IBM Parallel Environment Developer Edition
High Performance Computing Toolkit
Version 1 Release 3

Installation and Usage Guide



SC23-6940-01

IBM Parallel Environment Developer Edition
High Performance Computing Toolkit
Version 1 Release 3

Installation and Usage Guide



SC23-6940-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 333.

This edition applies to version 1, release 3, modification 0 of the IBM Parallel Environment Developer Edition High Performance Computing Toolkit (HPC Toolkit) (product numbers 5725-G61, 5725-M46, 5641-PD1, 5641-PD3, 5765-PED, and 5765-PDP) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC23-6940-00.

© Copyright IBM Corporation 2008, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures vii

Tables ix

About this information xi
Who should read this information xi
Conventions and terminology used in this information xi
Prerequisite and related information xii
Parallel Tools Platform component xii
How to send your comments xiii

Summary of changes xv

Limitations and restrictions xvii

Part 1. Introduction 1

Chapter 1. Introduction to the IBM HPC Toolkit 3
IBM PE Developer Edition components 3
HPC Toolkit user interfaces 4
Collecting performance data from a subset of parallel application tasks 5
Performance measurement tools 5
Instrumentation models. 6
The hpctInst utility 8
The hpcrun command 8

Part 2. Installation 9

Chapter 2. Installing the IBM High Performance Computing Toolkit 11
Installing the IBM HPC Toolkit on AIX systems . . 11
AIX installation using installp 12
AIX installation using SMIT 12
Installing the IBM HPC Toolkit on Linux systems . 13
Installing the IBM PE Developer Edition Eclipse feature 17

Part 3. The IBM HPC Toolkit Eclipse plug-in 19

Chapter 3. Using the IBM HPC Toolkit Eclipse plug-in 21
Preparing your application 21
Instrumenting your application at launch 22
Running the application 22
Instrumenting your application before launch . . . 31
Loading the application 32
Instrumenting your application. 34
Running the application 34
Using the Profile Configurations dialog 35
Viewing performance data 42
Viewing MPI traces. 47
Viewing I/O profiling data 48

Chapter 4. Using hardware performance counters in the plug-in . . 51
Preparing your application 51
Instrumenting your application at launch 51
Running your application 51
Instrumenting your application before launch . . . 53
Instrumenting your application. 53
Running your application 55
Viewing hardware performance counter data . . . 58

Chapter 5. Using MPI profiling in the plug-in 59
Preparing your application 59
Instrumenting your application at launch 59
Running your application 59
Instrumenting your application before launch . . . 60
Instrumenting your application. 61
Running your application 62
Viewing MPI profiling data 64

Chapter 6. Using OpenMP profiling in the plug-in 67
Preparing your application 67
Instrumenting your application at launch 67
Running your application 67
Instrumenting your application before launch . . . 68
Instrumenting your application. 68
Running your application 69
Viewing OpenMP performance data 69

Chapter 7. Using I/O profiling in the plug-in 73
Preparing your application 73
Instrumenting your application at launch 73
Running your application 73
Instrumenting your application before launch . . . 75
Instrumenting your application. 75
Running your application 76
Viewing I/O Data 77

Part 4. The peekperf GUI 81

Chapter 8. Using the peekperf GUI . . . 83
Preparing your application 83
Loading an application 84
Instrumenting your application. 89
Running your application 89

© Copyright IBM Corp. 2008, 2013

Viewing performance data 91
Viewing MPI traces 95
Viewing I/O profiling data 97
Compiling, editing, and running programs using peekperf 98
Miscellaneous peekperf features 99

Chapter 9. Using Hardware performance counters in peekperf . . 101
Preparing your application 101
Instrumenting your application 101
Running your program 103
Viewing hardware performance counter data . . . 104

Chapter 10. Using MPI profiling in peekperf 107
Preparing your application 107
Instrumenting your application 107
Running your application 111
Viewing MPI profiling data 111

Chapter 11. Using OpenMP profiling in peekperf 117
Preparing your application 117
Instrumenting your application 117
Running your application 119
Viewing OpenMP profiling performance data. . . 119

Chapter 12. Using I/O Profiling in peekperf 123
Preparing your application 123
Instrumenting your application 123
Running your application 125
Viewing I/O profiling data 125

Part 5. The X Windows performance profiler (Xprof). . . . 131

Chapter 13. Using the X Windows performance profiler (Xprof) 133
Before you begin using Xprof 133
Requirements and limitations 133
Comparing Xprof and the gprof command . . . 134
Compiling applications to be profiled 134
Starting the Xprof GUI 135
Specifying Xprof flags from the command line . . 136
Loading files from the Xprof GUI 140
Specifying the binary executable file. 143
Specifying profile data files. 144
Specifying Xprof flags from the GUI. 145
Setting the file search sequence 146
Setting default paths 146
Setting alternative paths 146
Changing the search sequence 147
Understanding the Xprof display 147
The Xprof main window 148
Xprof's main menus 149
Xprof's hidden menus 149
The display status field 150
How functions are represented 150
How calls between functions are depicted . . . 152
How library clusters are represented 152
Controlling how the display is updated 152
Other viewing options 153
Controlling the graphic style of the function call tree. 153
Controlling the orientation of the function call tree 153
Controlling the representation of the function call tree. 154
Filtering what you see 155
Restoring the status of the function call tree . . . 155
Displaying the entire function call tree 156
Excluding and including specific objects 156
Filtering shared library functions 156
Filtering by function characteristics 157
Including and excluding parent and child functions 160
Clustering libraries 162
Clustering functions 162
Unclustering functions 164
Locating specific objects in the function call tree 165
Locating and displaying parent functions 166
Locating and displaying child functions 166
Locating and displaying ancestor functions . . . 167
Locating and displaying descendant functions . . 167
Locating and displaying functions on a cycle . . . 167
Obtaining performance data for your application 167
Obtaining basic data 167
Understanding basic function data 167
Understanding basic call data 169
Basic cluster data 169
Understanding information boxes 169
Using the function menu statistics report option 170
Getting detailed data from reports 171
Understanding the flat profile report 172
Understanding the call graph profile report . . 174
Understanding the function index report . . . 177
Understanding the function call summary report 178
Understanding the library statistics report . . . 179
Saving reports to a file 180
Looking at your code. 182
Viewing the source code. 183
Viewing the disassembler code 185
Saving screen images of profiled data 187
Customizing Xprof resources 190
Setting Xprof resource variables 190
Controlling fonts 190
Controlling the appearance of the Xprof main window 191
Controlling variables related to the file menu 192
Controlling variables related to the screen dump option 193
Controlling variables related to the view menu 195
Controlling variables related to the filter menu 196

Part 6. The hardware performance counter tools. 197

Chapter 14. Using the hardware performance counter tools 199
Using the hpccount command 199
Using the hpcstat command 200
Using the libhpc library 201
Understanding CPU hardware counter multiplexing. 203
Understanding derived metrics 204
Understanding MFlop issues 204
Understanding inheritance 204
Understanding inclusive and exclusive event counts 205
Understanding parent-child relationships 205
Handling of overlap issues 206
Understanding measurement overhead 206
Handling multithreaded program instrumentation issues 207
Considerations for MPI programs 207
General considerations 207
Hardware performance counter plug-ins . . . 208

Part 7. The MPI and I/O profiling libraries 213

Chapter 15. Using the MPI profiling library 215
Compiling and linking with libmpitrace 215
Controlling traced tasks 216
Additional trace controls 216
Customizing MPI profiling data 217
Understanding MPI profiling utility functions . . 218
Performance data file naming 218

Chapter 16. Using the I/O profiling library 219
Preparing your application 219
Setting I/O profiling environment variables . . . 219
Specifying I/O profiling library module options 221
Running your application 223
Performance data file naming 224

Part 8. Using the hpctInst command 225

Chapter 17. Instrumenting your application using hpctInst 227
Instrumenting your application for hardware performance counters 227
Instrumenting your application for MPI profiling 228
Instrumenting your application for OpenMP profiling 229
Instrumenting your application for I/O profiling 229

Part 9. Command and API reference 231

Chapter 18. Commands 233
dataview - View an I/O trace 234
hpccount - Report hardware performance counter statistics for an application 236
hpcrun - Launch a program to collect profiling or trace data. 241
hpcstat - Reports a system-wide summary of hardware performance counter statistics 244
hpctInst - Instrument applications to obtain performance data 248
peekperf - Provides a GUI for the HPC Toolkit . . 252
peekview - Display an MPI trace file 254
Xprof - Display the application as a function tree 256

Chapter 19. Application programming interfaces 261
hpm_error_count, f_hpm_error - Verify a call to a libhpc function 263
hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment 265
hpmStart, f_hpmstart - Identify the starting point for an instrumented region of code 269
hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code 271
hpmStop, f_hpmstop - Identify the end point of an instrumented region of code 274
hpmTerminate, f_hpmterminate - Generate HPM statistic files and shut down HPM 276
hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code 278
hpmTstartx, f_hpmtstartx - Identify the starting point for an instrumented region of code 280
hpmTstop, f_hpmtstop - Identify the end point of an instrumented region of code 283
MT_get_allresults - Obtain statistical results . . . 285
MT_get_calleraddress - Obtain the address of the caller of an MPI function 288
MT_get_callerinfo - Obtain source code information 289
MT_get_elapsed_time - Obtains elapsed time. . . 291
MT_get_environment - Returns run-time environment information 292
MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred 293
MT_get_mpi_counts - Obtain the the number of times a function was called. 294
MT_get_mpi_name - Returns the name of the specified MPI function 295
MT_get_mpi_time - Obtain elapsed time 296
MT_get_time - Get the elapsed time. 297
MT_get_tracebufferinfo - Obtain information about MPI trace buffer usage 298
MT_output_text - Generate performance statistics 299
MT_output_trace - Control whether an MPI trace file is created 300
MT_trace_event - Control whether an MPI trace event is generated 301
MT_trace_start, mt_trace_start - Start or resume the collection of trace events 303
MT_trace_stop, mt_trace_stop - Suspend the collection of trace events 305

Part 10. Appendixes 307

Appendix A. Performance data file naming 309
File naming conventions. 309
File naming examples 311

Appendix B. Derived metrics descriptions 315
Derived metrics description for AIX and Linux on Power6 315
Derived metrics description for AIX and Linux on POWER7 and POWER7+ 316
Derived metrics description for x86 Nehalem and Westmere. 317
Derived metrics description for x86 Sandy Bridge 318
Derived metrics description for x86 Ivy Bridge . . 322

Appendix C. HPC Toolkit environment variables 327

Accessibility features for IBM PE Developer Edition 331
Accessibility features 331
Keyboard navigation 331
IBM and accessibility 331

Notices 333
Trademarks 335

Index 337

Figures

1. HPCT Preferences dialog 22
2. Profile Configurations dialog. 23
3. Resources tab 24
4. Application tab 25
5. Arguments tab 26
6. Environment tab 27
7. Performance Analysis tab 28
8. Common tab 29
9. HPM options pane 30
10. HPM instrumentation view 31
11. Initial HPCT perspective 32
12. HPCT perspective after opening the application 33
13. Expanded HPM instrumentation view . . . 34
14. Initial Profile Configurations dialog 35
15. Profile Configurations dialog Resources tab 36
16. Profile Configurations dialog Application tab 37
17. Profile Configurations dialog Arguments tab 38
18. Profile Configurations dialog Environment tab 39
19. Profile Configurations dialog Performance Analysis tab 40
20. Profile Configurations dialog HPM settings tab 41
21. HPCT perspective after loading performance data 43
22. Performance Data Summary 43
23. Performance Data Summary view 44
24. Performance data - FilterDialog 45
25. Metric Browser view 45
26. Metrics FilterDialog 46
27. Performance Data Detail view 47
28. MPI trace view 48
29. MIO summary view 48
30. MIO trace view 49
31. Profile Configurations dialog with the instrumentation points tree fully expanded . . 52
32. Eclipse plug-in the HPM instrumentation view 54
33. Profile Configurations dialog Common settings 56
34. Profile Configurations dialog hardware counter settings 57
35. Profile Configurations dialog with the instrumentation points tree fully expanded . . 60
36. Eclipse plug-in MPI instrumentation view 61
37. Profile Configurations dialog MPI settings panel. 63
38. Performance data summary view with MPI profiling data 64
39. MPI trace view 65
40. Profile Configurations dialog with the instrumentation points tree fully expanded . . 68
41. OpenMP instrumentation view 69
42. Performance data summary view with OpenMP profiling data. 70
43. Profile Configurations dialog with the I/O instrumentation points tree fully expanded . . 74
44. I/O Instrumentation view. 75
45. Profile Configurations dialog MIO settings panel. 77
46. Performance data summary view with I/O profiling data 78
47. MIO tree view 78
48. MIO trace view 79
49. Data view table view 80
50. Peekperf Main Window at startup 84
51. Additional linking options (AIX only). . . . 85
52. Peekperf data collection 85
53. Peekperf data collection window with expanded application structure 87
54. Peekperf search results window. 88
55. Peekperf environment settings window 89
56. Peekperf run dialog 1 of 3 90
57. Peekperf run dialog 2 of 3 90
58. Peekperf run dialog 3 of 3 91
59. Peekperf data visualization 92
60. Peekperf expanded data visualization window 93
61. Peekperf data filtering dialog 94
62. Peekperf Metric Browser window 94
63. Peekperf performance data table window 95
64. Peekperf trace viewer window 96
65. Peekperf trace viewer identifier window 96
66. Peekperf I/O profiling selection window 97
67. Peekperf data plot window 98
68. Peekperf HPM data collection window 102
69. Peekperf hardware counter group selection dialog 104
70. Peekperf task rank selection dialog 105
71. Peekperf MPI profiling data collection window 108
72. Peekperf MPI profiling data collection with selected instrumentation 110
73. Peekperf MPI profiling data visualization window 112
74. Peekperf MPI task rank selection dialog 113
75. Peekperf MPI trace viewer identifier dialog 113
76. Peekperf trace viewer 114
77. Peekperf trace viewer detailed view 115
78. Peekperf OpenMP data collection window 118
79. Peekperf OpenMP data visualization window 120
80. Peekperf I/O profiling data collection window 124
81. Peekperf I/O profiling data visualization window 126
82. Peekperf I/O profiling data selection window 127
83. Peekperf I/O profiling data plot window 128
84. Peekperf I/O profiling data table window 129
85. The Xprof main window. 141
86. The Load Files Dialog. 142
87. The binary executable file dialog 143
88. The gmon.out profile data file(s) area 144
89. The Command Line Options area 146
90. The Xprof main window with application loaded 148

© Copyright IBM Corp. 2008, 2013

91. Function boxes and arcs in the Xprof display 151
92. The Xprof main window with left-to-right format selected 154
93. The Filter by Function Names Dialog 158
94. The Filter by CPU Time Dialog 159
95. The Filter by Call Counts Dialog 160
96. The Xprof main window with functions unclustered 162
97. The Xprof main window with one library cluster box collapsed 163
98. The Xprof main window with one library cluster box removed 165
99. An example of a function box label 168
100. An example of a call arc label 169
101. The Function Level Statistics Report window 170
102. The Flat Profile window 173
103. The Call Graph Profile window 175
104. The called/total, call/self, called/total field of the Call Graph Profile window. 176
105. The name/index/parents/children field of the Call Graph Profile window. 177
106. The Function Index window 178
107. The Function Call Summary window 179
108. The Library Statistics window 180
109. The Source Code window 184
110. The Disassembler Code window 186
111. The Screen Dump Options dialog 188

Tables

1. Conventions xi
2. AIX software requirements for IBM HPC Toolkit 11
3. IBM HPC Toolkit AIX fileset names for the platform-specific license 13
4. IBM HPC Toolkit AIX fileset names for the multiplatform license 13
5. Linux software requirements for IBM HPC Toolkit 13
6. Platform specific Linux RPM file names for the IBM HPC Toolkit. 16
7. Multiplatform Linux RPM file names for the IBM HPC Toolkit. 16
8. Profile Configurations dialog Common environment variables 56
9. HPM profile configuration dialog environment variables 57
10. MPI trace profile configuration environment variables 63
11. MPI trace timeline navigation 66
12. MIO profile configuration environment variables 77
13. MIO trace processing actions. 79
14. Xprof command line flags 136
15. Xprof main menus 149
16. Xprof's hidden menus 149
17. Specifying Xprof fonts 191
18. Specifying Xprof report fonts 191
19. Specifying Xprof main window size 191
20. Specifying Xprof main window colors 191
21. Specifying the number of function boxes 191
22. Specifying function box colors 191
23. Specifying highlighted function box colors 192
24. Specifying de-emphasized function box colors 192
25. Specifying the size of the load files dialog 192
26. Requesting confirmation before Xprof overwrites files 192
27. Specifying the Xprof file search path 192
28. Specifying the Xprof file search sequence 193
29. Specifying Xprof screen capture destination 193
30. Requesting Xprof screen captures in color 193
31. Specifying the number of gray shades for Xprof screen captures 193
32. Specifying Xprof screen capture delay 193
33. Specifying Xprof delay before grab value 194
34. Specifying Xprof screen capture orientation 194
35. Requesting Xprof screen capture annotation 194
36. Specifying Xprof screen capture directory 194
37. Specifying Xprof screen capture printer 195
38. Specifying Xprof overview window size 195
39. Specifying Xprof overview window highlight area color 195
40. Control Xprof highlight area tracking 195
41. Specify Xprof function box format 195
42. Specifying Xprof function call tree layout 195
43. Specifying Xprof function box clustering 196
44. Specifying Xprof function call arc expansion 196
45. Plug-ins shipped with the tool kit. 209
46. MPI profiling utility functions 218
47. MIO analysis modules 221
48. MIO module options 221
49. MIO pf module options 221
50. MIO trace module options 222
51. MIO recov module options 223
52. Commands 233
53. APIs 261
54. Controlling the collection of MPI trace events for any function with an identifier 301
55. File naming conventions for various tools and their types 309
56. HPC Toolkit environment variables 327

© Copyright IBM Corp. 2008, 2013

About this information

This information applies only to IBM® Parallel Environment (PE) Developer Edition Version 1.3.

Attention:

In this information unit, IBM PE Developer Edition Eclipse client and IBM PE Developer Edition Workbench are synonymous and can be used interchangeably.

Who should read this information

This information is for parallel application developers looking to improve productivity throughout the edit, compile, debug, and run development cycle. A set of integrated performance analysis tools is provided to help the application developer tune a serial or parallel application.

Conventions and terminology used in this information

Table 1 shows the conventions used in this information:

Table 1. Conventions

Convention	Usage
bold	**Bold** words or characters represent system elements that you must use literally, such as commands, flags, path names, directories, file names, values, and selected menu options.
bold underlined	**Bold underlined** keywords are defaults. These take effect if you do not specify a different keyword.
`constant width`	Examples and information that the system displays appear in `constant-width` typeface.
italic	*Italic* words or characters represent variable values that you must supply. *Italics* are also used for information unit titles, for the first use of a glossary term, and for general emphasis in text.
<*key*>	Angle brackets (less-than and greater-than) enclose the name of a key on the keyboard. For example, <**Enter**> refers to the key on your terminal or workstation that is labeled with the word *Enter*.

\	In command examples, a backslash indicates that the command or coding example continues on the next line. For example:
	`mkcondition -r IBM.FileSystem -e "PercentTotUsed > 90" \`
	`-E "PercentTotUsed < 85" -m d "FileSystem space used"`

{*item*}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
[*item*]	Brackets enclose optional items in format and syntax descriptions.
<**Ctrl-***x*>	The notation <**Ctrl-***x*> indicates a control character sequence. For example, <**Ctrl-c**> means that you hold down the control key while pressing <**c**>.
item...	Ellipses indicate that you can repeat the preceding item one or more times.

© Copyright IBM Corp. 2008, 2013

Table 1. Conventions (continued)

Convention	Usage
|	• In *syntax* statements, vertical lines separate a list of choices. In other words, a vertical line means *Or*. • In the margin of the document, vertical lines indicate technical changes to the information.

Prerequisite and related information

IBM Parallel Environment Runtime Edition

The IBM Parallel Environment Runtime Edition library consists of:

- *IBM Parallel Environment Runtime Edition: Installation*, SC23-6780
- *IBM Parallel Environment Runtime Edition: LAPI Programming Guide*, SA23-2272
- *IBM Parallel Environment Runtime Edition: Messages*, SC23-6782
- *IBM Parallel Environment Runtime Edition: MPI Programming Guide, SC23-6783*
- *IBM Parallel Environment Runtime Edition: MPI Subroutine Reference*, SC23-6784
- *IBM Parallel Environment Runtime Edition: NRT API Programming Guide*, SC23-6785
- *IBM Parallel Environment Runtime Edition: Operation and Use*, SC23-6781
- *IBM Parallel Environment Runtime Edition: PAMI Programming Guide*, SA23-2273

In addition, see the *IBM Parallel Environment Developer Edition: High Performance Computing Toolkit Installation and Usage Guide*, SC23-6940.

To access the most recent IBM PE Runtime Edition or IBM PE Developer Edition documentation in PDF and HTML format, refer to the IBM Cluster Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp).

The current IBM PE Runtime Edition or IBM PE Developer Edition documentation is also available in PDF format from the IBM Publications Center (http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss).

To easily locate a book in the IBM Publications Center, supply the book's publication number. The publication number for each of the Parallel Environment books is listed after the book title in the preceding lists.

Parallel Tools Platform component

For information about how to use the Eclipse client Parallel Tools Platform (PTP) component, go to http://help.eclipse.org/kepler/index.jsp (http://help.eclipse.org/kepler/index.jsp) and search for these documents:

- *Parallel Development User Guide*
- *Photran User's Guide*
- *Remote Development Tools User Guide*

These documents are available both on the web and as online Help that can be accessed through the Eclipse user interface.

To access the most recent Eclipse client PTP component installation documentation in PDF and HTML format, refer to the IBM Cluster Information Center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp):

- *IBM PE Developer Edition Eclipse Client for AIX and Linux: Installation Guide*, SC23-7030

The current Eclipse client PTP component installation book is also available in PDF format from the IBM Publications Center (http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss). To easily locate a book in the IBM Publications Center, supply the book's publication number.

How to send your comments

Your feedback is important in helping us to produce accurate, high-quality information. If you have any comments about this book or any other IBM High Performance Computing Toolkit documentation, send your comments by e-mail to:

mhvrcfs@us.ibm.com

Include the book title and order number, and, if applicable, the specific location of the information you have comments on (for example, a page number or a table number).

For technical information and to exchange ideas related to high performance computing, go to:

- HPC Central (https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Welcome%20to%20High%20Performance%20Computing%20%28HPC%29%20Central)
- HPC Central Technical Forum (https://www.ibm.com/developerworks/community/forums/html/forum?id=11111111-0000-0000-0000-000000001056)

Summary of changes

The following summarizes changes to the IBM Parallel Environment (PE) Developer Edition Version 1 Release 3 product and library.

Within each information unit in the library, a vertical line in the margin next to text and illustrations indicates technical changes or additions made to the previous edition of the information.

Attention:

In this information unit, IBM PE Developer Edition Eclipse client and IBM PE Developer Edition Workbench are synonymous and can be used interchangeably.

New information

- Support for POWER7+™ hardware counters and Intel's Ivy Bridge microarchitecture hardware counters has been added.
- A new, streamlined sequence to instrument and launch the application has been added.
- Updates to the plug-in Help have been incorporated to describe the new features.
- Enhancements to the Parallel Tools Platform (PTP) developer workflow have been added.
- Enhancements to the integration of module's support in PTP have been made.
- Support for MPICH2 on AIX® has been added, but is restricted to the level of the MPICH2 library provided by IBM PE Runtime Edition on AIX.
- The two HPC Toolkit installation images have different license names to distinguish them from each other. It is important to not mix installation images with the different licenses.
- For Linux on POWER® and Linux on x86, installation images for both platform-specific and multiplatform licenses are available as xCAT kit packages for installation on an xCAT cluster. Follow the xCAT procedures for installing HPC Toolkit kit packages found on the IBM HPC Software Kits web site (http://sourceforge.net/apps/mediawiki/xcat/index.php?title=IBM_HPC_Software_Kits).
- Prebuilt Scalable Debug Manager (SDM) binaries for AIX, Linux on POWER and Linux on x86 are now available with the IBM PE Developer Edition Workbench for automatic deployment of SDM for debug sessions.
- Support for submitting batch background application jobs to the LSF® job scheduler from the Parallel Tools Platform has been added. Monitoring and controlling the execution status of those jobs is done by using a PTP GUI interface.

Deleted information

- Information about HFA and CAU counters has been removed.
- The IBM Power® 775 is not supported with version 1.3 of the IBM HPC Toolkit, so references to it have been removed.

© Copyright IBM Corp. 2008, 2013

Limitations and restrictions

The following limitations and restrictions apply:

- On Power Architecture®, the dynamic instrumentation tools (**hpctInst**) will only operate reliably on executables with a text segment size less than 32 MB.
- On x86 Linux systems, the only supported tools are the hardware performance counter tool, the MPI profiling tool and the application I/O profiling tool, where the only instrumentation model supported is that you link your application with the corresponding IBM HPC Toolkit runtime library for that tool. Instrumenting the application using **hpctInst**, **peekperf**, or the Eclipse plug-in is not supported.
- The Xprof disassembler code view is not available for 64-bit Linux programs.
- Using the **hpcrun** command to launch an MPI program compiled with **-hpcprof** is only supported on AIX.
- Instrumentation of executables compiled and linked with the **-pg** flag using the IBM Eclipse HPC Toolkit plug-in, **peekperf**, or **hpctInst** is not supported.
- Using the **hpcrun** command to launch applications that use dynamic tasking is not supported.
- On Linux running on Power Architecture, instrumenting programs with MPI and HPM at the same time is not supported.
- The only **--tracestore** option that is valid for the **hpcrun** command when collecting I/O trace is **memory**. Although the **--tracestore** option accepts file names for both MPI and I/O tracing, I/O tracing should not be used unless the **memory** option was chosen.
- HPC Toolkit using the **hpcrun** command, which is intended to filter the volume of data collected for large numbers of application tasks, has been tested to 3500 application tasks. When collecting MPI profiling and trace data you might notice performance degradation as the task count increases.
- Support for MPICH2 on AIX has been added, but is restricted to the level of the MPICH2 library provided by IBM PE Runtime Edition on AIX.
- In order to gather hardware performance counters on Intel's Ivy Bridge microarchitecture, an OS level of at least Red Hat Enterprise Linux 6.4 or SLES 11 SP 3 is required.
- The IBM Power 775 is not supported with version 1.3 of the IBM HPC Toolkit.

© Copyright IBM Corp. 2008, 2013

Part 1. Introduction

The topics in this part provide an introduction to the IBM High Performance Computing Toolkit.

© Copyright IBM Corp. 2008, 2013

Chapter 1. Introduction to the IBM HPC Toolkit

The HPC Toolkit is an integrated set of performance analysis tools that help the application developer tune a serial or parallel application. The HPC Toolkit is a component of IBM Parallel Environment Developer Edition. IBM Parallel Environment (PE) Developer Edition Version 1 Release 3 provides an Eclipse-based integrated development environment (IDE) that improves parallel application developer productivity throughout the edit, compile, debug, and run development cycle for C, C++ and Fortran applications.

IBM PE Developer Edition components

The IBM PE Developer Edition is a companion to IBM PE Runtime Edition.

The components of the IBM PE Developer Edition include:

- Eclipse 4.3 (Kepler) platform and the following other Eclipse components:
  - Parallel Tools Platform (PTP) including the RDT (Remote Development Tools) and support for IBM Parallel Environment and LoadLeveler®
    - PTP is an Eclipse based application development environment that contains an integrated set of tools to help you edit, compile, run, debug, and analyze your parallel application. Advanced tools included with PTP include static analysis tools to locate errors before the code is compiled, refactoring tools to modify code while preserving behavior, and an integrated parallel debugger. PTP supports a broad range of architectures and job schedulers, and provides the ability to easily add support for additional systems.
  - C/C++ Development Tools (CDT)
    - Development tools for C/C++
  - Fortran Development Tools (Photran)
    - Development tools for Fortran, including state-of-the-art refactoring tools for Fortran.
- Specific add-ons in the IBM PE Developer Edition from IBM include:
  - IBM High Performance Computing Toolkit - performance analysis tools:

    This document focuses primarily on the IBM High Performance Computing Toolkit (IBM HPC Toolkit), which is a collection of tools that you can use to analyze the performance of parallel and serial applications that are written in C or FORTRAN, running on AIX or Red Hat Enterprise Linux 6.3 and 6.4 operating systems on IBM Power Systems servers. Applications running on Red Hat 6.3 and Red Hat 6.4 and SLES 11 SP 2 or later, on IBM System x® with the Intel microarchitecture codename Nehalem, Westmere, Ivy Bridge, and Sandy Bridge family of processors are also supported. The Xprof GUI also supports C++ applications.

    These tools perform the following functions:

    - Provide access to hardware performance counters for performing low-level analysis of an application, including analyzing cache usage and floating-point performance
    - Profile and trace an MPI application for analyzing MPI communication patterns and performance problems

© Copyright IBM Corp. 2008, 2013

- Profile an OpenMP application for analyzing OpenMP performance problems and to help you determine if an OpenMP application properly structures its processing for best performance
- Profile application I/O for analyzing an application's I/O patterns and whether you can improve the application's I/O performance
- Profile an application's execution for identifying hotspots in the application, and for locating relationships between functions in your application to help you better understand the application's performance

– IBM PAMI and LAPI user assistance features:
- Provides features that augment the C/C++ Eclipse editor for ease of entering PAMI and LAPI APIs, locating them in source code, getting information about API usage and arguments, and so on.

– IBM XLC and XLF Compiler Transformation Report Feedback Viewer
- Provides a source-linked view of items identified by the IBM XLC/XLF Compiler Transformation reports.
- For details about transformation reports on optimization and other features of the IBM XL C/C++ and IBM XL FORTRAN compilers, see information on IBM AIX compilers and IBM Linux compilers.

HPC Toolkit user interfaces

- You can collect performance data from an instrumented application by setting environment variables for the specific analysis tool and then invoking the application from the command line.
- You can modify your application to add calls to the hardware performance counter tool, rebuild your application linking with the hardware performance counter library, and then run your application to get hardware performance counter data.
- You can link your application directly with the MPI profiling and trace library then run your application to get MPI trace and profiling data.
- You can link your application with the I/O profiling and trace library to get I/O profiling and trace information.

The IBM HPC Toolkit provides three primary interfaces for you to use:

- The first is the Eclipse IBM HPC Toolkit plug-in. This plug-in is an extension to the Eclipse IDE that you can use to run hardware performance counter analysis, MPI profiling and tracing, OpenMP profiling, and I/O profiling. The plug-in allows you to select the parts of your application that are to be instrumented, instrument those parts of the application, run the instrumented application, and view the resulting performance data. The plug-in allows you to sort and filter the data to help you find the performance problems in the application.

  For more information, see the topics in Part 3, "The IBM HPC Toolkit Eclipse plug-in," on page 19.
- The second interface is **peekperf**. The **peekperf** interface is an AIX or Linux based GUI that you can use to run hardware performance counter analysis, MPI profiling and tracing, OpenMP profiling, and I/O profiling. The **peekperf** GUI allows you to select the parts of your application that are to be instrumented, instrument those parts of the application, run the instrumented application and view the resulting performance data. The **peekperf** GUI allows you to sort and filter the data to help you find the performance problems in the application.

  For more information, see the topics in Part 4, "The peekperf GUI," on page 81.

- The third interface is **Xprof**, which you can use to view low-level profiling data for your application. The **Xprof** interface allows you to view the performance data in **gmon.out** files, generated by compiling your application using the **–pg** compiler flag. You can view the profiling data, identify hotspots in the application, view relationships between functions in the application, zoom in to areas of the application of greater interest, and sort and filter the data to help identify hotspots in the application.

  For more information, see the topics in Part 5, "The X Windows performance profiler (Xprof)," on page 131.

Collecting performance data from a subset of parallel application tasks

By default, the IBM HPC Toolkit generates and collects performance data for all tasks in the application. If the application has a large number of tasks, the overhead of collecting performance data from all application tasks may be excessive.

The IBM HPC Toolkit provides a command, **hpcrun,** that you can use to collect performance data from a subset of application tasks. The **hpcrun** command allows you to specify a number of application tasks to collect performance data from and the performance metric to use in determining which tasks to collect performance data from.

The **hpcrun** command will collect performance data from the number of application tasks you specify that have the minimum value for the performance metric you specify, the number of application tasks you specify that have the maximum value for the performance metric you specify, the task closest to the average for the metric you selected and application task zero.

You can collect performance data by specifying a metric of **ELAPSED_TIME** to use elapsed wall clock time as the performance metric, or by specifying a metric of **CPU_TIME** to use CPU time as the performance metric.

If you are collecting MPI trace or I/O trace data, you can either collect trace data in memory in each application task or store it to temporary files. If the size of the individual trace files is small, then you should use the in-memory model for better performance. Otherwise, use the temporary file model for trace file generation.

See "hpcrun - Launch a program to collect profiling or trace data" on page 241 for details of specific command line flags and environment variables.

Performance measurement tools

The following tools are provided to help you obtain performance measurements:

- Hardware performance counters, including:
  - Measurements for cache misses at all levels of cache
  - The number of floating point instructions executed
  - The number of load instructions resulting in TLB misses
  - Other measurements that are supported by your hardware

  These measurements help the algorithm designer or developer identify and eliminate performance bottlenecks. The hardware performance counter tools allow you to run individual tasks of an MPI application with different groups of

hardware counters monitored in each MPI task, so that you can obtain measurements for more than one hardware counter group within a single execution.

These tools also allow you to summarize or aggregate hardware performance counter measurements from a set of MPI tasks, using plug-ins provided with the IBM HPC Toolkit or provided by you. On AIX, you can multiplex or time slice multiple hardware counter groups in a single task, allowing you to get hardware performance counter events from multiple groups in the same application process.

- MPI profiling and trace, where MPI profiling obtains performance metrics including time spent in each MPI function call and MPI trace creates a trace of MPI function calls in your application so you can view MPI communication patterns in your application.
- OpenMP profiling, where you can obtain information about time spent in OpenMP constructs in your program, information about overhead in OpenMP constructs, and information about how workload is balanced across OpenMP threads in your application.
- Application I/O profiling, where you can obtain information about I/O calls made in your application to help you understand application I/O performance and identify possible I/O performance problems in your application.
- Application profiling, where you can identify functions in your application where the most time is being spent, or where the amount of time spent is different from your expectations. This information is presented in a graphical display that helps you better understand the relationships between functions in your application.

On x86 Linux systems the only supported tools are the hardware performance counter tool, the MPI profiling tool and the application I/O profiling tool, where the only instrumentation model supported is that you link your application with the corresponding IBM HPC Toolkit runtime library for that tool. Instrumenting the application using **hpctInst**, **peekperf** or the Eclipse plug-in is not supported.

Instrumentation models

The IBM HPC Toolkit has two instrumentation models:

- The first model is one in which the application executable is rewritten with the instrumentation specified by you. You specify the instrumentation using the Eclipse plug-in, the **peekperf** GUI, or the **hpctInst** command-line interface. The IBM HPC Toolkit rewrites the application executable, adding the requested calls to the instrumentation libraries into the executable. This process does not require any modifications to the application source code, and does not require the application to be relinked.
- The second instrumentation model can be used when obtaining measurements using the Hardware Performance Monitor (HPM) or to control the generation of the MPI trace and MPI profiling information when using MPI profiling. In this model, you insert calls to functions in the instrumentation library in to your application source, then recompile and relink your application.

When you instrument your application, you should choose only one of the instrumentation models. This is because any calls to instrumentation functions that you code in your application might interfere with the instrumentation calls that are inserted by use of the Eclipse plug-in, **peekperf**, or **hpctInst** tools.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you can use the toolkit. In order to set these environment variables, you should run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation.

To initialize the IBM HPC Toolkit, issue the following command:

```
. /opt/ibmhpc/ppedev.hpct/env_sh
```

For **csh**, the script is:

```
source /opt/ibmhpc/ppedev.hpct/env_csh
```

If you are using the Eclipse plug-in, you do not set these environment variables directly. Instead, you set them using Eclipse dialogs as described in Chapter 3, "Using the IBM HPC Toolkit Eclipse plug-in," on page 21 and "Loading the application" on page 32.

The IBM HPC Toolkit requires your application to be compiled and linked using the **-g** flag. For Power Linux systems, you must also specify the **-Wl,--hash-style=sysv -emit-stub-syms** flags when compiling your application. If your application has not been compiled and linked using these flags, the Eclipse plug-in, **peekperf**, and **hpctInst** will not be able to instrument your application. If you plan to use **Xprof** to view profiling data for your application, your application must also be compiled and linked using the **–pg** flag.

After your application has been instrumented, you must ensure that it, and any data files that it requires, are properly distributed on the system on which the application will run. If your application resides in a globally accessible file system, such as a GPFS™ file system, you should not have to do anything to distribute the application. If your system is set up so that only local file systems are used, you must manually copy the application and any data files it requires to each node on which the application will run, using system utilities that are most appropriate for your system.

You also need to make sure that the application environment has been properly set up. This includes making sure that any environment variables that are required to control the instrumentation in your application are properly set.

After you have set up the application environment, you can invoke the application as you would normally. The performance measurements you requested will be obtained while the application runs, and the results will be written when the application completes.

By default, the performance measurement data is written to the current working directory for the application. Before you can view the performance data, you must ensure that all the performance data files are accessible to the node on which you will run the visualization tools. If the current working directory for the application resides in a global file system, you should not need to do anything to make the performance data files accessible to the visualization tools. If the current working directory resides in local file systems on each node on which the application ran, you need to collect all the performance data files that you want to view in to a single directory that is accessible to the visualization tools. You can use any system utilities appropriate for your system to move the performance data files to a directory that is accessible to the visualization tools.

The topics that follow describe how to:

- Install the IBM HPC Toolkit
- Use the GUI tools to instrument, run, and analyze the performance of an application
- Instrument your application by inserting calls to the instrumentation libraries in your application
- Use the command line instrumentation tools

The hpctInst utility

The IBM HPC Toolkit also provides the **hpctInst** utility, which you can use on Power Architecture to instrument the application without using the Eclipse IBM HPC Toolkit plug-in or the **peekperf** GUI. You can specify the types of instrumentation you want to use as well as the locations within the application which are to be instrumented. The **hpctInst** utility rewrites the application binary with the instrumentation you selected. Then, you can run that instrumented executable to obtain the same types of performance measurements that you could by using the Eclipse IBM HPC Toolkit plug-in or **peekperf**.

For more information about the **hpctInst** utility, see "hpctInst - Instrument applications to obtain performance data" on page 248.

The hpcrun command

The IBM HPC Toolkit includes the **hpcrun** command that you can use to collect a subset of performance data from an application with many parallel tasks instead of collecting data from all tasks.

For more information about the **hpcrun** command, see "hpcrun - Launch a program to collect profiling or trace data" on page 241.

Part 2. Installation

The topics in this part provide information about installing the IBM High Performance Computing Toolkit.

© Copyright IBM Corp. 2008, 2013

Chapter 2. Installing the IBM High Performance Computing Toolkit

The topics that follow provide information about installing the IBM HPC Toolkit on AIX and Linux systems.

Installing the IBM HPC Toolkit on AIX systems

The IBM High Performance Computing Toolkit can be installed on IBM Power Systems™ servers that are supported by IBM Parallel Environment (PE).

The IBM HPC Toolkit has the following software requirements:

Table 2. AIX software requirements for IBM HPC Toolkit

Function	**Software**	**AIX 7.1 Service Pack 3**
All	Operating system	AIX 7.1 Service Pack 3
Compiling C applications	IBM VAC compiler	12.1 or later
Compiling FORTRAN applications	IBM xlf compiler	14.1 or later
Access hardware performance counters	bos.pmapi	7.1 Service Pack 3
Running the Xprof GUI	X11.motif.lib	7.1.0.0 or later
Run an application instrumented with **hpctInst**, **peekperf**, or the IBM HPC Toolkit Eclipse plug-in	xlsmp.rte	Compatible with OS level
Compile and run MPI or PAMI programs	Parallel Environment Runtime Edition	1.3.0.6 or later

In addition, you must have the **/proc** file system configured and enabled on any system where you run applications instrumented by the IBM HPC Toolkit.

You must install and accept the Parallel Environment Developer Edition license (ppedev.loc.license) on each node where you intend to install the IBM HPC Toolkit installation images before you can install the IBM HPC Toolkit.

Prerequisite software must be installed, following the documentation for that software's installation, before you install the IBM HPC Toolkit.

There are two sets of installation images for the IBM HPC Toolkit, each with different names for the AIX version:

- One set uses the platform-specific license and it uses the **ppedev** prefix
- One set uses the multiplatform license and it uses the **ppedev_pa** prefix

Each set of install images contains two installation images for the AIX version:

- **ppedev.hpct** and **ppedev_pa.hpct**

© Copyright IBM Corp. 2008, 2013

The **ppedev.hpct** or **ppedev_pa.hpct** installation image must be installed on any node where you intend to use the X based GUI tools, instrument an application, or compile an application where you have coded calls to the instrumentation libraries.

- **ppedev.rte** and **ppedev_pa.rte**

  The **ppedev.rte** or **ppedev_pa.rte** installation image must be installed on all nodes where applications instrumented with the toolkit will run, and also on any node where the **ppedev.rte** or **ppedev_pa.rte** installation image is installed.

To use the plug-in, the IBM PE Developer Edition Workbench must be installed. For more information, see the *IBM PE Developer Edition Eclipse Client for AIX and Linux: Installation Guide*, SC23-7030.

You can install the IBM HPC Toolkit using either the **installp** command or by using SMIT. You can install using either the installation media, or by copying the installation images to a directory accessible to the node and installing from that directory.

AIX installation using installp

If the Parallel Environment Developer Edition product license has not been previously accepted, you must have a copy of either the **ppedev.loc.license** (for the platform-specific license) or the **ppedev_pa.loc.license** (for the multiplatform license) **installp** image in the same directory as the IBM HPC Toolkit **installp** images.

To install the **ppedev.rte** or the **ppedev_pa.rte** installation image using **installp** (assuming the installation images are stored in **~/images**), enter:

```
installp -a -I -X -Y -d ~/images ppedev.rte
```

or:

```
installp -a -I -X -Y -d ~/images ppedev_pa.rte
```

To install the **ppedev.hpct** installation image using **installp** (assuming the installation images are stored in **~/images**), enter:

```
installp -a -I -X -Y -d ~/images ppedev.hpct
```

or:

```
installp -a -I -X -Y -d ~/images ppedev_pa.hpct
```

To install the **ppedev.hpct** and **ppedev.rte** images assuming they are located in **~/images**, enter:

```
installp -a -I -X -Y -d ~/images ppedev.*
```

or:

```
installp -a -I -X -Y -d ~/images ppedev_pa.*
```

AIX installation using SMIT

To initially install the installation image using SMIT, follow these instructions:

1. Insert the distribution medium in the installation device, unless you are installing over a network.
2. Enter:

   ```
   smit install_latest
   ```

This command invokes SMIT and takes you directly to its window for installing software.

3. Press **List**

   A window opens listing the available INPUT devices and directories for software.

4. Select the installation device or directory from the list of available INPUT devices. The window listing the available INPUT devices and directories closes and the original SMIT window indicates your selection.

5. Press **Do**

   The SMIT window displays the default installation parameters.

6. Type the appropriate file name, as shown in Table 3 or Table 4:

Table 3. IBM HPC Toolkit AIX fileset names for the platform-specific license

If you want to install:	**Type this in the SOFTWARE to install field:**
All of the IBM HPC Toolkit software	**ppedev.***
ppedev.hpct	**ppedev.hpct**
ppedev.rte	**ppedev.rte**

Table 4. IBM HPC Toolkit AIX fileset names for the multiplatform license

If you want to install:	**Type this in the SOFTWARE to install field:**
All of the IBM HPC Toolkit software	**ppedev_pa.***
ppedev_pa.hpct	**ppedev_pa.hpct**
ppedev_pa.rte	**ppedev_pa.rte**

1. After choosing the appropriate software, you might also want to change other options on the panel, as needed. For example, the panel also asks whether or not you want to expand the file systems.
2. When you are prompted, answer **yes** to expand the file systems if you want to allow the file system to be expanded.
3. Type **yes** in the **ACCEPT new license agreements** field. If the eLicense is not accepted, none of the IBM HPC Toolkit software components will be installed.
4. Press **Do**. The system installs the installation image.

Installing the IBM HPC Toolkit on Linux systems

The IBM High Performance Computing Toolkit can be installed on IBM Power Systems servers, which are supported by IBM Parallel Environment (PE) Runtime Edition for Linux.

Table 5 shows the IBM HPC Toolkit software requirements:

Table 5. Linux software requirements for IBM HPC Toolkit

Function	**Software**	**Power Red Hat Linux**	**x86 Red Hat Linux**	**x86 SLES Linux**
All	Operating system	Red Hat Enterprise Linux 6.3 or 6.4	Red Hat Enterprise Linux 6.3 or 6.4	SLES 11 SP2 or later

Table 5. Linux software requirements for IBM HPC Toolkit (continued)

Function	Software	Power Red Hat Linux	x86 Red Hat Linux	x86 SLES Linux
Compiling C applications	C compiler	IBM VAC 12.1 or later	GNU compiler version provided on the OS media or Intel C compiler version 11.1 or higher	GNU compiler version provided on the OS media or Intel C compiler version 11.1 or higher
Compiling FORTRAN applications	Fortran compiler	IBM xlf 14.1 or later	GNU compiler version provided on the OS media or Intel Fortran compiler version 11.1 or higher	GNU compiler version provided on the OS media or Intel Fortran compiler version 11.1 or higher
Using IBM HPC Toolkit GUIs	64-bit libstdc++	Version provided on the OS media	Version provided on the OS media	Version provided on the OS media
Using IBM HPC Toolkit GUIs	64-bit libXp	Version provided on the OS media	Version provided on the OS media	Version provided on the OS media
Using IBM HPC Toolkit GUIs	64-bit libXt	Version provided on the OS media	Version provided on the OS media	Version provided on the OS media
Using IBM HPC Toolkit GUIs	64-bit libXext	Version provided on the OS media	Version provided on the OS media	Version provided on the OS media
Using IBM HPC Toolkit GUIs	X11 (Xorg) runtime libraries	Version provided on the OS media	Version provided on the OS media	Version provided on the OS media
Using Xprof	32-bit libstdc++	4.1.2-14 or later	N/A	N/A
Using Xprof	32-bit libXp	1.0.0-8.1 or later	N/A	N/A
Using Xprof	32-bit libXt	1.0.2-3.1 or later	N/A	N/A
Using Xprof	32-bit libXext	1.0.1-2.1 or later	N/A	N/A
Using Xprof	32-bit xorg-x11-libs	N/A	N/A	N/A
Using Xprof	32-bit openmotif	2.3.0.0-3 or later	N/A	N/A
Using Xprof	32-bit openmotif-libs	N/A	N/A	N/A
Compile and run MPI or PAMI programs	Parallel Environment Runtime Edition	1.3.0.6 or later	1.3.0.6 or later	1.3.0.6 or later

There are two sets of installation images for the IBM HPC Toolkit, each with different names for the Linux version:

- One set uses the platform-specific license and it uses the **ppedev** prefix
- One set uses the multiplatform license and it uses the **ppedev_pa** prefix

Each set contains three installation images for the AIX version:

- **ppedev.license** and **ppedev_pa.license**

  The **ppedev.license** or **ppedev_pa.license** installation images must be installed on all the nodes where you intend to use the IBM HPC Toolkit.
- **ppedev.hpct** and **ppedev_pa.hpct**

  The **ppedev.hpct** or **ppedev_pa.hpct** installation image must be installed on any node where you intend to use the X based GUI tools, instrument an application, or compile an application where you have coded calls to the instrumentation libraries.
- **ppedev.rte** and **ppedev_pa.rte**

The **ppedev.rte** or **ppedev_pa.rte** installation image must be installed on all nodes where applications instrumented with the toolkit will run, and also on any node where the **ppedev.rte** or **ppedev_pa.rte** installation image is installed.

You must install and accept the IBM PE Developer Edition license on each node where you intend to install the IBM HPC Toolkit installation images before you can install the IBM HPC Toolkit. In addition, when you install IBM HPC Toolkit, you must select all RPMs from the same license set, either the platform-specific license or the multiplatform license.

To install the license:

1. Create a directory, for example:
   ~/images
2. Copy the following to that directory:
   ppedev_license-1.3.0-0.*arch*.rpm

   or:
   ppedev_pa_license-1.3.0-0.*arch*.rpm

   where:

   arch
   Is either ppc64 or x86_64.
3. Change the directory (**cd**) to that directory.
4. Invoke the command:
   rpm -i ppedev_license-1.3.0-0.*arch*.rpm

   or:
   rpm -i ppedev_pa_license-1.3.0-0.*arch*.rpm

   where:

   arch
   Is either ppc64 or x86_64.
5. Set the **IBM_PPEDEV_LICENSE_ACCEPT** environment variable to **no**, for example, export:
   IBM_PPEDEV_LICENSE_ACCEPT=no
6. Invoke the following command to accept the license for the platform-specific licensed installation images:
   /opt/ibmhpc/ppedev.hpct/lap/accept_ppedev_license.sh

   or invoke the following command to accept the license for the multiplatform licensed installation images:
   /opt/ibmhpc/ppedev.hpct/lapPA/accept_ppedev_license.sh
7. Accept the license once you have reviewed the license terms.
8. If you have already reviewed the IBM PE Developer Edition license, you can set:
   IBM_PPEDEV_LICENSE_ACCEPT=yes

   before invoking:
   /opt/ibmhpc/ppedev.hpct/lap/accept_ppedev_license.sh

   or:

```
/opt/ibmhpc/ppedev.hpct/lapPA/accept_ppedev_license.sh
```

in order to accept the license without reviewing it again.

Prerequisite software must be installed, following the documentation for that software's installation, before you install the IBM HPC Toolkit.

The Linux version of the IBM HPC Toolkit has several RPMs that must be installed. The RPM containing the GUIs and instrumentation tools require the run-time support RPM to be installed.

For platform-specific licensing for the IBM HPC Toolkit as shown in Table 6, the RPMs for each platform are distributed on one DVD.

Table 6. Platform specific Linux RPM file names for the IBM HPC Toolkit

Purpose	Where installed	Linux on Power RPM	X86 RPM
IBM PE Developer Edition license	login and compute nodes	ppedev_license-1.3.0-0.ppc64.rpm	ppedev_license-1.3.0-0.x86_64.rpm
IBM HPC Toolkit GUIs and instrumentation tools	login node	ppedev_hpct-1.3.0-0.ppc64.rpm	ppedev_hpct-1.3.0-0.x86_64.rpm
IBM HPC Toolkit run-time support	login and compute nodes	ppedev_runtime-1.3.0-0.ppc64.rpm	ppedev_runtime-1.3.0-0.x86_64.rpm

Multiplatform licensing for the IBM HPC Toolkit is available on the IBM Passport Advantage® web site (http://www-01.ibm.com/software/lotus/passportadvantage/pao_customer.html). The RPMs shown in Table 7 for each platform are available for individual download or are distributed on one DVD.

Table 7. Multiplatform Linux RPM file names for the IBM HPC Toolkit

Purpose	Where installed	Linux on Power RPM	X86 RPM
IBM PE Developer Edition license	login and compute nodes	ppedev_pa_license-1.3.0-0.ppc64.rpm	ppedev_pa_license-1.3.0-0.x86_64.rpm
IBM HPC Toolkit GUIs and instrumentation tools	login node	ppedev_pa_hpct-1.3.0-0.ppc64.rpm	ppedev_pa_hpct-1.3.0-0.x86_64.rpm
IBM HPC Toolkit run-time support	login and compute nodes	ppedev_pa_runtime-1.3.0-0.ppc64.rpm	ppedev_pa_runtime-1.3.0-0.x86_64.rpm

For information about the Eclipse-based client (Workbench) RPM file names, see the *IBM PE Developer Edition Eclipse Client for AIX and Linux: Installation Guide*, SC23-7030.

You can install the RPMs separately or you can install them as a group. If you install RPMs individually, you must install the run-time support RPM first, followed by the RPM containing the GUIs and instrumentation tools (on compile, link and login nodes).

To install RPMs individually, use the **rpm –i** *RPM_name* command, for example:

```
rpm –i ppedev_runtime_1.3.0-0.ppc64.rpm
```

To install RPMs as a group, copy the RPMs into a directory, for example, **~/images** then issue the command:

```
rpm –i ~/images/*.rpm
```

Installing the IBM PE Developer Edition Eclipse feature

The IBM HPC Toolkit Eclipse plug-in can be installed on any system that is supported by the IBM PE Developer Edition Eclipse client (Workbench).

Prerequisite installation

The IBM PE Developer Edition Eclipse plug-in includes installation images for Eclipse 4.3 including plug-ins for IBM HPC Toolkit, Parallel Tools Platform (PTP) and prerequisite plug-ins for both IBM HPC Toolkit and PTP. IBM PE Developer Edition also includes an Eclipse update site image for use in installing the IBM HPC Toolkit plug-in, the PTP plug-ins, and prerequisite plug-ins into an existing Eclipse 4.3 installation.

Installation instructions for a new Eclipse 4.3 installation and for installing the IBM HPC Toolkit plug-in into an existing Eclipse 4.3 installation are included in the following information units:

- *IBM PE Developer Edition Eclipse Client for AIX and Linux: Installation Guide*, SC23-7030

For information about accessing the most recent documentation, see "Prerequisite and related information" on page xii.

Part 3. The IBM HPC Toolkit Eclipse plug-in

The topics in this part provide information about using the IBM HPC Toolkit Eclipse plug-in.

© Copyright IBM Corp. 2008, 2013

Chapter 3. Using the IBM HPC Toolkit Eclipse plug-in

The IBM HPC Toolkit Eclipse plug-in is part of the IBM Parallel Environment Developer Edition and is the user interface for the IBM HPC Toolkit.

You can use it to profile your application and obtain performance measurements in the following areas:

- Hardware performance counter measurements (see Chapter 4, "Using hardware performance counters in the plug-in," on page 51)
- MPI profiling (see Chapter 5, "Using MPI profiling in the plug-in," on page 59)
- OpenMP profiling (see Chapter 6, "Using OpenMP profiling in the plug-in," on page 67)
- Application I/O profiling (see Chapter 7, "Using I/O profiling in the plug-in," on page 73)

There are two ways to profile your application and obtain performance measurements:

- You can specify your application executable and instrumentation points as part of the profile launch configuration, and have the plug-in as part of launching your application, dynamically instrument your application and run it.
- You can manually instrument your application using the **HPCT** perspective.

  You open the **HPCT** perspective by clicking the Eclipse **Window** menu, clicking the **Open Perspective** selection, clicking the **Other** selection and selecting **HPCT** from the list that is displayed. When the **HPCT** perspective opens, you can load the application executable and manually instrument the application. Then, you specify this instrumented application in the profile launch configuration so that it runs when launched.

In either case, after the execution finishes, the plug-in maps performance measurements to your source code and allows you to sort and filter that data.

Note: When using the Eclipse plug-in with x86 class Linux applications, the options to open a binary and generate an instrumented application, while present, are not supported.

Preparing your application

You must compile and link your application using the **-g** compiler flag so that the plug-in has the line number and symbol table information that it needs in order to show you the instrumentation points in your application, and so that the application can be instrumented.

Note: You must not compile your application with the **-pg** flag.

When you compile an application on a Power Linux system, you will need to use the **-Wl,--hash-style=sysv -Wl,--emit-stub-syms** compiler flags.

When you try to load a FORTRAN Linux 64-bit application, the plug-in might be unable to locate the main entry point for your application. If the plug-in cannot locate the main entry point, it will issue a message suggesting that you set the **PSIGMA_MAIN** environment variable. If you see this message, you should set

© Copyright IBM Corp. 2008, 2013

PSIGMA_MAIN to the main entry point of your application, or the name of the main program of your FORTRAN application. You can set the **PSIGMA_MAIN** environment variable by adding an entry for the **PSIGMA_MAIN** environment variable to the list in the Environment tab of the profile configuration dialog (see Figure 6 on page 27).

If you are running your application on an AIX system and your application is linked with additional libraries other than **libpmapi**, **libpthreads**, or **libxlsmp,** then you must specify the additional libraries in the **HPCT Preferences** dialog. If your application was linked using non-default linker flags, you need to set those flags in the **HPCT Preferences** dialog.

The **HPCT Preferences** dialog is opened by clicking the Eclipse **Window** menu, clicking the **Preferences** menu option, and selecting the **HPCT** entry from the left side of the preferences dialog. Enter your non-default linker flags into the **Linking options for AIX** field.

Note: Do not change the path to the HPC Toolkit installation or the SCI installation unless you are directed to do so by IBM support.

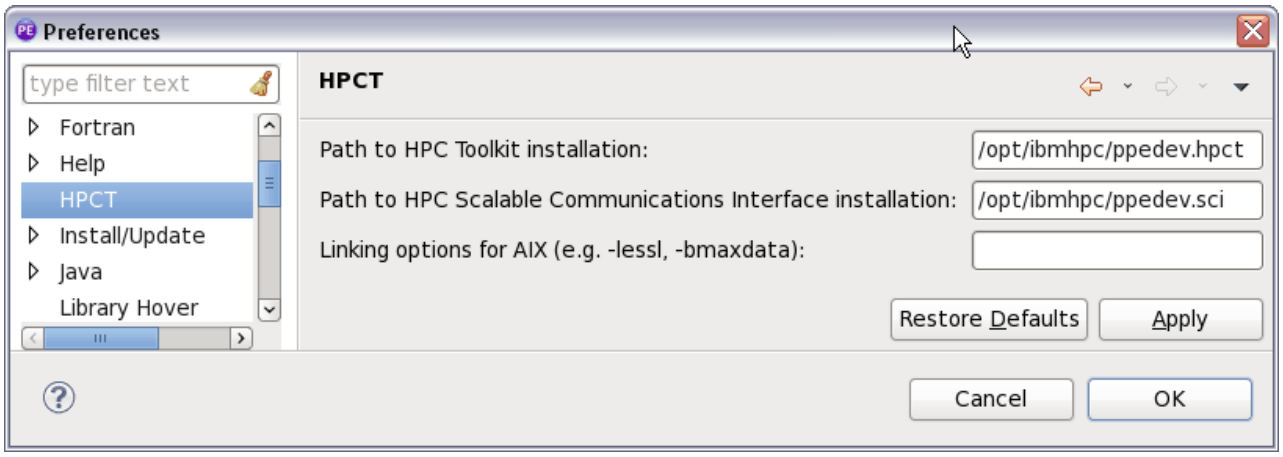


Figure 1. HPCT Preferences dialog

Instrumenting your application at launch

Assume that you have created a synchronized project called, **hpm_Linux_ppc64** using Parallel Tools Platform (PTP) and compiled your application as described in "Preparing your application" on page 21. Now you can run your application.

Running the application

You can use the Eclipse **Profile Configurations** dialog to both instrument and run your parallel application. This dialog provides panels where you can create a profile configuration containing the required settings to both instrument and run your application. These settings are saved within Eclipse so that you can run the application multiple times without having to enter application settings again.

You open the **Profile Configurations** dialog by clicking the drop-down list next to the **Profile** icon on the Eclipse main toolbar or by clicking the **Profile Configurations** option under the **Run** menu in the Eclipse main menu.

Figure 2 shows how the **Profile Configurations** dialog initially looks:

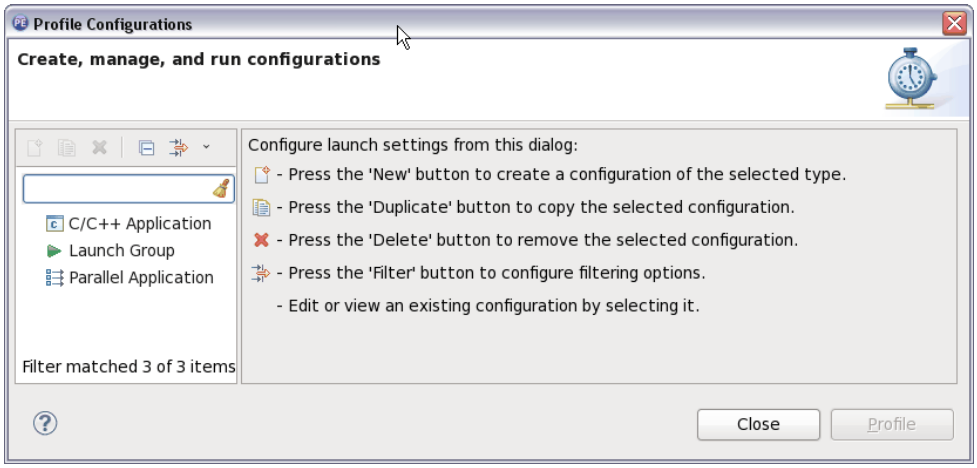


Figure 2. Profile Configurations dialog

If the profile configuration for your application does not exist, you can create a new parallel performance configuration (profile configuration) from this dialog. To do this, select **Parallel Application** from the list on the left side of the dialog, then click the **New** icon just above that list. You can have as many profile configurations as you want, and can have multiple profile configurations for the same application each with different instrumentation points.

If you have created a run configuration for your application, that run configuration is displayed in the list of available profile configurations. The run configuration can be reused for performance analysis runs and it saves the additional information required to profile applications.

Once you have created a profile configuration, you must enter data in the **Resources**, **Application**, **Arguments**, **Environment**, and **Performance Analysis** tabs before running your application. There must not be any errors on any of the other tabs within the configuration, where any tab has an error icon within the tab.

The **Resources** tab is used to specify the IBM PE target system configuration to be used and the IBM PE options required to run the application. Figure 3 on page 24 shows the **Resources tab**:

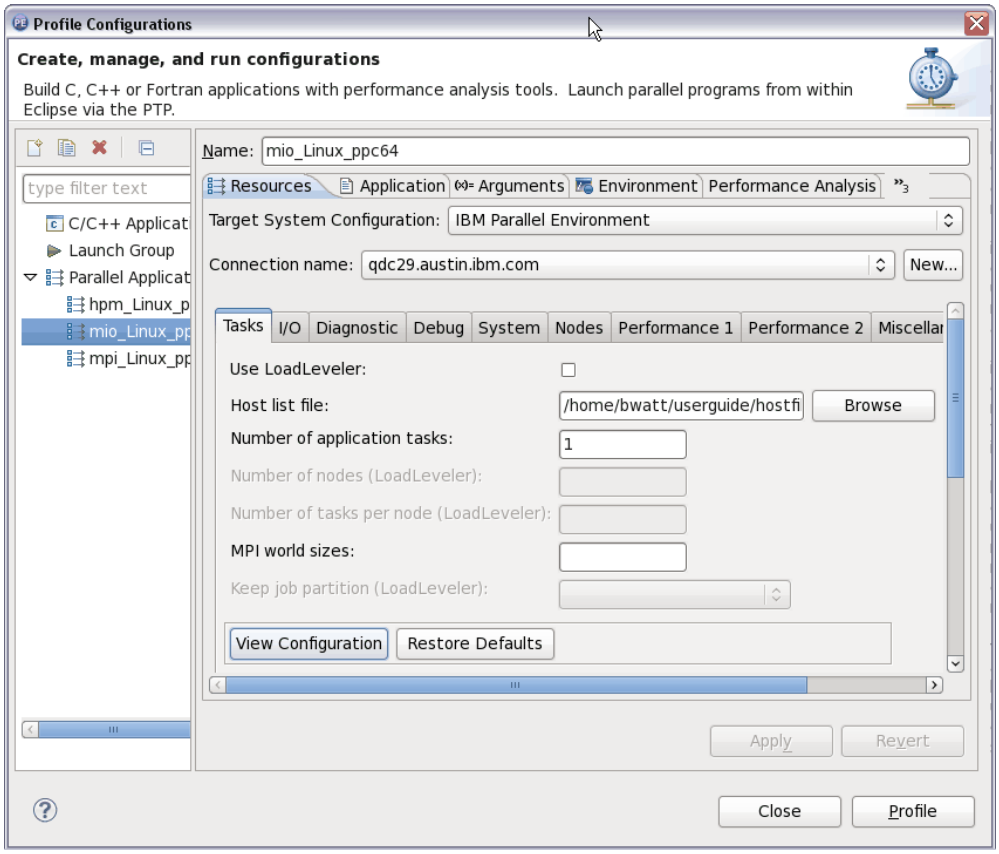


Figure 3. Resources tab

The first step in filling out the **Resources** tab is to select a Target System Configuration from the drop-down list at the top of the panel. Select IBM Parallel Environment from the **Target System Configuration** drop-down list. Next, select a connection name from the **Connection name** drop-down list or create a new connection by clicking the **New** button next to the drop-down. Then, you can enter the required IBM PE run-time options by clicking the tabs on the panel and filling in the desired values.

Figure 4 on page 25 shows the **Application** tab, which specifies the application program:

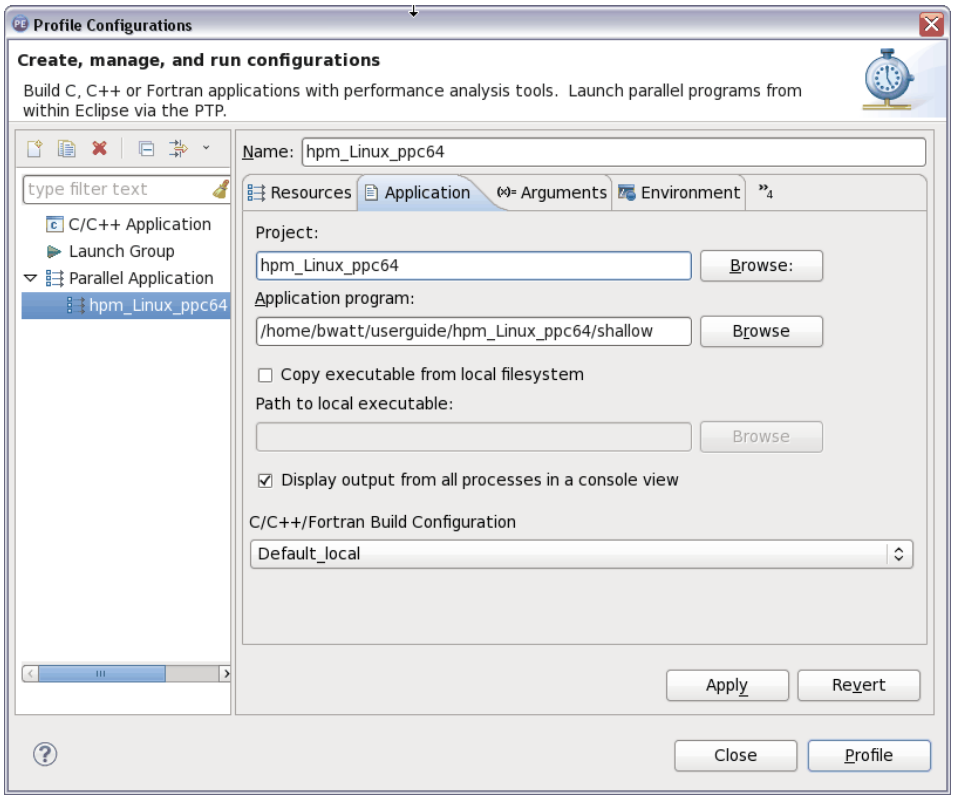


Figure 4. Application tab

The **Application** tab requires you to specify the project containing your application. Select the project by clicking the **Browse** button next to the project name and select your application's project from the list. The project that you select must have at least one build configuration defined, and you must select a build configuration in the **Application** tab.

The **Application program** field specifies the absolute path name to your executable on the target system. When you select your application, make sure that you *do not* select the instrumented version, which will have **.inst** appended to the original application name.

The next tab you fill out is the **Arguments** tab, which looks like Figure 5 on page 26. Specify any application program arguments in the **Parallel program arguments** field.

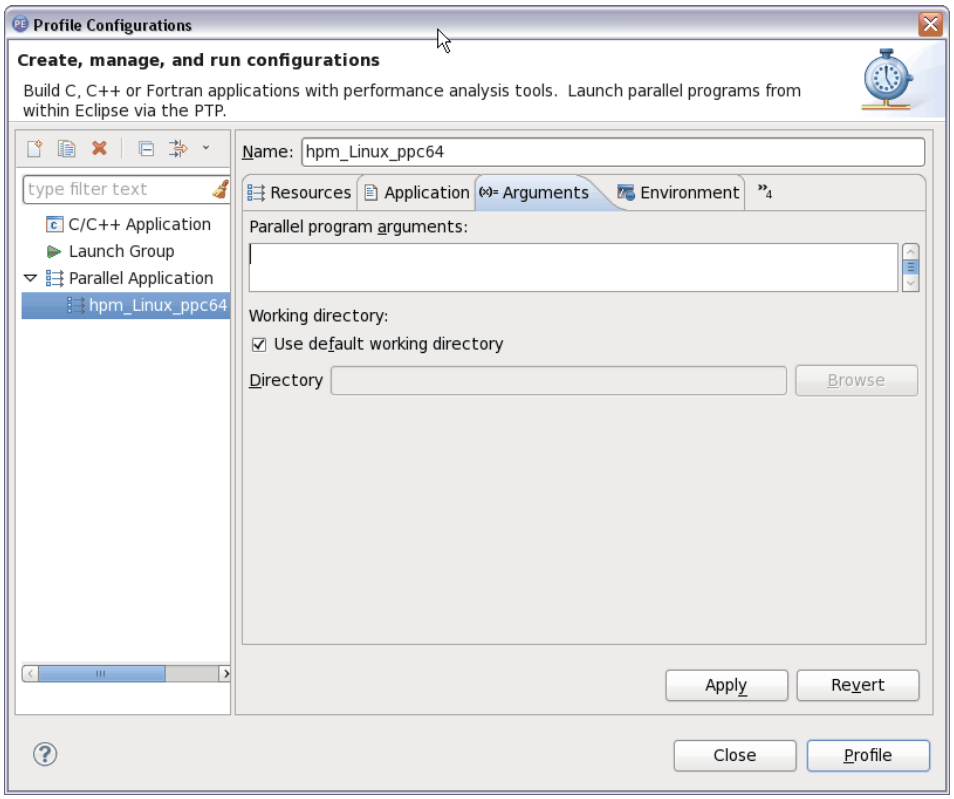


Figure 5. Arguments tab

Next, fill out the **Environment** tab which looks like Figure 6 on page 27, where you specify environment variables used by your application. You can add environment variables to the set passed to your application by clicking the **New** button and filling in the environment variable name and value in the pop-up dialog.

You can modify existing environment variables by clicking the environment variable in the list, clicking **Edit**, and modifying the setting in the pop-up dialog. You can remove environment variables from the set passed to your application by clicking the environment variable name in the list and clicking **Remove**.

Note: Do not set IBM PE environment variables or IBM HPC Toolkit environment variables in the **Environment** tab because those settings might conflict with values set on other panels.

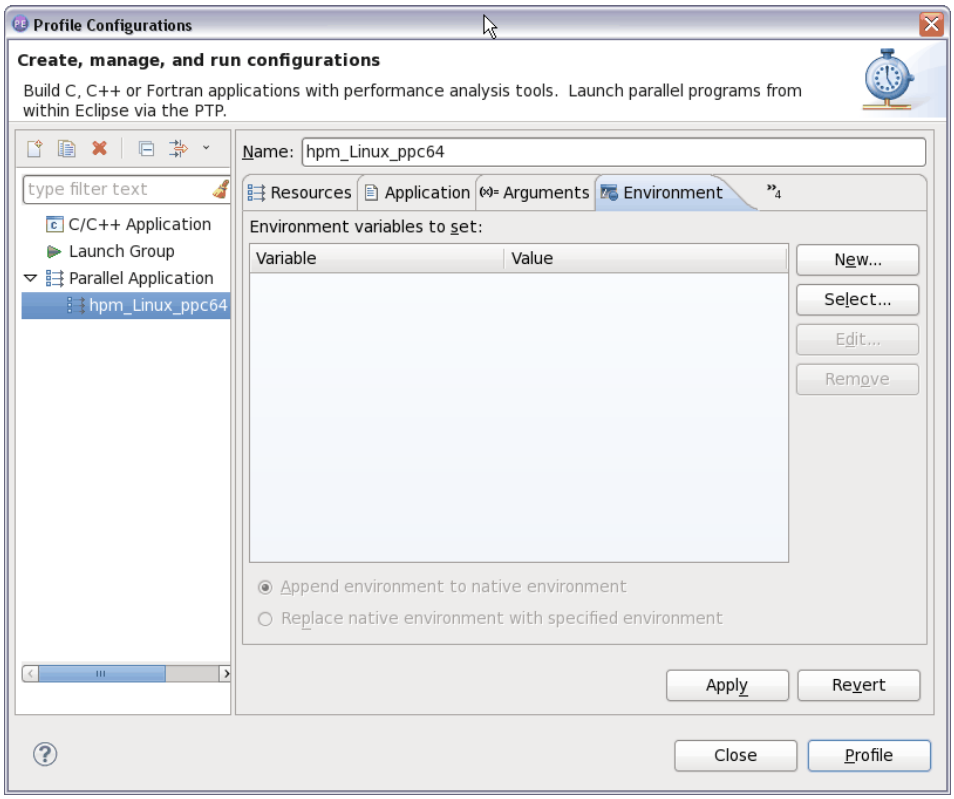


Figure 6. Environment tab

Next, fill out the **Performance Analysis** tab. This tab specifies the IBM HPC Toolkit run-time options and it looks like Figure 7 on page 28. To use the IBM HPC Toolkit plug-in, you must select **HPC Toolkit** from the **Select tool** drop-down list.

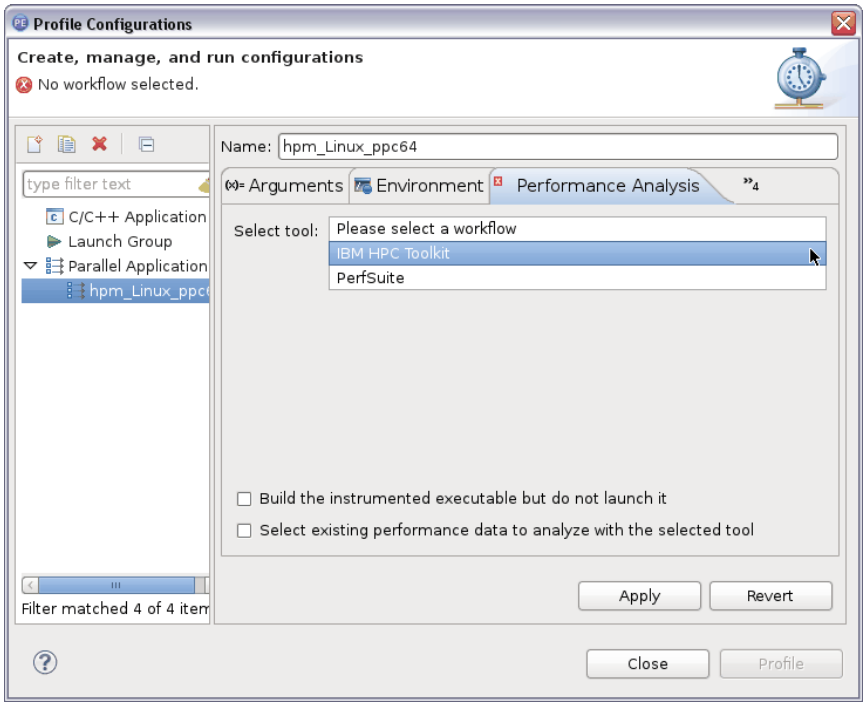


Figure 7. Performance Analysis tab

Once selected the application is analyzed, which must be compiled as described in "Preparing your application" on page 21, the rest of the tab is filled out and you can click on the **Common**, **HPM**, **MPI**, **OpenMP**, or **MIO** tabs to set the instrumentation points and run-time options for all tools as required:

- Hardware performance counter tool
- MPI trace tool
- OpenMP analysis tool
- The I/O analysis tool

For applications that have a large number of tasks, you can reduce the amount of performance data generated and collected by specifying settings on the **Common** tab within the **HPC Toolkit** tab of the **Profile Configurations** dialog. The **Common** tab looks like Figure 8 on page 29.

Enter the performance data file name into the **Output file name** field.

To generate unique filenames, check the **Generate unique filenames** checkbox.

Note: For more information about file names for the **Output file name** or **Generate unique filenames** fields, see Appendix A, "Performance data file naming," on page 309.

If you want to collect performance data for a subset of application tasks, then check the **Use hpcrun** checkbox.

When performance data is collected for a subset of application tasks, you must specify the collection criteria and number of tasks.

To specify the collection criteria, select **ELAPSED_TIME** to determine the subset by elapsed wall clock time or **CPU_TIME** to determine the subset by CPU time.

Specify the number of application tasks that performance data will be collected for by filling in the **Exception task count** field. The plug-in will select the number of tasks with the minimum value for the selected collection criteria, the number of tasks with the maximum value for the selected collection criteria, the task closest to the average for the selection criteria, and task zero.

When collecting traces for MPI tracing or I/O profiling, you must select the collection model to use by specifying either **memory** or the path names for a pair of temporary files to hold MPI trace data and I/O trace data in the form *MPI_temporary trace*, I/O *temporary trace* in the **Trace collection mode** field. If you specify **memory**, then the trace for each application task is stored in application memory until **MPI_Finalize** is called. If you specify a pair of path names, then trace data is temporarily collected in those files until the application exits, then the final trace files are generated.

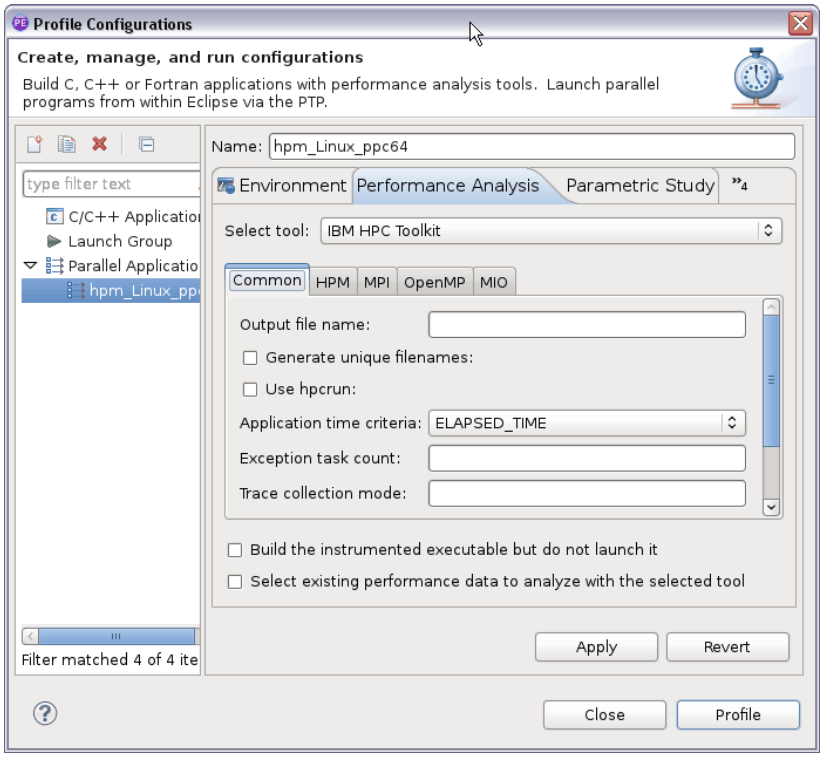


Figure 8. Common tab

The tool option tab for each tool looks similar to Figure 9 on page 30 of the HPM options pane.

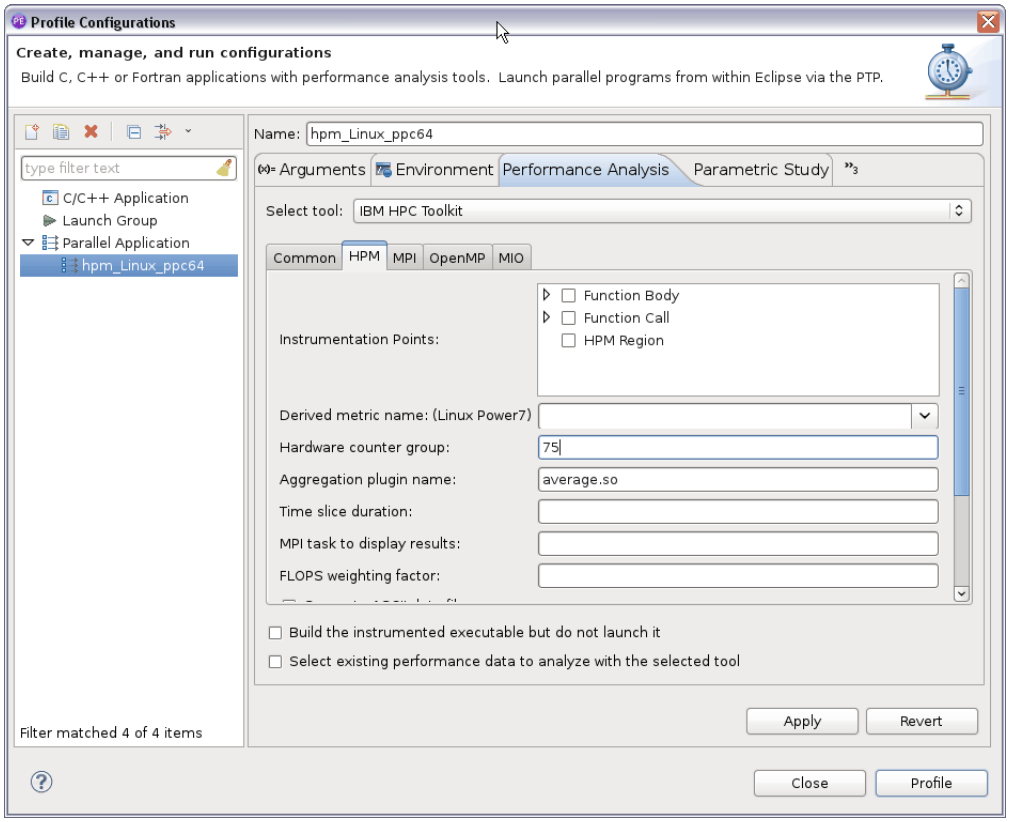


Figure 9. HPM options pane

The **HPM**, **MPI**, **OpenMP**, and **MIO** tabs are updated with a tree showing the possible instrumentation points for each type of instrumentation. You can expand and collapse nodes in the instrumentation point trees in each of these tabs by clicking the **+** and **-** icons next to nodes in the tree. You can expand or collapse all the nodes in the tree by clicking the **Expand** or **Collapse** buttons, respectively, as well as by clicking the **Expand** or **Collapse** options from the view's drop-down menu or the view's pop-up menu. For example, the HPM instrumentation view can be expanded to look like Figure 10 on page 31. Select the checkboxes in the tree to specify one or more instrumentation points.

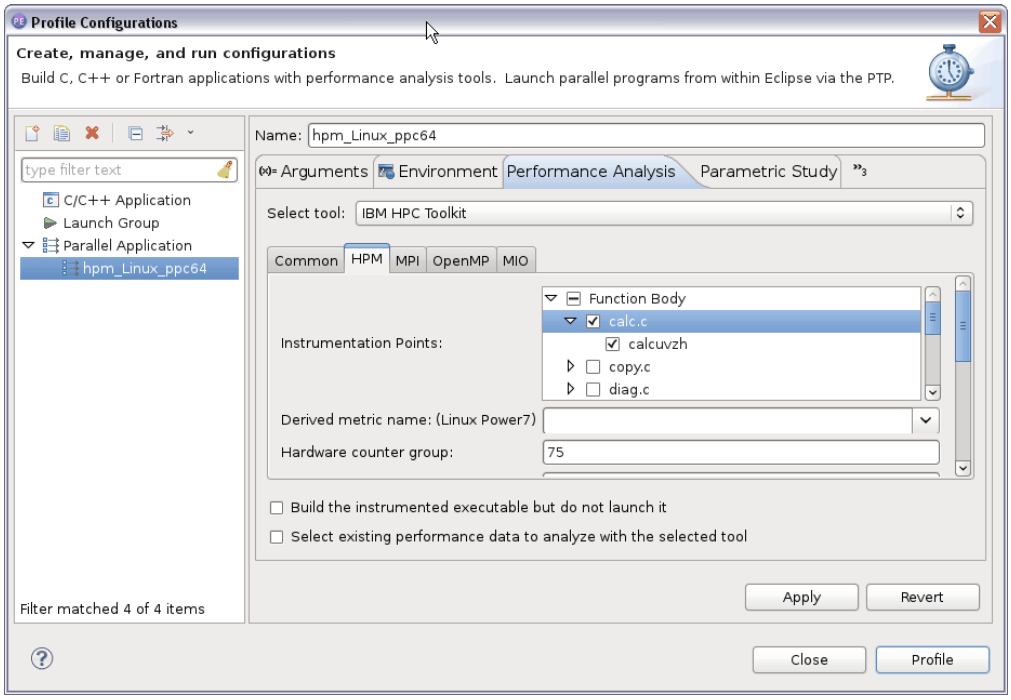


Figure 10. HPM instrumentation view

Once you have filled in all the panels in the **Profile Configurations** dialog and have no error messages displayed, you can run your application by clicking the **Profile** button. Your application runs using the specified settings. When your application completes, Eclipse displays a pop-up dialog asking you if you want to view the performance data. If you click **Yes**, then Eclipse switches to the **HPCT** perspective and adds the visualization files just generated to the list in the **Open Performance Data** view in that perspective.

For more information about running a parallel application using PTP, see the Parallel Development User Guide (http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.ptp.doc.user/html/toc.html) or use the Help Menu by selecting **Help ▸ Help Contents**.

Instrumenting your application before launch

When you open the HPCT perspective, your Eclipse window will look similar to Figure 11 on page 32, assuming you have created a synchronized project called **hpm_Linux_ppc64** using PTP and have compiled your application. Now load your application.

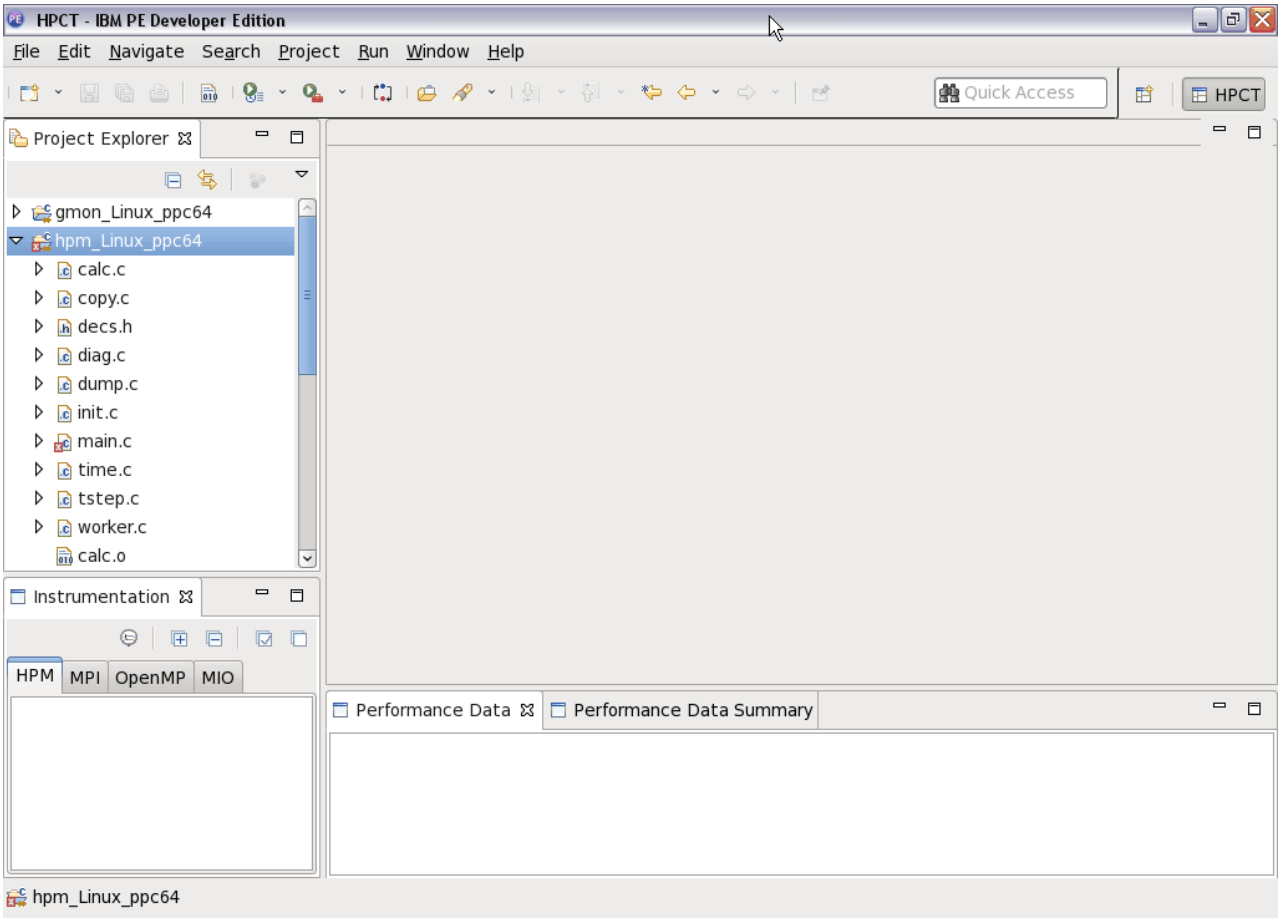


Figure 11. Initial HPCT perspective

Loading the application

Load the application, which must be compiled as described in "Preparing your application" on page 21 by right clicking over the executable name in the **Project Explorer** view, then clicking **HPCT** in the pop-up menu that is displayed, and by clicking **Open executable** in the second pop-up menu.

The Eclipse window will look like Figure 12 on page 33 after the executable is opened.

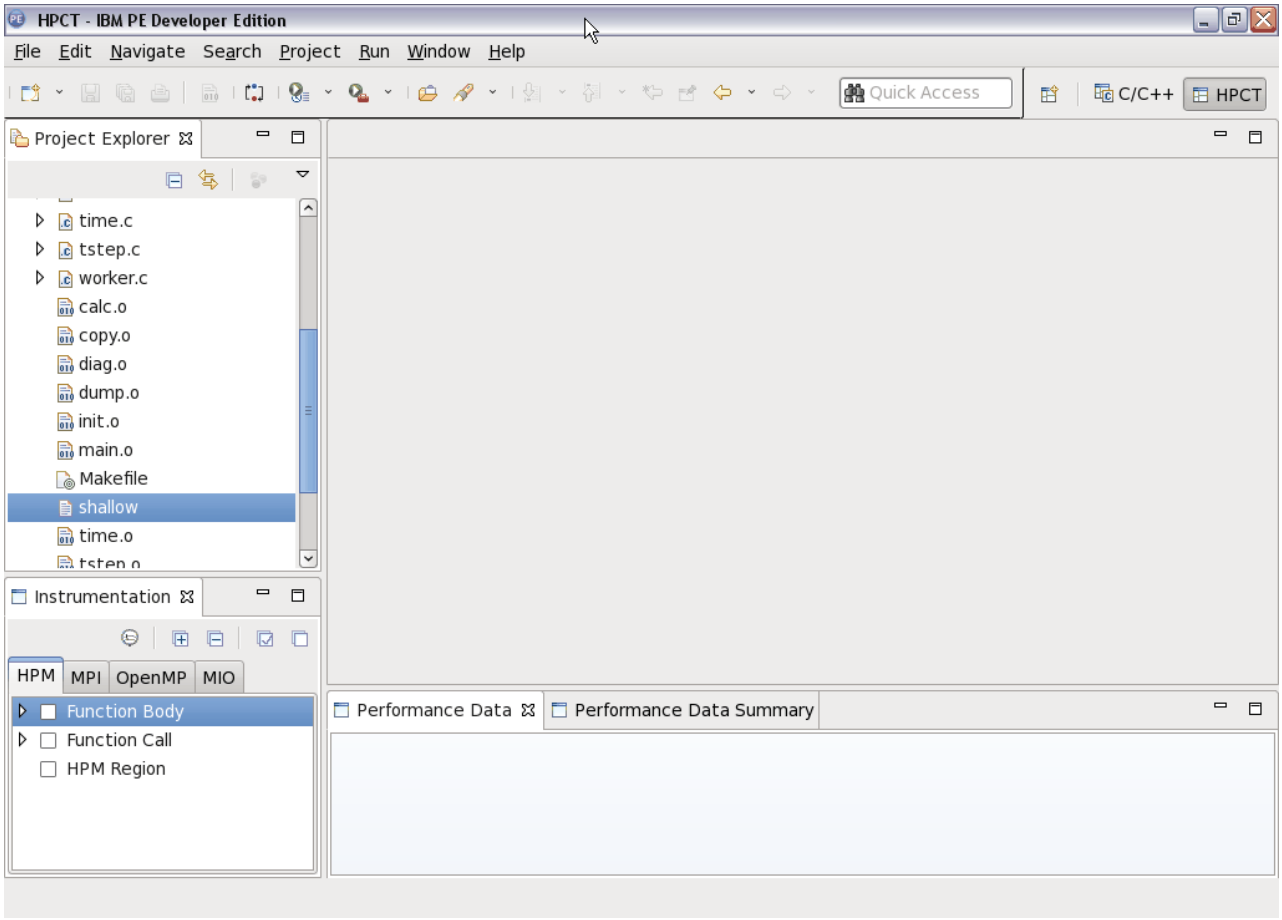


Figure 12. HPCT perspective after opening the application

Once the application has been opened, the HPCT perspective will look like Figure 13 on page 34 after you click the application path name. If you double click a file name in the Instrumentation view, the plug-in will attempt to open the selected file in an editor window as shown in Figure 13 on page 34.

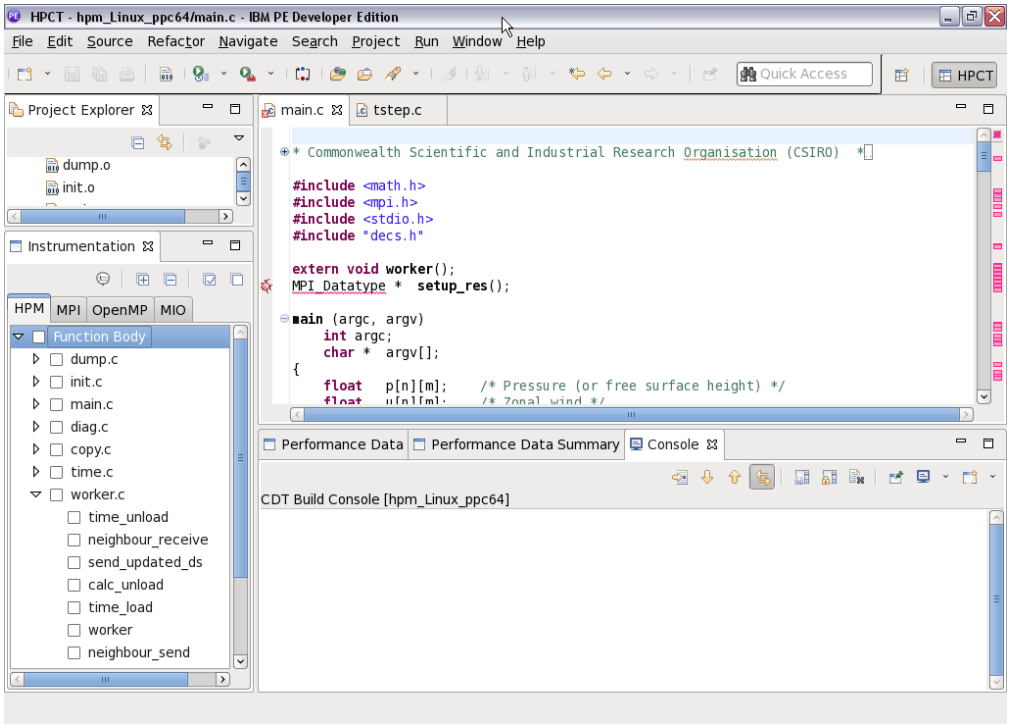


Figure 13. Expanded HPM instrumentation view

Instrumenting your application

Once the application has been opened, you can instrument the application (see Figure 13).

When you select the application to be instrumented, the **HPM**, **MPI**, **OpenMP**, and **MIO** tabs in the Instrumentation view are updated with a tree showing the possible instrumentation points for each type of instrumentation. You can expand and collapse nodes in the instrumentation point trees in each of these tabs by clicking the **+** and **–** icons next to nodes in the tree. You can expand or collapse all the nodes in the tree by clicking the **Expand** or **Collapse** buttons, respectively, as well as by clicking the **Expand** or **Collapse** options from the view's drop-down menu or the view's pop-up menu. For example, the HPM instrumentation view can be expanded to look like Figure 13. Select the checkboxes in the tree to specify one or more instrumentation points. Then select the **Instrument executable** icon or the **Instrument executable** option from the view's pop-up menu to instrument your application.

Running the application

Once you have an instrumented application, you can run the application in one of two ways:

- Using the **Profile Configurations** dialog
- Using a command shell on the target system

Using the Profile Configurations dialog

You can use the Eclipse **Profile Configurations** dialog to run your instrumented parallel application. This dialog provides panels where you can create a profile configuration containing the required settings to run your application. These settings are saved within Eclipse so that you can run the application multiple times without having to enter application settings again.

You open the **Profile Configurations** dialog by clicking the drop-down list next to the profile icon on the Eclipse main toolbar or by clicking the **Profile Configurations** option under the **Run** menu in the Eclipse main menu. The dialog initially looks like Figure 14:

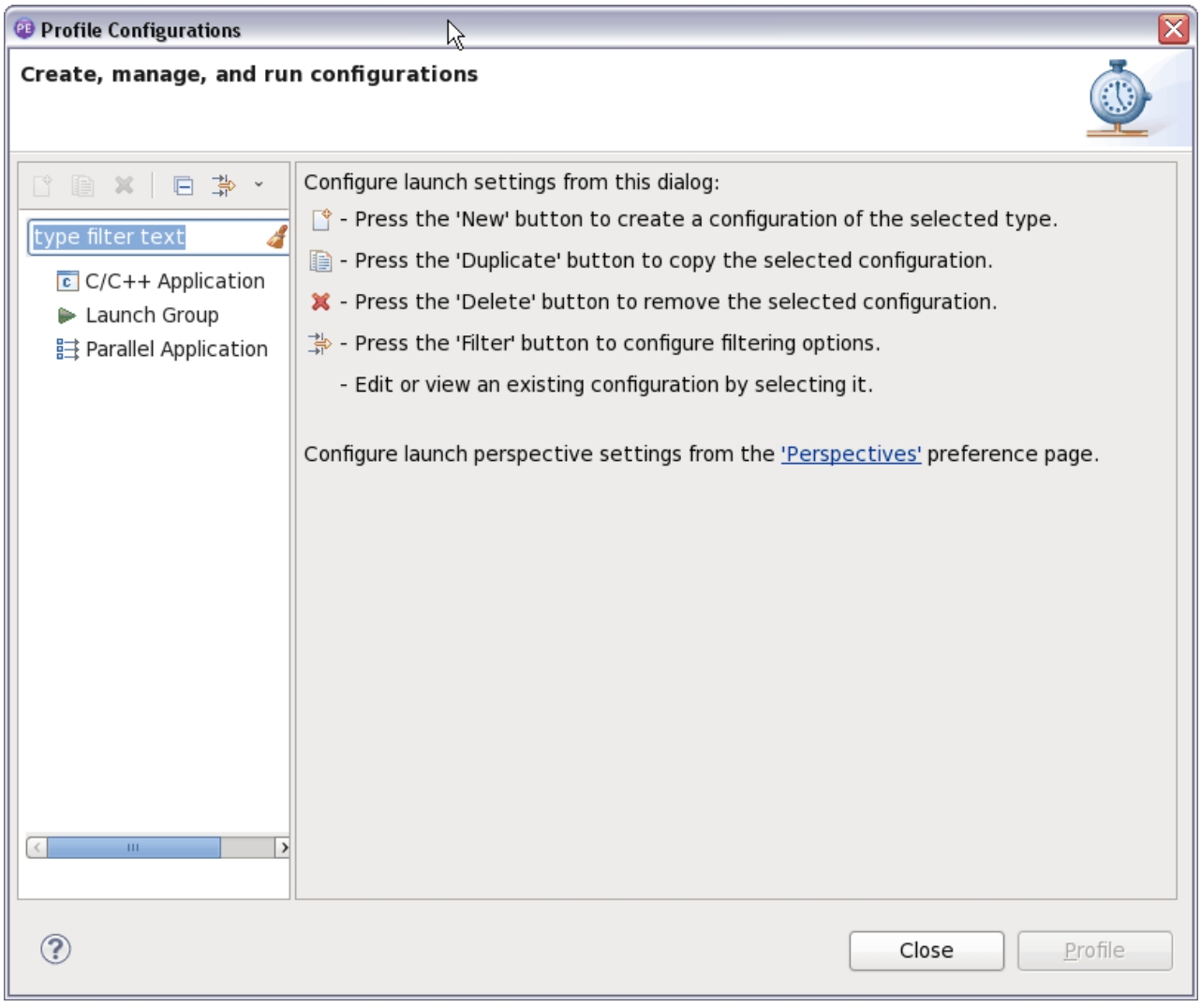


Figure 14. Initial Profile Configurations dialog

If the profile configuration for your application does not exist, you can create a new parallel performance configuration (profile configuration) from this dialog. To do this, select **Parallel Application** from the list on the left side of the dialog, then click the **New** icon just above that list. You can have as many profile configurations as you want, and can have multiple profile configurations for the same application.

If you have created a run configuration for your application, that run configuration will appear in the list of available profile configurations. The run configuration can be reused for performance analysis runs and it will save the additional information required to profile applications.

Once you have created a profile configuration, you must enter data in the **Resources**, **Application**, **Arguments**, **Environment**, and **Performance Analysis** tabs before running your application. There must not be any errors on any of the other tabs within the configuration where any tab that has an error will have an error icon within the tab.

The **Resources** tab is used to specify the IBM PE target system configuration to be used and the IBM PE options required to run the application. The **Resources** tab looks like Figure 15:

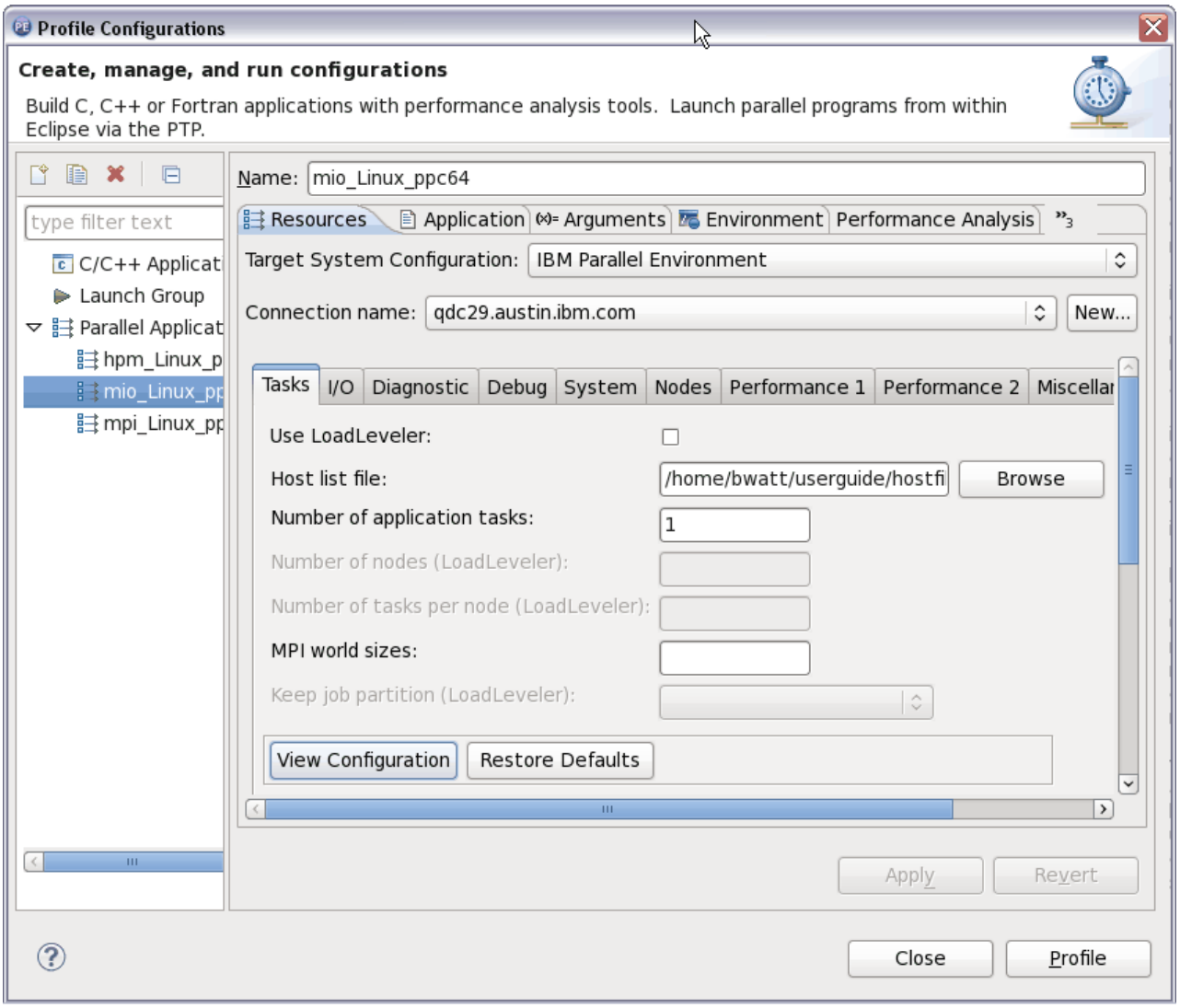


Figure 15. Profile Configurations dialog Resources tab

The first step in filling out the **Resources** tab is to select a Target System Configuration from the drop-down list at the top of the panel. Select IBM Parallel Environment from the **Target System Configuration** drop-down list. Then select a

connection name from the **Connection name** drop-down list or create a new connection by clicking the **New** button next to the drop-down list. Then you can enter the required IBM PE run-time options by clicking the tabs on the panel and filling in the desired values.

The **Application** specifies the application program, and it looks like Figure 16:

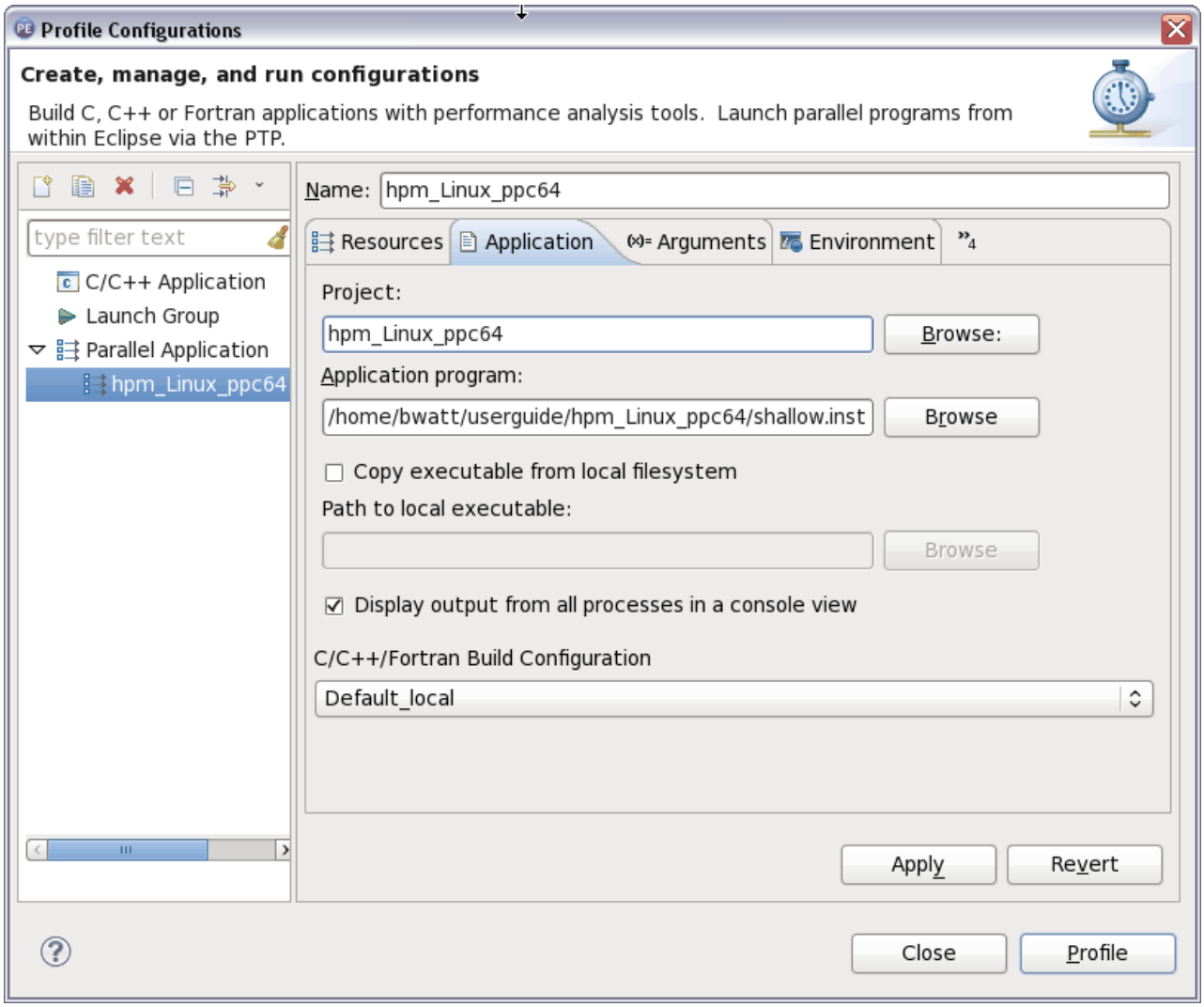


Figure 16. Profile Configurations dialog Application tab

The **Application** tab requires you to specify the project containing your application. Select the project by clicking the **Browse** button next to the project name and select your application's project from the list. The project that you select must have at least one build configuration defined, and you must select a build configuration in the **Application** tab.

The **Application program** field specifies the absolute path name to your executable on the target system. When you select your application, make sure that you select the instrumented version, which will have **.inst** appended to the original application name.

The next tab you fill out is the **Arguments** tab, which looks like Figure 17. Specify any application program arguments in the **Parallel program arguments** field.

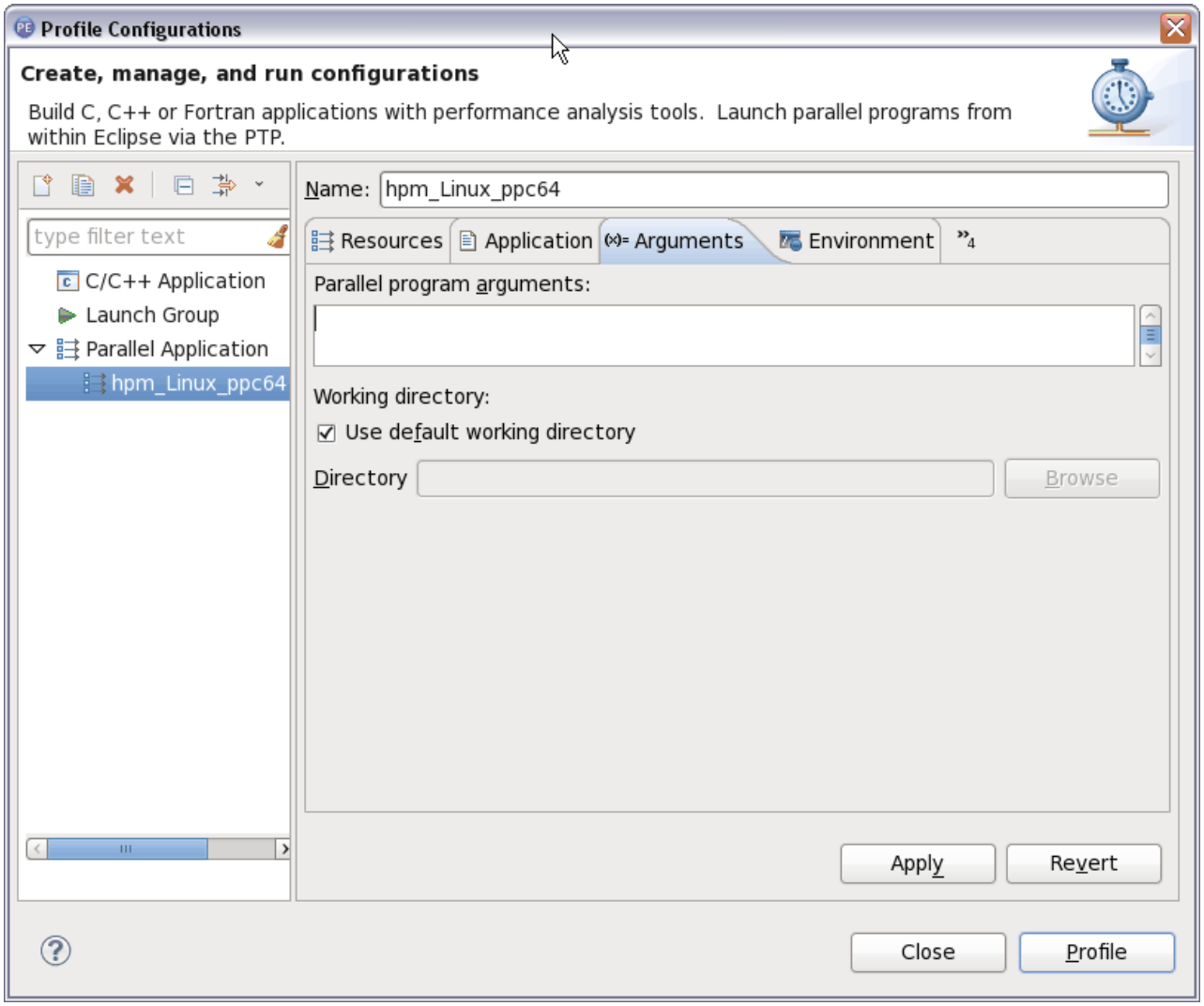


Figure 17. Profile Configurations dialog Arguments tab

Next, fill out the **Environment** tab, which looks like Figure 18 on page 39 where you specify environment variables used by your application. You add environment variables to the set passed to your application by clicking the **New** button and filling in the environment variable name and value in the pop-up dialog.

You can modify existing environment variables by clicking the environment variable in the list, clicking **Edit** and modifying the setting in the pop-up dialog. You can remove environment variables from the set passed to your application by clicking the environment variable name in the list and clicking **Remove**.

You should not set IBM PE environment variables or IBM HPC Toolkit environment variables in the **Environment** tab because those settings might conflict with values set on other panels.

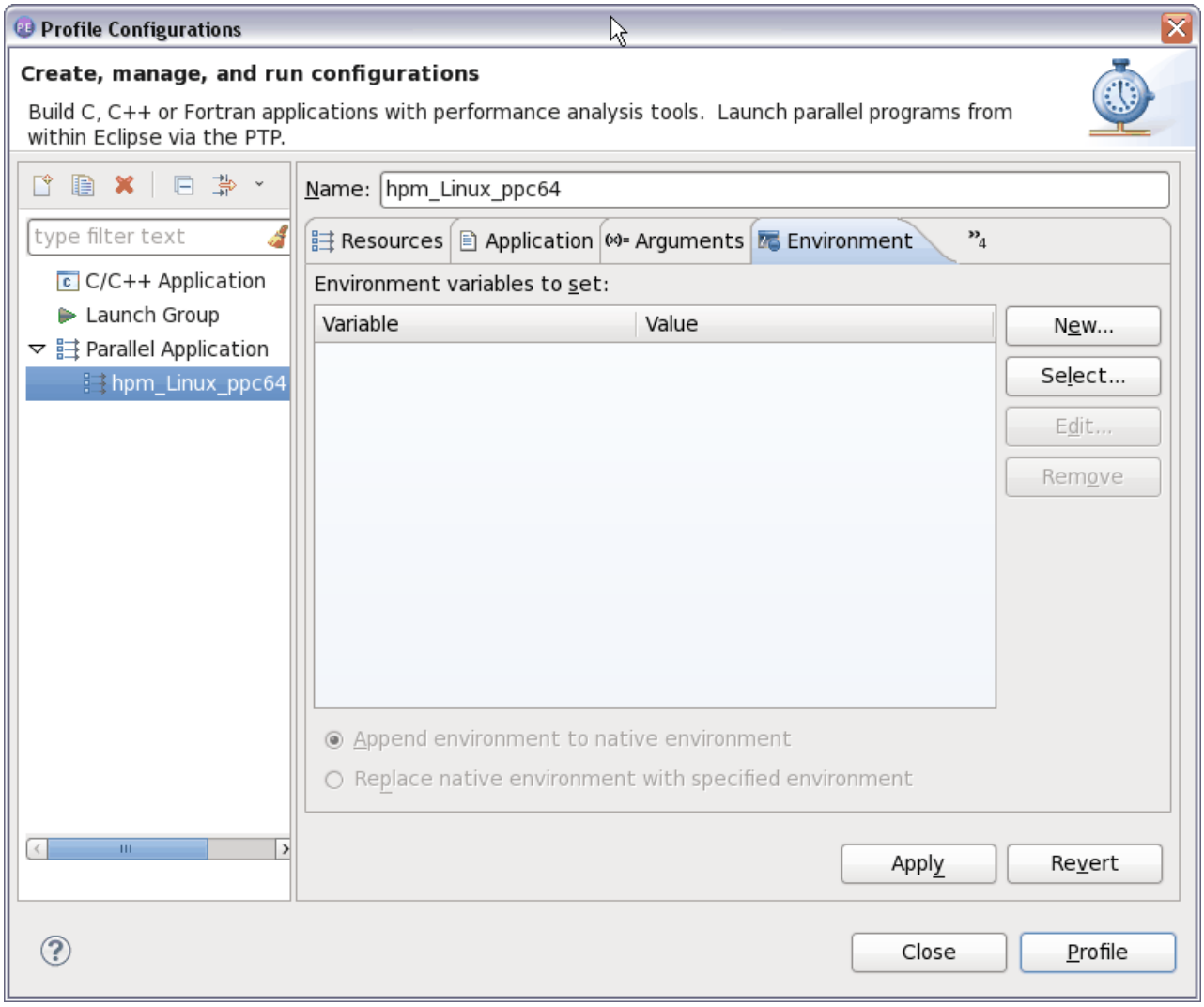


Figure 18. Profile Configurations dialog Environment tab

Next, fill out the **Performance Analysis** tab. This tab specifies the IBM HPC Toolkit run-time options and it looks like Figure 19 on page 40. To use the IBM HPC Toolkit plug-in, you must select **HPC Toolkit** from the **Select tool** drop-down list.

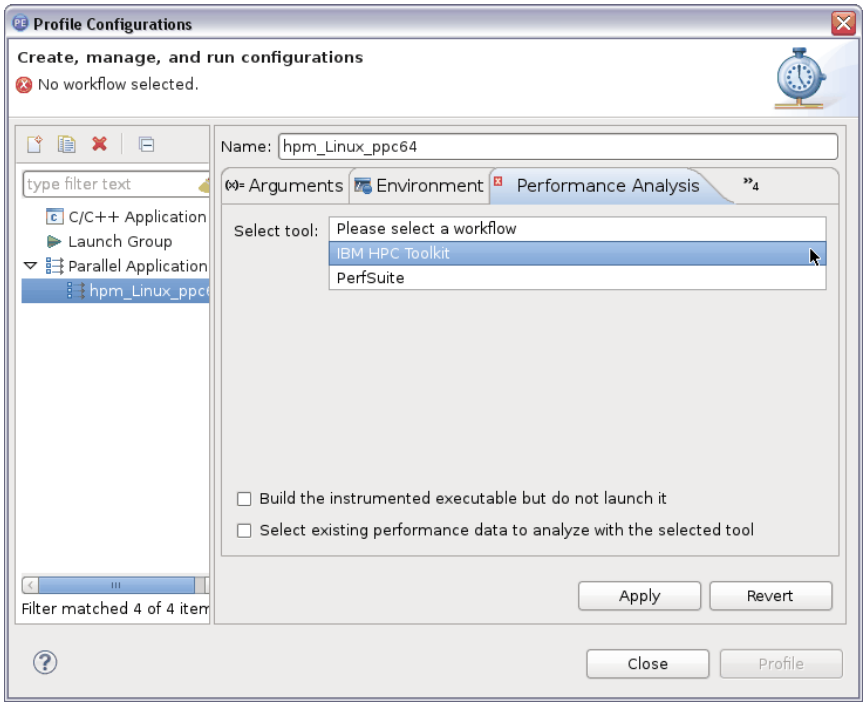


Figure 19. Profile Configurations dialog Performance Analysis tab

Once selected, the tab is filled out and you can click the **Common**, **HPM**, **MPI**, **OpenMP**, or **MIO** tabs to set the run-time options for all data collection tools as required:

- The hardware performance counter tool
- The MPI trace tool
- OpenMP analysis tool
- The I/O analysis tool

For applications that have a large number of tasks, you can reduce the amount of performance data generated and collected by specifying settings on the **Common** tab within the **HPC Toolkit** tab of the **Profile Configurations** dialog. The **Common** tab looks like Figure 8 on page 29.

Enter the performance data file name into **Output file name** field.

To generate unique filenames, check the **Generate unique filenames** checkbox.

Note: For more information about file names for the **Output file name** or **Generate unique filenames** fields, see Appendix A, "Performance data file naming," on page 309.

If you want to collect performance data for a subset of application tasks, then check the **Use hpcrun** checkbox.

When performance data is collected for a subset of application tasks, you must specify the collection criteria and number of tasks.

To specify the collection criteria, select **ELAPSED_TIME** to determine the subset by elapsed wall clock time or **CPU_TIME** to determine the subset by CPU time.

Specify the number of application tasks that performance data will be collected for by filling in the **Exception task count** field. The plug-in will select the number of tasks with the minimum value for the selected collection criteria, the number of tasks with the maximum value for the selected collection criteria, the task closest to the average for the selection criteria, and task zero.

When collecting traces for MPI tracing or I/O profiling, you must select the collection model to use by specifying either **memory** or the pathnames for a pair of temporary files to hold MPI trace data and I/O trace data in the form *MPI_temporary trace*, *I/O temporary trace* in the **Trace collection mode** field. If you specify **memory**, then the trace for each application task is stored in application memory until **MPI_Finalize** is called. If you specify a pair of path names, then trace data is temporarily collected in those files until the application exits, then the final trace files are generated.

The tool option tab for each tool looks similar to Figure 20 of the **HPM** options pane.

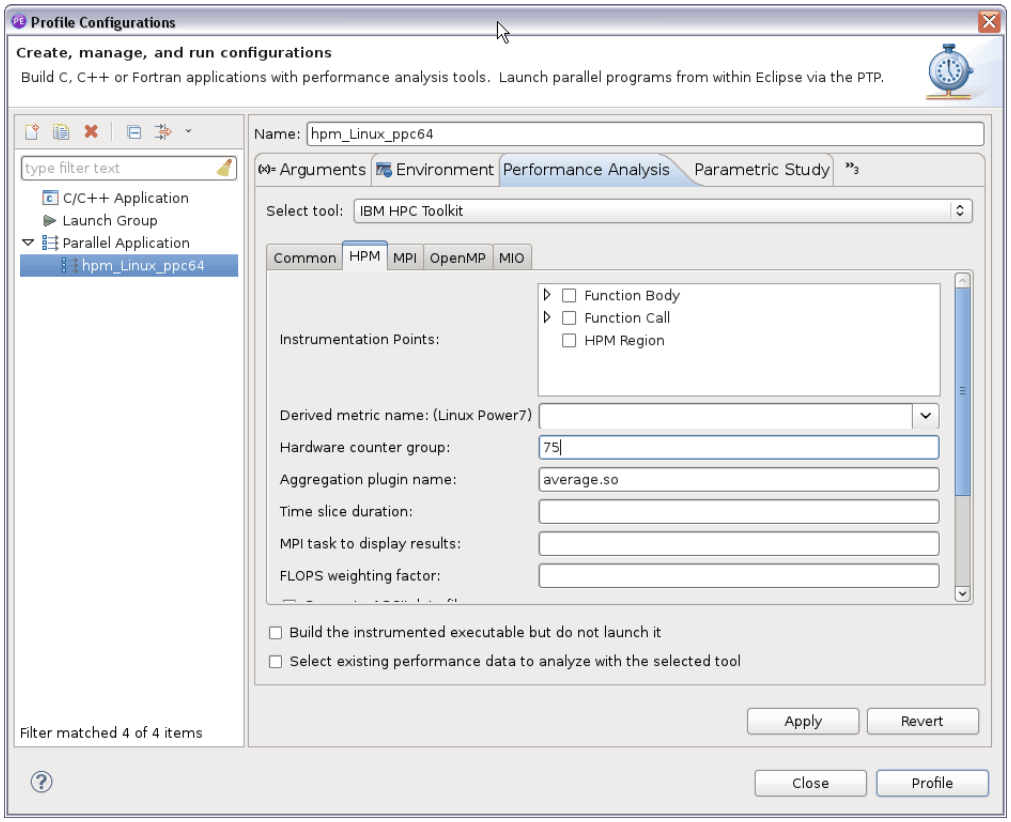


Figure 20. Profile Configurations dialog HPM settings tab

Figure 10 on page 31 shows the **HPM**, **MPI**, **OpenMP**, and **MIO** tabs that show run-time options.

Note: Do not select any instrumentation points, because the application executable is already instrumented.

Once you have filled in all the panels in the **Profile Configurations** dialog and have no error messages displayed, you can run your application by clicking the

Profile button. Your application will run using the specified settings. When your application completes, Eclipse will display a pop-up dialog asking you if you want to view the performance data. If you click **Yes**, then Eclipse will switch to the **HPCT** perspective and add the visualization files just generated to the list in the **Open Performance Data** view in that perspective.

For more information about running a parallel application using PTP, see the Parallel Development User Guide (http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.ptp.doc.user/html/toc.html) or use the Help Menu by selecting **Help ▸ Help Contents**.

Viewing performance data

The plug-in adds the performance data files created by your application to the list in the **Performance Data** view and to the list of files for your application project in the **Project Explorer** view automatically. If you did not respond **yes** to the pop-up dialog asking if the performance data files should be loaded, then you must add them manually.

You add performance data files to the list in the **Performance Data** view by selecting one or more performance data files (**.viz** files) then right clicking over the name of one of the selected files, clicking **HPCT** from the pop-up menu and then selecting **Open performance files** from the second pop-up menu. Multiple files can be selected by clicking the first selected file, then by pressing the **Shift** key and clicking the last file in the set of files you want to open or by pressing the **Ctrl** key and clicking over each individual file you want to add to the set of performance data files to open. When you are opening a set of performance data files, all selected files must have the **.viz** extension.

Once the list is updated, the Eclipse window looks like Figure 21 on page 43.

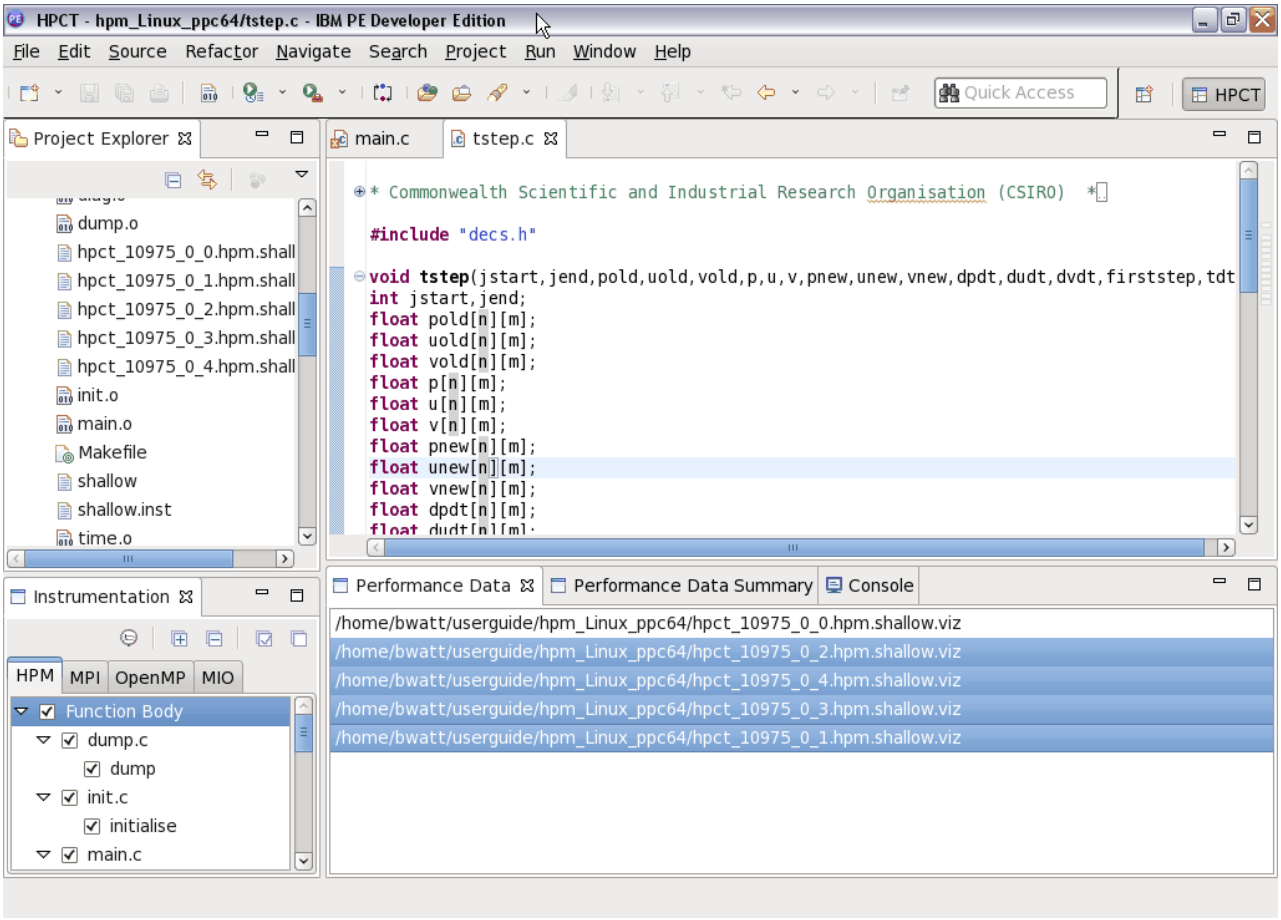


Figure 21. HPCT perspective after loading performance data

You view performance data files by clicking one or more file names in the **Performance Data** view list. The top selected file in the list is displayed in the **Performance Data Summary** view. After the files are opened, the Eclipse window looks like Figure 22.

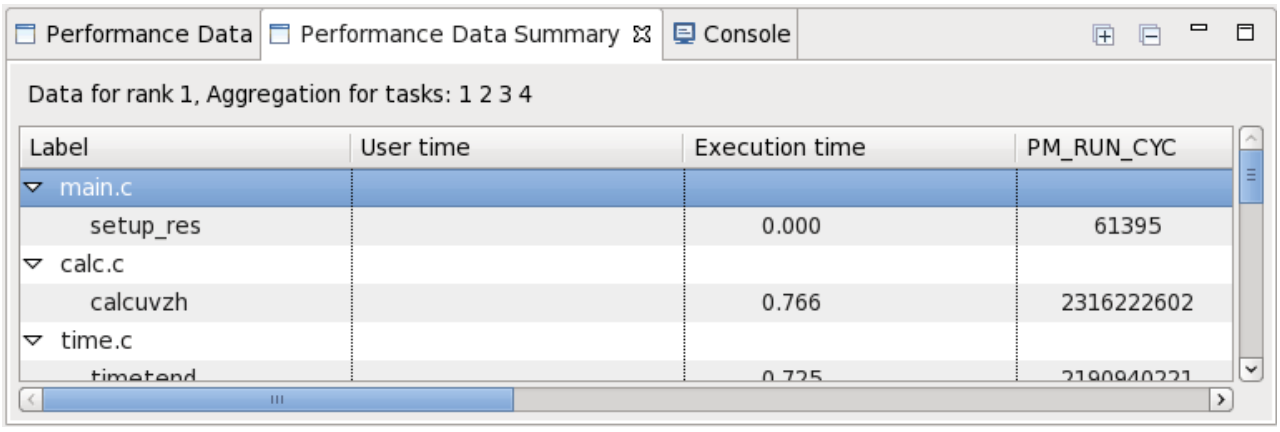
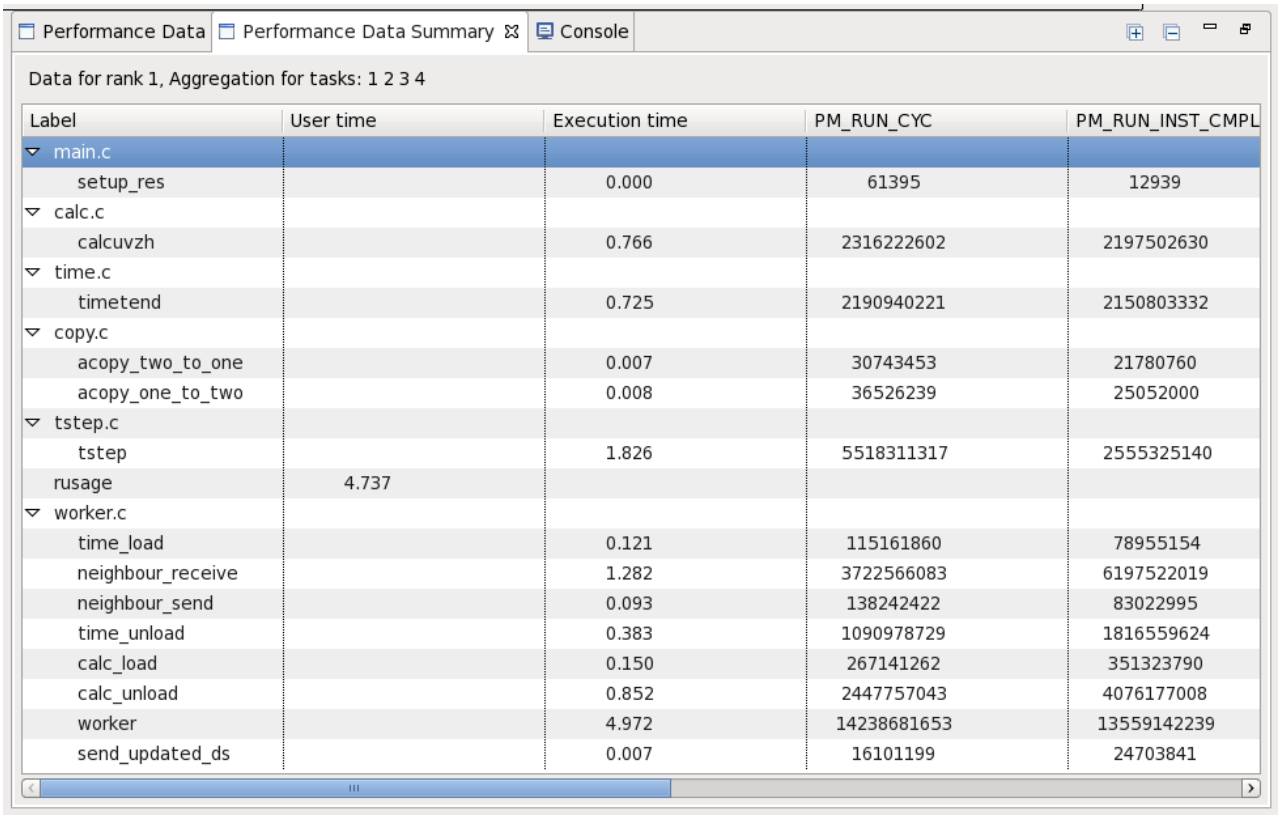


Figure 22. Performance Data Summary

If multiple files are selected, the data from the top selection will be displayed in the **Performance Data Summary** view. You can view data for a different task by

right-clicking in the view, selecting **Show Data for Other Rank** from the pop-up menu, and selecting the task or rank you need. All of the selected files from the same tool will be displayed in table views and metric browser views as requested.

The **Performance Data → Tree View** window displays a tree view of the data collected. You can expand nodes in the tree by clicking the **+** icons next to the nodes or by selecting the **Expand the Tree** option from the pop-up menu that appears when you right-click in this window. You can collapse nodes by clicking the '**-**' icon next to the nodes or by selecting **Collapse the Tree** from the same pop-up menu.



Label	User time	Execution time	PM_RUN_CYC	PM_RUN_INST_CMPL
main.c				
setup_res		0.000	61395	12939
calc.c				
calcuvzh		0.766	2316222602	2197502630
time.c				
timetend		0.725	2190940221	2150803332
copy.c				
acopy_two_to_one		0.007	30743453	21780760
acopy_one_to_two		0.008	36526239	25052000
tstep.c				
tstep		1.826	5518311317	2555325140
rusage	4.737			
worker.c				
time_load		0.121	115161860	78955154
neighbour_receive		1.282	3722566083	6197522019
neighbour_send		0.093	138242422	83022995
time_unload		0.383	1090978729	1816559624
calc_load		0.150	267141262	351323790
calc_unload		0.852	2447757043	4076177008
worker		4.972	14238681653	13559142239
send_updated_ds		0.007	16101199	24703841

Figure 23. Performance Data Summary view

Each leaf node in the tree shows some of the data that was obtained for each instrumented location in the application that was actually executed. You can sort the tree by any column. To do this, click the column header of any column.

If you right-click in the window, a pop-up menu with several options appears, including the options to expand and collapse the tree. If you select the **Set Filter** option from this menu, a pop-up dialog appears where you can set filtering criteria to reduce the data shown in this window.

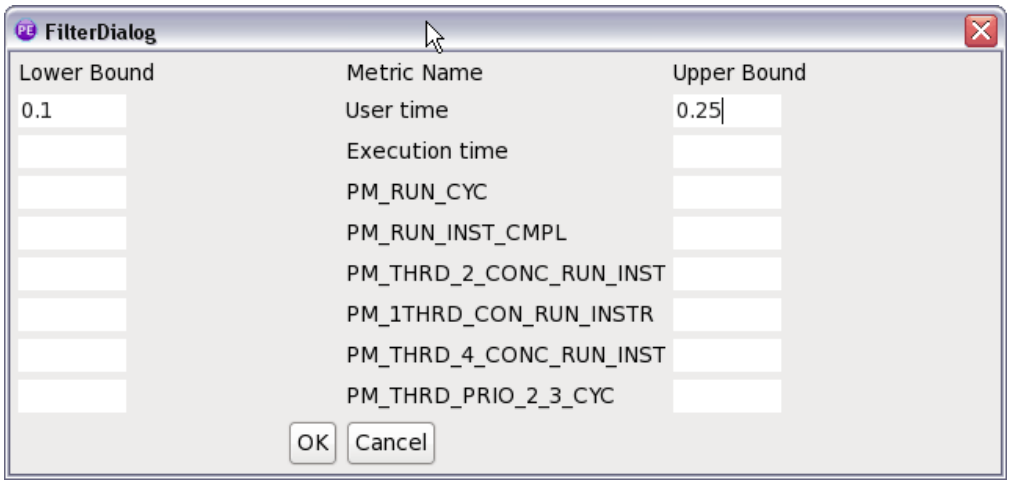


Figure 24. Performance data - FilterDialog

This dialog has fields for upper and lower bounds for each data column in the tree. After entering upper and lower bounds for a field, click **OK**. Only rows with a data value within those bounds will be shown. You can also enter only a lower bound or an upper bound. You can enter filter criteria for one or more columns. You can remove the filter by either opening the filter dialog, clearing all the fields and clicking **OK** or by selecting **No Filter** from the pop-up menu in the data visualization window.

If you left-click on a node in the tree where data is shown and the source file containing the code where the instrumentation point is located is opened, the plug-in highlights the corresponding code region in the source code window. If you double click on an instrumentation point and the source file containing the source code where the instrumentation point is located is not already opened, then the plug-in will attempt to open the source file. If you right-click on a node in the tree in which data is shown and select **Show Metric Browser**, a metric browser window opens, and displays a table containing additional data for all application tasks that executed code at that instrumentation point. You can have more than one of these tables open at the same time.

Figure 25 shows the **Metric Browser** view:

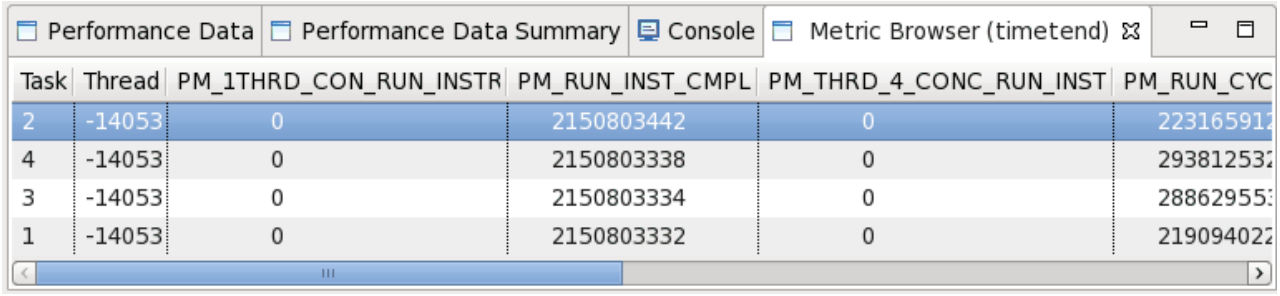


Figure 25. Metric Browser view

To sort data for any column in this table, click on the column header.

You can select the columns that you want to display in this table by right clicking in the table and clicking the **Show Other Metrics** menu and then adding or deleting metrics using the filter dialog shown in Figure 26 on page 46:

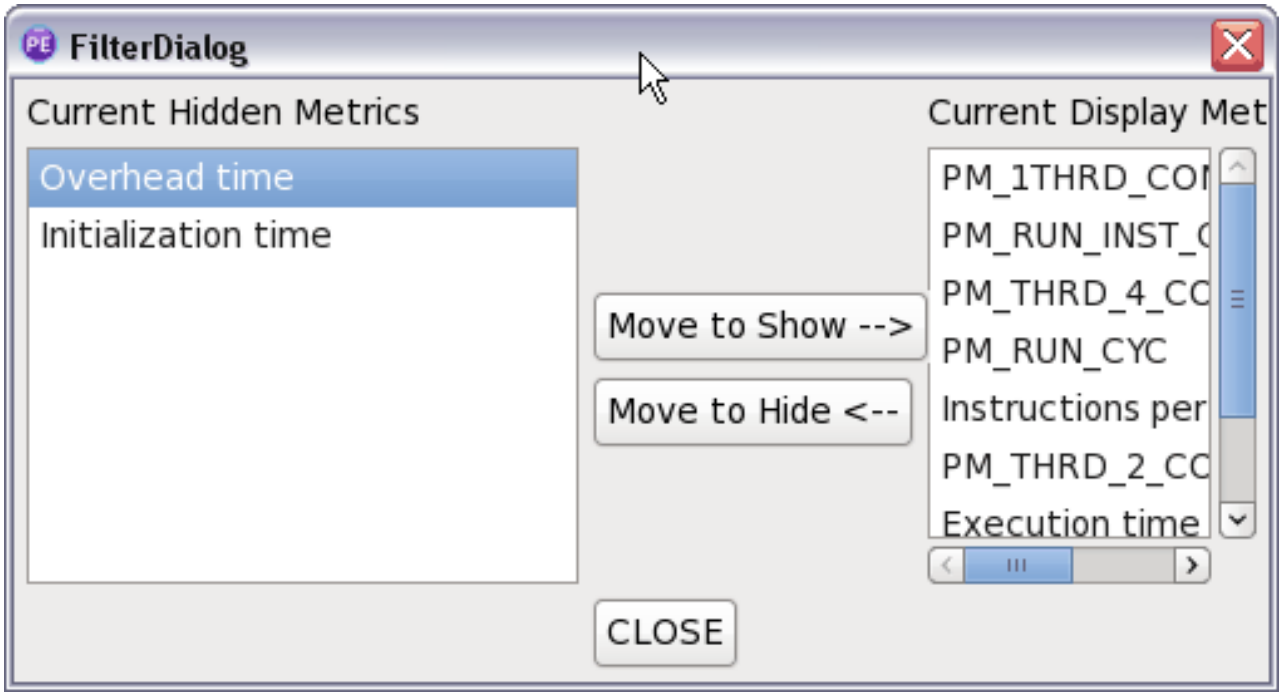
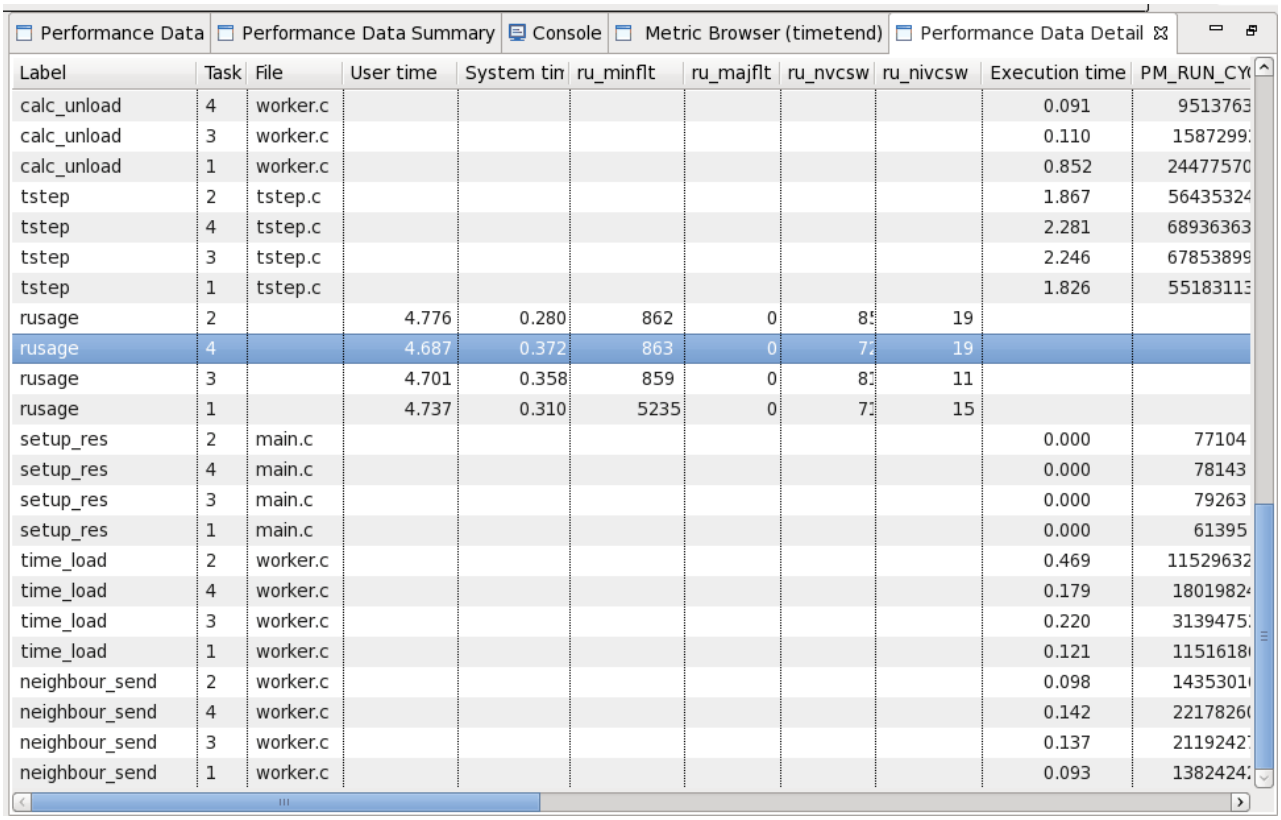


Figure 26. Metrics FilterDialog

Click one or more metrics from the left pane and click the **Move to Show** → button to add metrics. Click one or more metrics from the right pane and click the **Move to Hide** ← button to remove metrics.

You can display the data in the performance data tree view in tabular form by clicking the **Show Data as a Flat Table** option in the pop-up menu in that view. Selecting that option results in a new view similar to Figure 27 on page 47:



Label	Task	File	User time	System tin	ru_minflt	ru_majflt	ru_nvcsw	ru_nivcsw	Execution time	PM_RUN_CY(
calc_unload	4	worker.c							0.091	9513763
calc_unload	3	worker.c							0.110	1587299
calc_unload	1	worker.c							0.852	24477570
tstep	2	tstep.c							1.867	56435324
tstep	4	tstep.c							2.281	68936363
tstep	3	tstep.c							2.246	67853899
tstep	1	tstep.c							1.826	55183113
rusage	2		4.776	0.280	862	0	85	19		
rusage	4		4.687	0.372	863	0	72	19		
rusage	3		4.701	0.358	859	0	83	11		
rusage	1		4.737	0.310	5235	0	71	15		
setup_res	2	main.c							0.000	77104
setup_res	4	main.c							0.000	78143
setup_res	3	main.c							0.000	79263
setup_res	1	main.c							0.000	61395
time_load	2	worker.c							0.469	11529632
time_load	4	worker.c							0.179	1801982
time_load	3	worker.c							0.220	3139475
time_load	1	worker.c							0.121	1151618
neighbour_send	2	worker.c							0.098	1435301
neighbour_send	4	worker.c							0.142	2217826
neighbour_send	3	worker.c							0.137	2119242
neighbour_send	1	worker.c							0.093	1382424

Figure 27. Performance Data Detail view

To sort data for any column in this table, click on the column header.

Viewing MPI traces

You can view a trace of MPI activity in your application if you instrument your application with MPI instrumentation.

After you instrument your application and run it, you can view the trace by selecting the **Load MPI Trace** option from the pop-up menu that is available by right-clicking in the **Performance Data Summary** view. When you do this, a file selector dialog opens and you select the name of the file that the MPI trace will be copied to on your local system. After you specify the trace path name, the plug-in downloads the MPI trace and shows it in the **MPI Trace** view. The Eclipse window looks like Figure 28 on page 48.

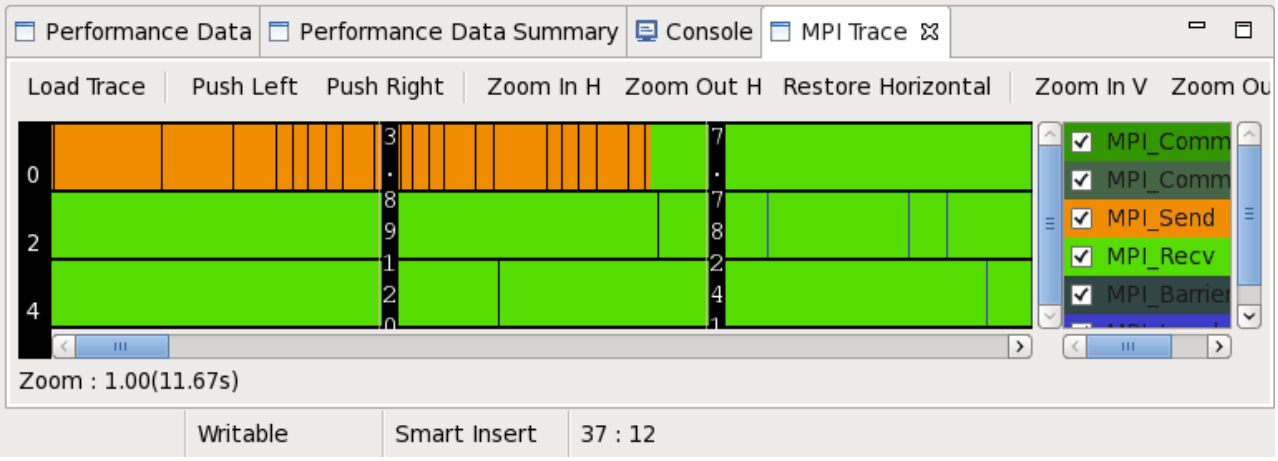


Figure 28. MPI trace view

You can obtain information about each MPI call, filter the trace display to show only interesting MPI calls, and zoom in and out of the trace display. The details of how to use the trace display are described in Chapter 5, "Using MPI profiling in the plug-in," on page 59.

Viewing I/O profiling data

You can view a plot of I/O activity in your application if you instrument it with MIO instrumentation.

After you instrument your application and run it, you can view the plot by selecting the **Load IO Trace** option from the pop-up menu available by right-clicking in the **Performance Data Summary** view. When you do this, Figure 29 is displayed. Folders in the tree are expanded and collapsed by clicking the **+** and **-** icons to the left of the folders.

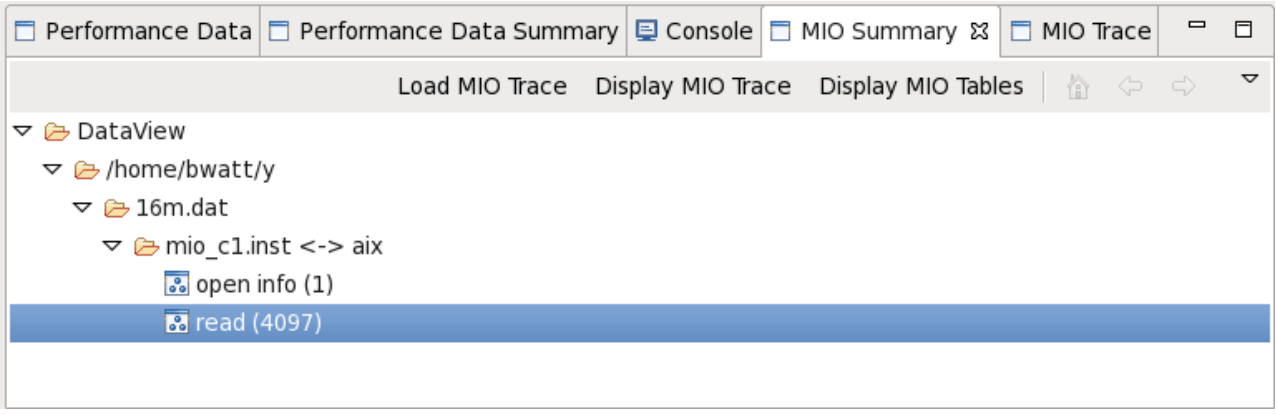


Figure 29. MIO summary view

Select the metrics you are interested in from the tree in the **MIO Summary** view, then press the **Display MIO Trace** button in the view's toolbar to view the trace. The Eclipse window will look like Figure 30 on page 49:

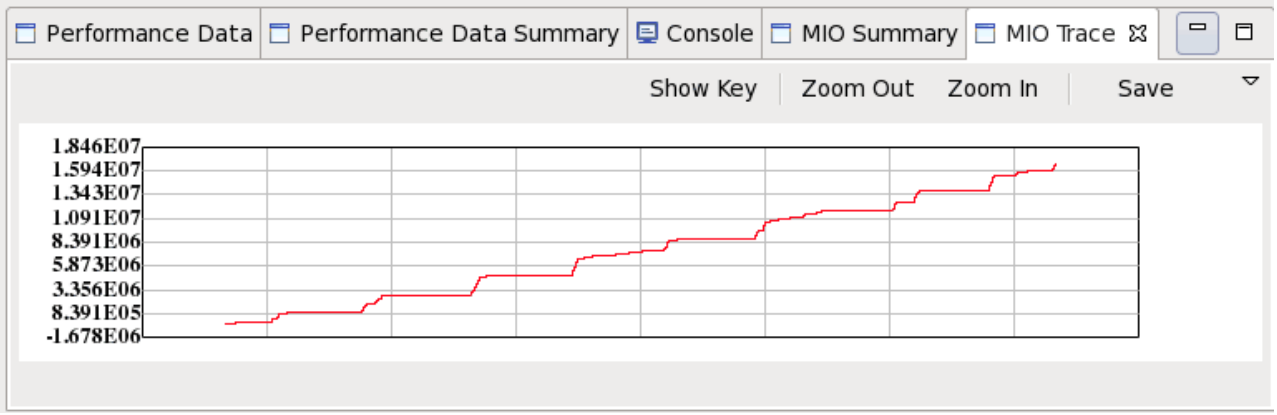


Figure 30. MIO trace view

You can view the I/O activity plot and measure I/O performance at any point in the plot, The details of how to use this plot are described in Chapter 7, "Using I/O profiling in the plug-in," on page 73.

Chapter 4. Using hardware performance counters in the plug-in

The following topics provide information about using hardware performance counters in the Eclipse plug-in.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

Your application should not contain any calls to functions listed in Table 53 on page 261 under "Hardware performance monitoring", because those calls might interfere with correct operation of the instrumentation you insert using the plug-in. Your application should not be linked with the hardware performance counter library, (**libhpc.a** for AIX or **libhpc.so** for Linux). Your application must be compiled and linked with the **-g** flag. When you compile an application on a Power Linux system, you need to use the **-Wl,--hash-style=sysv -WI,--emit-stub-syms** compiler flags.

Note: On x86 systems, the instrumentation process described here is not supported and you must insert appropriate instrumentation calls into your application, then rebuild your application, linking with the **libhpc** hardware performance counter library.

You start the Workbench and then follow the steps described in "Instrumenting your application at launch" or "Instrumenting your application before launch" on page 53 to set up, instrument, and run your application.

Instrumenting your application at launch

You can start and use the IBM HPC Toolkit Eclipse plug-in as described in "Instrumenting your application at launch" on page 22.

Running your application

Open the **Profile Configuration** dialog and fill in the **Resources**, **Application**, **Arguments**, and **Environment** tabs as described in "Running the application" on page 22.

When you select the **Performance Analysis** tab and select the IBM HPC Toolkit tool, the application is analyzed and the rest of this tab is filled in with **Common**, **HPM**, **MPI**, **OpenMP**, and **MIO** tabs. For hardware performance counters you need only fill in the **Common** and **HPM** tabs, that is, you can ignore the **MPI**, **OpenMP** and **MIO** tabs. To fill in the **Common** tab, see Figure 8 on page 29. Next select the **HPM** tab.

Figure 31 on page 52 shows the **Profile Configuration** dialog with the instrumentation points tree fully expanded.

© Copyright IBM Corp. 2008, 2013

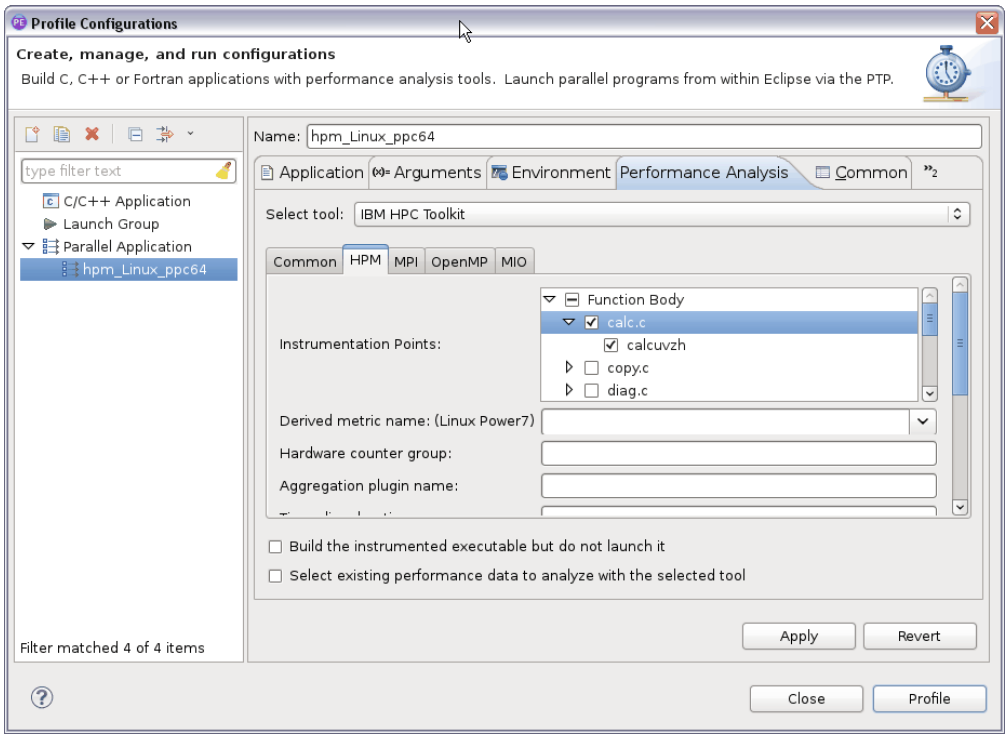


Figure 31. Profile Configurations dialog with the instrumentation points tree fully expanded

There are three classes of instrumentation shown in this tree:

1. Function entry and exit points, labeled **Function Body**. The leaf nodes in this section of the tree are labeled with the name of the function.
2. Function call sites, or locations, labeled **Function Call** (not shown). The leaf nodes in this section of the tree are labeled with the name of the function being called and with the line number of the function call appended.
3. A user-defined region, labeled **HPM Region** (not shown), which is a user-specified region of code, such as a loop, that is instrumented.

You can select any region of code to be instrumented, but be careful to ensure that the code at both the beginning and ending of each region is executed in each execution of that region. Leaf nodes in this section of the table are labeled with the source file name with the starting and ending line numbers for the region appended.

If you instrument a function call site, the instrumentation obtains hardware performance counter measurements for that instance of the function being called and reports them independently of any other function call to that same function. If you instrument function entry and exit points, the instrumentation obtains hardware performance counter measurements for that function for every time that function is called, regardless of the caller. If you instrument a user-defined region, hardware performance measurements are obtained specifically for that region.

User-defined regions can only be created manually while editing your source. If you want to instrument user-defined regions, see "Instrumenting your application before launch" on page 53.

You can select any combination of function call sites, function entry and exit points and user-defined regions that you want to instrument. You can select instrumentation by individually selecting leaf nodes, or you can select a group of

leaf nodes by selecting a higher-level node in the tree. You can also deselect instrumentation by clicking again on a checked node.

Fill in any other fields in the **HPM** tab with run-time options. For more information about settings on the HPM tab, use Table 8 on page 56 and Table 9 on page 57.

Run your application by pressing the **Profile** button. Your application runs using the specified setting. See "Viewing hardware performance counter data" on page 58.

Instrumenting your application before launch

You can start and use Eclipse and load the application binary as described in "Instrumenting your application before launch" on page 31.

Instrumenting your application

The first step in instrumenting your application is to open the executable as described in "Loading the application" on page 32. After the executable has been opened, make sure the **HPM** tab in the **Instrumentation** view is visible by clicking it. Figure 32 on page 54 shows the results in an Eclipse window:

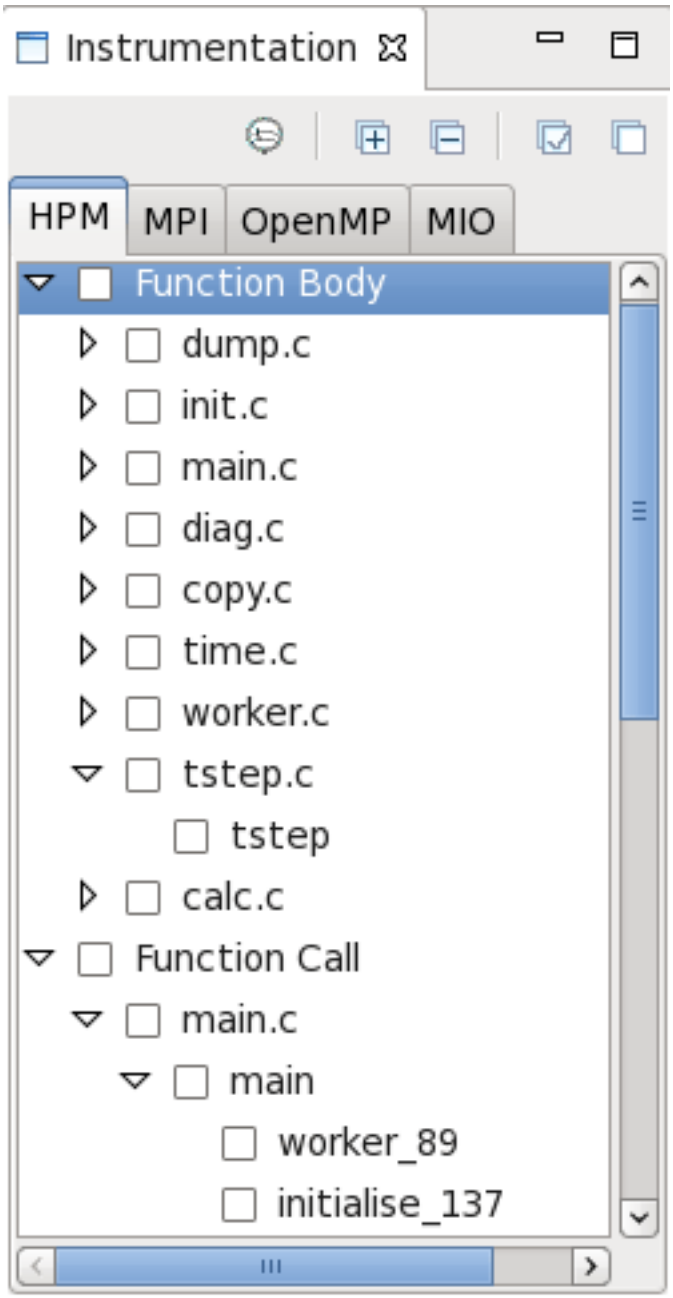


Figure 32. Eclipse plug-in the HPM instrumentation view

Figure 32 shows the instrumentation points tree fully expanded. There are three classes of instrumentation shown in this tree:

1. Function entry and exit points, labeled **Function Body**. The leaf nodes in this section of the tree are labeled with the name of the function.
2. Function call sites, or locations, labeled **Function Call**. The leaf nodes in this section of the tree are labeled with the name of the function being called and with the line number of the function call appended.
3. A user-defined region, labeled **HPM Region** (not shown), which is a user-specified region of code, such as a loop, that is instrumented.

You can select any region of code to be instrumented, but be careful to ensure that the code at both the beginning and ending of each region is executed in each

execution of that region. Leaf nodes in this section of the table are labeled with the source file name with the starting and ending line numbers for the region appended.

If you instrument a function call site, the instrumentation obtains hardware performance counter measurements for that instance of the function being called and reports them independently of any other function call to that same function. If you instrument function entry and exit points, the instrumentation obtains hardware performance counter measurements for that function for every time that function is called, regardless of the caller. If you instrument a user-defined region, hardware performance measurements are obtained specifically for that region.

User-defined regions are specified by highlighting a region of source code and then adding that region to the list of user-defined regions in the data collection window. To add a region to the user-defined region list, select that region by selecting the required source file in the source code window, left-clicking at the starting line of the region, and dragging while the left mouse button is pressed until all required source code lines are highlighted. After the required lines are selected, right-click in the source code window and select the **Add to HPM** option from the pop-up menu. You can have as many user-defined regions as you want, and user-defined regions could overlap.

You can select any combination of function call sites, function entry and exit points and user-defined regions that you want to instrument. You can select instrumentation by individually selecting leaf nodes, or you can select a group of leaf nodes by selecting a higher-level node in the tree. You can also deselect instrumentation by clicking again on a checked node.

After you have selected the set of instrumentation that you want, you instrument the application by right-clicking in the **Instrumentation** view, then clicking **Instrument Executable**, or by clicking the **Instrument** icon in the toolbar in the **Instrumentation** view, or clicking the **Instrument Executable** in the **Instrumentation** view.

Running your application

Before you can run your application, you need to make sure the run-time environment is set up correctly. If you are running your application using the **Profile Configurations** dialog, you should make sure that all options are set correctly in the **Common** and **HPM** tabs of the **Performance Analysis** pane of the **Profile Configurations** dialog, If you are running your application using a command-line shell, make sure you have run the IBM HPC Toolkit setup script and set any additional environment variables required to obtain performance data.

Figure 33 on page 56 shows what the **Common** settings looks like:

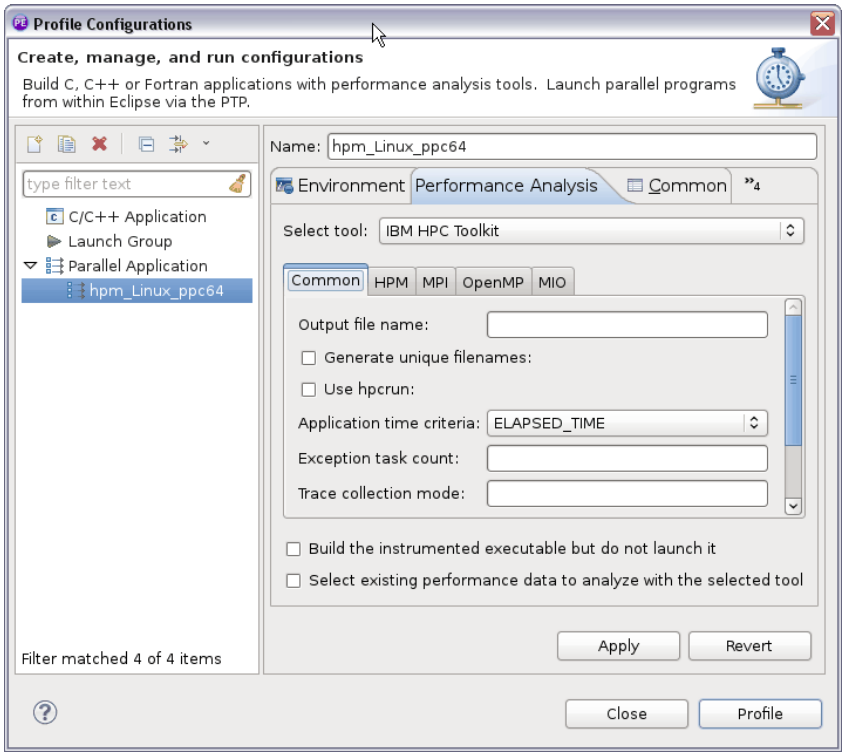


Figure 33. Profile Configurations dialog Common settings

For more information about the settings on this panel, use Table 8 to reference the corresponding environment variable description in "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265.

Table 8. Profile Configurations dialog Common environment variables

Profile configuration field	**Environment variable name**
Output file name	**HPC_OUTPUT_NAME**
Generate unique filenames	**HPC_UNIQUE_FILE_NAME**
Use **hpcrun**	**HPC_USE_HPCRUN**
Application time criteria	**HPC_EXCEPTION_METRIC**
Exception task count	**HPC_EXCEPTION_COUNT**
Trace collection mode	**HPC_TRACE_STORE**

If you are using the **Profile Configurations** dialog, the **HPM** tab of the **Profile Configurations** dialog looks like Figure 34 on page 57:

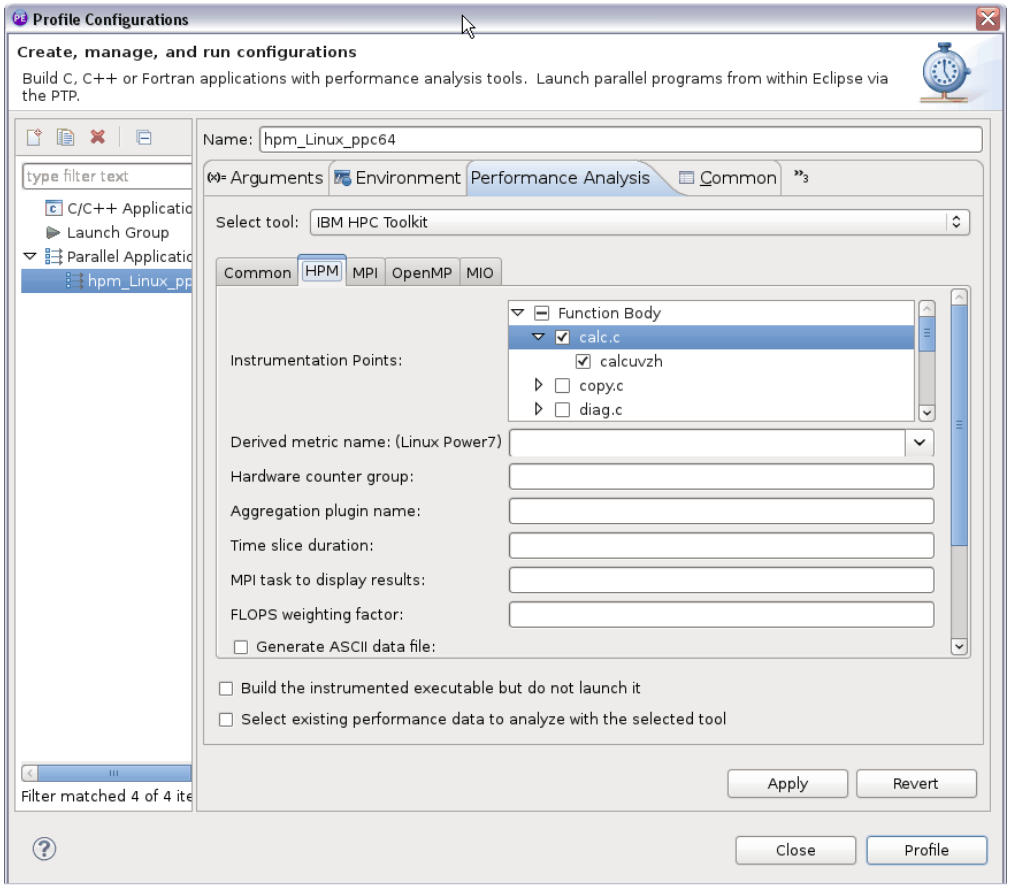


Figure 34. Profile Configurations dialog hardware counter settings

For more information about settings on this panel, use Table 9 to reference the corresponding environment variable description in "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265.

Table 9. HPM profile configuration dialog environment variables

Profile configuration field	Environment variable name
Hardware counter group	**HPM_EVENT_SET**
Aggregation plugin name	**HPM_AGGREGATE**
Time slice duration	**HPM_SLICE_DURATION**
MPI task to display results	**HPM_PRINT_TASK**
FLOPS weighting factor	**HPM_DIV_WEIGHT**
Generate ASCII data file	**HPM_ASC_OUTPUT**
Generate visualization files	**HPM_VIZ_OUTPUT**
Write to stdout	**HPM_STDOUT**
Exclusive values	**HPM_EXCLUSIVE_VALUES**
Print formulas	**HPM_PRINT_FORMULA**

In particular, you should be sure that **Generate visualization files** and **Hardware counter group** or the **HPM_VIZ_OUTPUT** and **HPM_EVENT_SET** environment variables are set correctly.

You can run your application as described in "Running the application" on page 34.

Viewing hardware performance counter data

If you run your application using the **Profile Configurations** dialog, the plug-in prompts you, asking if you want to open your performance data files after the application completes, and it opens them if you requested that they be opened. Otherwise, you must open the performance data files manually.

"Viewing performance data" on page 42 describes how to open performance data files and view different presentations of that data.

Chapter 5. Using MPI profiling in the plug-in

The topics that follow provide information for using MPI profiling in the plug-in.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

Your application should not call any of the functions described in Table 53 on page 261 under "MPI profiling" because those calls might interfere with the instrumentation you insert using the plug-in. Your application should not be linked with the MPI profiling library, **libmpitrace.a** for AIX or **libmpitrace.so** for Linux. Your application must be compiled and linked with the **-g** flag. When you compile an application on a Power Linux system, you need to use the **-Wl,--hash-style=sysv -WI, --emit-stub-syms** compiler flags.

Note: On x86 systems, the instrumentation process described here is not supported and you must link your application with the **libmpi** MPI trace library.

You cannot use the MPI profiling library to create a trace for an application that issues MPI function calls from multiple threads in the application.

You start the Workbench and then follow the steps described in "Instrumenting your application at launch" on page 51 or "Instrumenting your application before launch" on page 53 to set up, instrument, and run your application.

Instrumenting your application at launch

You can start and use the IBM HPC Toolkit Eclipse plug-in as described in "Instrumenting your application at launch" on page 22.

Running your application

Open the **Profile Configurations** dialog and fill in the **Resources**, **Application**, **Arguments**, and **Environment** tabs as described in "Running the application" on page 22.

When you select the **Performance Analysis** tab and select the IBM HPC Toolkit tool, the application is analyzed and the rest of this tab is filled in with **Common**, **HPM**, **MPI**, **OpenMP**, and **MIO** tabs. For MPI profiling, you need only fill in the **Common** and **MPI** tabs, that is, you can ignore the **HPM**, **OpenMP** and **MIO** tabs. To fill in the **Common** tab, see Figure 8 on page 29. Next select the **MPI** tab.

Figure 35 on page 60 shows the **Profile Configuration** dialog with the instrumentation points tree fully expanded.

© Copyright IBM Corp. 2008, 2013

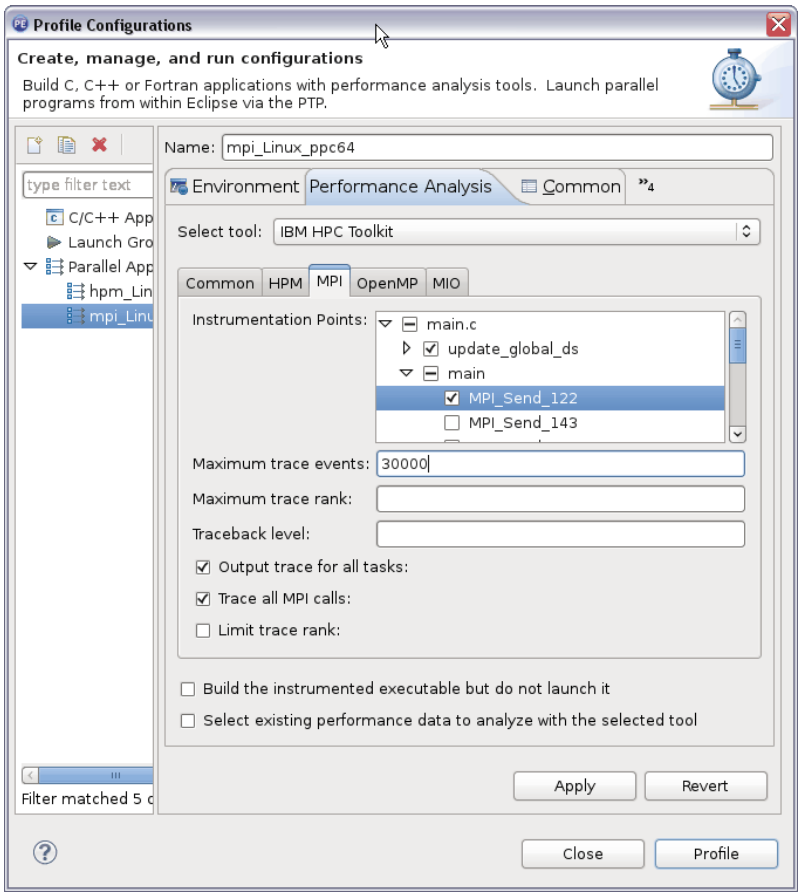


Figure 35. Profile Configurations dialog with the instrumentation points tree fully expanded

Figure 35 shows the **Profile Configurations** dialog with the instrumentation points tree fully expanded. The tree shows all MPI function calls that can be instrumented. For MPI profiling, the plug-in labels each instrumentation location with the name of the MPI function at that line in the application and the line number where the MPI function call is located.

You can select as many instrumentation locations in your application as you like. If you select a leaf node, only that MPI function call is instrumented. If you select a node labeled with an application function name or file name, all MPI function calls within the file or function named by the enclosing node are instrumented. You can deselect an MPI function call by clicking the corresponding highlighted node in the instrumentation points tree.

Fill in any other fields in the **MPI** tab with run-time options. For more information about settings on the MPI tab, use Table 10 on page 63.

Run your application by pressing the **Profile** button. Your application runs using the specified setting. For more information, see "Viewing MPI profiling data" on page 64.

Instrumenting your application before launch

You can start and use Eclipse and load the application binary as described in "Instrumenting your application before launch" on page 31.

Instrumenting your application

The first step in instrumenting your application is to open the executable as described in "Loading the application" on page 32. After the executable has been opened, make sure the **MPI** tab in the **Instrumentation** view is visible by clicking it. Figure 36 shows the instrumentation view:

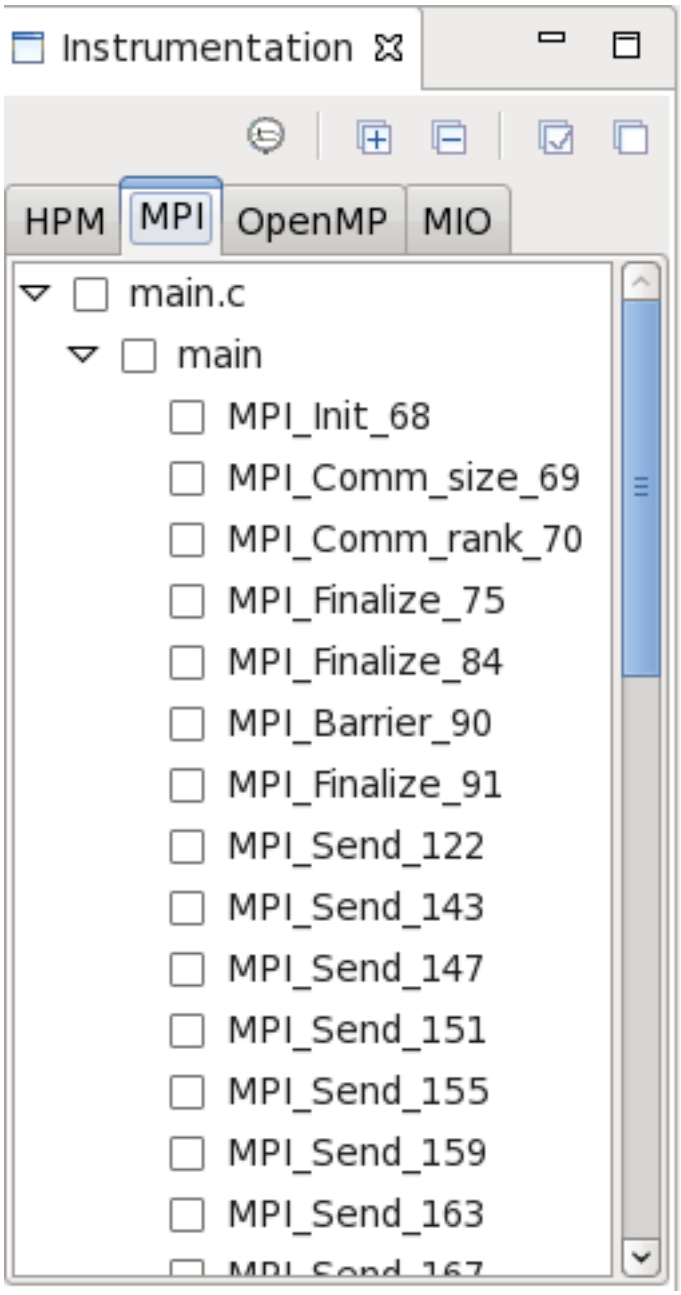


Figure 36. Eclipse plug-in MPI instrumentation view

The **MPI** view shows the instrumentation points tree fully expanded showing all MPI function calls that can be instrumented. For MPI profiling, the plug-in labels each instrumentation location with the name of the MPI function at that line in the application and the line number where the MPI function call is located.

You can select as many instrumentation locations in your application as you like. If you select a leaf node, only that MPI function call is instrumented. If you select a node labeled with an application function name or file name, all MPI function calls within the file or function named by the enclosing node are instrumented. You can deselect an MPI function call by clicking the corresponding highlighted node in the instrumentation points tree.

You can also select MPI function calls to be instrumented by left-clicking over a starting line in the source code window and dragging to the end of the region of interest while holding the left mouse button down. After you have selected a region of code, you can select all MPI function calls in that region to be instrumented or not instrumented by right-clicking in the source code window and selecting either **Select MPI** or **Deselect MPI** respectively. When you do this, the plug-in updates the tree in the **Instrumentation** view accordingly.

After you have selected the set of instrumentation that you want, you instrument the application by right-clicking in the **Instrumentation** view, then clicking **Instrument Executable**, or by clicking the **Instrument** icon in the toolbar in the **Instrumentation** view.

Running your application

Before you can run your application, you need to make sure the run-time environment is set up correctly. If you are running your application using the **Profile Configurations** dialog, make sure all options are set correctly in the **MPI** tab of the **HPC Toolkit** pane of the **Profile Configurations** dialog. If you are running your application using a command-line shell, make sure you have run the IBM HPC Toolkit setup script and set any additional environment variables required to obtain performance data.

If you are using the Profile Configurations dialog, the **MPI** tab of the **Profile Configurations** dialog looks like Figure 37 on page 63:

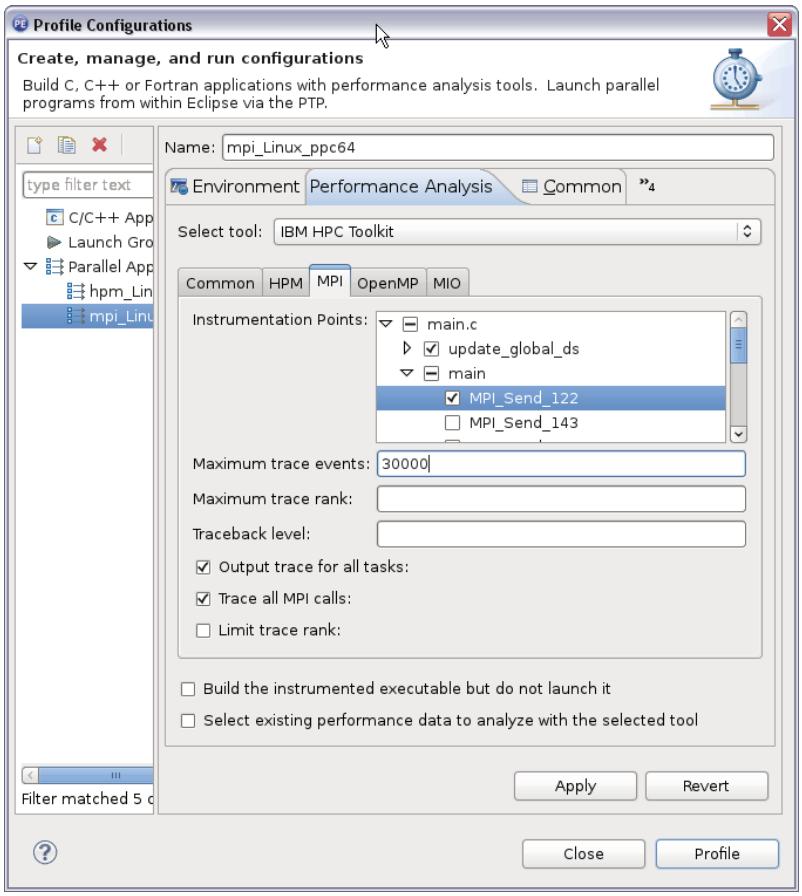


Figure 37. Profile Configurations dialog MPI settings panel

For more information about the settings on this panel, use Table 10 to reference the corresponding environment variable description in "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303.

Table 10. MPI trace profile configuration environment variables

Profile Configurations field name	Environment variable
Maximum trace events	**MAX_TRACE_EVENTS**
Maximum trace rank	**MAX_TRACE_RANK**
Traceback level	**TRACEBACK_LEVEL**
Output trace for all tasks	**OUTPUT_ALL_RANKS**
Trace all MPI calls	**TRACE_ALL_EVENTS**
Limit trace rank	**TRACE_ALL_TASKS**

In particular, you should consider the options that control which tasks are traced. By default, the MPI profiling library generates trace files only for the application tasks with the minimum, maximum, and median MPI communication time. This is also true for task zero, if task zero is not the task with minimum, maximum, or median MPI communication time. If you need trace files generated for additional tasks, make sure **Output trace for all tasks** or the **OUTPUT_ALL_RANKS** environment variable is set correctly. Depending on the number of tasks in your application, you should make sure the **Maximum trace rank** and **Limit trace rank** or **MAX_TRACE_RANK** and **TRACE_ALL_TASKS** environment variables are set

correctly. If your application executes many MPI function calls, you might need to set the value of **Maximum trace events** or the **MAX_TRACE_EVENTS** environment variable to a higher number than the default 30,000 MPI function calls.

You can run your application as described in "Running the application" on page 34.

Viewing MPI profiling data

The plug-in tries to open the performance data files after your application has completed execution, if you requested it to do so when you ran your application. If the plug-in does not automatically open the performance data files, you can open them manually by selecting the performance data (***.viz**) files in the **Project Explorer** view, right-clicking over the selected set of files, clicking the **HPCT** option in the pop-up menu and selecting **Open Performance Files** in the second pop-up menu. Figure 38 shows the instrumentation points tree in the **Performance Data Summary** view fully expanded with MPI profiling performance data.

Performance Data | Performance Data Summary | Console

Data for rank 2, Aggregation for tasks: 2 4

Label	Count	WallClock	Transfered Bytes
main.c			
main(main.c)			
MPI_Comm_size_69	1	0.000045	0.000000
MPI_Comm_rank_70	1	0.000004	0.000000
MPI_Barrier_90	1	0.001386	0.000000
SUMMARY			
MPI_Recv	11294	5.817011	11563040.000000
MPI_Send	320	0.000371	328960.000000
MPI_Isend	9500	0.019715	9728000.000000
MPI_Comm_rank	2	0.000012	0.000000
MPI_Barrier	1	0.001386	0.000000
MPI_Comm_size	2	0.000050	0.000000
worker.c			
neighbour_send(worker.			
MPI_Isend_310	9500	0.019715	9728000.000000
worker(worker.c)			
MPI_Comm_rank_93	1	0.000008	0.000000
MPI_Recv_95	1	0.000024	16.000000
MPI_Comm_size_92	1	0.000005	0.000000
MPI_Recv_120	256	0.002154	262144.000000
MPI_Recv_124	256	0.002062	262144.000000
MPI_Recv_128	256	0.002086	262144.000000

Figure 38. Performance data summary view with MPI profiling data

This view shows the number of times each function call was executed, the total time spent executing that function call, and the amount of data transferred by that function call. You can see a more detailed view of the data you obtained by right-clicking over a leaf node in the view and selecting **Show Metric Browser**, which opens a metrics browser window, or by right-clicking in white space in the

view and selecting **Show Data as a Flat Table** from the pop-up menu, which opens a table view of the performance data.

The data visualization window shows performance data from a single task at one time. If you want to see performance data from a different task, right-click in the white space in the window and select **Show Data for Other Rank** from the pop-up menu. A pop-up dialog appears containing a drop-down list that allows you to select the task indices of other tasks that can be selected.

You can view a trace of MPI activity in your application by right-clicking in the view and selecting **Load MPI Trace** from the pop-up menu. When you do this, a file selector dialog is displayed, prompting you to specify a file name where the trace is downloaded. After you click **Save** in the file selector dialog, the trace is downloaded and the **MPI Trace** view opens.

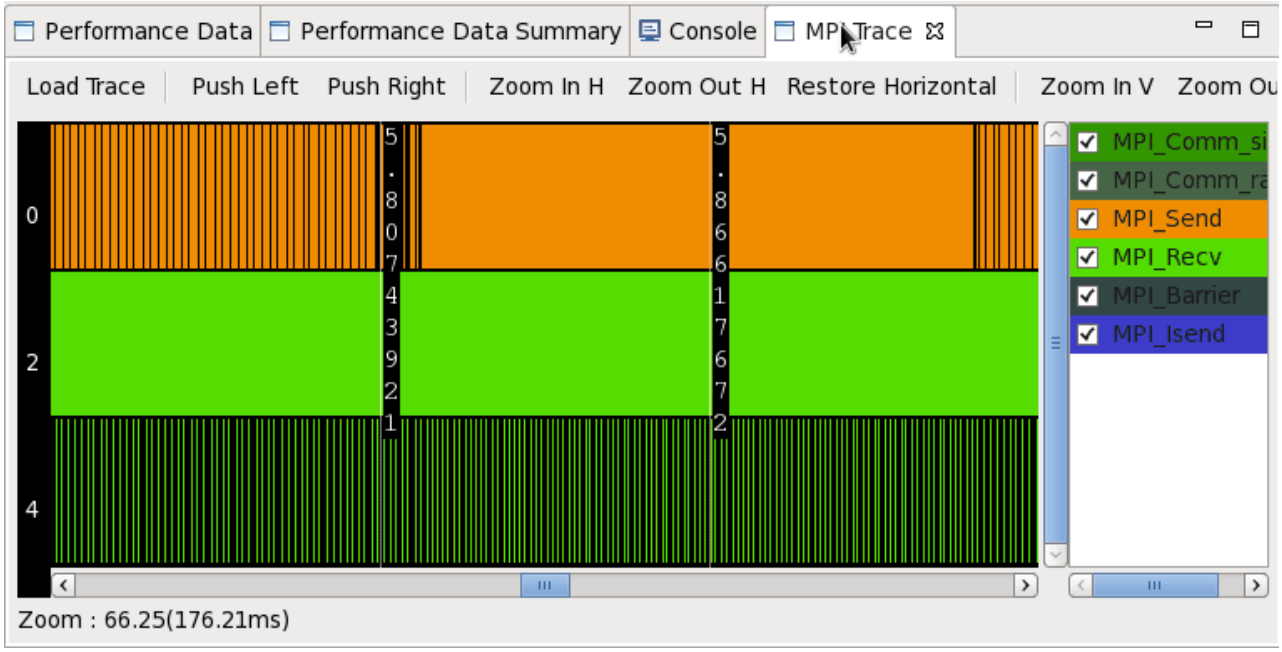


Figure 39. MPI trace view

The pane on the right-hand side of the **MPI Trace View** is used to select which MPI functions are displayed in the trace on the left side of the view. Initially, all MPI function calls that were executed are displayed. You can hide or display trace events for specific MPI functions by clicking on the MPI function labels in the list. Trace events are displayed for MPI functions that have their labels checked while trace events are hidden for MPI functions with unchecked labels.

The left-hand pane is the MPI trace viewer. This pane displays a timeline-based view of your application's execution, where the Y axis is the application task rank and the X axis is elapsed time. Each MPI function call is represented by a block drawn in the color matching the MPI function label in the right-hand pane. The numbers in the middle of the trace are timestamps identifying those points in the trace timeline.

You can navigate through the trace using the buttons at the top of the view as described in Table 11 on page 66:

Table 11. MPI trace timeline navigation

Button	Action
Load Trace	Loads a MPI trace file
Push Left	Scrolls the trace left (decreasing time)
Push Right	Scrolls the trace right (increasing time)
Zoom In H	Zoom in on the X (time) axis
Zoom Out H	Zoom out on the X (time) axis
Restore Horizontal	Reset the X axis to its initial state
Zoom In V	Zoom in on the Y (task) axis
Zoom Out V	Zoom out on the Y (task) axis
Restore Vertical	Reset the Y axis to its initial state

You can also scroll through the trace by using the scroll bars at the bottom and right of the pane. You can also zoom in on the X axis by left-clicking at a point in the timeline, then dragging the mouse left or right to the desired ending point in the timeline then releasing the left mouse button.

If you left-double-click over a MPI function's trace entry, the plug-in will attempt to display the source file where the MPI function call was made and position the source file view to the line where the MPI function call was made. If you right click and hold the right mouse button over a MPI function's trace entry, the plug-in displays a pop-up with additional information about the function call.

Chapter 6. Using OpenMP profiling in the plug-in

The topics that follow will provide information about using OpenMP profiling in the plug-in.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

Your application must be compiled and linked with the **-g** compiler flag. When you compile an application on a Power Linux system, you need to use the **-Wl,--hash-style=sysv -WI,--emit-stub-syms** compiler flags. When you compile an OpenMP application you must also use the **-qsmp=omp** flag.

You start the Workbench and then follow the steps described in "Instrumenting your application at launch" on page 51 or "Instrumenting your application before launch" on page 53 to set up, instrument, and run your application.

Instrumenting your application at launch

You can start and use the IBM HPC Toolkit Eclipse plug-in as described in "Instrumenting your application at launch" on page 22.

Running your application

Open the **Profile Configuration** dialog and fill in the **Resources**, **Application**, **Arguments**, and **Environment** tabs as described in "Running the application" on page 22.

When you select the **Performance Analysis** tab and select the IBM HPC Toolkit tool, the application is analyzed and the rest of this tab is filled in with **Common**, **HPM**, **MPI**, **OpenMP**, and **MIO** tabs. For OpenMP profiling, you need only fill in the **Common** and **OpenMP** tabs, that is, you can ignore the **HPM**, **MPI** and **MIO** tabs. To fill in the **Common** tab, see Figure 8 on page 29. Next select the **OpenMP** tab.

Figure 40 on page 68 shows the **Profile Configurations** dialog with the instrumentation points tree fully expanded.

© Copyright IBM Corp. 2008, 2013

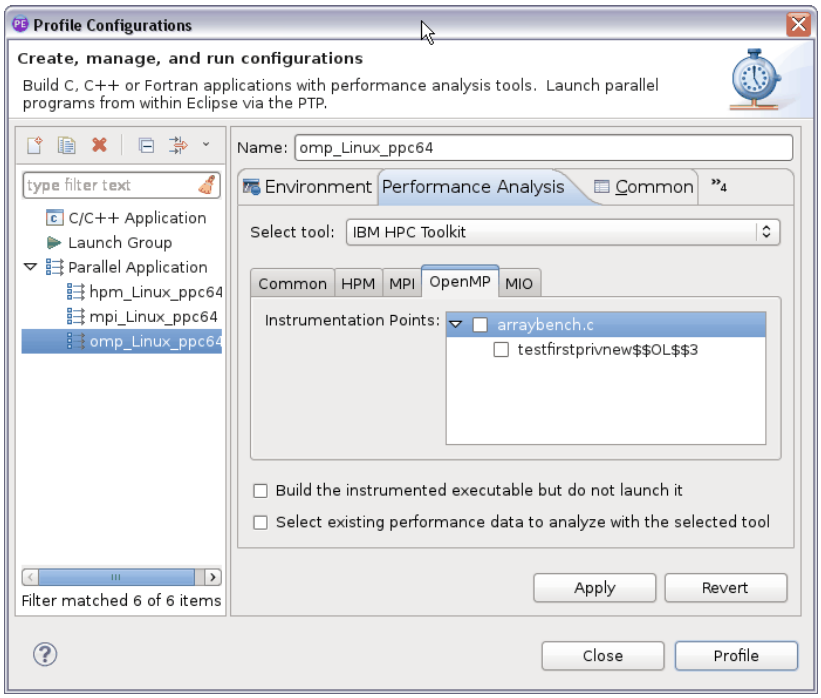


Figure 40. Profile Configurations dialog with the instrumentation points tree fully expanded

The tree shows all **OpenMP** function calls that can be instrumented. Leaf nodes in this tree represent individual OpenMP regions in your application. Leaf nodes are labeled with the function name generated by the compiler for that OpenMP region. You can select instrumentation at any level in the tree. If you select a leaf node, only that location is instrumented. If you select a non-leaf node, all leaf nodes that are children of the selected node are instrumented.

Note: The IBM HPC Toolkit only supports OpenMP profiling instrumentation for OpenMP regions that are not nested within other OpenMP constructs at run time. If you set up instrumentation so that nested parallel constructs are instrumented, results are unpredictable.

Run your application by pressing the **Profile** button. Your application runs using the specified setting. For more information, see "Viewing OpenMP performance data" on page 69.

Instrumenting your application before launch

You can start and use Eclipse and load the application binary as described in "Instrumenting your application before launch" on page 31.

Instrumenting your application

The first step in instrumenting your application is to open the executable as described in "Loading the application" on page 32. After the executable has been opened, make sure the **OpenMP** tab in the **Instrumentation** view is visible by clicking it. This results in an Eclipse window that looks like Figure 41 on page 69:

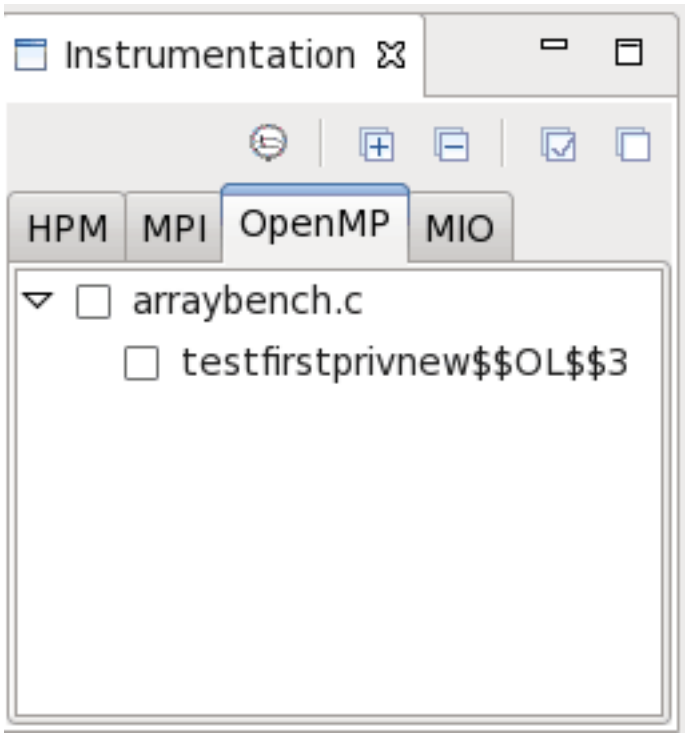


Figure 41. OpenMP instrumentation view

This view shows you the locations in your application that can be instrumented for OpenMP profiling. Leaf nodes in this tree represent individual OpenMP regions in your application. Leaf nodes are labeled with the function name generated by the compiler for that OpenMP region. You can select instrumentation at any level in the tree. If you select a leaf node, only that location is instrumented. If you select a non-leaf node, all leaf nodes that are children of the selected node are instrumented.

Note: The IBM HPC Toolkit only supports OpenMP profiling instrumentation for OpenMP regions that are not nested within other OpenMP constructs at run time. If you set up instrumentation so that nested parallel constructs are instrumented, results are unpredictable.

After you have selected the set of instrumentation that you want, you instrument the application by right-clicking in the **Instrumentation** view, then clicking **Instrument Executable**, or by clicking the **Instrument** icon in the toolbar in the **Instrumentation** view.

Running your application

No additional setup is required before running your application. You can run your application as described in "Running the application" on page 34.

Viewing OpenMP performance data

The plug-in attempts to open the performance data files after your application has completed execution, if you requested it to do so when you ran your application. If the plug-in does not automatically open the performance data files, you can open them manually by selecting the performance data (***.viz**) files in the **Project**

Explorer view, right clicking over the selected set of files, clicking the **HPCT** option in the pop-up menu and selecting **Open Performance Files** in the second pop-up menu. Figure 42 shows the instrumentation points tree in the **Performance Data Summary** view fully expanded with OpenMP performance data.

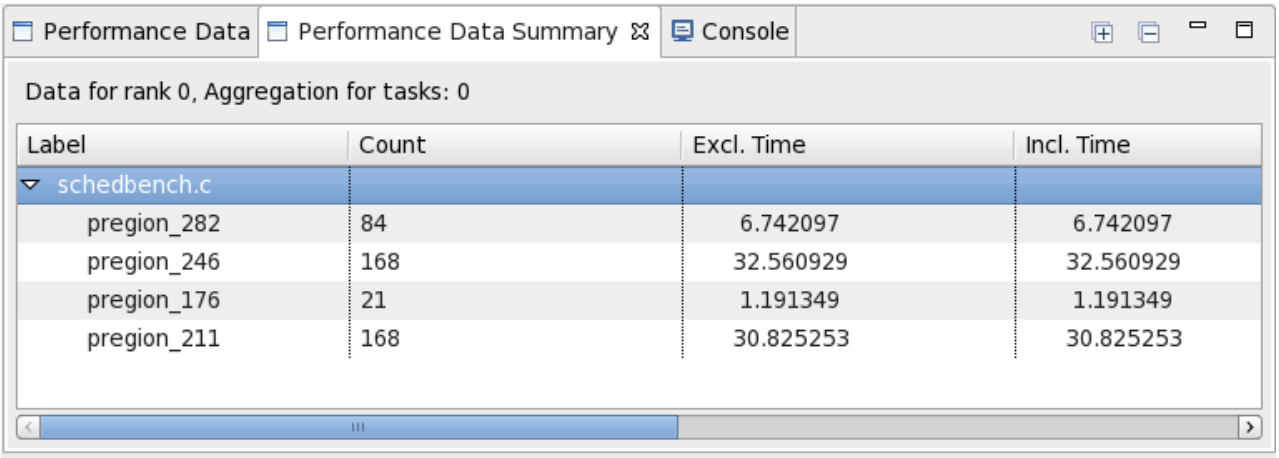


Figure 42. Performance data summary view with OpenMP profiling data

The data is displayed in a tree format in which source files are the major nodes in the tree and the individual instrumentation points are the leaf nodes. Individual leaf nodes are labeled with the type of OpenMP instrumentation and the starting line number of the instrumented OpenMP construct.

You can view detailed data for a leaf node by right-clicking over a leaf node and selecting **Show Metric Browser** from the pop-up menu. A metric browser view shows data for each process and thread that executed the OpenMP construct. You can view all of your performance measurements in a tabular form by selecting the **Show Data as a Flat Table** option from the pop-up menu that appears when you right-click within the **Performance Data Summary** view.

The data displayed in the **Performance Data Summary** view is a summary of the performance data, displaying the maximum value from all threads for the following statistics:

Count
The number of times the event was executed.

Exclusive time (Excl. Time)
The total time (Does not include time inside of other events).

Inclusive time (Incl. Time)
Wall clock time for the event (including other events).

Percentage of total overhead (% Overhead)
100 * (thread time – computation time) / thread time

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))

Average thread time
(Avg. Comp Time): SUM(comp time(i)) / Number of threads

The metrics browser view shows the following statistics for each OpenMP thread:

Task (Task)
the MPI Task ID.

Thread (thread)
The OpenMP thread ID.

Time in master(Time in Master):
The time in the master thread (Wall Clock Time).

Thread time(TT: Thread Time):
The wall clock time for the execution of each thread.

Computation time: (CT: Comp. Time)
The time per thread in the body of the OpenMP construct.

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))

Total overhead: (TO: TT – CT):
The thread time – comp time.

Percentage of the total overhead due to barrier (%TO (Barrier))
100 * barrier time / incl time. The barrier time is measured as the time between the end of the thread execution and end of the last thread.

Percentage of the total overhead due to the runtime library (%TO (RTL))
100 * (RTL time – barrier time) / incl time

The **Performance Data Detail** view shows the following statistics for each process and thread in the application:

Count:
The number of times the event was executed.

Exclusive time (Excl. Time)
The total time (Does not include time inside of other events).

Inclusive time (Incl. Time)
Wall clock time for the event (including other events).

Percentage of total overhead (% Overhead)
100 * (thread time – computation time) / thread time.

Thread time: (TT: Thread Time)
The wall clock time for the execution of each thread.

Computation time: (CT: Comp. Time)
The time per thread in the body of the OpenMP construct.

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i))

Average thread time (Avg. Comp Time)
SUM(comp time(i)) / Number of threads

Total overhead: (TO: TT – CT)
The thread time – comp time.

Percentage of the total overhead due to barrier (%TO (Barrier))
100 * barrier time / incl time. The barrier time is measured as the time between the end of the thread execution and end of the last thread.

Percentage of the total overhead due to the runtime library (%TO (RTL))
100 * (RTL time – barrier time) / incl time

Chapter 7. Using I/O profiling in the plug-in

The topics that follow provide information about using I/O profiling in the plug-in.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

Your application must be compiled and linked with the **-g** compiler flag. When you compile an application on a Power Linux system, you will need to use the **-Wl,--hash-style=sysv -WI,--emit-stub-syms** compiler flags.

Note: On x86 systems, the instrumentation process described here is not supported. You must link your application with the I/O profiling and trace library (**libhpctkio**) and set **TKIO_ALTLIB** to point to the correct version of **get_hpcmio_ptrs.so**.

You start the Workbench and then follow the steps described in "Instrumenting your application at launch" on page 51 or "Instrumenting your application before launch" on page 53 to set up, instrument, and run your application.

Instrumenting your application at launch

You can start and use the IBM HPC Toolkit Eclipse plug-in as described in "Instrumenting your application at launch" on page 22.

Running your application

Open the **Profile Configuration** dialog and fill in the **Resources**, **Application**, **Arguments**, and **Environment** tabs as described in "Running the application" on page 22.

When you select the **Performance Analysis** tab and select the IBM HPC Toolkit tool, the application is analyzed and the rest of this tab is filled in with **Common**, **HPM**, **MPI**, **OpenMP**, and **MIO** tabs. For I/O profiling, you need only fill in the **Common** and **MIO** tabs, that is, you can ignore the **HPM**, **MPI** and **OpenMP** tabs. To fill in the **Common** tab, see Figure 8 on page 29. Next select the **MIO** tab.

© Copyright IBM Corp. 2008, 2013

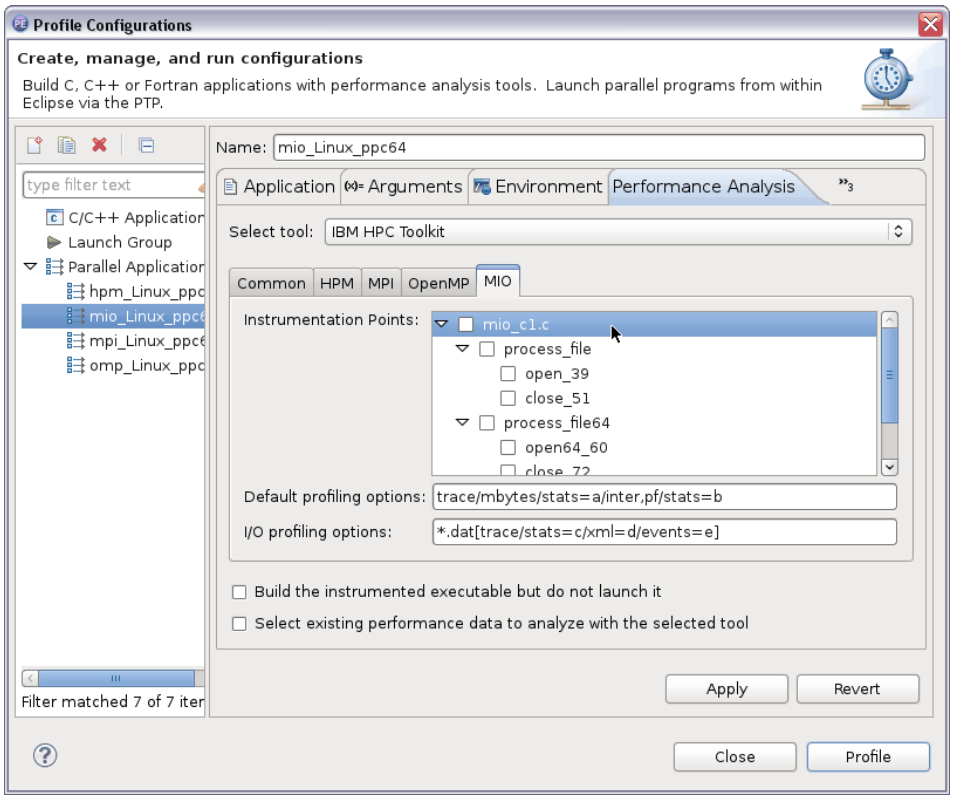


Figure 43. Profile Configurations dialog with the I/O instrumentation points tree fully expanded

Figure 43 shows the **Profile Configurations** dialog with the I/O instrumentation points tree fully expanded. The leaf nodes are labeled with the name of the system call at that location and the line number in the source file. If you select leaf nodes, instrumentation is placed only at those specific instrumentation points. If you select a non-leaf node, instrumentation is placed at all leaf nodes that are child nodes of the selected non-leaf node.

In order for I/O profiling to work correctly, you must instrument at least the **open** and **close** system calls that open and close any file for which you want to obtain performance measurements.

I/O profiling works by intercepting I/O system calls for any files for which you want to obtain performance measurements. In order to obtain the performance measurement data, the IBM HPC Toolkit uses the **I/O profiling options (MIO_FILES)** settings or environment variables. See Table 12 on page 77 for a description of the MIO-related environment variables and their settings.

At a minimum, you must set the **I/O profiling options** setting or the **MIO_FILES** environment variable to:

- Specify a file-name-matching pattern
- Use the MIO trace module
- Specify the **xml** and **events** options to the trace module

Setting **I/O Profiling options** or **MIO_FILES** to ***[trace/xml/events={mioevents}]** sets up MIO to collect the performance data that the plug-in requires. Setting **MIO_FILES** to this value:

- Instructs MIO to apply the options to all files opened by the application

- Generates the performance data in the XML file format required by the plug-in.
- Specifies that the I/O trace file is written to **hpct.mio.mioevents.iot**. For more information, see Appendix A, "Performance data file naming," on page 309.

Run your application by pressing the **Profile** button. Your application runs using the specified setting. For more information, see "Viewing I/O Data" on page 77.

Instrumenting your application before launch

You can start and use Eclipse and load the application binary as described in "Instrumenting your application before launch" on page 31.

Instrumenting your application

The first step in instrumenting your application is to open the executable as described in "Loading the application" on page 32. After the executable has been opened, make sure the **MIO** tab in the **Instrumentation** view is visible by clicking it. This results in an Eclipse window that looks like Figure 44:

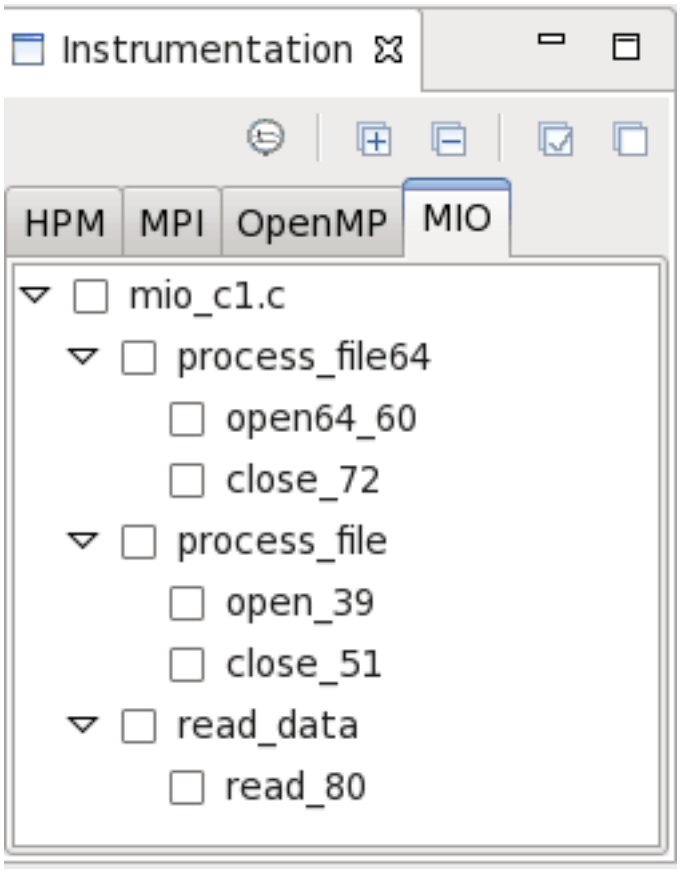


Figure 44. I/O Instrumentation view

The I/O view shows the instrumentation points tree fully expanded. The leaf nodes are labeled with the name of the system call at that location and the line number in the source file. If you select leaf nodes, instrumentation is placed only

at those specific instrumentation points. If you select a non-leaf node, instrumentation is placed at all leaf nodes that are child nodes of the selected non-leaf node.

In order for I/O profiling to work correctly, you must instrument at least the **open** and **close** system calls that open and close any file for which you want to obtain performance measurements.

After you have selected the set of instrumentation that you want, you instrument the application by right-clicking in the **Instrumentation** view, then clicking **Instrument Executable**, or by clicking the **Instrument** icon in the toolbar in the **Instrumentation** view.

Running your application

I/O profiling works by intercepting I/O system calls for any files for which you want to obtain performance measurements. In order to obtain the performance measurement data, the IBM HPC Toolkit uses the **I/O profiling options** (**MIO_FILES)** settings or environment variables. See Table 12 on page 77 for a description of the MIO-related environment variables and their settings.

At a minimum, you must set the **I/O profiling options** setting or the **MIO_FILES** environment variable to:

- Specify a file-name-matching pattern
- Use the MIO trace module
- Specify the **xml** and **events** options to the trace module

Setting **I/O Profiling options** or **MIO_FILES** to ***[trace/xml/events={mioevents}]** sets up MIO to collect the performance data that the plug-in requires. Setting **MIO_FILES** to this value:

- Instructs MIO to apply the options to all files opened by the application
- Generates the performance data in the XML file format required by the plug-in.
- Specifies that the I/O trace file is written to **hpct.mio.mioevents.iot**. For more information, see Appendix A, “Performance data file naming,” on page 309.

If you are using the Profile Configurations dialog, the **MIO** tab of the Profile Configurations dialog looks like Figure 45 on page 77:

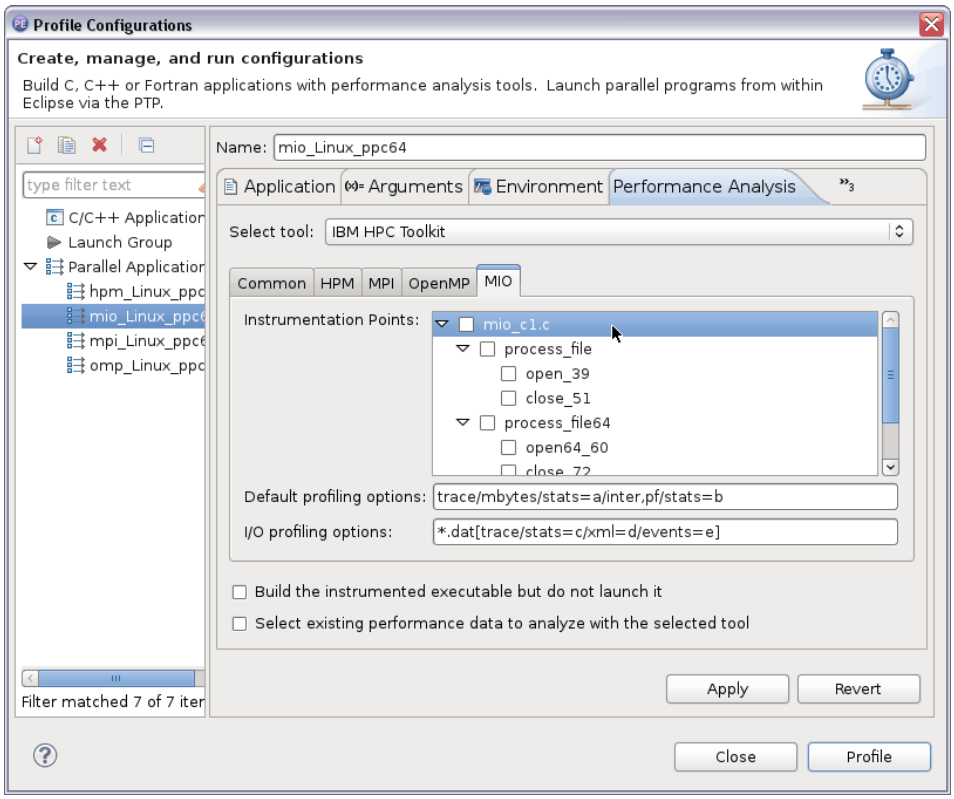


Figure 45. Profile Configurations dialog MIO settings panel

For more information about settings on this panel, use Table 12 to reference the corresponding environment variable description in the "Setting I/O profiling environment variables" on page 219.

Table 12. MIO profile configuration environment variables

Profile configuration field name	Environment variable
Default profiling options	**MIO_DEFAULTS**
I/O profiling options	**MIO_FILES**

Viewing I/O Data

After your application completes, the plug-in attempts to display the I/O profiling data that was collected when the application was run. The plug-in displays the data in a tree format, in which the top-level node is the file that the application read or wrote and the leaf nodes are the I/O function calls your application issued for that file. Figure 46 on page 78 shows the data visualization window with this tree fully expanded.

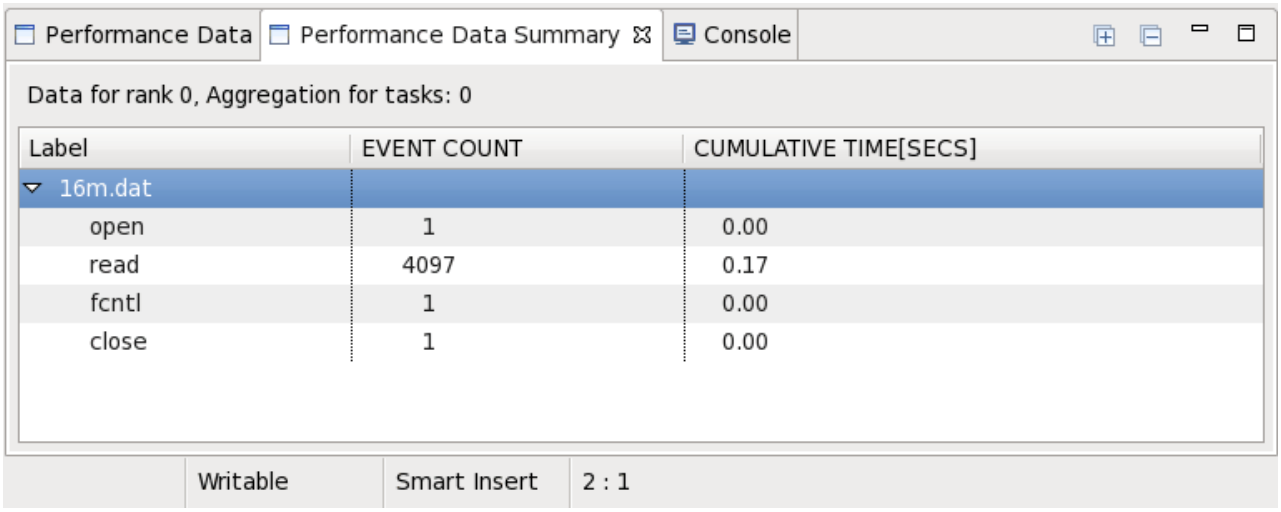


Figure 46. Performance data summary view with I/O profiling data

Each row shows the time spent in an I/O function call and the number of times that function call was executed.

You can view detailed data for a leaf node by right-clicking over it and selecting **Show Metric Browser** from the pop-up menu. A metric browser window contains data for each process that executed that I/O function. You can view all of your performance measurements in a tabular form by selecting the **Show Data as a Flat Table** option from the pop-up menu that appears when you right-click within the **Performance Data Summary** view.

You can view a plot of your I/O measurements by right-clicking in the **Performance Data Summary** view, selecting **Load IO Trace** from the pop-up menu that appears, and specifying the location to download the I/O trace file. After the trace is loaded, the Eclipse window looks like Figure 47:

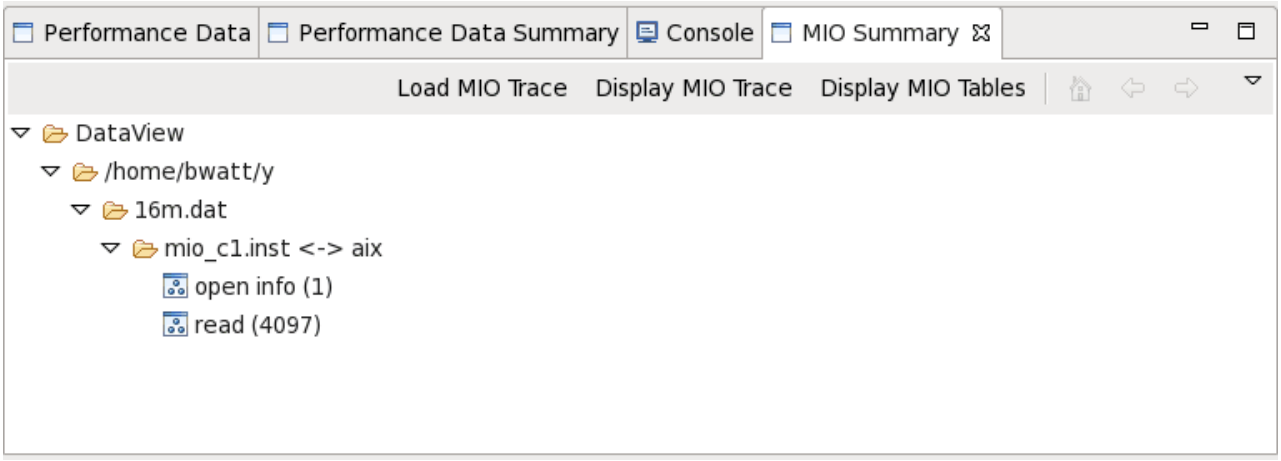


Figure 47. MIO tree view

The **MIO Summary** view contains a tree view of the MIO performance data files. The top-level nodes represent individual performance data files. The next level nodes represent individual files that the application accessed. The next level nodes

represent the application program, and the leaf nodes represent the I/O function calls executed in the application. You can select leaf nodes to include the data from those nodes in the plot window.

You can use the buttons in the view's toolbar or the menu options in the view's drop-down menu to perform the following actions.

Table 13. MIO trace processing actions

Button	Action
Load MIO Trace	Load a new I/O trace file.
Display MIO Trace	Display a new I/O trace file.
Display MIO Tables	Display data from the I/O trace in a tabular format.

After you select one or more leaf nodes from the tree and click the **Display MIO Trace** button, the Eclipse window looks like Figure 48:

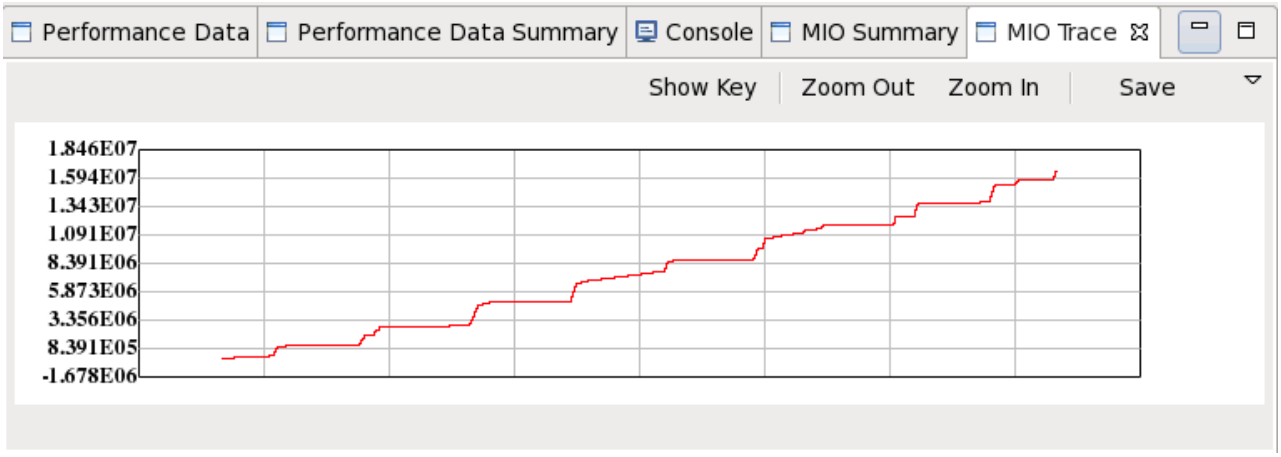


Figure 48. MIO trace view

When the graph is initially displayed, the Y axis represents the file position, in bytes. The X axis of the graph always represents time in seconds.

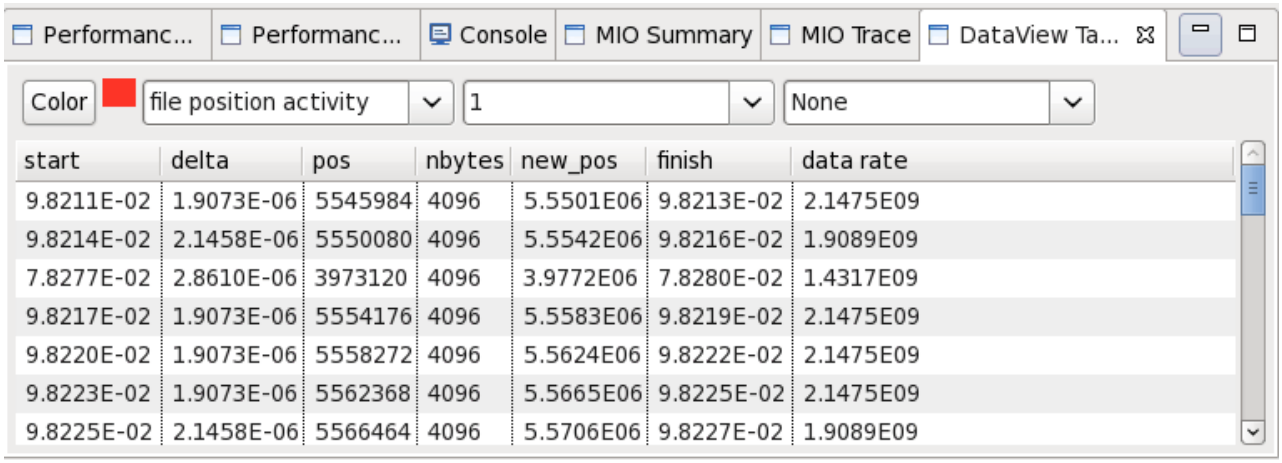
You can zoom in to an area of interest in the graph by left-clicking at one corner of the desired area and dragging the mouse while holding the left button to draw a box around the area of interest and releasing the left mouse button. When you release the left mouse button, the plug-in redraws the graph, showing the area of interest. You can then zoom in and out of the graph by clicking the **Zoom In** and **Zoom Out** buttons at the top of the graph window. As you drag the mouse, the plug-in displays the X and Y coordinates of the lower left corner of the box and the upper right corner of the box and the slope of the line between those two corners as text in the status bar area at the bottom of the view.

You can determine the I/O data transfer rate at any area in the plot by right-clicking over the desired starting point in the plot and holding down the right mouse button, while tracing over the section of the plot of interest. The coordinates of the starting and ending points of the selection region and the data transfer rate (slope) are displayed in the status area at the bottom of the view.

You can save the current plot to a **jpeg** (**.jpg**) file by clicking the **Save** button at the top of the view. A file selector dialog appears, which allows you to select the path name of the file to which the screen image is written.

You can display a pop-up dialog that lists the colors in the current graph and the I/O functions they are associated with by clicking the **Show Key** button at the top of the view.

You can view the I/O profiling data in tabular form and modify the characteristics of the current plot by selecting the **Display MIO Tables** button at the top of the **MIO Summary** view. A window appears, similar to Figure 49, showing the I/O profiling data.



start	delta	pos	nbytes	new_pos	finish	data rate
9.8211E-02	1.9073E-06	5545984	4096	5.5501E06	9.8213E-02	2.1475E09
9.8214E-02	2.1458E-06	5550080	4096	5.5542E06	9.8216E-02	1.9089E09
7.8277E-02	2.8610E-06	3973120	4096	3.9772E06	7.8280E-02	1.4317E09
9.8217E-02	1.9073E-06	5554176	4096	5.5583E06	9.8219E-02	2.1475E09
9.8220E-02	1.9073E-06	5558272	4096	5.5624E06	9.8222E-02	2.1475E09
9.8223E-02	1.9073E-06	5562368	4096	5.5665E06	9.8225E-02	2.1475E09
9.8225E-02	2.1458E-06	5566464	4096	5.5706E06	9.8227E-02	1.9089E09

Figure 49. Data view table view

There are four widgets at the top of the table view that you can use to modify the characteristics of the current plot. You can change the values in these widgets as desired. The selections you make in this view are effective the next time you click the **Display MIO Trace** button in the **MIO Summary** view.

The colored square at the upper left specifies the color to use when drawing the plot. If you click this square, a color selector dialog appears, which allows you to select the color you want to be used in drawing the plot.

The second widget from the left, labeled **file position activity**, selects the metric to be used for the Y and X axis of the plot and also affects the format of the plot. If you select **file position activity**, the Y axis represents **file position** and the X axis represents **time**. If you select **data delivery rate**, the Y axis represents the **data transfer rate** and the X axis represents **time**. If you select **rate vs. pos**, the Y axis represents **data transfer rate** and the X axis represents the **start position** in the file

The third widget from the left specifies the pixel width for the graph that is drawn when the file position metric has been selected from the second widget from the left.

The rightmost widget specifies the metric that has its numeric value displayed next to each data point. You can select any column displayed in the table, or **none** to plot each point with no accompanying data value.

Part 4. The peekperf GUI

The topics in this part provide information about using the **peekperf** GUI.

© Copyright IBM Corp. 2008, 2013

Chapter 8. Using the peekperf GUI

The **peekperf** GUI is a second user interface for the IBM HPC Toolkit. You use the GUI for instrumenting your application, running your instrumented application, and obtaining performance measurements in the following areas:

- Hardware performance counter measurements
- MPI profiling
- OpenMP profiling
- Application I/O profiling

The **peekperf** GUI maps performance measurements to your source code, and allows you to sort and filter that data.

The **peekperf** GUI is started by issuing the command **peekperf**. When you issue the **peekperf** command, the **peekperf** GUI main window is displayed. When the main window opens, you can load an application executable, open visualization data files containing your performance measurements, and open application source files.

You can also invoke **peekperf**, specifying the name of your application, the name of the visualization data files, or both. If you invoke **peekperf** in this manner, **peekperf** automatically opens the files you specified when the main window opens.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, run one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

The following topics explain the individual windows and menus in the **peekperf** GUI. Each type of instrumentation has some unique menus and options in these windows. Those unique menus and options are explained in later topics, which cover the use of **peekperf** for each instrumentation type.

Note: When using **peekperf** with x86 class Linux applications, the options to open a binary and generate an instrumented application are not available.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

You must compile and link your application using the **–g** compiler flag so that **peekperf** will have the line number and symbol table information that it needs in order to show you the instrumentation points in your application, and so that the application can be instrumented.

When you compile an application on a Power Linux system, you will need to use the **-Wl,--hash-style=sysv -emit-stub-syms** compiler flags.

© Copyright IBM Corp. 2008, 2013

When you try to load a Linux 64-bit application, **peekperf** might be unable to locate the main entry point for your application. If **peekperf** cannot locate the main entry point, it will issue a message on the console from which it was invoked, suggesting that you set the **PSIGMA_MAIN** environment variable. If you see this message, you should set **PSIGMA_MAIN** to the main entry point for your application, or the name of the main program for FORTRAN applications.

Loading an application

When you invoke **peekperf** using the **peekperf** command, Figure 50 is displayed:

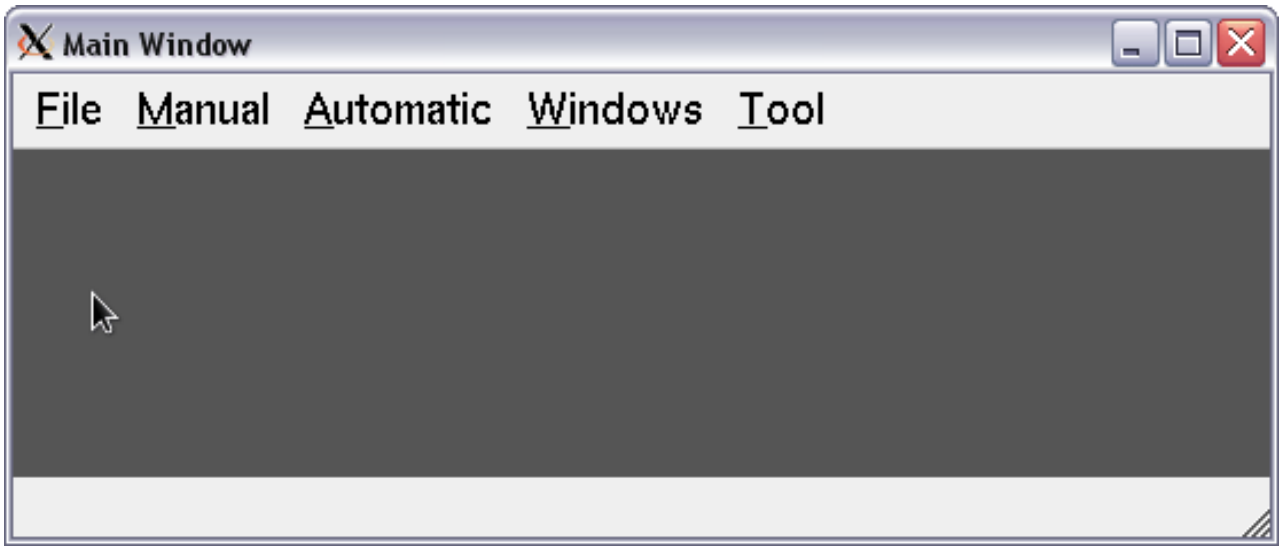


Figure 50. Peekperf Main Window at startup

Load the application, which must be compiled and linked with the **–g** flag, by clicking the **File** menu, then clicking the **Open Binary** selection. A file selector dialog opens. Select the executable file to be opened, then click the **Open** button in that dialog.

On AIX, after you do this, the Linking Option dialog opens, in which you can specify additional arguments, such as libraries to link with your application when loading it in to **peekperf**. If your application requires shared libraries other than **libpmapi**, **libpthreads**, or **libxlsmp**, you must specify them here. If your application requires special linking options, those must also be specified here.

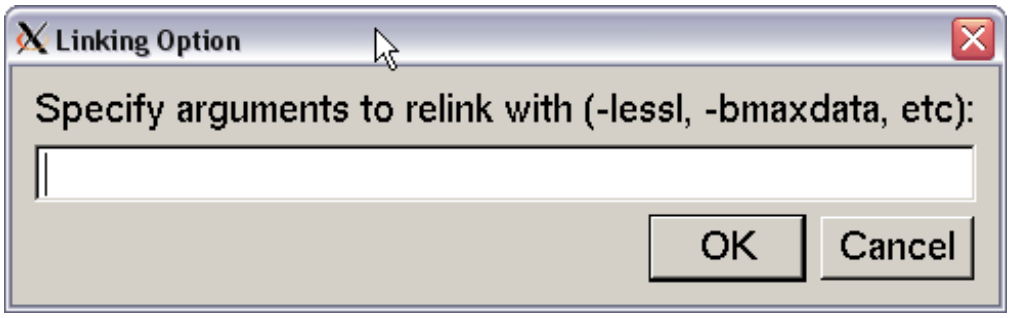


Figure 51. Additional linking options (AIX only)

Specify any linking arguments that you need, then click **OK**. The **peekperf** main window looks similar to Figure 52:

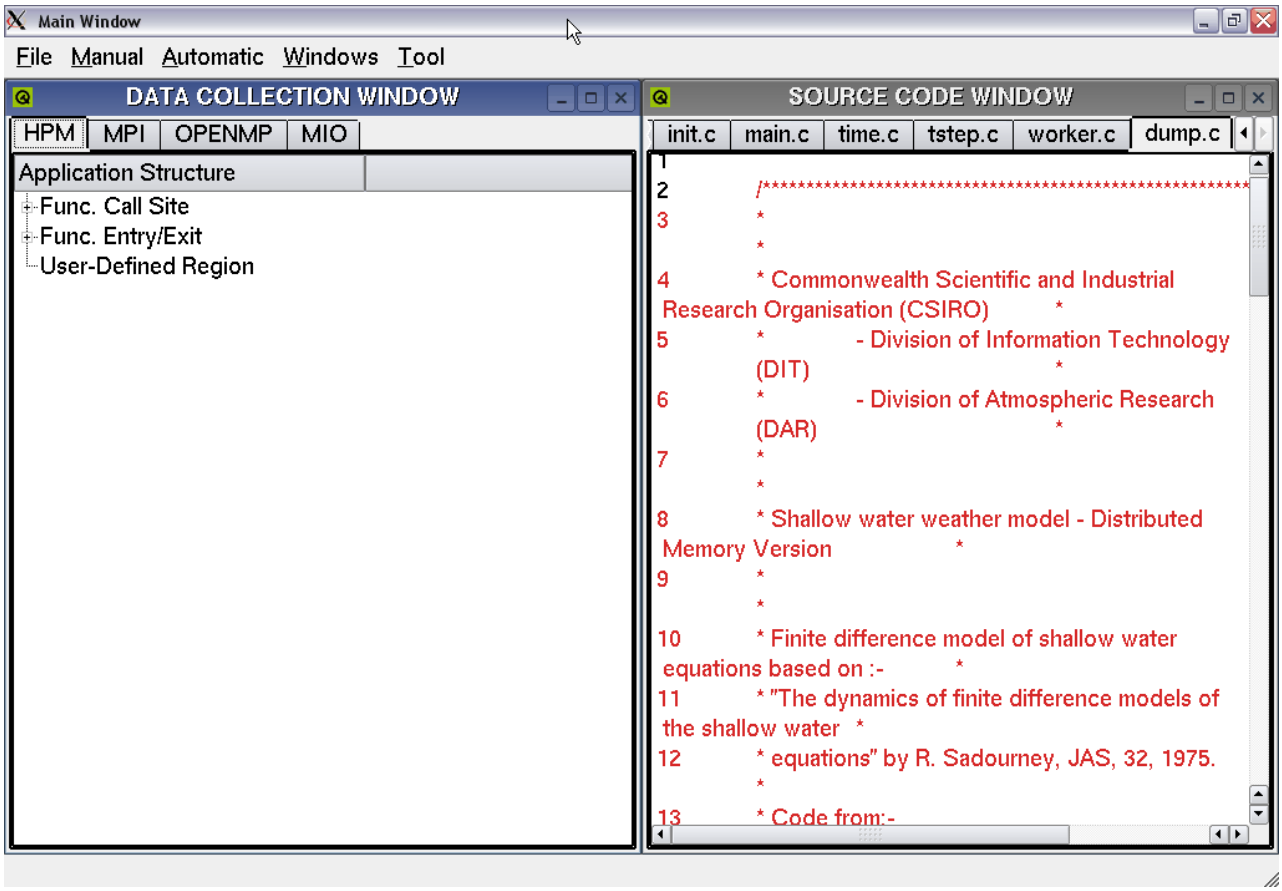


Figure 52. Peekperf data collection

The data collection window has four tabs for the different types of instrumentation that are available. If you click a tab, the data collection window shows you a tree view of the application structure that contains all of the places you can insert that type of instrumentation in the application.

You can instrument your application with as many of the four types of instrumentation as you like in a single session. Note that each type of instrumentation has a small overhead associated with it, and that overhead might affect measurements obtained by other types of instrumentation.

You can expand the tree in the data collection window by clicking the + icons at each node or by right-clicking in the white space in the data visualization window and then clicking **Expand the Tree** in the pop-up menu. This gives you a view showing all the instrumentation points in the application. You can collapse nodes in the tree by clicking the '**–**' icons at each node or by clicking **Collapse the Tree** in the same pop-up menu.

If **peekperf** can find the source files, it opens a source code window showing the application source code. If there are multiple source files in the application, there are tabs at the top of the source code window for selecting the source file to view. As you select instrumentation in the source code window, the affected regions of source code are highlighted in the source code window.

If **peekperf** cannot find the source code for your application, it displays a pop-up dialog asking you if you want to specify the location of the source files. If you choose to do so, a file selector dialog that you use to select the location of the source files is also displayed.

The tree in the data collection window panel presents the program structure and will be created based on the type of performance data. For example, the tree in the HPM panel contains two sub trees. The **Func. Entry/Exit** sub tree shows all the functions in the application. The call sites for each function call are shown in the **Func. Call Site** sub tree.

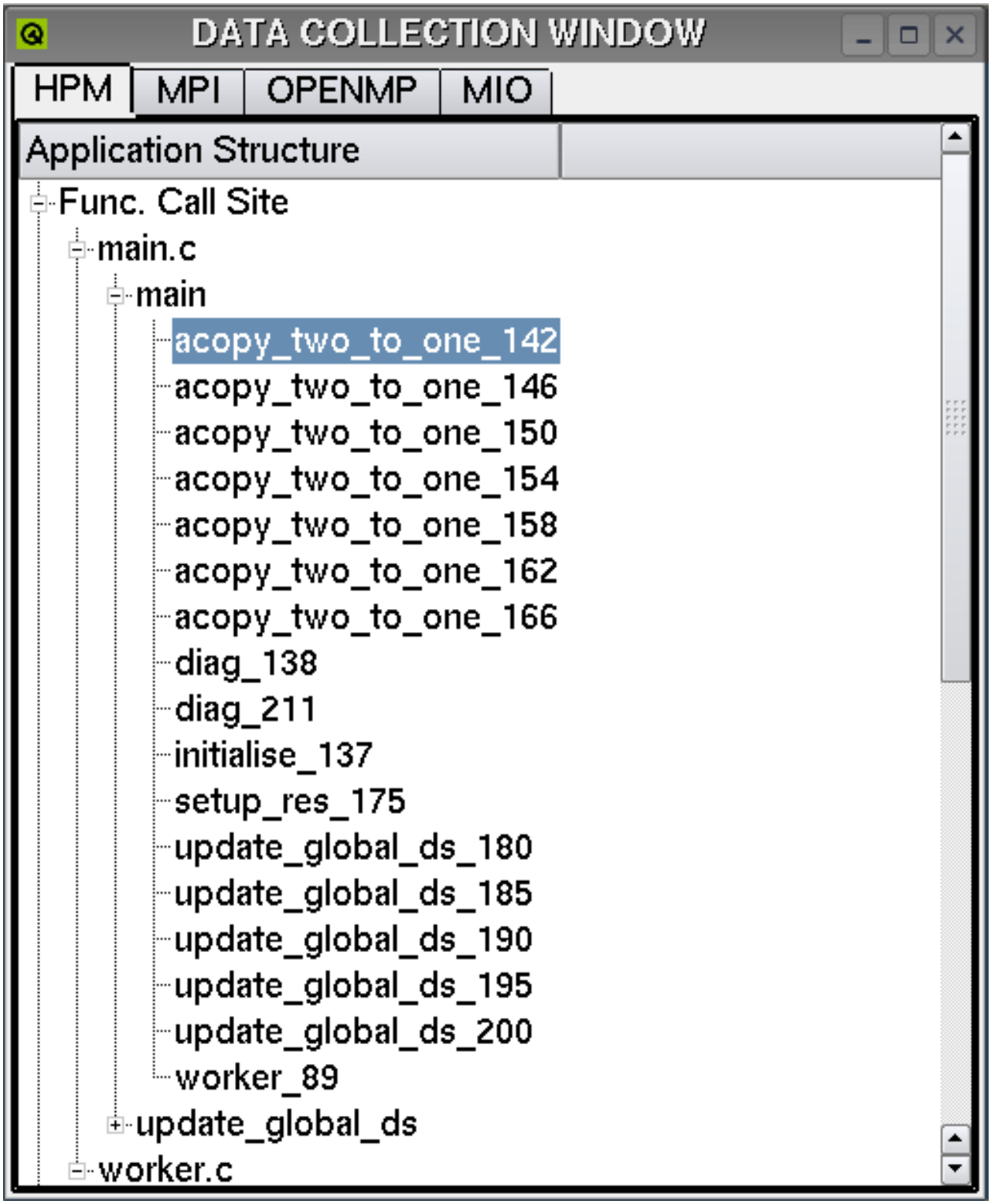


Figure 53. Peekperf data collection window with expanded application structure

The **peekperf** GUI provides some searching capability in this tree. Select the **Search** option in the context menu (that appears when you right-click in white space in the data collection window) and input the keyword. It searches the entire

tree and gives you the information related to the keyword. Figure 54 shows an example of when the keyword is **one**.

Search Results for acopy

	Path	File	Function	Call Site
1	/home/bwatt	copy.c	acopy_one_to_two	
2	/home/bwatt	copy.c	acopy_two_to_one	
3	/home/bwatt	main.c	main	acopy_two_to_one_142
4	/home/bwatt	main.c	main	acopy_two_to_one_146
5	/home/bwatt	main.c	main	acopy_two_to_one_150
6	/home/bwatt	main.c	main	acopy_two_to_one_154
7	/home/bwatt	main.c	main	acopy_two_to_one_158
8	/home/bwatt	main.c	main	acopy_two_to_one_162
9	/home/bwatt	main.c	main	acopy_two_to_one_166
10	/home/bwatt	main.c	update_global_ds	acopy_one_to_two_259
11	/home/bwatt	worker.c	neighbour_receive	acopy_one_to_two_334
12	/home/bwatt	worker.c	neighbour_send	acopy_two_to_one_308
13	/home/bwatt	worker.c	send_updated_ds	acopy_two_to_one_209
14	/home/bwatt	worker.c	worker	acopy_one_to_two_110
15	/home/bwatt	worker.c	worker	acopy_one_to_two_114
16	/home/bwatt	worker.c	worker	acopy_one_to_two_118
17	/home/bwatt	worker.c	worker	acopy_one_to_two_122
18	/home/bwatt	worker.c	worker	acopy_one_to_two_126
19	/home/bwatt	worker.c	worker	acopy_one_to_two_130
20	/home/bwatt	worker.c	worker	acopy_one_to_two_134

Figure 54. Peekperf search results window

The points you want to instrument are selected by clicking the left mouse button on the nodes in the tree in the data collection window. For example, when you click the **main.c** node, the **main.c** node and all the children of the **main.c** node are selected and highlighted. If you want to deselect it, click the left mouse button again on the selected node. All the children including this node will be deselected. If you want to clear all your current selected instrumentation, select the **Clear Instrumentation** option from the **Tool** menu. Note that at this point, the instrumented application is not generated yet. You are only selecting the places to put the instrumentation.

Instrumenting your application

After you browse through each panel and identify the instrumentation you want, where you must have at least one instrumentation point selected, click the **Generate an Instrumented Application** option from the **Automatic** menu. When **peekperf** has created an instrumented application, it will pop up an alert to indicate that the application has been instrumented. It will also tell you the name of the instrumented application.

Running your application

After you have instrumented your application, you can run your application from the **peekperf** GUI. The instrumented application will run and generate the data files containing measurements from the instrumentation that you selected.

If your application requires environment variables, such as those used by Parallel Environment Runtime Edition, and those environment variables were not set before you invoked **peekperf**, you must set them before running the application. You set environment variables by selecting the **Set the Environment Variable** option from the **Automatic** menu. After you select this option, a pop-up dialog appears in which you fill in the name of the environment variable and the value.

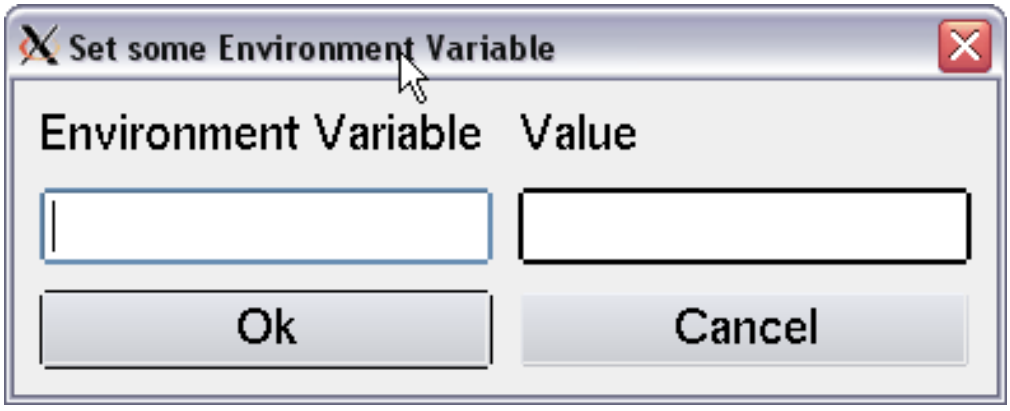


Figure 55. Peekperf environment settings window

After you enter a name and value, click **OK** to set the environment variable. The environment variable will remain set for the duration of your **peekperf** session, or until you change it again.

Run the application by selecting **Run an Instrumented Application** from the **Automatic** menu. **Peekperf** displays a set of dialogs requesting information needed to run the application. The first dialog asks if you want to modify a script or other file before running the application. If you click **yes**, you can edit any file before running your application.

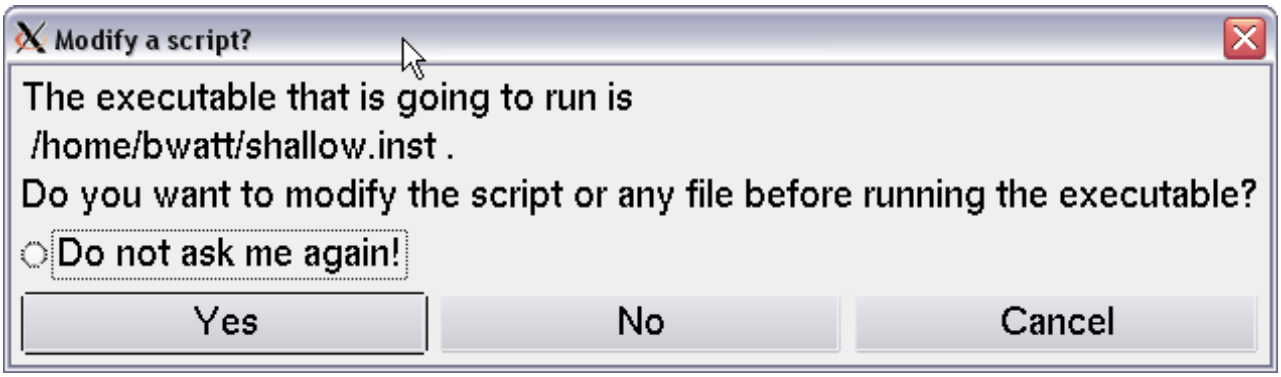


Figure 56. Peekperf run dialog 1 of 3

If you select **Yes**, a file selector dialog is displayed allowing you to select the file that you want to edit. After you select the file, an xterm window running the **vi** editor opens. You can edit the file and save your changes. If you select **No**, **peekperf** proceeds to the next dialog. If you select **Cancel**, the application is not executed.

The next dialog allows you to modify the path of the command you want to run.



Figure 57. Peekperf run dialog 2 of 3

This dialog is initially filled in with the path name of the instrumented application. You can change this path to the path of a script that sets up the environment for your application and invokes it. You can also specify command line arguments for your application. Click **OK** to proceed, or **Cancel** to cancel application execution.

If you are running an application with a large number of tasks, you may generate and collect performance data from a subset of tasks by specifying the **hpcrun** command in Figure 57, followed by command line parameters for the **hpcrun** command, and then the application name and application parameters.

See “Collecting performance data from a subset of parallel application tasks” on page 5 for a description of the **hpcrun** command.

The third dialog asks if you want **peekperf** to attempt to open the instrumentation data (**.viz**) files after your application runs.

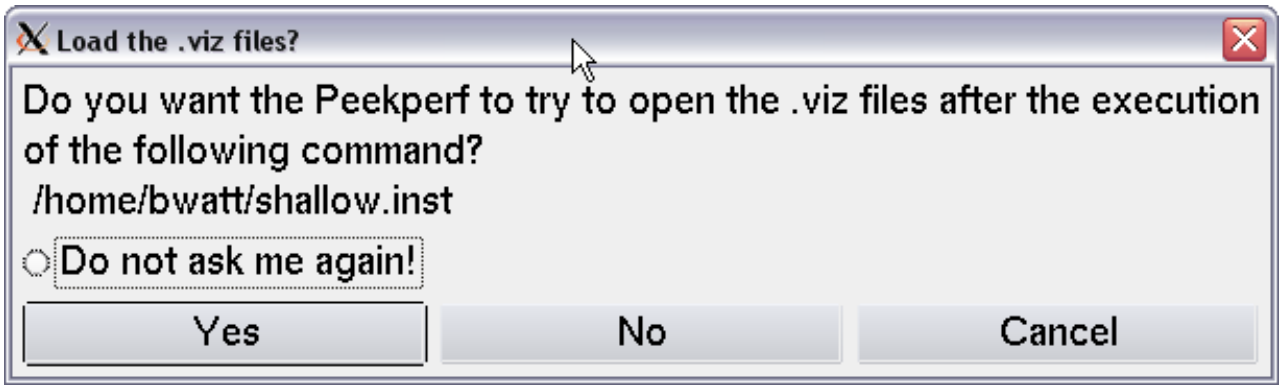


Figure 58. Peekperf run dialog 3 of 3

Select **Yes** if you want **peekperf** to attempt to open the **.viz** files, **No** if you do not want **peekperf** to try to open the **.viz** files, or **Cancel** to cancel application execution. After you select **Yes**, your application runs to completion.

Note: Figure 56 on page 90 and Figure 58 have a button labeled **Do not ask me again** and Figure 57 on page 90 has a button labeled **Remember the command**. If you select any of these buttons, **peekperf** remembers the settings in that dialog and does not display the dialog again in your **peekperf** session. If you exit and restart **peekperf**, these dialogs appear again the first time you run an application.

Viewing performance data

After your application completes, if you specified that **peekperf** should automatically open the performance data files, **peekperf** will try to open the **.viz** files that were generated by running your instrumented application. If you did not specify that **peekperf** should automatically open the **.viz** files, you can manually open them by selecting the **Open Performance Data** option from the **File** menu. **Peekperf** displays a file selector dialog in which you can select one or more **.viz** files to open.

After the **.viz** files are opened, **peekperf** will display them in a data visualization window as shown in Figure 59 on page 92:

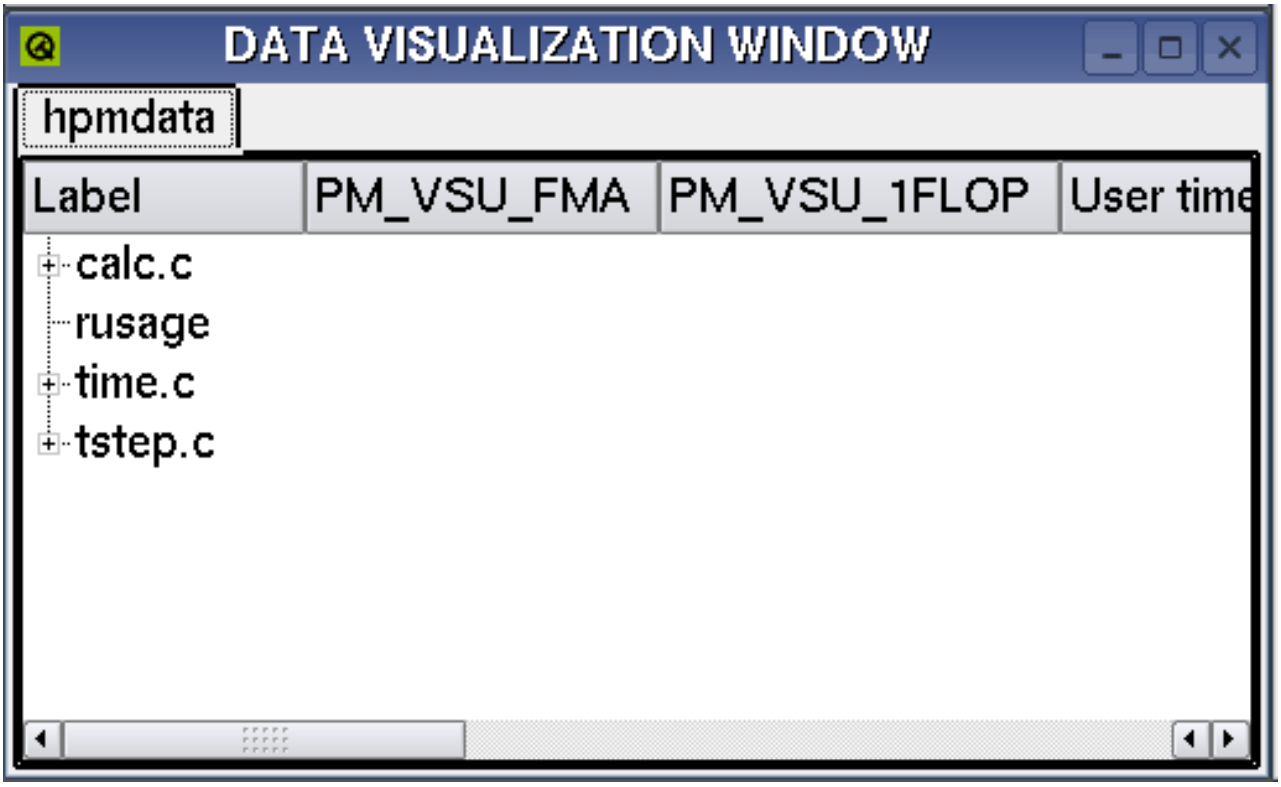


Figure 59. Peekperf data visualization

The data visualization window has one or more tabs, where each tab contains the data from one set of **.viz** files. You can switch between sets of performance data by clicking the tabs for each set of data.

The data visualization window displays a tree view of the data collected. You can expand nodes in the tree by clicking the + icons next to the nodes or by selecting the **Expand the Tree** option from the pop-up menu that appears when you right-click in white space in this window. You can collapse nodes by clicking the '**-**' icon next to the nodes or by selecting **Collapse the Tree** from the same pop-up menu.

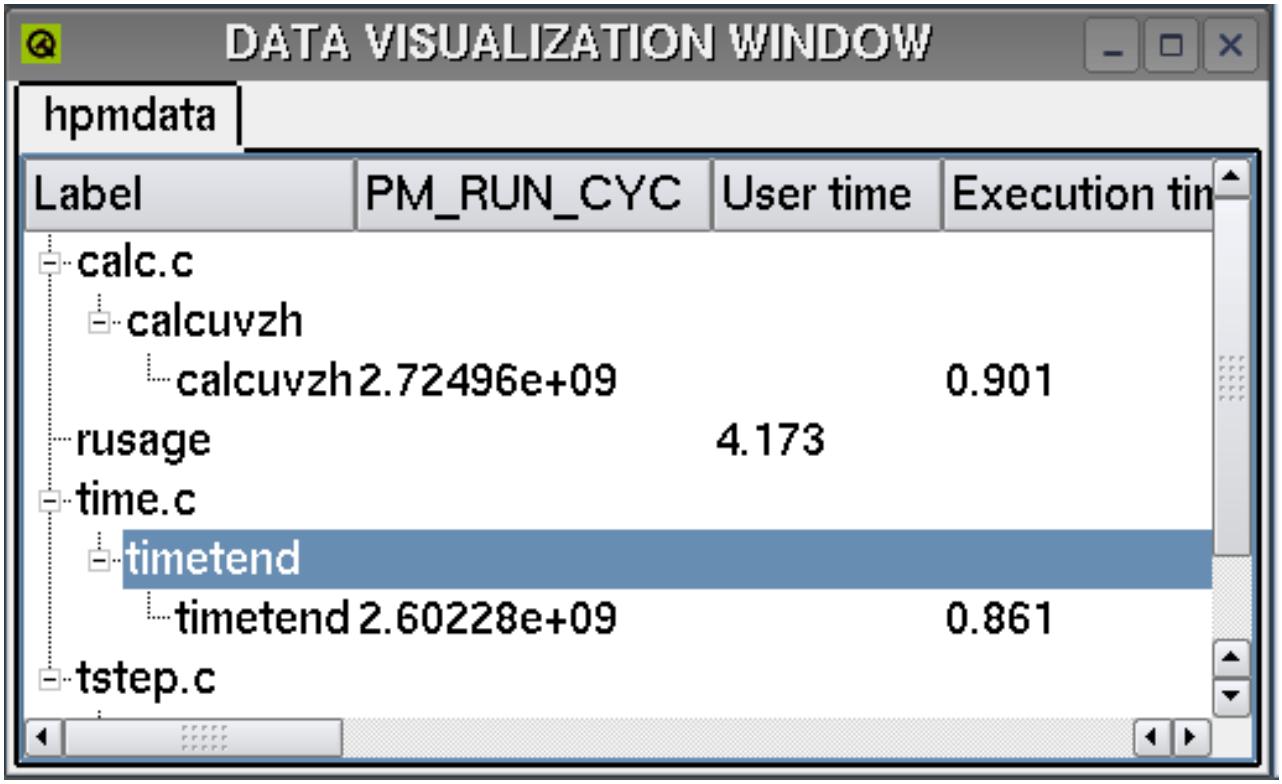


Figure 60. Peekperf expanded data visualization window

Each leaf node in the tree shows some of the data that was obtained for each instrumented location in the application that was actually executed. You can sort the tree by any column. To do this, click the column header for the desired column.

If you right-click in white space in the data visualization window, a pop-up menu with several options appears, including the options to expand and collapse the tree. If you select the **Set Filter** option from this menu, a pop-up dialog appears where you can set filtering criteria to reduce the data shown in this window.

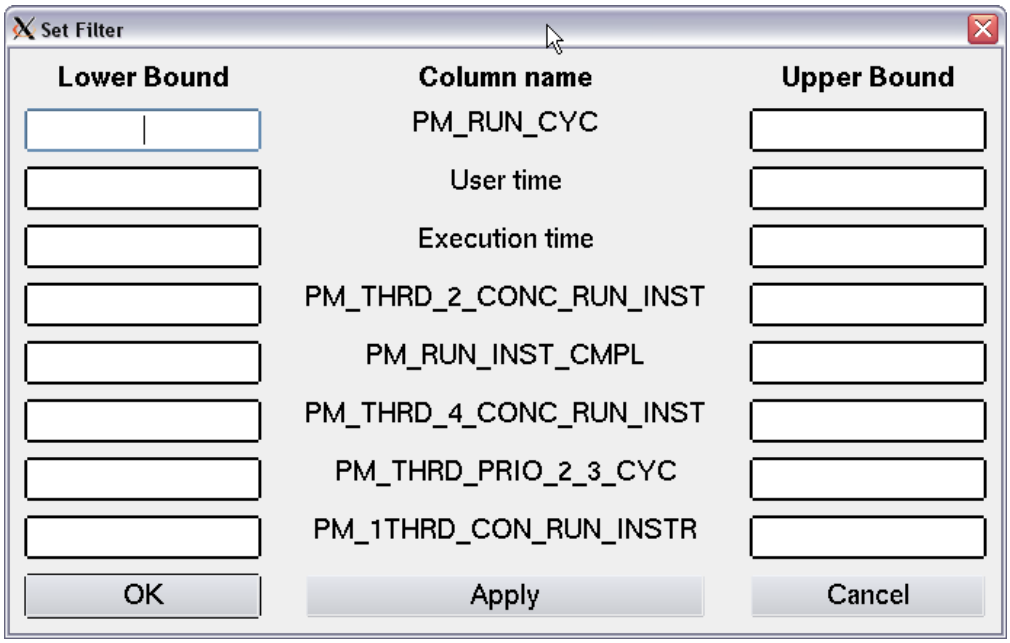


Figure 61. Peekperf data filtering dialog

This dialog has fields for upper and lower bounds for each data column in the tree. If you enter upper and lower bounds for a field then click **OK** or **Apply**, only rows with a data value within those bounds will be shown. You can also enter only a lower bound or an upper bound. You can enter filter criteria for one or more columns. You can remove the filter by either opening the filter dialog, clearing all the fields and clicking **OK** or by selecting **No Filter** from the pop-up menu in the data visualization window.

If you left-click on a node in the tree where data is shown, **peekperf** highlights the corresponding code region in the source code window. If you right-click on a node in the tree in which data is shown, then select Show Metric Browser window from the pop-up menu that appears, a Metric Browser window opens. The Metric Browser window displays a table containing additional data for all application tasks that executed code at that instrumentation point. You can have more than one of these tables open at the same time.

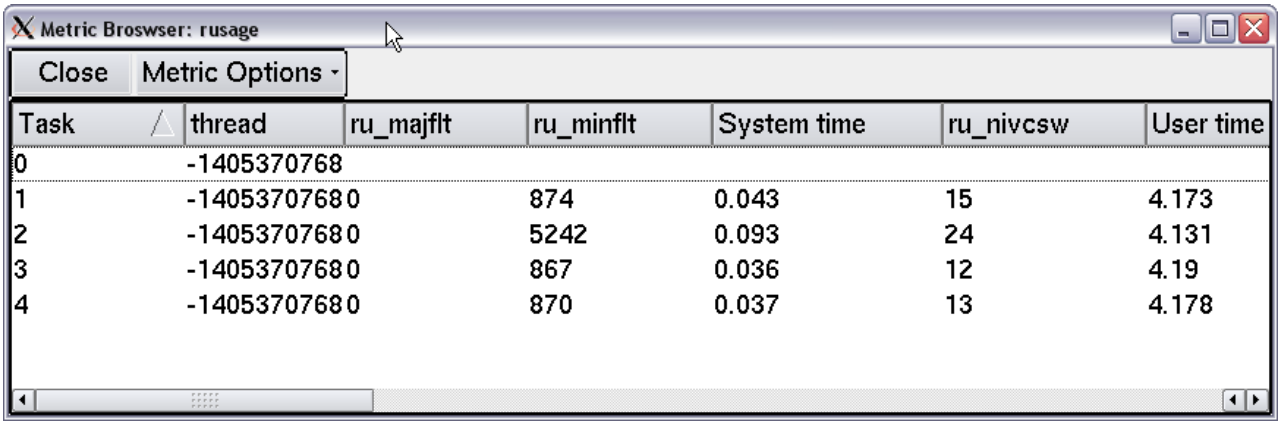
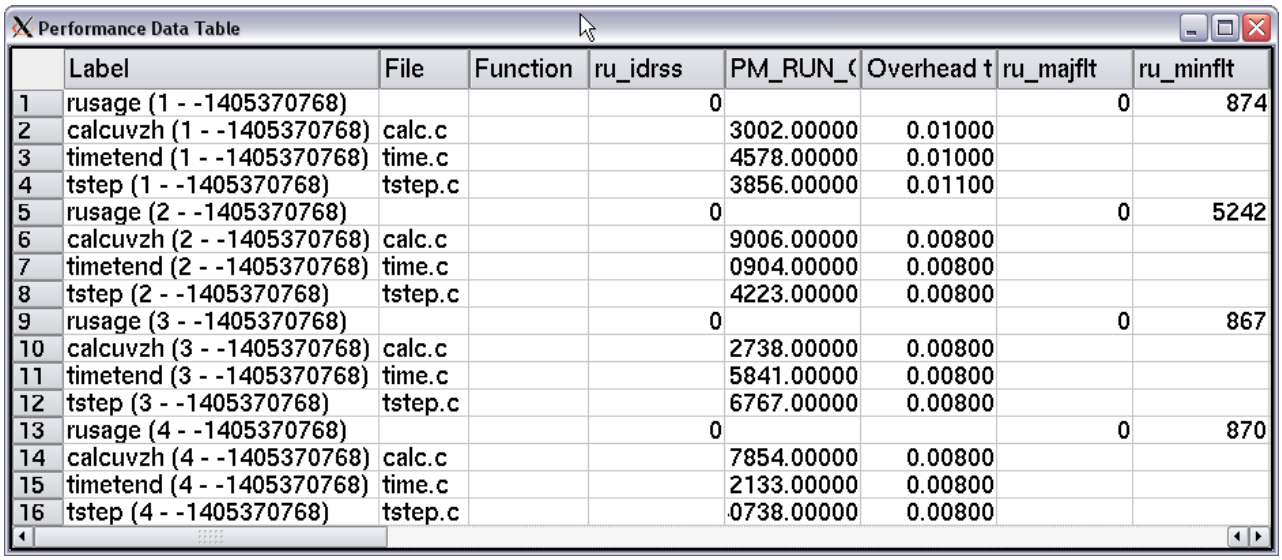


Figure 62. Peekperf Metric Browser window

You can sort data in this table by any column by clicking the column header for the desired column.

You can select the columns that you want to display in this table by clicking the **Metric Options** menu and then selecting a column from the menu. Selecting a menu option toggles the display state of the selected column, where checked menu items represent columns that are already visible.

You can display the data in the data visualization window in tabular form by clicking the **Show as Table** option in the pop-up menu in that window. Selecting that option results in a pop-up window similar to Figure 63:



	Label	File	Function	ru_idrss	PM_RUN_C	Overhead t	ru_majflt	ru_minflt
1	rusage (1 - -1405370768)			0			0	874
2	calcuvzh (1 - -1405370768)	calc.c			3002.00000	0.01000		
3	timetend (1 - -1405370768)	time.c			4578.00000	0.01000		
4	tstep (1 - -1405370768)	tstep.c			3856.00000	0.01100		
5	rusage (2 - -1405370768)			0			0	5242
6	calcuvzh (2 - -1405370768)	calc.c			9006.00000	0.00800		
7	timetend (2 - -1405370768)	time.c			0904.00000	0.00800		
8	tstep (2 - -1405370768)	tstep.c			4223.00000	0.00800		
9	rusage (3 - -1405370768)			0			0	867
10	calcuvzh (3 - -1405370768)	calc.c			2738.00000	0.00800		
11	timetend (3 - -1405370768)	time.c			5841.00000	0.00800		
12	tstep (3 - -1405370768)	tstep.c			6767.00000	0.00800		
13	rusage (4 - -1405370768)			0			0	870
14	calcuvzh (4 - -1405370768)	calc.c			7854.00000	0.00800		
15	timetend (4 - -1405370768)	time.c			2133.00000	0.00800		
16	tstep (4 - -1405370768)	tstep.c			0738.00000	0.00800		

Figure 63. Peekperf performance data table window

You can sort this table on any column by clicking the column header for the desired column. You can hide columns in the table by right-clicking in the data area of the desired column in the table, then clicking the option in the pop-up menu to hide that column. You can show hidden columns by right-clicking anywhere in the data area of the table and then selecting the option to make the desired column visible again. You can save the table in a CSV format, readable by most spreadsheet programs, by right-clicking in the data area of the table, then selecting the **Save Table to File** option. A file selector dialog appears, in which you can specify the path name of the file and then click the **Save** button to save the table data.

Viewing MPI traces

You can view a trace of MPI activity in your application if you instrument your application with MPI instrumentation. After you instrument your application and run it, you can view the trace by selecting the **View Tracer** option from the pop-up menu that is available by right-clicking in the data visualization window. When you do this, the windows shown in Figure 64 on page 96 and Figure 65 on page 96 open to display the MPI trace.

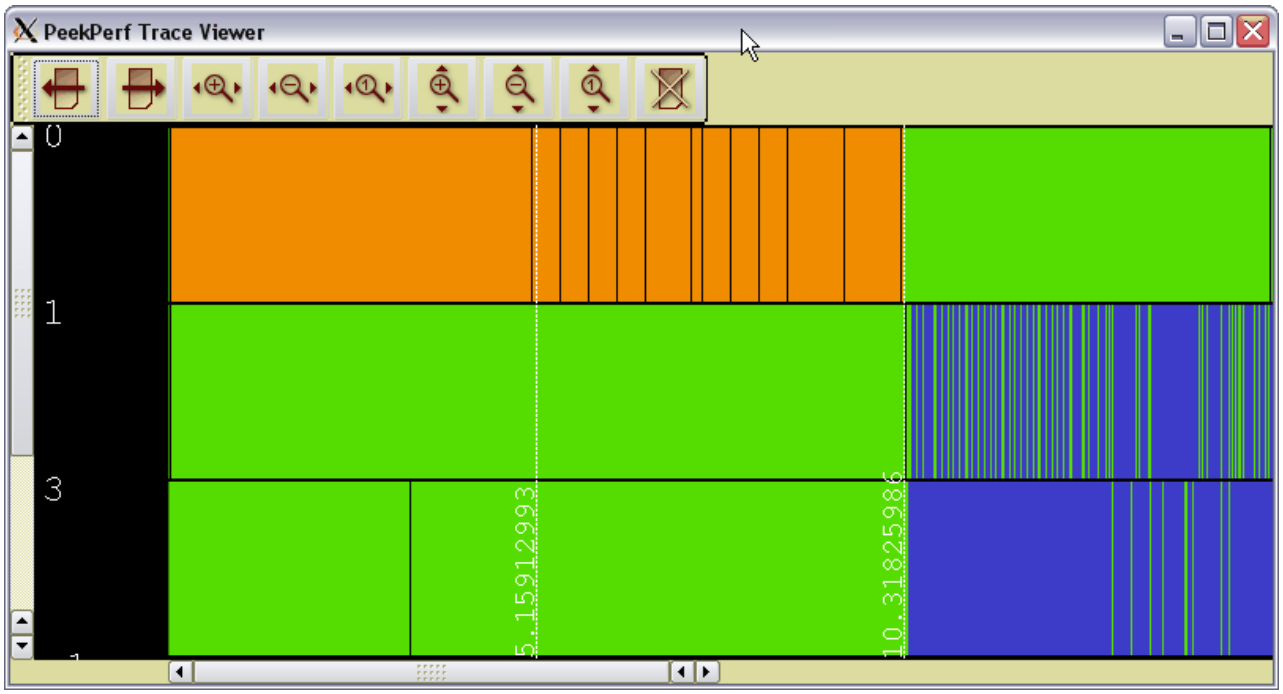


Figure 64. Peekperf trace viewer window

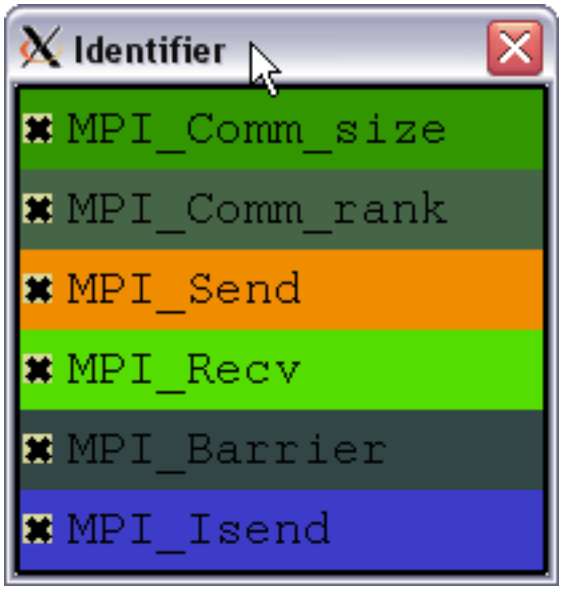


Figure 65. Peekperf trace viewer identifier window

You can obtain information about each MPI call, filter the trace display to show only interesting MPI calls, and zoom in and out of the trace display. The details of how to use the trace display are described in "Viewing MPI profiling data" on page 64.

Viewing I/O profiling data

You can view a plot of I/O activity in your application if you instrument it with MIO instrumentation. After you instrument your application and run it, you can view the plot by selecting the **View Tracer** option from the pop-up menu available by right-clicking in the data visualization window. When you do this, Figure 66 is displayed:

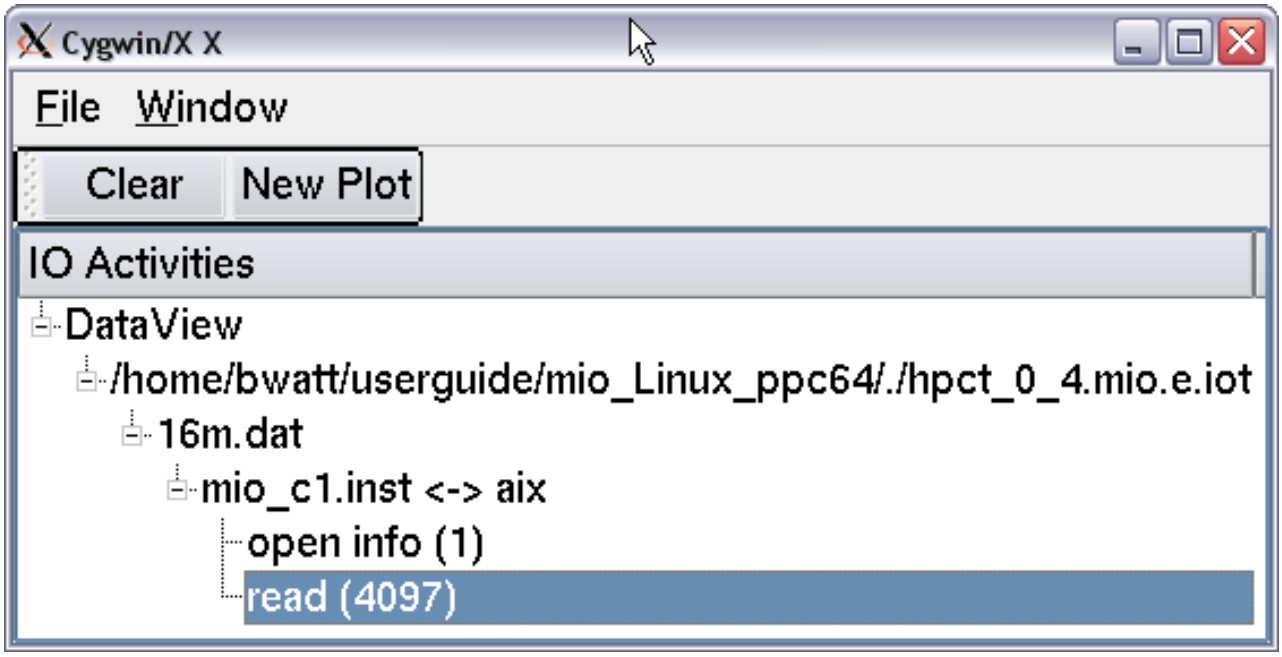


Figure 66. Peekperf I/O profiling selection window

Select the metrics you are interested in, then press the **New Plot** button to display the DataView Plot window.

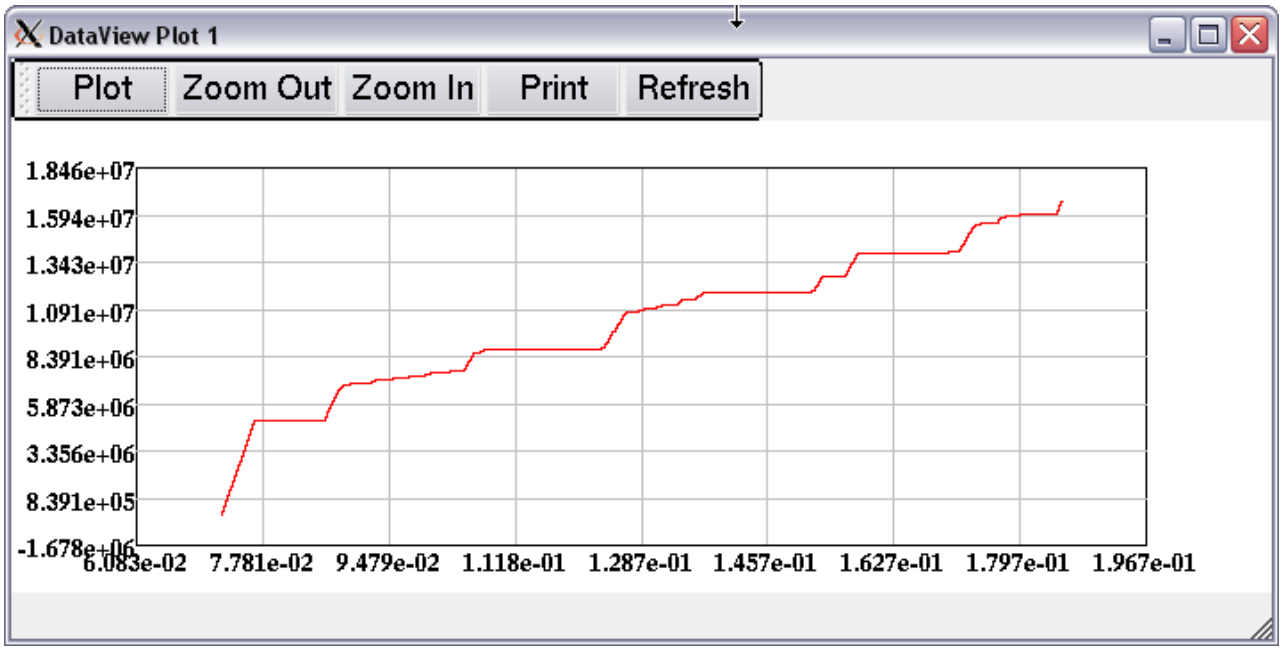


Figure 67. Peekperf data plot window

You can view the I/O activity plot, measure I/O performance at any point in the plot, and tailor the plot with additional information. The details of how to use this plot are described in Chapter 12, "Using I/O Profiling in peekperf," on page 123.

Compiling, editing, and running programs using peekperf

You can edit, compile, and run applications within the **peekperf** GUI.

To edit a source file, right-click in the source code window and select the **Edit** option from the pop-up menu. The **peekperf** GUI opens an xterm window, which allows you to edit the source code using the **vi** editor. After you save the changes to the source file, if you left-click in the source code window, **peekperf** detects the changed file and prompts you with a pop-up dialog notifying you that the source file might have changed and asking if you want to reload the source file into the source code window. If you click **yes**, the source code is reloaded. You can also edit any file from within **peekperf** by selecting the **Edit** option from the **Manual** menu. The **peekperf** GUI displays a file selector dialog prompting you for the path name of the file to edit. After you select the file and click the **Open** button in the file selector dialog, **peekperf** opens an xterm window where the **vi** editor is run. You can edit the source file and save the changed file, if needed.

You can compile an application by selecting the **Compile** command from the **Manual** menu. The **peekperf** GUI displays a pop-up dialog prompting you for the compile command to run. This can be a compiler invocation, a **make** command, or any other command that you use to compile an application. When you click **OK**, **peekperf** runs the command you specified, where the current working directory is the directory where you started **peekperf**. When your compilation command finishes, **peekperf** displays a dialog containing all of the error messages issued by that command.

You can run an application in **peekperf** by selecting the **Run** option from the **Manual** menu. When you do this, **peekperf** displays a file selector dialog from

which you can select the application to be executed. After you click the **Open** button, **peekperf** displays the same three pop-up dialogs, as described in "Running your application" on page 89, and then runs the application.

Miscellaneous peekperf features

The following miscellaneous actions can be performed within **peekperf**:

- Open one or more source files by selecting the **Open Sources** option in the **File** menu. When you select this option, **peekperf** displays a file selector dialog in which you can select one or more source files. When you click **OK** in the file selector dialog, **peekperf** loads the selected source files into the source code window, with a separate tab for each source file.
- Open one or more **.viz** files by selecting the **Open Performance Data** option in the **File** menu. When you select this option, **peekperf** displays a file selector dialog in which you can select one or more **.viz** files. When you click the **Open** button in this dialog, **peekperf** loads the selected **.viz** files into the data visualization window, creating a new tab in that window.
- Close the current application binary by selecting **Close Binary** from the **File** menu. The **peekperf** GUI unloads the application binary and closes the data collection window. You can select another application to work with by selecting the **Open Binary** option in the file menu.
- Close the source file currently displayed in the source code window by either selecting the **Close the Active Source File** option in the **File** menu or by right-clicking in the source code window and selecting **Close this File** from the pop-up menu that appears.
- Close all source files by selecting **Close all Sources** in the **File** menu. The **peekperf** GUI closes all source files it has loaded and then closes the source code window.
- Close the visualization data file, visible in the data visualization window, by selecting the **Close Active Data** option from the **File** menu.
- Close all visualization data files and the data visualization window by selecting the **Close All Data** option from the **File** menu.

Chapter 9. Using Hardware performance counters in peekperf

The topics that follow will describe how to use the hardware performance counters in **peekperf**.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

You start **peekperf** and load the application binary as described in Chapter 8, "Using the peekperf GUI," on page 83.

You must ensure that several environment variables that are required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

Instrumenting your application

Note: On x86 systems, the instrumentation process described here is not supported. You must insert appropriate instrumentation calls into your application, then rebuild your application, linking with the **libhpc** hardware performance counter library.

After the data collection window opens, you should select the **HPM** tab in that window. This tab shows the possible instrumentation, similar to Figure 68 on page 102.

Your application should not contain any calls to functions that are listed in Table 53 on page 261 under "Hardware performance monitoring", because those calls might interfere with correct operation of the instrumentation you insert using **peekperf**. Your application should not be linked with the hardware performance counter library, (**libhpc.a** for AIX or **libhpc.so** for Linux). Your application must be compiled and linked with the **–g** compiler flag.

© Copyright IBM Corp. 2008, 2013

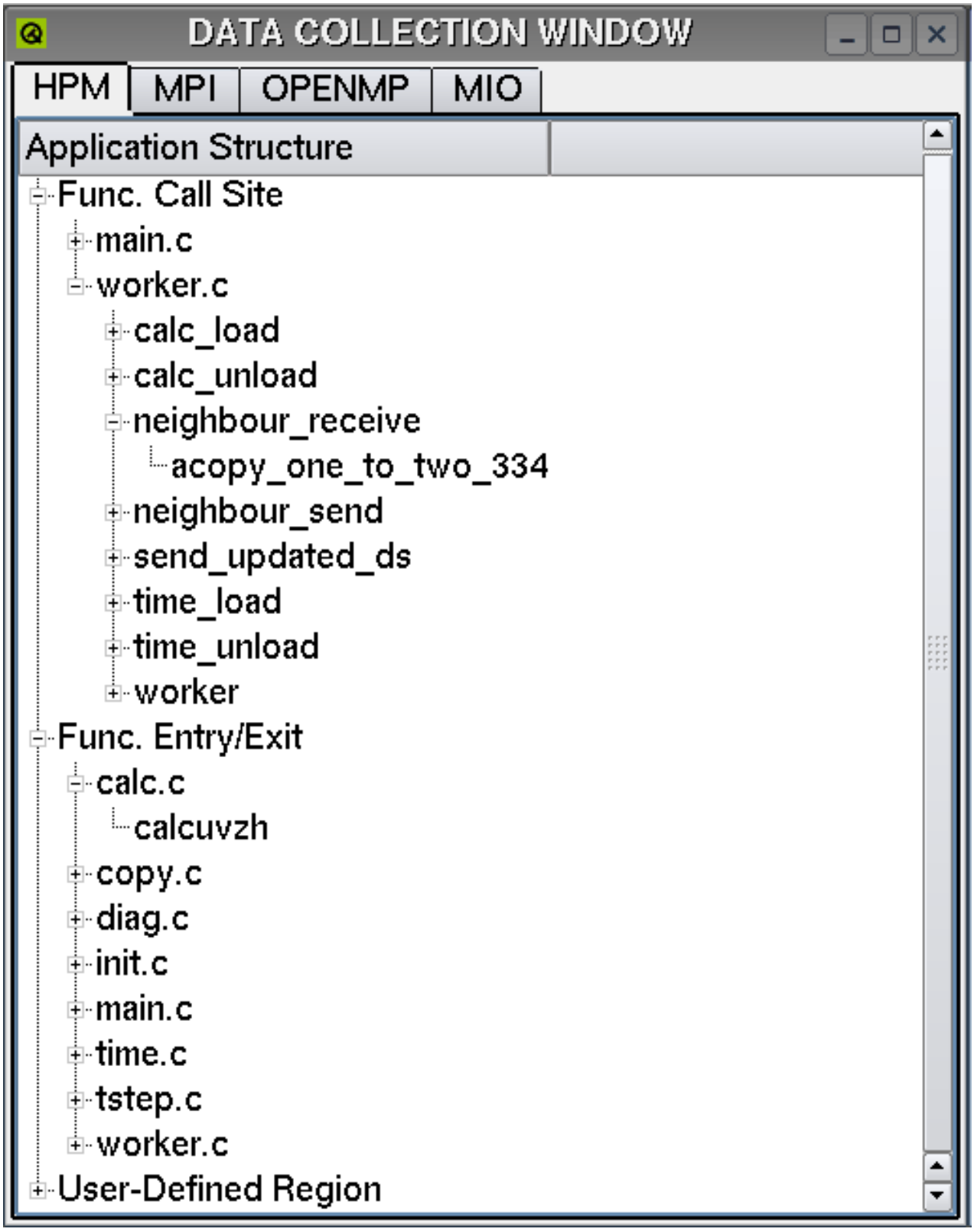


Figure 68. Peekperf HPM data collection window

Figure 68 shows the application structure tree fully expanded. There are three classes of instrumentation shown in this tree:

- The first class is instrumentation of function call sites, or locations in the application where a function call is made. The leaf nodes in this section of the tree are labeled with the name of the function being called, with the line number of the function call appended.
- The second class of instrumentation is function entry and exit points. The leaf nodes in this section of the tree are labeled with the name of the function.
- The third class of instrumentation is the user-defined region, which is a user-specified region of code, such as a loop, that is instrumented.

You can select any region of code to be instrumented, but be careful to ensure that the code at both the beginning and ending of each region is executed in each execution of that region. Leaf nodes in this section of the tree are labeled with the source file name with the starting and ending line numbers for the region appended.

If you instrument a function call site, the instrumentation obtains hardware performance counter measurements for that instance of the function being called, and reports them independently of any other function call to that same function. If you instrument function entry and exit points, the instrumentation obtains hardware performance counter measurements for that function for every time that function is called, regardless of the caller. If you instrument a user-defined region, hardware performance measurements are obtained specifically for that region.

User-defined regions are specified by highlighting a region of source code and then adding that region to the list of user-defined regions in the data collection window. To add a region to the user-defined region list, select that region by selecting the desired source file in the source code window, left-clicking at the starting line of the region, and dragging while the left mouse button is pressed until all desired source code lines are highlighted. After the desired lines are selected, right-click in the source code window and select the **Add to HPM** option from the pop-up menu. You may have as many user-defined regions as you want, and user-defined regions may overlap.

If necessary, you can remove a region of code from the user-defined region list by selecting the same lines of source code as you selected when creating the region, right-clicking, and selecting the **Remove from HPM** option from the pop-up menu.

You can select any combination of function call sites, function entry and exit points and user-defined regions that you want to instrument. You can select instrumentation by individually selecting leaf nodes, or you can select a group of leaf nodes by selecting a higher level node in the tree. You can also deselect instrumentation by clicking again on a highlighted node.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running your program

Before you can run your instrumented application, you must make sure that any environment variables required by the hardware performance counter instrumentation are set correctly. The description of “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 265 contains a complete list of environment variables that can be used. You must ensure that the **HPM_VIZ_OUTPUT** environment variable is either unset or set to

yes. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

You can set these environment variables before you start **peekperf**, or you can set them by selecting the **Set the Environment Variable** option from the **Automatic** menu. Any environment variables you set within **peekperf** remain set for the remainder of the **peekperf** session unless you reset them.

You should also select the hardware counter group before you run your program. You can do this by selecting the **Set the Counter Group** option from the pop-up menu that appears when you right-click in the white space in the data collection window. A pop-up dialog appears after you do this.

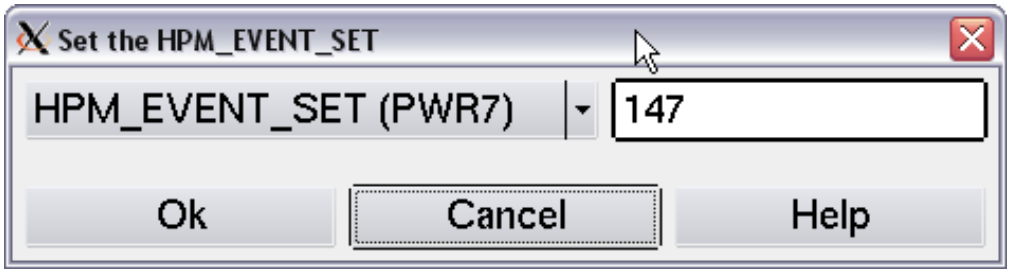


Figure 69. Peekperf hardware counter group selection dialog

Select the specific hardware architecture on which your application is running and then enter the hardware counter group number in the input field. If you click the **Help** button, **peekperf** will pop up another window that contains a list of hardware counter groups, the hardware counters in each group, and a short description of each counter. Click **OK** to select the hardware counter group you want.

After you have set up the environment for your application, you can run it by selecting the **Run an Instrumented Application** from the **Automatic** menu and responding to the dialogs as described previously.

Viewing hardware performance counter data

The **Peekperf** GUI tries to open the **.viz** files after your application has completed execution if you requested **peekperf** to do so when you ran your application. If **peekperf** does not automatically open the **.viz** files, you can open them manually by selecting **Open Performance Data** from the **File** menu and selecting **.viz** files from the file selector dialog.

You can see a more detailed view of the data you obtained by right-clicking over a leaf node in the data visualization window, then select **Show Metric Browser Window** from the pop-up menu that appears, which opens a metrics browser window, or by right-clicking in white space in the data visualization window and selecting **Show as Table** from the pop-up menu, which opens a table view of the performance data.

If you load multiple **.viz** files into the data visualization window, you can view each process's performance data separately. If you want to see performance data from a different process, right-click in the white space in the window and select

Show Other Rank from the pop-up menu. A pop-up dialog appears containing a dropdown list that includes the process IDs of other tasks that can be selected.

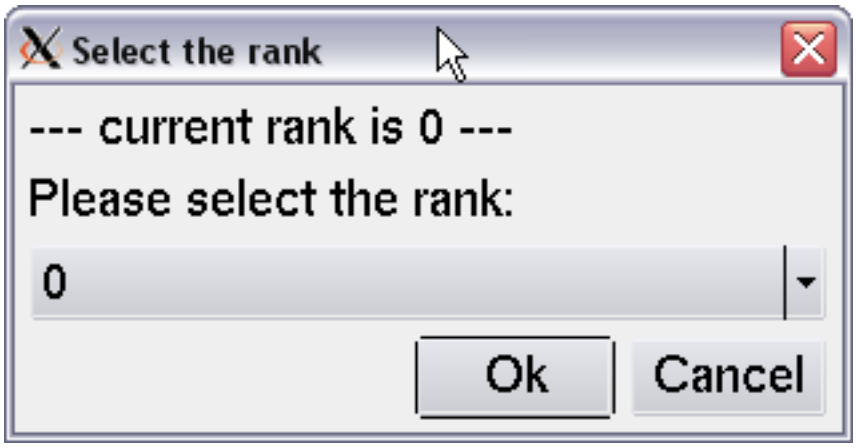


Figure 70. Peekperf task rank selection dialog

Chapter 10. Using MPI profiling in peekperf

The following topics will provide information about using MPI profiling in **peekperf**.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

You cannot use the MPI profiling library to create a trace for an application that issues MPI function calls from multiple threads in the application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

You start **peekperf** and load the application binary as described in Chapter 8, “Using the peekperf GUI,” on page 83.

Instrumenting your application

Your application should not call any of the functions described in Table 53 on page 261 under "MPI profiling", because those calls might interfere with the instrumentation you insert using **peekperf**. Your application should not be linked with the MPI profiling library, **libmpitrace.a** for AIX or **libmpitrace.so** for Linux.

Note: On x86 systems, the instrumentation process described here is not supported. You must relink your application, linking with the **libmpitrace** MPI profiling and trace library.

After the data collection window opens, you should select the **MPI** tab to show the possible instrumentation points in the application. The data collection window that appears is similar to Figure 71 on page 108:

© Copyright IBM Corp. 2008, 2013

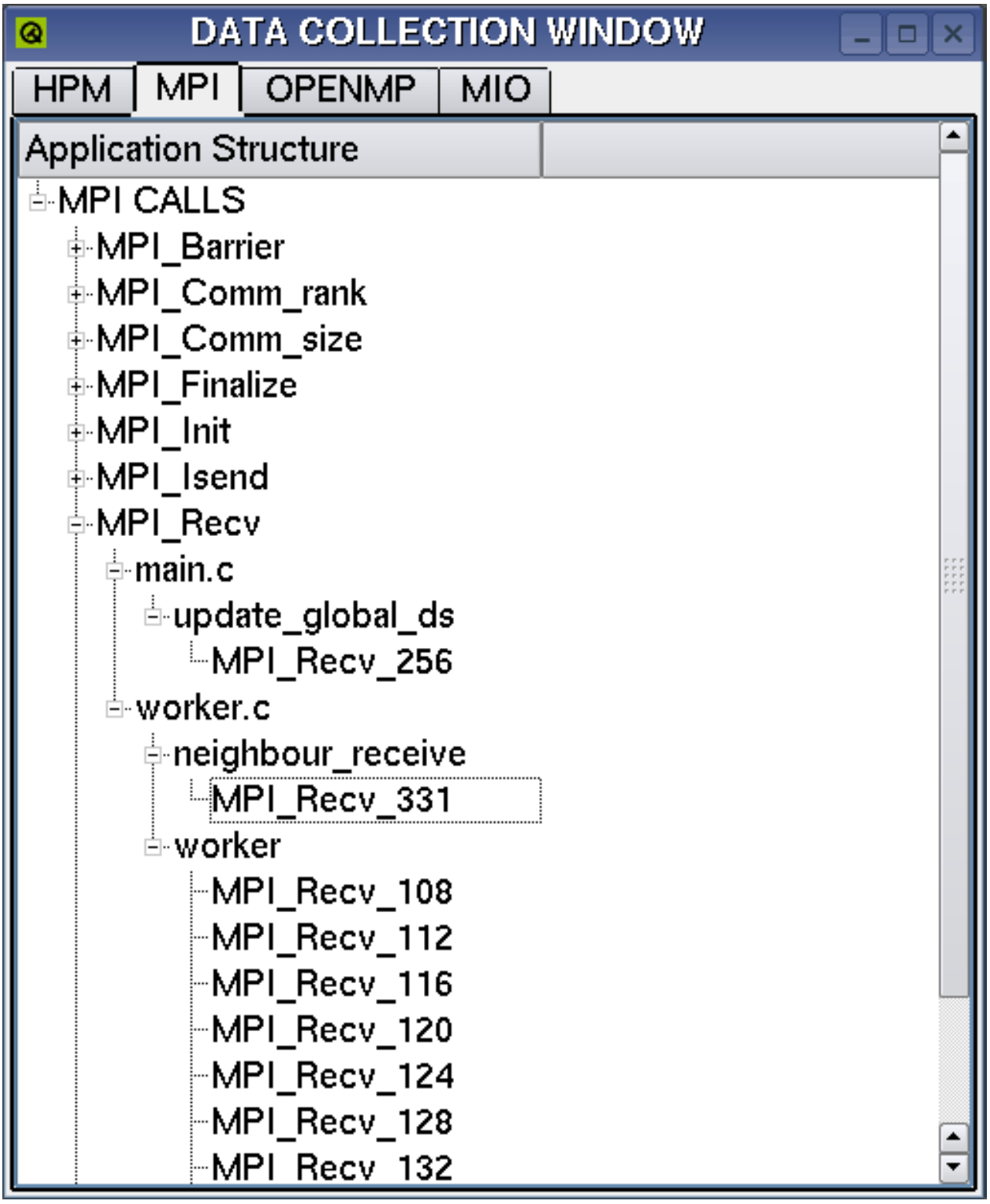


Figure 71. Peekperf MPI profiling data collection window

The data collection window is partially expanded, showing the **MPI_Recv()** calls that can be instrumented. For MPI profiling, **peekperf** labels each instrumentation location with the name of the MPI function at that line in the application and the line number where the MPI function call is located.

You can select as many instrumentation locations in your application as you like. If you select a leaf node, only that MPI function call is instrumented. If you select a node labeled with an application function name or file name, all MPI function calls of the type identified by the enclosing node are instrumented. If you select a node labeled with an MPI function name, all MPI calls for that MPI function in the application are instrumented. You can instrument all MPI function calls in the application by selecting the **MPI CALLS** node. You can deselect an MPI function call by clicking the corresponding highlighted node in the application structure tree.

You can also select MPI function calls to be instrumented by left-clicking over a starting line in the source code window and dragging to the end of the region of interest while holding the left mouse button down. After you have selected a region of code, you can select all MPI function calls in that region to be instrumented or not instrumented by right-clicking in the source code window and selecting either **Add to MPI** or **Remove from MPI** respectively. When you do this, **peekperf** updates the tree in the application structure window accordingly.

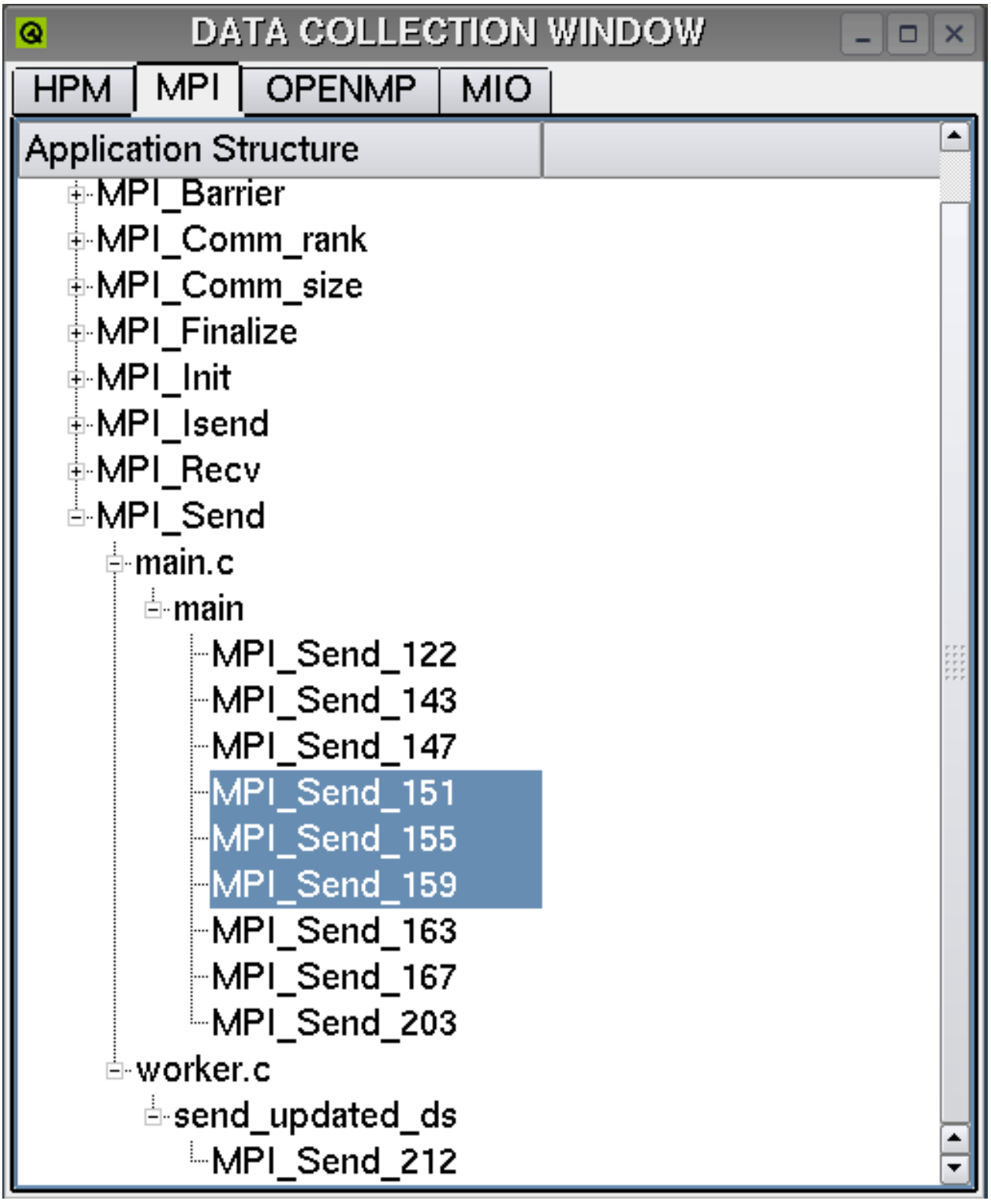


Figure 72. Peekperf MPI profiling data collection with selected instrumentation

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running your application

Before you can run your instrumented application, you must ensure that any environment variables required by the MPI profiling library or by Parallel Environment Runtime Edition are set correctly. The environment variables used by the MPI profiling library are listed in the description of "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303.

By default, if you are not invoking the application using **hpcrun**, the MPI profiling library will generate trace files only for the application tasks with the minimum, maximum, and median MPI communication time. A trace will be also be generated for task zero if task zero is not the task with minimum, maximum, or median MPI communication time. If you need trace files generated for additional tasks, set the **OUTPUT_ALL_RANKS** environment variable to **yes**. Depending on the number of tasks in your application, you might need to set the **MAX_TRACE_RANK** and **TRACE_ALL_TASKS** environment variables to **yes**. If your application executes many MPI function calls, you might need to set the value of the **MAX_TRACE_EVENTS** to a higher number than the default 30,000 MPI function calls.

If you invoke your application using **hpcrun**, then the **tracefile** will contain traces for application tasks as determined from **hpcrun** option settings. See "Collecting performance data from a subset of parallel application tasks" on page 5 for a description of the **hpcrun** command.

After you have set up the environment for your application, run it by selecting the **Run an Instrumented Application** from the **Automatic** menu and respond to the dialogs as described previously.

Viewing MPI profiling data

The **peekperf** GUI tries to open the **.viz** files after your application has completed execution, if you requested **peekperf** to do so when you ran your application. If **peekperf** does not automatically open the **.viz** files, you can open them manually by selecting **Open Performance Data** from the **File** menu and selecting **.viz** files from the file selector dialog. Figure 73 on page 112 shows the application structure tree in the data visualization window partially expanded.

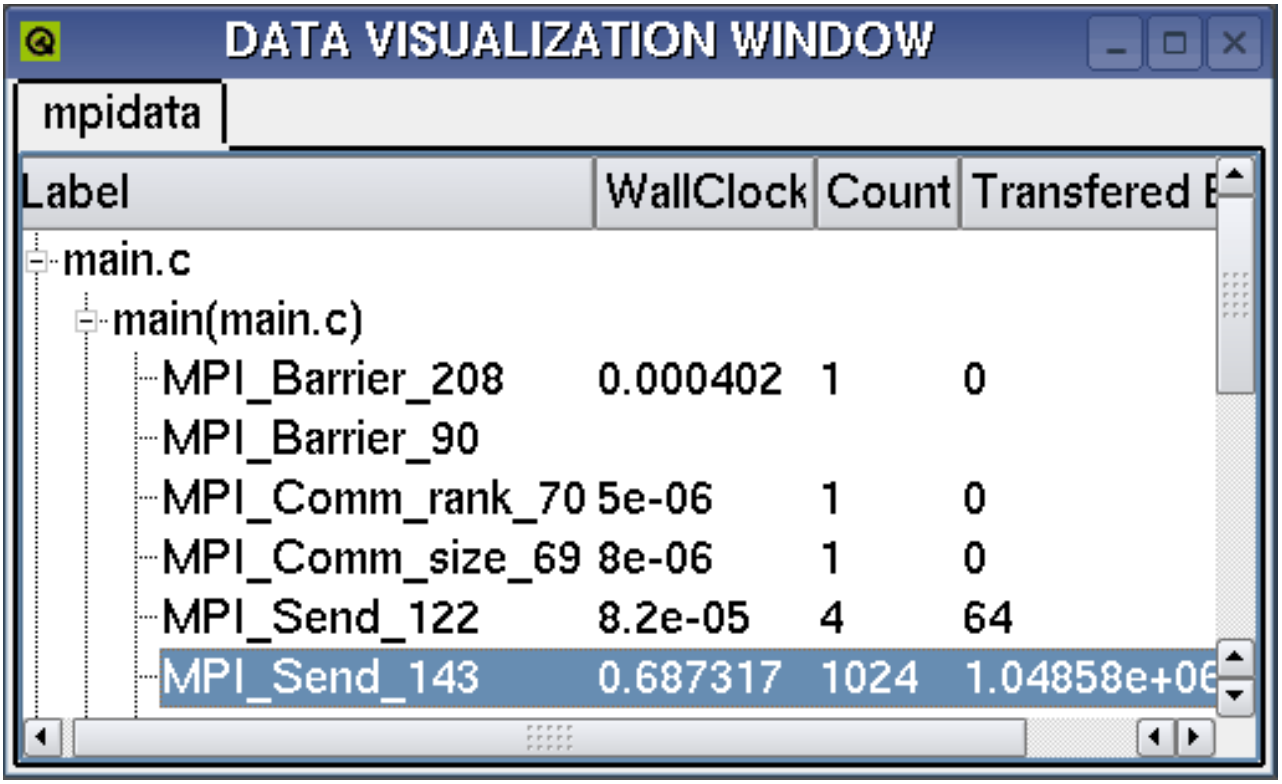


Figure 73. Peekperf MPI profiling data visualization window

The data visualization window shows the number of times each function call was executed, the total time spent executing that function call, and the amount of data transferred by that function call. You can see a more detailed view of the data you obtained by right-clicking over a leaf node in the data visualization window, then selecting **Show Metric Browser Window** from the pop-up menu that appears, which opens a metrics browser window, or by right-clicking in white space in the data visualization window and selecting **Show as Table** from the pop-up menu, which opens a table view of the performance data.

The data visualization window shows performance data from a single task at one time. If you want to see performance data from a different task, right-click in the white space in the window and select **Show Other Rank** from the pop-up menu. A pop-up dialog appears containing a dropdown list that allows you to select the task indices of other tasks that can be selected.

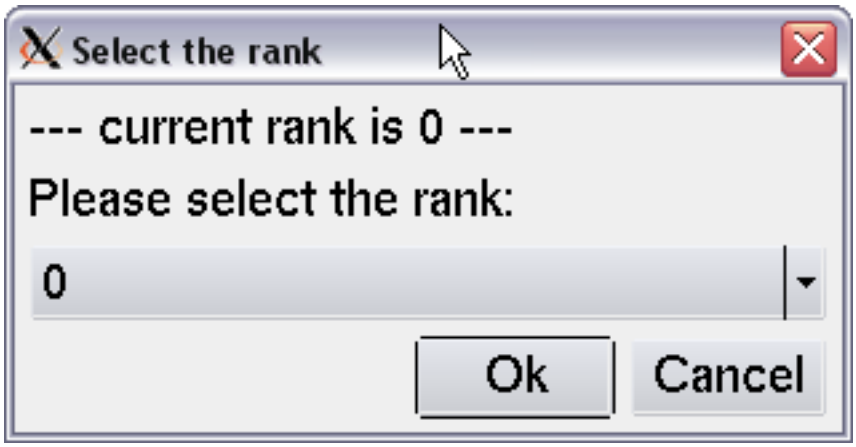


Figure 74. Peekperf MPI task rank selection dialog

Select the desired task and click **OK** to view a different task's performance data.

You can view a trace of MPI activity in your application by right-clicking in white space in the data visualization window and selecting **View Tracer** from the pop-up menu. When you do this, Figure 75 and Figure 76 on page 114 are displayed.

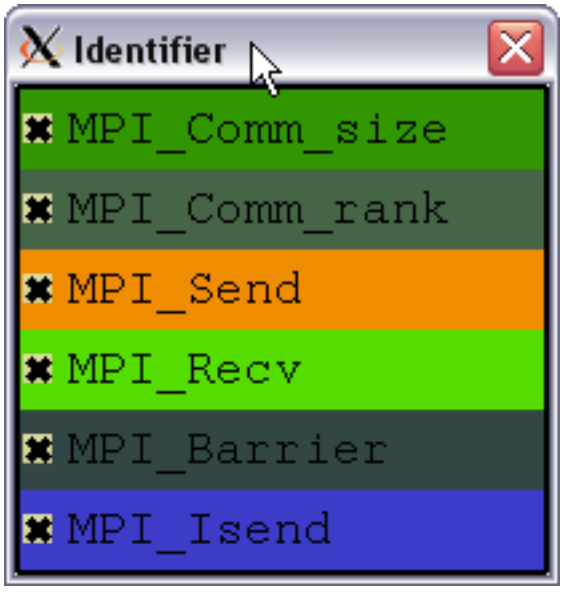


Figure 75. Peekperf MPI trace viewer identifier dialog

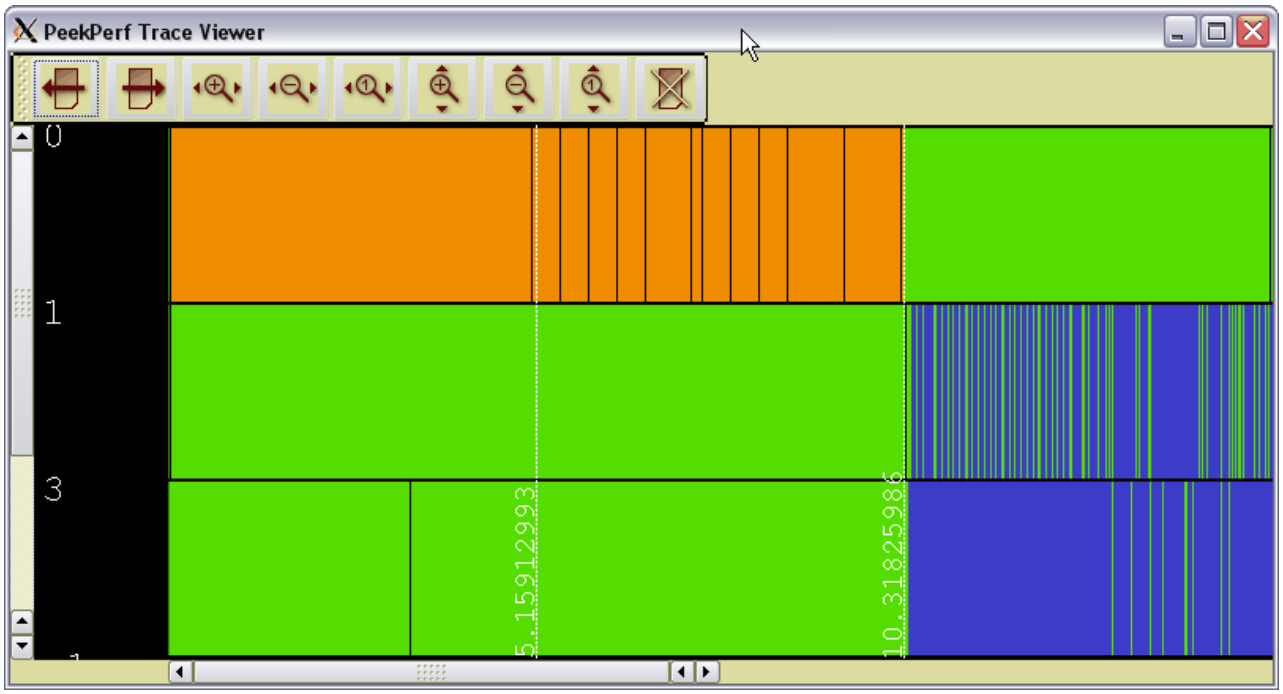


Figure 76. Peekperf trace viewer

You can use Figure 75 on page 113, labeled **Identifier**, to select the types of MPI function calls that are displayed in the **PeekPerf Trace Viewer** window. Initially, all MPI function calls that were executed are displayed. You can hide or display trace events for specific MPI functions by clicking on the MPI function labels in the identifier window. Trace events are displayed for MPI functions that have their labels checked while trace events are hidden for MPI functions with unchecked labels.

Figure 76 is the MPI trace viewer window. This window displays a timeline-based view of your application's execution, where the Y axis is the application task rank and the X axis is elapsed time. Each MPI function call is represented by a block drawn in the color matching the MPI function label in the identifier window.

You can use the icons at the top of the **PeekPerf Trace Viewer** window to navigate through the trace. From left to right, the icons perform the following tasks:

- Scroll the trace data left.
- Scroll the trace data right.
- Zoom the trace display in to display a more detailed level of the X axis.
- Zoom the trace display out to display a less detailed level of the X axis.
- Reset the X axis to its initial, non-zoomed, view.
- Enlarge the trace display along the Y axis.
- Shrink the trace display along the Y axis.
- Reset the Y axis to its initial, non-zoomed view.
- Close the trace display windows.

You can also scroll the trace data horizontally and vertically using the scrollbars at the left and bottom of the trace window. You can zoom in to display a more detailed view of the trace by picking a starting point in the trace, left-clicking, and

dragging the mouse to the end of the region of interest while holding the left mouse button down. When you release the mouse button, **peekperf** zooms in to display the selected region.

Figure 77 shows a region of the trace that has been zoomed in to where there are alternating **MPI_Send()** and **MPI_Recv()** calls.

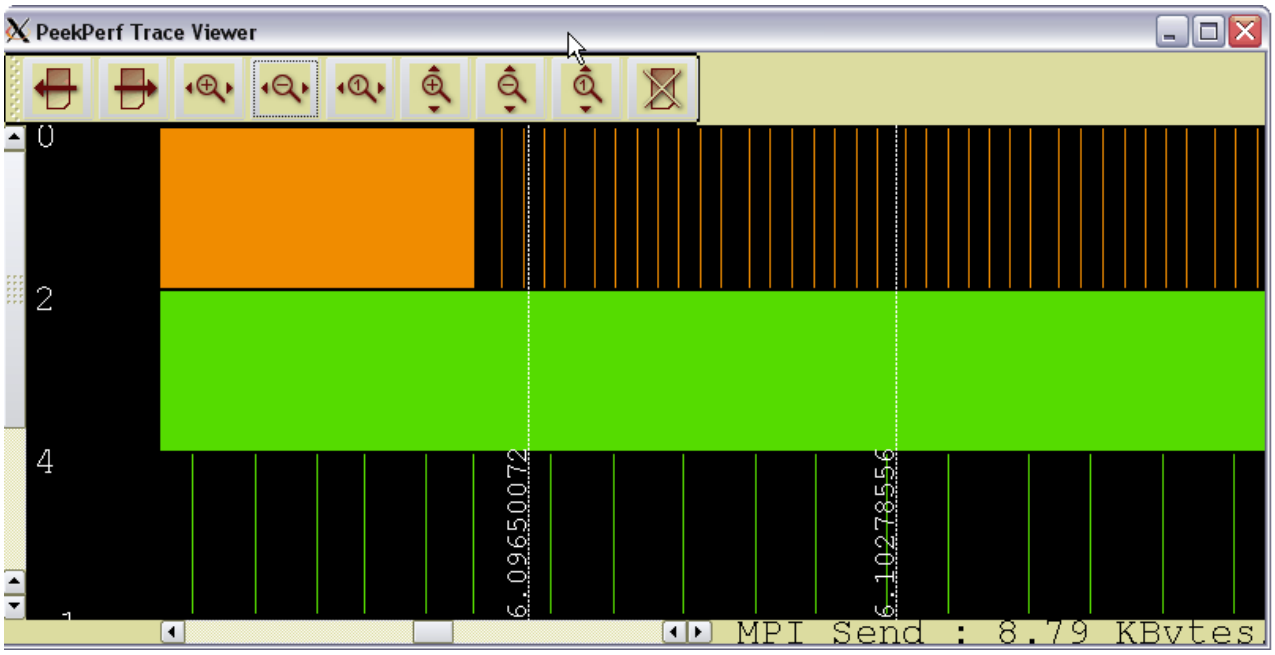


Figure 77. Peekperf trace viewer detailed view

If you left-click over a rectangle for a specific MPI call, **peekperf** displays the name of the MPI function and the data rate for a communication function call in the lower right corner of the trace window. **Peekperf** also positions the source code window to the specific MPI function call in the source code and highlights that MPI function call.

If you right-click over a rectangle for a specific MPI function call, **peekperf** will display a pop-up dialog box that displays additional detail for the MPI function call, as long as the right mouse button is pressed.

Chapter 11. Using OpenMP profiling in peekperf

The topics that follow will provide information about using OpenMP profiling in **peekperf**.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, you should run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

Note: For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

You start **peekperf** and load the application binary as described in Chapter 8, "Using the peekperf GUI," on page 83. There are no additional compiler or link options that are required to use OpenMP profiling.

Instrumenting your application

After the data collection window opens, select the **OPENMP** tab in that window. This tab shows you the instrumentation locations, similar to Figure 78 on page 118 in which the application structure tree has been fully expanded.

© Copyright IBM Corp. 2008, 2013

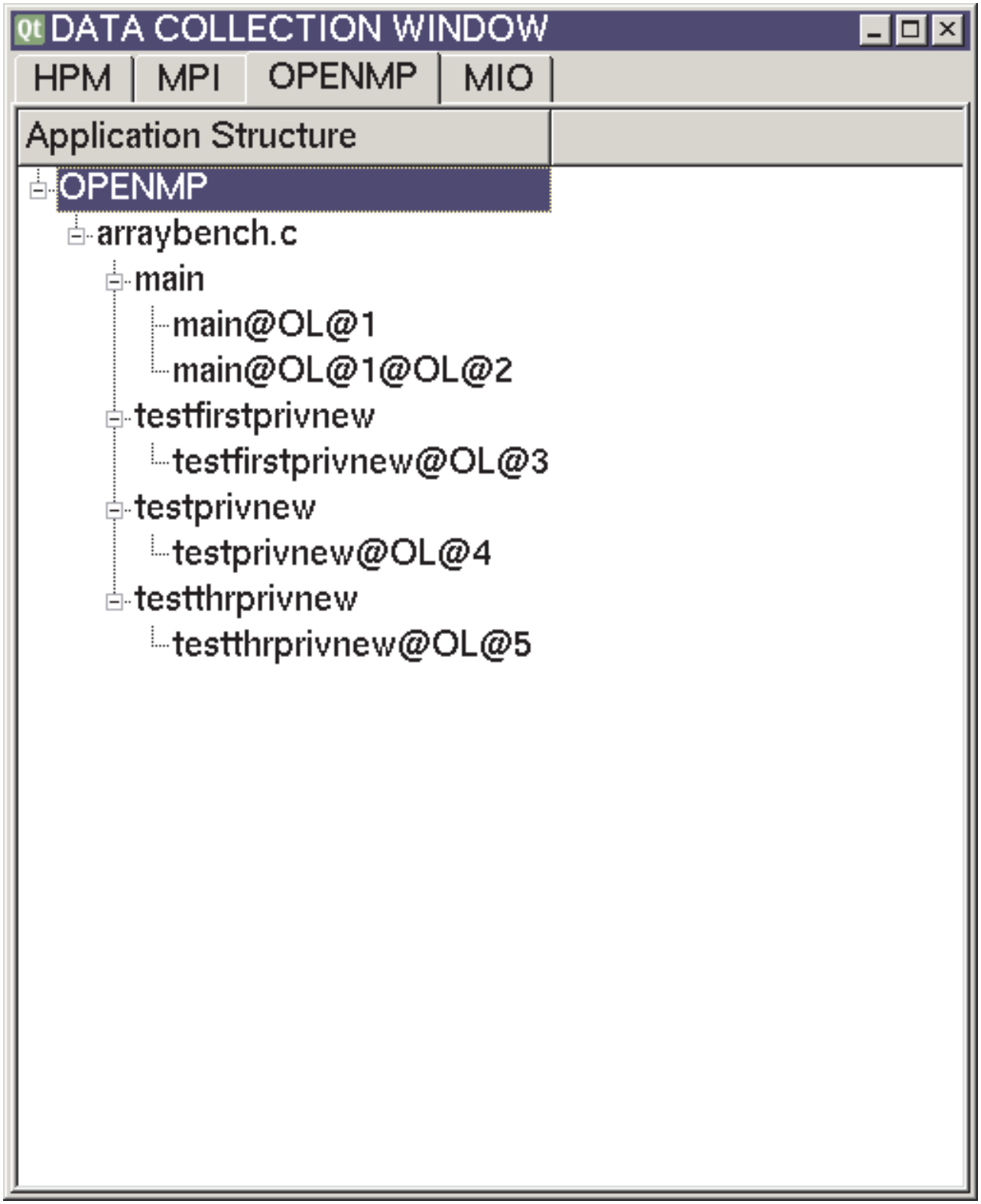


Figure 78. Peekperf OpenMP data collection window

This window shows you the locations in your application that can be instrumented for OpenMP profiling. Leaf nodes in this tree represent individual OpenMP regions in your application. Leaf nodes are labeled with the function name generated by the compiler for that OpenMP region. You can select instrumentation at any level

in the tree. If you select a leaf node, only that location will be instrumented. If you select a non-leaf node, all leaf nodes that are children of the selected node will be instrumented.

Note: The IBM HPC Toolkit only supports OpenMP profiling instrumentation for OpenMP regions that are not nested within other OpenMP constructs at run time. If you set up instrumentation so that nested parallel constructs are instrumented, results will be unpredictable.

You can control the level of instrumentation for each class of OpenMP constructs by setting the **POMP_LOOP**, **POMP_PARALLEL**, and **POMP_USER** environment variables before you instrument the application. Refer to "hpctInst - Instrument applications to obtain performance data" on page 248 for the description of each environment variable's settings. You set these environment variables by selecting the **Set the Environment Variable** option from the **Automatic** menu, and filling in the name of each environment variable and its value in the pop-up dialog that appears.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running your application

You can run your instrumented application by selecting the **Run an Instrumented Application** option from the **Automatic** menu and responding to the pop-up dialogs that were described in Chapter 8, "Using the peekperf GUI," on page 83. There are no special steps that must be followed to run an application with OpenMP instrumentation.

Viewing OpenMP profiling performance data

After your application completes, the **peekperf** GUI displays your performance measurements in the data visualization window, if you requested **peekperf** to open the **.viz** files on application completion. The main GUI window looks similar to Figure 79 on page 120:

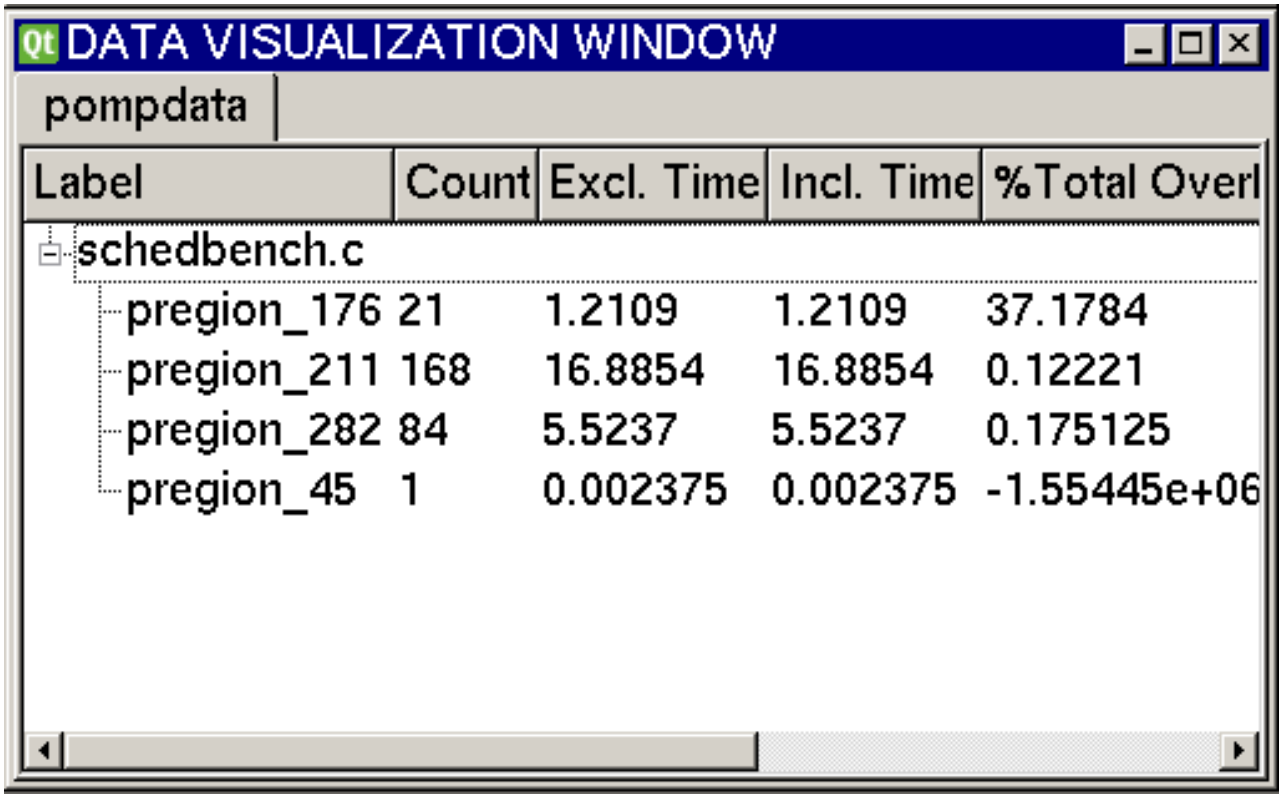


Figure 79. Peekperf OpenMP data visualization window

The data is displayed in a tree format in which source files are the major nodes in the tree and the individual instrumentation points are the leaf nodes. Individual leaf nodes are labeled with the type of OpenMP instrumentation and the starting line number of the instrumented OpenMP construct.

You can view detailed data for a leaf node by right-clicking over a leaf node. A metric browser window contains data for each process and thread that executed the OpenMP construct. You can view all of your performance measurements in a tabular form by selecting the **Show as Table** option from the pop-up menu that appears when you right-click on white space within the data visualization window.

The data displayed in the data visualization window is a summary view of the performance data, displaying the maximum value from all threads for the following statistics:

Count
The number of times the event was executed.

Exclusive time (Excl. Time)
The total time (does not include time inside of other events).

Inclusive time (Incl. Time)
The wall clock time for the event (including other events).

Percentage of total overhead (% Overhead)
100 * (thread time – computation time) / thread time.

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i)).

Average thread time (Avg. Comp Time)
SUM(comp time(i)) / Number of threads.

The Metrics Browser window shows the following statistics for each OpenMP thread:

Task (Task)
The MPI Task ID.

Thread (thread)
The OpenMP thread ID.

Time in master: (Time in Master)
The time in the master thread (wall clock time).

Thread time: (TT: Thread Time)
The wall clock time for the execution of each thread.

Computation time: (CT: Comp. Time)
The time per thread in the body of the OpenMP construct.

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i)).

Total overhead: (TO: TT – CT)
The thread time – comp time.

Percentage of the total overhead due to barrier (%TO (Barrier))
100 * barrier time / incl time. The barrier time is measured as the time between the end of the thread execution and end of the last thread.

Percentage of the total overhead due to the runtime library (%TO (RTL))
100 * (RTL time – barrier time) / incl time.

The performance data table window shows the following statistics for each process and thread in the application:

Count
The number of times the event was executed.

Exclusive time (Excl. Time)
The total time (Does not include time inside of other events).

Inclusive time (Incl. Time)
The wall clock time for the event (including other events).

Percentage of total overhead (% Overhead)
100 * (thread time – computation time) / thread time.

Thread time: (TT: Thread Time)
The wall clock time for the execution of each thread.

Computation time: (CT: Comp. Time)
The time per thread in the body of the OpenMP construct.

Percentage of imbalance (% imbalance)
100 * (comp time – MIN(comp time(i))) / MIN(comp time(i)).

Average thread time (Avg. Comp Time)
SUM(comp time(i)) / Number of threads.

Total overhead: (TO: TT – CT)
The thread time – comp time.

Percentage of the total overhead due to barrier (%TO (Barrier))
100 * barrier time / incl time. The barrier time is measured as the time between the end of the thread execution and end of the last thread.

Percentage of the total overhead due to the runtime library (%TO (RTL))
100 * (RTL time – barrier time) / incl time.

Chapter 12. Using I/O Profiling in peekperf

The topics that follow provide information about using I/O Profiling in **peekperf**.

Preparing your application

Note: You must not compile your application with the **-pg** flag.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **peekperf**. In order to set these environment variables, run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

You start **peekperf** and load the application binary as described in Chapter 8, "Using the peekperf GUI," on page 83. If you are using the instrumentation process described in the next topic, there are no additional compiler or linking options that are required to use I/O profiling.

Instrumenting your application

Note: On x86 systems, the instrumentation process described here is not supported. You must link your application with the I/O profiling and trace library (**libhpctkio**) and set **TKIO_ALTLIB** to point to the correct version of **get_hpcmio_ptrs.so**.

After the data collection window opens, select the **MIO** tab. This tab shows the possible instrumentation, similar to Figure 80 on page 124:

© Copyright IBM Corp. 2008, 2013

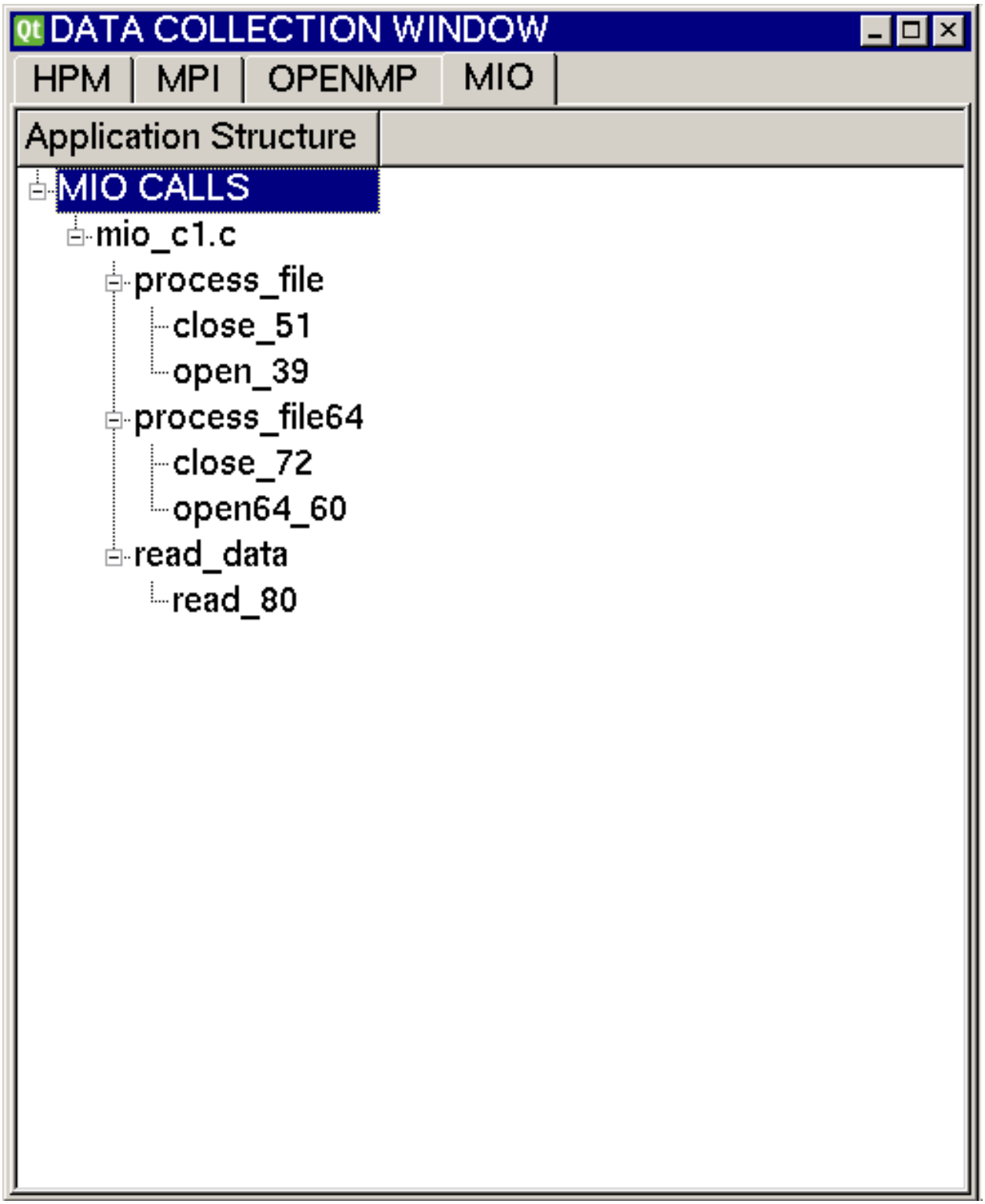


Figure 80. Peekperf I/O profiling data collection window

Figure 80 shows the application structure tree fully expanded. The leaf nodes are labeled with the name of the system call at that location and the line number in the source file. If you select leaf nodes, instrumentation is placed only at those

specific instrumentation points. If you select a non-leaf node, instrumentation is placed at all leaf nodes that are child nodes of the selected non-leaf node.

In order for I/O profiling to work correctly, you must instrument at least the **open** and **close** system calls that open and close any file for which you want to obtain performance measurements.

After you have selected the set of instrumentation that you want, you instrument the application by selecting **Generate an Instrumented Application** from the **Automatic** menu.

Running your application

I/O profiling works by intercepting I/O system calls for any files that you want to obtain performance measurements for. In order to obtain the performance measurement data, the IBM HPC Toolkit uses the **MIO_FILES** environment variable. See "Setting I/O profiling environment variables" on page 219 for a description of the MIO-related environment variables and their settings.

As a minimum, you must set the **MIO_FILES** environment variable to:

- Specify a file-name-matching pattern.
- Use the MIO trace module.
- Specify the **xml** and **events** options to the trace module.

Setting **MIO_FILES** to:

```
*[trace/xml/events={mioevents}]
```

sets up MIO to collect the performance data that **peekperf** requires. Setting **MIO_FILES** to this value:

- Instructs MIO to apply the options to all files opened by the application
- Generates the performance data in the XML file format required by **peekperf**
- Specifies that the I/O trace file will be written to **hpct.mio.mioevents.iot**. For more information, see Appendix A, "Performance data file naming," on page 309.

After you have set the appropriate environment variables, you can run your instrumented application by selecting **Run an Instrumented Application** from the **Automatic** menu and responding to the pop-up dialogs described in Chapter 8, "Using the peekperf GUI," on page 83.

Viewing I/O profiling data

After your application completes, **peekperf** tries to display the I/O profiling data that was collected when the application was run (if you requested the data be displayed). **Peekperf** displays the data in a tree format, in which the top-level node is the file that the application read or wrote and the leaf nodes are the I/O function calls your application issued for that file. Figure 81 on page 126 shows the data visualization window with this tree fully expanded.

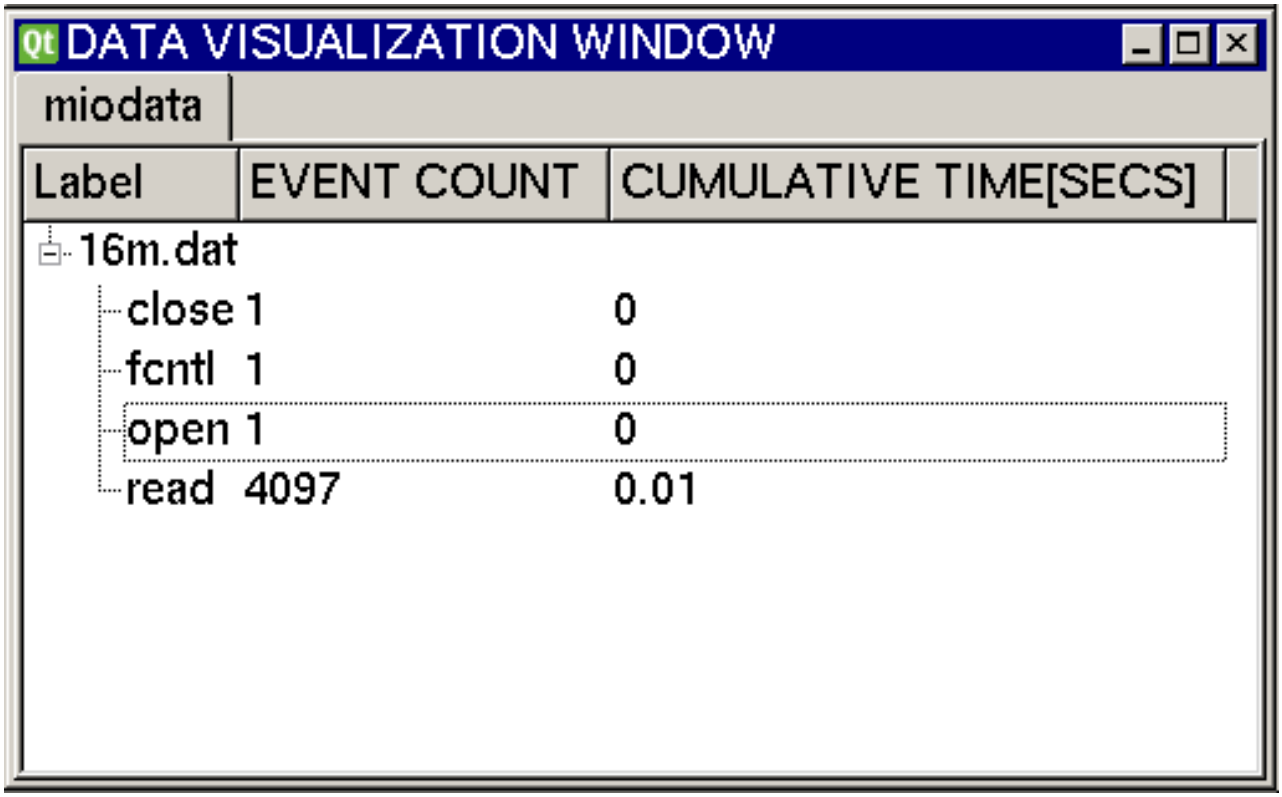


Figure 81. Peekperf I/O profiling data visualization window

Each row shows the time spent in an I/O function call and the number of times that function call was executed.

You can view detailed data for a leaf node by right-clicking over it. A metric browser window contains data for each process that executed that I/O function. You can view all of your performance measurements in a tabular form by selecting the **Show as Table** option from the pop-up menu that appears when you right-click on white space within the data visualization window.

You can view a plot of your I/O measurements by right-clicking in white space in the data visualization window and then selecting **View Tracer** from the pop-up menu that appears. **Peekperf** displays a window that looks similar to Figure 82 on page 127:

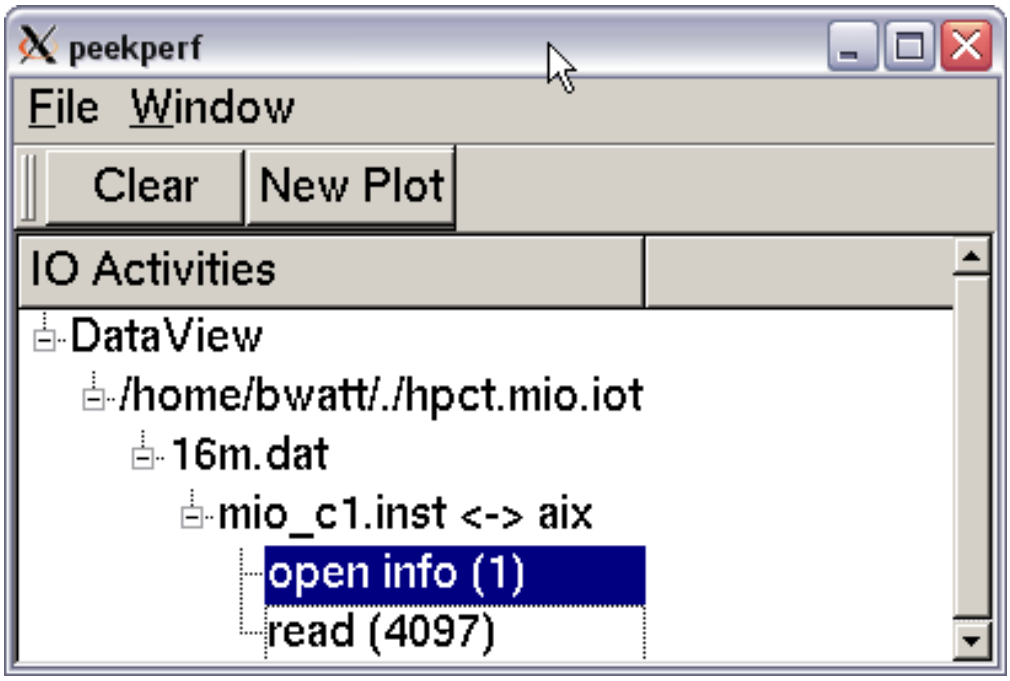


Figure 82. Peekperf I/O profiling data selection window

The window contains a tree view of the MIO performance data files. The top-level nodes represent individual performance data files. The next level nodes represent individual files that the application accessed. The next level nodes represent the application program, and the leaf nodes represent the I/O function calls executed in the application. You can select nodes at any level of the tree to include the data from those nodes in the plot window.

The **File** menu has several options:

`Read Event`
Opens a file selector dialog in which you can select a file containing I/O profiling data. **Peekperf** adds the selected I/O profiling data file to the IO Activities tree.

`New Plot`
Opens a new I/O profiling graph, displaying the I/O profiling data from the selected nodes in the IO Activities tree.

`Add to Current Plot`
Adds the I/O profiling data from the selected nodes in the IO Activities tree to the data plotted in the currently active I/O profiling graph.

`Remove from Current Plot`
Removes the I/O profiling data for the selected nodes in the IO Activities tree from the currently active I/O profiling graph.

`Edit Table`
Opens a table view of the I/O profiling data for the selected nodes in the IO Activities tree.

The **Window** menu displays a list of open I/O profiling graph and data view tables. You can select a window that you want to view from this menu.

The **Clear** button just below the menu bar clears the selection state for all nodes in the IO Activities tree. The **New Plot** button just below the menu bar opens a new I/O profiling graph using data for the nodes selected in the IO Activities tree. At least one node must be selected in the IO Activities tree in order to display a new plot.

When you select a node from the IO Activities tree and then select **New Plot** from the file menu or click the **New Plot** button, an I/O profiling graph window, similar to Figure 83, showing data from reading a file sequentially, opens.

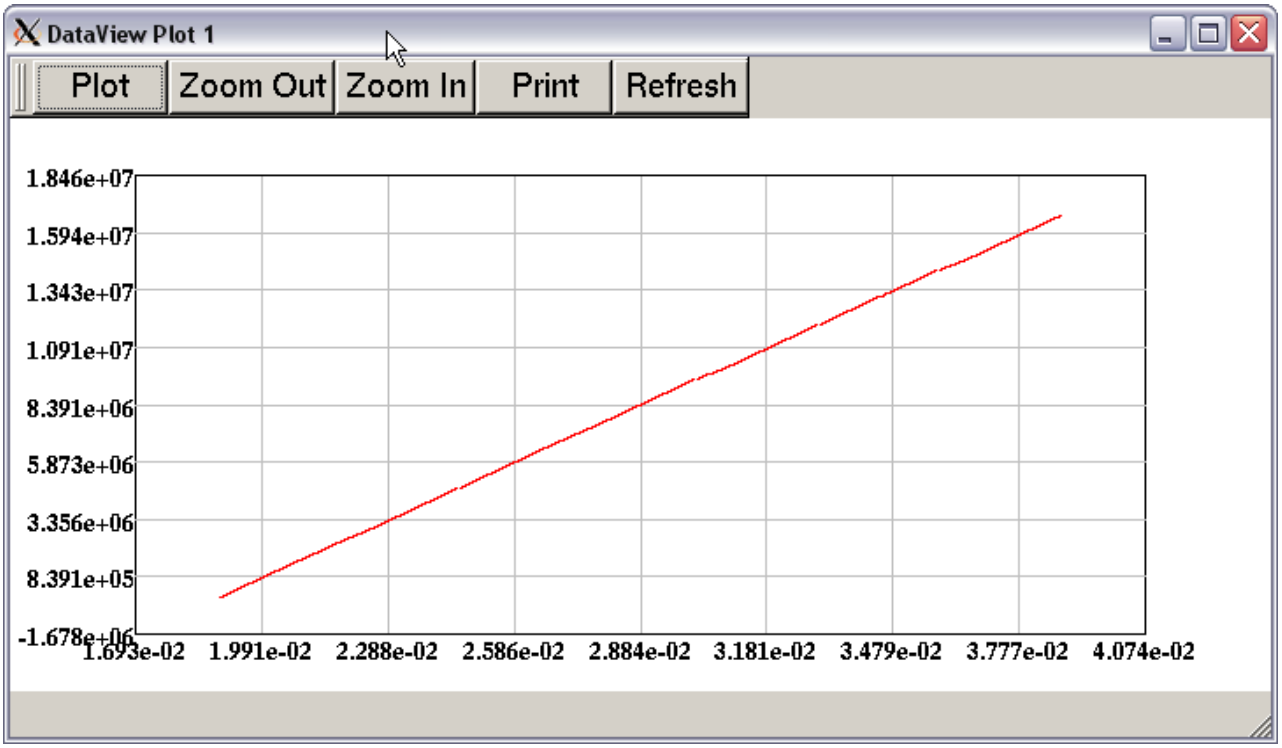


Figure 83. Peekperf I/O profiling data plot window

When the graph is initially displayed, the Y axis represents the file position, in bytes. The X axis of the graph always represents time in seconds.

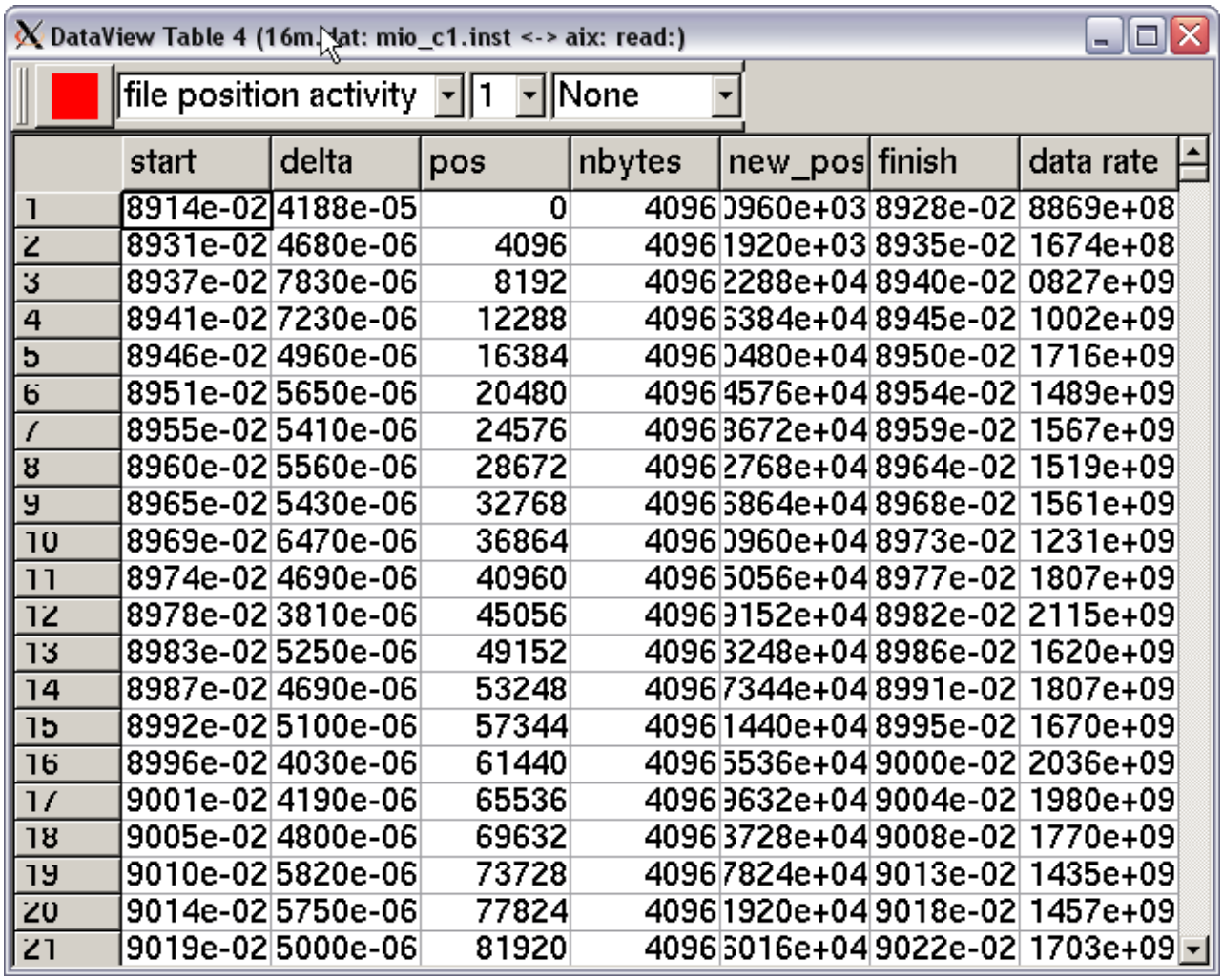
You can zoom in to an area of interest in the graph by left-clicking at one corner of the desired area and dragging the mouse while holding the left button to draw a box around the area of interest and releasing the left mouse button. When you release the left mouse button, **peekperf** redraws the graph, showing the area of interest. You can then zoom in and out of the graph by clicking the **Zoom In** and **Zoom Out** buttons at the top of the graph window. As you drag the mouse, **peekperf** displays the X and Y coordinates of the lower left corner of the box and the upper right corner of the box and the slope of the line between those two corners as text in the status bar area at the bottom of the window.

You can determine the I/O data transfer rate at any area in the plot by right-clicking over the desired starting point in the plot and holding down the right mouse button, while tracing over the section of the plot of interest. The coordinates of the starting and ending points of the selection region and the data transfer rate (slope) are displayed in the status area at the bottom of the plot window.

You can print the current plot, either to a printer or to a Portable Document Format (PDF) file, by clicking the **Print** button at the top of the plot window. A printer setup dialog appears, where you can specify how the file should be printed.

You can select the printer from the list of available printers in the drop down box at the top of the dialog. If you select the **Print to file** checkbox, you can specify the path to the file in which the plot will be written as a PDF file.

You can view the I/O profiling data in tabular form and modify the characteristics of the current plot by selecting **Edit Table** from the **File** menu in the window that appeared when you selected the **View Tracer** option from the pop-up menu in the data visualization window. A window appears, similar to Figure 84 showing the I/O profiling data.



	start	delta	pos	nbytes	new_pos	finish	data rate
1	8914e-02	4188e-05	0	4096	0960e+03	8928e-02	8869e+08
2	8931e-02	4680e-06	4096	4096	1920e+03	8935e-02	1674e+08
3	8937e-02	7830e-06	8192	4096	2288e+04	8940e-02	0827e+09
4	8941e-02	7230e-06	12288	4096	6384e+04	8945e-02	1002e+09
5	8946e-02	4960e-06	16384	4096	0480e+04	8950e-02	1716e+09
6	8951e-02	5650e-06	20480	4096	4576e+04	8954e-02	1489e+09
7	8955e-02	5410e-06	24576	4096	8672e+04	8959e-02	1567e+09
8	8960e-02	5560e-06	28672	4096	2768e+04	8964e-02	1519e+09
9	8965e-02	5430e-06	32768	4096	6864e+04	8968e-02	1561e+09
10	8969e-02	6470e-06	36864	4096	0960e+04	8973e-02	1231e+09
11	8974e-02	4690e-06	40960	4096	5056e+04	8977e-02	1807e+09
12	8978e-02	3810e-06	45056	4096	9152e+04	8982e-02	2115e+09
13	8983e-02	5250e-06	49152	4096	3248e+04	8986e-02	1620e+09
14	8987e-02	4690e-06	53248	4096	7344e+04	8991e-02	1807e+09
15	8992e-02	5100e-06	57344	4096	1440e+04	8995e-02	1670e+09
16	8996e-02	4030e-06	61440	4096	5536e+04	9000e-02	2036e+09
17	9001e-02	4190e-06	65536	4096	9632e+04	9004e-02	1980e+09
18	9005e-02	4800e-06	69632	4096	3728e+04	9008e-02	1770e+09
19	9010e-02	5820e-06	73728	4096	7824e+04	9013e-02	1435e+09
20	9014e-02	5750e-06	77824	4096	1920e+04	9018e-02	1457e+09
21	9019e-02	5000e-06	81920	4096	6016e+04	9022e-02	1703e+09

Figure 84. Peekperf I/O profiling data table window

There are four widgets at the top of the table window that you can use to modify the characteristics of the current plot. You can change the values in these widgets as desired, then click the **Plot** button in the current plot window to apply the selections you made to the current plot.

The colored square at the upper left specifies the color to use when drawing the plot. If you click this square, a color selector dialog appears, which allows you to select the color you want to be used in drawing the plot.

The second widget from the left, labeled **file position activity**, selects the metric to be used for the Y and X axis of the plot and also affects the format of the plot. If you select **file position activity**, the Y axis represents **file position** and the X axis represents **time**. If you select **data delivery rate**, the Y axis represents the **data transfer rate** and the X axis represents **time**. If you select **rate vs. pos,** the Y axis represents **data transfer rate** and the X axis represents the **start position** in the file

The third widget from the left specifies the pixel width for the graph that is drawn when the file position metric has been selected from the second widget from the left.

The right-most widget specifies the metric that will have its numeric value displayed next to each data point. You can select any column displayed in the table, or **none** to plot each point with no accompanying data value.

Part 5. The X Windows performance profiler (Xprof)

The topics in this part provide information about using the X Windows performance profiler (Xprof).

© Copyright IBM Corp. 2008, 2013

Chapter 13. Using the X Windows performance profiler (Xprof)

The X Windows Performance Profiler (Xprof) tool helps you analyze your parallel or serial application's performance. It uses procedure-profiling information to construct a graphical display of the functions within your application. Xprof provides quick access to the profiled data, which lets you identify the functions that are the most CPU-intensive. The graphical user interface (GUI) also lets you manipulate the display in order to focus on the application's critical areas.

The following Xprof topics are covered:

- "Before you begin using Xprof"
- "Starting the Xprof GUI" on page 135
- "Customizing Xprof resources" on page 190

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

Before you begin using Xprof

Xprof lets you profile both serial and parallel applications. Serial applications generate a single profile data file, while a parallel application produces multiple profile data files. You can use Xprof to analyze the resulting profiling information.

Xprof provides a set of resource variables that let you customize some of the features of the Xprof window and reports.

Requirements and limitations

To use Xprof, your application must be compiled with the **-pg** flag. For more information, see "Compiling applications to be profiled" on page 134.

On AIX, you can generate profile data files for a subset of application tasks instead of generating profile data files for all application tasks. This can be helpful when running applications with large numbers of tasks where the number of performance data files created impacts system performance or the amount of generated data is too large to analyze practically.

You can do this by compiling your application with the **-hpcprof** compilation scripts provided with PE Runtime Edition, then by invoking the **hpcrun** command to run your application. If you compile your application with the **-hpcprof** option, then you must run the application using the **hpcrun** command.

See "Collecting performance data from a subset of parallel application tasks" on page 5 for a description of the **hpcrun** command.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke Xprof. In order to set these environment variables, you should run the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar

© Copyright IBM Corp. 2008, 2013

shell command, you should invoke the **env_sh** script as **. env_sh**. If you are using **csh**, you should invoke the **env_csh** script as **source env_csh**.

Note: Beginning with AIX 5.3, you can generate a new format of the thread-level profiling **gmon.out** files. Xprof does not support this new format, so you must set the **GPROF** environment variable to ensure that you produce the previous format of the **gmon.out** files. For more information, see the **gprof** man page.

Like the **gprof** command, Xprof lets you analyze CPU (busy) usage only. It does not provide other kinds of information, such as CPU idle, I/O, or communication information.

If you compile your application on one processor, and analyze it on another, you must first make sure that both processors have similar library configurations, at least for the system libraries used by the application. For example, if you run a FORTRAN application on a server, then try to analyze the profiled data on a workstation, the levels of FORTRAN run-time libraries must match and must be placed in a location on the workstation that Xprof recognizes. Otherwise, Xprof produces unpredictable results.

Because Xprof collects data by sampling, functions that run for a short amount of time might not show any CPU use. Xprof does not give you information about the specific threads in a multithreaded program.

Xprof presents the data as a summary of the activities of all the threads.

Comparing Xprof and the gprof command

With Xprof, you can produce the same tabular reports that you might be accustomed to seeing with the **gprof** command. As with **gprof**, you can generate the Flat Profile, Call Graph Profile, and Function Index reports.

Unlike **gprof**, Xprof provides a GUI that you can use to profile your application. Xprof generates a graphical display of your application's performance, as opposed to a text-based report. Xprof also lets you profile your application at the source statement level.

From the Xprof GUI, you can use all of the same command line flags as **gprof**, as well as some additional flags that are unique to Xprof.

Compiling applications to be profiled

To use Xprof, you must compile and link your application with the **-pg** flag of the compiler command. This applies regardless of whether you are compiling a serial or parallel application. You can compile and link your application all at once, or perform the compile and link operations separately. The following is an example of how you would compile and link all at once:

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

The following is an example of how you would first compile your application and then link it. To compile, do the following:

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

To link, do the following:

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

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

The **-pg** flag compiles and links the application so that when you run it, the CPU usage data is written to one or more output files, depending on whether the program, is a serial or MPI program and whether the **filename:prefix** specification is included in the **GPROF** environment variable setting. For information about how files are named, see Appendix A, "Performance data file naming," on page 309.

Note: The **-pg** flag is not a combination of the **-p** and the **-g** compiler flags.

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

The Xprof GUI lets you view included functions. Your application must also be compiled with the **-g** flag in order for Xprof to display the included functions.

In addition to the **-pg** flag, the **-g** flag is also required for source-statement profiling.

Starting the Xprof GUI

To start Xprof, enter the **Xprof** command on the command line. You must also specify the binary executable file, one or more profile data files, and optionally, one or more flags, which you can do in one of two ways. You can either specify the files and flags on the command line along with the **Xprof** command, or you can enter the **Xprof** command alone, then specify the files and flags from within the GUI.

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

To start Xprof and specify the binary executable file, one or more profile data files, and one or more flags, type:

Xprof a.out gmon.out... [*flag...*]

where:

a.out
Is the binary executable file.

gmon.out...
Is the name of your profile data file (or files).

flag...
Is one or more of the flags listed in the Table 14 on page 136.

Specifying Xprof flags from the command line

You can specify many of the same command-line flags with the **Xprof** command that you do with **gprof**, as well as one additional flag (**-disp_max**), which is specific to **Xprof**. The command-line flags let you control the way Xprof displays the profiled output.

You can specify the flags in Table 14 from the command line or from the Xprof GUI (see "Specifying Xprof flags from the GUI" on page 145 for more information).

Table 14. Xprof command line flags

Use this flag:	To:	For example:
-a	Add alternative paths to search for source code and library files, or changes the current path search order. When using this flag, you can use the "at" symbol (@) to represent the default file path, in order to specify that other paths be searched before the default path.	To set an alternative file search path so that **Xprof** searches **pathA**, the default path, then **pathB**, enter: `Xprof -a pathA:@:pathB`
-b	Suppress the printing of the field descriptions for the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports when they are written to a file with the **Save As** option of the **File** menu.	Enter: `Xprof -b a.out gmon.out`
-c	Load the specified configuration file. If this flag is used on the command line, the configuration file name specified with it will appear in the **Configuration File (-c):** text field in the **Load Files Dialog** and in the **Selection** field of the **Load Configuration File Dialog**. When the **-c** flag and the **-disp_max** flag are specified on the command line, the **-disp_max** flag is ignored, but the value that was specified with it will appear in the **Initial Display (-disp_max):** field in the **Load Files Dialog** the next time this dialog is opened.	To load the configuration file, **myfile.cfg**, enter: `Xprof a.out gmon.out -c myfile.cfg`
-disp_max	Set the number of function boxes that **Xprof** initially displays in the function call tree. The value supplied with this flag can be any integer between **0** and **5000**. **Xprof** displays the function boxes for the most CPU-intensive functions through the number you specify. For example, if you specify 50, **Xprof** displays the function boxes for the 50 functions in your program with the highest CPU usage. After this, you can change the number of function boxes that are displayed using the **Filter** menu options. This flag has no effect on the content of any of the **Xprof** reports.	To display the function boxes for the 50 most CPU-intensive functions in the function call tree, enter: `Xprof -disp_max 50 a.out gmon.out`

Table 14. Xprof command line flags (continued)

Use this flag:	To:	For example:
-e	De-emphasize the general appearance of the function box for the specified function in the function call tree, and limits the number of entries for this function in the **Call Graph Profile** report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function box for the specified function is made unavailable. The box size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for a specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.	To de-emphasize the appearance of the function boxes for **foo** and **bar** and their qualifying descendants in the function call tree, and limit their entries in the **Call Graph Profile** report, enter: `Xprof -e foo -e bar a.out gmon.out`

Table 14. Xprof command line flags (continued)

Use this flag:	To:	For example:
-E	Change the general appearance and label information of the function box for the specified function in the function call tree. This flag also limits the number of entries for this function in the **Call Graph Profile** report, and changes the CPU data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call tree, the function box for the specified function is made unavailable, and the box size and shape also changes so that it appears as a square of the smallest permitted size. In addition, the CPU time shown in the function box label, appears as 0. The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This flag also causes the CPU time spent by the specified function to be deducted from the CPU total on the left in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the self and descendants columns for this entry is set to **0**. In addition, the amount of time that was in the descendant's column for the specified function is subtracted from the time listed under the descendant's column for the profiled function. As a result, be aware that the value listed in the % time column for most profiled functions in this report will change.	To change the display and label information for **foo** and **bar**, as well as their qualifying descendants in the function call tree, and limit their entries and data in the **Call Graph Profile** report, enter: `Xprof -E foo -E bar a.out gmon.out`

Table 14. Xprof command line flags (continued)

Use this flag:	To:	For example:
-f	De-emphasize the general appearance of all function boxes in the function call tree, *except* for that of the specified function and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes except for that of the specified function and its descendants are made unavailable. The size of these boxes and the content of their labels remain the same. For the specified function and its descendants, the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.	To de-emphasize the display of function boxes for all functions in the function call tree except for **foo**, **bar**, and their descendants, and limit their types of entries in the Call Graph Profile report, enter: **Xprof -f foo -f bar a.out gmon.out**
-F	Change the general appearance and label information of all function boxes in the function call tree *except* for that of the specified function and its descendants. In addition, the number of entries in the Call Graph Profile report for the non-specified and non-descendant functions is limited, and the CPU data associated with them is changed. The **-F** flag overrides the **-E** flag. In the function call tree, the function box for the specified function is made unavailable, and its size and shape also changes so that it appears as a square of the smallest permitted size. In addition, the CPU time shown in the function box label appears as 0. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. When this is the case, the time in the self and descendants columns for this entry is set to **0**. As a result, be aware that the value listed in the % time column for most profiled functions in this report will change.	To change the display and label information of the function boxes for all functions except the functions foo and bar and their descendants, and limit their types of entries and data in the Call Graph Profile report, enter: **Xprof -F foo -F bar a.out gmon.out**
-h | -?	Display the **Xprof** command's usage statement.	**Xprof -h**
-L	Specify an alternative path name for locating shared libraries. If you plan to specify multiple paths, use the **Set File Search Path** option of the **File** menu on the Xprof GUI. See "Setting the file search sequence" on page 146 for more information.	To specify **/lib/profiled/libc.a:shr.o** as an alternative path name for your shared libraries, enter: **Xprof -L /lib/profiled/libc.a:shr.o**

Table 14. Xprof command line flags (continued)

Use this flag:	To:	For example:

-s	Produce the **gmon.sum** profile data file (if multiple **gmon.out** files are specified when Xprof is started). The **gmon.sum** file represents the sum of the profile information in all the specified profile files. Note that if you specify a single **gmon.out** file, the **gmon.sum** file contains the same data as the **gmon.out** file.	To write the sum of the data from three profile data files on AIX, **gmon.0_1.out**, **gmon.0_2.out**, and **gmon.0_3.out**, in to a file called **gmon.sum**, enter:
		`Xprof -s a.out gmon.0_1out gmon.0_2.out \ gmon.0_3.out`
		To write the sum of the data from three profile data files on Linux, **profdir.0_0/gmon.out**, **profdir.0_1/gmon.out**, and **profdir.0_2/gmon.out**, in to a file called **gmon.sum**, enter:
		`Xprof -s a.out profdir.0_0/gmon.out \ profdir.0_1/gmon.out profdir.0_2/gmon.out`

-z	Include functions that have both zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports. A function will not have a call count if the file that contains its definition was not compiled with the **-pg** flag, which is common with system library files.	To include all functions used by the application that have zero CPU usage and no call counts in the **Flat Profile**, **Call Graph Profile**, and **Function Index** reports, enter: `Xprof -z a.out gmon.out`

After you enter the **Xprof** command, the Xprof main window appears and displays your application's data.

Loading files from the Xprof GUI

If you enter the **Xprof** command on its own, you can then specify an executable file, one or more profile data file(s), and any flags, from within the Xprof GUI. You use the **Load File** option of the **File** menu to do this.

If you enter the **Xprof -h** or **Xprof -?** command, **Xprof** displays the usage statement for the command and then exits.

When you enter the **Xprof** command alone, the Xprof main window appears. Because you did not load an executable file or specify a profile data file, the window will be empty, as shown in Figure 85 on page 141. All that is visible is a menu bar at the top with dropdowns for **File**, **View**, **Filter**, **Report** and **Utility**.

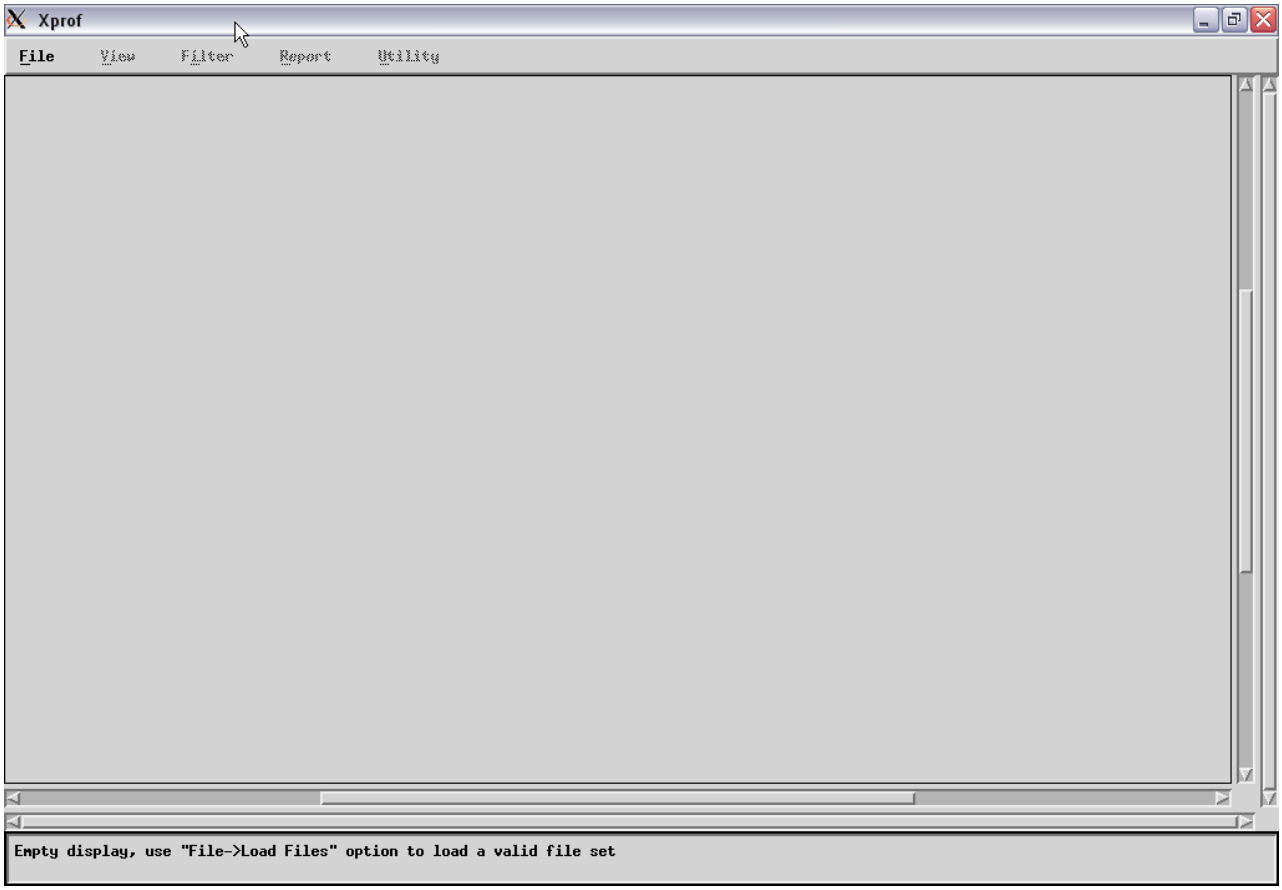


Figure 85. The Xprof main window

From the Xprof GUI, select **File**, then **Load File** from the menu bar. The **Load Files Dialog** will appear, as shown in Figure 86 on page 142. The **Load Files Dialog** is split into three different sections. There are two boxes, side-by-side at the top, and one long box at the bottom that are described in more detail in the following three figures.

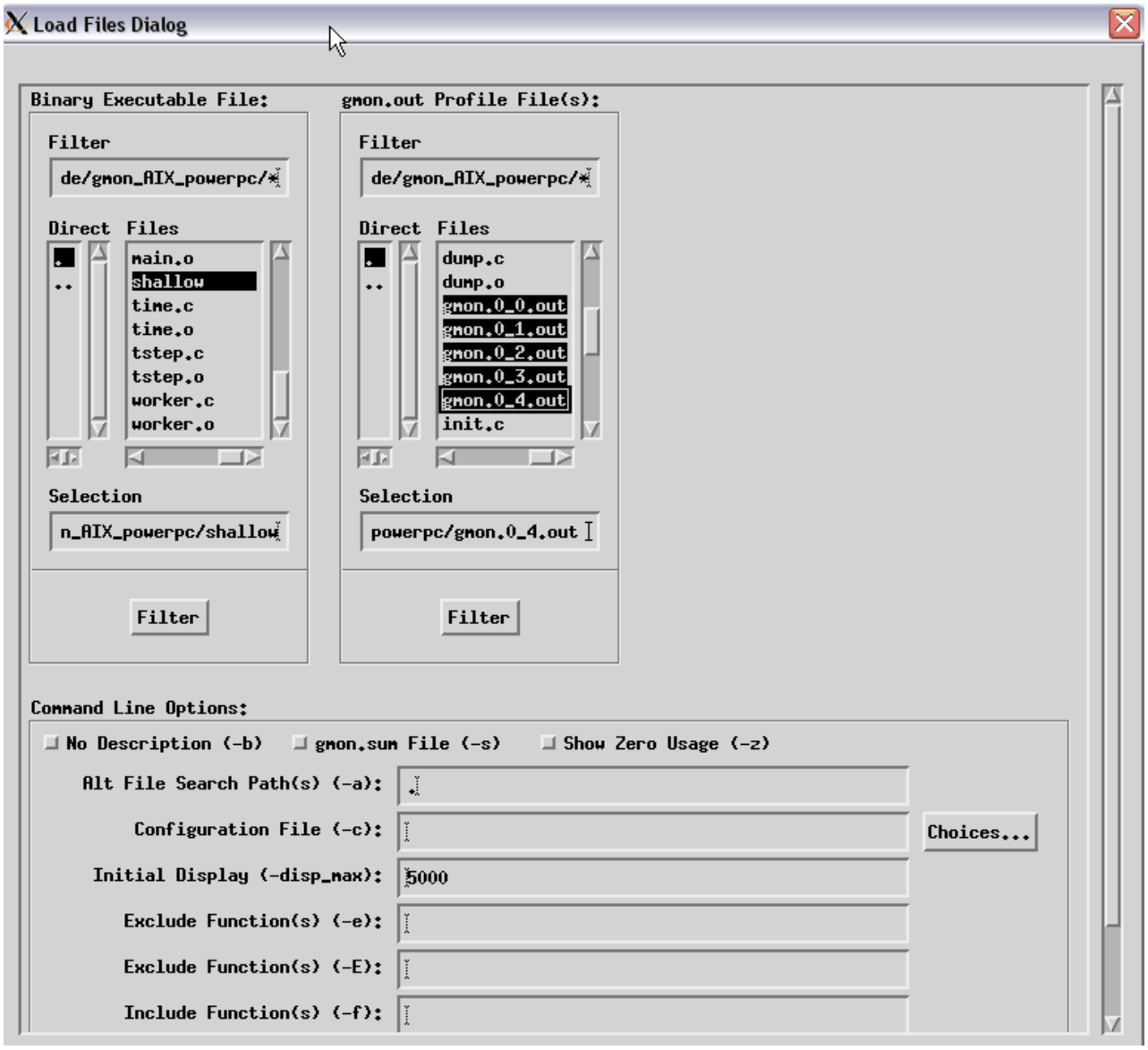


Figure 86. The Load Files Dialog

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

To load the files for the application you want to profile, you must specify the following:

- The binary executable file
- One or more profile data files

Optionally, you can also specify one or more command-line flags.

Specifying the binary executable file

You specify the binary executable file from the **Binary Executable File:** area of the **Load Files Dialog**. There is a **Filter** box at the top that shows the path of the file to load. Underneath the **Filter** box, there are two selection boxes, side by side that are labeled **Directory** and **Files**. The one on the left is to select the **Directory** in which to locate the executable file, and the one on the right is a listing of the files that are contained in the directory that is selected in the **Directory** selection box. There is a **Selection** box that shows the file selected and at the bottom there is a **Filter** button.



Figure 87. The binary executable file dialog

Use the scroll bars of the **Directories** and **Files** selection boxes to locate the executable file you want to load. By default, all of the files in the directory from which you called Xprof appear in the **Files** selection box.

To make locating your binary executable files easier, the **Binary Executable File:** area includes a **Filter** button. Filtering lets you limit the files that are displayed in

the **Files** selection box to those of a specific directory or of a specific type. For information about filtering, see "Filtering what you see" on page 155.

Specifying profile data files

You specify one or more profile data files from the **gmon.out Profile Data File(s):** area of the **Load Files Dialog**. There is a **Filter** box at the top that shows the path of the file to use as input. Underneath the **Filter** box, there are two selection boxes, side by side that are labeled **Directory** and **Files**. The one on the left is to select the **Directory** in which to locate the profile file, and the one on the right is a listing of the files that are contained in the directory that is selected in the **Directory** selection box. There is a **Selection** box that shows the file selected and at the bottom there is a **Filter** button.

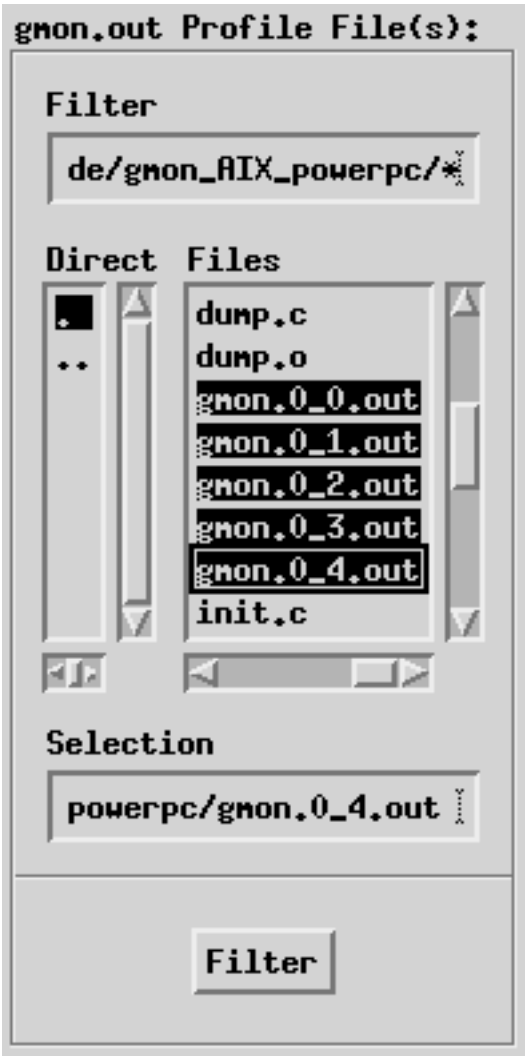


Figure 88. The gmon.out profile data file(s) area

When you start Xprof using the **Xprof** command, you are not required to indicate the name of the profile data file. If you do not specify a profile data file, Xprof searches your directory for the presence of a file named **gmon.out** and, if found, places it in the **Selection** field of the **gmon.out Profile Data File(s)** area, as the

default. Xprof then uses this file as input, even if it is not related to the binary executable file you specify. Because this will cause Xprof to display incorrect data, it is important that you enter the correct file in to this field. If the profile data file you want to use is named something other than what appears in the **Selection** field, you must replace it with the correct file name.

Use the scroll bars of the **Directories** and **Files** selection boxes to locate one or more of the profile data (**gmon.out**) files you want to specify. The file you use does not have to be named **gmon.out**, and you can specify more than one profile data file.

To make locating your output files easier, the **gmon.out Profile Data File(s)** area includes a **Filter** button. Filtering lets you limit the files that are displayed in the **Files** selection box to those in a specific directory or of a specific type. For information about filtering, see "Filtering what you see" on page 155.

Specifying Xprof flags from the GUI

Specify flags from the **Command Line Options** area of the **Load Files Dialog** (see "Specifying Xprof flags from the command line" on page 136). There are three check boxes side-by-side at the top:

- **No description** (**-b**)
- **gmon.sum File** (**-s**)
- **Show Zero Usage** (**-z**)

Below that, there are eight boxes corresponding to **Xprof** command line flags:

- **Alt File Search Paths** (**-a**)
- **Configuration File** (**-c**)
- **Initial Display** (**-disp_max**)
- **Exclude Functions** (**-e**)
- **Exclude Functions** (**-E**)
- **Include Functions** (**-f**)
- **Include Functions** (**-F**)
- **Alt Library Path** (**-L**)

There is also a **Choices** button next to the **Configuration File (-c)**, box which will bring up a dialog that allows you to choose a configuration file.

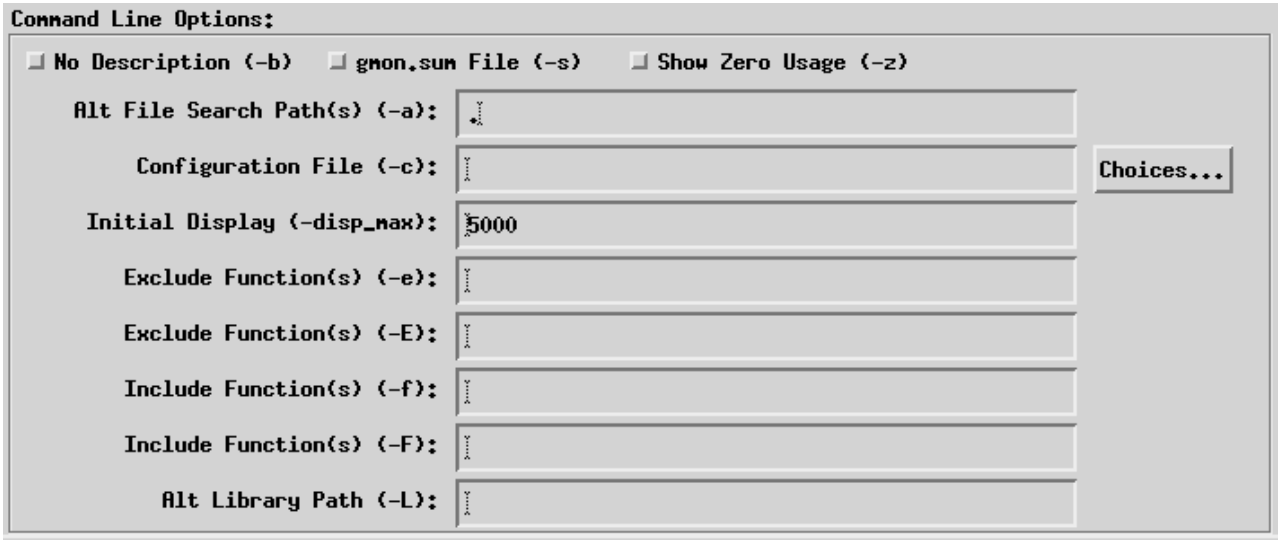


Figure 89. The Command Line Options area

After you have specified the binary executable file, one or more profile data files, and any command-line flags you want to use, click **OK** to save the changes and close the window. Xprof loads your application and displays its performance data.

Setting the file search sequence

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

Setting default paths

For library files, Xprof uses the paths recorded in the specified **gmon.out** files. If you use the **-L** flag, the path you specify with it will be used instead of those in the **gmon.out** files.

Note: The **-L** flag enables only one path to be specified and you can use this flag only once.

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

Setting alternative paths

You specify the alternative paths with the **Set File Search Paths** option of the **File** menu.

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

To specify alternative paths, do the following:

1. Select the **File** menu, and then the **Set File Search Paths** option. The **Alt File Search Path Dialog** appears.

2. Enter the name of the path in the **Enter Alt File Search Path(s)** text field. You can specify more than one path by separating each path name with a colon (:) or a space.

   Notes:

   a. You can use the "at" symbol (@) with this option to represent the default file path, in order to specify that other paths be searched before the default path. For example, to set the alternative file search paths so that Xprof searches **pathA**, the default path, then **pathB**, type:

      ```
      pathA:@:pathB
      ```

      in the **Alt File Search Path(s) (-a)** field.

   b. If @ is used in the alternative search path, the two buttons in the **Alt File Search Path Dialog** will be unavailable, and will have no effect on the search order.

3. Click **OK**. The paths you specified in the text field become the alternative paths.

Changing the search sequence

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

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

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

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

Understanding the Xprof display

The primary difference between Xprof and the **gprof** command is that Xprof gives you a graphical picture of your application's CPU consumption in addition to textual data.

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

The Xprof main window

The Xprof main window contains a graphical representation of the functions and calls within your application, as well as their interrelationships. The window provides five menus.

After an example application has been loaded, the Xprof main window shows one cluster box displaying a function call tree, with an arc pointing down to another cluster box displaying a function call tree.

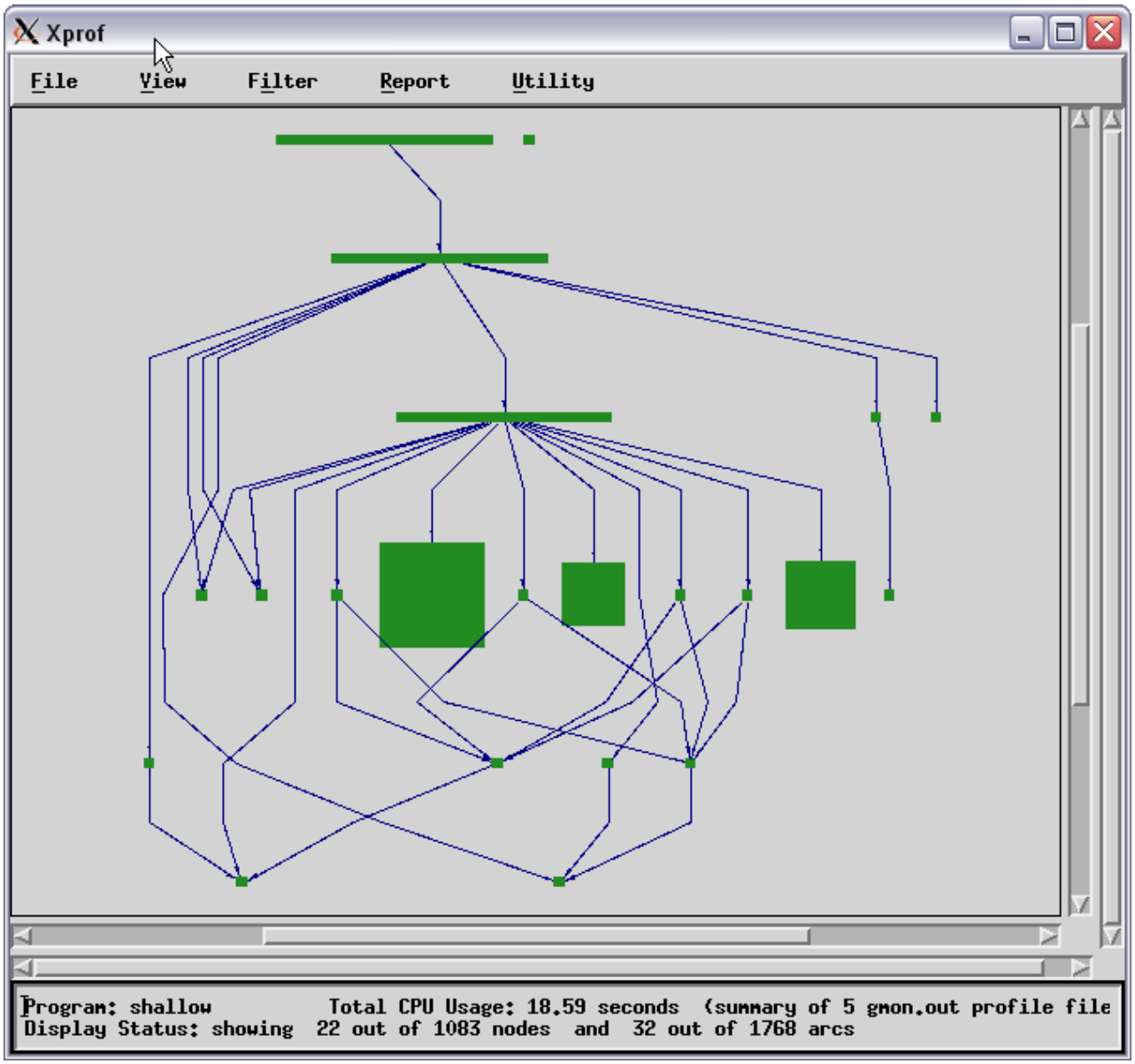


Figure 90. The Xprof main window with application loaded

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

Note: When Xprof first opens, by default, the function boxes for your application will be clustered by library. A cluster box appears around each library, and the function boxes and arcs within the cluster box are reduced in size. To see more detail, you must uncluster the functions. To do this, select the **File** menu and then the **Uncluster Functions** option.

Xprof's main menus

Table 15 lists the Xprof main menus:

Table 15. Xprof main menus

Menu name	Description
File	The **File** menu lets you specify the executable (**a.out**) files and profile data (**gmon.out**) files that Xprof will use. It also lets you control how your files are accessed and saved.
View	The **View** menu lets you focus on specific portions of the function call tree in order to get a better view of the application's critical areas.
Filter	The **Filter** menu lets you add, remove, and change specific parts of the function call tree. By controlling what Xprof displays, you can focus on the objects that are most important to you.
Report	The **Report** menu provides several types of profiled data in a textual and tabular format. In addition to presenting the profiled data, the options of the Report menu let you do the following: • Display textual data • Save it to a file • View the corresponding source code • Locate the corresponding function box or call arc in the function call tree
Utility	The **Utility** menu contains one option, **Locate Function By Name**, which lets you highlight a particular function in the function call tree.

Xprof's hidden menus

Table 16 lists Xprof's hidden menus:

Table 16. Xprof's hidden menus

Menu name	Description
Function	The **Function** menu lets you perform a number of operations for any of the functions shown in the function call tree. You can access statistical data, look at source code, and control which functions are displayed. The **Function** menu is not visible from the Xprof window. You access it by right-clicking on the function box of the function in which you are interested. By doing this, you open the **Function** menu, and select this function as well. Then, when you select actions from the **Function** menu, the actions are applied to this function.

Table 16. Xprof's hidden menus (continued)

Menu name	Description
Arc	The **Arc** menu lets you locate the caller and callee functions for a particular call arc. A call arc is the representation of a call between two functions within the function call tree. The **Arc** menu is not visible from the Xprof window. You access it by right-clicking on the call arc in which you are interested. By doing this you open the **Arc** menu, and select that call arc as well. Then, when you perform actions with the **Arc** menu, they are applied to that call arc.
Cluster Node	The **Cluster Node** menu lets you control the way your libraries are displayed by Xprof. To access the **Cluster Node** menu, the function boxes in the function call tree must first be clustered by library. For information about clustering and unclustering the function boxes of your application, see "Clustering libraries" on page 162. When the function call tree is clustered, all the function boxes within each library appear within a cluster box. The **Cluster Node** menu is not visible from the Xprof window. You access it by right-clicking on the edge of the cluster box in which you are interested. By doing this you open the **Cluster Node** menu, and select that cluster as well. Then, when you perform actions with the **Cluster Node** menu, they are applied to the functions within that library cluster.

The display status field

At the bottom of the Xprof window is a single field that provides the following information:

- Name of your application
- Number of **gmon.out** files used in this session
- Total amount of CPU used by the application
- Number of functions and calls in your application, and how many of these are currently displayed

How functions are represented

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

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

Functions can also be represented in average mode. In average mode, the size and shape of each function box is determined by the average CPU time used on that

function alone, among all loaded **gmon.out** files, and the standard deviation of CPU time for that function among all loaded **gmon.out** files. The height of each function node represents the average CPU time, among all the input **gmon.out** files, used on the function itself. The width of each node represents the standard deviation of CPU time, among the **gmon.out** files, used on the function itself. The average mode representation is available only when more than one **gmon.out** file is entered. For more information about summary mode and average mode, see "Controlling the representation of the function call tree" on page 154.

Under each function box in the function call tree is a label that contains the name of the function and related CPU usage data. For information about the function box labels, see "Obtaining basic data" on page 167.

Figure 91 shows the function boxes for two functions, **sub1** and **printf**, as they would appear in the Xprof display.

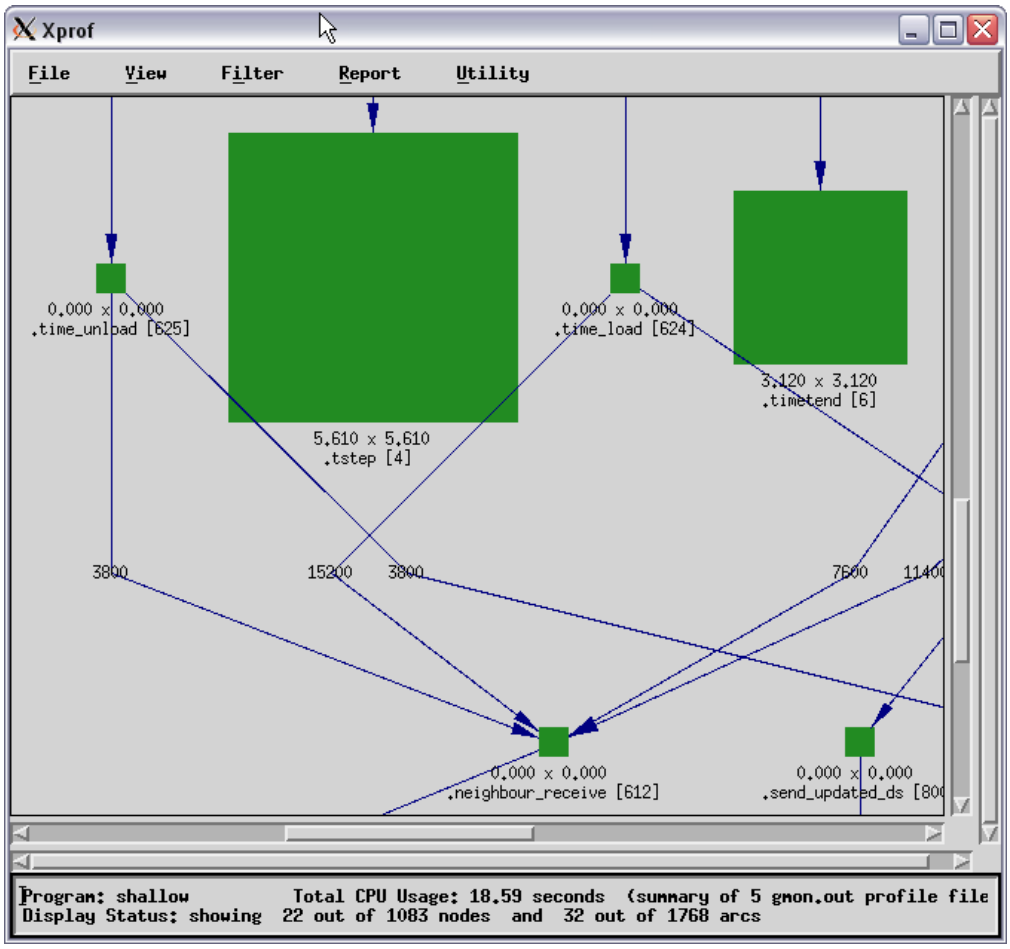


Figure 91. Function boxes and arcs in the Xprof display

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

How calls between functions are depicted

The calls made between each of the functions in the function call tree are represented by blue arrows extending between their corresponding function boxes. These lines are called call arcs. Each call arc appears as a solid blue line between two functions. The arrowhead indicates the direction of the call; the function represented by the function box it points to is the one that receives the call. The function making the call is known as the caller, while the function receiving the call is known as the callee.

Each call arc includes a numeric label that indicates how many calls were exchanged between the two corresponding functions.

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

How library clusters are represented

Xprof lets you collect the function boxes and call arcs that belong to each of your shared libraries in to cluster boxes.

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

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

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

Controlling how the display is updated

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

Other viewing options

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

Controlling the graphic style of the function call tree

You can choose between two-dimensional and three-dimensional function boxes in the function call tree. The default style is two-dimensional. To change to three-dimensional, select the **View** menu, and then the **3-D Image** option. The function boxes in the function call tree now appear in three-dimensional format.

Controlling the orientation of the function call tree

You can choose to have Xprof display the function call tree in either top-to-bottom or left-to-right format. The default is top-to-bottom. To see the function call tree displayed in left-to-right format, select the **View** menu, and then the **Layout: Left→Right** option. The function call tree now displays in left-to-right format, as shown in Figure 92 on page 154:

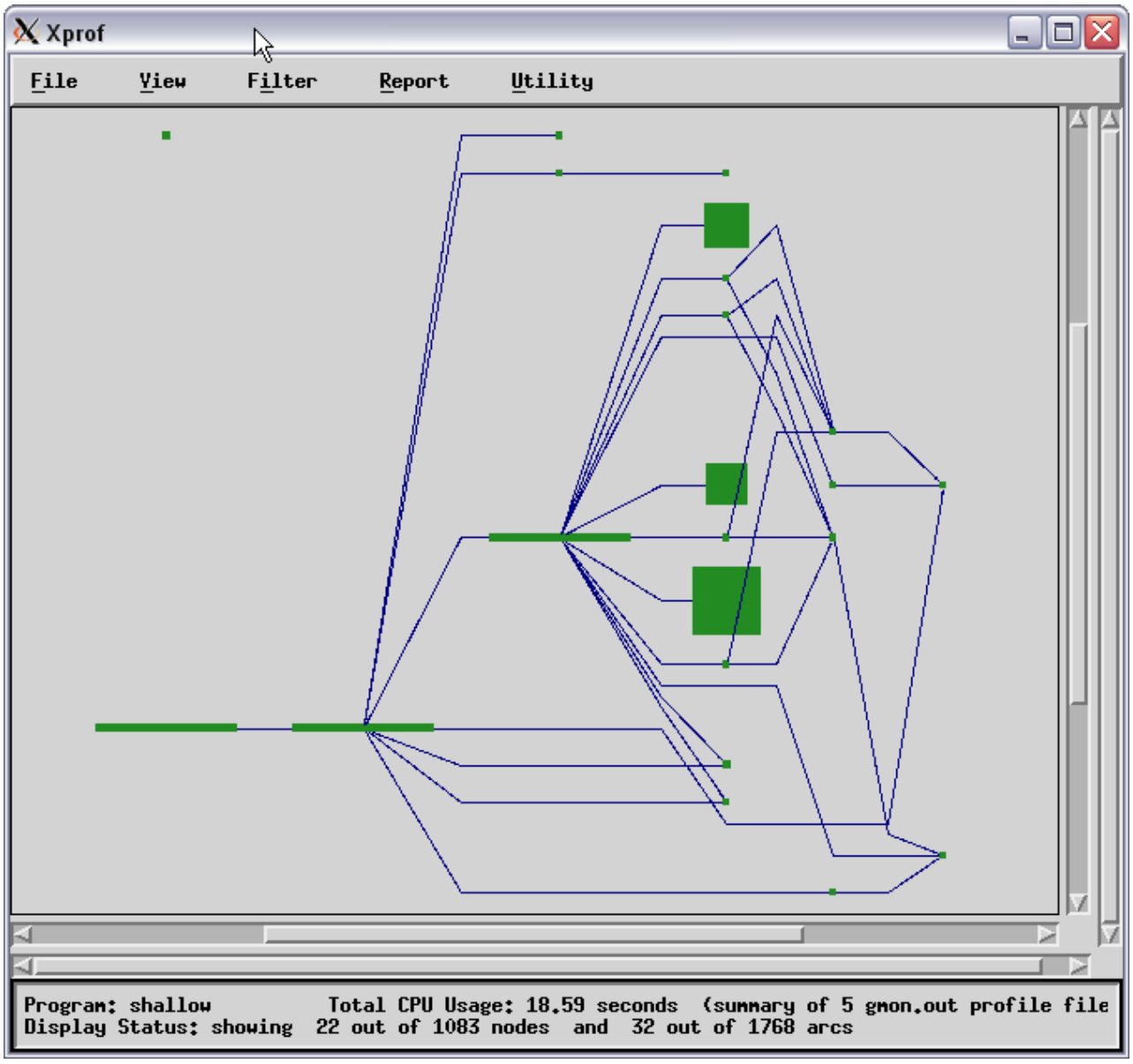


Figure 92. The Xprof main window with left-to-right format selected

Controlling the representation of the function call tree

You can choose to have Xprof represent the function call tree in either summary mode or average mode. These choices are only available when processing more than one **gmon.out** file.

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

time used on the function and its descendant functions. When the display is in summary mode, the **Summary Mode** option is unavailable and the **Average Mode** option is activated.

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

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

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

Filtering what you see

When Xprof first opens, the entire function call tree appears in the main window. This includes the function boxes and call arcs that belong to your executable file as well as the shared libraries that it uses. You can simplify what you see in the main window, and there are several ways to do this.

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

Restoring the status of the function call tree

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

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

The options that activate the **Undo** option include the following:

- In the main **File** menu:
  - Load Configuration
- In the main **Filter** menu:

  - Show Entire Call Tree
  - Hide All Library Calls
  - Add Library Calls
  - Filter by Function Names
  - Filter by CPU Time
  - Filter by Call Counts
- In the **Function** menu:
  - Immediate Parents
  - All Paths To
  - Immediate Children
  - All Paths From
  - All Functions on The Cycle
- In the **Function Display Options** submenu of the Function menu:
  - Show This Function Only
  - Hide This Function
  - Hide Descendant Functions
  - Hide This & Descendant Functions

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

Displaying the entire function call tree

When you first open Xprof, by default, all the function boxes and call arcs of your executable and its shared libraries appear in the main window. After that, you can choose to filter out specific items from the window. However, there might be times when you want to see the entire function call tree again, without having to reload your application. To do this, select the **Filter** menu, and then the **Show Entire Call Tree** option. Xprof erases whatever is currently displayed in the main window and replaces it with the entire function call tree.

Excluding and including specific objects

There are a number of ways that Xprof lets you control the items that display in the main window. You will want to include or exclude certain objects so that you can more easily focus on the things that are of most interest to you.

Filtering shared library functions

In most cases, your application will call functions that are within shared libraries. By default, these shared libraries display in the Xprof window along with your executable file. As a result, the window can get crowded and obscure the items that you most need to see. If this is the case, you can filter the shared libraries from the display. To do this, select the **Filter** menu, and then the **Remove All Library Calls** option.

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

If you removed the library calls from the display, you might want to restore them. To do this, select the **Filter** menu and then the **Add Library Calls** option.

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

To add only specific function boxes back in to the display, do the following:

1. Select the **Filter** menu, and then the **Filter by Function Names** option. The **Filter By Function Names Dialog** appears.
2. In the **Filter By Function Names Dialog**, select **add these functions to graph**, and then type the name of the function you want to add in the **Enter function name(s)** field. If you enter more than one function name, you must add a blank space between each function name string. Regular expressions are also supported.

   If there are multiple functions in your program that include the string you enter in their names, the filter applies to each one. For example, if you specified **sub** and **print**, and your program also included functions named **sub1**, **psub1**, and **printf**, the **sub**, **sub1, psub1**, **print**, and **printf** functions would all be added to the graph.
3. Click **OK**. One or more function boxes appear in the Xprof display with the function call tree.

Filtering by function characteristics

The **Filter** menu of Xprof offers the following options that enable you to add or subtract function boxes from the main window, based on specific characteristics:

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

Each option uses a different window to let you specify the criteria by which you want to include or exclude function boxes from the window.

To filter by function names, do the following:

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

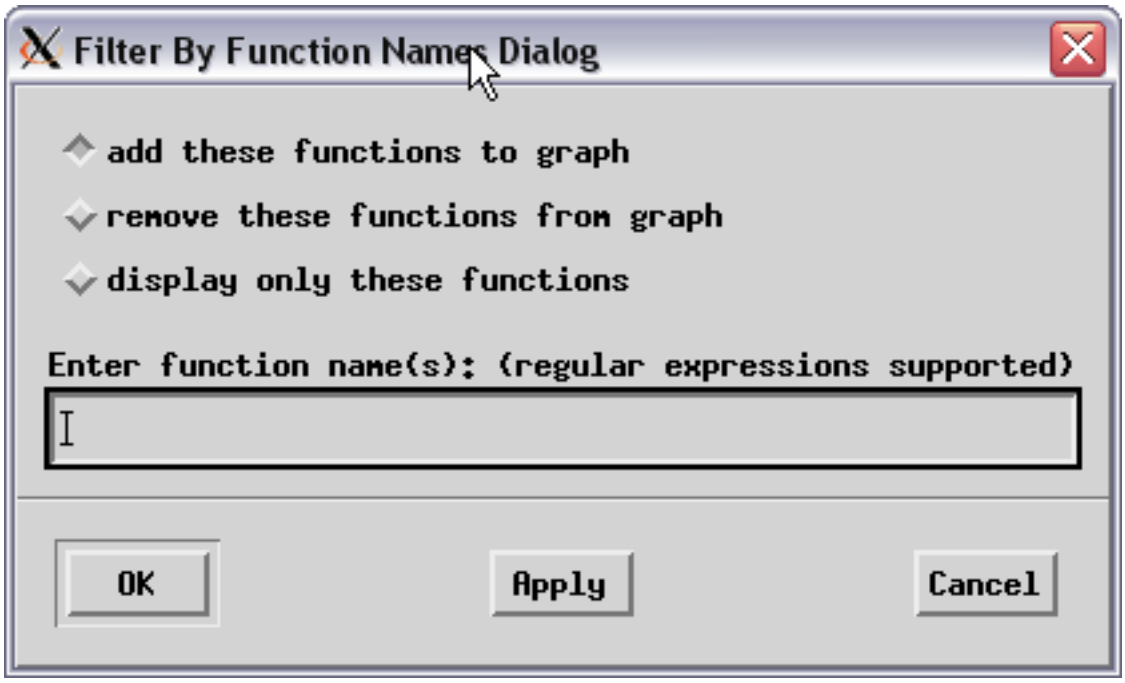


Figure 93. The Filter by Function Names Dialog

The **Filter By Function Names Dialog** includes the following options:

- **Add these functions to graph**
- **Remove these functions from the graph**
- **Display only these functions**
- **Enter function names()** field for specifying a set of function names

2. From the **Filter By Function Names Dialog**, select the option, and then type the name of the function (or functions) to which you want it applied in the **Enter function name(s)** field.

   For example, if you want to remove the function box for a function called **printf** from the main window, click the remove this function from the graph button, and type **printf** in the **Enter function name(s)** field.

   You can enter more than one function name in this field. If there are multiple functions in your program that include the string you enter in their names, the filter will apply to each one. For example, if you specified **sub** and **print,** and your program also included functions named **sub1**, **psub1**, and **printf**, the option you chose would be applied to the **sub**, **sub1**, **psub1**, **print**, and **printf** functions.

3. Click **OK**. The contents of the function call tree now reflect the filtering options you specified.

To filter by CPU time, do the following:

1. Select the **Filter** menu and then the **Filter by CPU Time** option. The **Filter By CPU Time Dialog** appears:

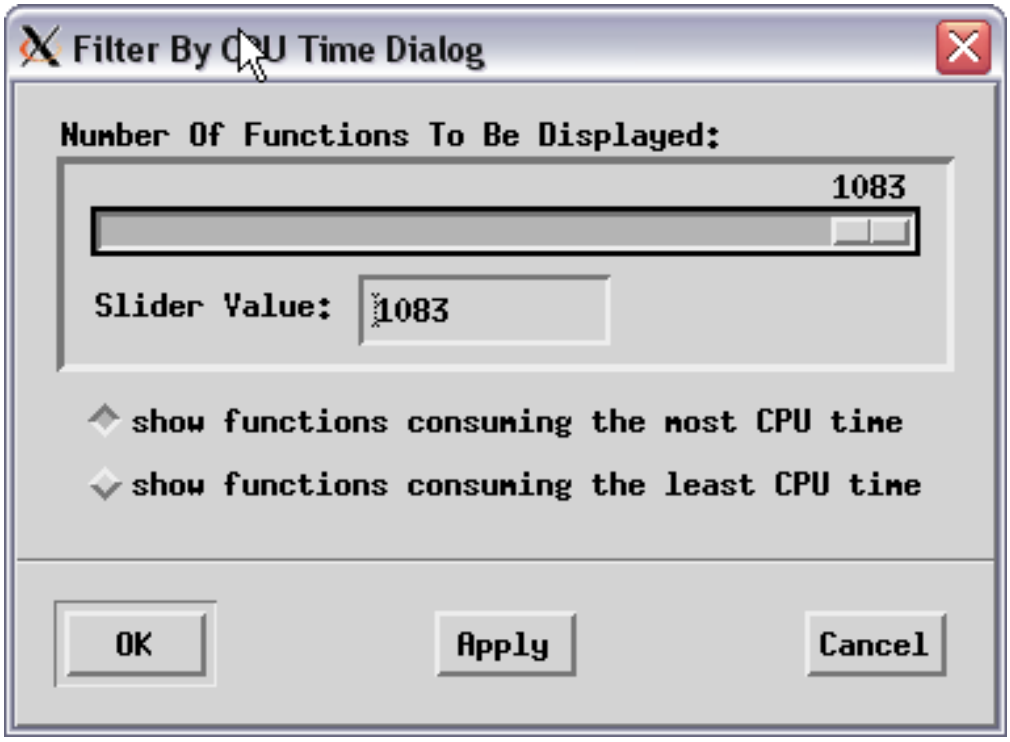


Figure 94. The Filter by CPU Time Dialog

The **Filter By CPU Time Dialog** includes the following options:

- A slider bar to select the number of functions to display
- A value field to select the number of functions to display
- Show functions consuming the most CPU time
- Show functions consuming the least CPU time

2. Select the option you want (**show functions consuming the most CPU time** is the default).
3. Select the number of functions to which you want it applied. You can move the slider in the **Number Of Functions To Be Displayed** bar until the desired number appears, or you can enter the number in the **Slider Value** field. The slider and **Slider Value** field are synchronized so when the slider is updated, the text field value is updated also. If you enter a value in the text field, the slider is updated to that value when you click **Apply** or **OK.** For example, to display the function boxes for the 10 functions in your application that consumed the most CPU, you would select the **show functions consuming the most CPU** button, and specify 10 with the slider or enter the value 10 in the text field.
4. Click **Apply** to show the changes to the function call tree without closing the dialog. Click **OK** to show the changes and close the dialog.

To filter by call counts, do the following:

1. Select the **Filter** menu and then the **Filter by Call Counts** option. The **Filter By Call Counts Dialog** appears.

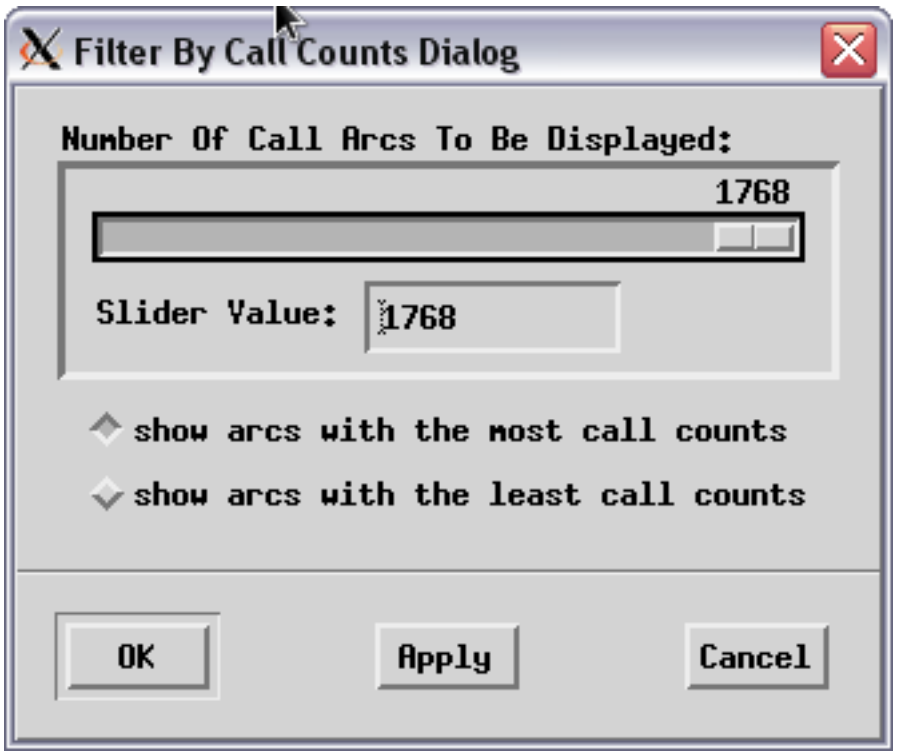


Figure 95. The Filter by Call Counts Dialog

The **Filter By Call Counts Dialog** includes the following options:

- A slider bar to select the number of functions to display
- A value field to select the number of functions to display
- **Show arcs with the most call counts**
- **Show arcs with the least call counts**

2. Select the option you want (**show arcs with the most call counts** is the default).
3. Select the number of call arcs to which you want it applied. If you enter a value in the text field, the slider is updated to that value when you click **Apply** or **OK**. For example, to display the 10 call arcs in your application that represented the least number of calls, you would select the **show arcs with the least call counts** button, and specify 10 with the slider or enter the value 10 in the text field.
4. Click **Apply** to show the changes to the function call tree without closing the dialog. Click **OK** to show the changes and close the dialog.

Including and excluding parent and child functions

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

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

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

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

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

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select the **Immediate Children** option, and then the **Show Child Functions Only** option. Xprof erases the current display and replaces it with only the function boxes for the function you chose, as well as its child functions.

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

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select the **Immediate Parents** option, and then the **Show Parent Functions Only** option. Xprof erases the current display and replaces it with only the function boxes for the function you chose, as well as its parent functions.

You might want to view the function boxes for both the parent and child functions of the function in which you are interested, without erasing the rest of the function call tree. This is especially true if you chose to display the function boxes for two or more of the most CPU-intensive functions with the **Filter by CPU** option of the **Filter** menu (you suspect that more than one function is consuming too much CPU). Do the following:

1. Place your mouse cursor over the function box in which you are interested, and press the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select the **Immediate Parents** option, and then the **Add Parent Functions to Tree** option. Xprof leaves the current display as it is, but adds the parent function boxes.
3. Place your mouse cursor over the same function box and press the right mouse button. The **Function** menu appears.
4. From the Function menu, select the **Immediate Children** option, and then the **Add Child Functions to Tree** option.

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

Clustering libraries

When you first open the Xprof window, by default, the function boxes of your executable file, and the libraries associated with it, are clustered. Because Xprof shrinks the call tree of each library when it places it in a cluster, you must uncluster the function boxes if you want to look closely at a specific function box label.

You can see much more detail for each function, when your display is in the unclustered or expanded state, then when it is in the clustered or collapsed state. Depending on what you want to do, you must cluster or uncluster (collapse or expand) the display.

The Xprof window can be visually crowded, especially if your application calls functions that are within shared libraries; function boxes representing your executable functions as well as the functions of the shared libraries are displayed. As a result, you might want to organize what you see in the Xprof window so you can focus on the areas that are most important to you. You can do this by collecting all the function boxes of each library into a single area, known as a library cluster.

Figure 96 shows an application with its function boxes unclustered.

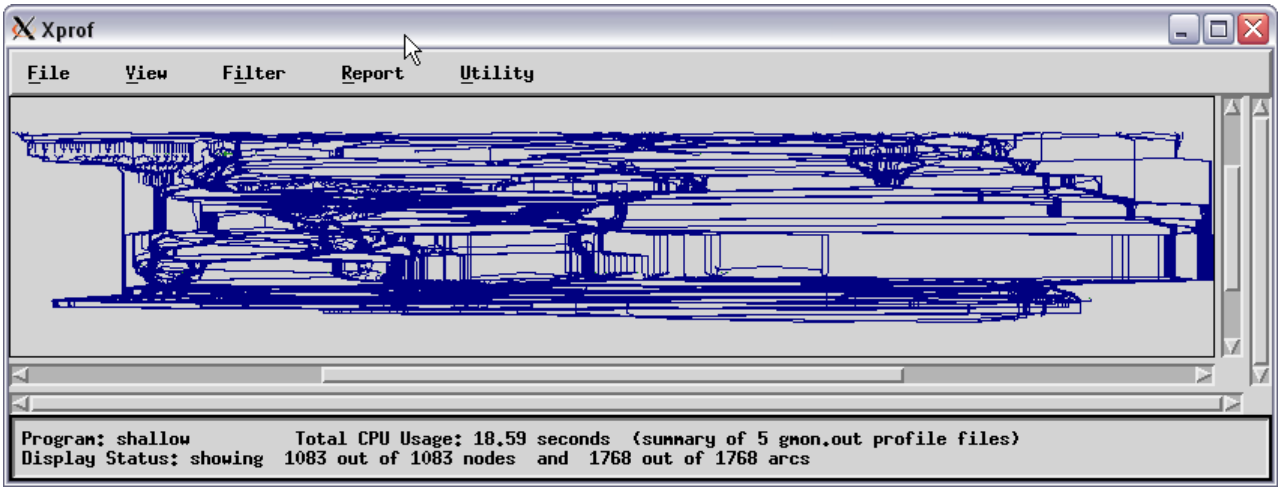


Figure 96. The Xprof main window with functions unclustered

Clustering functions

If the functions within your application are unclustered, you can use an option of the **Filter** menu to cluster them. To do this, select the **Filter** menu and then the **Cluster Functions by Library** option. The libraries within your application appear within their respective cluster boxes.

After you cluster the functions in your application you can further reduce the size (also referred to as collapse) of each cluster box by doing the following:

1. Place your mouse cursor over the edge of the cluster box and press the right mouse button. The **Cluster Node** menu appears.

2. Select the **Collapse Cluster Node** option. The cluster box and its contents now appear as a small solid green box. In Figure 95 on page 160, the **/lib/profiled/libc.a:shr.o** library is collapsed.

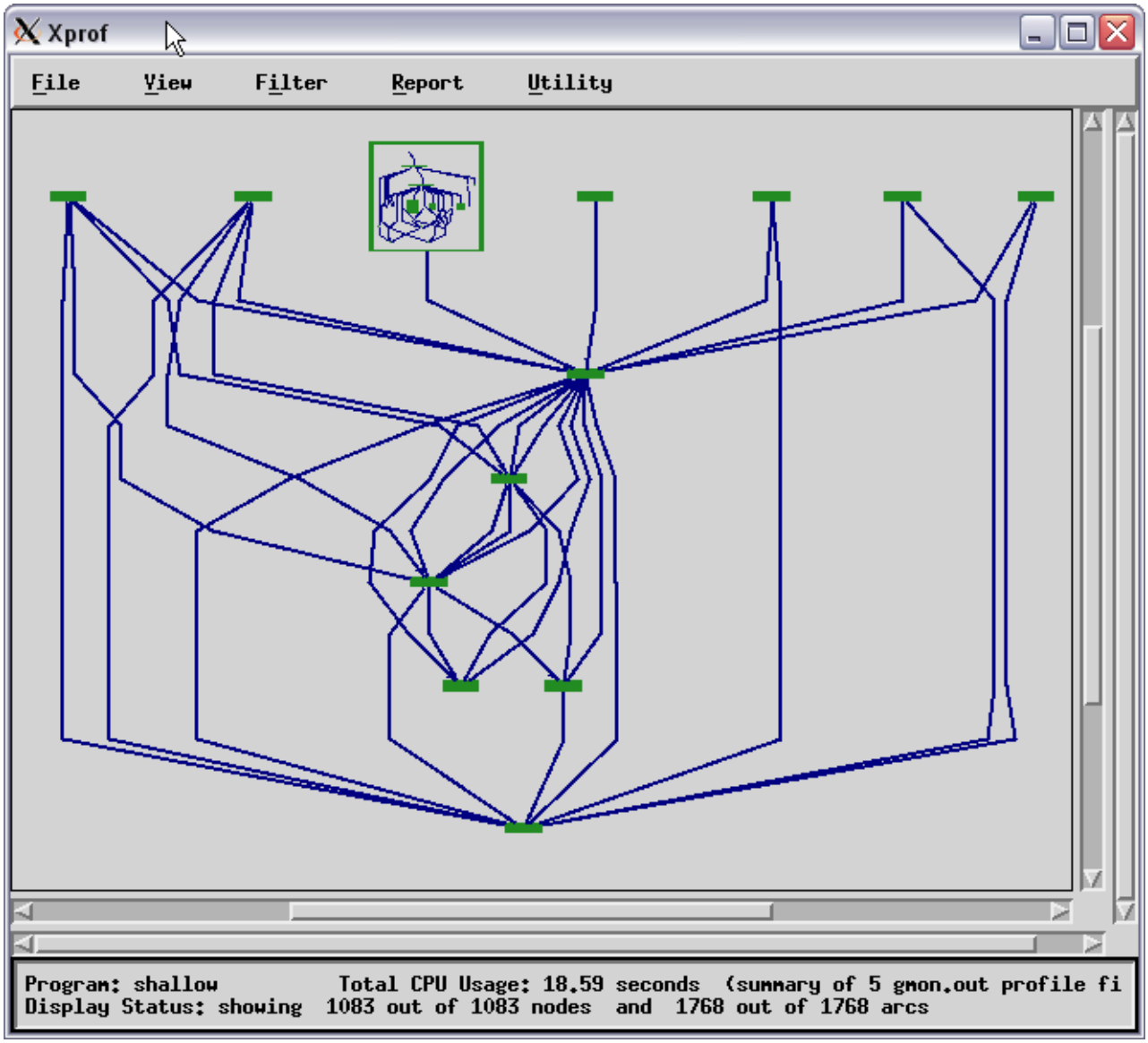


Figure 97. The Xprof main window with one library cluster box collapsed

To return the cluster box to its original condition (expand it), do the following:

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

Unclustering functions

If the functions within your application are clustered, you can use an option of the **Filter** menu to uncluster them. To do this, select the **Filter** menu, and then the **Uncluster Functions** option. The cluster boxes disappear and the functions boxes of each library expand to fill the Xprof window.

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

1. Place your mouse cursor over the edge of the cluster box that contains the executable and press the right mouse button. The **Cluster Node** menu appears.
2. Select the **Remove Cluster Box** option. The cluster box is removed and the function boxes and call arcs that represent the executable functions, now appear in full detail. The function boxes and call arcs of the shared libraries remain within their cluster boxes, which now appear smaller to make room for the unclustered executable function boxes. Figure 96 on page 162 shows an application with its cluster box removed. Its shared library remains within its cluster box.

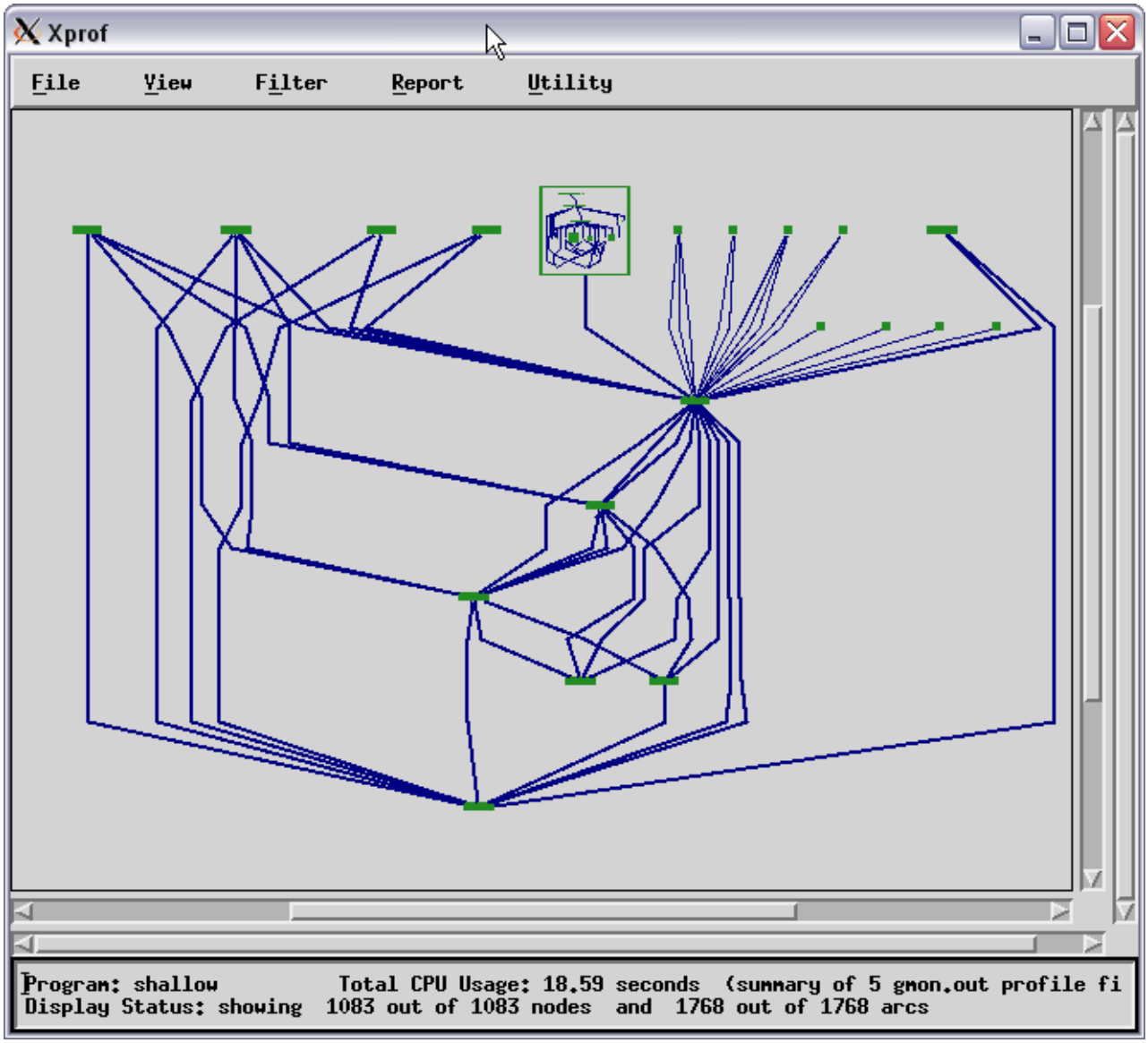


Figure 98. The Xprof main window with one library cluster box removed

Locating specific objects in the function call tree

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

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

1. Select the **Utility** menu, and then the **Locate Function By Name** option. The **Search By Function Name Dialog** appears.

2. Type the name of the function you want to locate in the **Enter function name** field. The function name you type here must be a continuous string (it cannot include blanks). Regular expressions are supported.
3. Click **OK** or **Apply**. The corresponding function box is highlighted (its color changes to red) in the function call tree and Xprof zooms in on its location. To display the function call tree in full detail again, go to the **View** menu and use the **Overview** option.

You might want to see only the function boxes for the functions that you are concerned with, in addition to other specific functions that are related to it. For example, if you want to see all the functions that directly called the function in which you are interested, it might not be easy to separate these function boxes when you view the entire call tree. You would want to display them, as well as the function of interest, alone.

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

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

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

Locating and displaying parent functions

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

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

Locating and displaying child functions

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

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

Locating and displaying ancestor functions

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

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

Locating and displaying descendant functions

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

1. Click the function box of interest with the right mouse button. The **Function** menu appears.
2. From the **Function** menu, select **All Paths From** then **Show Descendant Functions Only**. Xprof redraws the display to show you only the function boxes for the function of interest and its descendant functions.

Locating and displaying functions on a cycle

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

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

Obtaining performance data for your application

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

Obtaining basic data

Xprof makes it easy to get data on specific items in the function call tree. After you have located the item you are interested in, you can get data a number of ways. If you are having trouble locating a function in the function call tree, see "Locating specific objects in the function call tree" on page 165.

Understanding basic function data

Below each function box in the function call tree is a label that contains basic performance data, similar to the following:

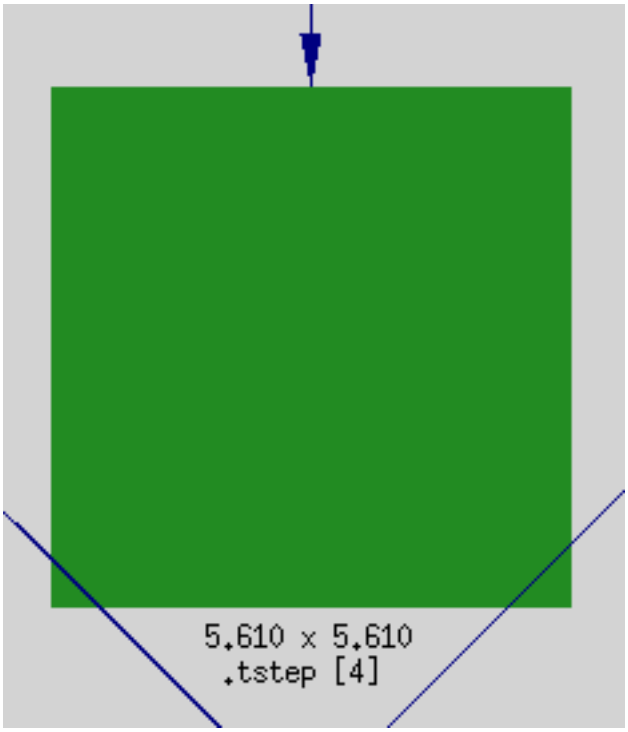


Figure 99. An example of a function box label

The label contains the name of the function, its associated cycle, if any, and its index. In Figure 99, the name of the function is **sub1**. It is associated with cycle 1 and its index is 5. Also, depending on whether the function call tree is viewed in summary mode or average mode, the label will contain different information.

If the function call tree is viewed in summary mode, the label will contain the following information:

- The total amount of CPU time (in seconds) this function spent on itself plus the amount of CPU time it spent on its descendants (the number to the left of the **x**).
- The amount of CPU time (in seconds) this function spent only on itself (the number to the right of the **x**).

If the function call tree is viewed in average mode, the label will contain the following information:

- The average CPU time (in seconds), among all the input **gmon.out** files, used on the function itself
- The standard deviation of CPU time (in seconds), among all the input **gmon.out** files, used on the function itself

For more information about summary mode and average mode, see "Controlling the representation of the function call tree" on page 154.

Because labels are not always visible in the Xprof window when it is fully zoomed out, you might need to zoom in on it in order to see the labels. For information about how to do this, see "Understanding information boxes" on page 169.

Understanding basic call data

Call arc labels appear over each call arc. The label indicates the number of calls that were made between the two functions (from caller to callee). For example, in the screen capture below, there are three arcs pointing to a function box. Each arc has a call arc label that indicates the number of calls that were made between the two functions, and in this example the arc labels are 3, 4, and 4.

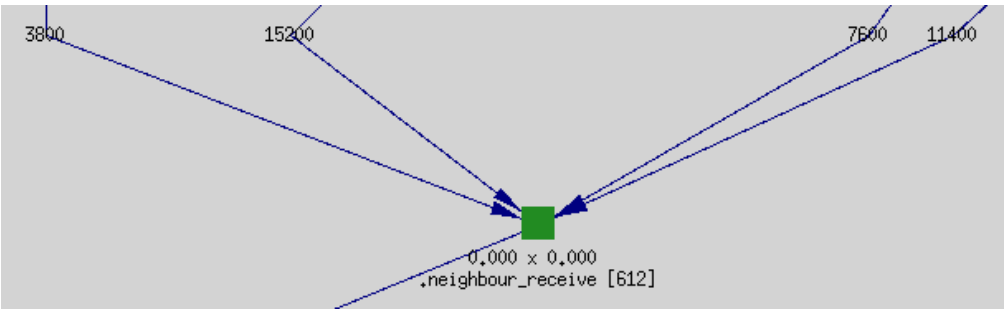


Figure 100. An example of a call arc label

To see a call arc label, you can zoom in on it. For information about how to do this, see "Understanding information boxes."

Basic cluster data

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

Understanding information boxes

For each function box, call arc, and cluster box, a corresponding information box gives you the same basic data that appears on the label. This is useful when the Xprof display is fully zoomed out and the labels are not visible. To access the information box, click on the function box, call arc, or cluster box (place the mouse pointer over the edge of the box) with the left mouse button. The information box appears.

For a function, the information box contains the following:

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

For a call, the information box contains the following:

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

For a cluster, the information box contains the following:

- The name of the library

- The total CPU usage (in seconds) consumed by the functions within it

Using the function menu statistics report option

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

When you select the **Statistics Report** menu option, the **Function Level Statistics Report** window appears.

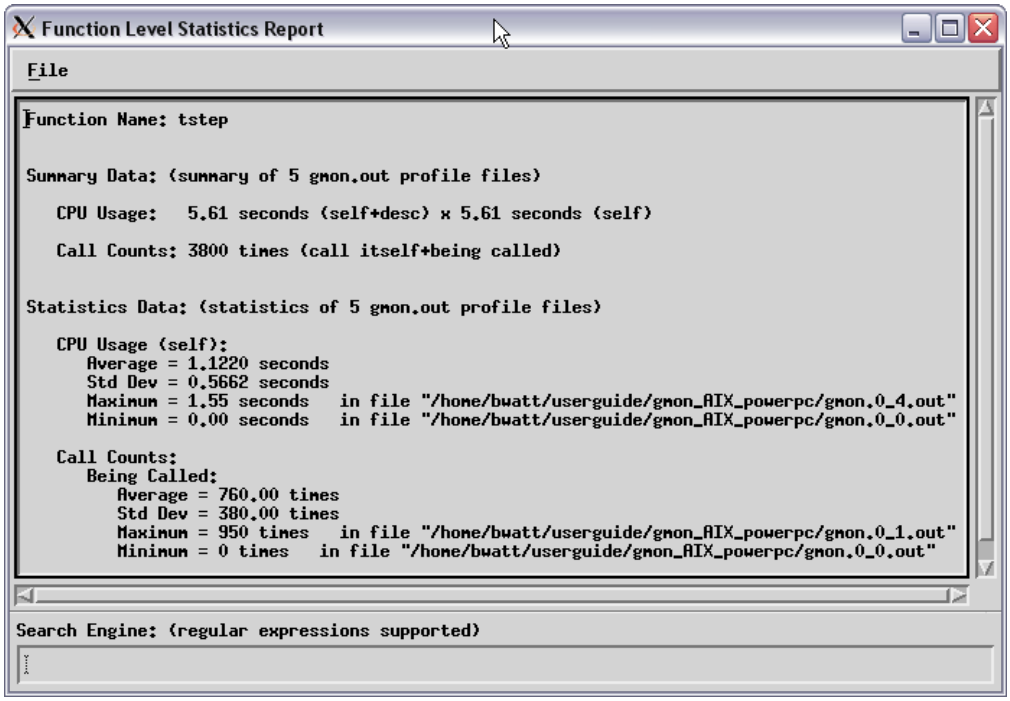


Figure 101. The Function Level Statistics Report window

The **Function Level Statistics Report** window provides the following information:

Function Name
The name of the function you selected.

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

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

Call Counts
This field indicates the number of times this function called itself, plus the number of times it was called by other functions.

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

The **CPU Usage field** indicates:

Average
The average CPU time used by the data in each **gmon.out** file.

Std Dev
Standard deviation. A value that represents the difference in CPU usage samplings, per function, from one **gmon.out** file to another. The smaller the standard deviation, the more balanced the workload.

Maximum
Of all the **gmon.out** files, the maximum amount of CPU time used. The corresponding **gmon.out** file appears to the right.

Minimum
Of all the **gmon.out** files, the minimum amount of CPU time used. The corresponding **gmon.out** file appears to the right.

The **Call Counts** field indicates:

Average
The average number of calls made to this function or by this function, for each **gmon.out** file.

Std Dev
Standard deviation. A value that represents the difference in call count sampling, per function, from one **gmon.out** file to another. A small standard deviation value in this field means that the function was almost always called the same number of times in each **gmon.out** file.

Maximum
The maximum number of calls made to this function or by this function in a single **gmon.out** file. The corresponding **gmon.out** file appears to the right.

Minimum
The minimum number of calls made to this function or by this function in a single **gmon.out** file. The corresponding **gmon.out** file appears to the right.

Getting detailed data from reports

Xprof provides performance data in textual and tabular format. This data is provided in various tables called reports. Similar to the **gprof** command, Xprof generates the Flat Profile, Call Graph Profile, and Function Index reports, as well as two additional reports.

You can access the Xprof reports from the **Report** menu. The **Report** menu displays the following reports:

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

Each report window includes a **File** menu. Under the **File** menu is the **Save As** option, which lets you save the report to a file. For information about using the **Save File Dialog** to save a report to a file, see "Saving the call graph profile, function index, and flat profile reports to a file" on page 181.

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

Each of the Xprof reports are explained in the topics that follow.

Understanding the flat profile report

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

Unless you specified the **-z** flag, the Flat Profile report does not include functions that have no CPU usage and no call counts. The data presented in the **Flat Profile** window is the same data that is generated with the **gprof** command.

The Flat Profile report looks similar to Figure 102 on page 173:

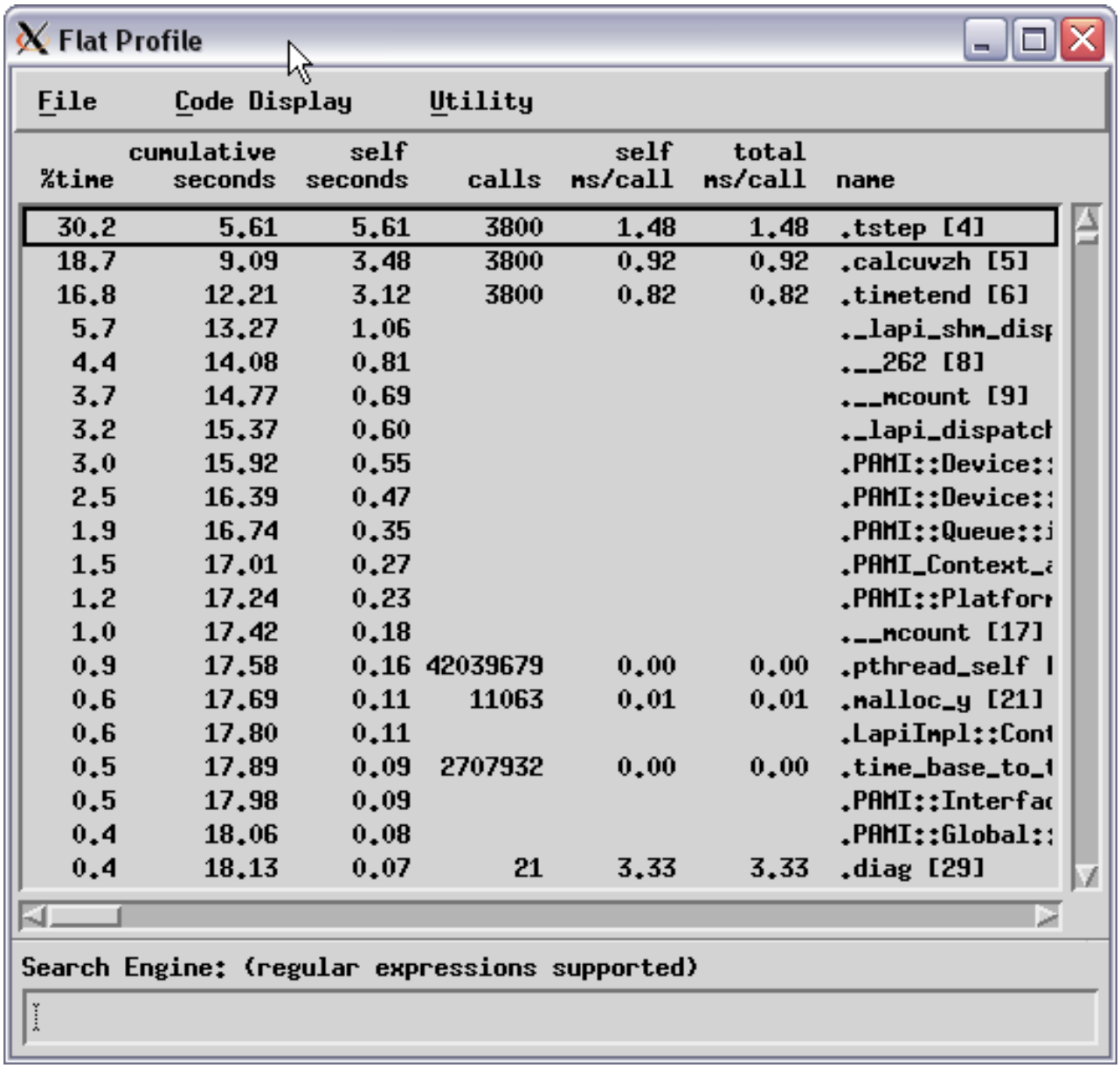


Figure 102. The Flat Profile window

The **Flat Profile** window contains the following fields:

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

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

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

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

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

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

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

Understanding the call graph profile report

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

Unless you specified the **-z** flag, the Call Graph Profile report does not include functions whose CPU usage is 0 (zero) and have no call counts. The data presented in the **Call Graph Profile** window is the same data that is generated with the **gprof** command.

The Call Graph Profile report looks similar to Figure 103 on page 175:

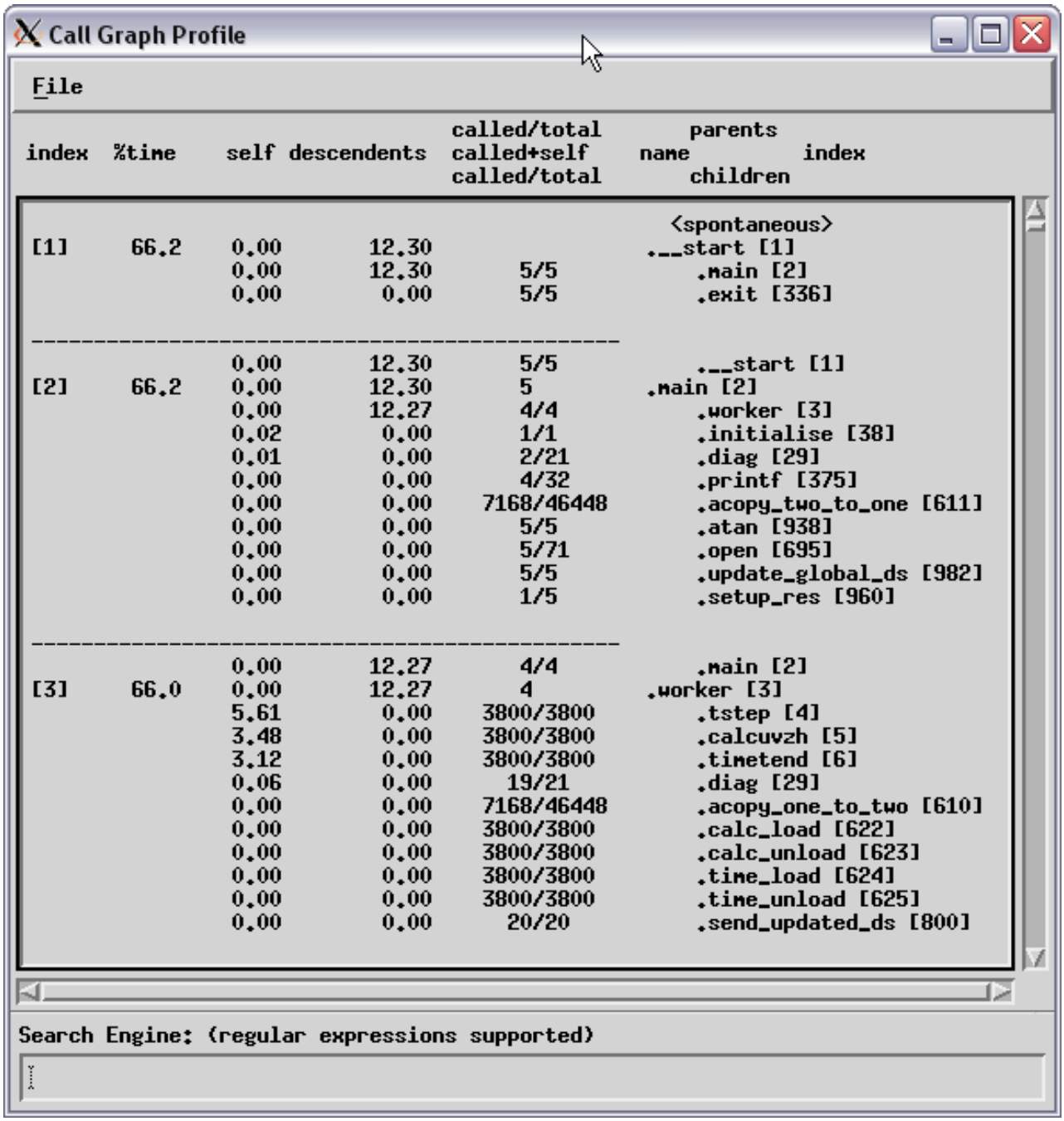


Figure 103. The Call Graph Profile window

The **Call Graph Profile** window contains the following fields:

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

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

self
The number of seconds this function spends within itself.

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

called/total, called+self, called/total
The heading of this column refers to the different kinds of calls that take place within your program. The values in this field correspond to the functions listed in the name, index, parents, children field to its right. Depending on whether the function is a parent, a child, or the function of interest (the function with the index listed in the index field of this row), this value might represent the number of times that:

- A parent called the function of interest
- The function of interest called itself, recursively
- The function of interest called a child

In Figure 104, **sub2** is the function of interest, **sub1** and **main** are its parents, and **printf** and **sub1** are its children.

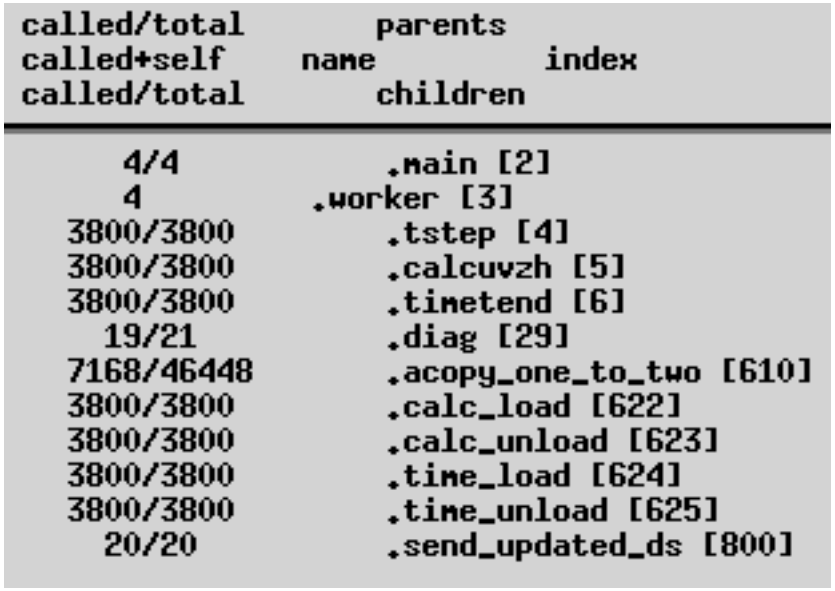


Figure 104. The called/total, call/self, called/total field of the Call Graph Profile window

called/total
For a parent function, the number of calls made to the function of interest, as well as the total number of calls it made to all functions.

called+self
The number of times the function of interest called itself, recursively.

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

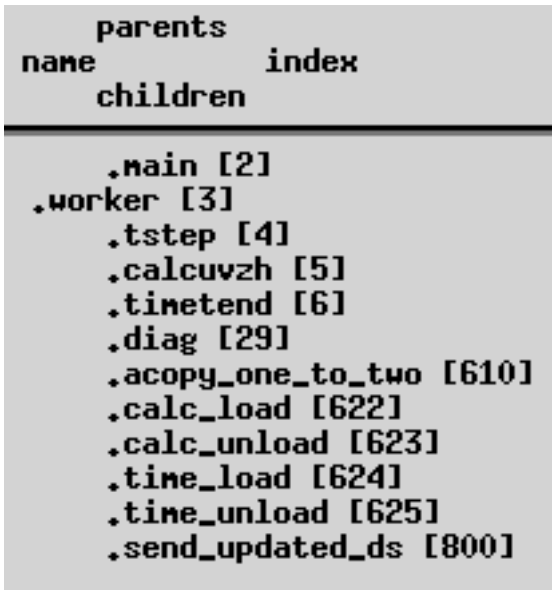


Figure 105. The name/index/parents/children field of the Call Graph Profile window

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

index
The index of the function in the Call Graph Profile report. This number corresponds to the index that appears in the index column of the Call Graph Profile report and on the function box labels in the function call tree.

parents
The parents of the function. A parent is any function that directly calls the function in which you are interested.

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

children
The children of the function. A child is any function that is directly called by the function in which you are interested.

Understanding the function index report

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

Unless you specified the **-z** flag, the Function Index report does not include functions that have no CPU usage and no call counts.

Like the **Flat Profile** menu option, the **Function Index** menu option includes a **Code Display** menu, so you can view source code or disassembler code. See "Looking at your code" on page 182 for more information.

The Function Index report looks similar to Figure 106:

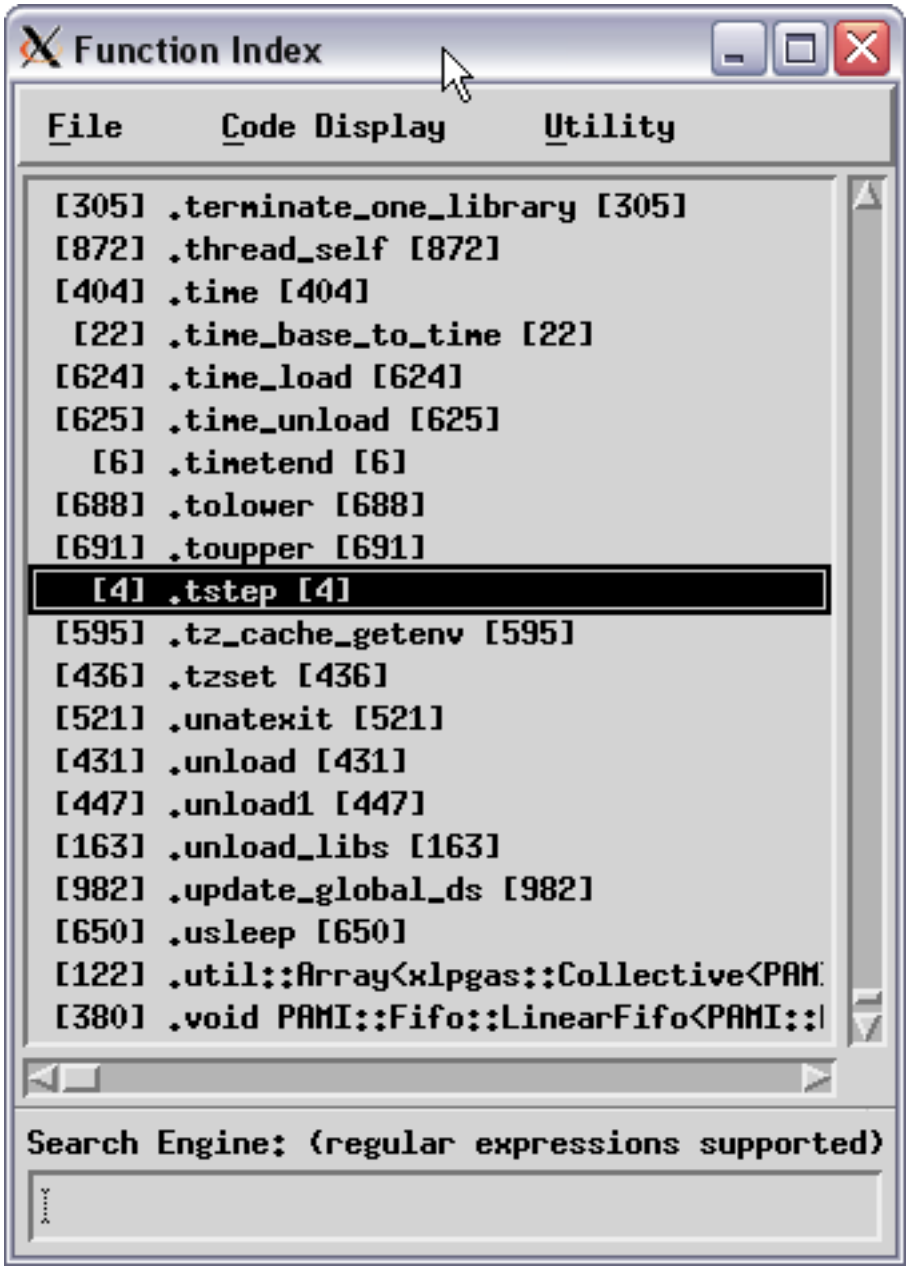


Figure 106. The Function Index window

Understanding the function call summary report

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

The Function Call Summary report looks similar to Figure 107:

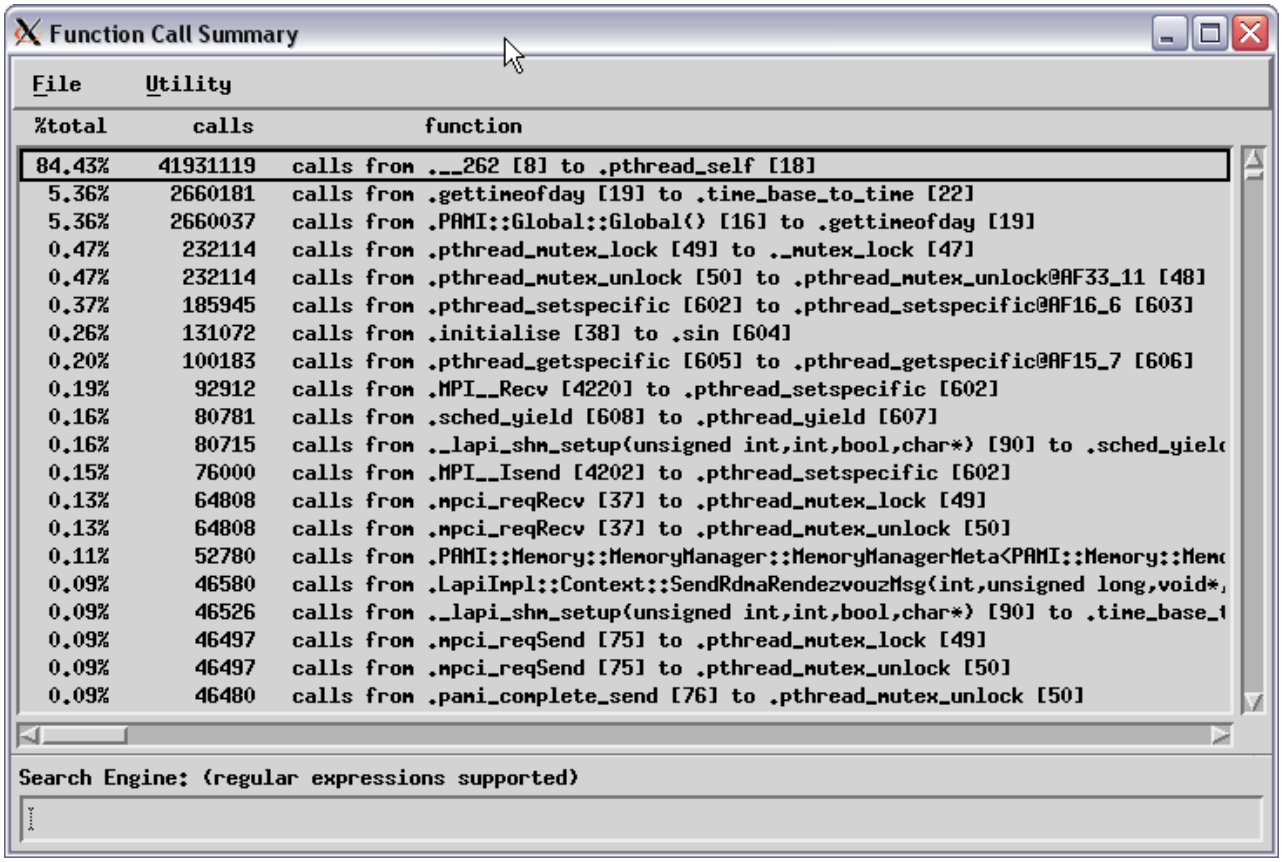


Figure 107. The Function Call Summary window

The **Function Call Summary** window contains the following fields:

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

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

function
The name of the caller function and the callee function

Understanding the library statistics report

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

The Library Statistics report looks similar to Figure 108 on page 180:

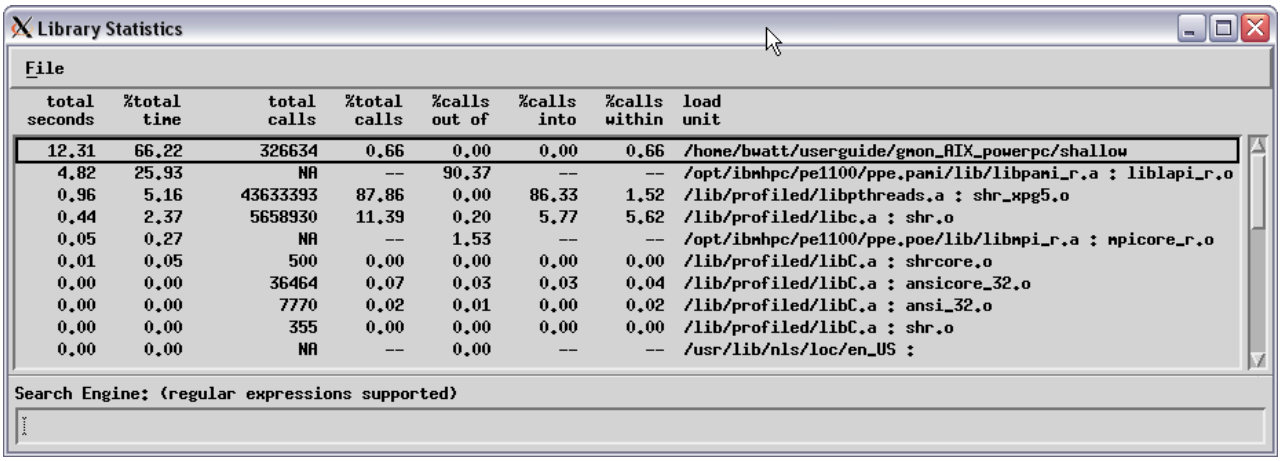


Figure 108. The Library Statistics window

The **Library Statistics** window contains the following fields:

total seconds
The total CPU usage of the library, in seconds.

%total time
The percentage of the total CPU usage that was consumed by this library.

total calls
The total number of calls that this library generated.

%total calls
The percentage of the total calls that this library generated.

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

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

%calls within
The percentage of the total number of calls made between the functions within this library.

load unit
The library's full path name.

Saving reports to a file

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

Saving a single report

To save a single report, go to the **Report** menu on the Xprof main window and select the report you want to save. Each report window includes a **File** menu. Select the **File** menu and then the **Save As** option to save the report. A save dialog appears, which is named according to the report from which you selected the **Save As** option. For example, if you chose **Save As** from the **Flat Profile** window, the save window is named **Save Flat Profile Dialog**.

Saving the call graph profile, function index, and flat profile reports to a file

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

To save the reports, do the following:

1. Specify the file into which the profiled data should be placed. You can specify either an existing file or a new one. To specify an existing file, use the scroll bars of the **Directories** and **Files** selection boxes to locate the file. To make locating your files easier, you can also use the **Filter** button (see "Filtering what you see" on page 155 for more information). To specify a new file, type its name in the **Selection** field.
2. Click **OK**. A file that contains the profiled data appears in the directory you specified, under the name you gave it.

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

The **File Selection** fields of the save dialogs follow Motif standards.

Saving summarized data from multiple profile data files

If you are profiling a parallel program, you can specify more than one profile data (**gmon.out**) file when you start Xprof. The **Save gmon.sum As** option of the **File** menu lets you save a summary of the data in each of these files to a single file.

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

To create a summary file, do the following:

1. Select the **File** menu, and then the **Save gmon.sum As** option. The **Save gmon.sum Dialog** appears.
2. Specify the file into which the summarized, profiled data should be placed. By default, Xprof puts the data into a file called **gmon.sum**. To specify a new file, type its name in the **Selection** field. To specify an existing file, use the scroll bars of the **Directories** and **Files** selection boxes to locate the file you want. To make locating your files easier, you can also use the **Filter** button (see "Filtering what you see" on page 155 for information).
3. Click **OK**. A file that contains the summary data appears in the directory you specified, under the name you specified.

Saving a configuration file

The **Save Configuration** menu option lets you save the names of the functions that are displayed currently to a file. Later, in the same Xprof session or in a different

session, you can read this configuration file in using the **Load Configuration** menu option. For more information, see "Loading a configuration file."

To save a configuration file, do the following:

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

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

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

Loading a configuration file

The **Load Configuration** menu option lets you read in a configuration file that you saved. See "Saving a configuration file" on page 181 for more information. The **Load Configuration** menu option automatically reconstructs the function call tree according to the function names recorded in the configuration file.

To load a configuration file, do the following:

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

   You can also load the file with the **-c** flag. For more information, see Table 14 on page 136.
2. Select a configuration file from the dialog's **Files** list or specify a file name in the **Selection** field and click **OK**. The function call tree is redrawn to show only those function boxes for functions that are listed in the configuration file and are called within the program that is currently represented in the display. All corresponding call arcs are also drawn.

   If the **a.out** name, that is, the program name in the configuration file, is different from the **a.out** name in the current display, a confirmation dialog asks you whether you still want to load the file.
3. If after loading a configuration file, you want to return the function call tree to its previous state, select the **Filter** menu, and then the **Undo** option.

Looking at your code

Xprof provides several ways for you to view your code. You can view the source code or the disassembler code for your application, for each function. This also applies to any included function code that your application might use.

To view source or included function code, use the **Source Code** window. To view disassembler code, use the **Disassembler Code** window. You can access these windows through the **Report** menu of the Xprof GUI or the **Function** menu of the function you are interested in.

Viewing the source code

Both the **Function** menu and **Report** menu permit you to access the **Source Code** window, from which you can view your code.

To access the **Source Code** window through the **Function** menu, click the function box you are interested in with the right mouse button. The **Function** menu appears.

From the **Function** menu, select the **Show Source Code** option. The **Source Code** window appears.

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

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

Using the source code window

The **Source Code** window shows you the source code file for the function you specified from the **Flat Profile** window or the **Function** menu. The **Source Code** window looks similar to Figure 109 on page 184:

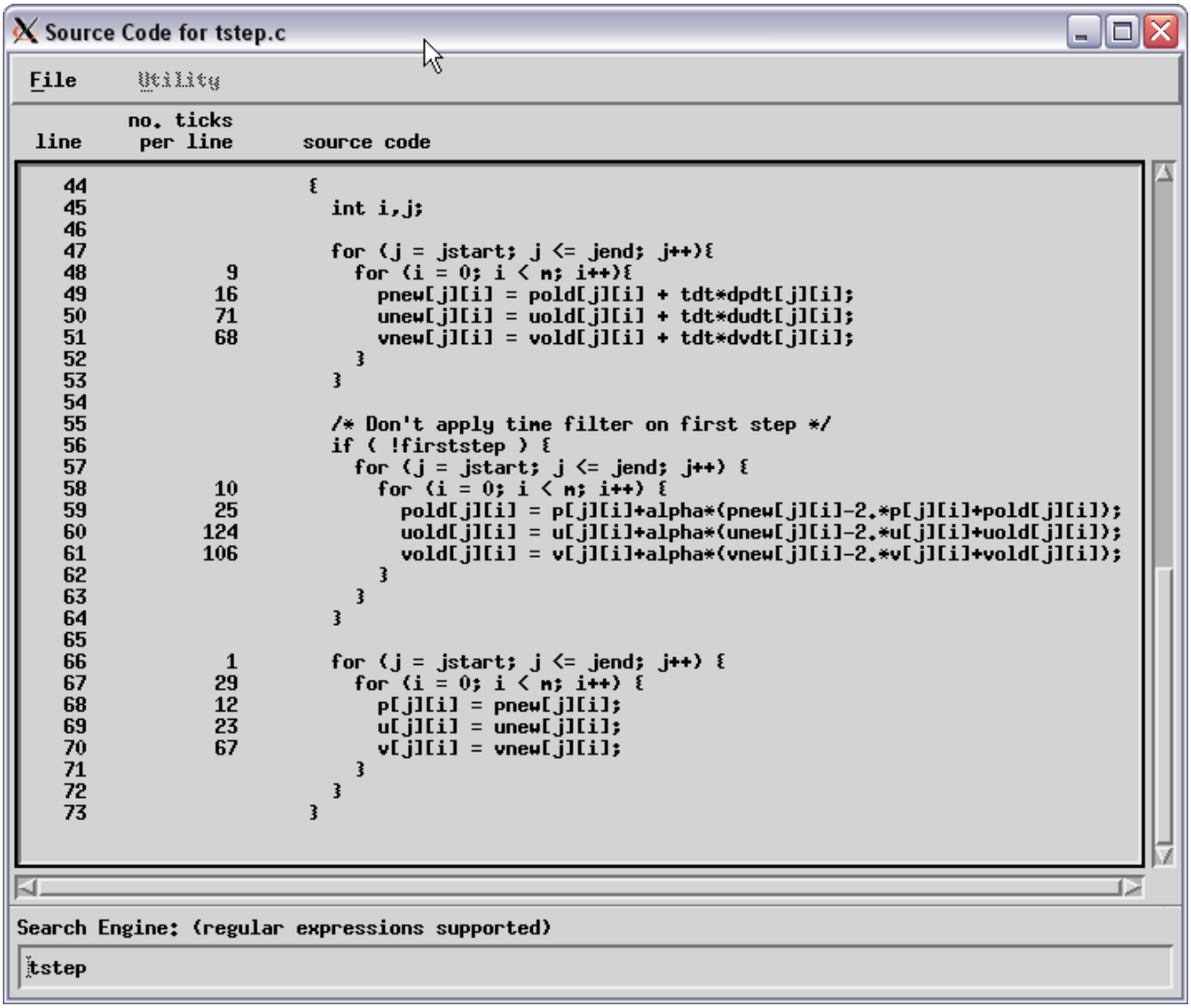


Figure 109. The Source Code window

The **Source Code** window contains information in the following fields:

line
The source code line number.

no. ticks per line
Each tick represents .01 seconds of CPU time used. The value in this field represents the number of ticks used by the corresponding line of code. For example, if the number 3 appeared in this field, for a source statement, this source statement would have used .03 seconds of CPU time. The CPU usage data only appears in this field if you used the **-g** flag when you compiled your application. Otherwise, this field is blank.

source code
The application's source code.

The **Source Code** window contains the following menus:

File
The **Save As** option lets you save the annotated source code to a file. When you select this option, the **Save Source Code Dialog** appears. For more

information about using the **Save Source Code Dialog**, see "Saving the call graph profile, function index, and flat profile reports to a file" on page 181.

To close the **Source Code** window, select the **Close** option from the **File** menu.

Utility
This menu contains the **Show Included Functions** option.

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

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

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

Viewing the disassembler code

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

To access the **Disassembler Code** window through the **Function** menu, do the following:

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

To access the **Disassembler Code** window through the **Report** menu, do the following:

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

Using the disassembler code window

The **Disassembler Code** window shows you only the disassembler code for the function you specified by using the **Report** menu or the **Function** menu. The **Disassembler Code** window looks similar to Figure 110 on page 186:

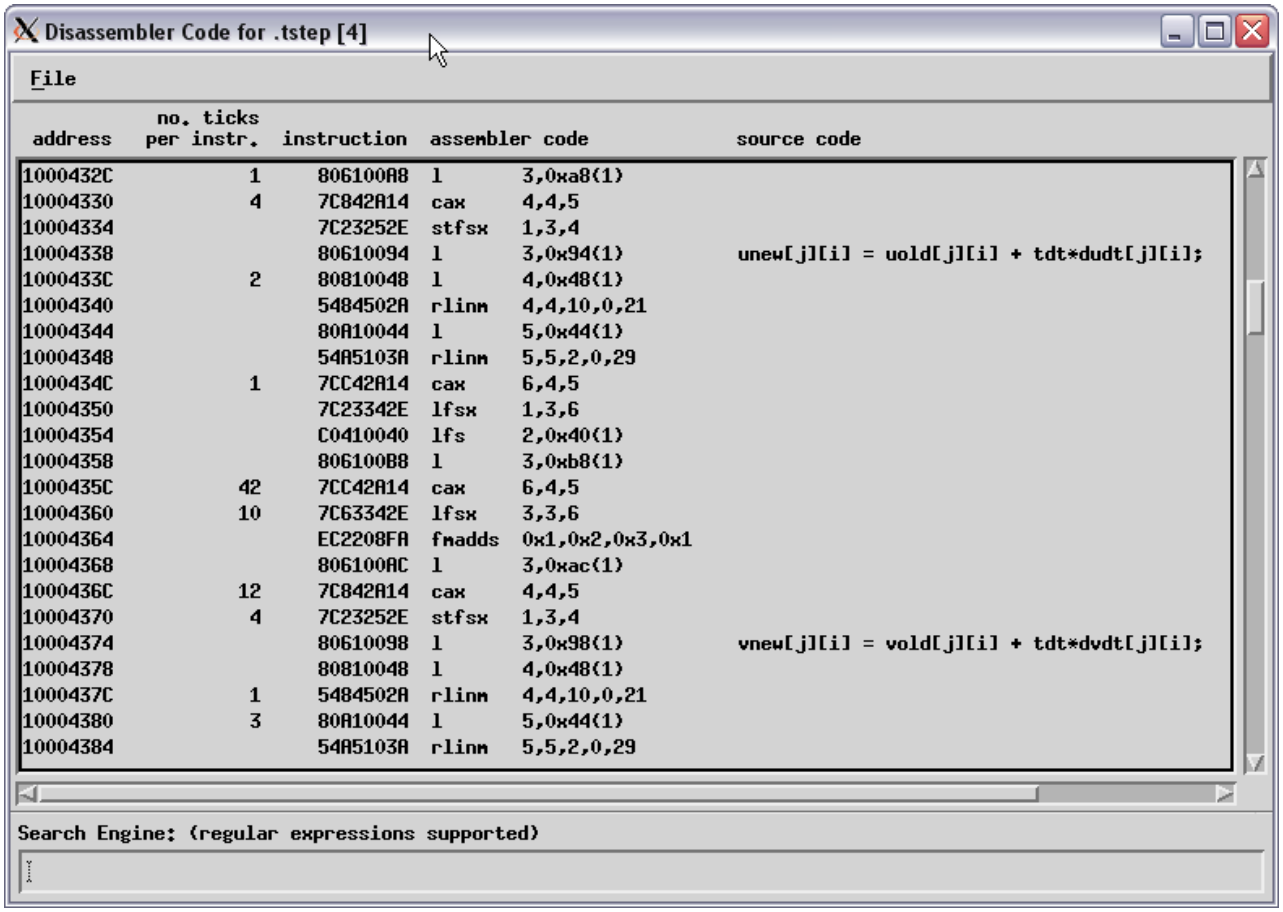


Figure 110. The Disassembler Code window

The **Disassembler Code** window contains information in the following fields:

address
The address of each instruction in the function you selected (from the **Flat Profile** window, the **Function Index** window, or the function call tree).

no. ticks per instr.
Each tick represents .01 seconds of CPU time used. The value in this field represents the number of ticks used by the corresponding instruction. For instance, if the number 3 appeared in this field, this instruction would have used .03 seconds of CPU time.

instruction
The hexadecimal representation of the execution instruction.

assembler code
The execution instruction's corresponding assembler code.

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

The **Search Engine** field at the bottom of the **Disassembler Code** window lets you search for a specific string in your disassembler code.

The **Disassembler Code** window contains one menu:

File

Select **Save As** to save the annotated disassembler code to a file. When you select this option, the **Save Disassembler Code Dialog** appears. For information on using the **Save Disassembler Code Dialog**, see "Saving the call graph profile, function index, and flat profile reports to a file" on page 181.

To close the **Disassembler Code** window, select **File** and then **Close**.

Saving screen images of profiled data

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

To capture a window image, do the following:

1. Select **File** and then **Screen Dump**. **The Screen Dump** menu opens.
2. From the **Screen Dump** menu, select **Set Options**. The **Screen Dump Options Dialog** (Figure 111 on page 188) is displayed:

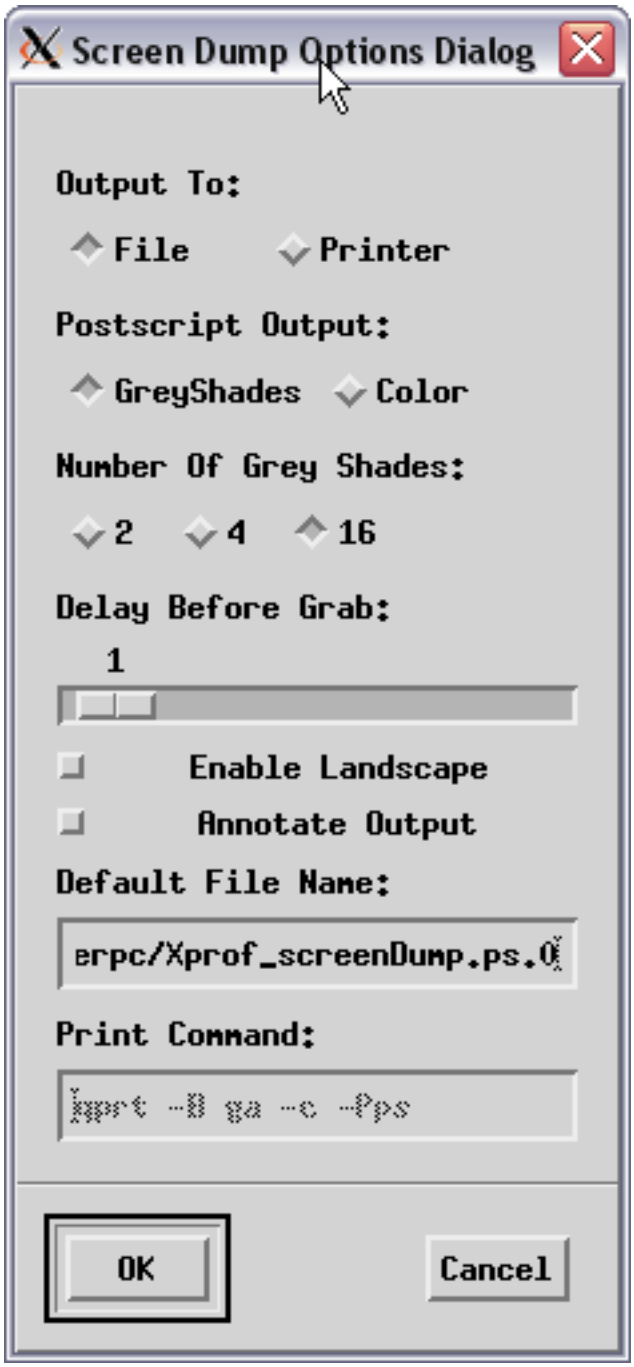


Figure 111. The Screen Dump Options dialog

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

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

   If you would like to save the image to a file, select the **File** button. This file, by default, is named **Xprof_screenDump.ps.0**, and is displayed in the **Default File Name** field of this dialog. When you select the **File** button, the text in the **Print Command** field greys out.

To send the image directly to a printer, select the **Printer** button. The image is sent to the printer you specify in the **Print Command** field of this dialog. When you specify the **Printer** option, a file of the image is not saved. Also, selecting this option causes the text in the **Default File Name** field is made unavailable.

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

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

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

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

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

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

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

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

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

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

4. Click **OK**. The **Screen Dump Options Dialog** closes.

After you have set your screen dump options, you need to select the window, or portion of a window, you want to capture. From the Screen Dump menu, select the **Select Target Window** option. A cursor that looks like a person's hand appears after the number of seconds you specified. To cancel the capture, click the right mouse button. The hand-shaped cursor will revert to normal and the operation will be terminated.

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

To capture a portion of the Xprof window, do the following:

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

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

Customizing Xprof resources

You can customize certain features of an X-Windows application. For example, you can customize its colors, fonts, and orientation. This topic lists each of the resource variables you can set for the application.

Customize a resource by assigning a value to a resource name in a standard X-Windows format. The suggested method is to put the resource settings into the **$HOME/.Xresources** file, then issue the command:

```
xrdb $HOME/.Xresources
```

The resource settings are essentially your personal preferences for how you would like the X-Windows application to look.

For example, to increase the size of the Xprof main window to be larger than the defaults and change the default background color to white, you could put the following in the **$HOME/.Xresources** file:

```
Xprof*mainW.height: 900
Xprof*mainW.width: 1100
Xprof*background: white
```

Setting Xprof resource variables

You can use the following resource variables to control the appearance and behavior of Xprof. The values listed in this section are the defaults; you can change these values to suit your preferences.

Controlling fonts

Note: The following tables show the resource variable name and its default value. You can override the default value by specifying a value of your choice in the **.Xresources** file.

To specify the font for the labels that appear with function boxes, call arcs, and cluster boxes:

Table 17. Specifying Xprof fonts

Use this resource variable:	Default value
*narc*font	fixed

To specify the font used in textual reports:

Table 18. Specifying Xprof report fonts

Use this resource variable:	Default value
Xprof*fontList	rom10

Controlling the appearance of the Xprof main window

To specify the size of the main window:

Table 19. Specifying Xprof main window size

Use this resource variable:	Default value
Xprof*mainW.height	700
Xprof*mainW.width	900

To specify the foreground and background colors of the main window:

Table 20. Specifying Xprof main window colors

Use this resource variable:	Default value
Xprof*foreground	black
Xprof*background	light grey

To specify the number of function boxes that are displayed when you first open the Xprof main window:

Table 21. Specifying the number of function boxes

Use this resource variable:	Default value
Xprof*InitialDisplayGraph	5000

You can use the **-disp_max** flag to override this value.

To specify the colors of the function boxes and call arcs of the function call tree:

Table 22. Specifying function box colors

Use this resource variable:	Default value
Xprof*defaultNodeColor	forest green
Xprof*defaultArcColor	royal blue

To specify the color in which a specified function box or call arc is highlighted:

Table 23. Specifying highlighted function box colors

Use this resource variable:	Default value
Xprof*HighlightNode	red

To specify the color in which de-emphasized function boxes appear:

Table 24. Specifying de-emphasized function box colors

Use this resource variable:	Default value
Xprof*SuppressNode	grey

Function boxes are de-emphasized with the **-e**, **-E**, **-f**, and **-F** flags.

Controlling variables related to the file menu

To specify the size of the **Load Files Dialog**, use the following:

Table 25. Specifying the size of the load files dialog

Use this resource variable:	Default value
Xprof*loadFile.height	785
Xprof*loadFile.width	725

The **Load Files Dialog** is called by the **Load Files** option of the **File** menu.

To specify whether a confirmation dialog box should appear whenever a file will be overwritten:

Table 26. Requesting confirmation before Xprof overwrites files

Use this resource variable:	Default value
Xprof*OverwriteOK	False

The value **True** would be equivalent to selecting the **Set Options** option from the **File** menu, and then selecting the **Forced File Overwriting** option from the **Runtime Options Dialog**.

To specify the alternative search paths for locating source or library files:

Table 27. Specifying the Xprof file search path

Use this resource variable:	Default value
Xprof*fileSearchPath	**.** (refers to the current working directory)

The value you specify for the search path is equivalent to the search path you would designate from the **Alt File Search Path Dialog**. To get to this window, choose the **Set File Search Paths** option from the **File** menu.

To specify the file search sequence (whether the default or alternative path is searched first):

Table 28. Specifying the Xprof file search sequence

Use this resource variable:	Default value
Xprof*fileSearchDefault	True

The value **True** is equivalent to selecting the **Set File Search Paths** from the **File** menu, and then the **Check default path(s) first** option from the **Alt File Search Path Dialog**.

Controlling variables related to the screen dump option

To specify whether a screen dump will be sent to a printer or placed in a file:

Table 29. Specifying Xprof screen capture destination

Use this resource variable:	Default value
Xprof*PrintToFile	True

The value **True** is equivalent to selecting the **File** button in the **Output To** field of the **Screen Dump Options Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify whether the PostScript screen dump will created in color or in shades of grey:

Table 30. Requesting Xprof screen captures in color

Use this resource variable:	Default value
Xprof*ColorPscript	False

The value **False** is equivalent to selecting the **GreyShades** button in the **PostScript Output** area of the **Screen Dump Options Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify the number of grey shades that the PostScript screen dump will include (if you selected **GreyShades** in the **PostScript Output** area):

Table 31. Specifying the number of gray shades for Xprof screen captures

Use this resource variable:	Default value
Xprof*GreyShades	16

The value **16** is equivalent to selecting the **16** button in the **Number of Grey Shades** field of the **Screen Dump Options Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify the number of seconds that Xprof waits before capturing a screen image:

Table 32. Specifying Xprof screen capture delay

Use this resource variable:	Default value
Xprof*GrabDelay	1

The value **1** is the default for the **Delay Before Grab** option of the **Screen Dump Options Dialog**, but you can specify a longer interval by entering a value here. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To set the maximum number of seconds that can be specified with the slider of the **Delay Before Grab** option:

Table 33. Specifying Xprof delay before grab value

Use this resource variable:	Default value
Xprof*grabDelayScale.maximum	30

The value **30** is the maximum for the **Delay Before Grab** option of the **Screen Dump Options Dialog**. This means that users cannot set the slider scale to a value greater than 30. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify whether the screen dump is created in landscape or portrait format:

Table 34. Specifying Xprof screen capture orientation

Use this resource variable:	Default value
Xprof*Landscape	False

The value **True** is the default for the **Enable Landscape** option of the **Screen Dump Options Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify whether you would like information about how the image was created to be added to the PostScript screen dump:

Table 35. Requesting Xprof screen capture annotation

Use this resource variable:	Default value
Xprof*Annotate	False

The value **False** is the default for the **Annotate Output** option of the **Screen Dump Options Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify the directory that will store the screen dump file (if you selected **File** in the **Output To** field):

Table 36. Specifying Xprof screen capture directory

Use this resource variable:	Default value
Xprof*PrintFileName	/tmp/Xprof_screenDump.ps.0

The value you specify is equivalent to the file name you would designate in the **File Name** field of the **Screen Dump Dialog**. You access the **Screen Dump Options Dialog** by selecting **Screen Dump** and then **Set Option** from the **File** menu.

To specify the printer destination of the screen dump (if you selected **Printer** in the **Output To** field):

Table 37. Specifying Xprof screen capture printer

Use this resource variable:	Default value
Xprof*PrintCommand	**qprt -B ga -c -Pps**

Controlling variables related to the view menu

To specify the size of the **Overview** window:

Table 38. Specifying Xprof overview window size

Use this resource variable:	Default value
Xprof*overviewMain.height	300
Xprof*overviewMain.width	300

To specify the color of the highlight area of the **Overview** window:

Table 39. Specifying Xprof overview window highlight area color

Use this resource variable:	Default value
Xprof*overviewGraph*defaultHighlightColor	sky blue

To specify whether the function call tree is updated as the highlight area is moved (immediate) or only when it is stopped and the mouse button released (delayed):

Table 40. Control Xprof highlight area tracking

Use this resource variable:	Default value
Xprof*TrackImmed	True

The value **True** is equivalent to selecting the **Immediate Update** option from the **Utility** menu of the **Overview** window. You access the **Overview** window by selecting the **Overview** option from the **View** menu.

To specify whether the function boxes in the function call tree appear in two-dimensional or three-dimensional format:

Table 41. Specify Xprof function box format

Use this resource variable:	Default value
Xprof*Shape2D	True

The value **True** is equivalent to selecting the **2-D Image** option from the **View** menu.

To specify whether the function call tree appears in top-to-bottom or left-to-right format:

Table 42. Specifying Xprof function call tree layout

Use this resource variable:	Default value
Xprof*LayoutTopDown	True

The value **True** is equivalent to selecting the **Layout: Top** and **Bottom** option from the **View** menu.

Controlling variables related to the filter menu

To specify whether the function boxes of the function call tree are clustered or unclustered when the Xprof main window is first opened:

Table 43. Specifying Xprof function box clustering

Use this resource variable:	Default value
Xprof*ClusterNode	True

The value **True** is equivalent to selecting the **Cluster Functions by Library** option from the **Filter** menu.

To specify whether the call arcs of the function call tree are collapsed or expanded when the Xprof main window is first opened:

Table 44. Specifying Xprof function call arc expansion

Use this resource variable:	Default value
Xprof*ClusterArc	True

The value **True** is equivalent to selecting the **Collapse Library Arcs** option from the Filter menu.

Part 6. The hardware performance counter tools

The topics in this part provide information about using the hardware performance counter tools.

© Copyright IBM Corp. 2008, 2013

Chapter 14. Using the hardware performance counter tools

The IBM HPC Toolkit provides a command-line tool called **hpccount** and a library called **libhpc** that access hardware performance counters to help you analyze your application's performance. You can use the **hpccount** command to report hardware performance counter measurements for your entire application. You can obtain measurements from a single hardware counter group. You can multiplex multiple groups of hardware counters so that you can get an estimate of hardware performance counter events for multiple groups in a single run of your application. The **hpccount** command also can report derived metrics, which are additional measurements computed from hardware performance counter measurements to help you better understand your application's performance.

You can use the **hpcstat** command to obtain overall system statistics for hardware performance counters. The **hpcstat** command requires root access in order to obtain system-wide statistics.

You can use the **libhpc** library to make more precise measurements of hardware performance counter events by placing calls to regions of code in your application that are of interest. The **libhpc** library provides the same features as the **hpccount** command. In addition, this library supports the use of plug-ins to aggregate or reduce hardware performance counter measurements made in multiple tasks of an MPI application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the hardware performance counter tools. In order to set these environment variables, you should run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

Using the hpccount command

The **hpccount** command is essentially used the same way as the **time** command; in the simplest invocation, the user types:

```
hpccount program
```

If you are using the **hpccount** command to measure the performance of a parallel program, you should invoke **hpccount** as **poe hpccount** *program*. If you invoke **hpccount** as **hpccount poe** *program*, you will measure the performance of the poe command, and not the performance of your application.

As a result, **hpccount** appends various performance information at the end of the screen (in other words, to **stdout**). In particular, it prints resource usage statistics, hardware performance counter information, and derived hardware metrics.

The resource usage statistics are directly taken from a call to **getrusage**(). For more information on the resource usage statistics, refer to the **getrusage** man pages. In particular, the Linux man page for **getrusage**() states that not all fields are meaningful under Linux. The corresponding lines in **hpccount** output are not provided.

© Copyright IBM Corp. 2008, 2013

If you specify the **–l** flag, the **hpccount** command displays a list of the hardware performance counter groups that are available on the processor from which you invoked the **hpccount** command.

If you specify the **–g** flag, you can specify the hardware performance counter group from which you want to count events from. If you do not specify the **–g** flag, the **hpccount** command uses a default hardware counter group (for more information, see "hpccount - Report hardware performance counter statistics for an application" on page 236). If you specify a comma-separated list of hardware performance counter groups, **hpccount** multiplexes the use of the specified hardware performance counter groups in your application process. See "Understanding CPU hardware counter multiplexing" on page 203 for more information.

You can obtain derived metric descriptions for your application by using the **–x** flag. Derived metrics are additional performance metrics computed from the hardware performance counter measurements you collected. See Appendix B, "Derived metrics descriptions," on page 315 for more information.

You can specify a data aggregation plug-in using the **–a** flag or the **HPM_AGGREGATE** environment variable. See "Considerations for MPI programs" on page 207 for more information.

If you plan to view the data obtained by running the **hpccount** command with the Eclipse plug-in or **peekperf**, you should make sure that the **HPM_VIZ_OUTPUT** environment variable is set to **yes**.

Using the hpcstat command

The **hpcstat** is a simple system-wide monitor that is based on hardware performance counters. To run **hpcstat**, root privileges are required. The usage is very similar to that of the **vmstat** command. You can invoke **hpcstat** by issuing:

```
hpcstat
```

If you specify the **–l** flag, the **hpcstat** command displays a list of the hardware performance counter groups that are available on the processor on which you invoked the command.

If you specify the **–g** flag, you can specify the hardware performance counter group from which you want to count events. If you do not specify the **–g** flag, the **hpcstat** command will use a default hardware counter group as described in "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244.

You can invoke **hpcstat** so that it periodically queries the requested hardware performance counter groups by using the **-I** flag to specify the number of times to query hardware performance counter groups and using the **-I** flag to specify the interval between queries.

The output of the **hpcstat** command is written to **stdout** and to a file, which consists of hardware performance counter information and derived hardware metrics.

Using the libhpc library

The **hpccount** command provides hardware performance counter information and derived hardware metrics for the whole program. If this information is required for only part of the program, instrumentation with the **libhpc** library is required. This library provides a programming interface to start and stop performance counting within an application program.

The **libhpc** library API includes the following function calls:

- **hpmInit**() for initializing the instrumentation library.
- **hpmTerminate**() for generating the reports and visualization data files and shutting down the **libhpc** environment.
- **hpmStart**() for identifying the start of a region of code in which hardware performance counter events will be counted.
- **hpmStop**() for identifying the end of the instrumented region.

The **libhpc** library provides variants of the **hpmStart**() and **hpmStop**() function calls, which you can use in threaded code, and where you need to explicitly identify parent/child relationships between nested instrumentation regions. **libhpc** implements both C and FORTRAN versions of each function call.

The **hpmStart**() and **hpmStop**() function calls, or their variants, must be executed in pairs, where for each **hpmStart**() function call, a corresponding **hpmStop**() function call must be executed.

The part of the application program between the start and stop of performance counting is called an instrumentation region. You assign a unique integer number as the region identifier. You specify this region identifier in the call to the **hpmStart**() function. A simple case of an instrumented program region might look similar to the following:

```
hpmInit( 0, "my program" );
hpmStart( 1, "outer call" );
do_work();
hpmStart( 2, "computing meaning of life" );
do_more_work();
hpmStop( 2 );
hpmStop( 1 );
hpmTerminate( 0 );
```

Calls to **hpmInit**() and **hpmTerminate**() surround the instrumented region. Every instrumentation region starts with **hpmStart**() and ends with **hpmStop**(). The region identifier is the first parameter to the latter two functions. As shown in the example, **libhpc** supports multiple instrumentation regions and overlapping instrumentation regions. Each instrumented region can also be called multiple times. When **hpmTerminate**() is encountered, the counted values are collected and printed or written to performance data files.

The example program above provides an example of two properly nested instrumentation regions. For region 1 we can consider the *exclusive* time and *exclusive* counter values. By that we mean the difference of the values for region 1 and region 2. The original values for region 1 would be called *inclusive* for matter of distinction. The terms *inclusive* and *exclusive* for the embracing instrumentation region are chosen to indicate whether counter values and times for the contained regions are included or excluded. For more details see “Understanding inclusive and exclusive event counts” on page 205.

Any C source file containing calls to any function contained in the **libhpc** library should include the **libhpc.h** header. FORTRAN source files containing calls to functions in the **libhpc** library should include the **f_hpc.h** header. Or, if the FORTRAN source file is compiled using the **–qintsize=8** compiler flag, it should include the **f_hpc_i8.h** header file. All of these header files are located in the **${IHPCT_BASE}/include** directory.

FORTRAN source files that include either the **f_hpc.h** or **f_hpc_i8.h** header file should also be processed by the C pre-processor before compilation, for instance by specifying the **–qsuffix=cpp=f** compiler flag. Another option is to use the **.F** Fortran source file extension.

You must link your application with the **libhpc** library, using the **–lhpc** linking option. You must also link your application with the **libm** library, using the **–lm** linking option. On AIX systems, you must also link with the **PMAPI** library, using the **–lpmapi** linking option. When using the **libhpc** library, compile and link your application as a threaded program (for instance, using the **xlc_r** or **xlf_r** commands), or link with the **pthreads** library using the **-lpthreads** linking option. When linking **libhpc** with your application, you must specify the correct library, using either the **–L${IHPCT_BASE}/lib** or **–L${IHPCT_BASE}/lib64** linking option.

A sample compilation and link for AIX is:

```
xlc -o testprog -g -q64 -I$IHPCT_BASE/include testprog.c -L$IHPCT_BASE/lib64 -lhpc \
 -lpmapi -lpthreads
```

A sample compilation and link for pLinux is:

```
xlc -o testprog -g -q64 -I$IHPCT_BASE/include testprog.c -L$IHPCT_BASE/lib64 -lhpc
```

A sample compilation and link for xLinux is:

```
cc -o testprog -g -m64 -I$IHPCT_BASE/include testprog.c -L$IHPCT_BASE/lib64 -lhpc
```

Like the **hpccount** command, you can obtain hardware performance counter measurements using multiple hardware performance groups as described in “Understanding CPU hardware counter multiplexing” on page 203. Derived metrics will be computed if the events within the group you selected contain the events necessary to compute the derived metric. If you are instrumenting an MPI program, you can use plug-ins to aggregate performance data from multiple tasks or to filter that data. See “Considerations for MPI programs” on page 207 for more detail about these plug-ins.

Note that **libhpc** collects information and performs summarization during run time. Thus, there could be a considerable overhead if instrumentation regions are inserted inside inner loops. The **libhpc** library uses the same set of hardware performance counter groups used by **hpccount**.

If an error occurs internally in **libhpc**, the program is not automatically terminated. Instead, the **libhpc** library sets an error indicator and lets the user handle the error. For details, see “hpm_error_count, f_hpm_error - Verify a call to a libhpc function” on page 263.

Understanding CPU hardware counter multiplexing

The idea behind multiplexing is to run several CPU hardware counter groups concurrently. This is accomplished by running the first CPU group for a short time interval, then switching to the next CPU group for the next short time interval. This is repeated in a round-robin fashion for the CPU groups until the event counting is eventually stopped.

Hardware Performance Monitor (HPM) supports multiplexing by specifying CPU groups as a comma separated list. If you are using the **hpccount** command, then you can specify the multiplexed hardware performance CPU counter groups one of two ways, for instance:

```
hpccount -g 1,2 program
```

or:

```
export HPM_EVENT_SET='1,2'
hpccount program
```

If you are running an application program that has been compiled and linked with **libhpc**, specify the multiplexed CPU hardware counter groups by setting the **HPM_EVENT_SET** environment variable before running your application. For example:

```
export HPM_EVENT_SET='1,2'
```

Multiplexing means that none of the specified CPU groups has been run on the whole code, and it is unknown what fraction of the code was measured with which group. It is assumed that the workload is sufficiently uniform that the measured event counts can be (more or less) safely calibrated as if the groups have been run separately on the whole code.

On AIX, the default time interval for measuring one CPU group on the application is 100 milliseconds. If the total execution time is shorter than this time interval, only the first group in the list is measured. All other counter values from the other CPU hardware counter groups are 0 in this case. This might result in NaNQ values for some derived metrics (see Appendix B, "Derived metrics descriptions," on page 315), if the formula for computing the derived metric requires division by the counter which has a zero value.

On AIX, the duration of this time interval can be controlled by setting the following environment variable.

```
export HPM_SLICE_DURATION=integer value in ms
```

This value is passed directly to AIX (more precisely to the **bos.pmapi** component of AIX). AIX requires the value to be between 10 milliseconds and 30 seconds.

On Linux, the kernel chooses the intervals used when multiple CPU groups are specified.

The data for each CPU group is printed in a separate section, with separate timing information, and is written separately to the visualization data files used as input to **peekperf**.

For MPI applications, the form of the output depends on the chosen aggregation plug-in as described in "Considerations for MPI programs" on page 207. Without specifying an aggregator plug-in (in other words, with the default plug-in), the

data for each CPU hardware performance counter group is printed in a separate section with separate timing information for each CPU hardware performance counter group. To combine the data from the specified groups in to one big group with more counters, use the local merge aggregator plug-in (**loc_merge.so**), which is described in "Plug-ins shipped with the tool kit" on page 208.

Understanding derived metrics

Some of the hardware counter events are difficult to interpret. Sometimes a combination of events provides better information. Such a recombination of basic events will be called a derived metric. HPM also provides a list of derived metrics, which are defined in Appendix B, "Derived metrics descriptions," on page 315.

Since each derived metric has its own set of ingredients, not all derived metrics are printed for each group. HPM automatically finds those derived metrics that are computable and prints them. As a convenience to the user, the **-x** flag will print the value of the derived metric and its definition. If the **HPM_PRINT_FORMULA** environment variable is set to **yes**, derived metric formulas are also printed.

Understanding MFlop issues

For POWER7® servers, scalar FLOPS and scalar utilization rate are computed as derived metrics when the default hardware counter group is selected. There is no hardware counter group that includes all event counters for vector floating point operations, so a vector FLOPS rate cannot be computed automatically. Use groups containing event counters PM_VSU_1FLOP, PM_VSU_2FLOP, PM_VSU_4FLOP, PM_VSU_8FLOP, PM_VSU_16FLOP and PM_VSU_2FLOP_DOUBLE to compute vector FLOPS. Note that when computing vector FLOPS, you must multiply the event count by the number in the counter name, for instance multiply PM_VSU_2FLOP by 2 to get the correct count.

Understanding inheritance

On AIX and Linux, the *counter virtualization* and the group (in other words, set of events) that is actually monitored is inherited from the process by any of its children. Children in this context mean threads or processes spawned by the parent process. AIX and Linux differ in the ability of the parent process to access the counter values of its children.

- On AIX, all counter values of a process group can be collected.
- On Linux, counter values are only available to the parent if the child has exited.

The **hpccount** utility makes use of this inheritance. If **hpccount** is called for a program, the returned counter values are the sum of the counter values of the program and all of the threads and processes spawned by it at the time the values are collected. For Linux, this has to be restricted to the sum of counter values of all children that have finished at the time the values are collected. Even the latter is enough to catch the values of all threads of an OpenMP program.

Suppose you are using a small program named **taskset** to bind threads to CPUs. When **hpccount** is invoked to run **taskset** as follows:

```
hpccount taskset -g num program_name
```

hpccount would first enable hardware event counting for the application **taskset**. This command then spawns the program *program name*, which inherits all hardware counter settings. At the end **hpccount** would print counter values (and derived metrics) based on the sum of events for *taskset* and the called program. Because **taskset** is a very small application, the **hpccount** results would mostly represent the performance of the program *program name*, which is what the user intended.

Understanding inclusive and exclusive event counts

For an example of an application fragment, where the term *exclusive values* applies, see "Using the libhpc library" on page 201. That application fragment provides an example of two properly nested instrumentation regions. For region 1, exclusive time and exclusive counter values are the difference between the values for region 1 and region 2, or excluding the counts of events within the scope of region 2. The original values for region 1 would be called inclusive values since those values also include the count of events that occurred within the scope of region 2. The terms inclusive and exclusive for the enclosing instrumentation region are chosen to indicate whether counter values and times for the contained regions are included or excluded.

The extra computation of exclusive values generates overhead that is not always wanted. Therefore, the computation of exclusive values is only carried out if the environment variable **HPM_EXCLUSIVE_VALUES** is set to **yes** or if the **HPM_ONLY_EXCLUSIVE** parameter is used as described in "Understanding parent-child relationships." The exact definition of *exclusive* is based on parent-child relationships among the instrumented regions. Roughly spoken, the exclusive value for the parent is derived from the inclusive value of the parent reduced by the inclusive value of all children.

Understanding parent-child relationships

The IBM HPC Toolkit provides an automatic search for parents, which is supposed to closely mimic the behavior for strictly nested instrumented regions. For strictly nested instrumented sections, the call to **hpmStart**() or **hpmTstart**() for the parent must occur prior to the corresponding call for the child. In a multithreaded environment, however, this causes problems if the children are executed on different threads. In a kind of race condition, a child might mistake its brother for its father. This generates flawed parent child relationships, which change with every execution of the program. To avoid the race condition safely, the search for a parent region is restricted to calls from the same thread only, because only these exhibit a race condition free call history. The parent region found in this history is the last call of the same kind (in other words, both were started with **hpmStart**() or both were started with **hpmTstart**() or their corresponding FORTRAN equivalents) that has not posted a matching **hpmStop**() or **hpmTstop**() meanwhile. If no parent is found that matches these rules, the child is declared an orphan. Therefore, automatic parent child relations are never established across different threads.

There might be situations in which the automatic parent child relations prove unsatisfactory. To help this matter, calls are provided in the HPM API to enable you to establish the relations of your choice. These functions are **hpmStartx**() and **hpmTstartx**() and their FORTRAN equivalents. The first two parameters of this function are the ID of the instrumented section and the ID of the parent instrumented section.

The user has the following choices for the parent ID:

HPM_AUTO_PARENT
Triggers the automatic search and is equivalent to the **hpmStart**() and **hpmTstart**() functions.

HPM_ONLY_EXCLUSIVE
This is essentially the same as **HPM_AUTO_PARENT**, but sets the exclusive flag to **true** on this instance only. The environment variable **HPM_EXCLUSIVE_VALUES** sets this flag globally for all instrumented sections.

HPM_NO_PARENT
This suppresses any parent child relations.

An integer
This must be the ID of an instrumented section with the following restrictions:

- It has to be active when this call to **hpmStartx**() or **hpmTstartx**() is made.
- It has to be of the same kind (in other words, both were started with **hpmStart**() or both were started with **hpmTstart**() or their corresponding FORTRAN equivalents).

Handling of overlap issues

As you can establish almost arbitrary parent-child relationships, the definition of the explicit duration or explicit counter values is not obvious.

Each instrumented section can be represented by the corresponding subset of the time line of application execution. Actually this subset is a finite union of intervals with the left or lower boundaries marked by calls to **hpmStart[x]**/**hpmTstart[x]**() and the right or upper boundaries marked by calls to **hpmStop**()/**hpmTstop**(). The duration is the accumulated length of this union of intervals. The counter values are the number of those events that occur within this subset of time.

The exclusive times and values are the times and values when no child has a concurrent instrumented section. Hence, the main step in defining the meaning of exclusive values is defining the subset of the time line with which they are associated. This is done in several steps:

- Represent the parent and every child by the corresponding subset of the time line (henceforth called the parent set and the child sets).
- Take the union of the child sets.
- Reduce the parent set by the portion that is overlapping with this union.
- Take the difference of the parent set with the union of the child sets, using set theoretic terms.

The exclusive duration is the accumulated length of the resulting union of intervals. The exclusive counter values are the number of those events that occur within this subset of time.

Understanding measurement overhead

Instrumentation overhead is caught by calls to the wall clock timer at entry and exit of calls to **hpmStart[x]**(), **hpmStop**(), **hpmTstart[x]**(), **hpmTstop**(). The accumulated instrumentation overhead for each instrumented section is printed in the ASCII output (***.txt**) file.

Based on the magnitude of the overhead, you can decide what to do with this information.

- If the overhead is several orders of magnitude smaller than the total duration of the instrumented section, you can safely ignore the overhead timing.
- If the overhead is the same order of magnitude as the total duration of the instrumented section, the results might be inaccurate since the instrumentation overhead is a large part of the collected event counts.
- If the overhead is within 20% of the measured wall clock time, a warning is printed to the ASCII output file.

To make the use of **libhpc** thread safe, mutexes are set around each call to **hpmStart[x]**(), **hpmStop**(), **hpmTstart[x]**(), **hpmTstop**(), which adds to the measurement overhead. Results are unpredictable if your program is using multiple threads and you set this environment variable.

Handling multithreaded program instrumentation issues

When placing instrumentation inside of parallel regions, you should use different ID numbers for each thread, as shown in the following FORTRAN example:

```
!$OMP PARALLEL
!$OMP&PRIVATE (instID) instID = 30+omp_get_thread_num()
call f_hpmtstart( instID, "computing meaning of life" )
!$OMP DO
do ...
do_work()
end do
call f_hpmtstop( instID )
!$OMP END PARALLEL
```

If two threads are using the same ID numbers when calling **hpmTstart**() or **hpmTstop**(), **libhpc** exits with the following error message:

```
HPM ERROR - Instance ID on wrong thread
```

If you place instrumentation calls in parallel loops or parallel regions, use the **hpmTstart**() and **hpmTstop**() function calls. If you use **hpmStart**() and **hpmStop**() function calls, **libhpc** attempts to count hardware performance counter events on all threads in the application (AIX only). Since application threads are not necessarily all executing the same code, the event counts obtained by calls to **hpmStart**() and **hpmStop**() might not be accurate.

Considerations for MPI programs

The follow topics discuss considerations for MPI programs.

General considerations

The **libhpc** library is inherently sequential, looking only at the hardware performance counters of a single process (and its children, as explained in "Understanding inheritance" on page 204). When the application is started, each MPI task is doing its own hardware performance counting and these instances are completely ignorant of each other, unless additional action is taken as described in the following subtopics. Consequently, each instance is writing its own output. For information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

For this reason, the environment variable **HPM_AGGREGATE** triggers some aggregation before (possibly) restricting the output to a subset of MPI tasks. The environment variable **HPM_AGGREGATE** takes a value, which is the name of a plug-in that defines the aggregation strategy. Each plug-in is a shared object file containing two functions called **distributor** and **aggregator**.

Hardware performance counter plug-ins

The following topics provide information about the hardware performance counter plug-ins.

Note: Hardware performance counter plug-ins that use MPI calls are only supported when linking your application with **libhpc** or instrumenting your application with the **hpctInst** command. When using the **hpccount** command, hardware performance counter plug-ins such as **average.so** and **single.so** that use MPI calls are not supported.

Understanding distributor functions

The motivating example for the distributor function is allowing a different hardware counter group on each MPI task. Therefore, the distributor is a subroutine that determines the MPI task ID (or MPI rank within **MPI_COMM_WORLD**) from the MPI environment for the current process, and sets or resets environment variables depending on this information. The environment variable can be any environment variable, not just the **HPM_EVENT_SET** environment variable, which specifies the hardware performance counter group.

The **distributor** function is called before any environment variable is evaluated by HPM. The settings of the environment variables done in the distributor take precedence over global environment variable settings.

The aggregator must adapt to the HPM group settings done by the distributor. This is why distributors and aggregators always come in pairs. Each plug-in contains a distributor and aggregator pair.

Understanding aggregator functions

The motivating example is the aggregation of the hardware performance counter data across MPI tasks. In the simplest case, this could be an average of the corresponding values. Hence this function is called:

- After the hardware performance counter data has been gathered
- Before the derived metrics are computed
- Before these data are printed

In general, the aggregator takes the raw results and rearranges them for output.

Also, depending on the MPI task rank the aggregator sets (or does not set) a flag to mark the current MPI task for HPM printing.

Plug-ins shipped with the tool kit

The following plug-ins are shipped with the IBM HPC Toolkit. You can specify a plug-in using the **-a** flag to the **hpccount** command or by using the **HPM_AGGREGATE** environment variable.

Table 45. Plug-ins shipped with the tool kit

Plug-in name	Description
mirror.so	This plug-in is called when no plug-in is requested. The aggregator mirrors the raw hardware performance counter data in a one-to-one fashion to the output function. It also flags each MPI task as a printing task. The corresponding distributor is an empty function. This plug-in does not use MPI and also works in a non-MPI context.
loc_merge.so	Does a local merge on each MPI task separately. It is identical to the mirror.so plug-in except for those MPI tasks that change the hardware performance counter groups in the course of the measurement (e.g. by multiplexing). The different counter data, which are collected for only part of the measuring interval, are proportionally extended to the whole interval and joined in to one big group that is used for derived metrics computation. This way, more derived metrics can be determined at the risk of computing invalid metrics. The user is responsible for using this plug-in only when it makes sense to use it. It also flags each MPI task as a printing task. The corresponding distributor is an empty function. This plug-in does not use MPI and also works in a non-MPI context.
single.so	Works the same as the **mirror.so** plug-in, but only on MPI task 0. The output on all other tasks is discarded. This plug-in uses MPI functions and cannot be used in a sequential context.
average.so	Is a plug-in for taking averages across MPI tasks. The distributor reads the environment variable **HPM_EVENT_SET** (which should be a comma-separated list of hardware performance counter group numbers) and distributes these group numbers in a round-robin fashion to the MPI tasks in the application. You can control the cluster of groups assigned to each task, by setting the environment variable **HPM_ROUND_ROBIN_CLUSTER** to the number of groups desired per task. The aggregator function creates an MPI communicator of all tasks with equal hardware performance counter group specifications. The communicator groups might be different from the original round-robin distribution. This could happen if the counting group has been changed on some of the MPI tasks after the first setting by the distributor function. Next, the aggregator computes the average for each hardware performance counter event across the subgroups formed by this communicator. Finally, it flags the MPI rank 0 in each group as a printing host. This plug-in uses MPI functions and cannot be used in a sequential context.

Why user-defined plug-ins are useful

This set of plug-ins is a starter kit and many more plug-ins might be desirable. Rather than taking the average of hardware performance counters across a set of MPI tasks, you could compute minimum or maximum values. You could also create a kind of a *history merge.so* by blending in results from previous measurements. You can write your own plug-ins using the interface described in "Understanding the distributor and aggregator interfaces" on page 210.

The source code for the supplied plug-ins is provided for you to use as examples in developing your own plug-ins. The source files and makefiles for the plug-ins are located in the **$IHPCT_BASE/examples/plugin** directory.

Understanding the distributor and aggregator interfaces

Each distributor and aggregator is a function returning an integer which is equal to zero on success and not equal to zero on error. In most cases the errors occur when calling a system call like **malloc**(), which sets the **errno** variable. If the distributor or aggregator returns the value of **errno** as the return code, the calling HPM tool can use the **errno** to display a meaningful error message. If returning **errno** is not viable, the function should return a negative value.

The function prototypes are defined in the **${IHPCT_BASE}/include/hpc_agg.h** header file:

```
#include "hpc_data.h"
int distributor(void);
int aggregator(int num_in, hpm_event_vector in,
               int *num_out, hpm_event_vector *out,
               int *is_print_task);
```

The distributor function has no parameters and is only required to set or reset environment variables using **setenv**(), if necessary, for correct operation of the aggregator function.

The aggregator function takes the current hardware performance counter values on each task as an input vector **in** and returns the aggregated values on the output vector **out** on selected or all MPI tasks. The aggregator is responsible for allocating the memory needed to hold the output vector **out**. The definition of the data types used for in and out are provided in the **${IHPCT_BASE}/include/hpc_data.h** header file.

Finally, the aggregator function must set (or reset) the flag, **is_print_task** to mark the current MPI task for HPM printing.

The hpm_event_vector **in** is a vector or list of **num_in** entries of type **hpm_data_item**. This data type is a struct containing members that describe the definition and the results of a single hardware performance counting task.

The following is from the **hpc_data.h** include file:

```
/*      NAME                INDEX   DESCRIPTION                          */

#define HPM_NTIM                8
#define HPM_TIME_WALLCLOCK      0   /* Wall clock time                      */
#define HPM_TIME_CYCLE          1   /* User time generated from ticks       */
#define HPM_TIME_USER           2   /* User time as reported by rusage()    */
#define HPM_TIME_SYSTEM         3   /* System time as reported by rusage()  */
#define HPM_TIME_START          4   /* Start time stamp (volatile)          */
#define HPM_TIME_STOP           5   /* Stop time stamp (volatile)           */
#define HPM_TIME_OVERHEAD       6   /* Overhead time for counter start/stop */
#define HPM_TIME_INIT           7   /* Overhead time for initialization     */

/*
 * These defines will be set depending on the group type
 * Each of the group types may have a different number of members
 */

#define HPM_GROUP_TYPE_UNSET   (0)
#define HPM_GROUP_TYPE_CPU     (1)
#define HPM_GROUP_TYPE_RUSAGE  (4)


typedef struct {
  int              num_data;        /* Number event info entries           */
```

```
  hpm_event_info  *data;            /* Event info list                     */
  double           times[HPM_NTIM]; /* Timing information                  */
  int              group_type;      /* cpu, rusage                         */
  int              mpi_task_id;     /* MPI task id if available            */
  int              instr_id;        /* id of the instrumented code         */
                                    /* section if applicable (-1 otherwise) */
  int              count;           /* number of entries/exits to the      */
                                    /* instrumented section (if applicable) */
  int              is_exclusive;    /* these are exclusive values          */
  int              xml_element_id;  /* arbitrary integer to identify       */
                                    /* hpm_data_items that translate into  */
                                    /* identical XML element shapes        */
  char            *description;     /* Description for this data item      */
  char            *xml_descr;       /* Descriptive header for XML output   */
} hpm_data_item;

typedef hpm_data_item *hpm_event_vector;
```

- The first element of the vector contains the data from a call to **getrusage**(). This vector element is the only element with its structure member **group_type** set to **HPM_GROUP_TYPE_RUSAGE** to distinguish it from hardware performance counter data.
- The count of events from an individual hardware performance counter group on one MPI task is contained in a single element of type **hpm_data_item**.
- If multiplexing is used, the results span several consecutive elements, each dedicated to one hardware performance counter group that takes part in the multiplex setting. On all but the first element, the member **is_mplex_cont** is set to **true** to indicate that these elements are continuations of the first element belonging to the same multiplex setup.
- If hardware performance counter groups are changed during the measurement, the results for different groups are recorded in different vector elements, but the **is_mplex_cont** flag is not set. This way results obtained using multiplexing can be distinguished from results obtained by an ordinary hardware performance counter group change.
- If several instrumented sections are used, each instrumented code section uses separate elements of type **hpm_data_item** to record the results. Each of these elements will have the member **instr_id** set to the value of the first argument of **hpmStart**() and the logical member **is_exclusive** set to **true** or **false** depending on whether the element hold inclusive or exclusive counter results as described in "Understanding inclusive and exclusive event counts" on page 205. Then all these different elements are concatenated in to a single vector.

The output vector is of the same format. Each vector element is used in the derived metrics computation separately (unless **is rusage** is equal to true). Then all vector elements and the corresponding derived metrics are printed in the order given by the vector **out**. The output of each vector element is preceded by the string pointed to by structure member **description** (which might include line feeds, as appropriate). The XML output will be labeled with the text pointed to by **xml_descr**. This way the input vector **in** is providing a complete picture of what has been measured on each MPI task. The output vector **out** is allowing complete control on what is printed on which MPI task in what order.

Getting the plug-ins to work

The sample plug-ins are compiled with the Makefile **${IHPCT_BASE}/examples/plugin/Makefile** using the command:

```
gmake
```

Note the following considerations when implementing your own plug-in:

- The Makefile distinguishes sequential and parallel plug-ins. The latter are compiled and linked with the MPI wrapper script for the compiler and linker. Unlike a static library, generation of a shared object requires linking, not just compilation.
- There are some restrictions to be observed when writing plug-in code.
  - The MPI standard document disallows calling **MPI_Init()** twice in the same process.
  - The distributor function is called by **hpmInit**(). If the distributor function contains MPI calls, the user's application is required to call MPI **Init**() prior to calling **hpmInit**(). To avoid this restriction, the distributor function must not call any MPI function. The MPI task ID should be extracted by inspecting environment variables, specifically the **MP_CHILD** environment variable, that have been set by the MPI software stack.
  - The aggregator function, however, usually cannot avoid calling MPI functions. Before calling **MPI_Init**(), it has to check whether the instrumented application has already done so, for example, by calling the MPI_**Initialized**() function. If the instrumented application is an MPI application, the aggregator function cannot be called after **MPI_Finalize**(). The aggregator function is called by **hpmTerminate**(). Hence **hpmTerminate**() has to be called between the calls to **MPI Init**() and **MPI Finalize**().
  - **libhpc** uses a call to **dlopen()** to access the plug-in and makes use of its functions.

Part 7. The MPI and I/O profiling libraries

The topics in this part provide information about using the MPI and I/O profiling libraries.

© Copyright IBM Corp. 2008, 2013

Chapter 15. Using the MPI profiling library

The MPI profiling library, **libmpitrace**, is a library that you can link with your MPI application to profile the MPI function calls in your application, or to create a trace of those MPI calls. When you link your application with this library, the library intercepts the MPI calls in your application, using the Profiled MPI (PMPI) interface defined by the MPI standard, and obtains the profiling and trace information it needs. This library also provides a set of functions that you can use to control how profiling and trace data is collected, as well as functions that you can use to customize the trace data.

Although the **libmpitrace** library can be used in a threaded application, it does not correctly record MPI trace events in an application in which MPI function calls are made on multiple threads. You should use **libmpitrace** only in single threaded applications or applications in which MPI function calls are made only on a single thread.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the MPI profiling library. In order to set these environment variables, run one of the setup scripts that are located in the top-level directory of your IBM HPC Toolkit installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

Compiling and linking with libmpitrace

When you compile your application, you must use the **–g** compiler flag so that the library can obtain the information it needs to map performance information back to application source code. You might want to consider compiling your application at lower optimization levels since compiler optimizations might affect the accuracy of mapping MPI function calls back to source code and the accuracy of the function call stack for an MPI function call.

Any C source file containing calls to functions in the **libmpitrace** library should include the **mpt.h** header file, which is located in the **${IHPCT_BASE}/include** directory. All applications must link the **libmpitrace** library with the application using the **–lmpitrace** linker flag. The **libmpitrace** library is located in **${IHPCT_BASE}/lib** for 32-bit applications and in **${IHPCT_BASE}/lib64** for 64-bit applications. Although it is not necessary for all linkers, it is recommended that the link command parameters be ordered such that the mpi program object files come first, before specifying the**–lmpitrace** linker flag.

If your POWER application is compiled and linked with the MPICH2 version of MPI, you must use the MPICH2 version of **libmpitrace**. The MPICH2 version of the **libmpitrace** library is located in **${IHPCT_BASE}/lib/mpich2** for 32-bit applications and in **${IHPCT_BASE}/lib64/mpich2** for 64-bit applications.

After you link your application with the **libmpitrace** library, you can run your application just as you normally would.

© Copyright IBM Corp. 2008, 2013

Controlling traced tasks

The **libmpitrace** library stores MPI trace events in memory for performance reasons. By default, the number of MPI trace events that are recorded is limited to 30,000 events or less. Additional MPI trace events beyond this number are discarded. You can override this default by setting the **MAX_TRACE_EVENTS** environment variable to the maximum number of MPI trace events to be recorded. Increasing this value means that additional memory will be used to store MPI trace events and that additional memory usage might affect your application program.

By default, for scalability, profiling data files and MPI function call events are generated for a maximum of four tasks in the application:

- Task 0
- The task with the minimum MPI communication time
- The task with the maximum MPI communication time
- The task with the median MPI communication time

If task 0 is the task with minimum, maximum, or median MPI communication time, at most, output files will be generated for only three tasks. If you want output to be generated for all MPI tasks, set the **OUTPUT_ALL_RANKS** environment variable to **yes** before running the application.

By default, the **libmpitrace** library traces MPI tasks 0 through 255 (or less if the application has fewer than 256 MPI tasks). If you need to see MPI traces from all tasks, you must set the **TRACE_ALL_TASKS** environment to **yes** before running the application. If you have an application with more than 256 MPI tasks, but you do not want to see traces from all MPI tasks, you can set the **MAX_TRACE_RANK** to the MPI task index of the highest numbered MPI task that you want traced.

By default, when **libmpitrace** obtains the calling address for each MPI function that is traced, it gets the address of the MPI function's immediate caller. If MPI functions are called from within another library, or deeply layered within your application, you might want **libmpitrace** to obtain the MPI function caller's address from one or more layers higher in the function call stack. You can specify how many levels to walk back in the function call stack by setting the **TRACEBACK_LEVEL** environment level to the number of levels to walk, where 0 means to obtain the address where the MPI function was actually called.

If you do not need to override the **MAX_TRACE_RANK** environment variable setting, you can set the **MT_BASIC_TRACE** environment variable to **yes**. If you do this, the trace library will operate with lower overhead for each MPI function call.

Additional trace controls

You can obtain additional control over MPI trace generation by using function calls in the **libmpitrace** library. You can trace selected sections of your application by bracketing areas of interest with calls to the **MT_trace_start**() and **MT_trace_stop**() functions. In order to use these functions, you must set the **TRACE_ALL_EVENTS** environment variable to **no** before running your application. When you start your application, tracing is initially suspended. When your application invokes the **MT_trace_start**() function, MPI trace event collection is resumed in the task where **MT_trace_start**() was called. Tracing continues until the **MT_stop_trace()** function

is called. At that time, MPI trace event collection is suspended in the task that called **MT_trace_stop**(). Tracing can be resumed again by a subsequent call to the **MT_trace_start**() function.

The **MT_trace_start**() and **MT_trace_stop**() functions can be called from C applications. FORTRAN applications can call the **mt_trace_start**() and **mt_trace_stop**() functions.

You can control which MPI function calls are traced by implementing your own version of the **MT_trace_event**() function. The C function prototype for this function is:

```
int MT_trace_event(int id);
```

where:

id Is an enumeration identifying the specific MPI function that is being executed. You should include the **mpi_trace_ids.h** header, located in the **${IHPCT_BASE}/include** directory, when you implement this function.

Your implementation of **MT_trace_event**() must return **1** if the MPI trace event should be recorded, and must return **0** if the MPI trace event should not be recorded.

You can control which MPI tasks should have MPI trace events recorded by implementing your own version of the **MT_output_trace**() function. The C function prototype for this function is:

```
int MT_output_trace(int task);
```

where:

task
Is the MPI task ID of the task calling this function. Your implementation of this function must return **1** if the MPI trace event is to be recorded and return **0** if the MPI trace event is not to be recorded.

Customizing MPI profiling data

You can create customized MPI profiling data by implementing your own version of the **MT_output_text**() function. The C function prototype for this function is:

```
int MT_output_text(void);
```

This function is called for each MPI task when that task calls **MPI_Finalize**(). If you implement your own version of the **MT_output_text**() function, you are responsible for generating all profiling data, in whatever format you require. You might use any of the functions described in "Understanding MPI profiling utility functions" on page 218 in your implementation of the **MT_output_text**() function.

Your implementation of this function should return **1** if it successfully completes and return **-1** if an error occurs in processing.

Understanding MPI profiling utility functions

The **libmpitrace** library provides a set of functions that you can use to obtain information about the execution of your application. You can use these functions when implementing your own versions of **MT_trace_event**(), **MT_output_trace**(), **MT_output_text**(), or anywhere else, including your own application code where they are useful.

There are several functions you can use to obtain information about types of MPI functions called in your application, as described in Table 46:

Table 46. MPI profiling utility functions

Function	Purpose
MT_get_mpi_counts	Determines how many times an MPI function is called.
MT_get_mpi_bytes	Determines the total number of bytes that are transferred by all calls to a specific MPI function.
MT_get_mpi_time	Determines the cumulative amount of time that is spent in all calls to a specific MPI function.
MT_get_mpi_name	Obtains the name of an MPI function, given the internal ID that is used by the IBM HPC Toolkit to refer to this MPI function.
MT_get_time	Determines the elapsed time since **MPI_Init**() was called.
MT_get_elapsed_time	Determines the elapsed time between calls to **MPI_Init** and **MPI_Finalize**.
MT_get_tracebufferinfo	Determines the size and current usage of the internal MPI trace buffer that is used by the IBM HPC Toolkit.
MT_get_calleraddress	Determines the address of the caller of a currently-active MPI function.
MT_get_callerinfo	Determines the source file and line number information for an MPI function call, using the address that is obtained by calling **MT_get_calleraddress**.
MT_get_environment	Obtains information about the MPI execution environment.
MT_get_allresults	Obtains statistical information about a specific MPI function call.

All of these functions are documented in Chapter 19, "Application programming interfaces," on page 261.

Performance data file naming

By default, the **libmpitrace** library generates three sets of output files in the current working directory. For more information, see Appendix A, "Performance data file naming," on page 309.

Chapter 16. Using the I/O profiling library

The topics that follow will provide information about using the I/O profiling library.

Preparing your application

The IBM HPC Toolkit provides a library that you can use to profile I/O calls in your application.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you use the I/O profiling library. In order to set these environment variables, run one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, you should invoke the **env_sh** script as **. env_sh**. If you are using **csh**, you should invoke the **env_csh** script as **source env_csh**.

In order to profile your application, if you are not instrumenting your application using **hpctInst**, **peekperf**, or the HPC Toolkit Eclipse plug-in, you must link your application with the **libhpctkio** library using the **–L$IHPCT_BASE/lib** and **–lhpctkio** linking options for 32-bit applications or using the **–L$IHPCT_BASE/lib64** and **–lhpctkio** linking options for 64-bit applications.

You must also set the **TKIO_ALTLIB** environment variable to the path name of an interface module used by the I/O profiling library before you invoke your application. For 32-bit applications, the **TKIO_ALTLIB** environment variable should be set to **$IHPCT_BASE/lib/get_hpcmio_ptrs.so**. For 64-bit applications, the **TKIO_ALTLIB** environment variable should be set to **$IHPCT_BASE/lib64/get_hpcmio_ptrs.so.** Optionally, the I/O profiling library can print messages when the interface module is loaded, and it can abort your application if the interface module cannot be loaded.

In order for the I/O profiling library to display a message when the interface module is loaded, you must append **/print** to the setting of the **TKIO_ALTLIB** environment variable. In order for the IO profiling library to abort your application if the interface module cannot be loaded, you must append **/abort** to the setting of the **TKIO_ALTLIB** environment variable. You might specify one, both, or none of these options.

Note: There are no spaces between the interface library path name and the options. For instance, to load the interface library for a 32-bit application, display a message when the interface library is loaded, and abort the application if the interface library cannot be loaded, you would issue the following command:

```
export TKIO_ALTLIB="$IHPCT_BASE/lib/get_hpcmio_ptrs.so/print/abort"
```

Setting I/O profiling environment variables

There are two environment variables that the I/O profiling library uses to determine the files for which I/O profiling is to be performed and the data that will be obtained by profiling. These environment variables should be set, as needed, before you run your application.

© Copyright IBM Corp. 2008, 2013

The first environment variable is **MIO_FILES**, which specifies one or more sets of file name and the profiling library options to be applied to that file, where the file name might be a pattern or an actual path name.

The second environment variable is **MIO_DEFAULTS**, which specifies the I/O profiling options to be applied to any file whose file name does not match any of the file name patterns specified in the **MIO_FILES** environment variable. If **MIO_DEFAULTS** is not set, no default actions are performed.

The file name that is specified in the **MIO_FILES** variable setting might be a simple file name specification, which is used as-is, or it might contain wildcard characters, where the allowed wildcard characters are:

- A single asterisk (*), which matches zero or more characters of a file name.
- A question mark (?), which matches a single character in a file name.
- Two asterisks (**), which match all remaining characters of a file name.

The I/O profiling library contains a set of modules that can be used to profile your application and to tune I/O performance. Each module is associated with a set of options. Options for a module are specified in a list, and are delimited by **/** characters. If an option requires a string argument, that argument should be enclosed in curly braces '{}', if the argument string contains a **/** character.

Multiple modules can be specified in the settings for both **MIO_DEFAULTS** and **MIO_FILES**. For **MIO_FILES**, module specifications are delimited by a pipe (|) character. For **MIO_DEFAULTS**, module specifications are delimited by commas (,).

Multiple file names and file name patterns can be associated with a set of module specifications in the **MIO_FILES** environment variable. Individual file names and file name patterns are delimited by colon (:) characters. Module specifications associated with a set of file names and file name patterns follow the set of file names and file name patterns, and are enclosed in square brackets ([]).

As an example of the **MIO_DEFAULTS** environment variable setting, assume that the default options for any file that does not match the file names or patterns specified in the **MIO_FILES** environment variable are that the **trace** module is to be used with the **stats** and **mbytes** options and that the **pf** module is also to be used with the **stats** and **mbytes** options. The **stats** option output for both **trace** and **pf** is written to a file with a subtype of **miostats**. For more information, see Appendix A, "Performance data file naming," on page 309. The setting of the **MIO_DEFAULTS** environment variable would be:

```
export MIO_DEFAULTS="trace/stats=miostats/mbytes,pf/stats=miostats/mbytes"
```

As an example of using the **MIO_FILES** environment variable, assume you have a program that does I/O to **/tmp/testdata** and some ***.txt** and ***.dat** files in your current working directory. The following setting will cause **/tmp/testdata** to use the trace module with the events option and files matching the patterns ***.txt** or ***.dat** to use the trace module with the **stats** and **events** options. The setting of the **MIO_FILES** environment variable would be:

```
export MIO_FILES="/tmp/testdata [trace/events={tmp_testdata}] \
*.txt : *.dat [trace/events={any_txt_or_dat}/stats={any_txt_or_dat}]"
```

After running the program the following performance data files will be created:

hpct.mio.tmp_testdata.iot
Contains the trace events for the **/tmp/testdata** file.

hpct.mio.txt
Contains the **trace** stats for the **/tmp/testdata** file.

hpct.mio.any_txt_or_dat.iot
Contains the trace events for the ***.txt** and ***.dat** files.

hpct.mio.any_txt_or_dat.txt
Contains the **trace** stats for the ***.txt** and ***.dat** files.

Specifying I/O profiling library module options

The following modules are available in the I/O profiling library:

Table 47. MIO analysis modules

Module	Purpose
mio	The interface to the user program.
pf	A data prefetching module.
trace	A statistics gathering module.
recov	Analyzes failed I/O accesses and retries in case of failure.

The **mio** module has the following options:

Table 48. MIO module options

Option	Purpose
mode=	Override the file access mode in the open system call.
nomode	Do not override the file access mode.
direct	Set the **O_DIRECT** bit in the open system call.
nodirect	Clear the **O_DIRECT** bit in the open system call.

The default option for the **mio** module is **nomode.**

The **pf** module has the following options:

Table 49. MIO pf module options

Option	Purpose
norelease	Do not free the global cache pages when the global cache file usage count goes to zero. The **release** and **norelease** options control what happens to a global cache when the file usage count goes to zero. The default behavior is to close and release the global cache. If a global cache is opened and closed multiple times, there could be memory fragmentation issues at some point. Using the norelease option keeps the global cache opened and available, even if the file usage count goes to zero.
release	Free the global cache pages when the global cache file usage count goes to zero.
private	Use a private cache. Only the file that opens the cache might use it.
global=	Use global cache, where the number of global caches is specified as a value between **0** and **255**. The default is **1**, which means that one global cache is used.
asynchronous	Use asynchronous calls to the child module.

Table 49. MIO pf module options (continued)

Option	Purpose
synchronous	Use synchronous calls to the child module.
noasynchronous	Alias for synchronous
direct	Use direct I/O.
nodirect	Do not use direct I/O.
bytes	Stats output is reported in units of bytes.
kbytes	Stats is reported in output in units of 12 bytes.
mbytes	Stats is reported in output in units of mbytes.
gbytes	Stats is reported in output in units of gbytes.
tbytes	Stats is reported in output in units of tbytes.
cache_size=	The total size of the cache (in bytes), between the values of **0** and **1GB**, with a default value of **64 K**.
page_size=	The size of each cache page (in bytes), between the value of **4096** bytes and **1GB**, with a default value of **4096**.
prefetch=	The number of pages to prefetch, between **1** and **100**, with a default of **1**.
stride=	Stride factor, in pages, between **1** and **1G** pages, with a default value of **1**.
stats=	Output prefetch usage statistics to the specified file. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 309.
nostats	Do not output prefetch usage statistics.
inter	Output intermediate prefetch usage statistics on **kill –USR1.**
nointer	Do not output intermediate prefetch usage statistics.
retain	Retain file data after close for subsequent reopen.
noretain	Do not retain file data after close for subsequent reopen.
listio	Use listio mechanism.
nolistio	Do not use listio mechanism.
tag=	String to prefix stats flow
notag	Do not use prefix stats flow.

The default options for the pf module are:

```
/nodirect/stats/bytes/cache_size=64k/page_size=4k/
prefetch=1/asynchronous/global/release/stride=1/nolistio/notag
```

The **trace** module has the following options:

Table 50. MIO trace module options

Option	Purpose
stats=	Output trace statistics to the specified file name. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 309.
nostats	Do not output statistics on close.

Table 50. MIO trace module options (continued)

Option	Purpose
events=	Generate a binary events file. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 309.
noevents	Do not generate a binary events file.
bytes	Output statistics in units of bytes.
kbytes	Output statistics in units of kilobytes.
mbytes	Output statistics in units of megabytes.
gbytes	Output statistics in units of gigabytes.
tbytes	Output statistics in units of terabytes.
inter	Output intermediate trace usage statistics on **kill –USR1**.
nointer	Do not output intermediate statistics.
xml=	Generate the statistics file in a format that can be viewed using the Eclipse plug-in or **peekperf**. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 309.

The default options for the **trace** module are:

```
/stats/noevents/nointer/bytes
```

The **recov** module has the following options:

Table 51. MIO recov module options

Option	Purpose
fullwrite	All writes are expected to be full writes. If there is a write failure due to insufficient space, the recov module retries the write.
partialwrite	All writes are not expected to be full writes. If there is a write failure due to insufficient space, there will be no retry.
stats=	Output **recov** module statistics to the specified file name. The name, if specified will be the subtype portion of the file name. For more information, see Appendix A, "Performance data file naming," on page 309.
nostats	Do not output **recov** statistics on file close.
command=	The system command to be issued on a write error.
open_command=	The system command to be issued on open error resulting from a connection that was refused.
retry=	Number of times to retry, between **0** and **100**, with a default of **1**.

The default options for the recov module are:

```
partialwrite/retry=1
```

Running your application

The I/O profiling options of most interest when using the IBM HPC Toolkit are the **stats** option, which specifies the name of the statistics file that contains data about the I/O performance of your application, and the events option which specifies the name of a trace file containing data that can be viewed within **peekperf**.

After you have compiled and linked your application as described in "Preparing your application" on page 219 and set the **MIO_FILES** and **MIO_DEFAULTS** environment variables, as needed, then you can run your application.

After you run your application, you can view trace files generated by the I/O profiling library using **peekperf**.

Performance data file naming

The names of the output files generated by the I/O profiling tool are dependent on the settings of options in the **MIO_FILES** or **MIO_DEFAULTS** environment variables and whether the application is a serial program, a MPI program, which does not use dynamic tasking, or a MPI program which uses dynamic tasking.

For information about how files are named, see Appendix A, "Performance data file naming," on page 309.

Part 8. Using the hpctInst command

The topics in this part provide information about how to instrument your application using the **hpctInst** command.

© Copyright IBM Corp. 2008, 2013

Chapter 17. Instrumenting your application using hpctInst

In addition to modifying your application source code to contain calls to instrumentation functions, you can use the **hpctInst** utility to instrument your application without modifying your application source code. The **hpctInst** utility creates a new copy of your application's executable containing the instrumentation that you specified using command-line flags to the **hpctInst** utility. You can instrument your application with **hpctInst** to obtain performance measurements for hardware performance counters, MPI profiling, OpenMP profiling, and I/O profiling.

Note: The **hpctInst** command is not supported on x86 Linux systems.

You must ensure that several environment variables required by the IBM HPC Toolkit are properly set before you invoke **hpctInst**. To set these environment variables, run one of the setup scripts that are located in the top-level directory of your installation. These setup scripts are located in the **/opt/ibmhpc/ppedev.hpct** directory. If you are using **sh**, **bash**, **ksh**, or a similar shell command, invoke the **env_sh** script as **. env_sh**. If you are using **csh**, invoke the **env_csh** script as **source env_csh**.

If you are going to instrument your application using **hpctInst**, you must compile the application using the **–g** compiler flag so that **hpctInst** can find the line number and symbol table information it needs to instrument the application. When you link your application, you should not link it with any libraries from the IBM HPC Toolkit. If you have a Power Linux application, you must link your application using the **-Wl,--hash-style=sysv –emit-stub-syms** flags.

Note: You must not compile your application with the **-pg** flag.

After you have instrumented your application, you should set any environment variables that are required by the instrumentation you requested. All of the environment variables described in the topics for hardware performance counters, MPI profiling, and I/O profiling can be used, as needed, when running an application instrumented with **hpctInst**. The exception is OpenMP profiling, in which the environment variables are used only to control the insertion of instrumentation when the **hpctInst** utility is run, and must be set before **hpctInst** is run. The OpenMP-specific environment variables are only described in "hpctInst - Instrument applications to obtain performance data" on page 248.

Note: If you instrument small, frequently-called functions, the instrumentation overhead might be significant, and the accuracy of performance measurements might be affected by this overhead.

Instrumenting your application for hardware performance counters

You can instrument your application to obtain hardware performance counter information in the following ways:

- You can instrument the entry and exit points of every function in your application by using the **–dhpm** option. When you do this, you obtain performance data that includes hardware performance counter totals for each

© Copyright IBM Corp. 2008, 2013

function in your application. You can use this data to identify functions that require tuning for improved performance.

- You can instrument your application to obtain hardware performance counter performance data at specific locations (function call sites) in your application, where a function is called, using the **–dhpm_func_call** option. If you use this option, you will obtain hardware performance counter information for the function called at the specified location. You can use this information to help you identify how a function called from multiple locations in your application performs from each individual location from which it is called.

  You specify the set of function call sites in the file specified as a parameter to the **–dhpm_func_call** option. You can specify locations to be instrumented such that only function calls from specific functions are instrumented, or only function calls within a specified region of source code. The following example shows a file that specifies that calls to function **sum** from function **compute** and calls to function **distribute_data** from source file **main.c** between lines 100 and 200 are instrumented.

  ```
  sum compute
  distribute_data main.c 100 200
  ```

- You can instrument selected regions of your source code by using the **–dhpm_region** option and specifying a file that contains a list of one or more regions of code to be instrumented. Regions of code might overlap. If regions of code overlap, then the considerations described in "Handling of overlap issues" on page 206 apply. The following example shows a file that specifies that regions of code between lines 1 and 100 of **main.c** and lines 100 to 300 of **report.c** are to be instrumented.

  ```
  main.c 1 100
  report.c 100 300
  ```

Instrumenting your application for MPI profiling

You can instrument your application for MPI profiling in the following ways:

- You can instrument the entire application so that all MPI calls in the application are traced by using the **–dmpi** option.
- You can instrument your application so that only specific MPI functions called from specific functions in your application are instrumented by using the **–dmpi_func_call** option. You specify the set of MPI function call sites in the file specified as a parameter to the **–dmpi_func_call** option. You can specify locations to be instrumented such that MPI function calls only from a specific function in your application are instrumented or only MPI function calls within a specified region of source code are instrumented. The following example shows a file that specifies that calls to the **MPI_Send()** function from function compute and calls to **MPI_Recv()** from source file **main.c** between lines 100 and 200 are instrumented:

  ```
  MPI_Send compute
  MPI_Recv main.c 100 200
  ```

- You can instrument selected regions of your source code in which all MPI function calls within that region are instrumented by using the **–dmpi_region** option and specifying a file that contains a list of one or more regions of code to be instrumented. The following example shows a file that specifies that regions of code between lines 1 and 100 of **main.c** and lines 100-300 of **report.c** are to be instrumented:

  ```
  main.c 1 100
  report.c 100 300
  ```

Instrumenting your application for OpenMP profiling

You can instrument your application for OpenMP profiling in the following ways:

- Instrument all parallel loops and parallel regions in your application by using the **–dpomp** option.
- Instrument all parallel regions in your application by using the **–dpomp_parallel** option.
- Instrument all parallel loops in your application by using the **–dpomp_loop** option.
- Instrument all user-written functions in your application by using the **–dpomp_user** option, or a selected subset of user-written functions by using the **–dpomp_userfunc** option. If you use the **–dpomp_userfunc** option, you specify the set of user-written functions to be instrumented in a file, which is a simple list of the functions to be instrumented, with one function name per line in that file.

If you want to instrument only selected parallel loops or parallel regions in your application, use the Eclipse plugin or **peekperf** GUIs because they allow you to select specific OpenMP constructs to instrument.

Instrumenting your application for I/O profiling

You can instrument your entire application for I/O profiling by using the **–dmio** option. You select the specific files that will have performance data obtained for them by setting I/O profiling environment variables as specified in "Setting I/O profiling environment variables" on page 219.

If you want to instrument only specific I/O function calls in your application, use the Eclipse plug-in or **peekperf** to instrument your application.

Part 9. Command and API reference

The topics in this part provide information about the commands and APIs associated with the following:

- Application launch
- Hardware performance monitoring
- MPI profiling and trace (MPI trace APIs)
- Application instrumentation
- Performance data visualization

See Table 52 on page 233 and Table 53 on page 261 for more information.

© Copyright IBM Corp. 2008, 2013

Chapter 18. Commands

Table 52 lists the commands and what they are used for:

Table 52. Commands

Used for:	Command
Application launch	"hpcrun - Launch a program to collect profiling or trace data" on page 241
Hardware performance monitoring	"hpccount - Report hardware performance counter statistics for an application" on page 236 "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244
Application instrumentation	"hpctInst - Instrument applications to obtain performance data" on page 248
Performance data visualization	"dataview - View an I/O trace" on page 234 "peekperf - Provides a GUI for the HPC Toolkit" on page 252 "peekview - Display an MPI trace file" on page 254 "Xprof - Display the application as a function tree" on page 256

© Copyright IBM Corp. 2008, 2013

dataview - View an I/O trace

Use the **dataview** command to view a trace file that is generated by the IBM HPC Toolkit Modular I/O (MIO) tool.

Syntax

dataview [*event_trace* [*event_trace*...]]
dataview [**-h**]
dataview [**--help**]

Parameters

event_trace
Name of the I/O trace file.

Flags

-h
--help Displays a **dataview** usage message.

Description

The **dataview** GUI can be used to view a trace file generated by the IBM HPC Toolkit I/O Profiling tool. The trace is displayed in a graph format, where in the initial display, the x axis represents elapsed time and the y axis represents file position. You can zoom in and out of this graph display and display I/O transfer rates for selected areas of the graph. You can also display the MIO trace in tabular format and use the tabular view to customize the MIO graph to show additional data.

When you start **dataview**, you can specify a MIO trace file to be viewed, or you can invoke **dataview** without specifying a MIO trace file to open. If you do not specify a MIO trace file, you can do so by selecting the **Read Event** option from the **File** menu in the initial **dataview** window.

When **dataview** starts, and a MIO trace file is loaded, **dataview** displays a window containing a tree view showing the instrumented I/O calls in the application that generated the trace file. You can select one or more of the I/O calls from this tree. If **dataview** was invoked from within **peekperf**, by selecting **View Tracer** from the Data Visualization Window pop-up menu that appears when right-clicking in that window, then when you select an I/O function call from the tree view, **peekperf** positions the source view window to show the I/O call that you selected.

After you select I/O calls, and you select **New Plot** at the top of the tree view, the graph is displayed. You can zoom in and out of the graph using the **Zoom In** and **Zoom Out** buttons at the top of the graph window. You can save the graph as a jpeg (**.jpg**) file by clicking the **Save** button, and print the graph by clicking the **Print** button. You can also zoom in the graph by left-clicking and dragging a rectangle around the area of the graph into which you want to zoom in to.

You can show the slope (data transfer rate) for file I/O by right-clicking over a part of the plot and dragging to draw a line over the plot. The slope of that line is displayed, along with starting and ending coordinates, in the status area at the bottom of the graph. If you drag over a part of the plot while right-clicking, the slope is the data rate in bytes per second for the traced fragment of the graph.

You can display the trace in tabular format by selecting one or more I/O calls from the tree view then clicking **Edit Table** in the **File** menu in the tree view. A table opens for each I/O call, showing the data for each time that I/O call was executed. You can select a column to sort the table by clicking over a column heading. If you right-click over a column, a pop-up menu appears in which you can hide displayed columns, show hidden columns, or save the table to a file in CSV (spreadsheet) format.

You can customize a MIO graph by making selections from the widgets at the top of the data view table. These selections take effect the next time you click **Plot** in a graph window or select **New Plot** from the tree window. There are four widgets that you can use to customize the graph.

The first widget is the color icon. If you select this icon, a color selector dialog appears, in which you can select the color of the displayed data.

The second widget selects the metric used for the y axis of the graph. You can select file position activity, data delivery rate, or rate vs. position. If you select file position activity, data is displayed in the graph as individual data points. If you select data delivery rate or rate versus position, data is displayed in the graph as numerical values at the locations at which points would be plotted.

The third widget specifies how many pixels are used to draw each data point.

The fourth widget specifies the metric whose numerical value is to be displayed at each point in the graph.

Environment variables

Miscellaneous environment variables

IHPCT_BASE
: Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Examples

To view a single I/O trace file, enter:

```
dataview hpct_0_0.mio.iot
```

hpccount - Report hardware performance counter statistics for an application

Use the **hpccount** command to report summary hardware performance counter and resource usage statistics for an application.

Syntax

hpccount [**-o** *name*] [**-u**] [**-n**] [**-x**] [**-g** *group*[,*group*,...]] [**-a** *plugin*] *program*
hpccount [**-h**]
hpccount [**-l**]
hpccount [**-L**]
hpccount [**-c**]

Flags

-a Specifies the name of a plug-in that defines the HPM data aggregation strategy. If the plug-in name contains a /, the name is treated as an absolute or relative path name. If the name does not contain a /, the plug-in is loaded following the rules for the **dlopen()** function call. The plug-in is a shared object file that implements the **distributor()** and **aggregator()** functions. See "Hardware performance counter plug-ins" on page 208 for more information.

-c Lists the available counters and the hardware counter events that can be counted by each counter.

-g *group*[,*group*,...]
A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups to be multiplexed. If this flag or the **HPM_EVENT_SET** environment variable is not set, a processor specific default group is used, as follows:

```
POWER6              127
POWER7, POWER7+     147
X86                   1
```

Note: Only CPU hardware counter groups can be specified when running **hpccount**.

-h Displays a usage message.

-l Lists the available hardware counter groups and the hardware counter events that are counted by each group.

-L Lists the available hardware counter groups and the counters contained in each group. The individual counter information will include a short name, the hardware event ID and a short text description. This flag is only available when running Linux.

-n Suppresses **hpccount** output to stdout.

-o *name*
Writes output to a file with the prefix of *name*.

- The file prefix *name* can be specified using option **-o** or using the environment variable **HPC_OUTPUT_NAME**. The option takes precedence if there are conflicting specifications. If not specified, the file prefix is **hpct**.
- The name *name* is expanded into different file names:

– *name*.**hpm**.*app*.**txt** is the file name for ASCII output, which is a one-to-one copy of the screen output.
– *name*.**hpm**.*app*.**viz** is the file name for the XML output.

For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

- Which of these output files are generated is governed by the additional environment variables, **HPM_ASC_OUTPUT** and **HPM_VIZ_OUTPUT**. If neither of these are set, only the ASCII output is generated. If at least one is set, the following rules apply:
  – **HPM_ASC_OUTPUT**, if set to **yes**, triggers the ASCII output
  – **HPM_VIZ_OUTPUT**, if set to **yes**, triggers the XML output
- Unless the **-a** option is chosen, there is one output for each MPI task.

-u Specifies that unique file names are used for generated ASCII and XML output files. The unique file name flag can be specified using option **-u** or using the environment variable **HPC_UNIQUE_FILE_NAME**. The option takes precedence if there are conflicting specifications. Unique file names are generated according to the following rules:

- For serial programs, a string *_processId* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that do not use dynamic tasking, a string _0_*taskRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that use dynamic tasking, a string *_worldId_worldRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.

For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

-x Displays formulas for derived metrics as part of the command output.

Description

The **hpccount** command provides comprehensive reports of events that are critical to performance on IBM systems. HPM is able to gather the usual timing information, as well as critical hardware performance metrics, such as the number of misses on all cache levels, the number of floating point instructions executed, and the number of instruction loads that cause TLB misses. These reports help the algorithm designer or programmer identify and eliminate performance bottlenecks.

The **hpccount** command invokes the target program and counts hardware performance counter events generated by the target program. The **hpccount** command reports this information after the target program completes.

If you are using the **hpccount** command to measure the performance of a parallel program, you should invoke **hpccount** as **poe hpccount program**. If you invoke **hpccount** as **poe hpccount program**, you will measure the performance of the **poe** command, and not the performance of your application.

For information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

Environment variables

Event selection environment variables

HPM_EVENT_SET

A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups to be multiplexed. If the **–g** flag is not used and **HPM_EVENT_SET** is not set, a processor-specific default group is used, as follows:

```
POWER6               127
POWER7, POWER7+      147
X86                   1
```

Note: Only CPU hardware counter groups can be specified when running **hpccount**.

HPM_SLICE_DURATION

Specifies the interval, in milliseconds, to be used when hardware counter groups are multiplexed. You can specify a value between **10** milliseconds and **30** seconds. The default is **100** milliseconds. This environment variable is supported for AIX only.

HPM_COUNTING_MODE

Specifies the CPU mode where counting will occur. Set this to a comma separated list of any combination of the following three possible values:

- **user**

  Set to **user** to have user side events counted.
- **kernel**

  Set to **kernel** to have kernel or system events counted.
- **hypervisor**

  Set to **hypervisor** to have hypervisor events counted.

The default setting for **hpccount** is **user**.

Output control environment variables

HPM_ASC_OUTPUT

Determines whether or not to generate an ASCII output file. The output file name is:

name.**hpm**.*app*.**txt**

where:

app
: Is the name of the executable.

Valid values are **yes** or **no**. The default value for **HPM_ASC_OUTPUT** is **true**, except if **HPM_VIZ_OUTPUT** has been set, in which case the default value for **HPM_ASC_OUTPUT** is **false**.

HPC_OUTPUT_NAME

Specifies the *name* prefix of the output files:

name.**hpm**.*app*.**txt**

and

name.**hpm**.*app*.**viz**

For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

HPM_PRINT_FORMULA
Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.

HPM_STDOUT
Set to **yes** to write ASCII output to stdout. If **HPM_STDOUT** is set to **no**, no output is written to stdout. The default is **yes**.

HPC_UNIQUE_FILE_NAME
Set to **yes** in order to generate unique file names for generated ASCII and XML output files. Set to **no** to generate the file name exactly as specified by **HPC_OUTPUT_NAME**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

HPM_VIZ_OUTPUT
Set to **yes** to generate an XML output file with the name:

name.hpm.*app*.viz

The default value for **HPM_VIZ_OUTPUT** is **false**.

Plug-in specific environment variables

HPM_ROUND_ROBIN_CLUSTER
HPM_ROUND_ROBIN_CLUSTER allows setting the number of groups distributed per task.

Without the environment variable **HPM_ROUND_ROBIN_CLUSTER** set, the **average.so** plug-in will distribute the group numbers from **HPM_EVENT_SET** in a round-robin fashion, one group to each of the MPI tasks in the application.

The default value for **HPM_ROUND_ROBIN_CLUSTER** is **1**. The default will be used if a value less than 1 is specified.

The number of groups spread out round robin-fashion to the tasks will be limited to the first "number of tasks times the setting of **HPM_ROUND_ROBIN_CLUSTER**" groups.

If a value greater than the number of groups in **HPM_EVENT_SET** is specified, **HPM_ROUND_ROBIN_CLUSTER** will be set to the number of groups specified in **HPM_EVENT_SET**.

It is possible that the number of groups does not distribute evenly to the tasks. The first task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups in **HPM_EVENT_SET**. If there are more tasks and groups left, the second task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups left in **HPM_EVENT_SET** and so on, until there are no groups unused in **HPM_EVENT_SET**. After the groups in **HPM_EVENT_SET** have been used once, and there are more tasks, the process will repeat until there are no more tasks.

The environment variable **HPM_ROUND_ROBIN_CLUSTER** is recognized only when the **average.so** aggregation plug-in is selected.

HPM_PRINT_TASK
Specifies the MPI task that has its results displayed. The default task number is zero. This environment variable is recognized only when the **single.so** aggregation plug-in is selected.

Miscellaneous environment variables

HPM_AGGREGATE
Specifies the name of a plug-in that defines the HPM data aggregation strategy. If the plug-in name contains a /, the name is treated as an absolute or relative path name. If the name does not contain a /, the plug-in is loaded following the rules for the **dlopen()** function call. The plug-in is a shared object file that implements the **distributor()** and **aggregator()** functions. See"Hardware performance counter plug-ins" on page 208 for more information.

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Files

name.**hpm.***app*.**txt**
The ASCII output from the **hpccount** invocation. This is a copy of the report displayed at the completion of **hpccount** execution.

name.**hpm.***app*.**viz**
An XML output file containing hardware performance counter data from **hpccount** execution. This file can be viewed using **peekperf**.

Examples

1. To list available counter groups for your processor, enter:

   ```
   hpccount -l
   ```

2. To report floating point load and store hardware counters for an application on POWER7 or POWER7+ hardware, enter:

   ```
   hpccount –o stats –g 146 testprog
   ```

3. To report floating point load and store hardware counters for an MPI program on POWER7 or POWER7+ hardware, enter:

   ```
   poe hpccount –o stats –g 146 –u testprog
   ```

hpcrun - Launch a program to collect profiling or trace data

Use the **hpcrun** command to launch a program and collect profiling or trace data from a subset of application tasks based on timing criteria.

Syntax

hpcrun [**--exmetric ELAPSED_TIME | CPU_TIME**] [**--excount** *num_tasks*] [**--tracestore memory |***mpi_fname*, *mio_fname*] *instrumented binary* [*args ...*]
hpcrun --help

Flags

--exmetric [ELAPSED_TIME | CPU_TIME]
Is the metric to be used to determine which task's data to collect. Specify **ELAPSED_TIME** to filter by elapsed (wall clock) time or **CPU_TIME** to filter by CPU time used. The default is **ELAPSED_TIME**.

--excount *num_tasks*
Is the number of tasks to be collected for minimum and maximum value of the metric specified by **--exmetric**. The default is **10** tasks.

--tracestore [memory |*mpi_fname*, *mio_fname*]
Specifies the mode for trace data collection. If memory is specified then trace data is held in each application task's memory until **MPI_Finalize** is called. If *mpi_fname*, *mio_fname* is specified, then this option specifies the path names of intermediate files where trace data from MPI trace and/or I/O trace are stored. The default is **memory**.

--help Displays a usage message.

Description

The **hpcrun** command is used to collect profiling and trace data from a subset of application tasks based on the metric the user specifies and the number of tasks that the user has selected for profiling and trace data collection. The **hpcrun** command can be invoked from the HPC Toolkit Eclipse plug-in, the **peekperf** GUI, or from the command line to run the application and collect performance data from the subset of application tasks.

Collecting profiling and trace data for a subset of tasks reduces the system resources required to process and store this data. It also reduces the volume of data that you have to analyze to evaluate your application's performance.

The subset of tasks is determined by querying each task for either the elapsed time or CPU time it consumed, depending on whether you specified the **ELAPSED_TIME** metric or the **CPU_TIME** metric.

The number of tasks is determined by the value the user specified for the **--exmetric** option. The **hpcrun** command will collect data for the number of tasks you specified with times closest to the minimum and maximum values for the metric you specified. Data will also be collected for tasks zero and the task closest to the average for the metric you specified if those tasks are not in the subsets closest to the minimum or maximum value of the metric you specified.

For example, if you specified **--excount 4** then the **hpcrun** command will collect data for the 4 tasks closest to the minimum metric value, the 4 tasks closest to the maximum metric value, and tasks zero and the task closest to the average metric value.

If you are collecting trace data, you have two options for handling that data. If you specify **--tracestore memory**, then the trace data for each application task is stored in that application's memory until **MPI_Finalize** is called. There is less processing overhead when trace data is stored in each task's memory, but this requires additional storage beyond what your application already uses.

For MPI trace, each traced MPI function call requires 52 bytes of memory. For I/O trace, each traced I/O system call requires approximately 44 bytes of memory.

If tracing requires too much memory, you can specify the **--tracestore** option with the *mpi_fname*, *mio_fname* parameters. If you specify this form of the **--tracestore** option, then MPI and/or I/O trace data will be stored in the respective user-specified temporary files where the pathname of the file is specified by the *mpi_fname* parameter for MPI trace and by the *mio_fname* parameter for I/O tracing.

Once trace processing is complete, **hpcrun** will process the temporary trace files to generate the final trace files and delete the temporary files.

If you want to use **hpcrun** to launch an AIX MPI program profiled with **-pg**, you must first add the **-hpcprof** flag when compiling the application using IBM PE Runtime compilation scripts.

Environment variables

HPC_EXCEPTION_COUNT
: Equivalent to the **--excount** option. The **--excount** option overrides this environment variable.

HPC_EXCEPTION_METRIC
: Equivalent to the **--exmetric** option. The **--exmetric** option overrides this environment variable.

HPC_TRACE_STORE
: Equivalent to the **--tracestore** option. The **--tracestore** option overrides this environment variable.

HPC_TRACE_MAX_BUFFERS
: Set this to a positive integer value that defines the number of in-memory buffers that are used to store trace events. When the in-memory buffers are full, a separate thread starts sending the data back to the home node. The default value is **1**. The values set by the environment variables **MAX_TRACE_EVENTS** (see "Controlling traced tasks" on page 216) and **HPC_TRACE_MAX_BUFFERS** buffers determine the maximum number of trace events that can be captured when an applications has been launched using **hpcrun**. The environment variable **HPC_TRACE_MAX_BUFFERS** only has an effect when you are *not* using the **memory** option of **HPC_TRACE_STORE** or **--tracestore**.

Miscellaneous environment variables

IHPCT_BASE

Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Related commands

"peekperf - Provides a GUI for the HPC Toolkit" on page 252

hpcstat - Reports a system-wide summary of hardware performance counter statistics

Use the **hpcstat** command to report a system-wide summary of hardware performance counter statistics.

Syntax

hpcstat [**-o** *name* [**-n**] [**-x**] [**-k**] [**-u**] [**-I** *time*] [**-U** *time*] [**-C** *count*] [**-g** *group*]
hpcstat [**-h**]
hpcstat [**-l**]
hpcstat [**-L**]
hpcstat [**-c**]

Flags

-c Lists the available counters and the hardware counter events that can be counted by each counter.

-C *count*
Specifies the number of times **hpcstat** reports statistics. The default is **1**.

-g *group*
A single value that specifies the hardware counter group to be used. If this flag or the **HPM_EVENT_SET** environment variable is not set, a processor-specific default group is used, as follows:

```
POWER6              127
POWER7, POWER7+     147
X86                   1
```

Note: Only CPU hardware counter groups can be specified when running **hpcstat**.

-h Displays a usage message.

-I *time* Specifies the interval, in seconds, for reporting statistics. The default sleep time is **10** seconds.

-k Specifies that only kernel side events are to be counted.

-l Lists the available hardware counter groups and the hardware counter events that are counted by each group.

-L Lists the available hardware counter groups and the counters contained in each group. The individual counter information will include a short name, the hardware event ID and a short text description. This flag is only available when running Linux.

-n Suppresses **hpcstat** output to stdout.

-o *name*
Writes output to a file with the prefix of *name*.

- The file prefix *name* can be specified using option **-o** or using the environment variable **HPC_OUTPUT_NAME**. The option takes precedence if there are conflicting specifications. The file prefix may contain a directory path. If not specified, the file prefix is **hpct**.
- The name *name* is expanded into different file names:
  - *name*.**hpm.hpcstat.txt** is the file name for ASCII output, which is a one-to-one copy of the screen output.
  - *name*.**hpm.hpcstat.viz** is the file name for the XML output.

- Which of these output files are generated is governed by the additional environment variables, **HPM_ASC_OUTPUT** and **HPM_VIZ_OUTPUT**. If neither of these are set, only the ASCII output is generated. If at least one is set, the following rules apply:
  - **HPM_ASC_OUTPUT**, if set to **yes**, triggers the ASCII output
  - **HPM_VIZ_OUTPUT**, if set to **yes**, triggers the XML output

-u Specifies that only user side events are to be counted.

-U *time*
Specifies the interval, in microseconds, for reporting statistics.

-x Displays formulas for derived metrics as part of the command output.

Description

The **hpcstat** tool reports system-wide hardware performance counter statistics and derived hardware metrics to **stdout** or to a file. If the **–C** flag and either the **–I** or **–U** flags are used, **hpcstat** reports hardware performance counter statistics on a periodic basis, similar to the **vmstat** command.

The **hpcstat** command requires the user to have **root** privileges.

Environment variables

Event selection environment variables

HPM_EVENT_SET
A single value that specifies the hardware counter group to be used. If the **–g** flag is not specified and **HPM_EVENT_SET** is not set, a processor-specific default group is used, as follows:

```
POWER6              127
POWER7, POWER7+     147
X86                   1
```

Note: Only CPU hardware counter groups can be specified when running **hpcstat**.

HPM_COUNTING_MODE
Specifies the CPU mode where counting will occur. Set this to a comma separated list of any combination of the following three possible values:

user Set to user to have user side events counted.

kernel Set to kernel to have kernel or system events counted.

hypervisor
Set to hypervisor to have hypervisor events counted.

The default setting for **hpcstat** is **user** and **kernel**.

Output control environment variables

HPM_ASC_OUTPUT
Determines whether or not to generate an ASCII output file. The output file name is:

name`.hpm.hpcstat.txt`

Valid values are **yes** or **no**. The default value for **HPM_ASC_OUTPUT** is **true**, except if **HPM_VIZ_OUTPUT** has been set, in which case the default value for **HPM_ASC_OUTPUT** is **false**.

HPC_OUTPUT_NAME
Specifies the *name* prefix of the output files:

name`.hpm.hpcstat.txt`

and

name`.hpm.hpcstat.viz`

For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309

HPM_PRINT_FORMULA
Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.

HPM_STDOUT
Set to **yes** to write ASCII output to stdout. If **HPM_STDOUT** is set to **no**, no output is written to stdout. The default is **yes**.

HPC_UNIQUE_FILE_NAME
Set to **yes** in order to generate unique file names for generated ASCII and XML output files. Set to **no** to generate the file name exactly as specified by **HPC_OUTPUT_NAME**. If **HPC_UNIQUE_FILE_NAME** is set to **yes**, the following rules apply:

- For serial programs, a string *_processId* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that do not use dynamic tasking, a string _0*_taskRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.
- For MPI programs that use dynamic tasking, a string *_worldID_worldRank* is appended to either the file prefix if no directory path is specified, or to the directory path if one is given.

HPM_VIZ_OUTPUT
Set to **yes** to generate an XML output file with the name:

name`.hpm.hpcstat.viz`

The default value for **HPM_VIZ_OUTPUT** is **false**.

Miscellaneous environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Files

name`.hpm.hpcstat.txt`
The ASCII output from the **hpcstat** invocation. This is a copy of the report displayed at the completion of **hpcstat** execution.

name`.hpm.hpcstat.viz`
An XML output file containing hardware performance counter data from **hpcstat** execution. This file can be viewed using **peekperf**.

Examples

1. To list hardware performance counter groups available on your processor, enter:

   ```
   hpcstat -l
   ```

2. To report floating point load and store hardware counters for the system every 30 seconds for five minutes on POWER7 or POWER7+ hardware, enter:

   ```
   hpcstat –u –C 10 –I 30 –g 146
   ```

hpctInst - Instrument applications to obtain performance data

Use the **hpctInst** command to instrument applications in order to obtain performance data.

Syntax

hpctInst [**-dhpm**] [**-dhpm_func_call** *file name*] [**-dhpm_region** *file name*]
[**-dmpi**] [**-dmpi_func_call** *file name*] [**-dmpi_region** *file name*]
[**-dmio**] [**-dpomp_parallel**{**none** | **EnterExit** | **BeginEnd**}]
[**-dpomp_user**{**none** | **BeginEnd**}] [**-dpomp_userfunc** *file name*]
[**-dpomp_loop**{**none** | **EnterExit** | **Chunks**}]
[**-dlink** *args*] *binary*
[**-h** | **--help**]

Note: This command applies only to Power Architecture.

Flags

-dhpm
Instrument all function entry and exit points with HPM instrumentation.

-dhpm_func_call *file name*
Instrument function call sites with HPM instrumentation, as specified by the contents of *file name*.

-dhpm_region *file name*
Instrument regions of code with HPM instrumentation, as specified by the contents of *file name*.

-dlink *args*
Specify additional libraries to link with the application, for AIX only. If your application uses shared libraries other than **libpmapi**, **libpthreads** or **libxlsmp**, you must specify those additional shared libraries using this option.

-dmpi
Instrument all MPI calls in the application with MPI profiling instrumentation.

-dmpi_func_call *file name*
Instrument MPI calls in functions at locations in the application, as specified by *file name*.

-dmpi_region *file name*
Instrument MPI calls in regions of code in the application, as specified by *file name*.

-dmio
Instrument all I/O calls in the application for I/O profiling.

-dpomp
Instrument all OpenMP parallel loop and parallel region constructs in the application for OpenMP profiling.

-dpomp_loop {none | EnterExit | Chunks}
Instrument parallel loops for OpenMP profiling.

-dpomp_parallel {none | EnterExit | BeginEnd}
Instrument parallel regions for OpenMP profiling.

-dpomp_user {none | BeginEnd}
Instrument user functions for OpenMP profiling.

-dpomp_userfunc *file name*
Instrument the user functions specified in *file name* for OpenMP profiling.

-h
--help Display an **hpctInst** usage message.

Description

The **hpctInst** command is used to rewrite an application with instrumentation as specified by command line flags and environment variables. The instrumented binary is written to a file called *binary*.**inst** in the current working directory. After an instrumented application has been created, set the appropriate environment variables for the instrumentation you have requested, then run the application.

Note: **hpctInst** is not supported for x86 applications.

If you invoke **hpctInst** with the **-dhpm_func_call** or the **-dmpi_func_call** flag, the format of each line in the file specified by *file name* is one of:

```
called_func[inst_function]
called_func [file_name [start_line [end_line]]]
```

where:

called_func
Is the name of the function being called. For the **-dmpi_func_call** option, *called_func* is the name of an MPI function.

inst_function
Is the name of the only function in which calls to *called_func* are instrumented.

file_name
Is the name of the only source file in which calls to *called_func* are instrumented.

start_line
Is the first line number in file name in which calls to *called_func* are instrumented.

end_line
Is the ending line number in file name in which calls to *called_func* are instrumented.

If *file_name* is specified, and both *start_line* and *end_line* are omitted, all calls to *called_func* in *file_name* are instrumented. The same *called_func* might be specified in one or more lines in this file.

If you invoke **hpctInst** with the **-dhpm_region** or **-dmpi_region** flags, the format of the each line in the file specified by *file_name* is:

```
file_name start_line end_line
```

where:

file_name
Is the name of the source file.

start_line
is the starting line number in *file_name* that will be instrumented.

end_line
Is the ending line number in *file_name* that will be instrumented.

The meanings of the OpenMP instrumentation flags, **-dpomp_***, are:

none No data is collected for this construct.

EnterExit
Data is collected for parallel region entry and exit and for loop entry and exit.

BeginEnd
Data is collected at begin and end of a parallel region. **EnterExit** data is also collected.

Chunks
Data is collected for each parallel execution of the region or loop.

If your application does not reside in a global file system, copy the instrumented binary to all nodes on which it will run. If you instrument an application for HPM, **hpctInst** creates a file named **.psigma.hpmhandle** in the same directory as the instrumented binary. If you instrument an application for OpenMP profiling, **hpctInst** creates a file named **.psigma.dpomphandle** in the same directory as the instrumented binary. You must ensure these files are accessible in the current working directory on all nodes on which the instrumented application will execute.

The application being instrumented must be compiled and linked with the **-g** flag. If you have a Power Linux application, it must also be linked with these flags:

```
-Wl,--hash-style=sysv -emit-stub-syms
```

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

LD_LIBRARY_PATH
Must be set to **$IHPCT_BASE/lib** when instrumenting 32-bit applications on Linux. Must be set to **$IHPCT_BASE/lib64** when instrumenting 64-bit applications on Linux.

POMP_LOOP
Specifies the level of OpenMP profiling instrumentation for OpenMP parallel regions. Values can be **none**, **EnterExit**, or **Chunks**. The **–dpomp_loop** flag overrides this environment variable.

POMP_PARALLEL
Specifies the level of OpenMP profiling instrumentation for OpenMP parallel regions. Values can be **none**, **EnterExit**, or **BeginEnd**. The **–dpomp_parallel** flag overrides this environment variable.

POMP_USER
Specifies the level of OpenMP profiling for user functions. Values might be **none** or **EnterExit**. The **–dpomp**_user flag overrides this environment variable.

Examples

1. To instrument all MPI calls in an application, enter:

   hpctInst –dmpi *testprog*

2. To instrument call sites for function **testfunc** from file **main.c** lines 1 through 100 with HPM instrumentation, enter:

   hpctInst –dhpm_func_call inst_spec *testprog*

   where the **inst_spec** file contains the single line:

   testfunc main.c 1 100

3. To instrument all parallel loops and parallel regions in an OpenMP application with **EnterExit** instrumentation, enter:

   hpctInst –dpomp_loop –dpomp_parallel *testprog*

peekperf - Provides a GUI for the HPC Toolkit

Use the **peekperf** command to instrument an application and to view application performance data.

Syntax

peekperf [**-num** *max_src_files*] [*binary*] [[*.*viz*] ...]
peekperf [**-h** | **--help**]

Flags

-num *max_src_files*
Specifies the maximum number of source files that can be opened without being prompted to select a set of files. The default limit is **15** files.

-h
--help Displays a usage message for **peekperf**.

Description

The **peekperf** GUI is the X Windows control center of the HPC Toolkit. It allows you to control the instrumentation, execute the application, and visualize and analyze the collected performance data within the same user interface. The **peekperf** GUI can display the data in the visualization (**.viz**) files from the various instrumentation libraries. If more than one visualization file is specified, **peekperf** combines the data from them for display. **peekperf** also provides filtering and sorting capabilities to help you analyze the data.

peekperf supports gathering the following types of performance data:

- Hardware Performance Monitor (HPM)
- MPI profiling and trace (MPI)
- OpenMP profiling (OpenMP)
- I/O profiling (MIO)

If you specify the application to be instrumented, using the *binary* parameter, **peekperf** parses the specified application executable to identify possible instrumentation points. If you do not specify the application executable on the command line, you can specify the application executable once **peekperf** has started by selecting the **Open Binary** selection from the **File** menu.

Note: When using **peekperf** with x86 applications, the options to open a binary and generate an instrumented application are not available.

If you specify the visualization file(s) on the command line, **peekperf** tries to load the visualization files as it starts up. You can also load visualization files by selecting the **Open Performance Data** selection from the **File** menu.

The **peekperf** GUI has three main windows:

- **Data collection**

  The data collection window opens when you open an executable, either by specifying it on the command line, or by selecting **Open Binary** from the **File** menu. The data collection window shows you separate tabs for each type of instrumentation data you can collect. Each tab shows a window containing a representation of the application structure and the locations within the

application that you can instrument for each type of data. You can expand or collapse nodes in this tree, as needed, and select or deselect the desired instrumentation points. If you right-click on a node, a pop-up menu appears in which you can select additional options, including options specific to the type of instrumentation.

- **Data visualization**

  The data visualization window shows a summary of the performance data files that have been loaded. Performance data files (visualization files) can be opened by specifying them on the **peekperf** command line, by selecting **Open Performance Data** from the **File** menu, or they will open automatically after an instrumented application completes. You can expand or collapse the tree to view data. You can sort data by clicking on a column heading. You can view additional detail in a metrics browser for a leaf node row in the tree by right-clicking over the leaf node row. If you right-click anywhere in the window other than over a leaf node row, a pop-up menu appears where you can apply filters to eliminate uninteresting rows from the view or to open a table or trace viewer appropriate to the type of visualization data.

- **Source code**

  The source code window appears when an executable is loaded or by clicking the **Open Sources** option in the **File** menu. The source code window shows a tabbed display of individual source files in the application. As you select instrumentation points in the data collection window, or select rows in the data visualization window, the corresponding lines of code are highlighted in the source code window. You can select regions of code to be instrumented by the HPM tool by clicking and dragging over the region of interest, then right-clicking and selecting **Add to HPM** from the pop-up menu.

It is necessary to compile the application with the **-g** flag in order to map the performance data back to source code. This allows you to more easily find bottlenecks and points for optimizations.

Environment variables

Miscellaneous environment variables

IHPCT_BASE
: Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Examples

1. To invoke **peekperf** without loading any performance data or parsing an application executable, enter:

   ```
   peekperf
   ```

2. To invoke **peekperf**, specifying an application executable to be parsed for instrumentation, enter:

   ```
   peekperf myapp
   ```

3. To invoke **peekperf** to view MPI performance data, enter:

   ```
   peekperf hpct*.mpi.viz
   ```

peekview - Display an MPI trace file

Use the **peekview** command to display an MPI trace file generated by an application that is instrumented by the IBM HPC Toolkit to obtain MPI profiling data.

Syntax

peekview [[**-b** | **--bright**] | [**-d** | **--dark**]] *tracefile* [*trace-file*] ...
peekview [**-h**]
peekview [**--h**]

Flags

-b
--bright
Displays the MPI trace with a bright background.

-d
--dark Displays the MPI trace with a dark background. This is the default display option.

-h
--help Displays a **peekview** usage message.

Description

The **peekview** X Windows GUI displays one or more MPI trace files that are generated using the MPI profiling tool in the IBM HPC Toolkit. Traces are displayed in a timeline format with the x axis representing elapsed time and the y axis representing MPI task index. Each unique MPI function is displayed in a different color so individual MPI function calls can be easily identified.

The **peekview** GUI displays a second window, the identifier window, which you can use to enable or disable the display of MPI events by MPI function type, so only the desired MPI function types are visible.

You can scroll through the trace display by using the scrollbars on the trace window. You can also zoom in and out of the trace by using the zoom icons at the top of the trace window, or by clicking and dragging a rubber-band box around a region of interest in the trace window.

If you left-click over a trace event, some information about that event is displayed in the status area at the bottom of the trace window. If you right-click over a trace event, a pop-up message with additional information about that event appears.

If **peekview** is invoked from within **peekperf**, by selecting the **View Tracer** option from the Data Visualization Window pop-up menu that appears when right-clicking in that window, then when you left-click on a trace event in the trace window, **peekperf** scrolls its source code window to display the line of source code corresponding to the MPI function call.

By default, **peekview** displays the trace using a dark background (the **–d** flag). If the **–b** flag is specified, each MPI event is drawn with a black rectangle around it, making it easier to see short duration MPI events. However, if there are many events in the MPI trace display, the **–b** flag could result in a distorted view of the MPI trace.

Environment variables

Miscellaneous environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

To display an MPI trace with a dark background, enter:

```
peekview –d hpct_0_0.mpi.mpt
```

Xprof - Display the application as a function tree

Use the **Xprof** command, which is a GUI-based performance profiling tool, to display the application as a function tree based on the run-time call structure and profiling statistics.

Syntax

Xprof [**-b**] [**-s**] [**-z**] [**-a** *path*] [**-c** *file*] [**-L** *pathname*]
[[**-e** *function*]...] [[**-E** *function*]...] [[**-f** *function*]...] [[**-F** *function*]...]
[**-disp_max** *number_of_functions*] [**program**] [[**gmon.out**]...]
Xprof [**-h**]
Xprof [**--help**]

Note: The **Xprof** command is only supported on Power Architecture.

Flags

-a *path* Specifies an alternate search path or paths for library files and source code files. If more than one path is specified, the paths must be embraced by a comma (,) and each path should be separated by either a colon (:) or a space.

-b Suppresses the printing of the field descriptions for the Flat Profile, Call Graph Profile, and Function Index reports when they are written to a file with the **Save As** option of the **File** menu.

-c *file* Loads a configuration file that contains information to be used to determine which functions will be displayed when **Xprof** is brought up.

-disp_max *number_of_functions*
Sets the number of function boxes that **Xprof** initially displays in the function call tree. The value supplied with this flag can be any integer between **0** and **5,000**. **Xprof** displays the function boxes for the most processor-intensive functions through the number you specify.

-e *function*
De-emphasizes the general appearance of the function box or boxes for the specified functions in the function call tree, and limits the number of entries for these functions in the Call Graph Profile report. This also applies to the specified function's descendants, as long as they have not been called by non-specified functions. In the function call tree, the function boxes for the specified functions appear grayed out. Its size and the content of the label remain the same. This also applies to descendant functions, as long as they have not been called by non-specified functions. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. The information for this entry remains unchanged. Entries for descendants of the specified function do not appear unless they have been called by at least one non-specified function in the program.

-E *function*
Changes the general appearance and label information of the function box or boxes for the specified functions in the function call tree. It also limits the number of entries for these functions in the Call Graph Profile report, and changes the processor data associated with them. These results also apply to the specified function's descendants, as long as they have not been called by non-specified functions in the program. In the function call

tree, the function box for the specified function appears grayed out, and its size and shape also changes so that it appears as a square of the smallest allowable size. In addition, the processor time shown in the function box label appears as **0** (zero). The same applies to function boxes for descendant functions, as long as they have not been called by non-specified functions. This option also causes the processor time spent by the specified function to be deducted from the left side processor total in the label of the function box for each of the specified function's ancestors. In the Call Graph Profile report, an entry for the specified function only appears where it is a child of another function, or as a parent of a function that also has at least one non-specified function as its parent. When this is the case, the time in the self and descendants columns for this entry is set to 0 (zero). In addition, the amount of time that was in the descendant's column for the specified function is subtracted from the time listed under the descendant's column for the profiled function. As a result, be aware that the value listed in the % time column for most profiled functions in this report will change.

-f *function*
De-emphasizes the general appearance of all function boxes in the function call tree, except for that of the specified function(s) and its descendant(s). In addition, the number of entries in the Call Graph Profile report for the non-specified functions and non-descendant functions is limited. The **-f** flag overrides the **-e** flag. In the function call tree, all function boxes except for that of the specified function(s) and its descendant(s) appear grayed out. The size of these boxes and the content of their labels remain the same. For the specified function(s), and its descendants, the appearance of the function boxes and labels remain the same. In the Call Graph Profile report, an entry for a non-specified or non-descendant function only appears where it is a parent or child of a specified function or one of its descendants. All information for this entry remains the same.

-h
--help Writes the **Xprof** usage message to STDERR and then exits.

-L *pathname*
Uses an alternate path name for locating shared libraries. If you plan to specify multiple paths, use the **Set File Search Paths** option of the **File** menu on the **Xprof** GUI.

-s If multiple **gmon.out** files are specified when **Xprof** is started, it produces the **gmon.sum** profile data file. The **gmon.sum** file represents the sum of the profile information in all the specified profile files.

Note: If you specify a single **gmon.out** file, the **gmon.sum** file contains the same data as the **gmon.out** file.

-z Includes functions that have both zero processor usage and no call counts in the Flat Profile, Call Graph Profile, and Function Index reports. A function does not have a call count if the file that contains its definition was not compiled with the **-pg** flag, which is common with system library files.

Description

The **Xprof** command is used to analyze the performance of serial and parallel applications at the function, source line, and machine code levels. **Xprof** uses data collected after running a program that is compiled and linked with the **-pg** flag. It

presents a graphical representation of the application functions in addition to providing textual data in several report windows. These presentation formats are intended to allow easy identification of the functions that are most processor-intensive.

Functions are represented by solid green boxes or nodes in the function call tree. The size and shape of each function node indicates its CPU usage. The height of each node represents the amount of CPU time it spent on executing itself. The width of each node represents the amount of CPU time it spent executing itself, plus its descendant functions. More detail such as source code mapping or statistical data can be displayed by selecting an individual function.

The calls made between each of the functions in the function call tree are represented by blue arrows extending between their corresponding function boxes. These lines are called call arcs. Selecting an arc shows the number of calls, the callee, and the caller.

The call graph can be filtered in a variety of ways including by CPU time, call counts, and function name. It can also be pruned or expanded based on library, ancestor, and descendant relationship.

Limitations

If you compile your application on one processor and analyze it on another, you must first make sure that both processors have similar library configurations, at least for the system libraries used by the application.

Because **Xprof** collects data by sampling, functions that run for a short amount of time might not show any CPU use.

On AIX systems, **Xprof** supports only the old format (prior to 5.3) **gmon.out** file; but it is easy to have the application program generate these. Refer to the **GPROF** environment variable for details.

The disassembly view of 64-bit Linux programs is not supported.

Environment variables

GPROF

Sets options for generating the **gmon.out** file. To have an effect, this environment variable must be set before running the user application.

```
GPROF=profile:profile-type,
    scale:scaling-factor,file:file-type,
    filename:file-name
```

Note: This environment variable applies only to AIX.

Where:

profile-type

Describes the type of profiling that needs to be performed. This can be either **process** or **thread**, where:

process
Indicates that profiling granularity is at process level

thread Indicates that profiling granularity is at thread level.

scaling-factor
Describes how much memory needs to be allocated for the Call Graph Profile. By default, the scaling factor is 2 for process-level profiling and 8 for thread-level profiling. A scaling factor of 2 indicates that a memory of 1/2 of the size of the process is allocated for every process or thread. A scaling factor of 8 indicates that a memory of 1/8th of the size of the process is allocated for every process or thread. This memory is the buffer area used to store the call graph information.

file-type
Describes the type of **gmon.out** file required. A value of **multi** indicates that one **gmon.out** file per process is required. A value of **multithread** indicates that one **gmon.out** file per thread is required. If an application is profiled with the **-pg** flag, and it does a fork, specifying **multi** generates one **gmon.out** file for the parent process and another for the child process. The naming convention for the generated **gmon.out** files is as follows:

- For **multi** file-type:

  prefix-**processname-pid.out**
- For **multithread** file-type:

  prefix-**processname-pid-Pthread***threadid*.**out**

file-name describes the prefix that needs to be used for the generated **gmon.out** files. By default, the prefix is **gmon**.

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Examples

1. To use **Xprof**, you must first compile your program (for example, **foo.c**) with **-pg**:

   ```
   xlc -pg -o foo foo.c
   ```

   When the program **foo** is executed, a **gmon.out** file will be generated. To invoke **Xprof**, enter:

   ```
   Xprof foo gmon.out
   ```

2. To use **Xprof** on your MPI program, you must first compile your program (for example, **foo.c**) with **-pg**.

   Note: To be able to map the functions displayed in the call graph by **Xprof** back to the source code, your program must also be compiled with **-g**.

   ```
   mpcc -g -pg -o foo foo.c
   ```

   When the MPI program ring is executed on AIX, by default, one **gmon.***worldId_rankId***.out** file is generated for each task involved in the parallel execution. To invoke **Xprof**, enter:

   ```
   Xprof ring gmon.*.out
   ```

   When the MPI program ring is executed on pLinux, a **gmon** file named **profdir**.*worldId_rankId*/**gmon.out** is generated for each task involved in the parallel execution. To invoke **Xprof**, enter:

   ```
   Xprof ring profdir*/gmon.out
   ```

   When invoking **Xprof** with multiple **gmon.out** files, they can be displayed as a summary or average of their performance statistics.

Chapter 19. Application programming interfaces

Table 53 lists the application programming interfaces (APIs) and what they are used for:

Table 53. APIs

Used for:	API
Hardware performance monitoring	• "hpm_error_count, f_hpm_error - Verify a call to a libhpc function" on page 263 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265 • "hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code" on page 278 • "hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code" on page 271 • "hpmStop, f_hpmstop - Identify the end point of an instrumented region of code" on page 274 • "hpmTerminate, f_hpmterminate - Generate HPM statistic files and shut down HPM" on page 276 • "hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code" on page 278 • "hpmTstartx, f_hpmtstartx - Identify the starting point for an instrumented region of code" on page 280 • "hpmTstop, f_hpmtstop - Identify the end point of an instrumented region of code" on page 283

© Copyright IBM Corp. 2008, 2013

Table 53. APIs (continued)

Used for:	API
MPI profiling customization	• "MT_get_allresults - Obtain statistical results" on page 285 • "MT_get_calleraddress - Obtain the address of the caller of an MPI function" on page 288 • "MT_get_callerinfo - Obtain source code information" on page 289 • "MT_get_elapsed_time - Obtains elapsed time" on page 291 • "MT_get_environment - Returns run-time environment information" on page 292 • "MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred" on page 293 • "MT_get_mpi_counts - Obtain the the number of times a function was called" on page 294 • "MT_get_mpi_name - Returns the name of the specified MPI function" on page 295 • "MT_get_mpi_time - Obtain elapsed time" on page 296 • "MT_get_time - Get the elapsed time" on page 297 • "MT_get_tracebufferinfo - Obtain information about MPI trace buffer usage" on page 298 • "MT_output_text - Generate performance statistics" on page 299 • "MT_output_trace - Control whether an MPI trace file is created" on page 300 • "MT_trace_event - Control whether an MPI trace event is generated" on page 301 • "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303 • "MT_trace_stop, mt_trace_stop - Suspend the collection of trace events" on page 305

hpm_error_count, f_hpm_error - Verify a call to a libhpc function

Use **hpm_error_count** or **f_hpm_error** to verify that a call to a **libhpc** function was successful.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
logical function f_hpm_error()
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

None.

Description

hpm_error_count is an external variable that HPM library functions set if an error occurs during a call to that function. The **f_hpm_error**() function is the equivalent FORTRAN function that returns a logical value indicating that an error occurred during an HPM library call.

If an HPM library call is successful, **hpm_error_count** is set to zero and **f_hpm_error**() returns **.false**. If an HPM library call fails, **hpm_error_count** is set to a non zero value and **f_hpm_error**() returns **.true**.

The **hpm_error_count** variable or **f_hpm_error**() function should be used at any point where you need to determine if an HPM library function call failed.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

None.

Examples

```
#include <libhpc.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
```

```
    hpmInit(0, "HPMTest");
    if (hpm_error_count) {
      printf("hpmInit error\n");
      exit(1);
    }
    .
    .
    .
    hpmTerminate(0);
    if (hpm_error_count) {
      printf("hpmTerminate error\n");
      exit(1);
    }
}
```

```
program hpmtest
#include "f_hpc.h"
call f_hpminit(0, 'HPMTest')
if (f_hpm_error() .eqv. .true.) then
  print *, 'f_hpminit error'
  stop 1
end if
.
.
.
call f_hpmterminate(0)
if (f_hpm_error() .eqv. .true.) then
  print *, 'f_hpmterminate error'
  stop 1
end if
end
```

hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment

Use **hpmInit** or **f_hpminit** to initialize the Hardware Performance Monitor (HPM) run-time environment.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmInit(int my_ID, const char *progName)
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpminit(integer my_ID, character progName(*))
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

my_ID Unused. Should be set to zero.

progName
Specifies the name of the program. If the **HPC_OUTPUT_NAME** environment variable is not set, this parameter is used as the file name prefix. If it is NULL, then the value **hpct** is used.

Description

This function initializes the run-time environment for obtaining hardware performance counter statistics. It optionally names the output files containing these statistics. It must be the first HPM function call executed in the application.

For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

Event selection environment variables

HPM_EVENT_SET
A single value that specifies the hardware counter group to be used, or a comma-delimited list of hardware counter groups. When multiple CPU

hardware counter groups are specified, they will be multiplexed. If **HPM_EVENT_SET** is not set, a processor-specific default group is used, as follows:

```
POWER6              127
POWER7, POWER7+     147
X86                   1
```

HPM_EXCLUSIVE_VALUES
Set to **yes** if exclusive counter values in nested counter regions are to be computed.

HPM_SLICE_DURATION
Specifies the interval, in milliseconds, to be used when CPU hardware counter groups are multiplexed. You can specify a value between **10** milliseconds and **30** seconds. The default is **100** milliseconds. This environment variable is supported for AIX only.

HPM_COUNTING_MODE
Specifies the CPU mode where counting will occur. Set this to a comma-separated list of any combination of the following three possible values:

user Set to user to have user-side events counted.

kernel Set to kernel to have kernel or system events counted.

hypervisor
Set to hypervisor to have hypervisor events counted.

The default setting for **libhpc.a** is **user**.

Output control environment variables

HPM_ASC_OUTPUT
Determines whether or not to generate an ASCII output file. The output file name is:

name.**hpm**[.*progName*].**txt**

where:

name
Is obtained from the **HPC_OUTPUT_NAME** environment variable.

progName
Is specified if the second parameter to the **hpmInit()** call is non-NULL.

Valid values are **yes** or **no**.

If neither **HPM_ASC_OUTPUT** or **HPM_VIZ_OUTPUT** are set, both the ASCII and XML output files are generated. If at least one is set, the following rules apply:

- **HPM_ASC_OUTPUT**, if set to **yes**, triggers the ASCII output
- **HPM_VIZ_OUTPUT**, if set to **yes**, triggers the XML output

HPC_OUTPUT_NAME
Specifies the *name* prefix of the output files:

name.**hpm**[.*progName*].**txt**

and

name.**hpm**[.*progName*].**viz**

where:

progName
Is specified if the second parameter to the **hpminit()** call is non-NULL.

If this environment variable is not set, then *name* is set to the value of the **hpct**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

HPM_PRINT_FORMULA
Set to **yes** to print the definitions of the derived metrics. Set to **no** to suppress this output. The default is **no**.

HPM_STDOUT
Set to **yes** to write ASCII output to stdout. If **HPM_STDOUT** is set to **no**, no output is written to stdout. The default is **yes**.

HPC_UNIQUE_FILE_NAME
Set to **yes** in order to generate unique file names for generated ASCII and XML output files. Set to **no** to generate the file name exactly as specified by **HPC_OUTPUT_NAME**. For more information about file naming conventions, see Appendix A, "Performance data file naming," on page 309.

HPM_VIZ_OUTPUT
Set to **yes** to generate an XML output file with the name:

name.hpm[.*progName*].viz

where:

name
Is specified by the **HPC_OUTPUT_NAME** environment variable.

progName
Is specified if the second parameter to the **hpmInit()** call is non-NULL.

If neither **HPM_ASC_OUTPUT** or **HPM_VIZ_OUTPUT** are set, both the ASCII and XML output files are generated. If at least one is set, the following rules apply:

- **HPM_ASC_OUTPUT**, if set to **yes**, triggers the ASCII output
- **HPM_VIZ_OUTPUT**, if set to **yes**, triggers the XML output

Plug-in specific environment variables

HPM_ROUND_ROBIN_CLUSTER
HPM_ROUND_ROBIN_CLUSTER allows setting the number of groups distributed per task.

Without the environment variable **HPM_ROUND_ROBIN_CLUSTER** set, the **average.so** plug-in will distribute the group numbers from **HPM_EVENT_SET** in a round-robin fashion, one group to each of the MPI tasks in the application.

The default value for **HPM_ROUND_ROBIN_CLUSTER** is **1**. The default will be used if a value less than 1 is specified.

The number of groups spread out round-robin fashion to the tasks will be limited to the first "number of tasks times the setting of **HPM_ROUND_ROBIN_CLUSTER**" groups.

If a value greater than the number of groups in **HPM_EVENT_SET** is specified, **HPM_ROUND_ROBIN_CLUSTER** will be set to the number of groups specified in **HPM_EVENT_SET**.

It is possible that the number of groups does not distribute evenly to the tasks. The first task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups in **HPM_EVENT_SET**. If there are more tasks and groups left, the second task will get at most **HPM_ROUND_ROBIN_CLUSTER** of the groups left in **HPM_EVENT_SET** and so on, until there are no groups unused in **HPM_EVENT_SET**. After the groups in **HPM_EVENT_SET** have been used once, and there are more tasks, this process will repeat until there are no more tasks.

The environment variable **HPM_ROUND_ROBIN_CLUSTER** is recognized only when the **average.so** aggregation plug-in is selected.

HPM_PRINT_TASK
Specifies the MPI task that has its results displayed. The default task number is zero. This environment variable is recognized only when the **single.so** aggregation plug-in is selected.

Miscellaneous environment variables

HPM_AGGREGATE
Specifies the name of a plug-in that defines the HPM data aggregation strategy. If the plug-in name contains a /, the name is treated as an absolute or relative path name. If the name does not contain a /, the plug-in is loaded following the rules for the **dlopen()** function call. The plug-in is a shared object file that implements the **distributor()** and **aggregator()** functions. See "Hardware performance counter plug-ins" on page 208 for more information.

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    hpmInit(0, "HPMTest");
    .
    .
    .
    hpmTerminate(0);
}
```

```
program hpmtest
#include "f_hpc.h"
call f_hpminit(0, 'HPMTest')
.
.
.
call f_hpmterminate(0)
end
```

hpmStart, f_hpmstart - Identify the starting point for an instrumented region of code

Use **hpmStart** or **f_hpmstart** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmStart(int inst_ID, const char *label);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmstart(integer inst_ID, character label(*))
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region.

label
: Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmStart**() function identifies the start of a region of code in which hardware performance counter events are to be counted. The end of the region is identified by a call to **hpmStop**() using the same *inst_ID*.

The **hpmStart**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmStop**() function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmStart**() and **hpmStop**() calls can be nested. When regions are nested, **hpmStart**() and **hpmStop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting.

If **hpmStart**() and **hpmStop**() functions are called in a threaded application, the count of hardware performance counter events is for the entire process rather than

for the specific thread on which the calls are made. If you need accurate counts for each thread, use **hpmTstart**() and **hpmTstop**().

The include files are located in the **${IHPCT_BASE}/include** directory.

Environment variables

See "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265.

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
    float x;
    x = 10.0;
    hpmInit(0, "HPMTest");
    hpmStart(1, "Region 1");
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate(0);
}
```

```
    program hpmtest
    #include "f_hpc.h"
    integer i
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = 10.0;
    call f_hpmstart(1, 'Region 1')
    do 10 i = 1, 100000
      x = x / 1.001
10 continue
    call f_hpmstop(1)
    call f_hpmterminate(0)
    end
```

hpmStartx, f_hpmstartx - Identify the starting point for an instrumented region of code

Use **hpmStartx** or **f_hpmstartx** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted, specifying explicit inheritance relationships for nested instrumentation regions.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmStartx(int inst_ID, int parent_ID, const char *label);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstartx(integer inst_ID, integer parent_ID, character label(*))
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region.

parent_ID
: Specifies the inheritance relationship for nested **hpmStart**() calls. This parameter must have one of the following values:
  - **HPM_AUTO_PARENT**
  - **HPM_ONLY_EXCLUSIVE**
  - **HPM_NO_PARENT**
  - The *inst_ID* of an active **hpmStart**() or **hpmStartx**() call

label
: Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmStartx**() function identifies the start of a region of code in which hardware performance counter events are to be counted, and explicitly specifies the parent relationship for an encompassing region instrumented by **hpmStart**() or **hpmStartx**(). The end of the region is identified by a call to **hpmStop**() using the same *inst_ID*.

The **hpmStartx**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmStop**()

function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmStartx**() and **hpmStop**() calls can be nested. When regions are nested, **hpmStartx**() and **hpmStop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting. For reporting of exclusive event counts, the proper parent relationship must be determined. If regions are perfectly nested, such as a set of nested loops, **hpmStart**() is sufficient for determining parent relationships. In more complicated nesting cases, **hpmStartx**() should be used to properly specify those relationships.

Parent relationships are specified by the **parent_ID** parameter which must have one of the following values:

HPM_AUTO_PARENT
: Automatically determine the parent for this **hpmStartx**() call. This is done by searching for the immediately preceding **hpmStart**() or **hpmStartx**() call executed on the current thread in which there has not been a corresponding call made to **hpmStop**().

HPM_ONLY_EXCLUSIVE
: This operates in the same way as if **HPM_AUTO_PARENT** was specified, and also acts as if the **HPM_EXCLUSIVE_VALUES** environment variable was set for this call to **hpmStartx**() only. If the **HPM_EXCLUSIVE_VALUES** environment variable was previously set, this parameter value is equivalent to specifying **HPM_AUTO_PARENT**.

HPM_NO_PARENT
: Specifies that this **hpmStartx**() call has no parent.

inst_ID
: For a previous **hpmStart**() or **hpmStartx**() call that is currently active, meaning the corresponding call to **hpmStop**() has not been made for this instance of execution.

If **hpmStartx**() and **hpmStop**() functions are called in a threaded application, the count of hardware performance counter events is for the entire process rather than for the specific thread on which the calls were made. If you need accurate counts for each thread, use **hpmTstartx**() and **hpmTstop**().

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

See “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 265.

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
    int j;
    float x;
    x = 10.0;
    hpmInit(0, "HPMTest");
    hpmStartx(1, HPM_NO_PARENT, "Region 1");
```

```
    for (i = 0; i < 100000; i++) {
       hpmStartx(2, 1, "Region 2");
       for (j = 0; j < 100000; j++) {
         x = x / 1.001;
       }
       hpmStop(2);
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate(0);
}
```

```
    program hpmtest
    #include "f_hpc.h"
    integer i
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = 10.0;
    call f_hpmstartx(1, HPM_NO_PARENT, 'Region 1')
    do 10 i = 1, 100000
      x = x / 1.001
 10 continue
    call f_hpmstop(1)
    call f_hpmterminate(0)
       end
```

hpmStop, f_hpmstop - Identify the end point of an instrumented region of code

Use **hpmStop** or **f_hpmstop** to identify the end point of an instrumented region of code starting with a call to **hpmStart**() or **hpmStartx**(), in which hardware performance counter events are to be counted. It also accumulates hardware performance counter events for that region.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmStop(int inst_ID);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmstop(integer inst_ID)
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region. This value must match the *inst_ID* specified in the corresponding **hpmStart**() or **hpmStartx**() function call.

Description

The **hpmStop**() function identifies the end of a region of code in which hardware performance counter events are to be monitored. The start of the region is identified by a call to the **hpmStart**() or **hpmStartx**() function using the same *inst_ID*. That function must be called for a specific *inst_ID* before the corresponding call to the **hpmStop**() function.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

None.

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
```

```
    float x;
    x = 10.0;
    hpmInit(0, "HPMTest");
    hpmStart(1, "Region 1");
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmStop(1);
    hpmTerminate(0);
}
```

```
    program hpmtest
    #include "f_hpc.h"
    integer i
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = 10.0;
    call f_hpmstart(1, 'Region 1')
    do 10 i = 1, 100000
      x = x / 1.001
 10 continue
    call f_hpmstop(1)
    call f_hpmterminate(0)
    end
```

hpmTerminate, f_hpmterminate - Generate HPM statistic files and shut down HPM

Use **hpmStop** or **f_hpmstop** to generate Hardware Performance Monitor (HPM) statistics files and to shut down the HPM environment.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmTerminate(int my_ID)
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmterminate(integer my_ID)
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

my_ID Unused. Should be set to zero.

Description

This function generates output files containing any hardware performance counter statistics obtained during the program's execution and shuts down the HPM runtime environment. This function must be called before the application exits in order to generate statistics. It must be the last HPM function called during program execution.

If the **HPM_AGGREGATE** environment variable is set, and the instrumented application is an MPI application, **hpmTerminate**() should be called before **MPI_Finalize()** is called, because the plug-in specified by the **HPM_AGGREGATE** environment variable might call MPI functions as part of its internal processing.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

See "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265.

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
```

```
    hpmInit(0, "HPMTest");
    .
    .
    .
    hpmTerminate(0);
}

program hpmtest
#include "f_hpc.h"
call f_hpminit(0, 'HPMTest')
.
.
.
call f_hpmterminate(0)
end
```

hpmTstart, f_hpmtstart - Identify the starting point for an instrumented region of code

Use **hpmTstart** or **f_hpmtstart** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted on a per-thread basis.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmTstart(int inst_ID, const char *label);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstart(integer inst_ID, character label(*))
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region.

label
: Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmTstart**() function identifies the start of a region of code in which hardware performance counter events are to be counted. The end of the region is identified by a call to **hpmTstop**() using the same **inst_ID**.

The **hpmTstart**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware performance counters that are being used. When the corresponding **hpmTstop**() function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmTstart**() and **hpmTstop**() calls can be nested. When regions are nested, **hpmTstart**() and **hpmTstop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting.

The only difference between the **hpmStart**() and **hpmTstart**() functions is that a call to **hpmStart**() results in reading the hardware performance counters for the entire

process while a call to **hpmTstart**() results in reading the hardware performance counters only for the thread from which the call to **hpmTstart**() was made.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

See "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265.

Examples

```
#include <libhpc.h>
void thread_func();
int main(int argc, char *argv[])
{
    hpmInit(0, "HPMTest");
    thread_func(); /* assume this function runs on
                      multiple threads */
    hpmTerminate(0);
}
void thread_func()
{
    int i;
    float x;
    x = 10.0;
    hpmTstart(1, "Region 1");
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmTstop(1);
}
```

```
    program hpmtest
    #include "f_hpc.h"
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = thread_func() ! Assume thread_func runs on
                      ! multiple threads
    call f_hpmterminate(0)
    end

  real*4 function thread_func()
  #include "f_hpc.h"
  real*4 x
   integer i
   x = 10.0;
   call f_hpmtstart(1, 'Region 1')
   do 10 i = 1, 100000
      x = x / 1.001
10 continue
   call f_hpmtstop(1)
   thread_func = x
  return
   end
```

hpmTstartx, f_hpmtstartx - Identify the starting point for an instrumented region of code

Use **hpmStartx** or **f_hpmstartx** to identify the starting point for an instrumented region of code in which hardware performance counter events are to be counted on a per-thread basis, specifying explicit inheritance relationships for nested instrumentation regions.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmTstartx(int inst_ID, int parent_ID, const char *label);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstartx(integer inst_ID, integer parent_ID, character label(*))
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region.

parent_ID
: Specifies the inheritance relationship for nested **hpmStart**() calls. This parameter must have one of the following values:
  - **HPM_AUTO_PARENT**
  - **HPM_ONLY_EXCLUSIVE**
  - **HPM_NO_PARENT**
  - The *inst_ID* of an active **hpmTStart**() or **hpmTStartx**() call

label
: Specifies the name associated with the instrumented code region. This name is used to identify the code region in the generated performance data.

Description

The **hpmTstartx**() function identifies the start of a region of code in which hardware performance counter events are to be counted, on a per-thread basis, and explicitly specifies the parent relationship for an encompassing region instrumented by **hpmTstart**() or **hpmTstartx**(). The end of the region is identified by a call to **hpmTstop**() using the same *inst_ID*.

The **hpmTstartx**() function assigns an identifier and a name to that region. When this function is executed, it records the starting value for the hardware

performance counters that are being used. When the corresponding **hpmTstop**() function call is executed, the hardware performance counters are read again and the difference between the current values and the starting values is accumulated.

Regions of code bounded by **hpmTstartx**() and **hpmTstop**() calls can be nested. When regions are nested, **hpmTstartx**() and **hpmTstop**() properly accumulate hardware events so they can be properly accounted for with both inclusive and exclusive reporting. For reporting of exclusive event counts, the proper parent relationship must be determined. If regions are perfectly nested, such as a set of nested loops, **hpmTstart**() is sufficient for determining parent relationships. In more complicated nesting cases, **hpmTstartx**() should be used to properly specify those relationships.

Parent relationships are specified by the *parent_ID* parameter which must have one of the following values:

HPM_AUTO_PARENT
: Automatically determine the parent for this **hpmTstartx**() call. This is done by searching for the immediately preceding **hpmTstart**() or **hpmTstartx**() call executed on the current thread in which there has not been a corresponding call made to **hpmStop**().

HPM_ONLY_EXCLUSIVE
: This operates in the same way as if **HPM_AUTO_PARENT** was specified, and also acts as if the **HPM_EXCLUSIVE_VALUES** environment variable was set for this call to **hpmTstartx**() only. If the **HPM_EXCLUSIVE_VALUES** environment variable was previously set, this parameter value is equivalent to specifying **HPM_AUTO_PARENT**.

HPM_NO_PARENT
: Specifies that this **hpmTstartx**() call has no parent.

inst_ID
: For a previous **hpmTsStart**() or **hpmTstartx**() call that is currently active, meaning the corresponding call to **hpmTstop**() has not been made for this instance of execution.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

See “hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment” on page 265.

Examples

```
#include <libhpc.h>
void thread_func();
int main(int argc, char *argv[])
{
    hpmInit(0, "HPMTest");
    thread_func(); /* assume this function runs on
                      multiple threads */
    hpmTerminate(0);
}
void thread_func()
{
    int i;
 int j;
    float x;
```

```
    x = 10.0;
    hpmTstartx(1, HPM_NO_PARENT, "Region 1");
    for (i = 0; i < 100000; i++) {
       hpmTstartx(2, 1, "Region 2");
   for (j = 0; j < 100000; j++) {
     x = x / 1.001;
       }
   hpmTstop(2);
       x = x / 1.001;
    }
    hpmTstop(1);
}

    program hpmtest
    #include "f_hpc.h"
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = thread_func() ! Assume thread_func runs on
                      ! multiple threads
    call f_hpmterminate(0)
    end

 real*4 function thread_func()
 #include "f_hpc.h"
 real*4 x
     integer i
     integer j
    x = 10.0;
    call f_hpmtstartx(1, HPM_NO_PARENT, 'Region 1')
    do 10 i = 1, 100000
      call f_hpmtstartx(2, 1, 'Region 2')
      do 20 j = 1, 100000
         x = x / 1.001
20  continue
      call f_hpmtstop(2)
      x = x / 1.001
10 continue
    call f_hpmtstop(1)
    thread_func = x
 return
    end
```

hpmTstop, f_hpmtstop - Identify the end point of an instrumented region of code

Use **hpmTstop** or **f_hpmtstop** to identify the end point of an instrumented region of code starting with a call to **hpmTstart()** or **hpmTstartx()** in which hardware performance counter events are to be counted. Also accumulates hardware performance counter events for that region.

Library

```
-lhpc
-lpmapi (AIX)
```

C syntax

```
#include <libhpc.h>
void hpmTstop(int inst_ID);
```

FORTRAN syntax

```
#include <f_hpc.h>
#include <f_hpc_i8.h>
subroutine f_hpmtstop(integer inst_ID)
```

Notes:

1. Use **f_hpc.h** for programs compiled without **–qintsize=8** and use **f_hpc_i8.h** for programs compiled with **–qintsize=8**.
2. FORTRAN programs that include either of these headers need to be preprocessed by the C preprocessor. This might require the use of the **xlf -qsuffix** option, for instance, **-qsuffix=cpp=f** or the gfortan option **-cpp**. Another option is to use the **.F** Fortran source file extension.

Parameters

inst_ID
: Specifies a unique value identifying the instrumented code region. This value must match the *inst_ID* specified in the corresponding **hpmTstart**() or **hpmTstartx**() function call.

Description

The **hpmTstop**() function identifies the end of a region of code in which hardware performance counter events are to be monitored. The start of the region is identified by a call to an **hpmTstart**() or **hpmTstartx**() function using the same *inst_ID*. The **hpmTstart**() or **hpmTstartx**() function must be called for a specific *inst_ID* before the corresponding call to the **hpmStop**() function.

The include files shown in the syntax are located in the **${IHPCT_BASE}/include** directory.

Environment variables

None.

Examples

```
#include <libhpc.h>
int main(int argc, char *argv[])
{
    int i;
```

```
    float x;
    x = 10.0;
    hpmInit(0, "HPMTest");
    hpmTstart(1, "Region 1");
    for (i = 0; i < 100000; i++) {
       x = x / 1.001;
    }
    hpmTstop(1);
    hpmTerminate(0);
}
```

```
    program hpmtest
    #include "f_hpc.h"
    integer i
    real*4 x
    call f_hpminit(0, 'HPMTest')
    x = 10.0;
    call f_hpmtstart(1, 'Region 1')
    do 10 i = 1, 100000
      x = x / 1.001
10 continue
    call f_hpmtstop(1)
    call f_hpmterminate(0)
    end
```

MT_get_allresults - Obtain statistical results

Use **MT_get_allresults** to obtain statistical results from performance data for an MPI function.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_get_allresults(int data_type, int mpi_id, struct MT_summarystruct *data);
```

Parameters

data_type
Specifies the type of data to be returned in the data parameter.

mpi_id An enumeration specifying the MPI function for which data is obtained.

data A structure, allocated by the user, containing the statistical data returned by calling this function.

Description

This function computes statistical data from performance data accumulated for an MPI function type or for all MPI functions in an application.

The *data_type* parameter specifies the statistical measurement that is returned by a call to this function as follows:

COUNTS
The number of times the specified MPI function was called.

BYTES
The total number of bytes of data transferred in calls to the specified MPI function.

COMMUNICATIONTIME
The total time spent in all calls to the specified MPI function.

STACK
The maximum stack address for any call to the specified MPI function.

HEAP
The maximum heap address for any call to the specified MPI function.

ELAPSEDTIME
Either the elapsed time between calls to **MPI_Init()** and **MPI_Finalize()**, or, if that value is zero, the elapsed time since **MPI_Init()** was called.

The *mpi_id* parameter specifies the MPI function for which statistics are to be computed. It must be either an enumeration from the table shown in the **MT_trace_event**() function man page, also found in the **$IHPCT_BASE/include/mpi_trace_ids.h** header file, or **ALLMPI_ID** to compute statistics for all MPI functions profiled by the MPI trace library in the application.

If the *mpi_id* parameter is specified as **ALLMPI_ID**, meaningful results are returned only when the *data_type* parameter is specified as **BYTES** or **COMMUNICATIONTIME**. In all other cases, the returned data is zero.

This function fills in the **MT_summarystruct** structure allocated by the user, in which the following fields are relevant:

int min_rank
The MPI task rank of the task corresponding to the value in the **min_result** field.

int max_rank
The MPI task rank of the task corresponding to the value in the **max_result** field.

int med_rank
The MPI task rank of the task corresponding to the value in the **med_result** field.

void *min_result
The minimum value from all tasks for the measurement specified by the *data_type* parameter.

void *max_result
The maximum value from all tasks for the measurement specified by the *data_type* parameter.

void *med_result
The median value from all tasks for the measurement specified by the *data_type* parameter.

void *avg_result
The average value from all tasks for the measurement specified by the *data_type* parameter.

void *sum_result
The sum of the measurements from all tasks for the measurement specified by the *data_type* parameter.

void *all_result
An array of measurements for all tasks, in MPI task rank order, for the measurement specified by the *data_type* parameter.

void *sorted_all_result
An array of measurements for all tasks, sorted in data value order, for the measurement specified by the *data_type* parameter.

int *sorted_rank
An array of MPI task ranks corresponding to the data values in the **sorted_all_result array**.

The datatype of the **min_result**, **max_result**, **med_result**, **avg_result** and **sum_result** fields depends on the value specified for the *data_type* parameter as follows:

COUNTS
long long

BYTES
double

COMMUNICATIONTIME
double

STACK
double

HEAP double

ELASPSEDTIME
double

You must cast the fields to the appropriate data type in your code.

The **all_result** and **sorted_all** result arrays are arrays of the same data type as the individual fields just described. You are responsible for freeing these arrays after they are no longer needed.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns **1** for successful completion. It returns **-1** if an error occurs.

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <mpi.h>
#include <mpi_trace_ids.h>
#include <stdio.h>
int MT_output_text()
{
  struct MT_summarystruct stats;
  MT_get_allresults(BYTES, SEND_ID, &stats);
  printf("Minimum bytes sent (%11.6f) by task %d\n",
            (double) stats.min_result, stats.min_rank);
  printf("Maximum bytes sent (%11.6f) by task %d\n",
            (double) stats.max_result, stats.max_rank);
  return 0;
}
```

MT_get_calleraddress - Obtain the address of the caller of an MPI function

Use **MT_get_calleraddress** to obtain the address of the caller of an MPI function.

Library

```
-lmpitrace
```

C syntax

```
void *MT_get_calleraddress(int level);
```

Parameters

level Specifies the number of levels to walk up the call stack to get the caller's address.

Description

This function can be used within your implementation of **MT_trace_event**() to obtain the address of the caller of an MPI function. If this function is called inside your implementation of **MT_trace_event**() and the level parameter is specified as zero, it obtains the address where the MPI function was called. If the level parameter is specified as **1**, this function returns the address where the function that called the current MPI function was called.

Returns

This function returns the caller's address as determined by the level parameter.

Environment variables

None.

Example

```
#include <mpt.h>
int MT_trace_event(int id)
{
    unsigned long caller_addr;
    caller_addr = MT_get_calleraddress(3);
    return 1;
}
```

MT_get_callerinfo - Obtain source code information

Use **MT_get_callerinfo** to obtain source code information about the caller of an MPI function.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
int MT_get_callerinfo(unsigned long addr, struct MT_callerstruct src_info);
```

Parameters

addr Specifies the address for which source code information is to be obtained.

src_info
Contains the source code information returned by this function for the specified caller address.

Description

This function can be used to obtain the source code information, including the file name and line number corresponding to the address specified by the ***addr*** parameter.

This function fills in the **MT_callerstruct** structure passed to it with the following information:

char **filepath*
The path name of the directory containing the source file.

char **filename*
The file name of the source file.

char **funcname*
The name of the function containing the caller address.

int *lineno*
The source line number corresponding to the address passed in the *addr* parameter.

In order for this function to work correctly, the application should be compiled and linked with the **–g** compiler option so that the required file name and line number information is contained in the executable.

Returns

This function returns zero if the source file information was obtained. It returns **-1** if the source file information could not be obtained.

Environment variables

None.

Example

```
#include <mpt.h>
#include <stdio.h>
int MT_trace_event(int id)
{
    struct MT_callerstruct src_info;
    int status;
    unsigned long caller_addr;
    caller_addr = MT_get_calleraddress(3);
    status = MT_get_callerinfo(caller_addr, &src_info);
    if (status == 0) {
      printf("%s was called from %s/%s(%s) line %d\n",
                MT_get_mpi_name(id), src_info.filepath,
                src_info.filename, src_info.funcname,
                src_info.lineno);
    }
    return 1;
}
```

MT_get_elapsed_time - Obtains elapsed time

Use **MT_get_elapsed_time** to get the elapsed time in seconds between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
double MT_get_elapsed_time();
```

Parameters

None.

Description

This function returns the elapsed time, in seconds, between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the time, in seconds, between a call to **MPI_Init()** and a call to **MPI_Finalize()**.

Environment variables

None.

Example

```
#include <stdio.h>
#include <mpt.h>
int MT_output_text()
{
    printf("Time between MPI_Init and MPI_Finalize is %11.6f seconds\n",
                MT_get_elapsed_time());
    return 0;
}
```

MT_get_environment - Returns run-time environment information

Use **MT_get_environment** to return information about the run-time environment for the application.

Library

-lmpitrace

C syntax

```
#include <mpt.h>;
void MT_get_environment(struct MT_envstruct *env);
```

Parameters

env A pointer to a structure, allocated by the user, which contains information about the application runtime environment.

Description

This function is used to obtain information about the application runtime environment by filling in the MT_envstruct structure allocated by the user. The following fields in the **MT_envstruct** structure are relevant:

int *mpirank*
The MPI task rank for this task in the application.

int *ntasks*
The number of tasks in the MPI application.

int *nmpi*
The maximum index allowed for *mpi_id* when calling **MT_get_mpi_counts()**, **MT_get_mpi_bytes()**, and **MT_get_mpi_time()**.

Environment variables

None.

Example

```
#include <mpt.h>
#include <stdio.h>
int MT_output_text()
{
    MT_envstruct env;
    MT_get_environment(&env);
    printf("MPI task rank is %d\n", env.mpirank);
    return 0;
}
```

MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred

Use **MT_get_mpi_bytes** to obtain the accumulated number of bytes transferred by a specific MPI function.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
double MT_get_mpi_bytes(int mpi_ID);
```

Parameters

mpi_ID
An enumeration identifying the MPI function.

Description

The **MT_get_mpi_bytes**() function returns the accumulated number of bytes transferred by this task for all MPI function calls corresponding to the enumeration specified as *mpi_ID*. The *mpi_ID* parameter must be any of the values as specified in Table 54 on page 301 or as specified in the **$IHPCT_BASE/include/mpi_trace_ids.h** header file. However, meaningful results are returned only for MPI functions that either send or receive data.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the accumulated number of bytes transferred by this task for the MPI function specified by the *mpi_ID* parameter.

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("Total bytes sent using MPI_Send: %f11.6 bytes\n",
               MT_get_mpi_bytes(SEND_ID));
    return 0;
}
```

MT_get_mpi_counts - Obtain the the number of times a function was called

Use **MT_get_mpi_counts** to obtain the number of times the specified MPI function was called.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
long long MT_get_mpi_counts(int mpi_ID);
```

Parameters

mpi_ID
An enumeration identifying the MPI function.

Description

The **MT_get_mpi_counts**() function returns the number of times the specified MPI function was called in this task. The ***mpi_ID*** parameter must be any of the values as specified in Table 54 on page 301 or as specified in the **$IHPCT_BASE/include/mpi_trace_ids.h** header file.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the number of times the MPI function, specified by the ***mpi_ID*** parameter, was called in this task.

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("MPI_Send called %lld times\n",
               MT_get_mpi_counts(SEND_ID));
    return 0;
}
```

MT_get_mpi_name - Returns the name of the specified MPI function

Use **MT_get_mpi_name** to return the name of the specified MPI function.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
char *MT_get_mpi_name(int mpi_ID);
```

Parameters

mpi_ID
An enumeration identifying the MPI function.

Description

The **MT_get_mpi_name**() function returns the name of the specified MPI function. The *mpi_ID* parameter must be any of the values as specified in Table 54 on page 301 or as specified in the **$IHPCT_BASE/include/mpi_trace_ids.h** header file.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the name of the MPI function specified by the *mpi_ID* parameter.

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("%s called %lld times\n",
               MT_get_mpi_name(SEND_ID),
               MT_get_mpi_counts(SEND_ID));
    return 0;
}
```

MT_get_mpi_time - Obtain elapsed time

Use **MT_get_mpi_time** to obtain the elapsed time, in seconds, spent in the specified MPI function.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
double MT_get_mpi_time(int mpi_ID);
```

Parameters

mpi_ID
: An enumeration identifying the MPI function.

Description

The **MT_get_mpi_time**() function returns the elapsed time spent in the specified MPI function in this task. The *mpi_ID* parameter must be any of the values as specified in Table 54 on page 301 or as specified in the **$IHPCT_BASE/include/mpi_trace_ids.h** header file.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the elapsed time spent, in seconds, in this task in the MPI function specified by the *mpi_ID* parameter.

Environment variables

IHPCT_BASE
: Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

Example

```
#include <stdio.h>
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_output_text()
{
    printf("MPI_Send spent %f11.6 seconds elapsed time\n",
               MT_get_mpi_time(SEND_ID));
    return 0;
}
```

MT_get_time - Get the elapsed time

Use **MT_get_time** to get the elapsed time, in seconds, since **MPI_Init()** was called.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
double MT_get_time();
```

Parameters

None.

Description

This function returns the time, in seconds, since **MPI_Init()** was called.

This function can be useful when you implement your own version of **MT_output_text**().

Returns

This function returns the time, in seconds, since **MPI_Init()** was called.

Environment variables

None.

Example

```
#include <stdio.h>
#include <mpt.h>
int MT_output_text()
{
    printf("MPI_Init was called %f11.6 seconds ago.\n",
               MT_get_time());
    return 0;
}
```

MT_get_tracebufferinfo - Obtain information about MPI trace buffer usage

Use **MT_get_tracebufferinfo** to obtain information about MPI trace buffer usage by the MPI trace library.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
int MT_get_tracebufferinfo(struct MT_tracebufferstruct *info);
```

Parameters

info A pointer to an **MT_tracebufferstruct** structure allocated by the user, which also contains the returned results from this function.

Description

This function obtains information about the internal MPI trace buffer used by the MPI trace library. This function fills in an **MT_tracebufferstruct** allocated by the user, where the following fields in this structure are relevant.

int number_events
The number of MPI trace events currently recorded in this buffer.

double total_buffer
The MPI trace buffer size, in megabytes.

double used_buffer
The amount of trace buffer used, in megabytes.

double free_buffer
The remaining free space in buffer, in megabytes.

Returns

This function returns **0** on successful completion and returns a nonzero value on failure.

Environment variables

None.

Example

```
#include <mpt.h>
#include <stdio.h>
int MT_output_text()
{
    MT_tracebufferstruct info;
    MT_get_tracebufferinfo(&info);
    printf("%d MPI events were recorded\n",
           info.number_events);
    printf("%11.6fMB of %11.6fMB trace buffer used\n",
           info.used_buffer, info.total_buffer);
    return 0;
}
```

MT_output_text - Generate performance statistics

Use **MT_output_text** to generate the performance statistics for your application when your application calls **MPI_Finalize()**.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
int MT_output_text();
```

Parameters

None.

Description

This function generates performance statistics for your application. The MPI trace library calls this function when **MPI_Finalize()** is called in your application. You can override the default behavior of this function, generating a summary of MPI performance statistics, by implementing your own version of **MT_output_text**() and linking it with your application.

Returns

This function returns **-1** if an error occurs. Otherwise this function returns **1**.

Environment variables

None.

Example

```
#include <mpt.h>
#include <mpi_trace_ids.h>
#include <stdio.h>
int MT_output_text()
{
    struct MT_summarystruct send_info;
    struct MT_summarystruct recv_info;
    MT_get_allresults(ELAPSEDTIME, SEND_ID, &send_info);
    MT_get_allresults(ELAPSEDTIME, RECV_ID, &recv_info);
    printf("MPI_Send task with min. elapsed time: %d\n",
              send_info.min_rank);
    printf("MPI_Send task with max. elapsed time: %d\n",
              send_info.max_rank);
    printf("MPI_Recv task with min. elapsed time: %d\n",
              recv_info.min_rank);
    printf("MPI_Recv task with max. elapsed time: %d\n",
              recv_info.max_rank);
    return 0;
}
```

MT_output_trace - Control whether an MPI trace file is created

Use **MT_output_trace** to control whether an MPI trace file is created for a specific task.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
int MT_output_trace(int rank);
```

Parameters

rank The MPI task rank of this task.

Description

This function controls whether an MPI trace file is generated for a specific MPI task. You can override the default MPI trace library behavior of generating a trace file for all tasks by implementing your own version of this function and linking it with your application. The MPI trace library calls your implementation of this function as part of its processing when your application calls **MPI_Finalize()**.

Returns

0 If an MPI trace file is not to be generated for the MPI task from which this function is called.

1 If an MPI trace file is to be generated for the MPI task from which this function is called.

Environment variables

None.

Example

```
#include <mpt.h>
int MT_output_trace(int rank)
{
    /* Generate trace files for even rank tasks only */
  if ((rank % 2) == 0) {
    return 1;
  }
  else {
    return 0;
  }
  }
```

MT_trace_event - Control whether an MPI trace event is generated

Use **MT_trace_event** to control whether an MPI trace event is generated for a specific MPI function call

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_trace_event(int mpi_ID)
```

Parameters

mpi_ID
An enumeration identifying the MPI function that is about to be traced.

Description

The **MT_trace_event**() function controls whether the MPI trace library should generate a trace event for an MPI function call. The default behavior of the MPI trace library is to generate trace events for all MPI function calls defined in the **mpi_trace_ids.h** header. You override the default MPI trace library behavior by implementing your own version of this function and linking it with your application. The MPI trace library calls your implementation of this function each time the MPI trace library is about to generate a trace event. You can control the collection of MPI trace events for any function with an identifier as listed in Table 54:

Table 54. Controlling the collection of MPI trace events for any function with an identifier

Function	Function	Function
COMM_SIZE_ID	COMM_RANK_ID	SEND_ID
SSEND_ID	RSEND_ID	BSEND_ID
ISEND_ID	ISSEND_ID	IRSEND_ID
IBSEND_ID	SEND_INIT_ID	SSEND_INIT_ID
RSEND_INIT_ID	BSEND_INIT_ID	RECV_INIT_ID
RECV_ID	IRECV_ID	SENDRECV_ID
SENDRECV_REPLACE_ID	BUFFER_ATTACH_ID	BUFFER_ATTACH_ID
PROBE_ID	IPROBE_ID	TEST_ID
TESTANY_ID	TESTALL_ID	TESTSOME_ID
WAIT_ID	WAITANY_ID	WAITALL_ID
WAITSOME_ID	START_ID	STARTALL_ID
BCAST_ID	BARRIER_ID	GATHER_ID
GATHERV_ID	SCATTER_ID	SCATTERV_ID
SCAN_ID	ALLGATHER_ID	ALLGATHERV_ID
REDUCE_ID	ALLREDUCE_ID	REDUCE_SCATTER_ID
ALLTOALL_ID	ALLTOALLV_ID	

Returns

1 If a trace event should be generated for the MPI function call.

0 If no trace event should be generated.

Environment variables

See "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303.

Example

```
#include <mpt.h>
#include <mpi_trace_ids.h>
int MT_trace_event(int id)
{
    /* Trace only MPI_Send and MPI_Recv calls */
    if ((id == RECV_ID) || (id == SEND_ID)) {
      return 1;
    }
    else {
       return 0;
    }
}
```

MT_trace_start, mt_trace_start - Start or resume the collection of trace events

Use **MT_trace_start** or **mt_trace_start** to start or resume the collection of trace events for all MPI calls.

Library

```
-lmpitrace
```

C syntax

```
#include <mpt.h>
void MT_trace_start()
```

FORTRAN syntax

```
subroutine mt_trace_start()
```

Parameters

None.

Description

MT_trace_start() is used to start the collection of MPI trace events or to resume the collection of MPI trace events, if collection of these events has been suspended by a call to **MT_trace_stop**(). The environment variable **TRACE_ALL_EVENTS** must be set to **no** for **MT_trace_start**() to have any effect.

Environment variables

IHPCT_BASE
Specifies the path name of the directory in which the IBM HPC Toolkit is installed (**/opt/ibmhpc/ppedev.hpct**).

MAX_TRACE_EVENTS
Specifies the maximum number of trace events that can be collected per task. The default is **30000**.

MAX_TRACE_RANK
Specifies the MPI task rank of the highest rank process that has MPI trace events collected. The default is **256**.

MT_BASIC_TRACE
Specifies whether the **MAX_TRACE_RANK** environment variable is checked. If **MT_BASIC_TRACE** is set to **yes**, then **MAX_TRACE_RANK** is ignored and the trace is generated with less overhead. If **MT_BASIC_TRACE** is not set, then the setting of **MAX_TRACE_RANK** is honored.

OUTPUT_ALL_RANKS
Set to **yes** to generate trace files for all MPI tasks. The default is to generate trace files only for task **0** and the tasks that have the minimum, maximum, and median total MPI communication time. If task **0** is the task with minimum, maximum, or median communication time, only three trace files are generated by default.

TRACE_ALL_EVENTS
Set to **yes** to generate a trace containing trace events for all MPI calls after

MPI_Init(). If this environment variable is set to **no**, collection of MPI trace events is controlled by **MT_trace_start()** and **MT_trace_stop()**. The default is **yes**.

TRACE_ALL_TASKS
Set to **yes** to generate MPI trace files for all MPI tasks in the application. The default is **no**, which results in generating trace files only for MPI tasks **0** through **255**.

TRACEBACK_LEVEL
Specifies the number of levels to walk back in the function call stack when recording the address of an MPI call. This can be used to record profiling information for the caller of an MPI function rather than the MPI function itself, which might be useful if the MPI functions are contained in a library. The default is **0**.

Example

```
#include <mpt.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MT_trace_start();
      /* MPI communication region of interest */
    MT_trace_stop();
      /* MPI communication region of no interest */
    MPI_Finalize();
}
```

```
program main
include 'mpif.h'
call mpi_init()
call mt_trace_start()
! MPI communication region of interest
call mt_trace_stop()
! MPI communication region of no interest
call mpi_finalize()
end
```

MT_trace_stop, mt_trace_stop - Suspend the collection of trace events

Use **MT_trace_stop** or **mt_trace_stop** to suspend the collection of trace events for all MPI calls.

Library

-lmpitrace

C syntax

```
#include <mpt.h>
void MT_trace_stop()
```

FORTRAN syntax

```
subroutine mt_trace_stop()
```

Parameters

None.

Description

MT_trace_stop() is used to suspend the collection of MPI trace. The environment variable **TRACE_ALL_EVENTS** must be set to **no** for **MT_trace_stop**() to have any effect. **MT_trace_start**() might be called after a call to **MT_trace_stop**() to resume collection of MPI trace events.

Environment variables

See "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303.

Example

```
#include <mpt.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MT_trace_start();
      /* MPI communication region of interest */
    MT_trace_stop();
      /* MPI communication region of no interest */
    MPI_Finalize();
}
```

```
program main
include 'mpif.h'
call mpi_init()
call mt_trace_start()
! MPI communication region of interest
call mt_trace_stop()
! MPI communication region of no interest
call mpi_finalize()
end
```

Part 10. Appendixes

© Copyright IBM Corp. 2008, 2013

Appendix A. Performance data file naming

The names of the performance data files generated by the HPC Toolkit are dependent on the setting of the **HPC_OUTPUT_NAME** environment variable, the **HPC_UNIQUE_FILE_NAME** environment variable, whether the application is a serial program or a parallel MPI program, and whether MPI dynamic tasking is used.

File naming conventions

Table 55 shows the file naming conventions for various tools and their types. The value of the **HPC_OUTPUT_NAME** environment variable specifies the *base* performance data file name, or if the value contains one or more file path separators, it specifies both the *dir* and *base* components of the performance data file name. If **HPC_OUTPUT_NAME** is not set, then there is no *dir*, and the value of *base* is **hpct**. The first character of the value of the **HPC_UNIQUE_FILE_NAME** environment variable specifies **Y**, **y**, or **1** to enable unique file naming, otherwise it is disabled.

Table 55. File naming conventions for various tools and their types

Tool	File naming	
HPM	**Type**	**Name**
	ascii	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].hpm[.*subtype*].txt
	viz	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].hpm[.*subtype*].viz
MPI	**Type**	**Name**
	ascii	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mpi.txt
	viz	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mpi.viz
	trace	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mpi.mpt
OpenMP	**Type**	**Name**
	ascii	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].omp.txt
	viz	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].omp.viz
MIO	**Type**	**Name**
	stats	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mio[.subtype].txt
	viz	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mio[.subtype].viz
	event	[*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].mio[.subtype].iot

© Copyright IBM Corporation © IBM 2008, 2013

Table 55. File naming conventions for various tools and their types (continued)

Tool	File naming

Profiling	**Type** **Name**
	gmon [*dir*[*_processId*]/]*base*[*_processId*][*_worldId_rankId*].gmon.out
	This **gmon** output file naming is only for the IBM Parallel Environment (PE) Runtime Edition execution environment running on the AIX operating system with applications compiled with **-pg** and **-hpcprof** flags, such as:
	`mpicc -pg -hpcprof -o applicationName applicationName.c`
	and run under the **hpcrun** command, such as:
	`hpcrun applicationName`
	On *all other* execution environments for applications compiled without **-hpcprof** and not run under the **hpcrun** command, the **gmon.out** file name follows the conventions of that environment. For example, a serial AIX or Linux execution environment outputs a **gmon.out** file (unless the **GPROF** environment variable defines a different prefix than **gmon**), and for a IBM Parallel Environment (PE) Runtime Edition execution environment see the topic, "Profiling programs with the prof and gprof commands" in *Parallel Environment Runtime Edition: Operation and Use* for the **gmon** output file naming convention.

where:

dir
: Is the optional directory path portion of the specified file name from the **HPC_OUTPUT_NAME** environment variable. The *dir* is appended with the process ID, *_processId*, only if the **HPC_UNIQUE_FILE_NAME** environment variable is set.

base
: Is the base name portion of the specified file name from the **HPC_OUTPUT_NAME** environment variable. The base is appended with the process ID, *_processId*, only if the **HPC_UNIQUE_FILE_NAME** environment variable is set and if a directory name is not present. The base is further appended with the world ID and rank ID, *_worldId_rankId*, only for a MPI parallel execution. In other words, the world ID and rank ID are not appended for a serial execution.

processId
: Is the optional process ID. It is the **getpid()** for a serial execution or the poe process ID for an MPI parallel execution.

worldId
: Is the optional world ID. For MPI without dynamic tasking it is zero (**0**). For MPI with dynamic tasking it is the MPI world ID.

rankId
: Is the optional rank ID. For MPI without dynamic tasking it is the rank ID. For MPI with dynamic tasking it is the rank ID within the world.

tool
: Is the tool name. The various tools are: **hpm**, **mpi**, **omp**, **mio** and **gmon**.

subtype
: Is the optional subtype. The subtype is appended only if one is provided or needed.

  - For the HPM tool, the subtype depends on whether the **hpccount** command, the **hpcstat** command, **libhpc** library, or a HPM instrumented binary is used:

- If the **hpccount** command is used, then the invoked executable's base name is the subtype.
- If the **hpcstat** command is used, then **hpcstat** is the subtype.
- If **libhpc** is used, then the *progName* argument of **hpmInit** is the subtype, unless the *progName* argument is NULL then the subtype is not defined.
- If the HPM instrumented binary is used, then the binary name is the subtype.

• For the MIO tool, the subtype depends on the **MIO_FILES** and **MIO_DEFAULTS** environment variables. The subtype is defined only if the **MIO_FILES** or **MIO_DEFAULTS** environment variables specify an entry for the trace module with the setting for its **stats**={*subtype*}, **xml**={*subtype*} or **events**={*subtype*} keywords respectively.

type
Is the type of file, which is also known as the file extension. The various types are: **txt**, **viz**, **mpt**, **iot**, and **out**.

File naming examples

The file naming conventions are shown in the following examples. For MPI applications, these examples assume **MP_PROCS** is set to 5.

• For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to ***sample***, and **HPC_UNIQUE_FILE_NAME** not set:
- A hardware counter ASCII data output file is named:
  ***sample*.hpm.*app*.txt**
- A hardware counter visualization data output file is named:
  ***sample*.hpm.*app*.viz**

• For a serial program that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to ***output*/*sample***, and **HPC_UNIQUE_FILE_NAME** not set:
- A hardware counter ASCII data output file is named:
  ***output*/*sample*.hpm.*app*.txt**
- A hardware counter visualization data output file is named:
  ***output*/*sample*.hpm.*app*.viz**

• For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to ***sample***, **HPC_UNIQUE_FILE_NAME** set to **y**, and the process ID is 92878:
- A hardware counter ASCII data output file is named:
  ***sample*_92878.hpm.*app*.txt**
- A hardware counter visualization data output file is named:
  ***sample*_92878.hpm.*app*.viz**

• For a serial program, named *app*, that is compiled, instrumented for HPM analysis, and executed with **HPC_OUTPUT_NAME** set to ***output/sample***, **HPC_UNIQUE_FILE_NAME** set to **y**, and the process ID is 185903:
- A hardware counter ASCII data output file is named:
  ***output*_185903/*sample*.hpm.*app*.txt**
- A hardware counter visualization data output file is named:
  ***output*_185903/*sample*.hpm.*app*.viz**

- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed without using dynamic tasking and with **HPC_OUTPUT_NAME** set to ***sample*** and **HPC_UNIQUE_FILE_NAME** not set:
  - Hardware counter ASCII data output files are named:

    ***sample*_0_0.hpm.*app*.txt**
    ***sample*_0_1.hpm.*app*.txt**
    ***sample*_0_2.hpm.*app*.txt**
    ***sample*_0_3.hpm.*app*.txt**
    ***sample*_0_4.hpm.*app*.txt**
  - Hardware counter visualization data output files are named:

    ***sample*_0_0.hpm.*app*.viz**
    ***sample*_0_1.hpm.*app*.viz**
    ***sample*_0_2.hpm.*app*.viz**
    ***sample*_0_3.hpm.*app*.viz**
    ***sample*_0_4.hpm.*app*.viz**
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed without using dynamic tasking and with **HPC_OUTPUT_NAME** set to ***sample***, **HPC_UNIQUE_FILE_NAME** set to **y**, and the process ID is 14150:
  - Hardware counter ASCII data output files are named:

    ***sample*_14150_0_0.hpm.*app*.txt**
    ***sample*_14150_0_1.hpm.*app*.txt**
    ***sample*_14150_0_2.hpm.*app*.txt**
    ***sample*_14150_0_3.hpm.*app*.txt**
    ***sample*_14150_0_4.hpm.*app*.txt**
  - Hardware counter visualization data output files are named:

    ***sample*_14150_0_0.hpm.*app*.viz**
    ***sample*_14150_0_1.hpm.*app*.viz**
    ***sample*_14150_0_2.hpm.*app*.viz**
    ***sample*_14150_0_3.hpm.*app*.viz**
    ***sample*_14150_0_4.hpm.*app*.viz**
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed with using dynamic tasking and with **HPC_OUTPUT_NAME** set to ***sample***, **HPC_UNIQUE_FILE_NAME** not set, and the world ID is 7:
  - Hardware counter ASCII data output files are named:

    ***sample*_7_0.hpm.*app*.txt**
    ***sample*_7_1.hpm.*app*.txt**
    ***sample*_7_2.hpm.*app*.txt**
    ***sample*_7_3.hpm.*app*.txt**
    ***sample*_7_4.hpm.*app*.txt**
  - Hardware counter visualization data output files are named:

    ***sample*_7_0.hpm.*app*.viz**
    ***sample*_7_1.hpm.*app*.viz**
    ***sample*_7_2.hpm.*app*.viz**
    ***sample*_7_3.hpm.*app*.viz**
    ***sample*_7_4.hpm.*app*.viz**
- For an MPI application, named *app*, that is compiled, instrumented for HPM analysis, and executed with using dynamic tasking and with **HPC_OUTPUT_NAME** set to ***sample***, **HPC_UNIQUE_FILE_NAME** set to **y**, the world ID is 7, and the process ID is 78727:
  - Hardware counter ASCII data output files are named:

    ***sample*_78727_7_0.hpm.*app*.txt**
    ***sample*_78727_7_1.hpm.*app*.txt**
    ***sample*_78727_7_2.hpm.*app*.txt**
    ***sample*_78727_7_3.hpm.*app*.txt**
    ***sample*_78727_7_4.hpm.*app*.txt**

– Hardware counter visualization data output files are named:

_sample_78727_7_0.hpm._app_.viz
_sample_78727_7_1.hpm._app_.viz
_sample_78727_7_2.hpm._app_.viz
_sample_78727_7_3.hpm._app_.viz
_sample_78727_7_4.hpm._app_.viz

- For an MPI application that is compiled, linked with the MPI tracing library to trace all MPI calls, and executed with **HPC_OUTPUT_NAME** and **HPC_UNIQUE_FILE_NAME** not set, the following output file names are generated. In this case, only output from ranks **0**, **1**, and **4** was generated: one for task **0**, which also has the median MPI communications time, one for the task with the minimum MPI communications time, and one for the ask with the maximum MP communications.
  – MPI ASCII data output files named:

  hpct_0_0.mpi.txt
  hpct_0_1.mpi.txt
  hpct_0_4.mpi.txt

  – MPI visualization data output files named:

  hpct_0_0.mpi.viz
  hpct_0_1.mpi.viz
  hpct_0_4.mpi.viz

  – A single MPI trace data output file named:

  hpct_0_0.mpi.mpt

  Note: For the same MPI application if **HPC_UNIQUE_FILE_NAME** is set to **y**, and the process ID is 31052, then each output file name that is generated contains the process ID (*processId*).
  – MPI ASCII data output files named:

  hpct_31052_0_0.mpi.txt
  hpct_31052_0_1.mpi.txt
  hpct_31052_0_4.mpi.txt

  – MPI visualization data output files named:

  hpct_31052_0_0.mpi.viz
  hpct_31052_0_1.mpi.viz
  hpct_31052_0_4.mpi.viz

  – A single MPI trace data output file named:

  hpct_31052_0_0.mpi.mpt

Appendix B. Derived metrics descriptions

The following topics provide information about the:

- "Derived metrics description for AIX and Linux on Power6"
- "Derived metrics description for AIX and Linux on POWER7 and POWER7+" on page 316
- "Derived metrics description for x86 Nehalem and Westmere" on page 317
- "Derived metrics description for x86 Sandy Bridge" on page 318
- "Derived metrics description for x86 Ivy Bridge" on page 322

Derived metrics description for AIX and Linux on Power6

The derived metrics implemented by **hpccount**, **hpcstat**, and **libhpc** are as follows:

Utilization rate
100.0 * user_time / wall_clock_time

Total FP load and store operations
fp_tot_ls = (PM_LSU_LDF + PM_FPU_STF) * 0.000001

MIPS PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
(double)PM_INST_CMPL / PM_CYC

Instructions per run cycle
(double)PM_RUN_INST_CMPL / PM_RUN_CYC

Instructions per load or store
(double)PM_INST_CMPL/(PM_LD_REF_L1 + PM_ST_REF_L1)

Percentage Instructions dispatched that completed
100.0 * PM_INST_CMPL / PM_INST_DISP

Number of loads per load miss
(double)PM_LD_REF_L1 / PM_LD_MISS_L1

Number of stores per store miss
(double)PM_ST_REF_L1 / PM_ST_MISS_L1

Number of load or stores per L1 miss
((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
/ ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1)

L1 cache hit rate
100.0 * (1.0 - ((double)PM_LD_REF_L1 + (double)PM_ST_REF_L1)
/ ((double)PM_ST_MISS_L1 + (double)PM_LD_MISS_L1))

Total Loads from local L2
tot_ld_l_L2 = (double)PM_DATA_FROM_L2 / (1024*1024)

Local L2 load traffic
L1_cache_line_size * tot_ld_l_L2

Local L2 load bandwidth per processor
L1_cache_line_size * tot_ld_l_L2 / wall_clock_time

© Copyright IBM Corp. 2008, 2013

Total Loads from local L3
tot_ld_l_L3 = (double)PM_DATA_FROM_L3 / (1024*1024)

Local L3 load traffic
L2_cache_line_size * tot_ld_l_L3

Local L3 load bandwidth per processor
L2_cache_line_size * tot_ld_l_L3 / wall_clock_time

Percentage loads from local L3 per cycle
100.0 * (double)PM_DATA_FROM_L3 / PM_CYC

Total Loads from local memory
tot_ld_lmem = (double)PM_DATA_FROM_LMEM / (1024*1024)

Local memory load traffic
L3_cache_line_size * tot_ld_lmem

Local memory load bandwidth per processor
L3_cache_line_size * tot_ld_lmem / wall_clock_time

HW floating point instructions (flips)
flips = ((double) PM_FPU_FIN) * 0.000001

HW floating point instructions per cycle
flips / PM_CYC

HW floating point instructions per run cycle
flips / PM_RUN_CYC

HW floating point instr. rate (HW flips per WCT)
flips * 0.000001 / wall_clock_time

HW floating point instructions per user time
flips * 0.000001 / user_time

Total scalar floating point operations
flops =((double)(PM_FPU_FLOP + PM_FPU_FMA)) *0.000001

Scalar flop rate (flops per WCT)
scalar flops * 0.000001/ wall_clock_time

Scalar flops per user time
scalar flops * 0.000001/ user_time

Algebraic floating point operations
aflops = ((double)(PM_FPU_1FLOP + 2.0 * PM_FPU_FMA)) * 0.000001

Algebraic flop rate (flops per WCT)
aflops * 0.000001 / wall_clock_time

Algebraic flops per user time
aflops * 0.000001 / user_time

Scalar FMA percentage
100.0 * 2 * PM_FPU_FMA / scalar flops

Scalar percentage of peak performance
100.0 * scalar flops * cycle_time/ (4 * user_time)

Derived metrics description for AIX and Linux on POWER7 and POWER7+

The derived metrics description for POWER7 and POWER7+ are as follows:

Utilization rate
100.0 * user_time / wall_clock_time

MIPS PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
(double)PM_INST_CMPL / PM_CYC

Instructions per run cycle
(double)PM_RUN_INST_CMPL / PM_RUN_CYC

Percentage Instructions dispatched that completed
100.0 * PM_INST_CMPL / PM_INST_DISP

Branches mispredicted percentage
100.0 * (double)PM_BR_MPRED / PM_BRU_FIN

Number of loads per load miss
(double)PM_LD_REF_L1 / PM_LD_MISS_L1

Total Loads from local L2
tot_ld_l_L2 = (double)PM_DATA_FROM_L2 / (1024*1024)

Local L2 load traffic
L1_cache_line_size * tot_ld_l_L2

Local L2 load bandwidth per processor
L1_cache_line_size * tot_ld_l_L2 / wall_clock_time

Percentage loads from local L2 per cycle
100.0 *(double)PM_DATA_FROM_L2 / PM_RUN_CYC

Total Loads from local L3
tot_ld_l_L3 = (double)PM_DATA_FROM_L3 / (1024*1024)

Local L3 load traffic
L2_cache_line_size * tot_ld_l_L3

Local L3 load bandwidth per processor
L2_cache_line_size * tot_ld_l_L3 / wall_clock_time

Percentage loads from local L3 per cycle
100.0 *(double)PM_DATA_FROM_L3 / PM_RUN_CYC

Total Loads from local memory
tot_ld_lmem = (double)PM_DATA_FROM_LMEM / (1024*1024)

Local memory load traffic
L3_cache_line_size * tot_ld_lmem

Local memory load bandwidth per processor
L3_cache_line_size * tot_ld_lmem / wall_clock_time

Derived metrics description for x86 Nehalem and Westmere

The derived metrics description for x86 Nehalem and Westmere are as follows:

MIPS PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
(double)PM_INST_CMPL / PM_CYC

Branches mispredicted percentage
100.0 * (double)PM_BR_MPRED / PM_BRU_FIN

L2 load traffic
L1_cache_line_size * PM_L2_LD

L2 load bandwidth per processor
L1_cache_line_size * PM_L2_LD / wall_clock_time

Percentage loads from L2 per cycle
100.0 * PM_L2_LD / PM_CYC

Total Loads from memory
tot_ld_mem = (PM_DATA_FROM_RMEM + PM_DATA_FROM_LMEM) / (1024*1024)

Memory load traffic
L3_cache_line_size * tot_ld_mem

Memory load bandwidth per processor
L3_cache_line_size * tot_ld_mem / wall_clock_time

Percentage loads from memory per cycle
100.0 * (PM_DATA_FROM_RMEM + PM_DATA_FROM_LMEM) / PM_CYC

Total Loads from local memory
tot_ld_lmem = (double)PM_DATA_FROM_LMEM / (1024*1024)

Local memory load traffic
L3_cache_line_size * tot_ld_lmem

Local memory load bandwidth per processor
L3_cache_line_size * tot_ld_lmem / wall_clock_time

Percentage loads from local memory per cycle
100.0 (double)PM_DATA_FROM_LMEM / PM_CYC

Percentage TLB misses per cycle
100.0 * (double)X_DTLB_LOAD_MISSES_ANY / PM_CYC

Load from L2 cache miss rate
100.0 * PM_L2_LD_MISS / PM_L2_LD

Load from L2 cache hit rate
100.0 * PM_L2_LD_HIT / PM_L2_LD

L2 cache miss rate
100.0 * X_L2_RQSTS_MISS / X_L2_RQSTS_REFERENCES

Instruction fetch from L2 cache miss rate
100.0 * X_L2_RQSTS_IFETCH_MISS / X_L2_RQSTS_IFETCHES

Instruction fetch from L2 cache hit rate
100.0 * X_L2_RQSTS_IFETCH_HIT / X_L2_RQSTS_IFETCHES

L2 cache store RFO miss rate
100.0 * X_L2_RQSTS_RFO_MISS / X_L2_RQSTS_RFOS

L2 cache store RFO hit rate =
100.0 * X_L2_RQSTS_RFO_HIT / X_L2_RQSTS_RFOS

Derived metrics description for x86 Sandy Bridge

The derived metrics for x86 Sandy Bridge are as follows:

MIPS PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
(double)PM_INST_CMPL / PM_CYC

Branches mispredicted percentage
100.0 * (double)PM_BR_MPRED / PM_BRU_FIN

Instruction fetch from L2 cache miss rate
100.0 * X_L2_RQSTS_IFETCH_MISS / X_L2_RQSTS_IFETCHES

Instruction fetch from L2 cache hit rate
100.0 * X_L2_RQSTS_IFETCH_HIT / X_L2_RQSTS_IFETCHES

L2 cache store RFO miss rate
100.0 * X_L2_RQSTS_RFO_MISS / X_L2_RQSTS_RFOS

L2 cache store RFO hit rate
100.0 * X_L2_RQSTS_RFO_HIT / X_L2_RQSTS_RFOS

L2 cache prefetch miss rate
100.0 * X_L2_RQSTS_PREFETCH_MISS / X_L2_RQSTS_PREFETCH_ALL

L2 cache store lock request hit rate
100.0 * (X_L2_STORE_LOCK_RQSTS_HIT_E +
X_L2_STORE_LOCK_RQSTS_HIT_M) / X_L2_STORE_LOCK_RQSTS_ALL

L2 cache store lock request miss rate
100.0 * X_L2_STORE_LOCK_RQSTS_MISS /
X_L2_STORE_LOCK_RQSTS_ALL

L2 cache prefetch hit rate
100.0 * X_L2_RQSTS_PREFETCH_HIT / X_L2_RQSTS_PREFETCH_ALL

last level cache miss rate
100.0 * X_LONGEST_LAT_CACHE_MISS /
X_LONGEST_LAT_CACHE_REFERENCE

Total SSE or AVX-128 floating-point operations
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_SCALAR_SINGLE +
X_FP_COMP_OPS_SSE_PACKED_SINGLE +
X_FP_COMP_OPS_SSE_SCALAR_DOUBLE) * 0.000001

SSE or AVX-128 double precision floating-point operations per second (WCT)
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_SCALAR_DOUBLE) * 0.000001 / wall_clock_time

SSE or AVX-128 single precision floating-point operations per second (WCT)
(X_FP_COMP_OPS_SSE_PACKED_SINGLE +
X_FP_COMP_OPS_SSE_SCALAR_SINGLE) * 0.000001 / wall_clock_time

SSE or AVX-128 double precision floating-point operations per second (user time)
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_SCALAR_DOUBLE) * 0.000001 / user_time

SSE or AVX-128 single precision floating-point operations per second (user time)
(X_FP_COMP_OPS_SSE_PACKED_SINGLE +
X_FP_COMP_OPS_SSE_SCALAR_SINGLE) * 0.000001 / user_time

Percentage of any resource stalls
100.0 * X_RESOURCE_STALLS_ANY / PM_CYC

Percentage of stalls due to store buffers
100.0 * X_RESOURCE_STALLS_SB / PM_CYC

Percentage of stalls due to load buffers
100.0 * X_RESOURCE_STALLS_LB / PM_CYC

Percentage of stalls due to resource scheduler
100.0 * X_RESOURCE_STALLS_RS / PM_CYC

Percentage of stalls due to reorder buffer
100.0 * X_RESOURCE_STALLS_ROB / PM_CYC

Percentage of stalls due to branches in progress
100.0 * X_RESOURCE_STALLS2_BOB_FULL / PM_CYC

Estimated percentage of L2 latency
12.0 * X_MEM_LOAD_UOPS_RETIRED_L2_HIT / PM_CYC

Estimated percentage of L3 hits
26.0 * X_MEM_LOAD_UOPS_RETIRED_LLC_HIT / PM_CYC

Estimated percentage of hits in the cache of other cores
43.0 * X_MEM_LOAD_UOPS_LLC_HIT_RETIRED_XSNP_HIT / PM_CYC

Estimated percentage of memory latency
200.0 * X_MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS / PM_CYC

Percentage of lock contention
100.0 * X_MEM_LOAD_UOPS_LLC_HIT_RETIRED_XSNP_HITM /
X_MEM_UOPS_RETIRED_LOCK_LOAD

Estimated percentage of store forward blocks
100.0 * X_LD_BLOCKS_STORE_FORWARD * 13.0 / PM_CYC

Estimated percentage of split loads
100.0 * X_MEM_UOPS_RETIRED_SPLIT_STORE * 5.0 / PM_CYC

Split store ratio
X_MEM_UOPS_RETIRED_SPLIT_STORE /
X_MEM_UOPS_RETIRED_ALL_STORE

Estimated 4K load or store aliasing conflict percentage
100.0 * X_LD_BLOCKS_PARTIAL_ADDRESS_ALIAS * 5.0 / PM_CYC

Estimated cost of STLB hits on loads percentage
100.0 * X_DTLB_LOAD_MISSES_STLB_HIT * 7.0 / PM_CYC

Cost of page walks percentage
100.0 * X_DTLB_LOAD_MISSES_WALK_DURATION / PM_CYC

STLB load miss walk cost percentage
100.0 * X_MEM_UOPS_RETIRED_STLB_MISS_LOAD /
X_MEM_UOPS_RETIRED_ALL_LOAD

STLB load miss average cost
X_DTLB_LOAD_MISSES_WALK_DURATION /
X_DTLB_LOAD_MISSES_WALK_COMPLETED

STLB store miss percentage
100.0 * X_MEM_UOPS_RETIRED_STLB_MISS_STORE /
X_MEM_UOPS_RETIRED_ALL_STORE

Percentage slow LEA instructions allocated
100.0 * X_PARTIAL_RAT_STALLS_SLOW_LEA_WINDOW / PM_CYC

Percentage floating point assists for X87 instructions
100.0 * X_FP_ASSIST_ANY / PM_INST_CMPL

Estimated percentage of AVX to SSE transition
75.0 * X_OTHER_ASSISTS_AVX_TO_SSE / PM_CYC

Estimated percentage of SSE to AVX transition
75.0 * X_OTHER_ASSISTS_SSE_TO_AVX / PM_CYC

Estimated percentage of AVX store assists
150.0 * X_OTHER_ASSISTS_AVX_STORE / PM_CYC

Percentage decoded ICache to legacy decode pipeline switch cost
100.0 * X_DSB2MITE_SWITCHES_PENALTY_CYCLES / PM_CYC

Average cost of decoded ICache to legacy decode pipeline switch
X_DSB2MITE_SWITCHES_PENALTY_CYCLES /
X_DSB2MITE_SWITCHES_COUNT

Percentage DSB exceeds way limit
100.0 * X_DSB_FILL_EXCEED_DSB_LINES / X_DSB_FILL_ALL_CANCEL

Estimated percentage of instruction length decoder cost
100.0 * X_ILD_STALL_LCP * 3.0 / PM_CYC

ICache miss per instruction retired
PM_L1_ICACHE_MISS / PM_INST_CMPL

Total vector operations
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_PACKED_SINGLE \+
X_SIMD_FP_256_PACKED_SINGLE + X_SIMD_FP_256_PACKED_DOUBLE
) * 0.000001

SSE vector operations
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_PACKED_SINGLE) * 0.000001

SSE vector instructions percentage
100.0 * (X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_PACKED_SINGLE) / PM_CYC

SSE vector instructions per instructions retired
(X_FP_COMP_OPS_SSE_PACKED_DOUBLE +
X_FP_COMP_OPS_SSE_PACKED_SINGLE) / PM_INST_CMPL

SSE double precision vector instructions percentage
100.0 * X_FP_COMP_OPS_SSE_PACKED_DOUBLE / PM_CYC

SSE double precision vector instructions per instructions retired
X_FP_COMP_OPS_SSE_PACKED_DOUBLE / PM_INST_CMPL

SSE single precision vector instructions percentage
100.0 * X_FP_COMP_OPS_SSE_PACKED_SINGLE / PM_CYC

SSE single precision vector instructions per instructions retired
X_FP_COMP_OPS_SSE_PACKED_SINGLE / PM_INST_CMPL

AVX vector operations
(X_SIMD_FP_256_PACKED_SINGLE +
X_SIMD_FP_256_PACKED_DOUBLE) * 0.000001

AVX vector instructions percentage
100.0 * (X_SIMD_FP_256_PACKED_SINGLE +
X_SIMD_FP_256_PACKED_DOUBLE) / PM_CYC

AVX vector instructions per instructions retired
(X_SIMD_FP_256_PACKED_SINGLE +
X_SIMD_FP_256_PACKED_DOUBLE) / PM_INST_CMPL

AVX double precision vector instructions percentage
100.0 * X_SIMD_FP_256_PACKED_DOUBLE / PM_CYC

AVX double precision vector instructions per instructions retired
X_SIMD_FP_256_PACKED_DOUBLE / PM_INST_CMPL

AVX single precision vector instructions percentage
100.0 * X_SIMD_FP_256_PACKED_SINGLE / PM_CYC

AVX single precision vector instructions per instructions retired
X_SIMD_FP_256_PACKED_SINGLE / PM_INST_CMPL

X87 instructions percentage
100.0 * X_FP_COMP_OPS_X87 / PM_CYC

X87 instructions per instructions retired
X_FP_COMP_OPS_X87 / PM_INST_CMPL

Derived metrics description for x86 Ivy Bridge

The derived metrics for x86 Ivy Bridge are as follows:

MIPS PM_INST_CMPL * 0.000001 / wall_clock_time

Instructions per cycle
(double)PM_INST_CMPL / PM_CYC

Branches mispredicted percentage
100.0 * (double)PM_BR_MPRED / PM_BRU_FIN

Instruction fetch from L2 cache miss rate
100.0 * X_L2_RQSTS_IFETCH_MISS / X_L2_RQSTS_IFETCHES

Instruction fetch from L2 cache hit rate
100.0 * X_L2_RQSTS_IFETCH_HIT / X_L2_RQSTS_IFETCHES

L2 cache store RFO miss rate
100.0 * X_L2_RQSTS_RFO_MISS / X_L2_RQSTS_RFOS

L2 cache store RFO hit rate
100.0 * X_L2_RQSTS_RFO_HIT / X_L2_RQSTS_RFOS

L2 cache prefetch miss rate
100.0 * X_L2_RQSTS_PREFETCH_MISS / X_L2_RQSTS_PREFETCH_ALL

L2 cache prefetch hit rate
100.0 * X_L2_RQSTS_PREFETCH_HIT / X_L2_RQSTS_PREFETCH_ALL

last level cache miss rate
100.0 * X_LONGEST_LAT_CACHE_MISS /
X_LONGEST_LAT_CACHE_REFERENCE

Percentage of any resource stalls
100.0 * X_RESOURCE_STALLS_ANY / PM_CYC

Percentage of stalls due to store buffers
100.0 * X_RESOURCE_STALLS_SB / PM_CYC

Percentage of stalls due to resource scheduler
100.0 * X_RESOURCE_STALLS_RS / PM_CYC

Percentage of stalls due to reorder buffer
100.0 * X_RESOURCE_STALLS_ROB / PM_CYC

Estimated percentage of L2 latency
12.0 * X_MEM_LOAD_UOPS_RETIRED_L2_HIT / PM_CYC

Estimated percentage of L3 hits
26.0 * X_MEM_LOAD_UOPS_RETIRED_LLC_HIT / PM_CYC

Estimated percentage of hits in the cache of other cores
43.0 * X_MEM_LOAD_UOPS_LLC_HIT_RETIRED_XSNP_HIT / PM_CYC

Percentage of lock contention
100.0 * X_MEM_LOAD_UOPS_LLC_HIT_RETIRED_XSNP_HITM /
X_MEM_UOPS_RETIRED_LOCK_LOAD

Estimated percentage of store forward blocks
100.0 * X_LD_BLOCKS_STORE_FORWARD * 13.0 / PM_CYC

Estimated percentage of split loads
100.0 * X_MEM_UOPS_RETIRED_SPLIT_STORE * 5.0 / PM_CYC

Split store ratio
X_MEM_UOPS_RETIRED_SPLIT_STORE /
X_MEM_UOPS_RETIRED_ALL_STORE

Estimated 4K load or store aliasing conflict percentage
100.0 * X_LD_BLOCKS_PARTIAL_ADDRESS_ALIAS * 5.0 / PM_CYC

Estimated cost of STLB hits on loads percentage
100.0 * X_DTLB_LOAD_MISSES_STLB_HIT * 7.0 / PM_CYC

Cost of page walks percentage
100.0 * X_DTLB_LOAD_MISSES_WALK_DURATION / PM_CYC

STLB load miss walk cost percentage
100.0 * X_MEM_UOPS_RETIRED_STLB_MISS_LOAD /
X_MEM_UOPS_RETIRED_ALL_LOAD

STLB load miss average cost
X_DTLB_LOAD_MISSES_WALK_DURATION /
X_DTLB_LOAD_MISSES_WALK_COMPLETED

STLB store miss percentage
100.0 * X_MEM_UOPS_RETIRED_STLB_MISS_STORE /
X_MEM_UOPS_RETIRED_ALL_STORE

Percentage floating point assists for X87 instructions
100.0 * X_FP_ASSIST_ANY / PM_INST_CMPL

Estimated percentage of AVX to SSE transition
75.0 * X_OTHER_ASSISTS_AVX_TO_SSE / PM_CYC

Estimated percentage of SSE to AVX transition
75.0 * X_OTHER_ASSISTS_SSE_TO_AVX / PM_CYC

Estimated percentage of AVX store assists
150.0 * X_OTHER_ASSISTS_AVX_STORE / PM_CYC

Percentage decoded ICache to legacy decode pipeline switch cost
100.0 * X_DSB2MITE_SWITCHES_PENALTY_CYCLES / PM_CYC

Average cost of decoded ICache to legacy decode pipeline switch
X_DSB2MITE_SWITCHES_PENALTY_CYCLES /
X_DSB2MITE_SWITCHES_COUNT

Estimated percentage of instruction length decoder cost
100.0 * X_ILD_STALL_LCP * 3.0 / PM_CYC

ICache miss per instruction retired
PM_L1_ICACHE_MISS / PM_INST_CMPL

Percentage that the microarchitecture pipeline is front end bound
100.0 * X_IDQ_UOPS_NOT_DELIVERED_CORE / 4.0 * PM_CYC

Percentage of micro-ops retired
100.0 * X_UOPS_RETIRED_SLOTS / 4.0 * PM_CYC

Percentage of micro-ops coming from the decoded ICache
X_IDQ_DSB_UOPS / (X_IDQ_DSB_UOPS + X_IDQ_MITE_UOPS + X_IDQ_MS_UOPS)

Percentage of micro-ops coming from the legacy decoder pipeline
X_IDQ_MITE_UOPS / (X_IDQ_DSB_UOPS + X_IDQ_MITE_UOPS + X_IDQ_MS_UOPS)

Percentage of micro-ops coming from the micro-sequencer
X_IDQ_MS_UOPS / (X_IDQ_DSB_UOPS + X_IDQ_MITE_UOPS + X_IDQ_MS_UOPS)

Clocks per instruction retired ratio (CPI)
PM_CYC / PM_INST_CMPL)

Branch mispredictions per micro-op retired
PM_BR_MPRED / PM_INST_CMPL)

Percentage of L1 data cache misses
X_L1D_PEND_MISSES / PM_CYC

L1 data cache misses per instruction retired
X_L1D_PEND_MISSES / PM_INST_CMPL

Back end bottlenecks
(X_CYCLE_ACTIVITY_CYCLES_NO_EXECUTE + X_UOPS_EXECUTED_THREAD) / PM_CYC

Percentage of branch mispredicts
20.0 * X_BR_MISP_RETIRED_ALL_BRANCHES / PM_CYC

Level 2 ICache miss per instruction retired
X_L2_RQSTS_IFETCH_MISS / PM_INST_CMPL

Percentage of load and store uops dispatched
100.0 * (X_UOPS_DISPATCHED_PORT_2 + X_UOPS_DISPATCHED_PORT_3 + X_UOPS_DISPATCHED_PORT_4) / PM_CYC

Percentage of load and store address uops dispatched
100.0 * (X_UOPS_DISPATCHED_PORT_2 + X_UOPS_DISPATCHED_PORT_3) / PM_CYC

Percentage of load uops dispatched
100.0 * (X_UOPS_DISPATCHED_PORT_2_LD + X_UOPS_DISPATCHED_PORT_3_LD) / PM_CYC

Percentage of store uops dispatched
100.0 * (X_UOPS_DISPATCHED_PORT_2_STA + X_UOPS_DISPATCHED_PORT_3_STA + X_UOPS_DISPATCHED_PORT_4) / PM_CYC

Percentage of computation and branch uops dispatched

100.0 * (X_UOPS_DISPATCHED_PORT_0 + X_UOPS_DISPATCHED_PORT_1 + X_UOPS_DISPATCHED_PORT_5) / PM_CYC

Percentage of Port 0 and Port 1 uops dispatched (approximates floating point performance)

100.0 * (X_UOPS_DISPATCHED_PORT_0 + X_UOPS_DISPATCHED_PORT_1) / PM_CYC

Percentage of Port 1 and Port 5 uops dispatched (approximates integer vector performance)

100.0 * (X_UOPS_DISPATCHED_PORT_1 + X_UOPS_DISPATCHED_PORT_5) / PM_CYC

Ratio of computation and branch uop dispatched per data store uop dispatched

(X_UOPS_DISPATCHED_PORT_0 + X_UOPS_DISPATCHED_PORT_1 + X_UOPS_DISPATCHED_PORT_5) / X_UOPS_DISPATCHED_PORT_4

Appendix C. HPC Toolkit environment variables

Table 56 is an alphabetical listing all of the HPC Toolkit environment variables and the command or API for which they are used. For a description of each environment variable, see the referenced command or API.

Table 56. HPC Toolkit environment variables

Environment variable	Used by:
GPROF	"Xprof - Display the application as a function tree" on page 256
HPC_EXCEPTION_COUNT	"hpcrun - Launch a program to collect profiling or trace data" on page 241
HPC_EXCEPTION_METRIC	"hpcrun - Launch a program to collect profiling or trace data" on page 241
HPC_OUTPUT_NAME	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPC_TRACE_MAX_BUFFERS	"hpcrun - Launch a program to collect profiling or trace data" on page 241
HPC_TRACE_STORE	"hpcrun - Launch a program to collect profiling or trace data" on page 241
HPC_UNIQUE_FILE_NAME	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_AGGREGATE	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcrun - Launch a program to collect profiling or trace data" on page 241 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_ASC_OUTPUT	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_COUNTING_MODE	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265

© Copyright IBM Corporation © IBM 2008, 2013

Table 56. HPC Toolkit environment variables (continued)

Environment variable	Used by:
HPM_EVENT_SET	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_EXCLUSIVE_VALUES	"hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_PRINT_FORMULA	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_PRINT_TASK	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_ROUND_ROBIN_CLUSTER	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_SLICE_DURATION **Note:** This environment variable is only supported for AIX.	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_STDOUT	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265
HPM_VIZ_OUTPUT	• "hpccount - Report hardware performance counter statistics for an application" on page 236 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265

Table 56. HPC Toolkit environment variables (continued)

Environment variable	Used by:
IHPCT_BASE	• "dataview - View an I/O trace" on page 234 • "hpcrun - Launch a program to collect profiling or trace data" on page 241 • "hpcstat - Reports a system-wide summary of hardware performance counter statistics" on page 244 • "hpctInst - Instrument applications to obtain performance data" on page 248 • "peekperf - Provides a GUI for the HPC Toolkit" on page 252 • "peekview - Display an MPI trace file" on page 254 • "Xprof - Display the application as a function tree" on page 256 • "hpmInit, f_hpminit - Initialize the Hardware Performance Monitor (HPM) run-time environment" on page 265 • "MT_get_allresults - Obtain statistical results" on page 285 • "MT_get_mpi_bytes - Obtain the accumulated number of bytes transferred" on page 293 • "MT_get_mpi_counts - Obtain the the number of times a function was called" on page 294 • "MT_get_mpi_name - Returns the name of the specified MPI function" on page 295 • "MT_get_time - Get the elapsed time" on page 297 • "MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
LD_LIBRARY_PATH	"hpctInst - Instrument applications to obtain performance data" on page 248
MAX_TRACE_EVENTS	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
MAX_TRACE_RANK	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
MT_BASIC_TRACE	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
OUTPUT_ALL_RANKS	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
POMP_LOOP	"hpctInst - Instrument applications to obtain performance data" on page 248
POMP_PARALLEL	"hpctInst - Instrument applications to obtain performance data" on page 248
POMP_USER	"hpctInst - Instrument applications to obtain performance data" on page 248
TRACE_ALL_EVENTS	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
TRACE_ALL_TASKS	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303
TRACEBACK_LEVEL	"MT_trace_start, mt_trace_start - Start or resume the collection of trace events" on page 303

Accessibility features for IBM PE Developer Edition

Accessibility features help users who have a disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in IBM PE Developer Edition:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers

The *IBM Cluster information center*, and its related publications, are accessibility-enabled. The accessibility features of the information center are described in the IBM Cluster information center (http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.addinfo.doc/access.html).

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

IBM and accessibility

See the IBM Human Ability and Accessibility Center (http://www.ibm.com/able/) for more information about the commitment that IBM has to accessibility.

© Copyright IBM Corp. 2008, 2013

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

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

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corporation © IBM 2008, 2013

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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

IBM Corporation
Mail Station P300
2455 South Road,
Poughkeepsie, NY 12601-5400
USA

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample

programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" (www.ibm.com/legal/copytrade.shtml).

Intel, is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Index

A

about this information xi
about Xprof
  before you begin 133
accessibility 331
  keyboard 331
  shortcut keys 331
additional trace controls 216
aggregator functions
  understanding 208
AIX and Linux on Power6
  derived metrics description 315
AIX and Linux on POWER7 and POWER7+
  derived metrics description 316
AIX installation using SMIT 12
AIX systems
  installing the IBM HPC Toolkit 11
alphabetical listing
  HPC Toolkit environment variables 327
alternative paths
  setting 146
ancestor functions
  locating and displaying 167
APIs 261
  f_hpm_error 263
  f_hpminit 265
  f_hpmstart 269
  f_hpmstartx 271
  f_hpmstop 274
  f_hpmterminate 276
  f_hpmtstart 278
  f_hpmtstartx 280
  f_hpmtstop 283
  hpm_error_count 263
  hpmInit 265
  hpmStart 269
  hpmStartx 271
  hpmStop 274
  hpmTerminate 276
  hpmTstart 278
  hpmTstartx 280
  hpmTstop 283
  MT_get_allresults 285
  MT_get_calleraddress 288
  MT_get_callerinfo 289
  MT_get_elapsed_time 291
  MT_get_environment 292
  MT_get_mpi_bytes 293
  MT_get_mpi_counts 294
  MT_get_mpi_name 295
  MT_get_mpi_time 296
  MT_get_time 297
  MT_get_tracebufferinfo 298
  MT_output_text 299
  MT_output_trace 300
  MT_trace_event 301
  mt_trace_start 303
  MT_trace_start 303
  mt_trace_stop 305
  MT_trace_stop 305

B

basic call data
  understanding 169
basic cluster data
  understanding 169
basic data
  obtaining 167
basic function data
  understanding 167
binary executable file
  specifying 143

C

call graph profile report
  understanding 174
changing the search sequence 147
child functions
  locating and displaying 166
clustering functions
  Xprof 162
clustering libraries
  Xprof 162
collecting performance data
  from a subset of application tasks 5
command and API reference 231
commands 233
  dataview 234
  hpccount 199, 236
  hpcrun 241
  hpcstat 200, 244
  hpctInst 248
  peekperf 252
  peekview 254
  Xprof 256
comparing Xprof and the gprof command 134
compiling and linking with libmpitrace 215
compiling applications to be profiled
  using the -pg flag 134
compiling, editing, and running programs
  using the peekperf GUI 98
configuration file
  loading 182
  saving 181
considerations for MPI programs 207, 208, 209
  distributor and aggregator interfaces 210
  plug-ins shipped with the tool kit 208
controlling how the display is updated 152
controlling the appearance of the Xprof main window 191
controlling the graphic style of the function call tree
  Xprof 153
controlling the orientation of the function call tree
  Xprof 153
controlling the representation of the function call tree
  Xprof 154
controlling traced tasks 216
controlling variables related to the file menu 192
controlling variables related to the filter menu 196
controlling variables related to the screen dump option 193
controlling variables related to the view menu 195

© Copyright IBM Corp. 2008, 2013

conventions
file naming 309
conventions and terminology xi
CPU hardware counter multiplexing
understanding 203
customizing MPI profiling data 217
customizing Xprof resources 190

D

data file naming
HPC_OUTPUT_NAME 309
HPC_UNIQUE_FILE_NAME 309
dataview command 234
default paths
setting 146
derived metrics 315
understanding 204
derived metrics description
for AIX and Linux on POWER7 and POWER7+ 316
for x86 Ivy Bridge 322
for x86 Nehalem and Westmere 317
for x86 Sandy Bridge 318
derived metrics description for AIX and Linux on Power6
hpccount, hpcstat, and libhpc 315
descendant functions
locating and displaying 167
descriptions
derived metrics 315
detailed data
getting from reports 171
disability 331
disassembler code
viewing 185
disassembler code window
using 185
display
controlling how it is updated 152
display status field, Xprof 150
displaying the entire function call tree
Xprof 156
distributor and aggregator interfaces
understanding 210
distributor functions
understanding 208

E

Eclipse plug-in
using hardware performance counters 51
excluding and including specific objects
Xprof 156

F

f_hpm_error API 263
f_hpminit API 265
f_hpmstart API 269
f_hpmstartx API 271
f_hpmstop API 274
f_hpmterminate API 276
f_hpmtstart API 278
f_hpmtstartx API 280
f_hpmtstop API 283
file naming conventions 309
file naming examples 311
file search sequence
setting 146
files
loading from the Xprof GUI 140
filtering by function characteristics
Xprof 157
filtering shared library functions
Xprof 156
filtering what you see
Xprof 155
flat profile report
understanding 172
fonts
controlling 190
function call summary report
understanding 178
function call tree
displaying 156
restoring the status 155
function characteristics
filtering 157
function index report
understanding 177
function menu statistics report option
using 170
functions
clustering 162
how calls between functions are depicted 152
how they are represented 150
unclustering 164
functions on a cycle
locating and displaying 167

G

getting detailed data from reports 171
getting the plug-ins to work 211

H

handling multithreaded program instrumentation issues 207
handling of overlap issues 206
hardware performance counter data
viewing 58
hardware performance counter plug-ins 208
hardware performance counter tools 199
CPU hardware counter multiplexing 203
derived metrics 204
hpccount command 199
hpcstat command 200
inclusive and exclusive event counts 205
inheritance, understanding 204
measurement overhead 206
MFlop issues 204
multithreaded program instrumentation 207
overlap issues 206
parent-child relationships 205
using the libhpc library 201
hardware performance counters
instrumenting your application 227
hardware performance counters in peekperf
using 101
hidden menus, Xprof 149
how calls between functions are depicted 152
how functions are represented 150
how library clusters are represented 152

HPC Toolkit Eclipse plug-in
using 21
HPC Toolkit environment variables
alphabetical listing 327
hpccount command 236
using 199
hpccount, hpcstat, and libhpc
derived metrics description for AIX and Linux on Power6 315
hpcrun command 8, 241
hpcstat command 244
using 200
hpctInst command 248
hpctInst utility 8
instrumenting your application 227
hpm_error_count API 263
hpmInit API 265
hpmStart API 269
hpmStartx API 271
hpmStop API 274
hpmTerminate API 276
hpmTstart API 278
hpmTstartx API 280
hpmTstop API 283

I

I/O profiling
instrumenting your application 229
using in the plug-in 73
I/O profiling data
viewing 48
I/O profiling environment variables
setting 219
I/O profiling in peekperf
using 123
I/O profiling library
using 219
I/O profiling library module options
performance data file naming 224
running your application 223
specifying 221
IBM High Performance Computing Toolkit
introduction 3
IBM HPC Toolkit
installing 11
introduction 3
IBM HPC Toolkit Eclipse plug-in
instrumenting your application 34
instrumenting your application at launch 22, 51, 59, 67, 73
instrumenting your application before launch 31, 60, 68, 75
loading the application 32
preparing your application 21
running the application 34
running the application at launch 22
using 21
IBM PE Developer Edition Eclipse feature
prerequisite installation 17
including and excluding parent and child functions
Xprof 160
inclusive and exclusive event counts
understanding 205
information boxes
understanding 169
inheritance
understanding 204
installing the IBM HPC Toolkit 11
installing the IBM HPC Toolkit on AIX systems 11, 12
installing the IBM HPC Toolkit on Linux systems 13
installing the IBM PE Developer Edition Eclipse feature 17
instrumentation models 6
instrumenting your application
IBM HPC Toolkit Eclipse plug-in 34
using hardware performance counters in peekperf 101
using hardware performance counters in the plug-in 53
using I/O profiling in peekperf 123
using I/O profiling in the plug-in 75
using MPI profiling in peekperf 107
using MPI profiling in the plug-in 61
using OpenMP profiling in peekperf 117
using OpenMP profiling in the plug-in 68
using the peekperf GUI 89
instrumenting your application at launch
IBM HPC Toolkit Eclipse plug-in 22, 51, 59, 67, 73
instrumenting your application before launch
IBM HPC Toolkit Eclipse plug-in 31, 60, 68, 75
using hardware performance counters in the plug-in 53
instrumenting your application for hardware performance counters 227
instrumenting your application for I/O profiling 229
instrumenting your application for MPI profiling 228
instrumenting your application for OpenMP profiling 229
instrumenting your application using the hpctInst utility 227
introduction
hpcrun command 8
hpctInst utility 8
IBM High Performance Computing Toolkit 3
introduction to the IBM HPC Toolkit 3
PE Developer Edition components 3
performance analysis 4
user interfaces 4

L

libhpc library
using 201
libraries
clustering 162
library clusters
how they are represented 152
library module options
specifying 221
library statistics report
understanding 179
limitations and restrictions xvii
Linux systems
installing the IBM HPC Toolkit 13
loading a configuration file 182
loading an application
using the peekperf GUI 84
loading files from the Xprof GUI 140
loading the application 32
locating and displaying ancestor functions 167
locating and displaying child functions 166
locating and displaying descendant functions 167
locating and displaying functions on a cycle 167
locating and displaying parent functions 166
locating specific objects in the function call tree 165

M

main menus, Xprof 149
main window, Xprof 148
measurement overhead
  understanding 206
MFlop issues
  understanding 204
miscellaneous peekperf features
  using the peekperf GUI 99
MPI profiling
  instrumenting your application 228
  using in peekperf 107
  using in the plug-in 59
MPI profiling data
  customizing 217
MPI profiling library
  using 215
MPI profiling utility functions
  understanding 218
MPI programs
  considerations 207
  general considerations 207
MPI trace tool
  performance data file naming 218
MPI traces
  viewing 47
MT_get_allresults API 285
MT_get_calleraddress API 288
MT_get_callerinfo API 289
MT_get_elapsed_time API 291
MT_get_environment API 292
MT_get_mpi_bytes API 293
MT_get_mpi_counts API 294
MT_get_mpi_name API 295
MT_get_mpi_time API 296
MT_get_time API 297
MT_get_tracebufferinfo API 298
MT_output_text API 299
MT_output_trace API 300
MT_trace_event API 301
mt_trace_start API 303
MT_trace_start API 303
mt_trace_stop API 305
MT_trace_stop API 305
multithreaded program instrumentation
  handling issues 207

O

obtaining basic data 167
obtaining performance data for your application 167
OpenMP profiling
  instrumenting your application 229
  using in peekperf 117
  using in the plug-in 67
other viewing options
  Xprof 153
overlap issues
  handling 206
overview
  instrumentation models 6
  performance measurement tools 5

P

parent and child functions
  including and excluding 160
parent functions
  locating and displaying 166
parent-child relationships
  understanding 205
peekperf command 252
peekperf GUI
  using 83
peekview command 254
performance analysis
  introduction to the IBM HPC Toolkit 4
  user interfaces 4
performance data 42
  obtaining for your application 167
performance data file naming
  HPC_OUTPUT_NAME 309
  HPC_UNIQUE_FILE_NAME 309
  I/O profiling library module options 224
  understanding 218
performance measurement tools 5
plug-ins
  getting them to work 211
  hardware performance counter 208
plug-ins shipped
  with the tool kit 208
preparing your application 21
  using hardware performance counters in peekperf 101
  using hardware performance counters in the plug-in 51
  using I/O profiling in peekperf 123
  using I/O profiling in the plug-in 73
  using MPI profiling in peekperf 107
  using MPI profiling in the plug-in 59
  using OpenMP profiling in peekperf 117
  using the peekperf GUI 83
prerequisite information xii
Profile Configurations dialog
  using 35
profile data files
  specifying 144

R

related information xii
  Parallel Tools Platform component xii
reports
  saving 181
  saving to a file 180
requirements and limitations
  Xprof 133
resource variables
  setting 190
restoring the status of the function call tree
  Xprof 155
restrictions and limitations xvii
running the application 34
running the application at launch 22
running your application
  I/O profiling library module options 223
  using hardware performance counters in the plug-in 55
  using I/O profiling in peekperf 125
  using I/O profiling in the plug-in 73, 76
  using MPI profiling in peekperf 111
  using MPI profiling in the plug-in 59, 62
  using OpenMP profiling in peekperf 119

running your application *(continued)*
using OpenMP profiling in the plug-in 67, 69
using the peekperf GUI 89
running your application at launch
using hardware performance counters in the plug-in 51
running your program
using hardware performance counters in peekperf 103

S

saving a configuration file 181
saving a single report 180
saving reports to a file 180, 181
saving screen images of profiled data 187
saving summarized data from multiple profile data files 181
screen images of profiled data
saving 187
search sequence
changing 147
setting alternative paths 146
setting default paths 146
setting I/O profiling environment variables 219
setting the file search sequence 146
setting Xprof resource variables 190
controlling fonts 190
shared library functions
filtering 156
shortcut keys
keyboard 331
single report
saving 180
SMIT
AIX installation 12
source code
viewing 183
source code window
using 183
specific objects in the function call tree
locating 165
specifying profile data files 144
specifying the binary executable file 143
specifying Xprof flags from the command line 136
specifying Xprof flags from the GUI 145
starting the Xprof GUI 135
summarized data
saving from multiple data files 181

T

tool kit
plug-ins shipped 208
tools
hardware performance counter 199
traced tasks
controlling 216
trademarks 335

U

unclustering functions
Xprof 164
understanding
distributor and aggregator interfaces 210
understanding aggregator functions 208
understanding basic call data 169
understanding basic cluster data 169
understanding basic function data 167
understanding CPU hardware counter multiplexing 203
understanding derived metrics 204
understanding distributor functions 208
understanding inclusive and exclusive event counts 205
understanding information boxes 169
understanding inheritance 204
understanding measurement overhead 206
understanding MFlop issues 204
understanding MPI profiling utility functions 218
understanding parent-child relationships 205
understanding the call graph profile report 174
understanding the flat profile report 172
understanding the function call summary report 178
understanding the function index report 177
understanding the library statistics report 179
understanding the Xprof display 147
understanding why user-defined plug-ins are useful 209
user-defined plug-ins
understanding 209
why they are useful 209
using hardware performance counters in peekperf 101
instrumenting your application 101
running your program 103
viewing hardware performance counter data 104
using hardware performance counters in the Eclipse plug-in 51
using hardware performance counters in the plug-in
instrumenting your application 53
instrumenting your application before launch 53
preparing your application 51
running your application 55
running your application at launch 51
using I/O profiling in peekperf 123
instrumenting your application 123
preparing your application 123
running your application 125
viewing I/O profiling data 125
using I/O profiling in the plug-in 73
instrumenting your application 75
preparing your application 73
running your application 73, 76
viewing I/O Data 77
using installp 12
using MPI profiling in peekperf 107
instrumenting your application 107
preparing your application 107
running your application 111
viewing MPI profiling data 111
using MPI profiling in the plug-in 59
instrumenting your application 61
preparing your application 59
running your application 59, 62
viewing MPI profiling data 64
using OpenMP profiling in peekperf 117
instrumenting your application 117
preparing your application 117
running your application 119
viewing performance data 119
using OpenMP profiling in the plug-in 67
instrumenting your application 68
preparing your application 67
running your application 67, 69
viewing OpenMP performance data 69
using the disassembler code window 185
using the function menu statistics report option 170
using the I/O profiling library 219

using the I/O profiling library *(continued)*
collecting performance data from a subset of application tasks 5
performance data file naming 224
running your application 223
setting I/O profiling environment variables 219
specifying module options 221
using the IBM HPC Toolkit Eclipse plug-in 21
using the libhpc library 201
using the MPI profiling library 215, 216
additional trace controls 216
compiling and linking with libmpitrace 215
customizing MPI profiling data 217
performance data file naming 218
understanding MPI profiling utility functions 218
using the peekperf GUI 83
compiling, editing, and running programs 98
instrumenting your application 89
loading an application 84
miscellaneous peekperf features 99
preparing your application 83
running your application 89
viewing I/O profiling data 97
viewing MPI traces 95
viewing performance data 91
using the Profile Configurations dialog 35
using the source code window 183

V

variables related to the file menu
controlling 192
variables related to the filter menu
controlling 196
variables related to the screen dump option
controlling 193
variables related to the view menu
controlling 195
viewing 42
disassembler code 182
hardware performance counter data 58
source code 182, 183
viewing hardware performance counter data
using hardware performance counters in peekperf 104
viewing I/O Data
using I/O profiling in the plug-in 77
viewing I/O profiling data 48
using I/O profiling in peekperf 125
using the peekperf GUI 97
viewing MPI profiling data
MPI profiling data
viewing 111
using MPI profiling in peekperf 111
using MPI profiling in the plug-in 64
viewing MPI traces 47
using the peekperf GUI 95
viewing OpenMP performance data
using OpenMP profiling in the plug-in 69
viewing OpenMP profiling performance data
using OpenMP profiling in peekperf 119
viewing options 153
viewing performance data 42
using the peekperf GUI 91
viewing the disassembler code 185

W

what are derived metrics 204
who should read this information xi

X

X Windows performance profiler (Xprof) 133
customizing Xprof resources 190
x86 Ivy Bridgederived metrics description 322
x86 Nehalem and Westmere
derived metrics description 317
x86 Sandy Bridge
derived metrics description 318
Xprof
customizing resources 190
display status field 150
hidden menus 149
looking at your code 182
main window 148, 149
specifying profile data files 144
specifying the binary executable file 143
using 133
Xprof command 256
Xprof display
understanding 147
Xprof flags
specifying from the command line 136
specifying from the GUI 145
Xprof GUI
starting 135
Xprof main window
controlling the appearance of 191
XWindows performance profiler (Xprof)
basic cluster data 169
before you begin 133
changing the search sequence 147
clustering functions 162
clustering libraries 162
comparing Xprof and the gprof command 134
compiling applications to be profiled 134
controlling fonts 190
controlling how the display is updated 152
controlling the appearance of the Xprof main window 191
controlling the graphic style of the function call tree 153
controlling the orientation of the function call tree 153
controlling the representation of the function call tree 154
controlling variables related to the file menu 192
controlling variables related to the filter menu 196
controlling variables related to the screen dump option 193
controlling variables related to the view menu 195
display status field 150
displaying the entire function call tree 156
excluding and including specific objects 156
filtering by function characteristics 157
filtering shared library functions 156
filtering what you see 155
getting detailed data from reports 171
hidden menus 149
how calls between functions are depicted 152
how functions are represented 150
how library clusters are represented 152
including and excluding parent and child functions 160
loading a configuration file 182
loading files from the Xprof GUI 140
locating and displaying ancestor functions 167

XWindows performance profiler (Xprof) *(continued)*
locating and displaying child functions 166
locating and displaying descendant functions 167
locating and displaying functions on a cycle 167
locating and displaying parent functions 166
locating specific objects in the function call tree 165
looking at your code 182
main menus 149
main window 148
obtaining basic data 167
obtaining performance data for your application 167
other viewing options 153
requirements and limitations 133
restoring the status of the function call tree 155
saving a configuration file 181
saving a single report 180
saving reports to a file 180
flat profile 181
function index 181
graph profile 181
saving screen images of profiled data 187
saving summarized data from multiple profile data files 181
setting alternative paths 146
setting default paths 146
setting the file search sequence 146
setting Xprof resource variables 190
specifying profile data files 144
specifying the binary executable file 143
specifying Xprof flags from the command line 136
specifying Xprof flags from the GUI 145
starting the Xprof GUI 135
unclustering functions 164
understanding basic call data 169
understanding basic function data 167
understanding information boxes 169
understanding the call graph profile report 174
understanding the flat profile report 172
understanding the function call summary report 178
understanding the function index report 177
understanding the library statistics report 179
understanding the Xprof display 147
using the disassembler code window 185
using the function menu statistics report option 170
using the source code window 183
viewing the disassembler code 185
viewing the source code 183



Product Number: 5725-G61
5725-M46
5641-PD1
5641-PD3
5765-PED
5765-PDP

Printed in USA

SC23-6940-01

