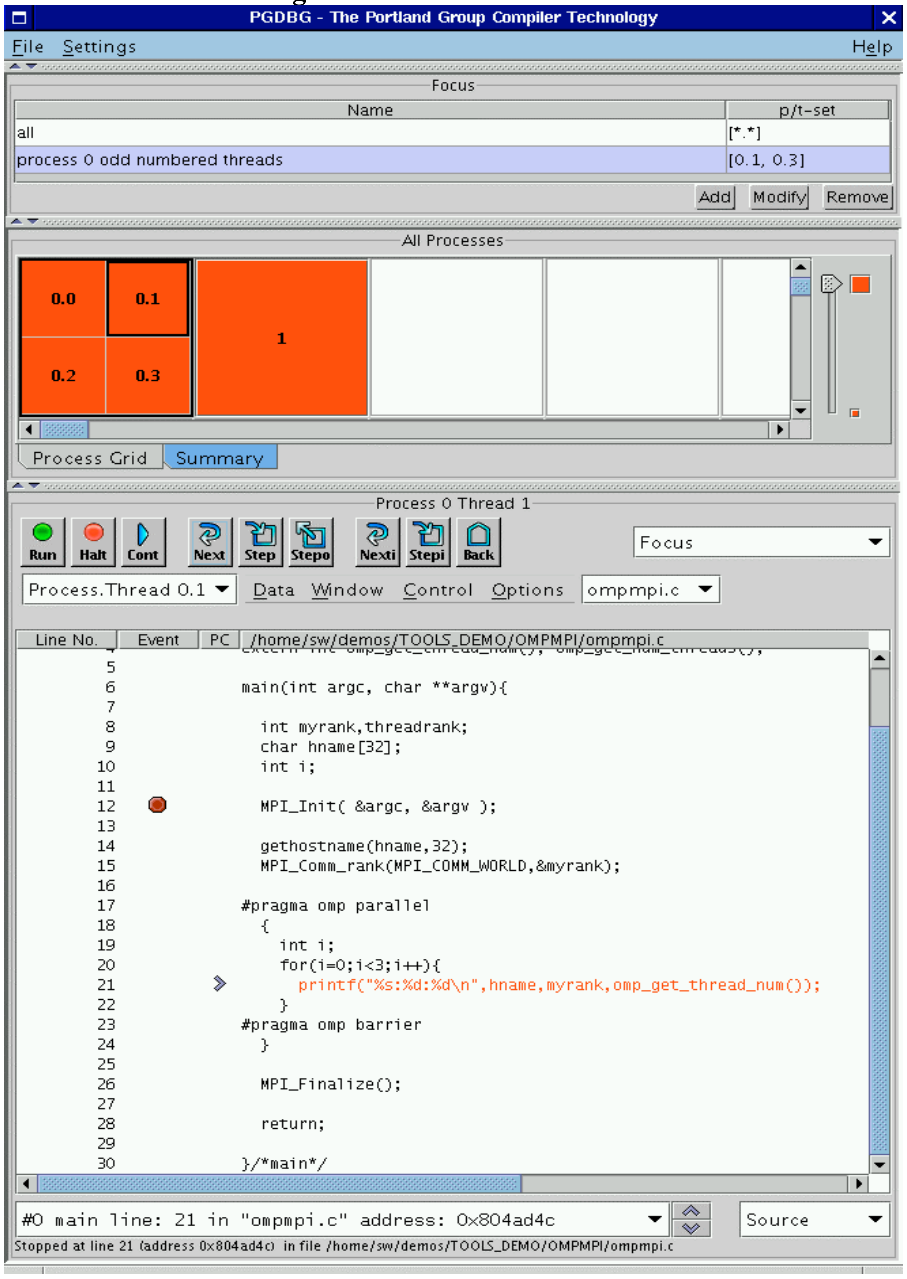


```
Enter Focus Name and p/t-set.

Example p/t-sets:

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

Focus Name> process 0 odd numbered threads

p/t-set> [0.1, 0.3]

Figure 1-13: Focus in the GUI


PGDBG - The Portland Group Compiler Technology
File Settings Help
Focus
Add Modify Remove
All Processes
0.0 0.1 0.2 0.3 1
Process Grid Summary
Process 0 Thread 1
Run Halt Cont Next Step Stepo Nexti Stepi Back
Focus
Process.Thread 0.1
Data Window Control Options
ompmpi.c
Line No. Event PC /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c
Source

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

```
main(int argc, char **argv){

  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: 0x804ad4c
Stopped at line 21 (address 0x804ad4c) in file /home/sw/demos/TOOLS_DEMO/OMPMPI/ompmpi.c
```

1.15.10 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 are parallelized. *Global* commands are not.

Table 1-17: *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. Ignores current p/t-set
Global Commands	Non-parallel commands

1.15.10.1 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. If a target p/t set does not exist, the current p/t-set is the prefix.

cont	next	nexti	step	stepi
stepout	sync	synci	halt	wait

Example:

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

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

The *next* command is applied to threads 1 and 2 of process 0.

Example:

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

This demonstrates the use of a prefix p/t-set. The *next* command is applied to thread 3 of process 0 only.

1.15.10.2 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 if not prefix p/t set exists. The *thread level* commands are:

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

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

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

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

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

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> [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
```

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

1.15.10.3 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 a prefix p/t-set are ignored.

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

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

1.15.11 Process and Thread Control

PGDBG supports thread and process control (‘*step*ping’, ‘*next*ing', '*cont*inuing' ...) everywhere in the program. Threads and processes can be advanced in groups anywhere in the program.

The *PGDBG* control commands are:

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

To describe those threads you wish to advance, 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. So if the current thread '*next*'s over an `_mp_init` call (at the beginning of every OpenMP parallel region), then all threads created by `_mp_init` will '*next*' into the parallel region.

A process inherits the control operation of the current process when it is created. So if the current process is '*cont*inuing' out of a call to `MPI_Init`, the new process will do the same.

The *PGDBG* GUI supports process/thread selection via the use of the process/thread grid. To change the current process/thread, click on the corresponding button in the grid. See *Section 1.12.1 Main Window* for sample of the graphical user interface.

Accompanying each grid is a set of toggle buttons with the labels *'all'* or *'current'*. These buttons can be used to construct a prefix p/t-set for the next command. The toggle buttons apply to the *'cont'*, *'stepout'*, *'next'*, *'nexti'*, *'step'*, *'stepi'*, *'halt'*, and *'wait'* buttons only.

1.15.12 Configurable Stop Mode

PGDBG lets you configure how threads and processes stop in relation to one another. *PGDBG* defines two new `pgienv` environment variables, `threadstop` and `procstop`, for this purpose. *PGDBG* defines two *stop modes*, synchronous (`sync`) and asynchronous (`async`).

Table 1-18: *PGDBG* Stop Modes

Command	Result
`sync`	Synchronous stop mode; when one thread stops at a 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 variable `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.

1.15.13 Configurable Wait mode

Wait mode describes when *PGDBG* will accept the next command. The wait mode is defined in terms of the execution state of the program. Wait mode describes to the debugger which threads/processes must be stopped before it will accept the next command. In certain situations, it is desirable to be able to enter commands while the program is running and not stopped. The *PGDBG* prompt will not appear until all processes/threads are stopped. However, a prompt may be available before all processes/threads have stopped. Pressing `<enter>` at the command line will bring up a prompt if it is available. The availability of the prompt is determined by the current wait mode and any pending wait commands (described below).

PGDBG accepts a compound statement at each prompt. Each compound statement is a bundle of commands, which are processed in order at once. The wait mode describes when to accept the next compound statement. *PGDBG* supports three wait modes:

Table 1-19: *PGDBG* Wait Modes

Command	Result
`any`	The prompt is available only after at least one thread has stopped since the last control command
`all`	The prompt is available only after all threads have 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 a next command.
- *Process wait mode* describes which processes *PGDBG* waits for before accepting a next command.

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

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

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

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

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

In TEXT mode *PGDBG* defaults to

```
threadwait  all
procwait    any
```

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

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

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

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

PGDBG defaults to the following in GUI mode:

```
threadwait none
procwait   none
```

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

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

The `threadwait` and `procwait` environment variables can be used to configure the behavior of `wait` (see *`pgienv`*).

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

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

Table 1-20: *PGDBG* Wait Behavior

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

Command	threadwait	procwait	Wait set
`wait all`	none|any	none|any	Wait for all t in T for at least one p in P
`Wait none`	all|none|any	all|none|any	Wait for no threads

1.15.14 Status Messages

Use the `pgienv` command to enable/disable various status messages. This can be useful in text mode in the absence of the graphical aids provided by the GUI.

```
pgienv verbose <bitmask>
```

Choose the debug status messages that are reported by *PGDBG*. The tool accepts an integer valued bit mask of the values described in the following table.

Table 1-21: *PGDBG* Status Messages

Thread	Format	Information
0x1	Standard	Report status information on current process/thread only. A message is printed only when the current thread stops. Also report when threads and processes are created and destroyed. Standard messaging cannot be disabled. (default)
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/exits a parallel region, or when the threads synchronize. The *PGDBG* OpenMP handler must be enabled.
0x16	Parallel	Report process-parallel events (default). Currently unused.
0x32	Symbolic debug information	Report any errors encountered while processing symbolic debug information (e.g. STABS, DWARF).

1.15.15 The PGDBG Command Prompt

The *PGDBG* command prompt reflects the current debug mode (See *Section 1.15.1 PGDBG Debug Modes).*

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

```
    pgdbg>
```

In threads-only debug mode, *PGDBG* displays the current p/t-set 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 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 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.

See *Section 1.15.1 PGDBG Debug Modes* for a description of the *PGDBG* debug modes.

1.15.16 Parallel Events

This section describes how to use a p/t-set to define an event across multiple threads and processes. *1.9.1.3 Events,* such as breakpoints and watchpoints, are user-defined events. User defined events are Thread Level commands (See *Section 1.15.10.2 Thread Level Commands* for details).

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

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 on threads 1,2,3 of
   process 0 only.
```

All other user events by default are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads.

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 data
   breakpoint for glob on all threads that are currently active.
```

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

Many events contain 'if' and 'do' clauses.

Example:

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

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 on threads 1,2,3 of
   process 0 only.
```

All other user events by default are set for the current thread only. A prefix p/t-set can be used to set user events on specific processes and threads.

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 data
   breakpoint for glob on all threads that are currently active.
```

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

Many events contain 'if' and 'do' clauses.

Example:

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

The breakpoint will fire only if `glob` is non-zero. The 'do' clause is executed if the breakpoint fires. The 'if' clause and the 'do' clause execute in the context of a single thread. The conditional in the 'if' and the body of the 'do' execute off of a single (same) thread; the thread that triggered the event. Think of the above definition as:

```
[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.

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.

```
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.

Example:

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

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

1.15.17 Parallel Statements

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

1.15.17.1 Parallel Compound/Block Statements

Example:

```
pgdbg [all] 0>[*] break main;

cont; wait; print f@11@i

ii.) pgdbg [all] 0>[*] break main;

[*]cont; [*]wait; [*]print f@11@i
```

i. and ii. are equivalent. Each command in a compound statement is executed in order. The target p/t-set is broadcast to all statements. Use the *wait* command if subsequent commands require threads to be stopped (the *print* command above). The commands in a compound statement are executed together in order.

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.

1.15.17.2 Parallel If, Else Statements

This section describes parallel 'if' and 'else' statements.

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

A prefix p/t-set parallelizes an 'if' statement. An 'if' statement executes in the context of the current thread by default. The above example is equivalent to:

```
[*] if (i==1) ==> [s]
    [s]break func; [s]c; [s]wait;
else ==> [s']
    [s']sync func2
```

Where [s] is the subset of [*] for which (i==1), and [s'] is the subset of [*] for which (i!=1).

1.15.17.3 Parallel While Statements

This section describes parallel 'while' statements.

Example:

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

A prefix p/t-set parallelizes a 'while' statement. A 'while' statement executes in the context of the current thread by default. The above example is equivalent to:

```
[*] ==> [s]
while(|[s]|){
[s] if (i<10) ==> [s]
    [s]n; [s]wait; [s]print i;
}
```

Where [s] is the subset of [*] for which (i<10). The 'while' statement terminates when [s] is the empty set (or a 'return') statement is executed in the body of the 'while'.

1.15.17.4 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 will return the last value evaluated.

1.16 OpenMP Debugging

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

- Entrance to parallel region
- Exit parallel region
- Nested parallel regions synchronization points
- Critical and exclusive sections
- Parallel sections

1.16.1 Serial vs. Parallel Regions

The initial thread is the thread with OpenMP ID 0. Conceptually, the initial thread is the only thread that is well defined (for the purpose of doing useful work) in a serial region of code. All threads are well defined in a parallel region of code. When the initial thread is in a serial region, the non-initial threads are busy waiting at the end of the last parallel region, waiting to be called down to do some work in the next parallel region. All threads enter the (next) parallel region only when the first thread has entered the parallel region, (i.e., the initial thread is not in a serial region.)

PGDBG source line level debugging operations (*next*, *step*,...) are not well defined for non-initial threads in serial regions since these threads are stuck in a busy loop, which is not compiled to include source line information. The instruction level *PGDBG* control commands (*nexti*, *stepi*, ...) are well defined if you want to advance through the described wait loop at the assembly level.

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

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

Upon entering a parallel region, *PGDBG* automatically changes the *thread stop mode* to *asynchronous stop mode* and the *threadwait mode* to `none'. This allows you to control each thread independently. For example, a *next* command, applied to all threads in a parallel region, will not complete successfully until all threads hit their next source line. With the *thread wait mode* set to `none', use the *halt* command on threads that hit barrier points.

To disable the automatic configuration of the *thread wait* and *thread stop* modes, see the *threadstopconfig* and *threadwaitconfig* options of the *pgienv* command (*Section 1.9.1.11 Miscellaneous)*.

The configuration of the *thread wait* and *stop* modes, as described above, occurs automatically for OpenMP programs only. When debugging a Linuxthread program, the *threadstop* and *threadwait* configuration options should be set using the *pgienv* command (*Section 1.9.1.11 Miscellaneous)*.

1.16.2 The PGDBG OpenMP Event Handler

PGDBG provides explicit support for OpenMP events. OpenMP events are points in a well-defined OpenMP program where the behavior of one thread depends on the location of another thread. For example, a thread may continue after another thread reaches a barrier point. The *PGDBG* OpenMP event handler is disabled by default. It can be enabled using the *omp pgienv* environment variable as shown below:

```
pgienv omp [ on | off ]
```

The *PGDBG* OpenMP event handler sets breakpoints before a parallel region, after a parallel region, and at each thread synchronization point. This causes a noticeable slowdown in performance of the program as it runs with the debugger.

The OpenMP event handler is deprecated as of *PGDBG* release 5.2.

1.17 MPI Debugging

1.17.1 Process Control

PGDBG is capable of debugging parallel-distributed MPI programs and hybrid distributed SMP programs. *PGDBG* is invoked via *MPIRUN* and automatically attaches to each MPI process as it is created.

See *Section 1.14.4 Process-Parallel Debugging* 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. Mind process dependencies.
- In order for a process to receive a message, the sender must be allowed to run.
- Process synchronization points, such as `MPI_Barrier`, will not return until all processes have hit the sync point.
- `MPI_Finalize` will not return for Process 0 until Process 1..n-1 exit.

A control command (*cont*, *step*,...) can be applied to a stopped process while other processes are running. A control command applied to a running process is applied to the stopped threads of that process, and is ignored by its running threads. Those threads that are held by the OpenMP event handler will also ignore the control command in most situations.

PGDBG automatically switches to process wait mode `none ('pgienv procwait none')` as soon as it attaches to its first MPI process. See the *pgienv* command and *Section 1.17.5 MPI Listener Processes* 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.

1.17.2 Process Synchronization

Use the *PGDBG sync* command to synchronize a set of processes to a particular point in the program.

```
pgdbg [all] 0.0> sync MPI_Finalize
```

This command runs all processes to `MPI_Finalize`.

```
   pgdbg [all] 0.0> [0:1.*] sync MPI_Finalize
```

This command runs process 0 and process 1 to `MPI_Finalize`.

A synchronize command will only successfully *sync* the target processes if the *sync* address is well defined for each member of the target process set, and all process dependencies are satisfied (otherwise the member could wait forever for a message for example). The debugger cannot predict if a text address is in the path of an executing process.

1.17.3 MPI Message Queues

PGDBG can dump the MPI message queues through the *mqdump* command (*Section 1.9.1.9 Memory Access*). In the *PGDBG* GUI, you can view the message queues by selecting the *Messages* item under the *Windows* menu. This command can also have a P/t-set prefix (*Section 1.15.6 P/t-set Notation)* to specify a subset of processes and/or threads. Figure 1-14 shows an example output of the *mqdump* command as seen in the GUI (the *PGDBG* text debugger produces the same output).

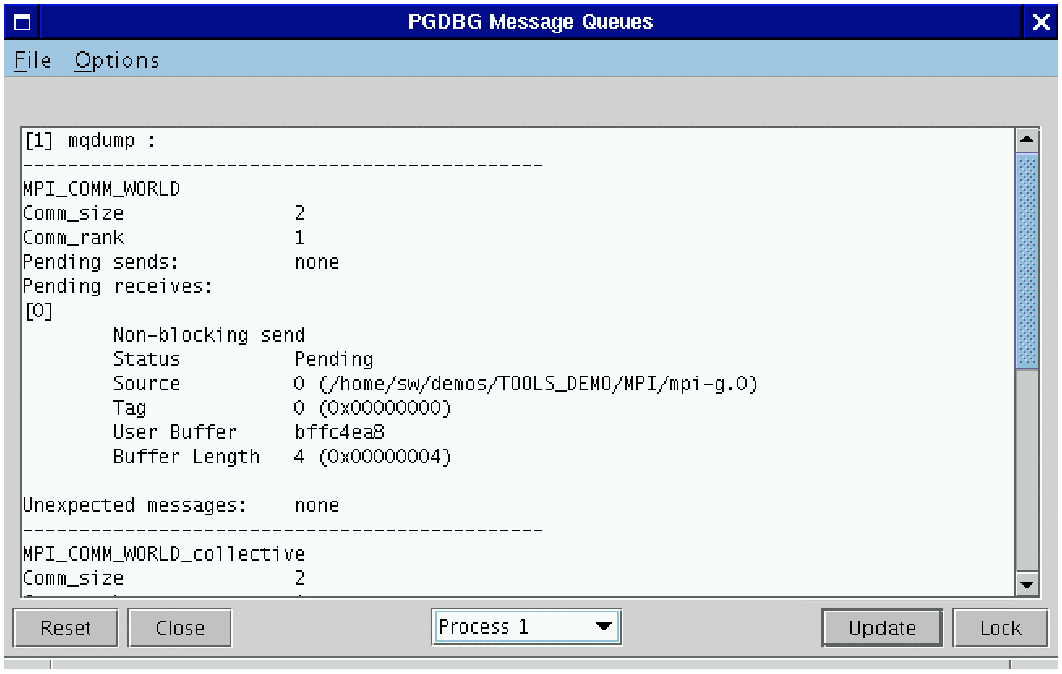
When using the GUI, a subwindow is displayed with the message queue output. Within the subwindow, you can select which process/threads to display with the *Context Selector* combo box located at the bottom of the subwindow (e.g., *Process1* in Figure 1-14).

The message queue dump is only available for MPI application debugging on the *CDK* licensed version of *PGDBG*. You may see the following error message if you invoke *mqdump:*

```
ERROR: MPI Message Queue library not found. Try setting
'PGDBG_MQS_LIB_OVERRIDE' environment variable.
```

If you see this message and you are using a *CDK* licensed version of *PGDBG*, then you may need to set the `PGDBG_MQS_LIB_OVERRIDE` environment variable to the absolute path of the `libtvmpich.so` or compatible library (normally located in `$PGI/lib`).

Figure 1-14: Messages Subwindow



```
[1] mqdump :
----------------------------------------------
MPI_COMM_WORLD
Comm_size               2
Comm_rank               1
Pending sends:          none
Pending receives:
[0]
        Non-blocking send
        Status          Pending
        Source          0 (/home/sw/demos/TOOLS_DEMO/MPI/mpi-g.0)
        Tag             0 (0x00000000)
        User Buffer     bffc4ea8
        Buffer Length   4 (0x00000004)

Unexpected messages:    none
----------------------------------------------
MPI_COMM_WORLD_collective
Comm_size               2
```

1.17.4 MPI Groups

PGDBG identifies each process by its COMMWORLD rank. In general, *PGDBG* currently ignores MPI groups.

1.17.5 MPI Listener Processes

Entering `Control-C` (^C) from the *PGDBG* command line can be used to halt all running processes. However, this is not the preferred method to use while debugging an MPI program. Entering ^C at the command line, sends a `SIGINT` signal to the debugger's children. This signal is never received by the MPI processes listed by the *procs* command (i.e., the initial and attached processes), `SIGINT` is intercepted in each case by *PGDBG*. *PGDBG* does not attach to the MPI listener processes that pair each MPI process. These processes handle IO requests among other things. As a result, a ^C from the command line will kill these processes resulting in undefined program behavior.

It is for this reason, that *PGDBG* automatically switches to process wait mode `none ('pgienv procwait none')` as soon as it attaches to its first MPI process. This allows the use of the *halt* command to stop running processes, without the use of ^C. The setting of `'pgienv procwait none'` allows commands to be entered while there are running processes.

Note: *halt* cannot interrupt a *wait* by definition of *wait*. ^C must be used for this, or careful use of *wait*.

1.17.6 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 communication method needed. The communication between the *PGDBG GUI* client *(Section 1.12 PGDBG Graphical User Interface)* and the *PGDBG* server is not secure, so use the command-line interface when using *ssh* for communication of remote *PGDBG* components.

Chapter 2

The *PGPROF* Profiler

This chapter introduces the *PGPROF* profiler. The profiler is a tool that analyzes data generated during execution of specially compiled *C*, *C++*, F77, F9x and HPF programs. The *PGPROF* profiler lets you discover which routines and lines were executed as well as how often they were executed and how much of the total time they consumed.

The *PGPROF* profiler also allows you to profile multi-process HPF or MPI programs, multi-threaded SMP programs (e.g., OpenMP or programs compiled with *–Mconcur,* etc.), or hybrid multi-process programs employing multiple processes with multiple SMP threads for each process. The multi-process information lets you select combined minimum and maximum process data, or select process data on a process-by-process basis. Multi-threaded information can be queried in the same way as on a per-process basis. This information can be used to identify communications patterns, and identify the portions of a program that will benefit the most from performance tuning.

2.1 Introduction

Profiling is a three-step process:

Compilation	Compiler switches cause special profiling calls to be inserted in the code and data collection libraries to be linked in.
Execution	The profiled program is invoked normally, but collects call counts and timing data during execution. When the program terminates, a profile data file is generated (e.g., *pgprof.out, gmon.out*, etc.).
Analysis	The *PGPROF* tool interprets the *pgprof.out* file to display the profile data and associated source files. The profiler supports routine level, line level and data collection modes. The next section provides definitions for these data collection modes.

2.1.1 Definition of Terms

Routine Level Profiling	Is the strategy of collecting call counts and execution times on a per routine (e.g., subroutine, subprogram, function, etc.) basis.
Function Level Profiling	Synonymous with *Routine Level Profiling*.
Line Level Profiling	Execution counts and times within each routine are collected in addition to routine level data. *Line Level* is somewhat of a misnomer because the granularity ranges from data for individual statements to data for large blocks of code, depending on the optimization level. At optimization level 0, the profiling is truly line level.
Basic Block	At optimization levels above 0, code is broken into basic blocks, which are groups of sequential statements with only one entry and one exit. Line level profile data is collected on basic blocks rather than individual statements at these optimization levels.
Virtual Timer	A statistical method for collecting time information by directly reading a timer which is being incremented at a known rate on a processor by processor basis.
Data Set	A profile data file is considered to be a data set.
Host	The system on which the *PGPROF* tool executes. This will generally be the system where source and executable files reside, and where compilation is performed.
Target Machine	The system on which a profiled program runs. This may or may not be the same system as the host.
GUI	Graphical User Interface. A set of windows, and associated menus, buttons, scrollbars, etc., that can be used to control the profiler and display the profile data.
Combo Box	A combo box is a GUI component consisting of a text field and a list of text items. In its *closed* or default state, it presents a text field of information with a small down arrow icon to its right. When the down arrow icon is left mouse clicked, the box *opens* and presents a list of choices for you to select.

Check Box	A check box is a GUI component consisting of a square or box icon that can be selected by left mouse clicking inside the square. The check box has a selected and an unselected state. In its selected state, a check mark will appear inside the box. The box is empty in its unselected state.
Radio Button	A radio button is a GUI component consisting of a circle icon that can be selected by left mouse clicking inside the circle. The radio button has a selected and an unselected state. In its selected state, the circle is filled in with a solid color, usually black. The circle is empty or unfilled when the button is in its unselected state.

2.1.2 Compilation

The following list shows driver switches that cause profile data collection calls to be inserted and libraries to be linked in the executable file:

–Mprof=func	insert calls to produce a *pgprof.out* file for function (routine) level data.
–Mprof=lines	insert calls to produce a *pgprof.out* file which contains both routine and line level data.
–Mprof=mpi	Link in MPI profile library which intercepts MPI calls in order to record message sizes and to count message sends and receives. Both line-level and function-level profiling are valid with this switch. For example: *–Mprof=mpi,func*
–pg	Enable sample based profiling. Running an executable with this option will produce a *gmon.out* profile data file. When working with sample based profiles, it is important that *PGPROF* knows the name of the executable. By default, *PGPROF* will assume that your executable is called *a.out*. To indicate a different executable, use the *–exe* command line argument or the *Set Executable…* option under the *File* menu in the GUI. See *Section 2.1.4 Profiler Invocation and Initialization* and *2.2 Graphical User Interface* for more information on changing the executable name.

***Note**: Not all profiler options are available in every compiler. Please consult your compiler's user guide for a complete list of profiling options. A list of available profiling options can also be generated with the compiler's –help switch.*

2.1.3 Program Execution

Once a program is compiled for profiling, it needs to be executed. The profiled program is invoked normally, but while running, it collects call counts and/or time data. When the program terminates, a profile data file is generated. Depending on the profiling method used, this data file is called *pgprof.out* or *gmon.out*.

To profile an MPI program, use *mpirun* to execute the program which was compiled and linked with the *–Mprof=mpi* switch. A separate data file is generated for each non-initial MPI process. The *pgprof.out* file acts as the "root" profile data file. It contains profile information on the initial MPI process and points to the separate data files for the rest of the processes involved in the profiling run.

2.1.4 Profiler Invocation and Initialization

Running the *PGPROF* profiler allows the profile data produced during the execution phase to be analyzed.

The *PGPROF* profiler is invoked as follows:

```
    % pgprof [options] [datafile]
```

If invoked without any options or arguments, the *PGPROF* profiler looks for the *pgprof.out* data file and the program source files in the current directory. The program executable name, as specified when the program was run, is usually stored in the profile data file. If all program-related activity occurs in a single directory, the *PGPROF* profiler needs no arguments.

If present, the arguments are interpreted as follows:

datafile	A single datafile name may be specified on the command line. If you are profiling an MPI application, then specifying a datafile that has been generated for a non-initial MPI process is not recommended. Using *PGPROF*, you can inspect profile information on a subset of processes (if you so choose.)*–s* Read commands from standard input. On hosts that have a GUI, this causes *PGPROF* to operate in a non-graphical mode. This is useful if input is being redirected from a file or if the user is remotely logged in to the host system.
–text	Same as –s.

–scale "files(s)"	Compare scalability of *datafile* with one or more files. You can specify a list of files by enclosing the list within quotes and separating each filename with a space. For example: `-scale "one.out two.out"` This example will compare the profiles *one.out* and *two.out* with *datafile* (or pgprof.out by default). If you only specify one file, then you do not need the quotes. If you are working with sample based profiles (e.g., gmon.out), then the executable for each profile should be the same for best results. See also the *Scalability Comparison…* item under the *File* menu (*Section 2.2.3.1 File Menu*).
–I srcdir	Specify the source file search path. The *PGPROF* profiler will always look for a program source file in the current directory first. If it does not find the source file in the current directory, it will consult the search path specified in *srcdir*. The *srcdir* argument is a string of one or more directories. When specifying more than one directory, each directory should be separated with a *path separator.* A path separator is platform dependent; a colon (:) on Linux/Solaris, and a semicolon (;) on Windows. Directories will then be searched in the order specified. When a directory with a filename that matches a source file is found, that directory is used. Below is an example for Linux/Solaris: *-I ../src:STEPS* In the above example, the profiler first looks for source files in the current directory, then it will look in the *../src* directory, followed by the *STEPS* directory. On Windows, you may specify this example the following way: *-I ..\src;STEPS* See also the *Set Source Directory…* item under the *File* menu (*Section 2.2.3.1 File Menu*).
–motif	Use the Motif GUI (Not available on all platforms).
–exe filename	Set the executable to *filename* (default is *a.out*).
-o filename	Same as *–exe*.
–title string	Set the title of the application to *string* (GUI only).
–V	Print version information.

–help	Prints a list of available command line arguments.
–usage	Same as *–help.*

2.1.5 Virtual Timer

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 to run a function. How the timer is incremented and at what frequency depends on the target machine. The timer is read from within the data collection functions and is used to accumulate COST and TIME values for each line, function, and the total execution time. The line level data is based on source lines; however, in some cases, there may be multiple statements on a line and the profiler will show data for each statement.

***Note**: Due to the timing mechanism used by the profiler to gather data, information provided for longer running functions will be more accurate than for functions that only execute for a short percentage of the timer's granularity. Refer to Section 2.1.7 Caveats for more profiler limitations.*

2.1.6 Profile Data

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

BYTES	For HPF and MPI profiles only. This is the number of message bytes sent and received by the function or line.
BYTES RECEIVED	For HPF and MPI profiles only. This is the number of bytes received by the function or line in a data transfer.
BYTES SENT	For HPF and MPI profiles only. This is the number of bytes sent by the function or line.
CALLS	This is the number of times a function is called.
COST	This is 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.
COUNT	This is the number of times a line or function is executed.
COVERAGE	This is the percentage of lines in a function that were executed at least

once.

LINE NUMBER	For line mode, this is the line number for that line. For function mode, this is the line number of the first line of the function.
MESSAGES	For HPF and MPI profiles only. This is the number of messages sent and received by the function or line.
RECEIVES	For HPF and MPI profiles only. This is the number of messages received by the function or line.
SENDS	For HPF and MPI profiles only. This is the number of messages sent by the function or line.
STMT ON LINE	For programs with multiple statements on a line, data is collected and displayed for each statement individually.
TIME	This is only the time spent within the function or executing the line. The TIME does not include time spent in functions called from this function or line. TIME may be displayed in seconds or as a percent of the total time.
TIME PER CALL	This is the TIME for a function divided by the CALLS to that function. TIME PER CALL is displayed in milliseconds.

The data provided by virtual timer profiling-based collection allows you to analyze relationships between functions and between processors.

2.1.7 Caveats

Collecting performance data for programs running on high-speed processors and parallel processors is a difficult task. There is no ideal solution. Since programs running on these processors tend to operate within large internal caches, external hardware cannot be used to monitor their behavior. The only other way to collect data is to alter the program itself, which is how this profiling process works. Unfortunately, it is impossible to do this without affecting the temporal behavior of the program. Every effort has been made to strike a balance between intrusion and utility, and to avoid generating misleading or incomprehensible data. It would, however, be unwise to assume the data is beyond question.

2.1.7.1 Clock Granularity

Many target machines provide a clock resolution of only 20 to 100 ticks per second. Under these circumstances a function must consume at least a few seconds of CPU time to generate meaningful line level times.

2.1.7.2 Optimization

At higher optimization levels, and especially with highly vectorized code, significant code reorganization may have occurred within functions. Most line profilers deal with this problem by disallowing profiling above optimization level 0. The *PGPROF* profiler allows line profiling at any optimization level, and significant effort was expended on associating the line level data with the source in a rational manner and avoiding unnecessary intrusion. Despite this effort, the correlation between source and data may at times appear inconsistent. Compiling at a lower optimization level or examining the assembly language source may be necessary to interpret the data in these cases.

2.2 Graphical User Interface

The *PGPROF* Graphical User Interface (GUI) is invoked using the command `pgprof`. This section describes how to use the profiler with the GUI on systems where it is supported. There may be minor variations in the GUI from host to host, depending on the type of monitor available, the settings for various defaults and the window manager used. Some monitors do not support the color features available with the *PGPROF* GUI. The basic interface across all systems remains the same, as described in this chapter, with the exception of the differences tied to the display characteristics and the window manager used.

There are two major advantages provided by the *PGPROF* GUI.

Source Interaction

The *PGPROF* GUI lets you view the program source for any known routine in the profiler whether or not line level profile data is available simply by selecting the routine name. Since interpreting profile data usually involves correlating the program source and the data, the source interaction provided by the GUI greatly reduces the time spent interpreting data. The GUI allows you to easily compare data on a per processor/thread basis, and identify problem areas of code based on processor/thread execution time differences for routines or lines.

Graphical Display of Data

It is often difficult to visualize the relationships between the various percentages and execution counts. The GUI allows bar graphs to be displayed which graphically represent these relationships. This makes it much easier to locate the 'hot spots' while scrolling through the data for a large program.

2.2.1 The *PGPROF* GUI Layout

After invoking *PGPROF*, the profiler will try to load the profile datafile that you specified on the command line (or a default *pgprof.out*). If no file was found, a file chooser dialog box will appear. Choose a profile datafile from the list or select *Cancel* to choose no file.

When you open a profile datafile, the profiler populates the following areas in the GUI, shown from top to bottom in Figure 2-1:

- *Profile Summary* – Below the "File…Settings…Help" menu bar is the profile summary area. Following the keyword *Profiled:* you will find the executable name, date last modified, and the amount of time consumed by the executable. You will also find the number of processes if the application that you are profiling has more than one process.

- *Profile Entry Combo Box* – Below the *Profile Summary* is the *Profile Entry Combo Box*. The string of characters displayed in this box is known as the *current profile entry*. This entry corresponds to the data highlighted in the profile tables mentioned below. You can change the current entry by entering a new entry or selecting an entry from the combo box. Left mouse click on the down arrow icon to show a list of previously viewed entries to choose from.

- *Navigation Buttons* – Use the left and right arrow buttons, located on the left of the *Profile Entry Combo Box*, to navigate between previously viewed profile entries.

- *Select Combo Box* – This combo box is located to the right of the *Profile Entry Combo Box.* Open the Select Combo Box to refine the criteria for displaying profile entries in the tables mentioned below. By default, the selection is set to *All* profile entries.

- *Top Left Table* – The Top Left Table, located below the *Navigation Buttons,* displays the static profile entry information. This includes filenames, routine names, and line numbers of the profile entries. When viewing line level information, this table will also show the source code if the source files are available. If this table has more than one entry in it, then you will see a column labeled *View*. See the description on the *Bottom Table* below for more information.

- *Top Right Table* – The Top Right Table displays profile summary information for each profile entry. To change what is displayed, select the *Processes* or *View* menus, discussed in *Sections 2.2.3.4 Processes Menu* and *2.2.3.5 View Menu* respectively. If you are viewing profile information at the routine level, and you compiled your program with *–Mprof=lines* or *–pg*, then you can double click the left mouse button to view its line level profile information.

- *Bottom Table* – The *Bottom Table* displays detailed profile information for the *current profile entry*. If you are profiling a multi-process program, then you will see a profile entry for each process in this table. If you are profiling a multi-threaded (or multi-process/multi-threaded) program, then you can view process or thread level profile information. A *Process/Thread Selector* (combo box) will appear in the lower right hand corner when profiling multi-threaded programs. Use this combo box to toggle between thread, process, or process/thread profile information. This combo box is demonstrated in Figure 2-2. This figure shows the Process/Thread Selector in its opened state. Three choices are available: Processes, Threads, Process.Threads.

  The heading in the leftmost column will read *Process(es)* by default. If you are profiling a multi-threaded application, then the heading in the leftmost column will reflect whatever is selected in the *Process/Thread Selector*. When the leftmost column is displaying processes or threads, then each entry will contain integers that represent process/thread IDs. When the leftmost heading is displaying processes and threads (denoted *Process(es).Threads* in the column heading), then each entry is a floating-point number of the form *(Process_ID).(Thread_ID).* Following the process/thread ID, you will see a *filename, routine name,* or *line number* enclosed in parentheses. This provides additional ownership information of the process/thread. It also acts as a minor sort key. See the discussion on *Sort*, *Section 2.2.3.6 Sort Menu*, for more information.

  This table will display process/thread information for the *current profile entry* by default. If you want to view other entries, use the *View* check boxes in the *Top Left Table* to select other entries. The *View* check boxes are demonstrated in Figure 2-9 in *Section 2.2.3.5 View Menu*. This allows you to easily compare more than one process/thread in the *Bottom Table*. When you *Print* the tables to a file or a printer, an entry with a checked *View* box gets printed with each profile entry. Again, this allows for easy comparison of more than one process/thread. See the *Print* option, under the *File* menu, in *Section 2.2.3.1 File Menu* for more information on printing.

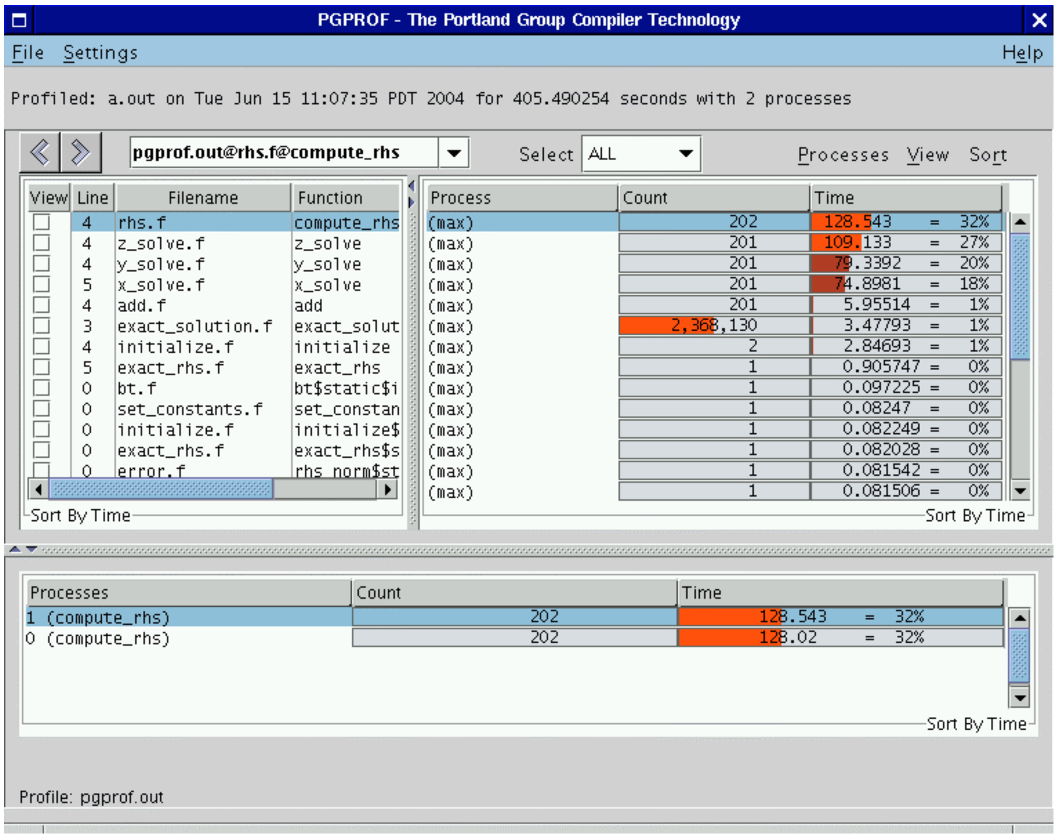
- *Profile Name* – The Profile Name area is located in the lower left hand corner of the GUI. It is preceded with the keyword *Profile:* This area displays the profile filename.

2.2.1.1 GUI Customization

Figure 2-1 shows how the GUI will look when you bring it up for the very first time. The default dimensions of the GUI are approximately `800 x 600`. It can be resized in whatever fashion that your window manager supports. The width of the *Top Left* and *Right* tables can be adjusted using the grey vertical divider located between the two tables. The height of the *Top Left, Right,* and *Bottom* tables can be adjusted using the grey horizontal divider. Both of these dividers can be dragged in the direction shown by arrow icons located on each divider. You can also left mouse click on these arrow icons to quickly "snap" the display in either direction.

After customizing your display, the GUI will remember the size of the main window and the location of each divider for your next *PGPROF* session. If you do not wish to save these settings when you exit *PGPROF,* then uncheck the *Save Settings on Exit* item under the *Settings* menu. We will discuss the *Settings* menu in more detail in S*ection 2.2.3.2 Settings Menu*.

Figure 2-1: Profiler Window



PGPROF - The Portland Group Compiler Technology

File Settings Help

Profiled: a.out on Tue Jun 15 11:07:35 PDT 2004 for 405.490254 seconds with 2 processes

pgprof.out@rhs.f@compute_rhs   Select ALL   Processes View Sort

View	Line	Filename	Function
☐	4	rhs.f	compute_rhs
☐	4	z_solve.f	z_solve
☐	4	y_solve.f	y_solve
☐	5	x_solve.f	x_solve
☐	4	add.f	add
☐	3	exact_solution.f	exact_solut
☐	4	initialize.f	initialize
☐	5	exact_rhs.f	exact_rhs
☐	0	bt.f	bt$static$i
☐	0	set_constants.f	set_constan
☐	0	initialize.f	initialize$
☐	0	exact_rhs.f	exact_rhs$s
☐	0	error.f	rhs_norm$st

Sort By Time

Process	Count	Time
(max)	202	128.543 = 32%
(max)	201	109.133 = 27%
(max)	201	79.3392 = 20%
(max)	201	74.8981 = 18%
(max)	201	5.95514 = 1%
(max)	2,368,130	3.47793 = 1%
(max)	2	2.84693 = 1%
(max)	1	0.905747 = 0%
(max)	1	0.097225 = 0%
(max)	1	0.08247 = 0%
(max)	1	0.082249 = 0%
(max)	1	0.082028 = 0%
(max)	1	0.081542 = 0%
(max)	1	0.081506 = 0%

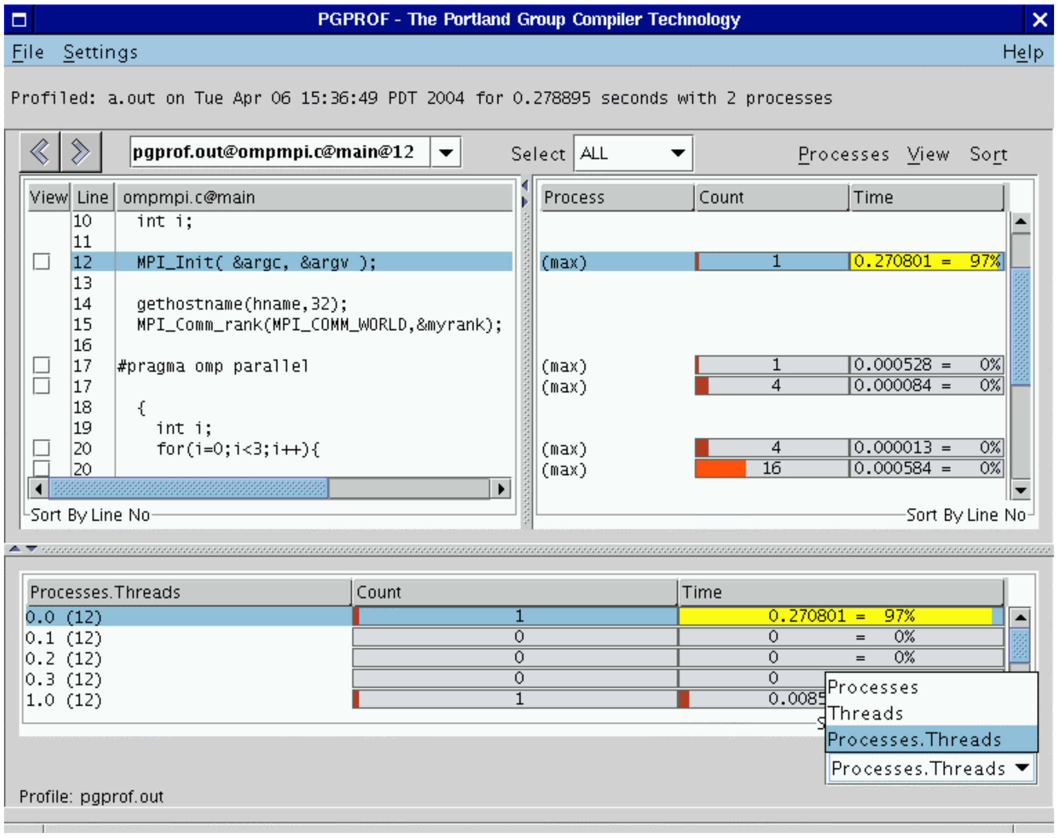
Sort By Time

Processes	Count	Time
1 (compute_rhs)	202	128.543 = 32%
0 (compute_rhs)	202	128.02 = 32%

Sort By Time

Profile: pgprof.out

Figure 2-2: *PGPROF* with Visible Process/Thread Selector



PGPROF - The Portland Group Compiler Technology

File Settings Help

```
Profiled: a.out on Tue Apr 06 15:36:49 PDT 2004 for 0.278895 seconds with 2 processes
```

pgprof.out@ompmpi.c@main@12   Select ALL   Processes View Sort

View	Line	ompmpi.c@main	Process	Count	Time
	10	int i;			
	11				
☐	12	MPI_Init(&argc, &argv);	(max)	1	0.270801 = 97%
	13				
	14	gethostname(hname,32);			
	15	MPI_Comm_rank(MPI_COMM_WORLD,&myrank);			
	16				
☐	17	#pragma omp parallel	(max)	1	0.000528 = 0%
☐	17		(max)	4	0.000084 = 0%
	18	{			
	19	int i;			
☐	20	for(i=0;i<3;i++){	(max)	4	0.000013 = 0%
☐	20		(max)	16	0.000584 = 0%

Sort By Line No   Sort By Line No

Processes.Threads	Count	Time
0.0 (12)	1	0.270801 = 97%
0.1 (12)	0	0 = 0%
0.2 (12)	0	0 = 0%
0.3 (12)	0	0
1.0 (12)	1	0.0085

Profile: pgprof.out

2.2.2 Profile Navigation

The profiler GUI is modeled after a web browser. The *current profile entry* can be thought of as an address, similar to a web page address (or URL). This address is displayed in the *Profile Entry Combo Box;* introduced in *Section 2.2.1 The PGPROF GUI Layout*. The address format follows:

```
(profileFilename)[@sourceFilename [@routineName [@lineNumber]]]
```

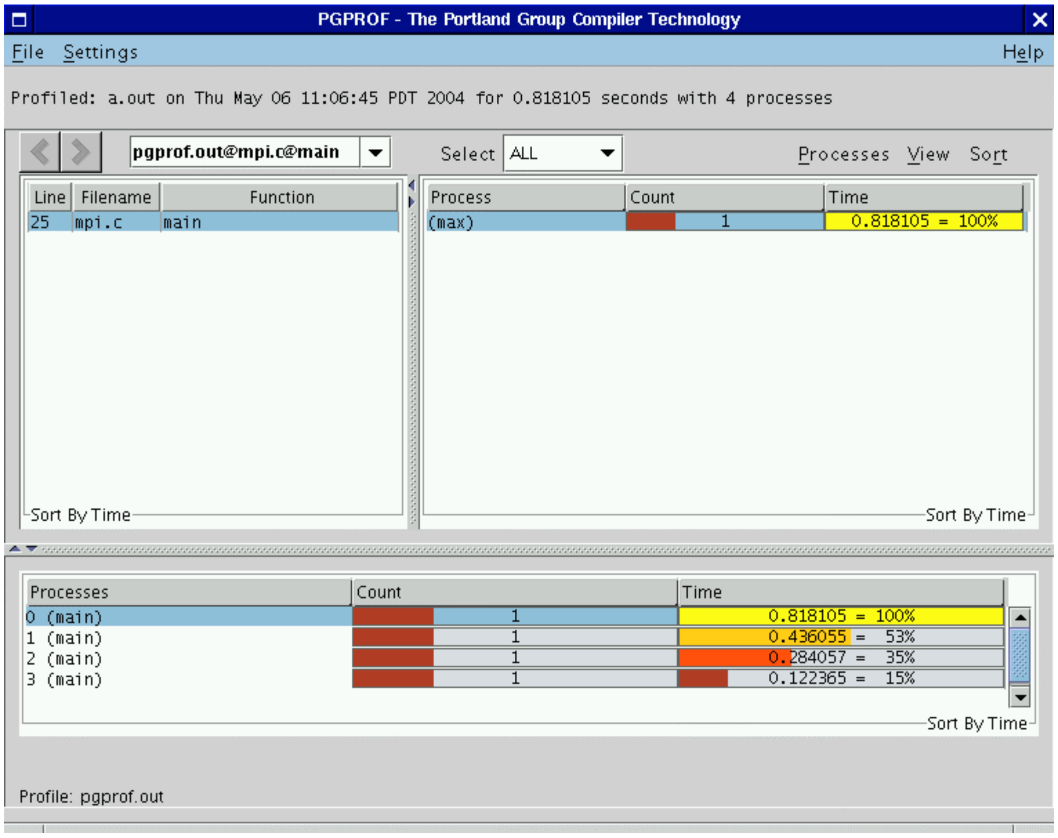
The only required argument of the address is the profile datafile (e.g., *pgprof.out, gmon.out,* etc.).

Each additional argument is separated by an '@'. For example, consider Figure 2-3 where we brought up a profile of an application with one routine called `main`. When you first bring up a profile, the first entry in the Top Left and Right tables is selected (highlighted) by default. The Profile Entry Combo Box reflects the selected entry by displaying its address. In this case, the Profile Entry Combo Box reads `pgprof.out.mpi.c@main`. This says that the *current profile entry* is the `main` routine located in file `mpi.c`. You can enter a different address in the *Profile Entry Combo Box* using the above address format. As mentioned in *Section 2.2.1 The PGPROF GUI Layout*, previously viewed profile entries can be selected with the *Profile Entry Combo Box* or with the *Navigation Buttons*.

To change the *current profile entry,* left mouse click on a new entry in the *Right Table.* You can also click on an entry in the *Left Table*, but there must be a corresponding entry in the *Right Table.* If you double click the left mouse button on a profile entry, you will *dive* into the selected profile entry. For example, let us assume that we compiled our program with *–Mprof=lines* and the *current profile entry* is *`pgprof.out@mpi.c@main`,* as shown in Figure 2-3. If you double click on the highlighted entry in the *Right Table,* the profiler will jump or *dive* to `main`'s line level information. Figure 2-4 shows this example after double clicking on `main`.

Diving into the profile, by double clicking, works at higher levels of profiling too. For example, if the *current profile entry* is `pgprof.out,` then double clicking on `pgprof.out` will show you a list of profiled files and their profile information. Double clicking on a file from this list will then take you to the *routine level profiling* information for that file, etc.

Figure 2-3: Example Routine Level Profile



Profiled: a.out on Thu May 06 11:06:45 PDT 2004 for 0.818105 seconds with 4 processes

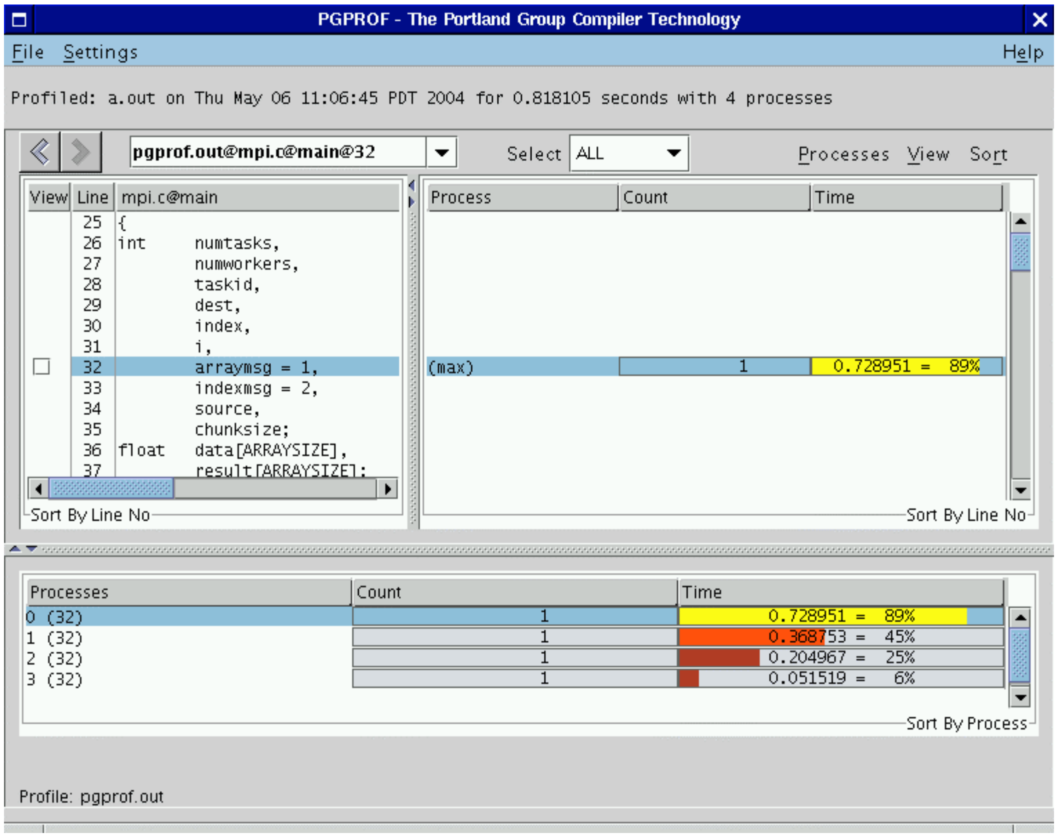
Line	Filename	Function
25	mpi.c	main

Process	Count	Time
(max)	1	0.818105 = 100%

Processes	Count	Time
0 (main)	1	0.818105 = 100%
1 (main)	1	0.436055 = 53%
2 (main)	1	0.284057 = 35%
3 (main)	1	0.122365 = 15%

Profile: pgprof.out

Figure 2-4: Example Line Level Profile



```
Profiled: a.out on Thu May 06 11:06:45 PDT 2004 for 0.818105 seconds with 4 processes
```

pgprof.out@mpi.c@main@32

```
View Line mpi.c@main
     25   {
     26   int      numtasks,
     27            numworkers,
     28            taskid,
     29            dest,
     30            index,
     31            i,
     32            arraymsg = 1,
     33            indexmsg = 2,
     34            source,
     35            chunksize;
     36   float    data[ARRAYSIZE],
     37            result[ARRAYSIZE];
```

Sort By Line No

Process	Count	Time
(max)	1	0.728951 = 89%

Sort By Line No

Processes	Count	Time
0 (32)	1	0.728951 = 89%
1 (32)	1	0.368753 = 45%
2 (32)	1	0.204967 = 25%
3 (32)	1	0.051519 = 6%

Sort By Process

Profile: pgprof.out

2.2.3 *PGPROF* Menus

As shown in Figures 2-1 through 2-4, there are five menus in the GUI: *File, Settings, Help, Processes, View,* and *Sort.* Sections *2.2.3.1 File Menu* through *2.2.3.6 Sort Menu* describe each menu in detail. Keystrokes, located next to menu items, represent keyboard short cuts for that particular item.

2.2.3.1 File Menu

The *File* menu contains the following items:

- *New Window (control N)* – Select this option to create a copy of the current profiler window on your screen.

- *Open Profile…* - Select this option to open another profile. After selecting this menu item, locate a new profile data file in a file chooser dialog box. Select the new file in the dialog by double clicking the left mouse button on it. A new profile window will appear with the selected profile. NOTE: You must first set the name of the profile's executable before opening a sample based profile (e.g., *gmon.out*). See the *Set Executable…* option below for more details.

- *Set Executable…* - Select this option to choose the executable that you are profiling. After selecting this menu item, locate the executable that you profiled in a file chooser dialog box. Select the executable by double clicking the left mouse button on it. When working with sample based profiles (e.g., *gmon.out*), the executable that you choose must match the executable that you profiled. By default, the profiler assumes that your executable is called `a.out`.

- *Set Source Directory…* - Select this option to set the location of the profiled executable's source files. The profiler will present you with a text field in a dialog box. Enter one or more directories in this text field. Each directory is separated by a *path separator*. The path separator is platform dependent; a colon (:) on Linux/Solaris and a semicolon (;) on Windows. These directories act as a search path when the profiler cannot find a source file in the current directory. Below is an example:

  ../src:STEPS

  After entering the above string into the dialog box, the profiler will first search for source files in the current directory, then in the *../src* directory, and finally in the *STEPS* directory. You can also set the directory through the *–I* command line option explained in *Section 2.1.4 Profiler Invocation and Initialization*. The same example on Windows follows:

  ..\src;STEPS

- *Scalability Comparison…* - Select this option to open another profile for scalability comparison. Follow the same directions for the *Open Profile…* option mentioned above. The new profile will contain a *Scale* column in its *Top Right* table. You can also open one or more profiles for scalability comparison through the *–scale* command line option explained in *Section 2.1.4 Profiler Invocation and Initialization.* . See also *Section 2.2.5 Scalability Comparison* for more information on scalability.

- *Print…* - The print option allows you to make a hard copy of the current profile data. The profiler will combine the data in all three profile tables and send the output to a printer. A printer dialog box will appear. You can select a printer from the *Print Service Name* combo box. Click on the *Print To File* check box to send the output to a file. Other print options may be available. However, they are dependent on your printer and the Java Runtime Environment (JRE).

- *Print to File…* - Same as *Print…* option except the output goes to a file. After selecting this menu item, a *save file* dialog box will appear. Enter or choose an output file in the dialog box. Click *Cancel* to abort the print operation.

2.2.3.2 Settings Menu

The *Settings* menu contains the following items:

- *Bar Chart Colors…* - This menu option will open a color chooser dialog box and a bar chart preview panel (Figure 2-5). There are four bar chart colors based on the percentage filled and three bar chart attributes. The *Filled Text Color* attribute represents the text color inside the filled portion of the bar chart. The *Unfilled Text Color* attribute represents the text color outside the filled portion of the bar chart. The *Background Color* attribute represents the color of the unfilled portion of the bar chart. Table 2-1 lists the default colors.

  To modify a bar chart or attribute color, click on its radio button. Next, choose a color from the *Swatches, HSB,* or *RGB* pane. Press the left mouse button on the *OK* button to accept the changes and close the dialog box. Click *Reset* to reset the selected bar chart or attribute to its previously selected color. Closing the window will also accept the changes. The GUI will remember your color selections for subsequent runs of *PGPROF* unless you uncheck the *Save Settings on Exit* box (see discussion on this option below).

- *Font…* - This menu option opens the *Font Chooser* dialog box (Figure 2-6). You can choose a new font from a list of fonts in this dialog's top combo box. You can also choose a new font size from a list of sizes in this dialog's bottom combo box. The font is previewed in the *Sample Text* pane to the left. The font does not change until you left mouse click *OK*. Click *Cancel* or close the dialog box to abort any changes. The default font is *monospace* size *12*.

- *Show Tool Tips* – Select this check box to enable tool tips. Tool tips are small temporary messages that pop-up when you position the mouse pointer over a component in the GUI. They provide additional information on what a particular component does. Unselect this check box to turn them off.

- *Restore Factory Settings…* - This option allows you to restore the default look and feel of the GUI. The GUI will look similar to Figure 2-1 after selecting this option.

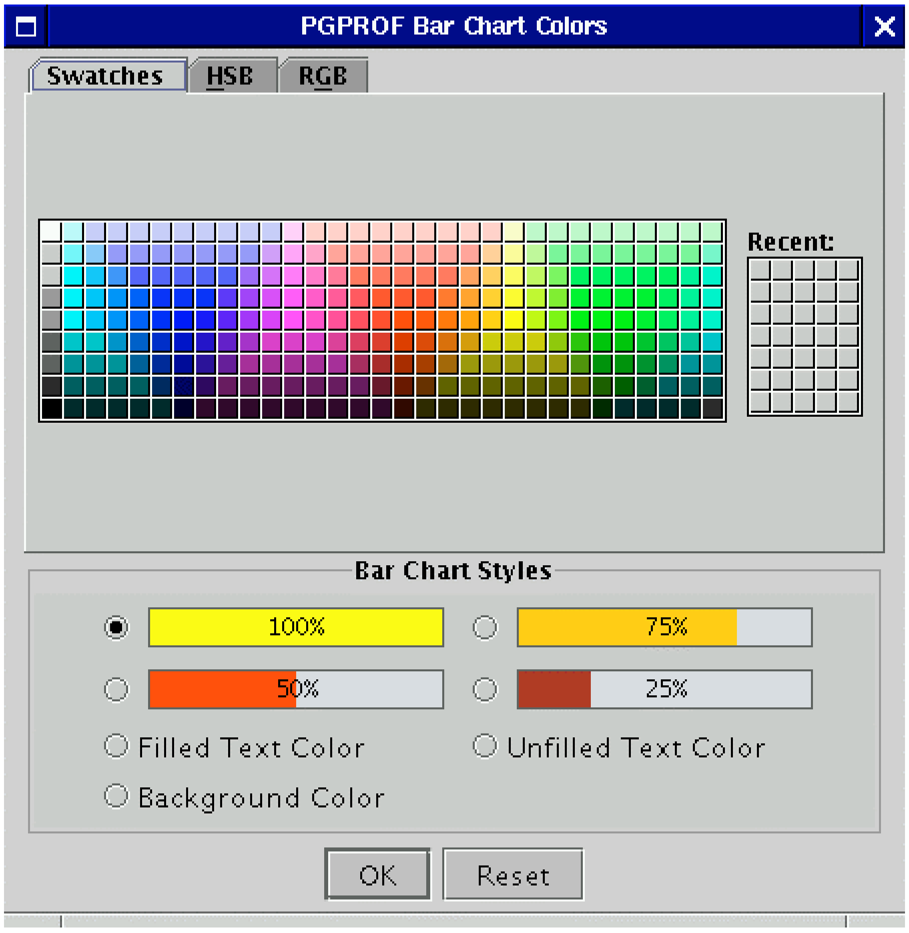
- *Restore Saved Settings…* - This option allows you to restore the look and feel of the GUI to the previously saved settings. See the *Save Settings on Exit* option for more information.

- *Save Settings on Exit* – When this check box is enabled, the GUI will save the current look and feel settings when you exit. These settings include the size of the main window, position of the horizontal/vertical dividers, the bar chart colors, the selected font, your tools tips preference, and the options selected in the *View* menu. When you start the GUI again on the same host machine, these saved settings are used. If you do not want to save these settings on exit, then uncheck this box. Unchecking this box disables the saving of settings for the current session only.

Table 2-1: Default Bar Chart Colors

Bar Chart Style/Attribute	Default Color
1-25%	Brown
51%-75%	Orange
76%-100%	Yellow
Filled Text Color	Black
Unfilled Text Color	Black
Background Color	Grey

Figure 2-5: Bar Chart Color Dialog Box



PGPROF Bar Chart Colors

Swatches | HSB | RGB

Recent:

Bar Chart Styles

[● 100%] ○ 75%

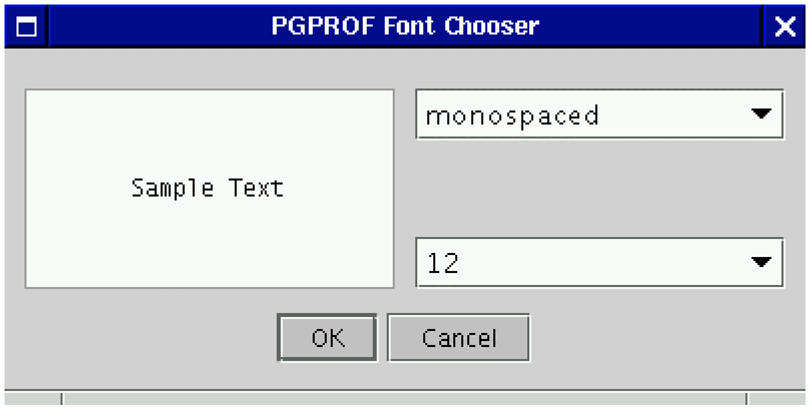
○ 50% ○ 25%

○ Filled Text Color ○ Unfilled Text Color

○ Background Color

OK Reset

Figure 2-6: Font Chooser Dialog Box



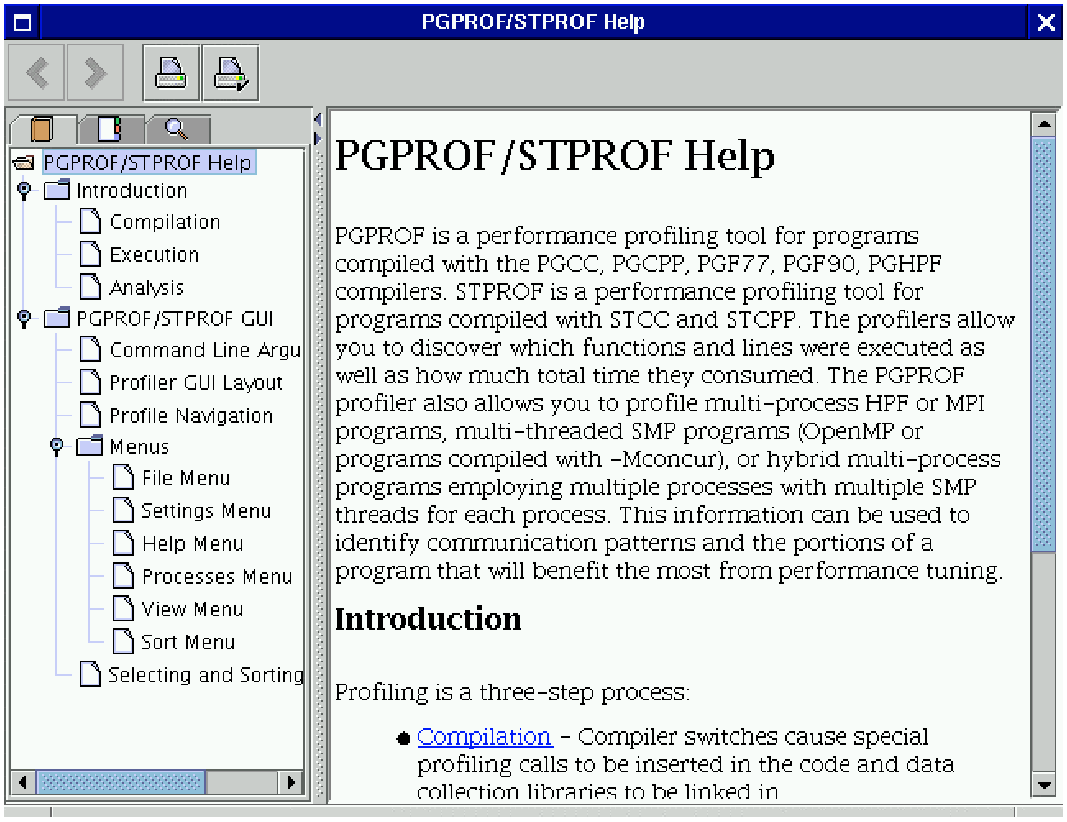
2.2.3.3 Help Menu

The *Help* menu contains the following items:

- *PGPROF Help…* - This option starts up *PGPROF's* integrated help utility as shown in Figure 2-7. The help utility includes an abridged version of this documentation. To find a help topic, use one of the follow tabs in the left panel: The "book" tab presents a table of contents, the "index" tab presents an index of commands, and the "magnifying glass" tab presents a search engine. Each help page (displayed on the right) may contain hyperlinks (denoted in underlined blue) to terms referenced elsewhere in the help engine. Use the arrow buttons to navigate between visited pages. The printer buttons allow you to print out the current help page.

- *About PGPROF…* - This option opens a dialog box with version and contact information for *PGPROF*.

Figure 2-7: *PGPROF* Help



PGPROF/STPROF Help

PGPROF is a performance profiling tool for programs compiled with the PGCC, PGCPP, PGF77, PGF90, PGHPF compilers. STPROF is a performance profiling tool for programs compiled with STCC and STCPP. The profilers allow you to discover which functions and lines were executed as well as how much total time they consumed. The PGPROF profiler also allows you to profile multi-process HPF or MPI programs, multi-threaded SMP programs (OpenMP or programs compiled with -Mconcur), or hybrid multi-process programs employing multiple processes with multiple SMP threads for each process. This information can be used to identify communication patterns and the portions of a program that will benefit the most from performance tuning.

Introduction

Profiling is a three-step process:

- Compilation - Compiler switches cause special profiling calls to be inserted in the code and data collection libraries to be linked in

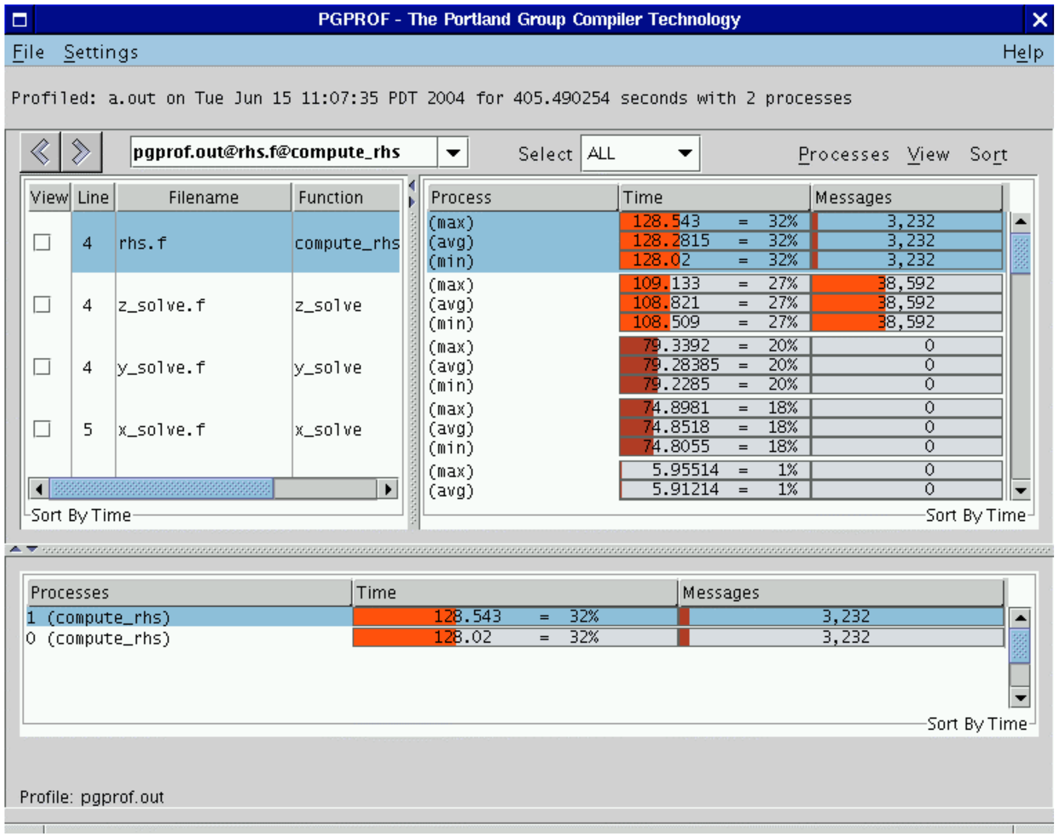
2.2.3.4 Processes Menu

The Processes menu is enabled for multi-process programs only. This menu contains three check boxes: *Min, Max,* and *Avg.* They represent the minimum process value, maximum process value, and average process value respectively. By default *Max*, is selected.

When *Max* is selected, the highest value for any profile data in the *Top Right Table* is reported. For example, when reporting *Time*, the longest time for each profile entry gets reported when *Max* is selected. When the *Min* process is selected, the lowest value for any profile data is reported in the *Right Table*. *AVG* reports the average value between all of the processes. You can select none, some, or all of these check boxes. When no check boxes are selected, the *Top, Left* and *Right Tables* are empty. If the *Process* check box under the *View* menu is selected, then each row of data in the *Right Table* is labeled *max, avg,* and *min* respectively.

Figure 2-8 illustrates *max, avg,* and *min* with the *Process* check box enabled.

Figure 2-8: *PGPROF* with *Max, Avg, Min* rows



Profiled: a.out on Tue Jun 15 11:07:35 PDT 2004 for 405.490254 seconds with 2 processes

View	Line	Filename	Function
☐	4	rhs.f	compute_rhs
☐	4	z_solve.f	z_solve
☐	4	y_solve.f	y_solve
☐	5	x_solve.f	x_solve

Process	Time	Messages
(max)	128.543 = 32%	3,232
(avg)	128.2815 = 32%	3,232
(min)	128.02 = 32%	3,232
(max)	109.133 = 27%	38,592
(avg)	108.821 = 27%	38,592
(min)	108.509 = 27%	38,592
(max)	79.3392 = 20%	0
(avg)	79.28385 = 20%	0
(min)	79.2285 = 20%	0
(max)	74.8981 = 18%	0
(avg)	74.8518 = 18%	0
(min)	74.8055 = 18%	0
(max)	5.95514 = 1%	0
(avg)	5.91214 = 1%	0

Processes	Time	Messages
1 (compute_rhs)	128.543 = 32%	3,232
0 (compute_rhs)	128.02 = 32%	3,232

Profile: pgprof.out

2.2.3.5 View Menu

The *View* menu allows you to select which columns of data that you wish to view in the *Top Left, Top Right,* and *Bottom* tables. This selection also affects the way that tables are printed to a file and a printer (see *Print* in *Section 2.2.3.1File Menu*).

The following lists *View* menu items and their definition. Please note that not all items may be available for a given profile.

- *Count* – Enables the *Count* column in the *Top Right* and *Bottom* tables. *Count* is associated with the number of times this profile entry has been visited during execution of the program. For function level profiling, *Count* is the number of times the routine was called. For line level profiling, *Count* is the number of times a profiled source line was executed.

- *Time* – Enables the *Time* column in the *Top Right* and *Bottom* tables. The time column displays the time spent in a routine (for function level profiling) or at a profiled source line (for line level profiling).

- *Cost* – Enables the *Cost* column in the *Top Right* and *Bottom* tables. Cost is defined as the execution time spent in this routine and all of the routines that it called. The column will contain all zeros if cost information is not available for a given profile.

- *Coverage* – Enables the *Cover* column in the *Top Right* and *Bottom* tables. Coverage is defined as the number of lines in a profile entry that were executed. By definition, a profiled source line will always have a coverage of 1. A routine's coverage is equal to the sum of all its source line coverages. Coverage is only available for line level profiling. The column will contain all zeros if coverage information is not available for a given profile.

- *Messages* – Enables the message count columns in the *Top Right* and *Bottom* tables. Use this menu item to display total (MPI) messages sent and received for each profile entry. This menu item contains *Message Sends* and *Message Receives* submenus for separately displaying the sends and receives in the *Top Right* and *Bottom* tables. The message count columns will contain all zeros if no messages were sent or received for a given profile.

- *Bytes* – Same as *Messages* except message byte totals are displayed instead of counts.

- *Scalability* – Enables the *Scale* column in the *Top Right* table. Scalability is used to measure the *linear speed-up* or *slow-down* of two profiles. This menu contains two check boxes: *Metric* and *Bar Chart.* When *Metric* is selected, the raw Scalability value is displayed. When *Bar Chart* is selected, a graphical representation of the metric is displayed. Scalability is discussed in *Section 2.2.5 Scalability Comparison*.

- *Processes…(control P)* - This menu item is enabled when you are profiling an application with more than one process. Use the *Processes* menu item to select individual processes for viewing in the *Bottom* table. When you select this item, a dialog box will appear with a text field. You can enter individual processes or a range of processes for viewing in this text field. Individual processes must be separated with a comma.

A range of processes must be entered in the form: `[start]-[end]`; where *start* represents the first process of the range and *end* represents the last process of the range. For example:

```
0,2-16,31
```

This tells the profiler that you want to view process 0, process 2 through 16, and process 31. The changes that you make to *Processes* remain active until you change them again or exit the profiler. Leave the text field blank to view all of the processes in the *Bottom* table.

- *Threads… (control T)* - Same as *Processes...* except it selects the threads rather than the processes viewed in the *Bottom* table.

- *Filename* – Enables the *Filename* column in the *Top Left* table.

- *Line Number* – Enables the *Line* column in the *Top Left* table.

- *Name* – Enables the *Function* (routine) name column in the *Top Left* table when viewing function level profiling.

- *Source* – Enables the *Source* column in the *Top Left* table when viewing line level profiles. If the source code is available, this column will display the source lines for the current routine. Otherwise, this column will be blank.

- *Statement Number* – Enables the *Stmt #* column in the *Top Right* table. Sometimes more than one statement is profiled for a given source line number. One example of this is a "C" *for* statement. The profiler will assign a separate row for each substatement in the *Top Left* and *Right* tables. In line level profiling, you will see duplicate line numbers in the *Line* column. Each substatement is assigned a statement number starting at 0. Any substatement numbered one or higher will have a '.' and their statement number tacked onto the end of their profile address. For example, consider Figure 2-9. Source lines 7 and 15 both have multiple profile entries. As shown in the *Profile Entry Combo Box*, the second entry for line 7 has the following address:

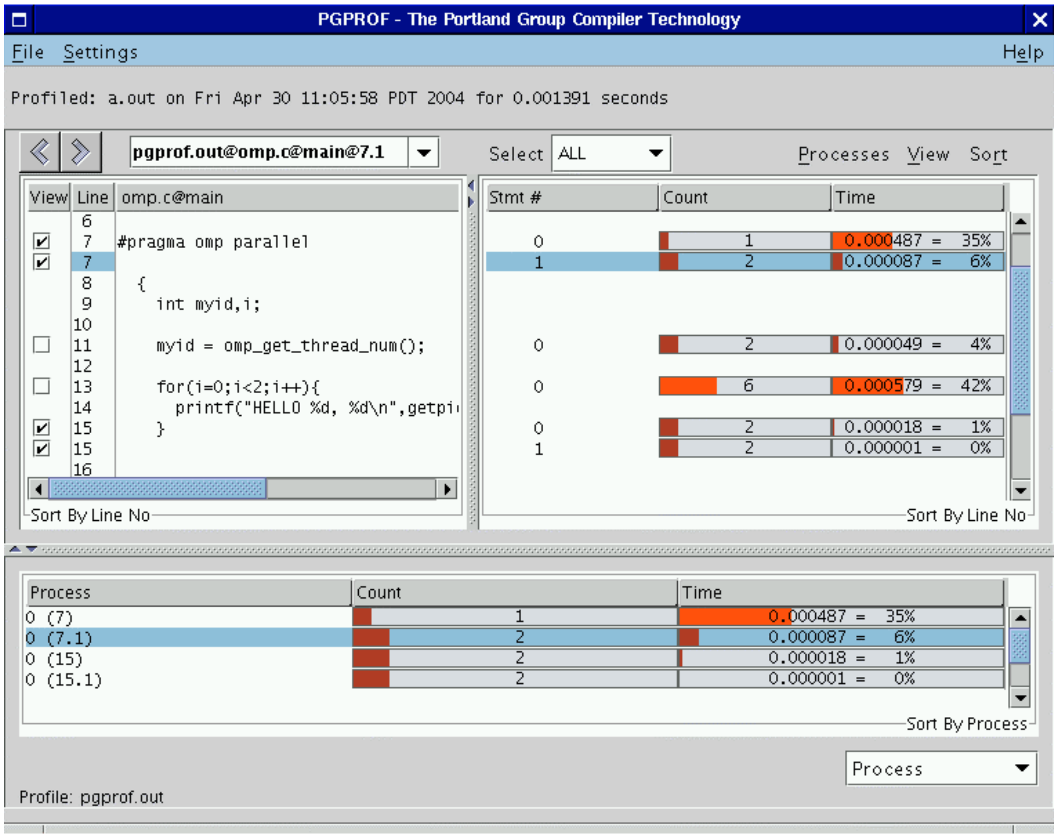
  pgprof.out@omp.c@main@7.1

  This line number convention is also reflected in the *Bottom* table of Figure 2-9 where the line number is enclosed in parentheses.

- *Process* – The menu option is enabled when more than one process was profiled for a given program. When you select its check box, a column labeled *Process* is displayed in the *Top Right* table. The values for the *Process* column depend on whatever was enabled in the *Processes* menu discussed in *Section 2.2.3.4 Processes Menu*.

- *Event1 – Event4* – If your installation supports hardware event counters, then up to four unique events can be displayed in the *Top Right* and *Bottom* tables. If you profiled with hardware event counters, then one to four event menu items will be enabled and have their name changed to their particular event. Each event can exist on some or all of the executing threads in the profiled application.

Figure 2-9: Source Lines with Multiple Profile Entries



```
Profiled: a.out on Fri Apr 30 11:05:58 PDT 2004 for 0.001391 seconds
```

View	Line	omp.c@main
	6	
☑	7	`#pragma omp parallel`
☑	7	
	8	`{`
	9	`int myid,i;`
	10	
☐	11	`myid = omp_get_thread_num();`
	12	
☐	13	`for(i=0;i<2;i++){`
	14	`printf("HELLO %d, %d\n",getpi`
☑	15	`}`
☑	15	
	16	

Stmt #	Count	Time
0	1	0.000487 = 35%
1	2	0.000087 = 6%
0	2	0.000049 = 4%
0	6	0.000579 = 42%
0	2	0.000018 = 1%
1	2	0.000001 = 0%

Process	Count	Time
0 (7)	1	0.000487 = 35%
0 (7.1)	2	0.000087 = 6%
0 (15)	2	0.000018 = 1%
0 (15.1)	2	0.000001 = 0%

The submenus *Count, Time, Cost, Coverage, Messages, Bytes,* and *Event1* through *Event4* contain three check boxes for selecting how the data is presented in each column. The first check box enables a raw number to be displayed. The second check box enables a percentage. The third check box is a bar chart.

When you select percentage, you will see a percentage based on the global value of the selected statistic. For example, in Figure 2-9, line 13 consumed `0.000579` seconds, or `42%` of the total time of `0.001391` seconds.

When you select the bar chart, you will see a graphical representation of the percentage. The colors are based on this percentage. For a list of default colors and their respective percentages, see the *Bar Chart Colors* option under the *Settings* menu (*Section 2.2.3.2 Settings Menu*).

2.2.3.6 Sort Menu

The sort menu allows you to alter the order in which profile entries appear in the *Top Left, Top Right,* and *Bottom* tables. The current sort order is displayed at the bottom of each table. In Figure 2-9, the tables have a "Sort by" clause followed with "Line No" or "Process". This indicates the sort order is by source line number or by process number respectively. In *PGPROF,* the default sort order is by *Time* for function level profiling and by *Line No* (source line number) for line level profiling. The sort is performed in descending order, from highest to lowest value, except when sorting by filename, function name, or line number. Filename, function name, and line number sorting is performed in ascending order; lowest to highest value. Sorting is explained in greater detail in *Section 2.2.4 Selecting and Sorting Profile Data*.

2.2.4 Selecting and Sorting Profile Data

Selecting and sorting affects what profile data is displayed and how it is displayed in *PGPROF*'s *Top Left, Top, Right*, and *Bottom* tables. The *Sort* menu, explained in *Section 2.2.3.6 Sort Menu*, allows you to change the sort order. The *Select Combo Box*, introduced in *Section 2.2.1 The PGPROF GUI Layout*, allows you to select which profile entries are displayed based on certain criteria.

2.2.4.1 Selecting Profile Data

By default, the profiler *selects* all profile entries for display in the *Top Left* and *Right* tables. To change the selection criteria, left mouse click on the *Select Combo Box* next to the *Select* label.

The following options are available:

- *All* – Default. Display all profile entries.
- *Count* – Select entries based on a count criteria. When you select this entry, an additional text field will appear with up and down arrow keys. Use the up and down arrow keys to increase the minimum value for count that you wish to see. You can also directly input your desired minimum in the text field. Profile entries with counts greater than this number will be displayed in the tables. In Figure 2-10, we are selecting all routines that have a count value of 2 or higher.

- *Coverage* – Select entries based on coverage. This option is similar to *Count* except the input value is a percentage. Profile entries with coverage that exceed the input percentage are displayed in the tables.

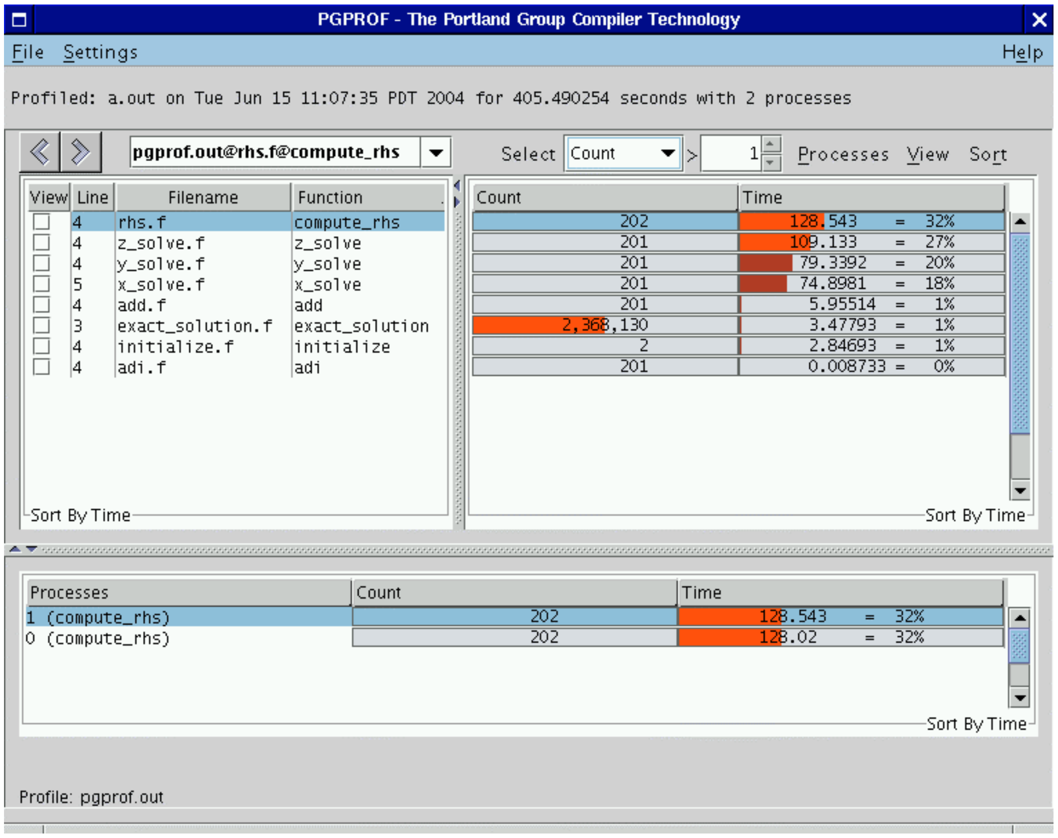
- *Profiled* – Select all entries in the *Top Left* table that have a corresponding entry in the *Top Right* table. See the discussion below for more information.

- *Time* – Same as *Coverage* except the criteria is based on percent of Time a profile entry consumes rather than *Coverage.*

- *Unprofiled* – Select all entries in the *Top Left* table that do not have a corresponding entry in the *Top Right* table. See the discussion below for more information.

When you select *Profiled* entries, you are selecting profile entries that have a corresponding entry in both the *Top Left* and *Right* tables. A profile entry may be listed in the *Top Left* table but not in the *Top Right* table. In this case, the entry is an *Unprofiled* entry. A *Profiled* entry is a point in the program in which profile information was collected. Depending on the profiling method used, this could be at the end of a *basic block* (e.g., *-Mprof=(func|line|mpi)* instrumented profiles*)* or when the profiling mechanism saved its state (e.g., *-pg* sample based profiles).

Figure 2-10: Selecting Profile Entries with Counts Greater Than 1


PGPROF - The Portland Group Compiler Technology
File Settings Help
pgprof.out@rhs.f@compute_rhs
Select Count > 1 Processes View Sort
Sort By Time
Profile: pgprof.out

Profiled: a.out on Tue Jun 15 11:07:35 PDT 2004 for 405.490254 seconds with 2 processes

View	Line	Filename	Function	Count	Time
☐	4	rhs.f	compute_rhs	202	128.543 = 32%
☐	4	z_solve.f	z_solve	201	109.133 = 27%
☐	4	y_solve.f	y_solve	201	79.3392 = 20%
☐	5	x_solve.f	x_solve	201	74.8981 = 18%
☐	4	add.f	add	201	5.95514 = 1%
☐	3	exact_solution.f	exact_solution	2,368,130	3.47793 = 1%
☐	4	initialize.f	initialize	2	2.84693 = 1%
☐	4	adi.f	adi	201	0.008733 = 0%

Processes	Count	Time
1 (compute_rhs)	202	128.543 = 32%
0 (compute_rhs)	202	128.02 = 32%

2.2.4.2 Sorting Profile Data

To change the sort order, select the *Sort* menu and left mouse click on the radio button next to the item that you wish to sort. For a definition of a particular item in this menu, see its description under the *View* menu in *Section 2.2.3.5 View Menu*.

You may notice that the *Bottom* table will display one of the following messages when sorting by *Filename, Name,* or *Line Number*:

- *Sort By Process*
- *Sort By Processes*

- *Sort By Threads*
- *Sort By Process.Threads*
- *Sort By Processes.Threads*

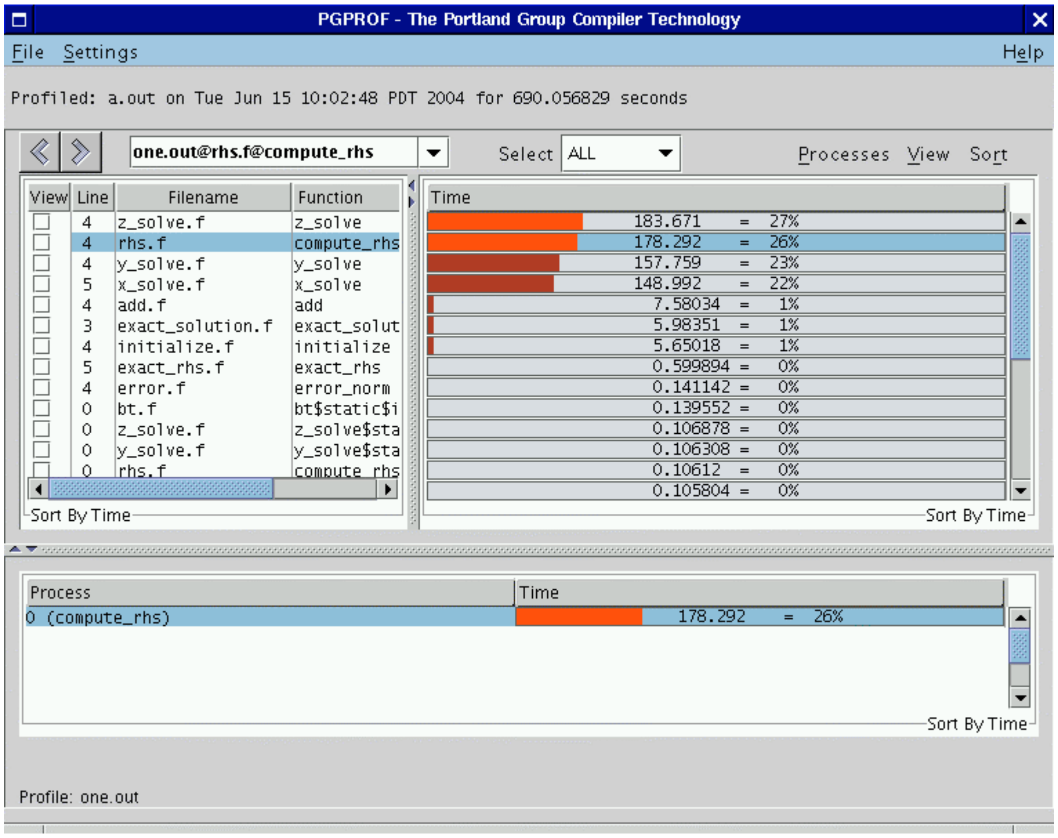
When you see one of these messages in the *Bottom* table, then the profiler is treating the process/thread number as the major sort key and the *Filename, Name,* or *Line Number* as the minor sort key. This allows you to easily compare two different profile entries with the same process/thread number. Use the check boxes under the *View* column in the *Top Left* table to compare more than one profile entry in the *Bottom* table. This is demonstrated in Figure 2-9.

2.2.5 Scalability Comparison

PGPROF has a *Scalability Comparison* feature that allows you to measure *linear speed-up* or *slow-down* between multiple executions of an application. You can measure scalability between executions with a varying number of processes or threads. To use scalability comparison, you should first generate two or more profiles for a given application. For best results, compare profiles from the same application using the same input data. The number of processes and/or threads used in each execution can be different. After generating two or more profiles, load one of them with *PGPROF.* Select the *Scalability Comparison* item under the *File* menu and choose another profile for comparison (*Section 2.2.3.1 File Menu*). A new profiler window will appear with a column called *Scale* in its *Top Right* table (*Section* 2.2.3.5 View Menu).

Figure 2-11 shows a profile of an application that was run with one process. Figure 2-12 shows a profile of the same application run with two processes. The profile in Figure 2-12 also has a *Scale* column in its *Top Right* table. Each profile entry that has timing information has a *Scale* value. The scale value measures the *linear speed-up* or *slow-down* for these entries across profiles. A scale value of zero indicates that adding processes or threads did not improve the execution time. A positive value means the time improved by that factor. A negative value means that the time slowed down by that factor.

Figure 2-11: Profile of an Application Run with 1 Process



Profiled: a.out on Tue Jun 15 10:02:48 PDT 2004 for 690.056829 seconds

one.out@rhs.f@compute_rhs

View	Line	Filename	Function	Time
☐	4	z_solve.f	z_solve	183.671 = 27%
☐	4	rhs.f	compute_rhs	178.292 = 26%
☐	4	y_solve.f	y_solve	157.759 = 23%
☐	5	x_solve.f	x_solve	148.992 = 22%
☐	4	add.f	add	7.58034 = 1%
☐	3	exact_solution.f	exact_solut	5.98351 = 1%
☐	4	initialize.f	initialize	5.65018 = 1%
☐	5	exact_rhs.f	exact_rhs	0.599894 = 0%
☐	4	error.f	error_norm	0.141142 = 0%
☐	0	bt.f	bt$static$i	0.139552 = 0%
☐	0	z_solve.f	z_solve$sta	0.106878 = 0%
☐	0	y_solve.f	y_solve$sta	0.106308 = 0%
☐	0	rhs.f	compute_rhs	0.10612 = 0%
				0.105804 = 0%

Sort By Time

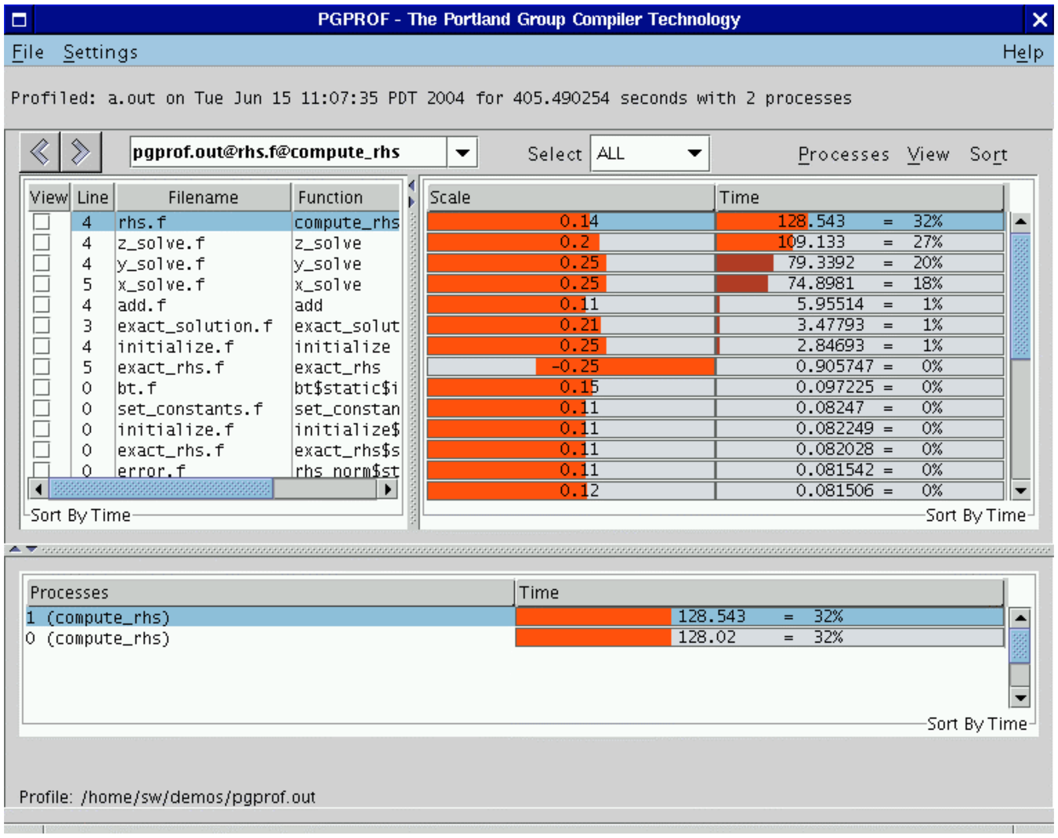
Sort By Time

Process	Time
0 (compute_rhs)	178.292 = 26%

Sort By Time

Profile: one.out

Figure 2-12: Profile with Visible Scale Column



PGPROF - The Portland Group Compiler Technology

File Settings Help

```
Profiled: a.out on Tue Jun 15 11:07:35 PDT 2004 for 405.490254 seconds with 2 processes
```

pgprof.out@rhs.f@compute_rhs   Select ALL   Processes View Sort

View	Line	Filename	Function	Scale	Time
☐	4	rhs.f	compute_rhs	0.14	128.543 = 32%
☐	4	z_solve.f	z_solve	0.2	109.133 = 27%
☐	4	y_solve.f	y_solve	0.25	79.3392 = 20%
☐	5	x_solve.f	x_solve	0.25	74.8981 = 18%
☐	4	add.f	add	0.11	5.95514 = 1%
☐	3	exact_solution.f	exact_solut	0.21	3.47793 = 1%
☐	4	initialize.f	initialize	0.25	2.84693 = 1%
☐	5	exact_rhs.f	exact_rhs	-0.25	0.905747 = 0%
☐	0	bt.f	bt$static$i	0.15	0.097225 = 0%
☐	0	set_constants.f	set_constan	0.11	0.08247 = 0%
☐	0	initialize.f	initialize$	0.11	0.082249 = 0%
☐	0	exact_rhs.f	exact_rhs$s	0.11	0.082028 = 0%
☐	0	error.f	rhs_norm$st	0.11	0.081542 = 0%
				0.12	0.081506 = 0%

Sort By Time   Sort By Time

Processes	Time
1 (compute_rhs)	128.543 = 32%
0 (compute_rhs)	128.02 = 32%

Sort By Time

Profile: /home/sw/demos/pgprof.out

Below is *PGPROF*'s formula for computing Scalability:

P_1 = number of processes or threads used in first run of application

P_2 = number of processes or threads used in second run of application

where $P_2 \geq P_1$

$Time_1$ = Execution time using P_1 processes or threads

$Time_2$ = Execution time using P_2 processes or threads

$Scalability = [(Time_1 - Time_2) \div Time_1] \times (P_1 \div P_2)$

By definition, *perfect linear speed-up* will give you a scalability value of one. Anything greater than one, gives you *super speed-up*. Similar negative values indicate *linear slow-down* and *super slow-down* respectively. Bar charts in the *Scale* column show positive values with bars extending from left to right and negative values with bars extending from right to left (Figure 2-12).

If you see a question mark ('?') in the *Scale* column, then *PGPROF* is unable to perform the scalability comparison for this profile entry. This may happen if the two profiles do not share the same executable or input data.

2.3 Command Language

The interface for non-GUI versions of the *PGPROF* profiler is a simple command language. This command language is available in GUI versions of the profiler using the *–s* or –text option. The language is composed of commands and arguments separated by white space. A `pgprof>` prompt is issued unless input is being redirected.

2.3.1 Command Usage

This section describes the profiler's command set. Command names are printed in bold and may be abbreviated as indicated. Arguments contained in [and] 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.

d[*isplay*] [*display options*] | all | none

Specify display information. This includes information on minimum values, maximum values, average values, or per processor data.

he[lp] [*command*]

Provide brief command synopsis. If the *command* argument is present only information for that command will be displayed. The character "?" may be used as an alias for *help*.

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.

l[ines] function [[>] *filename*]

Print (display) the line level data together with the source for the specified *function*. If the *filename* argument is present, the output will be placed in the named file. The '>' means redirect output, and is optional.

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.

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.)

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. If the *filename* argument is present, the output will be placed in the named file. The '>' means redirect output, and is optional.

q[uit] Exit the profiler.

sel[ect] `coverage | covered | uncovered | all` [[<] *cutoff*]

This is the coverage mode variant of the *select* command. The cutoff value is interpreted as a percentage and is only applicable to the *coverage* option. The '<' means less than, and is optional. The default is `coverage` < 100%.

sel[ect] `calls | time/call | time | cost | all` [[>] *cutoff*]

You can choose to display data for a selected subset of the functions. This command allows you 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 `time` > 1%.

si[ngleprocessl] process_num

For multiptocess profiles, focus on a single process.

sh[ell] arg1, arg2, argn...
: For a shell using the given arguments.

so[rt] [by] `calls | time/call | time | cost | name`
: (Profile Mode) Function level data is displayed as a sorted list. This command establishes the basis for sorting. The default is `time`.

so[rt] [by] `coverage | name`
: This is the coverage mode variant of the *sort* command. The default is `coverage`, which causes the functions to be sorted based on percentage of lines covered, in ascending order.

src[dir] directory
: Add the named *directory* to the source file search path. This is useful if you neglected to specify source directories at invocation.

s[tat] [no]min|[no]avg|[no]max|[no]proc|[no]all]
: Set which process fields to display or do not display with the no versions.

th[read] thread_num.
: Specify a thread for a multithreaded process profile.

t[imes] `raw | pct`
: Specify whether time-related values should be displayed as raw numbers or as percentages. The default is `pct`. This command does not exist in coverage mode.

!! 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 the most recent previous command starting with *string* from the history list.

Index