



TOTALVIEW USERS GUIDE

VERSION 8.7



TOTALVIEW
TECHNOLOGIES

Copyright © 2007–2009 by TotalView Technologies. All rights reserved

Copyright © 1998–2007 by Etnus LLC. All rights reserved.

Copyright © 1996–1998 by Dolphin Interconnect Solutions, Inc.

Copyright © 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of TotalView Technologies.

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

TotalView Technologies has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by TotalView Technologies. TotalView Technologies assumes no responsibility for any errors that appear in this document.

TotalView and TotalView Technologies are registered trademarks of TotalView Technologies.

TotalView uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at:

ftp://ftp.totalviewtech.com/support/toolworks/Microline_totalview.tar.Z.

All other brand names are the trademarks of their respective holders.

Book Overview



part I - Introduction

1	Getting Started with TotalView	3
2	About Threads, Processes, and Groups.....	15

part II - Setting Up

3	Getting Started with Remote Display Client.....	35
4	Setting Up a Debugging Session	51
5	Setting Up Remote Debugging Sessions	81
6	Setting Up MPI Debugging Sessions	97
7	Setting Up Parallel Debugging Sessions	131

part III - Using the GUI

8	Using TotalView Windows.....	165
9	Visualizing Programs and Data	179

part IV - Using the CLI

10	Using the CLI	199
11	Seeing the CLI at Work	213

part V - Debugging

12	Debugging Programs	223
13	Using Groups, Processes, and Threads	251
14	Examining and Changing Data.....	279
15	Examining Arrays	333
16	Setting Action Points.....	349
17	Evaluating Expressions	381
	Glossary	397
	Index	411

Contents



About This Book

TotalView Family Differences	xxiii
How to Use This Book	xxv
Using the CLI	xxvi
Audience	xxvi
Conventions	xxvii
TotalView Documentation	xxviii
Contacting Us	xxix

part I - Introduction

1 Getting Started with TotalView

Getting Started	3
Starting TotalView	4
What About Print Statements?	5
Examining Data	6
Examining Arrays	8
Seeing Groups of Variables	8
Setting Watchpoints	10
Debugging Multi-process and Multi-threaded Programs	10
Program Using Almost Any Execution Model	11
Supporting Multi-process and Multi-threaded Programs	11
Using Groups and Barriers	12
Memory Debugging	13
Introducing the CLI	14
What's Next	14

2 About Threads, Processes, and Groups

A Couple of Processes	15
Threads	18
Complicated Programming Models	19
Types of Threads	20
Organizing Chaos	22
Creating Groups	26

Simplifying What You're Debugging	30
---	----

part II - Setting Up

3 Getting Started with Remote Display Client

Using Remote Display	35
Installing the Client	36
Installing on Linux	36
Installing on Microsoft Windows	37
Sample Session	37
Naming Intermediate Hosts	39
Submitting a Job to a Batch Queuing System	40
Setting Up Your Systems and Security	40
Session Profile Management	41
Sharing Profiles	42
Remote Display Server and Viewer	42
Batch Scripts	43
tv_PBS.csh Script	43
tv_LoadLeveler.csh Script	44
Remote Display Commands	44
Session Profiles Area	45
Remote Host and Routing Area	45
Product Area	46
Using the Submit to Batch Queuing System Entries	47
File > Profile > Save	48
File > Profile > Delete	49
File > Profile > Import	49
File > Profile > Export	49
File > Exit	49

4 Setting Up a Debugging Session

Compiling Programs	51
Using File Extensions	52
Starting TotalView	53
Initializing TotalView	56
Exiting from TotalView	57
Loading Programs	58
Loading Programs Using the GUI	59
Loading Programs Using the CLI	61
Attaching to Processes	61
Detaching from Processes	62
Examining Core Files	63
Setting Command-line Arguments and Environment Variables	64
Altering Standard I/O	65
Viewing Process and Thread States	65
Seeing Attached Process States	67
Seeing Unattached Process States	67
Handling Signals	68
Setting Search Paths	70
Setting Startup Parameters	73
Setting Preferences	73

Setting Preferences, Options, and X Resources	78
---	----

5 Setting Up Remote Debugging Sessions

Setting Up and Starting the TotalView Server	81
Setting Single-Process Server Launch Options	83
Setting Bulk Launch Window Options	84
Starting the TotalView Server Manually	86
Using the Single-Process Server Launch Command	87
Bulk Server Launch Setting on an SGI Computers	88
Setting Bulk Server Launch on an HP Alpha Computer	89
Setting Bulk Server Launch on a Cray XT Series Computer	90
Setting Bulk Server Launch on an IBM RS/6000 AIX Computer	90
Disabling Autolaunch	91
Changing the Remote Shell Command	91
Changing Arguments	92
Autolaunching Sequence	92
Debugging Over a Serial Line	94
Starting the TotalView Debugger Server	94
Using the New Program Window	95

6 Setting Up MPI Debugging Sessions

Debugging MPI Programs	98
Starting MPI Programs	98
Starting MPI Programs Using File > New Program	98
Debugging MPICH Applications	100
Starting TotalView on an MPICH Job	100
Attaching to an MPICH Job	102
Using MPICH P4 procgroup Files	103
Debugging MPICH2 Applications	104
Downloading and Configuring MPICH2	104
Starting the mpd Daemon	104
Starting TotalView Debugging on an MPICH2 Job	105
Starting MPI Issues	105
MPI Rank Display	106
Displaying the Message Queue Graph Window	107
Displaying the Message Queue	109
About the Message Queue Display	109
Using Message Operations	110
Diving on MPI Processes	110
Diving on MPI Buffers	111
About Pending Receive Operations	111
About Unexpected Messages	111
About Pending Send Operations	112
Debugging Cray MPI Applications	112
Debugging HP Tru64 Alpha MPI Applications	112
Starting TotalView on an HP Alpha MPI Job	112
Attaching to an HP Alpha MPI Job	113
Debugging HP MPI Applications	113
Starting TotalView on an HP MPI Job	114
Attaching to an HP MPI Job	114
Debugging IBM MPI Parallel Environment (PE) Applications	115

Preparing to Debug a PE Application	115
Using Switch-Based Communications	115
Performing a Remote Login	116
Setting Timeouts.....	116
Starting TotalView on a PE Program	116
Setting Breakpoints	116
Starting Parallel Tasks	117
Attaching to a PE Job	117
Attaching from a Node Running poe	117
Attaching from a Node Not Running poe	118
Debugging IBM Blue Gene Applications	118
Debugging LAM/MPI Applications	119
Debugging QSW RMS Applications	120
Starting TotalView on an RMS Job	120
Attaching to an RMS Job.....	121
Debugging SiCortex MPI Applications	121
Debugging SGI MPI Applications	121
Starting TotalView on an SGI MPI Job	122
Attaching to an SGI MPI Job	122
Debugging Sun MPI Applications	123
Attaching to a Sun MPI Job	123
Debugging Parallel Applications Tips	124
Attaching to Processes	124
Parallel Debugging Tips	127
MPICH Debugging Tips	129
IBM PE Debugging Tips	129

7

Setting Up Parallel Debugging Sessions

Debugging OpenMP Applications	132
Debugging OpenMP Programs	132
About TotalView OpenMP Features	133
About OpenMP Platform Differences	133
Viewing OpenMP Private and Shared Variables	134
Viewing OpenMP THREADPRIVATE Common Blocks	136
Viewing the OpenMP Stack Parent Token Line	137
Using SLURM	137
Debugging IBM Cell	138
The PPU	140
The SPU	140
Cell Programming	140
PPU and SPU Executable Organization	140
PPU and SPU Executable Naming	141
Thread IDs	141
Breakpoints	141
Registers, Unions, and Casting	143
Debugging Cray XT Applications	144
Cray XT Catamount	144
Configuring TotalView	145
Using TotalView	146
Cray XT CNL	146
Debugging SiCortex Applications	147

Installation Notes	147
Using TotalView on SiCortex	147
MPI Debugging	148
Debugging Global Arrays Applications	148
Debugging PVM (Parallel Virtual Machine) and DPVM Applications	151
Supporting Multiple Sessions	151
Setting Up ORNL PVM Debugging	152
Starting an ORNL PVM Session	152
Starting a DPVM Session	153
Automatically Acquiring PVM/DPVM Processes	154
Attaching to PVM/DPVM Tasks	155
About Reserved Message Tags	156
Cleaning Up Processes	156
Debugging Shared Memory (SHMEM) Code	156
Debugging UPC Programs	158
Invoking TotalView	158
Viewing Shared Objects	158
Displaying Pointer to Shared Variables	160

part III - Using the GUI

8 Using TotalView Windows

Using Mouse Buttons	165
Using the Root Window	166
Using the Process Window	169
Viewing the Assembler Version of Your Code	171
Diving into Objects	173
Resizing and Positioning Windows and Dialog Boxes	175
Editing Text	176
Saving the Contents of Windows	177

9 Visualizing Programs and Data

Displaying Call Graphs	179
Visualizing Array Data	181
Command Summary	181
How the Visualizer Works	182
Viewing Data Types in the Visualizer	182
Viewing Data	183
Visualizing Data Manually	183
Using the Visualizer	184
Using Dataset Window Commands	185
Using View Window Commands	185
Using the Graph Window	186
Displaying Graph Views	186
Using the Surface Window	187
Displaying Surface Views	189
Manipulating Surface Data	191
Visualizing Data Programmatically	192
Launching the Visualizer from the Command Line	193
Configuring TotalView to Launch the Visualizer	193
Setting the Visualizer Launch Command	194

part IV - Using the CLI

10 Using the CLI

About the Tcl and the CLI	199
About The CLI and TotalView	200
Using the CLI Interface	201
Starting the CLI	201
Startup Example	202
Starting Your Program	203
About CLI Output	205
'more' Processing	206
Using Command Arguments	206
Using Namespaces	207
About the CLI Prompt	207
Using Built-in and Group Aliases	208
How Parallelism Affects Behavior	209
Types of IDs	209
Controlling Program Execution	210
Advancing Program Execution	210
Using Action Points	211

11 Seeing the CLI at Work

Setting the CLI EXECUTABLE_PATH Variable	214
Initializing an Array Slice	215
Printing an Array Slice	215
Writing an Array Variable to a File	217
Automatically Setting Breakpoints	218

part V - Debugging

12 Debugging Programs

Searching and Looking For Program Elements	223
Searching for Text	224
Looking for Functions and Variables	224
Finding the Source Code for Functions	225
Resolving Ambiguous Names	226
Finding the Source Code for Files	227
Resetting the Stack Frame	227
Editing Source Text	227
Manipulating Processes and Threads	228
Using the Toolbar to Select a Target	228
Stopping Processes and Threads	229
Using the Processes/Ranks Tab	229
Using the Threads Tab	230
Updating Process Information	231
Holding and Releasing Processes and Threads	231
Using Barrier Points	233
Holding Problems	234
Examining Groups	235
Placing Processes in Groups	236

Starting Processes and Threads	236
Creating a Process Without Starting It	237
Creating a Process by Single-Stepping	237
Stepping and Setting Breakpoints	237
Using Stepping Commands	239
Stepping into Function Calls	240
Stepping Over Function Calls	240
Executing to a Selected Line	241
Executing Out of a Function	241
Continuing with a Specific Signal	242
Killing (Deleting) Programs	243
Restarting Programs	243
Checkpointing	244
Fine-Tuning Shared Library Use	245
Preloading Shared Libraries	245
Controlling Which Symbols TotalView Reads	246
Specifying Which Libraries are Read	246
Reading Excluded Information	248
Setting the Program Counter	248
Interpreting the Status and Control Registers	250

13 Using Groups, Processes, and Threads

Defining the GOI, POI, and TOI	251
Setting a Breakpoint	252
Stepping (Part I)	253
Understanding Group Widths	254
Understanding Process Width	254
Understanding Thread Width	255
Using Run To and duntil Commands	255
Using P/T Set Controls	256
Setting Process and Thread Focus	257
Understanding Process/Thread Sets	257
Specifying Arenas	259
Specifying Processes and Threads	259
Defining the Thread of Interest (TOI)	259
About Process and Thread Widths	260
Specifier Examples	261
Setting Group Focus	262
Specifying Groups in P/T Sets	263
About Arena Specifier Combinations	264
'All' Does Not Always Mean 'All'	267
Setting Groups	268
Using the g Specifier: An Extended Example	269
Merging Focuses	271
Naming Incomplete Arenas	272
Naming Lists with Inconsistent Widths	273
Stepping (Part II): Examples	273
Using P/T Set Operators	275
Creating Custom Groups	276

14 Examining and Changing Data

Changing How Data is Displayed	279
Displaying STL Variables	280
Changing Size and Precision	282
Displaying Variables	283
Displaying Program Variables	284
Controlling the Information Being Displayed	285
Seeing Value Changes	285
Seeing Structure Information	287
Displaying Variables in the Current Block	287
Viewing Variables in Different Scopes as Program Executes	288
Scoping Issues	289
Freezing Variable Window Data	289
Locking the Address	289
Browsing for Variables	291
Displaying Local Variables and Registers	292
Dereferencing Variables Automatically	293
Examining Memory	294
Displaying Areas of Memory	296
Changing Types to Display Machine Instructions	297
Displaying Machine Instructions	297
Rebinding the Variable Window	298
Closing Variable Windows	298
Diving in Variable Windows	298
Displaying an Array of Structure's Elements	300
Changing What the Variable Window Displays	301
Viewing a List of Variables	303
Entering Variables and Expressions	303
Seeing Variable Value Changes in the Expression List Window	305
Entering Expressions into the Expression Column	305
Using the Expression List with Multi-process/Multi-threaded Programs	307
Reevaluating, Reopening, Rebinding, and Restarting	307
Seeing More Information	308
Sorting, Reordering, and Editing	309
Changing the Values of Variables	310
Changing a Variable's Data Type	311
Displaying C and C++ Data Types	312
Viewing Pointers to Arrays	314
Viewing Arrays	314
Viewing <code>typedef</code> Types	315
Viewing Structures	315
Viewing Unions	315
Casting Using the Built-In Types	315
Viewing Character Arrays (\$String Data Type)	318
Viewing Wide Character Arrays (\$wchar Data Types)	318
Viewing Areas of Memory (\$void Data Type)	319
Viewing Instructions (\$code Data Type)	320
Viewing Opaque Data	320
Type-Casting Examples	320
Displaying Declared Arrays	321

Displaying Allocated Arrays	321
Displaying the argv Array	321
Changing the Address of Variables	321
Displaying C++ Types	322
Viewing Classes	322
Displaying Fortran Types	324
Displaying Fortran Common Blocks	324
Displaying Fortran Module Data	324
Debugging Fortran 90 Modules	326
Viewing Fortran 90 User-Defined Types	327
Viewing Fortran 90 Deferred Shape Array Types	327
Viewing Fortran 90 Pointer Types	328
Displaying Fortran Parameters	329
Displaying Thread Objects	329
Scoping and Symbol Names	330
Qualifying Symbol Names	331

15 Examining Arrays

Examining and Analyzing Arrays	333
Displaying Array Slices	334
Using Slices and Strides	334
Using Slices in the Lookup Variable Command	336
Array Slices and Array Sections	336
Filtering Array Data Overview	337
Filtering by Comparison	338
Filtering for IEEE Values	339
Filtering a Range of Values	341
Creating Array Filter Expressions	341
Using Filter Comparisons	342
Sorting Array Data	342
Obtaining Array Statistics	343
Displaying a Variable in all Processes or Threads	345
Diving on a "Show Across" Pointer	346
Editing a "Show Across" Variable	346
Visualizing Array Data	347
Visualizing a "Show Across" Variable Window	347

16 Setting Action Points

About Action Points	349
Setting Breakpoints and Barriers	351
Setting Source-Level Breakpoints	352
Choosing Source Lines	352
Setting Breakpoints at Locations	353
Ambiguous Functions and Pending Breakpoints	354
Displaying and Controlling Action Points	355
Disabling Action Points	356
Deleting Action Points	356
Enabling Action Points	356
Suppressing Action Points	356
Setting Breakpoints on Classes and Virtual and Overloaded Functions	357
Setting Machine-Level Breakpoints	358

Setting Breakpoints for Multiple Processes	359
Setting Breakpoints When Using the fork()/execve() Functions	361
Debugging Processes That Call the fork() Function	361
Debugging Processes that Call the execve() Function.....	361
Example: Multi-process Breakpoint	362
Setting Barrier Points	362
About Barrier Breakpoint States	363
Setting a Barrier Breakpoint	363
Creating a Satisfaction Set	364
Hitting a Barrier Point	365
Releasing Processes from Barrier Points	365
Deleting a Barrier Point	365
Changing Settings and Disabling a Barrier Point	365
Defining Eval Points and Conditional Breakpoints	366
Setting Eval Points	367
Creating Conditional Breakpoint Examples	368
Patching Programs	368
Branching Around Code	369
Adding a Function Call	369
Correcting Code.....	369
About Interpreted and Compiled Expressions	370
About Interpreted Expressions	370
About Compiled Expressions	370
Allocating Patch Space for Compiled Expressions	371
Allocating Dynamic Patch Space	371
Allocating Static Patch Space	372
Using Watchpoints	373
Using Watchpoints on Different Architectures	374
Creating Watchpoints	375
Displaying Watchpoints	376
Watching Memory	377
Triggering Watchpoints	377
Using Multiple Watchpoints	377
Copying Previous Data Values	378
Using Conditional Watchpoints	378
Saving Action Points to a File	380

17 Evaluating Expressions

Why is There an Expression System?	381
Calling Functions: Problems and Issues	383
Expressions in Eval Points and the Evaluate Window	383
Using C++	384
Using Programming Language Elements	385
Using C and C++	385
Using Fortran	387
Fortran Statements	387
Fortran Intrinsic	388
..... Using the Evaluate Window	389
Writing Assembler Code	390
Using Built-in Variables and Statements	394
Using TotalView Variables	394

Using Built-In Statements	395
Glossary	397
INDEX	411

Figures



Chapter 1: Getting Started with TotalView

Figure 1.	The Process Window	4
Figure 2.	Action Point Properties Dialog Box	5
Figure 3.	Setting Conditions	6
Figure 4.	Patching Using an Eval Point	7
Figure 5.	Diving on a Structure and an Array	8
Figure 6.	Slicing and Filtering Arrays	9
Figure 7.	Visualizing an Array	9
Figure 8.	Tools > Expression List Window	10
Figure 9.	The Root Window	11
Figure 10.	Viewing Across Processes	12
Figure 11.	A Message Queue Graph	12
Figure 12.	Toolbar With Pulldown	13
Figure 13.	Process Tab	13

Chapter 2: About Threads, Processes, and Groups

Figure 14.	A Uniprocessor	16
Figure 15.	A Program and Daemons	16
Figure 16.	Mail Using Daemons to Communicate	16
Figure 17.	Two Computers Working on One Problem	17
Figure 18.	Threads	18
Figure 19.	Four-Processor Computer	19
Figure 20.	Four Processors on a Network	20
Figure 21.	Threads (again)	20
Figure 22.	User and Service Threads	21
Figure 23.	User, Service, and Manager Threads	22
Figure 24.	Five-Processes: Their Control and Share Groups	23
Figure 25.	Five Processes: Adding Workers and Lockstep Groups	24
Figure 26.	Five Processes and Their Groups on Two Computers	25
Figure 27.	Step 1: A Program Starts	26
Figure 28.	Step 1: A Program Starts	26
Figure 29.	Step 3: Creating a Process using exec()	27
Figure 30.	Step 5: Creating a Second Version	28
Figure 31.	Step 6: Creating a Remote Process	28

Figure 32.	Step 7: Creating a Thread	29
Figure 33.	Step 8: Hitting a Breakpoint	29
Figure 34.	Step 9: Stepping the Lockstep Group	30

Chapter 3: Getting Started with Remote Display Client

Figure 35.	Remote Display Components	36
Figure 36.	Remote Display Client Setup	37
Figure 37.	Remote Display Client Window	38
Figure 38.	Asking for Password	39
Figure 39.	Access By Options	40
Figure 40.	Remote Display Window: Showing Batch Options	41
Figure 41.	Session Profiles	42
Figure 42.	Local Data in a Stack Frame	43
Figure 43.	Access By Options	45
Figure 44.	Remote Host Information Area	47
Figure 45.	Choosing a Batch Queuing System	47
Figure 46.	Remote Display Client: Showing Batch Options	48
Figure 47.	Saving a Profile	49
Figure 48.	Deleting a Profile	49
Figure 49.	Exit Dialog Box	49

Chapter 4: Setting Up a Debugging Session

Figure 50.	File > New Program Dialog Box	54
Figure 51.	Startup and Initialization Sequence	57
Figure 52.	File > Exit Dialog Box	58
Figure 53.	Start a New Process	59
Figure 54.	Attach to an Existing Process	60
Figure 55.	Open a Core File	60
Figure 56.	Attaching to an existing process	61
Figure 57.	Thread > Continuation Signal Dialog Box	62
Figure 58.	Open a Core File	63
Figure 59.	Setting Command-Line Options and Environment Variables	64
Figure 60.	Resetting Standard I/O	65
Figure 61.	Root Window Showing Process and Thread Status	66
Figure 62.	Process and Thread Labels in the Process Window	66
Figure 63.	File > Signals Dialog Box	69
Figure 64.	File > Search Path Dialog Box	71
Figure 65.	Select Directory Dialog Box	72
Figure 66.	File > Preferences Dialog Box: Options Page	74
Figure 67.	File > Preferences Dialog Box: Action Points Page	74
Figure 68.	File > Preferences Dialog Box: Launch Strings Page	75
Figure 69.	File > Preferences Dialog Box: Bulk Launch Page	75
Figure 70.	File > Preferences Dialog Box: Dynamic Libraries Page	76
Figure 71.	File > Preferences Dialog Box: Parallel Page	76
Figure 72.	File > Preferences Dialog Box: Fonts Page	77
Figure 73.	File > Preferences Dialog Box: Formatting Page	77
Figure 74.	File > Preferences Dialog Box: Pointer Dive Page	78

Chapter 5: Setting Up Remote Debugging Sessions

Figure 75.	File > Preferences: Launch Strings Page	83
------------	---	----

Figure 76.	File > Preferences: Bulk Launch Page	84
Figure 77.	Manual Launching of Debugger Server	87
Figure 78.	Launching tvdsvr	93
Figure 79.	Multiple tvdsvr Processes	93
Figure 80.	Debugging Session Over a Serial Line	94
Figure 81.	Adding New Host	95

Chapter 6: Setting Up MPI Debugging Sessions

Figure 82.	File > New Program Dialog Box	99
Figure 83.	File > New Program Dialog Box: Parallel Tab	99
Figure 84.	File > New Program: Attach to an Existing Process	102
Figure 85.	Ranks Tab	106
Figure 86.	Tools > Message Queue Graph Window	107
Figure 87.	Tools > Message Queue Graph Options Window	108
Figure 88.	Tools > Message Queue Graph Options. Filter Tab	108
Figure 89.	Message Queue Window	110
Figure 90.	Message Queue Window Showing Pending Receive Operation	111
Figure 91.	Group > Attach Subset Dialog Box	124
Figure 92.	Stop Before Going Parallel Question Box	125
Figure 93.	File > Preferences: Parallel Page	126

Chapter 7: Setting Up Parallel Debugging Sessions

Figure 94.	Sample OpenMP Debugging Session	134
Figure 95.	OpenMP Shared Variable	135
Figure 96.	OpenMP THREADPRIVATE Common Block Variables	137
Figure 97.	OpenMP Stack Parent Token Line	137
Figure 98.	Cell Architecture	139
Figure 99.	A Cell Process	139
Figure 100.	Root Window for a Cell Program	141
Figure 101.	Action Point Properties Dialog Box	142
Figure 102.	Stop to Set Breakpoints Question	142
Figure 103.	Register Union	143
Figure 104.	Question Window for Global Arrays Program	150
Figure 105.	Tools > Global Arrays Window	150
Figure 106.	PVM Tasks and Configuration Window	155
Figure 107.	SHMEM Sample Session	157
Figure 108.	A Sliced UPC Array	159
Figure 109.	UPC Variable Window Showing Nodes	159
Figure 110.	A Pointer to a Shared Variable	160
Figure 111.	Pointer to a Shared Variable	161

Chapter 8: Using TotalView Windows

Figure 112.	Root Window	167
Figure 113.	Root Window Showing Two Host Computers	167
Figure 114.	Two Views of the Root Window	168
Figure 115.	Sorted and Aggregated Root Window	169
Figure 116.	A Process Window	170
Figure 117.	Line Numbers with Stop Icon and PC Arrow	171
Figure 118.	Address Only (Absolute Addresses)	172
Figure 119.	Assembly Only (Symbolic Addresses)	173

Figure 120.	Both Source and Assembler (Symbolic Addresses)	173
Figure 121.	Nested Dive	174
Figure 122.	Backward and Forward Buttons	175
Figure 123.	Resizing (and Its Consequences)	176
Figure 124.	File > Save Pane Dialog Box	177

Chapter 9: Visualizing Programs and Data

Figure 125.	Tools > Call Graph Dialog Box	180
Figure 126.	TotalView Visualizer Relationships	182
Figure 127.	A Three-Dimensional Array Sliced into Two Dimensions	183
Figure 128.	Sample Visualizer Windows	184
Figure 129.	Graph and Surface Visualizer Windows	185
Figure 130.	Visualizer Graph View Window	187
Figure 131.	Graph Options Dialog Box	187
Figure 132.	Sine wave Displayed in Three Ways	188
Figure 133.	A Surface View	188
Figure 134.	A Surface View of a Sine Wave	189
Figure 135.	Surface Options Dialog Box	189
Figure 136.	Four Surface Views	190
Figure 137.	File > Preferences Launch Strings Page	194

Chapter 10: Using the CLI

Figure 138.	The CLI, GUI and TotalView	200
Figure 139.	CLI xterm Window	202

Chapter 11: Seeing the CLI at Work

Chapter 12: Debugging Programs

Figure 140.	Edit > Find Dialog Box	224
Figure 141.	View > Lookup Variable Dialog Box	225
Figure 142.	Ambiguous Function Dialog Box	225
Figure 143.	View > Lookup Function Dialog Box	225
Figure 144.	Undive/Dive Controls	226
Figure 145.	Ambiguous Function Dialog Box	226
Figure 146.	View > Lookup Function Dialog Box	227
Figure 147.	The Toolbar	228
Figure 148.	The Processes Tab	230
Figure 149.	The Processes Tab: Showing Group Selection	230
Figure 150.	The Threads Tab	231
Figure 151.	Running To Barriers	234
Figure 152.	Control and Share Groups Example	236
Figure 153.	Action Point and Addresses Dialog Boxes	238
Figure 154.	Ambiguous Address Dialog Box	238
Figure 155.	Stepping Illustrated	239
Figure 156.	Thread > Continuation Signal Dialog Box	242
Figure 157.	Create Checkpoint and Restart Checkpoint Dialog Boxes	244
Figure 158.	Tools > Debugger Loaded Libraries Dialog Box	245
Figure 159.	Stopping to Set a Breakpoint Question Box	246
Figure 160.	File > Preferences: Dynamic Libraries Page	247
Figure 161.	Load All Symbols in Stack Context menu	248

Chapter 13: Using Groups, Processes, and Threads

Figure 162.	The P/T Set Control in the Process Window	256
Figure 163.	Width Specifiers	261
Figure 164.	Group > Custom Groups Dialog Box	277

Chapter 14: Examining and Changing Data

Figure 165.	An Untransformed Map	280
Figure 166.	A Transformed Map	280
Figure 167.	List and Vector Transformations	281
Figure 168.	File > Preferences Formatting Page	282
Figure 169.	A Tooltip	283
Figure 170.	Variable Window for a Global Variable	284
Figure 171.	Variable Window: Using More and Less	286
Figure 172.	Variable Window With "Change" Highlighting	286
Figure 173.	Variable Window Showing Last Value Column	287
Figure 174.	Displaying Scoped Variables	288
Figure 175.	Variable Window Showing Frozen State	290
Figure 176.	Locked and Unlocked Variable Windows	290
Figure 177.	Program Browser and Variable Windows (Part 1)	291
Figure 178.	Program Browser and Variable Window (Part 2)	292
Figure 179.	Diving on Local Variables and Registers	293
Figure 180.	File > Preferences Pointer Dive Page	294
Figure 181.	View > Examine Format > Structured Display	295
Figure 182.	View > Examine Format > Raw Display	295
Figure 183.	Variable Window for an Area of Memory	296
Figure 184.	Variable Window with Machine Instructions	297
Figure 185.	Undive/Redive Buttons	299
Figure 186.	Nested Dives	299
Figure 187.	Displaying a Fortran Structure	300
Figure 188.	Displaying C Structures and Arrays	301
Figure 189.	Dive in All	302
Figure 190.	The Tools > Expression List Window	303
Figure 191.	A Context Menu	304
Figure 192.	Expression List Window Context Menu	304
Figure 193.	Expression List Window With "Change" Highlighting	305
Figure 194.	Variable Window Showing Last Value Column	305
Figure 195.	The Tools > Expression List Window	306
Figure 196.	Using Methods in the Tools > Expression List Window	306
Figure 197.	Using Functions in the Tools > Expression List Window	307
Figure 198.	The Tools > Expression List Window Showing Column Selector	308
Figure 199.	Using an Expression to Change a Value	311
Figure 200.	Three Casts	313
Figure 201.	Displaying a Union	316
Figure 202.	Displaying wchar_t Data	319
Figure 203.	Editing the argv Argument	322
Figure 204.	Displaying C++ Classes That Use Inheritance	323
Figure 205.	Diving on a Common Block List in the Stack Frame Pane	325
Figure 206.	Fortran Modules Window	326
Figure 207.	Fortran 90 User-Defined Type	327
Figure 208.	Fortran 90 Pointer Value	328

Figure 209. Thread Objects Page on an IBM AIX Computer	330
Figure 210. Variable Window: Showing Variable Properties	331

Chapter 15: Examining Arrays

Figure 211. Stride Displaying the Four Corners of an Array	335
Figure 212. Fortran Array with Inverse Order and Limited Extent	336
Figure 213. An Array Slice and an Array Section	337
Figure 214. Array Data Filtering by Comparison	339
Figure 215. Array Data Filtering for IEEE Values	340
Figure 216. Array Data Filtering by Range of Values	341
Figure 217. Sorted Variable Window	342
Figure 218. Array Statistics Window	343
Figure 219. Viewing Across Threads	345
Figure 220. Viewing across an Array of Structures	346

Chapter 16: Setting Action Points

Figure 221. Action Point Symbols	350
Figure 222. Action Point Tab	351
Figure 223. Setting Breakpoints on Multiple Similar Addresses	352
Figure 224. Setting Breakpoints on Multiple Similar Addresses and on Processes	353
Figure 225. Action Point > At Location Dialog Box	353
Figure 226. Pending Breakpoints	354
Figure 227. Ambiguous Function Dialog Box	355
Figure 228. Action Point > Properties Dialog Box	355
Figure 229. Action Point > At Location Dialog Box	357
Figure 230. Action Point > Properties: Selecting	358
Figure 231. Breakpoint at Assembler Instruction	358
Figure 232. PC Arrow Over a Stop Icon	359
Figure 233. Action Point > Properties Dialog Box	359
Figure 234. File > Preferences: Action Points Page	360
Figure 235. Action Point > Properties Dialog Box	364
Figure 236. Stopped Execution of Compiled Expressions	371
Figure 237. Tools > Watchpoint Dialog Boxes	376

Chapter 17: Evaluating Expressions

Figure 238. Expression List Window: Accessing Array Elements	382
Figure 239. Displaying the Value of the Last Statement	384
Figure 240. Expression List Window: Showing Overloads	385
Figure 241. Class Casting	386
Figure 242. Tools > Evaluate Dialog Box	390
Figure 243. Waiting to Complete Message Box	390
Figure 244. Evaluating Information in Multiple Processes	391
Figure 245. Using Assembler Expressions	391

About This Book



This book describes how to use the TotalView® debugger, a source- and machine-level debugger for multi-process, multi-threaded programs. The information in this book assumes that you are familiar with programming languages, a UNIX or Linux operating system, and the processor architecture of the system on which you are running TotalView and your program.

This user guide combines information for all TotalView debuggers, whether they run within a Graphic User Interface (GUI) or in an xterm-like window where you must type commands. That version of TotalView is called the Command Line Interface (CLI). This book emphasizes the GUI interface, as it is easier to use. After you see what you can do using the GUI, you will know what you can do using the CLI.

TotalView doesn't change much from platform to platform. Differences between platforms are mentioned.

TotalView Family Differences

This manual describes TotalView Enterprise, TotalView Team, and TotalView Individual. Each of these allows you to use the CLI debugger as well. In all cases, TotalView Enterprise and TotalView Team have the same features. They differ in the way they are licensed. However, TotalView Individual is slightly different.



The most fundamental differences between TotalView Team and TotalView Enterprise are the way resources are shared and used. When you purchase TotalView Team, you are purchasing "tokens." These tokens represent debugging capabilities. For example, if you

TotalView Family Differences

have 64 tokens available, 64 programmers could be active, each debugging a one-process job; or 2 programmers, each debugging a 32 process job. In contrast, a TotalView Enterprise license is based on the number of users and the number of licensed processors. You'll find more precise information on our web site.

The major differences are:

TotalView Team & Enterprise	TotalView Individual	Comment
Execute on any licensed computer of the same architecture	Node locked.	You can execute TotalView Individual only on the computer you install it on.
Number of users is determined by license	Only one user	The TotalView Enterprise license limits the number of users. TotalView Team does not.
Number of processes limited by license. No limit on threads	No more than 16 processes and threads.	
Your license determines the number of processors upon which your program can run.	A program can execute on no more than two cores or four cores, depending upon license.	TotalView Enterprise licenses the full capabilities of all machines upon which it runs even if you are not using all of your machine's processors. TotalView Team creates a pool of tokens that you can use. See our web site for more information.
Processes can execute on any computer in the same network.	Remote processes are not allowed.	Processes must execute on the installed computer.
Remote X Server connections allowed.	No remote X Server connections are allowed.	Programmers cannot remotely log into a computer and then execute TotalView Individual.
Memory debugging is bundled. It is only bundled with Team Plus.	No memory debugging	

How to Use This Book

The information in this book is presented in five parts:

■ I: Introduction

This part contains an overview of some TotalView features and an introduction to the TotalView process/thread model. These sections give you a feel for what TotalView can do.

■ II: Setting Up

This part describes how to configure TotalView. No one will ever use all of the information in this part.

Chapter 3 contains general information. Chapters 4 through 6 tell you how to get your programs running under TotalView control. Chapter 4 explains how to get the TotalView Debugger Server (`tvdsvr`) running and how to reconfigure how TotalView launches the `tvdsvr`. In most cases, TotalView default works fine and you won't need this information.

Chapters 5 and 6 look at high performance computing environments such as MPICH, OpenMP, UPC, and the like. Most people never use more than one or two sections from these two chapters. You should go to the table of contents and find what you need instead of just browsing through this information.

■ III: Using the GUI

The chapters in this section describe some of the TotalView windows and how you use them. They also describe tools such as the **Visualizer** and the **Call Graph** that help you analyze what your program is doing.

■ IV: Using the CLI

The chapters in this section explain the basics of using the Command Line Interface (CLI) for debugging. CLI commands are not discussed in this book. You'll find that information in the "*TotalView Reference Guide*".

■ V: Debugging

In many ways, most of what precedes this part of the book is introductory material. So, if TotalView just comes up (and it should) and you understand what debuggers do, you can go directly to this information. This part explains how to examine your program and its data. It also contains information on setting the action points that allow you to stop and monitor your program's execution.

Chapter 12 is a detailed examination of the TotalView group, process, and thread model. Having a better understanding of this model makes it easier to debug multi-process and multi-threaded programs.

Using the CLI

To use the Command Line Interface (CLI), you need to be familiar with and have experience debugging programs with the TotalView GUI. CLI commands are embedded within a Tcl interpreter, so you get better results if you are also familiar with Tcl. If you don't know Tcl, you can still use the CLI, but you lose the ability to program actions that Tcl provides; for example, CLI commands operate on a set of processes and threads. By using Tcl commands, you can save this set and apply this saved set to other commands.

The following books are excellent sources of Tcl information:

- Ousterhout, John K. *Tcl and the Tk Toolkit*. Reading, Mass.: Addison Wesley, 1997.
- Welch, Brent B. *Practical Programming in Tcl & Tk*. Upper Saddle River, N.J.: Prentice Hall PTR, 1999.

There is also a rich set of resources available on the Web. A very good starting point is <http://www.tcltk.tk>.

The fastest way to gain an appreciation of the actions performed by CLI commands is to scan Chapter 1 of the *TotalView Reference Guide*, which contains an overview of CLI commands.

Audience

Many of you are very sophisticated programmers having a tremendous knowledge of programming and its methodologies, and almost all of you have used other debuggers and have developed your own techniques for debugging the programs that you write.

We know you are an expert in your area, whether it be threading, high-performance computing, or client/server interactions, and the like. So, rather than telling you about what you're doing, this book tells you about TotalView.

TotalView is a rather easy-to-use product. Nonetheless, we can't tell you how to use TotalView to solve your problems because your programs are unique and complex, and we can't anticipate what you want to do. So, what you'll find in this book is a discussion of the kinds of operations you can perform. This book, however, is not just a description of dialog boxes

and what you should click on or type. Instead, it tells you how to control your program, see a variable's value, and perform other debugging actions.

Information about what you do with a dialog box or the kinds of data you can type is in the online Help. If you prefer, HTML and PDF versions of this information is available on our Web site. If you have purchased TotalView, you can also post this HTML documentation on your intranet.

Conventions

The following table describes the conventions used in this book:

Convention	Meaning
[]	Brackets are used when describing parts of a command that are optional.
<i>arguments</i>	In a command description, text in italics represents information you type. Elsewhere, italics is used for emphasis.
Dark text	In a command description, dark text represents keywords or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.
Example text	In program listings, this indicates that you are seeing a program or something you'd type in response to a shell or CLI prompt. If this text is in bold, it's indicating that what you're seeing is what you'll be typing. If you're viewing this information online, example text is in color.
	This graphic symbol indicates that the information that follows— <i>which is printed in italics</i> —is a note. This information is an important qualifier to what you just read.
CLI:	The primary emphasis of this book is the GUI. It shows the windows and dialog boxes that you use. This symbol tells you the CLI command you use to do the same thing.

TotalView Documentation

The following table describes TotalView documentation:

Title	Contents	Online Help	HTML	PDF
Evaluating TotalView	Brochure that leads you to basic TotalView features. Contact sales@totalviewtech.com for a free copy.		✓	
TotalView Users Guide	Describes how to use the TotalView GUI and the CLI; this is the most used of all the TotalView books.	✓	✓	✓
TotalView New Features	Describes new features added to TotalView.	✓	✓	✓
Debugging Memory Using TotalView	Is a combined user and reference guide describing how to find your program's memory problems.	✓	✓	✓
TotalView Reference Guide	Contains descriptions of CLI commands, how you run TotalView, and platform-specific information.	✓	✓	✓
TotalView QuickView	Presents what you need to know to get started using TotalView. Contact sales@totalviewtech.com for a free copy.		✓	
CLI Command Summary	A reference card that contains a brief description of all CLI commands and their syntax. Contact sales@totalviewtech.com for a free copy.		✓	
TotalView Commands	Defines all TotalView GUI commands—this is the online Help.	✓	✓	✓
TotalView Installation Guide	Contains the procedures to install TotalView and the FLEXlm license manager.	✓	✓	✓
Platforms and System Requirements	Lists the platforms upon which TotalView runs and the compilers it supports.	✓	✓	✓
Creating Type Transformations	Describes in detail how you can create type transformations. This document is extremely technical.		✓	
TotalView Eclipse Plugin	Tells how you install and use the TotalView Eclipse plugin.		✓	

Contacting Us

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Our Internet email address for support issues is:

support@totalviewtech.com

For documentation issues, the address is:

documentation@totalviewtech.com

Our phone numbers are:

1-800-856-3766 in the United States

(+1) 508-652-7700 worldwide

If you are reporting a problem, please include the following information:

- The *version* of TotalView and the *platform* on which you are running TotalView.
- An *example* that illustrates the problem.
- A *record* of the sequence of events that led to the problem.

Contacting Us

Part I: Introduction



This part of the *TotalView Users Guide* contains two chapters.

Chapter 1: Getting Started with TotalView

Presents an overview of what TotalView is and the ways in which it can help you debug programs. If you haven't used TotalView before, reading this chapter lets you know what TotalView can do for you.

Chapter 2: About Threads, Processes, and Groups

Defines the TotalView model for organizing processes and threads. While most programmers have an intuitive understanding of what their programs are doing, debugging multi-process and multi-threaded programs requires an exact knowledge of what's being done. This chapter begins a two-part look at the TotalView process/thread model. This chapter contains introductory information. Chapter 13: "Using Groups, Processes, and Threads" on page 251 contains information on actually using these concepts.

Getting Started with TotalView



TotalView is a powerful, sophisticated, and programmable tool that lets you debug, analyze, and tune the performance of complex serial, multi-process, and multi-threaded programs.

If you want to jump in and get started quickly, go to our web site at <http://www.totalviewtech.com/Documentation> and select the "Getting Started" item.

This chapter contains the following sections:

- "Getting Started" on page 3
- "Debugging Multi-process and Multi-threaded Programs" on page 10
- "Using Groups and Barriers" on page 12
- "Memory Debugging" on page 13
- "Introducing the CLI" on page 14
- "What's Next" on page 14

Getting Started

The first steps you perform when debugging programs with TotalView are similar to those you perform using other debuggers:

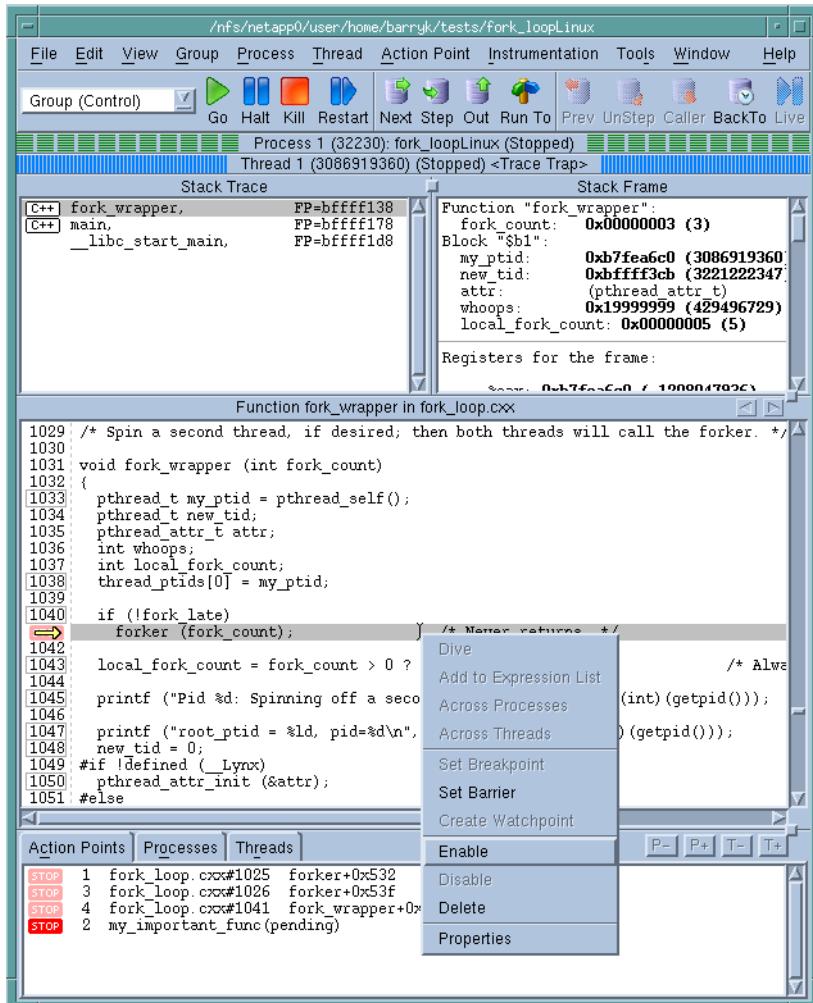
- You use the `-g` option when you compile your program.
- You start your program under TotalView control.
- You set a breakpoint.
- You examine data.

The way you do these things is similar to the way you do things in other debuggers. Where TotalView differs from what you're used to is in its raw power, the breadth of commands available, and its native ability to handle multi-process, multi-threaded programs.

Starting TotalView

After execution begins—by typing something like `totalview programname`—the TotalView Root and Process Windows appear. The window you'll spend the most time using is the Process Window.

Figure 1: The Process Window



You can start program execution in several ways. Perhaps the easiest way is to click the **Step** button in the toolbar. This gets your program started, which means that the initialization performed by the program gets done but no statements are executed.

A second way is to scroll your program to find where you want it to run to, select the line, then click on the **Run To** button in the toolbar. Or you can click on the line number, which tells TotalView to create a breakpoint on that line, and then click the **Go** button in the toolbar.

If your program is large, and usually it will be, you can use **Edit > Find** to locate the line for you. Or, if you want to stop execution when your pro-

gram reaches a subroutine, use **Action Point > At Location** to set a breakpoint on that routine before you click **Go**.

What About Print Statements?

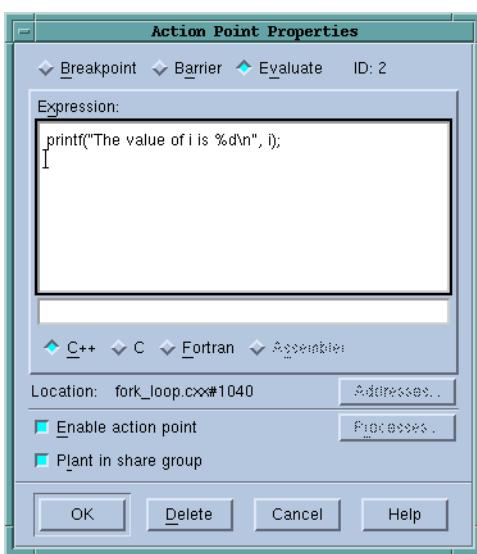
Most programmers learn to debug by using print statements. That is, you insert lots of **printf()** or **PRINT** statements in your code and then inspect what gets written. The problem with this is that every time you want to add a new statement, you need to recompile your program. Even worse, what gets printed is probably not in the right order when running multi-process, multi-threaded programs.

While TotalView is much more sophisticated in displaying information, you can still use **printf()** statements if that's your style, but you'll them in a more sophisticated way, and use them without recompiling your program. You'll do this by adding a breakpoint that prints information. When you open the **Action Point Properties**, which is shown in next figure.



In TotalView, a breakpoint is called an "action point" because TotalView breakpoints are much more powerful than the breakpoints you've used in other debuggers.

Figure 2: Action Point Properties Dialog Box

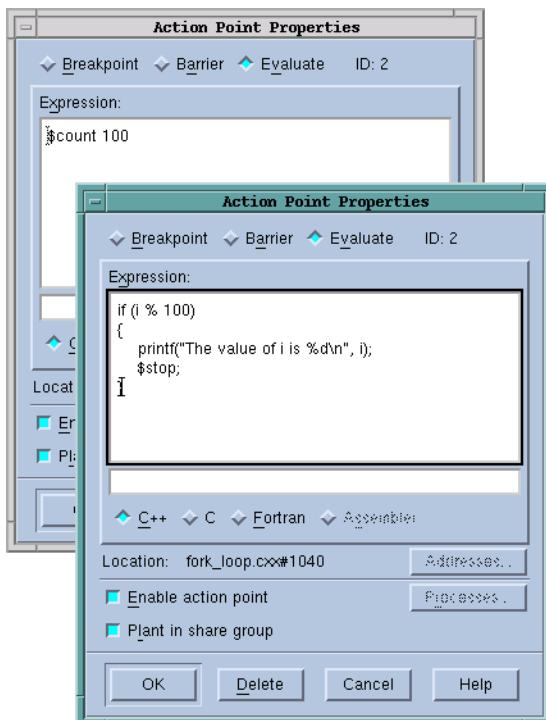


You can add any code you want to a breakpoint. Because there's code associated with this breakpoint, it is called an *eval point*. Here's where TotalView does things a little differently. When your program reaches this eval point, TotalView executes the code you've entered. In this case, TotalView prints the value of *i*.

Eval points do exactly what you tell them to do. In the example in the preceding figure, TotalView lets your program continue execute because you didn't tell it to stop. In other words, you don't have to stop program execution just to see information. You can, of course, tell TotalView to stop.

Figure 3 shows two eval points that stop execution. (One of them does something else as well.)

Figure 3: Setting Conditions



The eval point in the foreground uses programming language statements and a built-in debugger function to stop a loop every 100 iterations. It also prints the value of *i*. In contrast, the eval point in the background just stops the program every 100 times a statement gets executed.

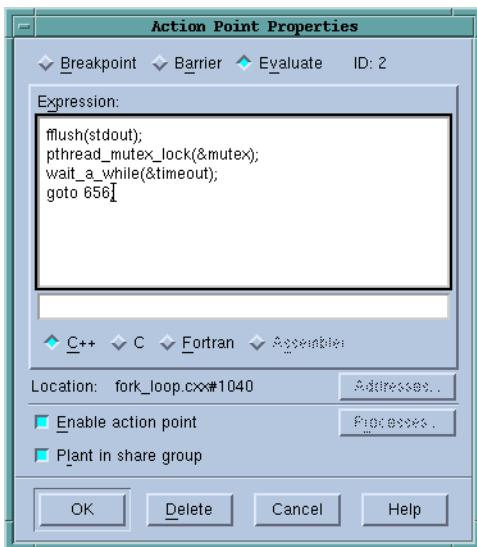
Eval points let you patch your programs and route around code that you want replaced. For example, suppose you need to change a bunch of statements. Just add these statements to an action point, then add a **goto** statement that jumps over the code you no longer want executed. For example, the eval point shown in the following figure tells TotalView to execute three statements and then skip to line 656. (See Figure 4 on page 7.)

Examining Data

Programmers use print statements as an easy way to examine data. They usually do this because their debugger doesn't have sophisticated ways of showing information. In contrast, Chapter 14, "Examining and Changing Data," on page 279 and Chapter 15, "Examining Arrays," on page 333 explain how TotalView displays data values. In addition, Chapter 9, "Visualizing Programs and Data," on page 179 describes how TotalView visualizes your data graphically.

Because data is difficult to see, the Stack Frame Pane (the pane in the upper right corner of the Process Window (see Figure 1 on page 4) has a list

Figure 4: Patching Using an Eval Point



of all variables that exist in your current routine. If the value is simple, you'll see its value in this pane.

If the value isn't simple, just dive on the variable to get more information.



Diving is something you can do almost everywhere in TotalView. What happens depends on where you are. In general, it either brings you to a different place in your program or shows you more information about what you're diving on. To dive on something, position the cursor over the item and click your middle mouse button or double-click using your left mouse button.

Diving on a variable tells TotalView to display a window that contains information about the variable. (As you read this manual, you'll come across many other types of diving.)

Some of the values in the Stack Frame Pane are in **bold** type. This lets you know that you can click on the value and then edit it.

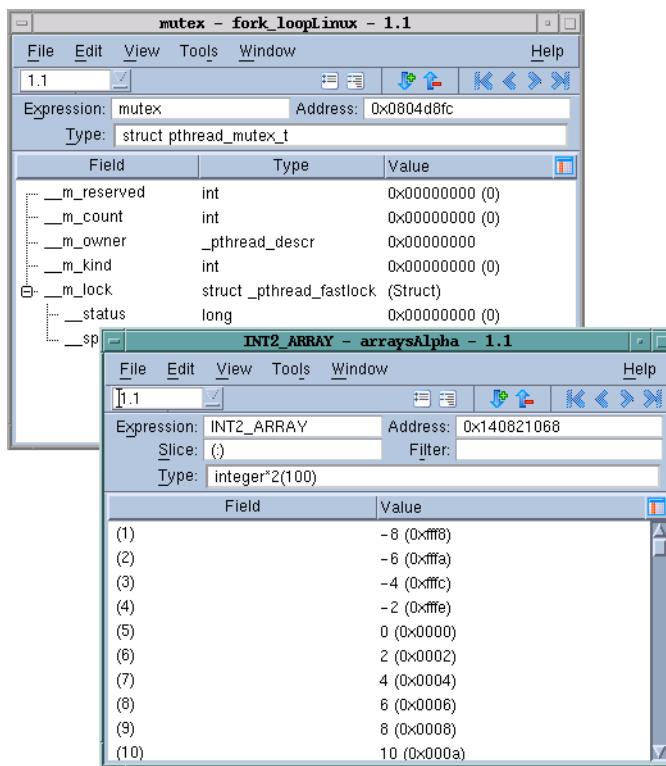
(Figure 5 on page 8 shows two Variable Windows. One window was created by diving on a structure and the second by diving on an array.

Because the data displayed in a Variable Window might not be simple, you can redive on this data. When you dive in a Variable Window, TotalView replaces the window's contents with the new information. If you don't want to replace the contents, you can use the **View > Dive Thread in New Window** command to display this information in a separate window.

If the data being displayed is a pointer, diving on the variable dereferences the pointer and then displays the data that is being pointed to. In this way, you can follow linked lists.

The upper right corner of a Variable Window has arrow buttons (◀◀▶▶). Selecting these buttons lets you undive and redive. For example, if you're following a pointer chain, click the center-left-pointing arrow to go back to where you just were. Click the center-right-pointing arrow to go forward to

Figure 5: Diving on a Structure and an Array



the place you previously dove on. The outermost two arrows do “undive all” and “redive all” operations.

Examining Arrays

Because arrays almost always have copious amounts of data, TotalView has a variety of ways to simplify how it should display this data.

The Variable Window in the upper left corner of the figure on the next page shows a basic *slice* operation. Slicing tells TotalView to display array elements whose positions are named within the slice. In this case, TotalView is displaying elements 6 through 10 in each of the array’s two dimensions. The other Variable Window in this figure combines a *filter* with a slice. A filter tells TotalView to display data if it meets some criteria that you specify. Here, the filter says “of the array elements that could be displayed, only display elements whose value is greater than 300.” (See Figure 6 on page 9.)

While slicing and filtering let you reduce the amount of data that TotalView displays, you might want to see the shape of the data. If you select the **Tools > Visualize** command, TotalView shows a graphic representation of the information in the Variable Window. (See Figure 7 on page 9.)

Seeing Groups of Variables

Variable Windows let you critically examine many aspects of your data. In many cases, you’re not interested in much of this information. Instead, all you’re interested in is the variable’s value. This is what the Expression List

Figure 6: Slicing and Filtering Arrays

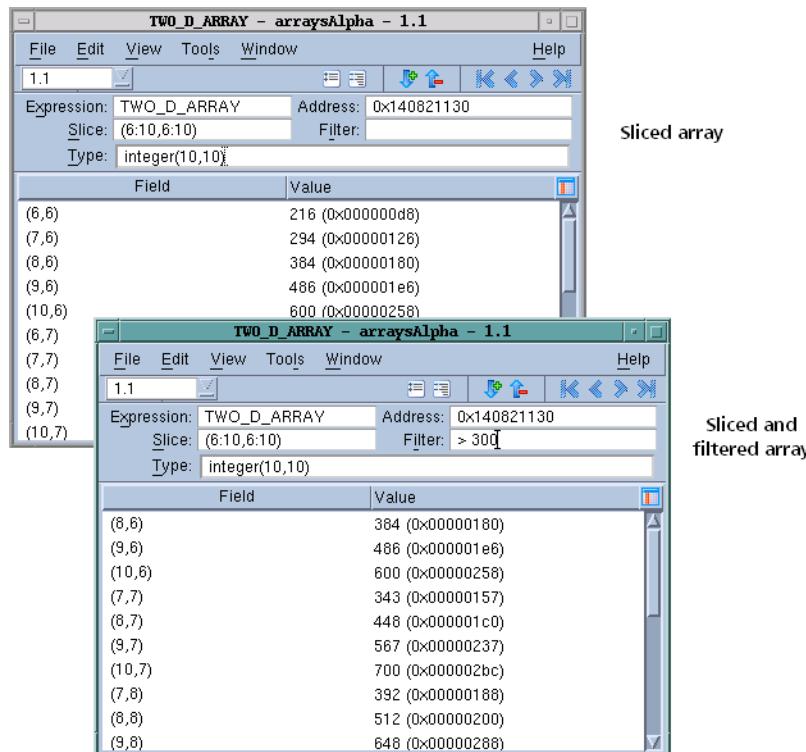
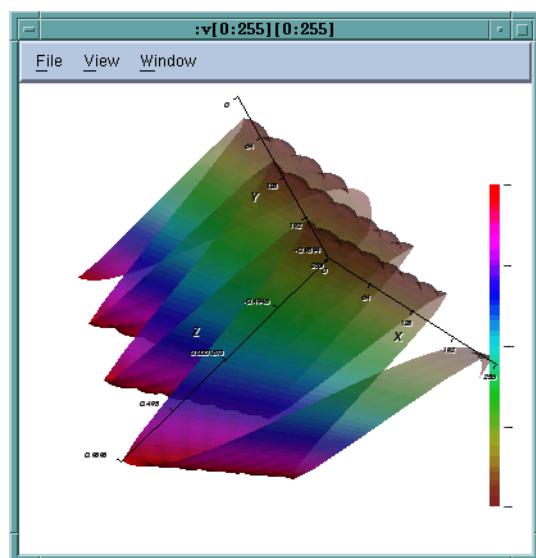


Figure 7: Visualizing an Array



Window is for. It also differs from the Variable Window in that it lets you see the values of many variables at the same time. (See Figure 8 on page 10.)

You can add variables to this window in several ways, such as:

- Type the variable's name in the Expression column.
- Select the variable in the Source or Stack Frame Panes or in a Variable Window, right-click, then select **Add to Expression List** from the context menu.

Figure 8: Tools > Expression List Window

The screenshot shows a window titled "diveinall_cLinux - 1,1". The menu bar includes File, Edit, View, Window, and Help. A toolbar with icons for search and refresh is at the top right. The main area is a table with two columns: "Expression" and "Value". The table contains the following data:

Expression	Value
i	0x00000003 (3)
d1_array	(class d1[3])
d1_array[1].d1_v	0x00000001 (1)
d1_array[i-1].d1_v	0x00000004 (4)

For more information, see "Viewing a List of Variables" on page 303.

Setting Watchpoints

Using watchpoints is yet another way to look at data. A TotalView watchpoint stops execution when a variable's data changes, no matter what instruction changed the data. That is, if you change data from 30 different statements, the watchpoint stops execution right after any of these 30 statements make a change. Another example is if something is trashing a memory location, you can put a watchpoint on that location and then wait until TotalView stops execution because the watchpoint was executed.

To create a watchpoint for a variable, select **Tools > Create Watchpoint** from the variable's Variable Window or by selecting **Action Points > Create Watchpoint** in the Process Window.

Debugging Multi-process and Multi-threaded Programs

When your program creates processes and threads, TotalView can automatically bring them under its control. If the processes are already running, TotalView can acquire them as well. You don't need to have multiple debuggers running: one TotalView is all you need.

The processes that your program creates can be local or remote. Both are presented to you in the same way. You can display them in the current Process Window or display them in an additional window.

The Root Window, which automatically appears after you start TotalView, contains an overview of all processes and threads being debugged. Diving on a process or a thread listed in the Root Window takes you quickly to the information you want to see. (See Figure 9 on page 11.)

Figure 9: The Root Window

	ID	Rank	Host	Status	Description
⊕	1		<local>	T	fork_linux (5 active threads)
⊕	2		<local>	T	fork_linux.1 (5 active threads)
⊕	3		<local>	T	fork_linux.2 (5 active threads)
⊕	4		<local>	T	fork_linux.1.1 (5 active threads)
⊕	4.1		<local>	T	in __select
⊕	4.2		<local>	T	in __select
⊕	4.3		<local>	T	in __select
⊕	4.4		<local>	T	in __select
⊕	5		<local>	T	fork_linux.1.2 (5 active threads)
⊕	6		<local>	T	fork_linux.1.1.1 (5 active threads)
⊕	7		<local>	T	fork_linux.2.1 (5 active threads)
⊕	8		<local>	T	fork_linux.3 (5 active threads)

If you need to debug processes that are already running, select the **File > New Program** command, then select **Attach to an existing process** on the left side of the dialog box. After selecting an entry and pressing the OK button, you can debug these processes in the same way as any other process or thread.

In the Process Window, you can switch between processes by clicking on a box within the Processes tab. Every time you click on one, TotalView switches contexts. Similarly, clicking on a thread, changes the context to that thread.

Program Using Almost Any Execution Model

In many cases, you'll be using one of the popular parallel execution models. TotalView supports MPI and MPICH, OpenMP, ORNL PVM (and HP Alpha DPVM), SGI shared memory (shmem), Global Arrays, and UPC. You could be using threading in your programs. Or, you can compile your programs using compilers provided by your hardware vendor or other vendors such as those from Intel and the Free Software Foundation (the GNU compilers).

Supporting Multi-process and Multi-threaded Programs

When debugging multi-process, multi-threaded programs, you often want to see the value of a variable in each process or thread simultaneously. Do this by telling TotalView to show the variable either across processes or threads. Figure 10 on page 12 shows how TotalView displays this information for a multi-threaded program.

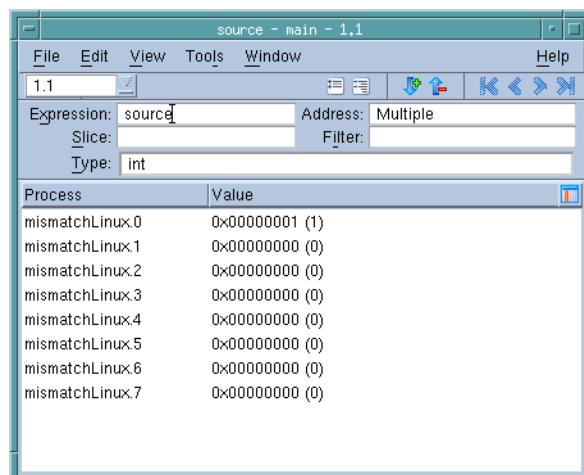
If you're debugging an MPI program, the **Tools > Message Queue Graph** Window graphically displays the program's message queues. (See Figure 11 on page 12.)

Clicking on the boxed numbers tells TotalView to place the associated process into a Process Window. Clicking on a number next to the arrow tells TotalView to display more information about that message queue.

This book contains many additional examples.

Using Groups and Barriers

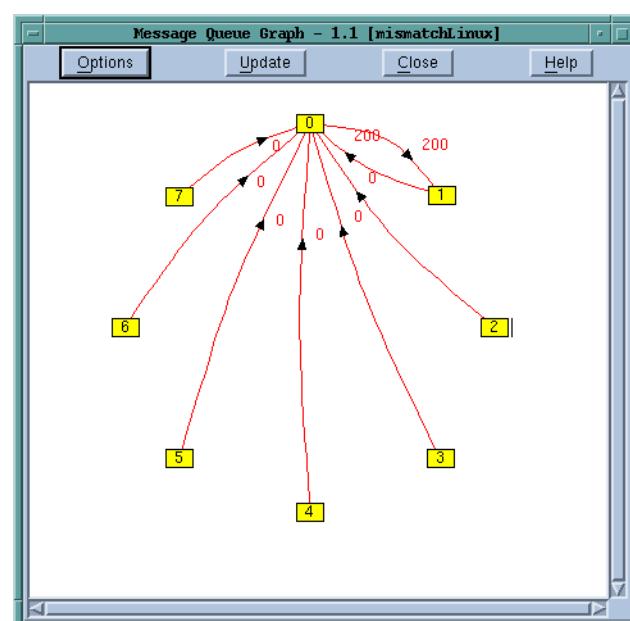
Figure 10: Viewing Across Processes



A screenshot of the TotalView interface titled "source - main - 1.1". The window has a menu bar with File, Edit, View, Tools, Window, and Help. Below the menu is a toolbar with icons for file operations. The main area contains a table with columns "Process" and "Value". The table lists processes mismatchLinux.0 through mismatchLinux.7, each with a value of 0x00000000 (0).

Process	Value
mismatchLinux.0	0x00000001 (1)
mismatchLinux.1	0x00000000 (0)
mismatchLinux.2	0x00000000 (0)
mismatchLinux.3	0x00000000 (0)
mismatchLinux.4	0x00000000 (0)
mismatchLinux.5	0x00000000 (0)
mismatchLinux.6	0x00000000 (0)
mismatchLinux.7	0x00000000 (0)

Figure 11: A Message Queue Graph



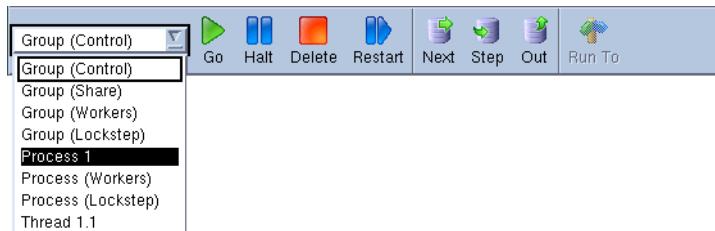
Using Groups and Barriers

When running a multi-process and multi-threaded program, TotalView tries to automatically place your executing processes into different groups.

While you can always individually stop, start, step, and examine any thread or process, TotalView lets you perform these actions on groups of threads and processes. In most cases, you do the same kinds of operations on the

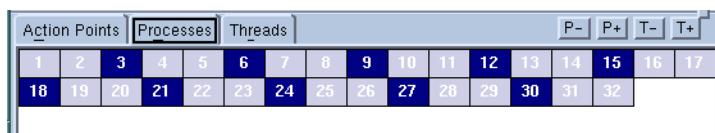
same kinds of things. The toolbar's pulldown menu lets you select the target of your action. Figure 12 shows this toolbar.

Figure 12: Toolbar With Pulldown



For example, if you are debugging an MPI program, you might select **Process (Workers)** from the Toolbar. (Chapter 13, "Using Groups, Processes, and Threads", describes the reasons for setting the pulldown this way.) The Processes/Ranks tab at the bottom of the window shows you which processes are within this group. (See Figure 13.)

Figure 13: Process Tab



This figure shows the Processes Tab after a group containing 10 processes was selected in the Toolbar's Group pulldown list. You can now see what processes are acted upon when you select a command such as Go or Step,

Memory Debugging

Trying to find memory problems with TotalView is a lot different from what you've grown accustomed to. The main difference is that it is a lot easier because MemoryScape is an integrated part of TotalView.. (MemoryScape can also be run as a separate stand-alone program.) MemoryScape is described in its own book.

Introducing the CLI

The Command Line Interpreter, or CLI, contains an extensive set of commands that you can type into a command window. These commands are embedded in a version of the Tcl command interpreter. When you open a CLI window, you can enter any Tcl statements that you could enter in any version of Tcl. You can also enter commands that TotalView Technologies has added to Tcl that allow you to debug your program. Because these debugging commands are native to this Tcl, you can also use Tcl to manipulate the program being debugged. This means that you can use the CLI to create your own commands or perform any kind of repetitive operation. For example, the following code shows how to set a breakpoint at line 1038 using the CLI:

```
dbreak 1038
```

When you combine Tcl and TotalView, you can simplify what you are doing. For example, the following code shows how to set a group of breakpoints:

```
foreach i {1038 1043 1045} {  
    dbreak $i  
}
```

Chapter 11, "Seeing the CLI at Work," on page 213 presents more realistic examples.

Information about the CLI is scattered throughout this book. Chapter 3 of the *TotalView Reference Guide* contains descriptions of most CLI commands.

What's Next

This chapter has presented just a few TotalView highlights. The rest of this book tells you more about TotalView.

All TotalView documentation is available on our Web site at <http://www.totalviewtech.com/Documentation> in PDF and HTML formats. You can also find this information in the online Help.

About Threads, Processes, and Groups



CHAPTER 2

While the specifics of how multi-process, multi-threaded programs execute differ greatly from one hardware platform to another, from one operating system to another, and from one compiler to another, all share some general characteristics. This chapter defines a general model for conceptualizing the way processes and threads execute.

This chapter presents the concepts of *threads*, *processes*, and *groups*. Chapter 13, "Using Groups, Processes, and Threads," on page 251 is a more exacting and comprehensive look at these topics.

This chapter contains the following sections:

- "A Couple of Processes" on page 15
- "Threads" on page 18
- "Complicated Programming Models" on page 19
- "Types of Threads" on page 20
- "Organizing Chaos" on page 22
- "Creating Groups" on page 26
- "Simplifying What You're Debugging" on page 30

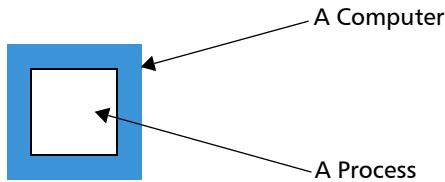
A Couple of Processes

When programmers write single-threaded, single-process programs, they can almost always answer the question "Do you know where your program is?" These types of programs are rather simple, looking something like what's shown in the figure on the next page.

If you use any debugger on these types of programs, you can almost always figure out what's going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then

A Couple of Processes

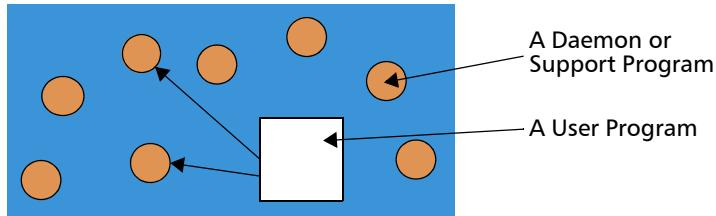
Figure 14: A Uniprocessor



inspect variables to see their values. If you suspect that there's a logic problem, you can step the program through its statements, seeing what happens and where things are going wrong.

What is actually occurring, however, is a lot more complicated, since a number of programs are always executing on your computer. For example, your computing environment could have daemons and other support programs executing, and your program can interact with them.

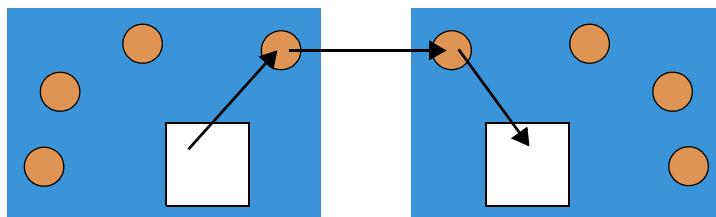
Figure 15: A Program and Daemons



These additional processes can simplify your program because it no longer has to do everything itself. It can hand off some tasks and not have to focus on how that work gets done.

The preceding figure shows an architecture where the application program just sends requests to a daemon. This architecture is very simple. The type of architecture shown in the next figure is more typical. In this example, an email program communicates with a daemon on one computer. After receiving a request, this daemon sends data to an email daemon on another computer. After the handoff, the programs do not interact. The program handing off the work just assumes that the work gets done. Some programs can work well like this. Most don't. Most computational jobs do better with a model that

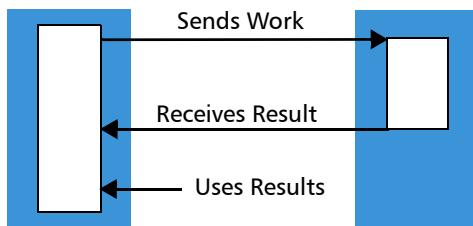
Figure 16: Mail Using Daemons to Communicate



This architecture has one program handing off work to another. After the handoff, the programs do not interact. The program handing off the work just assumes that the work gets done. Some programs can work well like this. Most don't. Most computational jobs do better with a model that

allows a program to divide its work into smaller jobs, and parcel this work to other computers. Said in a different way, this model has other machines do some of the first program's work. To gain any advantage, however, the work a program parcels out must be work that it doesn't need right away. In this model, the two computers act more or less independently. And, because the first computer doesn't have to do all the work, the program can complete its work faster.

Figure 17: Two Computers Working on One Problem



Using more than one computer doesn't mean that less computer time is being used. Overhead due to sending data across the network and overhead for coordinating multi-processing always means more work is being done. It does mean, however, that your program finishes sooner than if only one computer were working on the problem.

One problem with this model is how a programmer debugs what's happening on the second computer. One solution is to have a debugger running on each computer. The TotalView solution to this debugging problem places a server on each remote processor as it is launched. These servers then communicate with the main TotalView process. This debugging architecture gives you one central location from which you can manage and examine all aspects of your program.



You can also have TotalView attach to programs that are already running on other computers. In other words, programs don't have to be started from within TotalView to be debugged by TotalView.

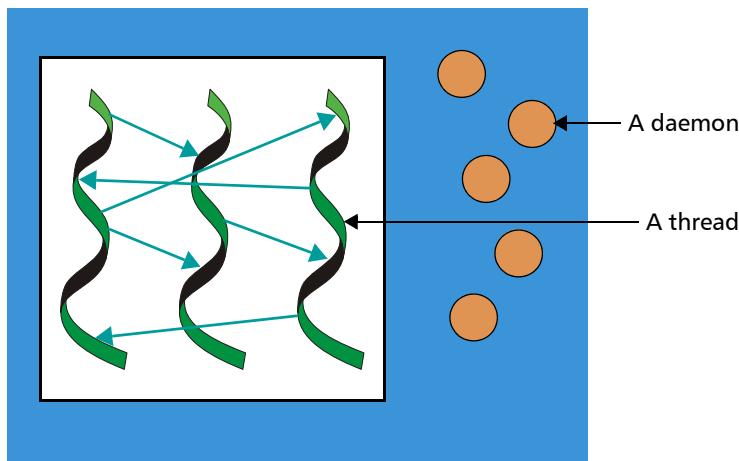
In all cases, it is far easier to write your program so that it only uses one computer at first. After you have it working, you can split up its work so that it uses other computers. It is likely that any problems you find will occur in the code that splits up the program or in the way the programs manipulate shared data, or in some other area related to the use of more than one thread or process. This assumes, of course, that it is practical to write your program as a single-process program. For some algorithms, executing a program on one computer means that it will take weeks to execute.

Threads

The operating system owns the daemon programs discussed in the previous section. These daemons perform a variety of activities, from managing computer resources to providing standard services such as printing.

If operating systems can have many independently executing components, why can't a program? Obviously, a program can and there are various ways to do this. One programming model splits the work off into somewhat independent tasks within the same process. This is the *threads* model.

Figure 18: Threads



This figure also shows the daemon processes that are executing. (The figures in the rest of this chapter won't show these daemons.)

In this computing model, a program (the main thread) creates threads. If they need to, these newly created threads can also create threads. Each thread executes relatively independently from other threads. You can, of course, program them to share data and to synchronize how they execute.

The debugging issue here is similar to the problem of processes running on different machines. In both, a debugger must intervene with more than one executing entity. It has to understand multiple address spaces and multiple contexts.

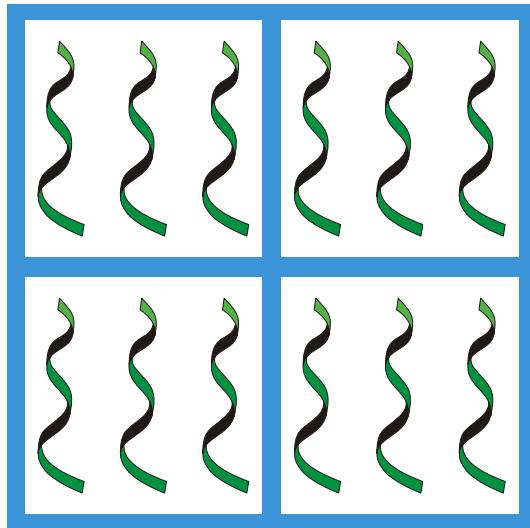
There's not a lot of difference between a multi-threaded or a multi-process program when you are using TotalView. The way in which TotalView displays process information is very similar to how it displays thread information.



Complicated Programming Models

While most computers have one or two processors, high-performance computing often uses computers with many more. And as hardware prices decrease, this model is starting to become more widespread. Having more than one processor means that the threads model shown in the figure in the previous section changes to look something like what is shown in Figure 19. (Only four cores are shown even though many more could on a chip.)

Figure 19: Four-Processor Computer



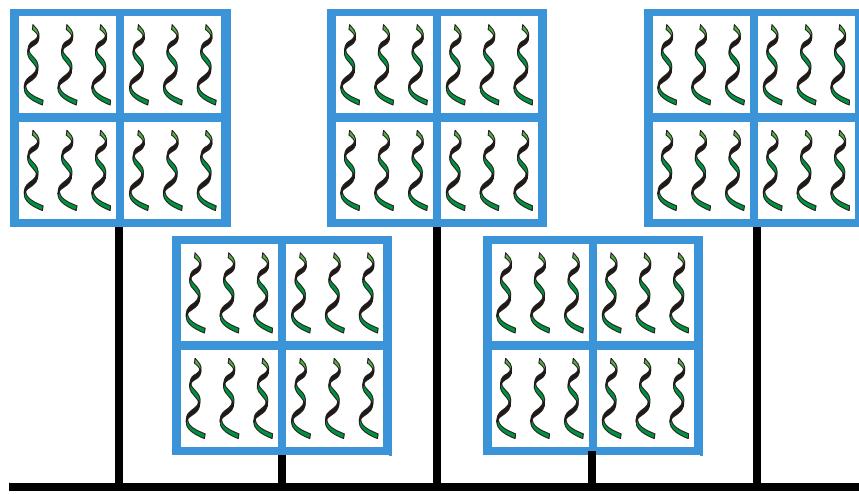
This figure shows four cores in one computer, each of which has three threads. This architecture is an extension to the model that links more than one computer together. Its advantage is that the processor doesn't need to communicate with other processors over a network as it is completely self-contained.

The next step is to join many multi-processor computers together. (See Figure 20 on page 20.) shows five computers, each with four processors, with each processor running three threads. If this figure shows the execution of one program, then the program is using 60 threads.

This figure depicts only processors and threads. It doesn't have any information about the nature of the programs and threads or even whether the programs are copies of one another or represent different executables.

At any time, it is next to impossible to guess which threads are executing and what a thread is actually doing. To make matters worse, many multi-processor programs begin by invoking a process such as `mpirun` or IBM `poe`, whose function is to distribute and control the work being performed. In this kind of environment, a program is using another program to control the workflow across processors.

Figure 20: Four Processors on a Network

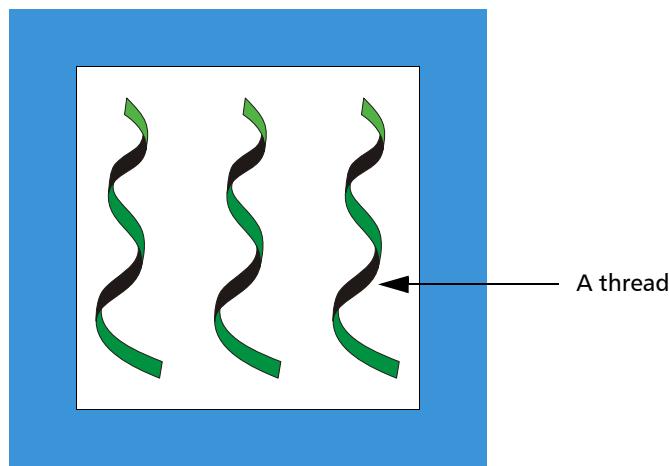


When there are problems working this way, traditional debuggers and solutions don't work. TotalView, on the other hand, organizes this mass of executing procedures for you and lets you distinguish between threads and processes that the operating system uses from those that your program uses.

Types of Threads

All threads aren't the same. The following figure shows a program with three threads. (See Figure 21.)

Figure 21: Threads (again)



Assume that all of these threads are *user threads*; that is, they are threads that perform some activity that you've programmed.

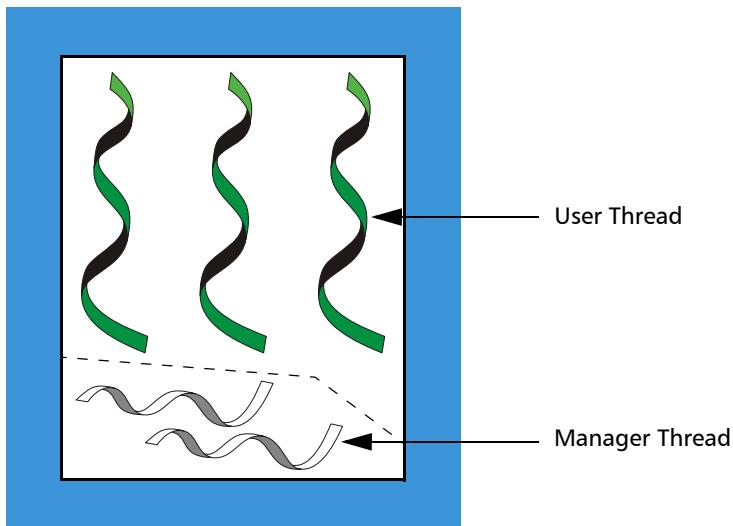


Many computer architectures have something called user mode, user space, or something similar. User threads means something else. The TotalView definition of a user thread is simply a unit of execution created by a program.

Because the program creates user threads to do its work, they are also called *worker threads*.

Other threads can also be executing. For example, there are always threads that are part of the operating environment. These threads are called *manager threads*. Manager threads exist to help your program get its work done. In the following figure, the horizontal threads at the bottom are user-created manager threads.

Figure 22: User and Service Threads



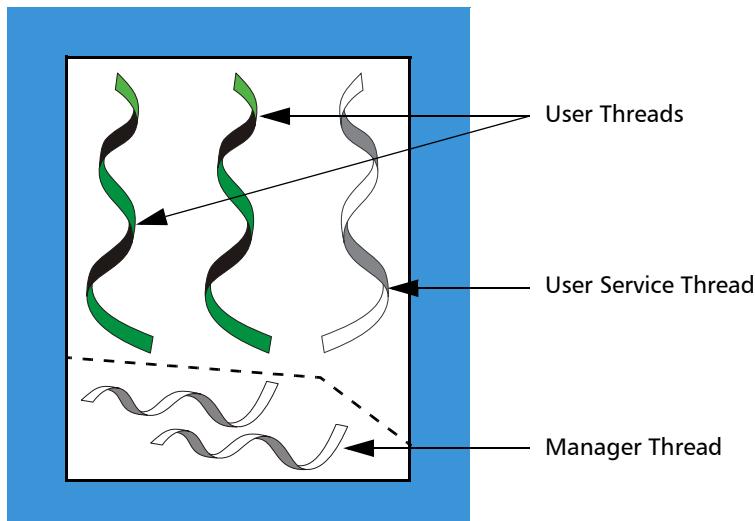
All threads are not created equal and all threads do not execute equally. Many programs also create manager-like threads. Since these user-created manager threads perform services for other threads, they are called *service threads*. (See Figure 23 on page 22.)

These service threads are also worker threads. For example, the sole function of a user service thread might be to send data to a printer in response to a request from the other two threads.

One reason you need to know which of your threads are service threads is that a service thread performs different types of activities than your other threads. Because their activities are different, they are usually developed separately and, in many cases, are not involved with the fundamental problems being solved by the program. Here are two examples:

- The code that sends messages between processes is far different than the code that performs fast Fourier transforms. Its bugs are quite different than the bugs that create the data that is being transformed.
- A service thread that queues and dispatches messages sent from other threads might have bugs, but the bugs are different than the rest of your code and you can handle them separately from the bugs that occur in nonservice user threads.

Figure 23: User, Service, and Manager Threads



Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that actively participate in an activity, rather than on threads performing subordinate tasks.

Although this last figure shows five threads, most of your debugging effort will focus on just two threads.

Organizing Chaos

It is possible to debug programs that are running thousands of processes and threads across hundreds of computers by individually looking at each. However, this is almost always impractical. The only workable approach is to organize your processes and threads into groups and then debug your program by using these groups. In other words, in a multi-process, multi-threaded program, you are most often not programming each process or thread individually. Instead, most high-performance computing programs perform the same or similar activities on different sets of data.

TotalView cannot know your program's architecture; however, it can make some intelligent guesses based on what your program is executing and where the program counter is. Using this information, TotalView automatically organizes your processes and threads into the following predefined groups:

- **Control Group:** All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program that has two distinct executables that run independently of one another has each executable in a separate control

group. In contrast, processes created by `fork()/exec()` are in the same control group.

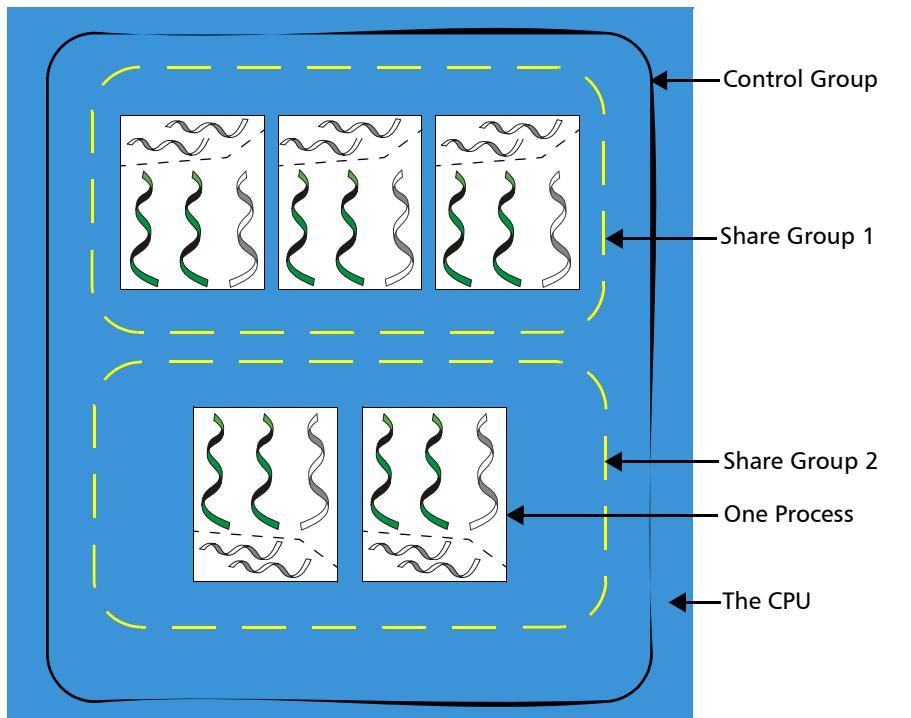
- **Share Group:** All the processes within a control group that share the same code. *Same code* means that the processes have the same executable file name and path. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.
- **Workers Group:** All the worker threads within a control group. These threads can reside in more than one share group.
- **Lockstep Group:** All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. By definition, all members of a lockstep group are within the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group. A lockstep group only means anything when the threads are stopped.

The control and share groups only contain processes; the workers and lockstep groups only contain threads.

TotalView lets you manipulate processes and threads individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent). For more information, see Chapter 13, "Using Groups, Processes, and Threads," on page 251.

The following figure shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within the processes. This figure shows a control group and two share groups within the control group.

Figure 24: Five-Processes: Their Control and Share Groups

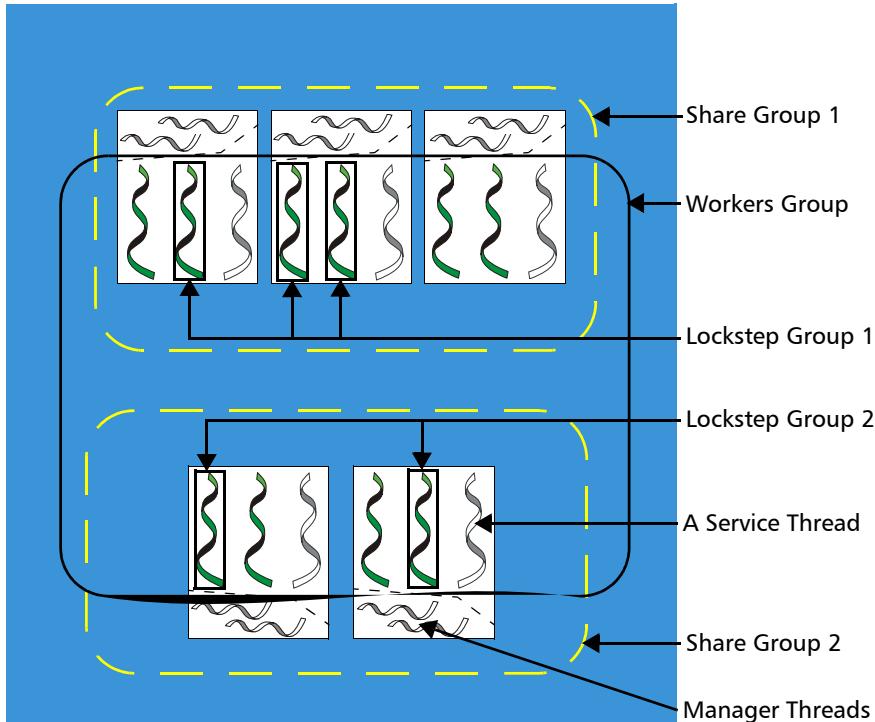


Many of the elements in this figure are used in other figures in this book. These elements are as follows:

CPU	The one outer square represents the CPU. All elements in the drawing operate within one CPU.
Processes	The five white inner squares represent processes being executed by the CPU.
Control Group	The large rounded rectangle that surrounds the five processes shows one control group. This diagram doesn't indicate which process is the main procedure.
Share Groups	The two smaller rounded rectangles having white dashed lines surround processes in a share group. This drawing shows two share groups within one control group. The three processes in the first share group have the same executable. The two processes in the second share group share a second executable.

The control group and the share group only contain processes. The next figure shows how TotalView organizes the threads in the previous figure. It adds a workers group and two lockstep groups. (See Figure 25.)

Figure 25: Five Processes:
Adding Workers and
Lockstep Groups



This figure doesn't show the control group since it encompasses everything in this figure. That is, this example's control group contains all of the program's lockstep, share, and worker group's processes and threads.

The additional elements in this figure are as follows:

Workers Group All nonmanager threads within the control group make up the workers group. This group includes service threads.

Lockstep Groups Each share group has its own lockstep group. The previous figure shows two lockstep groups, one in each share group.

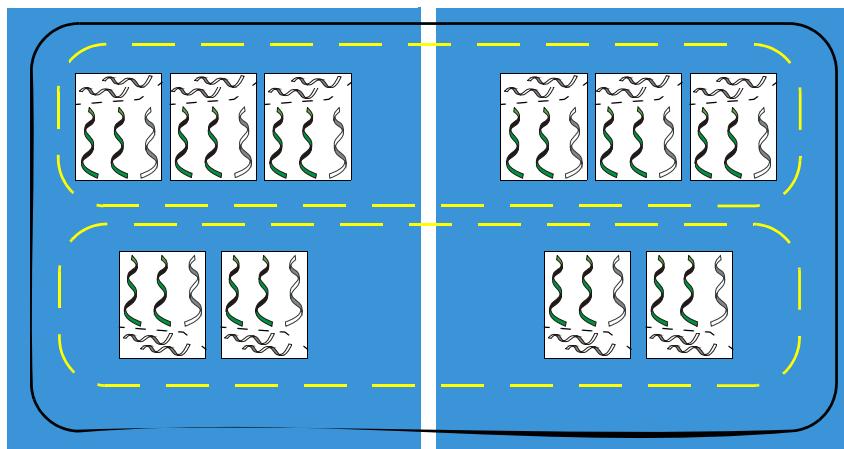
Service Threads Each process has one service thread. A process can have any number of service threads, but this figure only shows one.

Manager Threads

The ten manager threads are the only threads that do not participate in the workers group.

The following figure extends the previous figure to show the same kinds of information executing on two processors.

Figure 26: Five Processes and Their Groups on Two Computers



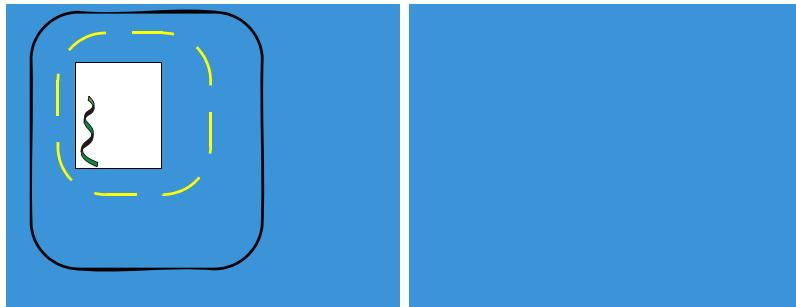
This figure differs from the other ones in this section because it shows ten processes executing within two processors rather than five processes within one processor. Although the number of processors has changed, the number of control and share groups is unchanged. This makes a nice example. However, most programs are not this regular.

Creating Groups

TotalView places processes and threads in groups as your program creates them. The exception is the lockstep groups that are created or changed whenever a process or thread hits an action point or is stopped for any reason. There are many ways to build this type of organization. The following steps indicate the beginning of how TotalView might do this.

- Step 1** TotalView and your program are launched and your program begins executing.

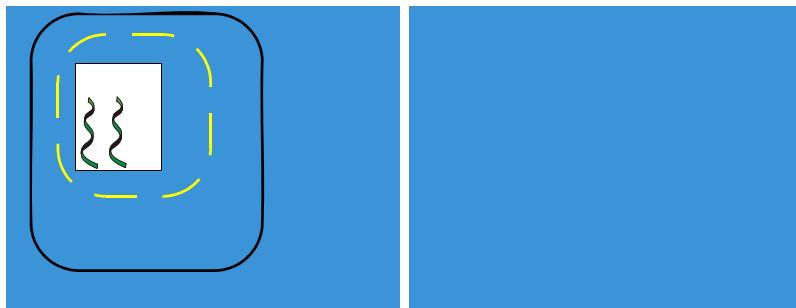
Figure 27: Step 1: A Program Starts



- **Control group:** The program is loaded and creates a group.
- **Share group:** The program begins executing and creates a group.
- **Workers group:** The thread in the `main()` routine is the workers group.
- **Lockstep group:** There is no lockstep group because the thread is running. (Lockstep groups only contain stopped threads.)

- Step 2** The program creates a thread.

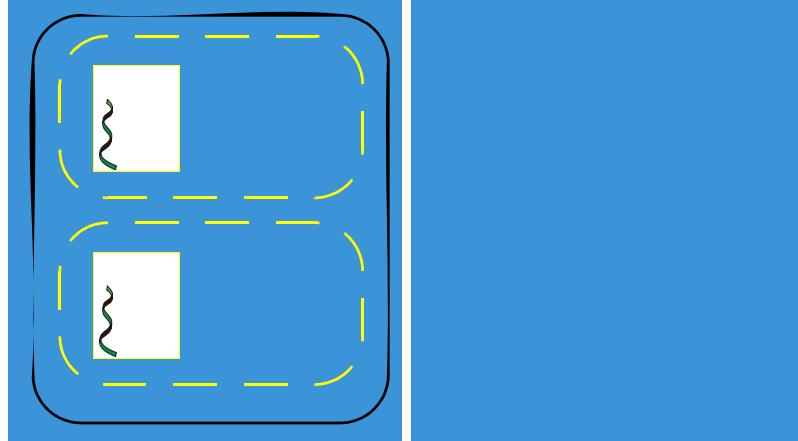
Figure 28: Step 1: A Program Starts



- **Control group:** The control group is unchanged.
- **Share group:** The share group is unchanged.
- **Workers group:** TotalView adds the thread to the existing group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

- Step 3** The first process uses the `exec()` function to create a second process. (See Figure 29.)

Figure 29: Step 3: Creating a Process using `exec()`



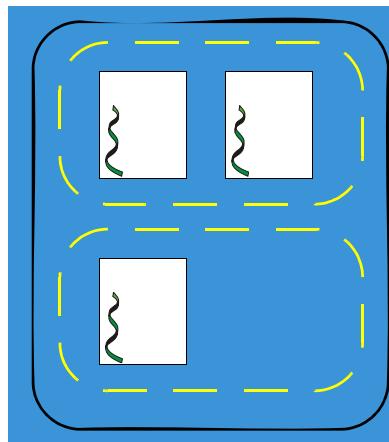
- **Control group:** The group is unchanged.
- **Share group:** TotalView creates a second share group with the process created by the `exec()` function as a member. TotalView removes this process from the first share group.
- **Workers group:** Both threads are in the workers group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

- Step 4** The first process hits a break point.

- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep group:** TotalView creates a lockstep group whose member is the thread of the current process. (In this example, each thread is its own lockstep group.)

- Step 5** The program is continued and TotalView starts a second version of your program from the shell. You attach to it within TotalView and put it in the same control group as your first process.

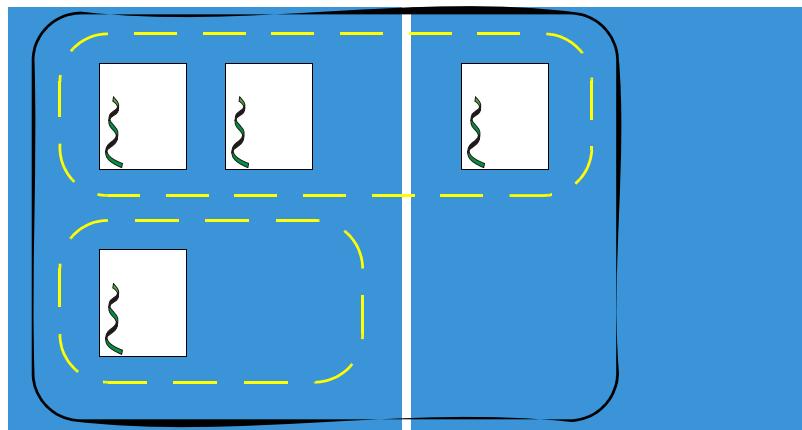
Figure 30: Step 5: Creating a Second Version



- **Control group:** TotalView adds a third process.
- **Share group:** TotalView adds this third process to the first share group.
- **Workers group:** TotalView adds the thread in the third process to the group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

- Step 6** Your program creates a process on another computer.

Figure 31: Step 6: Creating a Remote Process

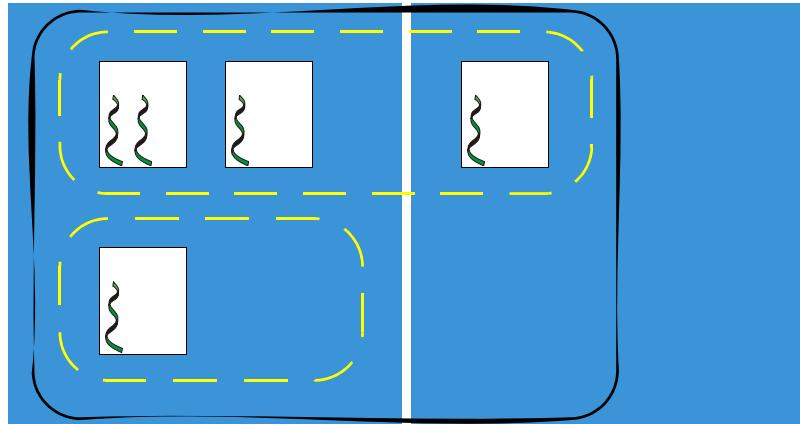


- **Control group:** TotalView extends the control group so that it contains the fourth process, which is running on the second computer.
- **Share group:** The first share group now contains this newly created process, even though it is running on the second computer.
- **Workers group:** TotalView adds the thread within this fourth process to the workers group.

- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 7 A process within the control group creates a thread. This adds a second thread to one of the processes.

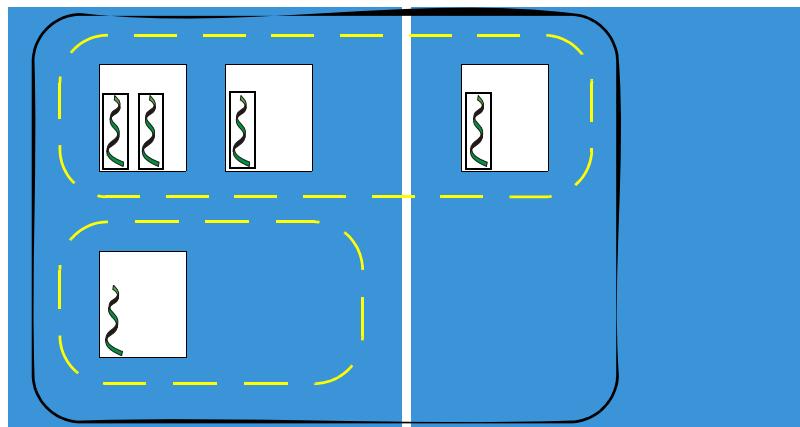
Figure 32: Step 7: Creating a Thread



- **Control group:** The group is unchanged.
- **Share group:** The group is unchanged.
- **Workers group:** TotalView adds a fifth thread to this group.
- **Lockstep group:** There are no lockstep groups because the threads are running.

Step 8 A breakpoint is set on a line in a process executing in the first share group. By default, TotalView shares the breakpoint. The program executes until all three processes are at the breakpoint.

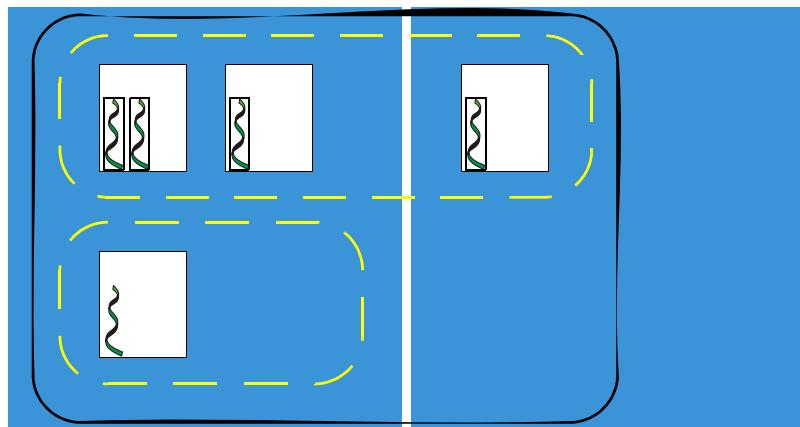
Figure 33: Step 8: Hitting a Breakpoint



- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep groups:** TotalView creates a lockstep group whose members are the four threads in the first share group.

Step 9 You tell TotalView to step the lockstep group.

Figure 34: Step 9: Stepping the Lockstep Group



- **Control group:** The group is unchanged.
- **Share group:** The groups are unchanged.
- **Workers group:** The group is unchanged.
- **Lockstep group:** The lockstep groups are unchanged. (There are other lockstep groups; this is explained in Chapter 13, "Using Groups, Processes, and Threads," on page 251.)

What Comes Next

This example could keep on going to create a more complicated system of processes and threads. However, adding more processes and threads won't change the basics of what has been covered.

Simplifying What You're Debugging

The reason you're using a debugger is because your program isn't operating correctly and the way you think you're going to solve the problem (unless it is a &%\$# operating system problem, which, of course, it usually is) is by stopping your program's threads, examining the values assigned to variables, and stepping your program so you can see what's happening as it executes.

Unfortunately, your multi-process, multi-threaded program and the computers upon which it is executing have lots of things executing that you want TotalView to ignore. For example, you don't want to be examining manager and service threads that the operating system, your programming environment, and your program create.

Also, most of us are incapable of understanding exactly how a program is acting when perhaps thousands of processes are executing asynchronously. Fortunately, there are only a few problems that require full asynchronous behavior at all times.

One of the first simplifications you can make is to change the number of processes. For example, suppose you have a buggy MPI program running on 128 processors. Your first step might be to have it execute in an 8-processor environment.

After you get the program running under TotalView control, run the process being debugged to an action point so that you can inspect the program's state at that point. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing.



TotalView lets you control as many groups, processes, and threads as you need to control. Although you can control each one individually, you might have problems remembering what you're doing if you're controlling large numbers of these things independently. TotalView creates and manages groups so that you can focus on portions of your program.

In most cases, you don't need to interact with everything that is executing. Instead, you want to focus on one process and the data that this process manipulates. Things get complicated when the process being investigated is using data created by other processes, and these processes might be dependent on other processes.

The following is a typical way to use TotalView to locate problems:

- 1 At some point, make sure that the groups you are manipulating do not contain service or manager threads. (You can remove processes and threads from a group by using the **Group > Custom Group** command.)

CLI: dgroups –remove

- 2 Place a breakpoint in a process or thread and begin investigating the problem. In many cases, you are setting a breakpoint at a place where you hope the program is still executing correctly. Because you are debugging a multi-process, multi-threaded program, set a *barrier point* so that all threads and process will stop at the same place.



Don't step your program except where you need to individually look at what occurs in a thread. Using barrier points is much more efficient. Barrier points are discussed in "Setting Barrier Points" on page 362 and online within the Action Point area within the Tip of the Week archive at <http://www.totalviewtech.com/Support/Tips/>.

- 3 After execution stops at a barrier point, look at the contents of your variables. Verify that your program state is actually correct.
- 4 Begin stepping your program through its code. In most cases, step your program synchronously or set barriers so that everything isn't running freely.

Things begin to get complicated at this point. You've been focusing on one process or thread. If another process or thread modifies the data and you become convinced that this is the problem, you need to go off to it and see what's going on.

You need to keep your focus narrow so that you're only investigating a limited number of behaviors. This is where debugging becomes an art. A multi-process, multi-threaded program can be doing a great number of things. Understanding where to look when problems occur is the art.

For example, you most often execute commands at the default focus. Only when you think that the problem is occurring in another process do you change to that process. You still execute in the default focus, but this time the default focus changes to another process.

Although it seems like you're often shifting from one focus to another, you probably will do the following:

- Modify the focus so that it affects just the next command. If you are using the GUI, you might select this process and thread from the list displayed in the Root Window. If you are using the CLI, you use the **dfocus** command to limit the scope of a future command. For example, the following is the CLI command that steps thread 7 in process 3:

dfocus t3.7 dstep

- Use the **dfocus** command to change focus temporarily, execute a few commands, and then return to the original focus.

This chapter is just an overview of the threads, processes, and groups. Chapter 13, "Using Groups, Processes, and Threads," on page 251 contains the details.

Part II: Setting Up



This section of the *TotalView Users Guide* contains information about running TotalView in the different types of environments in which you execute your program.

Chapter 3: Getting Started with Remote Display Client

This chapter describes how you can start and interact with TotalView when it is executing on another computer.

Chapter 4: Setting Up a Debugging Session

This chapter tells you what you need to know to start TotalView and tailor how it works.

Chapter 5: Setting Up Remote Debugging Sessions

When you are debugging a program that has processes executing on a remote computer, TotalView launches server processes for these remote processes. Usually, you don't need to know much about this. The primary focus of this chapter is what to do when you have problems.

If you aren't having problems, you probably won't ever look at the information in this chapter.

Chapter 6: Setting Up MPI Debugging Sessions

When you are debugging a program that has processes executing on a remote computer, TotalView launches server processes for these remote processes. Usually, you don't need to know much about this. The primary focus of this chapter is what to do when you have problems.

Debugging other kinds of parallel programs is discussed in the next chapter.

Chapter 7: Setting Up Parallel Debugging Sessions

TotalView lets you debug programs created using many different parallel environments, such as OpenMP, SHMEM, Global Arrays, UPC, and the like. This chapter discusses how to set up these environments.

Debugging MPI programs is discussed in the previous chapter.

Getting Started with Remote Display Client



Using Remote Display

CHAPTER 3

TotalView Remote Display lets you start and then view TotalView as it executes on another system. For example, if you are using a Microsoft Windows XP, invoking the Remote Display Client (this will be explained in a moment) displays a window into which you can enter information about how Remote Display can go from Windows to the system upon which TotalView will execute. As Remote Display invokes TotalView on the remote host, it need not be installed on your local machine.

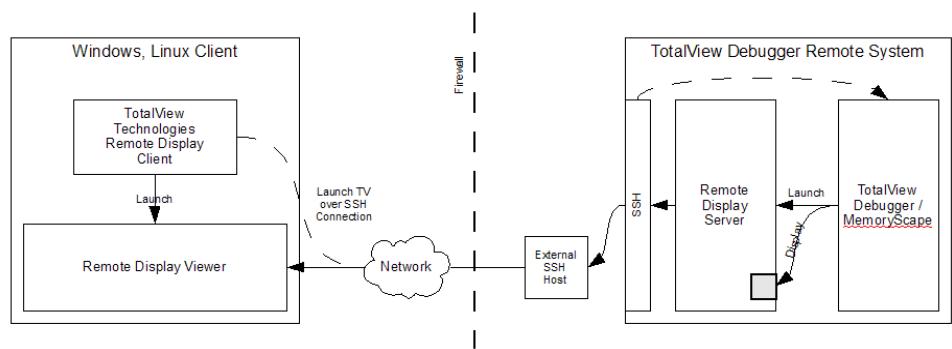
Remote Display is bundled into all TotalView releases beginning at version 8.6. However, the Client can run only on Linux x86, Linux x86-64, and Windows systems. No license is needed to run the Client, but the TotalView being run on any supported operating system must be a licensed version of TotalView 8.6 or greater.

TotalView Remote Display has three components: a Client, a Server, and a Viewer.

- The Client is a window that runs on Linux x86 and Linux x86-64 systems as well as Microsoft Windows XP and Vista.
- The Server is invisible. It manages the movement of information from the Viewer to the remote host and from the remote host back to the Client. The Server can run on all systems that TotalView supports. For example, you can run the Client on a Windows system and set up a Viewer environment on an IBM RS/6000 machine.
- The Viewer is a window that appears on the system upon which you are running the Client. All interactions between this window and the system running TotalView are handled by the Server.

Figure 35 shows how these components interact.

Figure 35: Remote Display Components



In this figure, the two large boxes represent the computer upon which you execute the Client and the remote system upon which TotalView will run. Notice where the Client, Viewer, and Server are located. The small box labelled External SSH Host is the gateway machine inside your network. In some cases, the Client may be inside your firewall. In others, it may be located outside of it. This figure also shows that the Server is created by TotalView or MemoryScape as it is contained within these programs and is created after the Client sends a message to TotalView or MemoryScape.

Installing the Client

Before you install the Client, TotalView must already be installed.

You can find the files used to install the client in two places.

- Remote Display Client files are within the **remote_display** subdirectory within your TotalView installation directory. Here you will find clients for Linux x86, x86-64, and Windows XP and Vista.
- Clients may also be downloaded from our web site by going to <http://www.totalviewtech.com/download.htm?Product=RemoteDisplayClient>.

Because Remote Display is built into TotalView, you do not need to have a separate license for it. Remote Display works with your product's license. If you have received an evaluation license, that license lets you use Remote Display on another system.

Installing on Linux

The installation procedure for the Client is straight-forward. The **remote_display** directory contains two tar files that are used on a linux-x86 or a linux 86-64 system.

- 1 Place a tar file within your **toolworks** installation directory if it is not already there. You can install the Client on as many Linux x86 and Linux x86-64 systems as you want as the Client is unlicensed. This means TotalView can be run from any Client and more than one person can be running Clients simultaneously. The only licensing requirement is that you have a license for the platform upon which TotalView will run. Of course, the number of users who can run TotalView simultaneously is specified in that product's license.

- 2** Type `tar xvf name_of_remote_display_file.tar`. This creates and populates a `remote_display/bin` directory.
- 3** Add the `remote_display` directory to your PATH environment variable. If you place this directory in your PATH, typing `remote_display_client.sh` invokes the Client.

Installing on Microsoft Windows

Before you install the Client, you must have installed TotalView on your Linux or UNIX system. The Client file, contained in your `remote_display` directory, is named `TVT_RDC_Installer.release_number.exe`. To use the installer:

- 1** Either copy the `exe` file to your Windows XP or Vista system or place it in a location that is directly accessible from your Windows machine.
- 2** Using File Explorer, navigate to the directory containing the installer and then double-click on the installer `exe` file. The installer responds by displaying the window shown in Figure 36.

Figure 36: Remote Display Client Setup



- 3** Click the **Next** button and follow the instructions on the displayed screen. As with many Windows applications, you are asked if the installer should create an entry in the start menu and place an icon on your desktop that, when clicked, invokes the Client. The very last screen has a check box that you should leave checked. This lets you confirm that Remote Display is installed correctly.

Sample Session

The TotalView Remote Display Client is simple to use. All you need do is fill in some information, the Client does the rest.

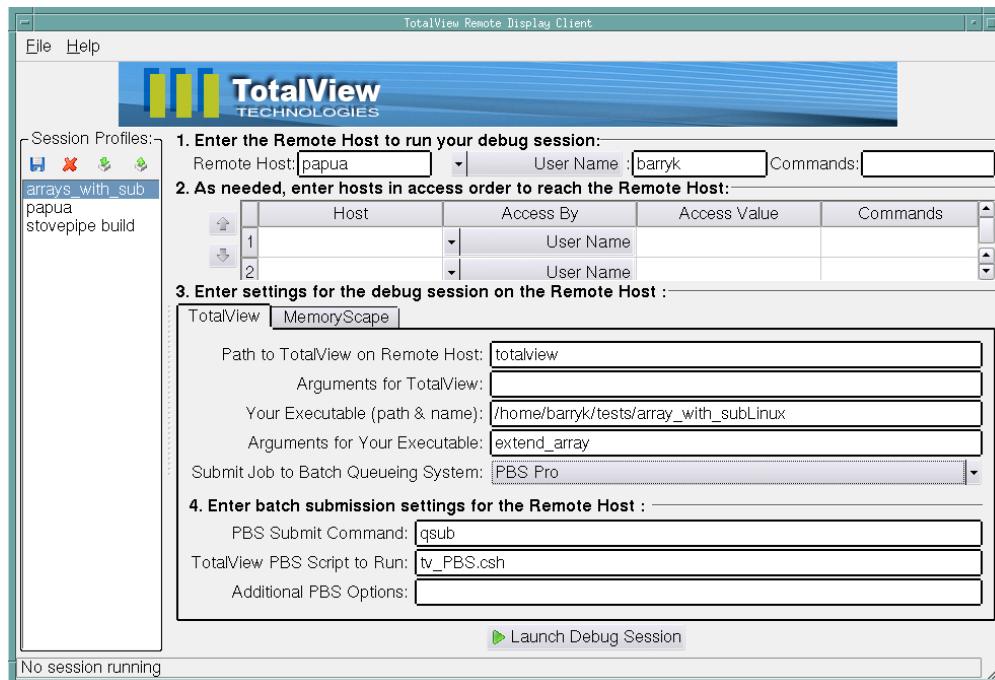
Invoke the Client by typing the following on a Linux system:

```
remote_display_client.sh
```

Using Remote Display

If you are running on Windows, either click the desktop icon or go to the TVT Remote Display item in the start menu. You'll soon see the window shown in Figure 37 on page 38.

Figure 37: Remote Display Client Window



This figure shows the **Enter batch submission settings for the Remote Host** area. This area is visible only after you select a batch system in the **Submit to Batch Queuing System** button.

If you ignore the edges of this window, there are no differences in the way the Client window displays on Linux or on Windows.

Begin by entering the following information:

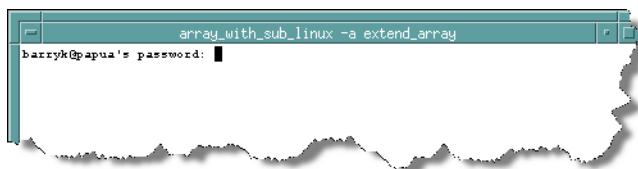
- The name of the machine upon which TotalView will execute. While the Client can only execute on Linux x86, Linux 86-64, and Windows XP and Vista systems, the remote system can be any system upon which you are licensed to run TotalView.
- Your user name, a public key file, or other **ssh** options. For more information, see "*Naming Intermediate Hosts*" on page 39.
- Commands to execute before TotalView begins.
- The directory in which TotalView resides.
- The path name of your executable. This can either be a full path name or a path name relative to your home directory. If you leave the executable text area empty, TotalView begins executing in exactly the same way as if you had typed **totalview** on the remote host. Generally, this means you need to add it to your remote host's PATH environment variable.

You can optionally add any command-line options that you would use if you were running on the remote system. You can also add any command-line options that your program requires.

If the host machine is not directly connected to the machine upon which you are running the Client, you must specify the route Remote Display will take to the remote host.

Next, press the **Launch Debug Session** button. In a moment, you'll see a window that asks for your password. (See Figure 38.)

Figure 38: Asking for Password



Depending upon the way you are connecting, you may be prompted twice for your password. The first prompt occurs when Remote Display is searching ports on a remote system. The second may or may not appear depending upon the way you access the remote host. You can often simplify logging in by using a public key file.

After you enter your remote host password, a window opens on the local Client system. This window contains TotalView as well as an xterm running on the remote host that you can use to enter operating system and other commands. If you do not add an executable name, TotalView displays its **File > New Program** dialog box. If you do enter a name, .

When you are done using TotalView, go back to the Client and terminate the Viewer and Server by pressing the **End Debug Session** button. (The **Launch Debug Session** button changes to this button after you launch the session.) A second method is to click the close button on the Viewer's window. This removes the Viewer Window but does not end the debugging session. You still need to select the Client's **End Debug Session** button. The second method might seem less useful as you still need to click the **End Debug Session** button. However, as your desktop may have many windows running on it, this can be the easier way because the Viewer often obscures the Client.

Naming Intermediate Hosts

If the Client system does not have direct access to the remote host, you must specify the path to the remote host. The order in which you enter hosts is the path Remote Display uses to reach your remote host. Buttons immediately to the left of this area let you change the order, and add, and delete lines.

Using Remote Display

The most common access method is by a user name. If this is wrong for your environment, click on the downward facing arrow in the **Access By** area. Figure 39 on page 40 shows your choices:

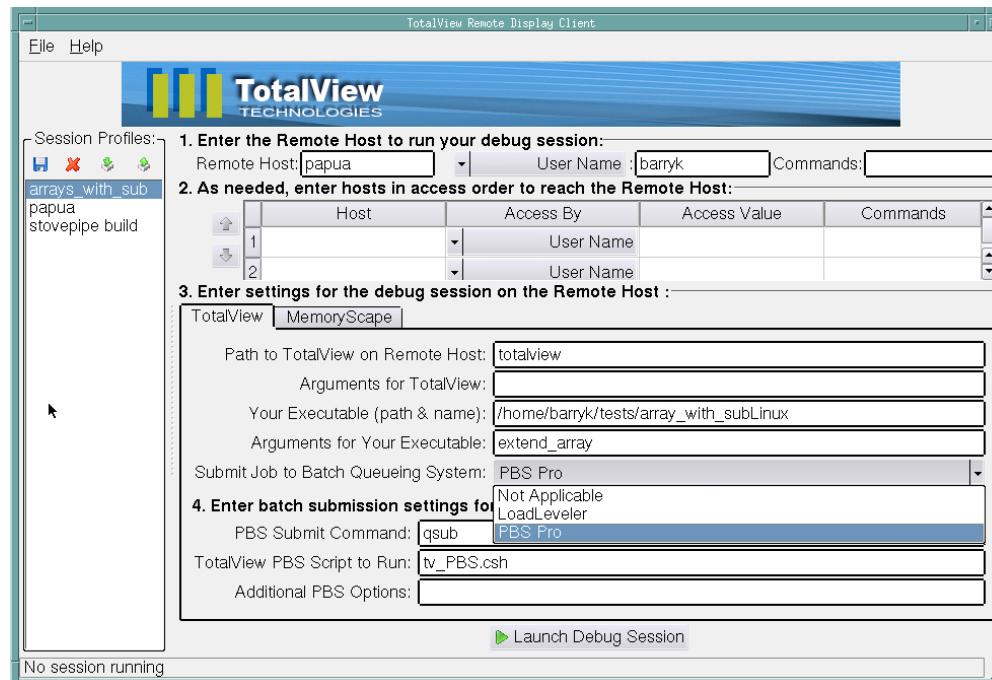
Figure 39: Access By Options

This screenshot shows the 'Access By Options' configuration window. It consists of three main sections: 1. Enter the Remote Host to run your debug session: A dropdown menu for 'Remote Host' set to 'papua', a 'User Name' field containing 'barryk', and a 'Commands' field. 2. As needed, enter hosts in access order to reach the Remote Host: A table with two rows. Row 1 has 'Host' set to '1' and 'Access By' set to 'User Name'. Row 2 has 'Host' set to '2' and 'Access By' set to 'User Name'. 3. Enter settings for the debug session: A table with two rows. Row 1 has 'Path to TotalView on Remote Host' set to 'totalview', 'Arguments for TotalView' empty, 'Your Executable (path & name)' set to '/home/barryk/tests/array_with_subLinux', and 'Arguments for Your Executable' set to 'extend_array'. Row 2 has 'Submit Job to Batch Queueing System' set to 'PBS Pro', 'Not Applicable' selected in a dropdown, and 'PBS Submit Command' set to 'qsub'. Below these sections are tabs for 'TotalView' and 'MemoryScape', with 'TotalView' currently selected.

Submitting a Job to a Batch Queuing System

TotalView Remote Display lets you submit jobs to the PBS Pro and LoadLeveler batch queuing systems. (See Figure 40.)

Figure 40: Remote Display Window: Showing Batch Options



Begin by selecting a batch system by clicking on the **Submit job to Batch Queuing** system pull down list. From this list, select either **PBS Pro** (which is the option shown) or **LoadLeveler**. Default values for the submit command and the script that Remote Display runs are filled in. The scripts for PBS Pro and Loadlever were installed when you installed TotalView.

You can, of course, change both of these values if that is what your system requires. Additional information about these scripts can be found in "Batch Scripts" on page 43.

The **Additional Options** area lets you enter arguments that are sent to the batch system. The options you add override options named in the batch script.

You're ready to launch. Do this by pressing the **Launch Debug Session** button. Behind the scenes, a job is submitted that will launch the Server and the Viewer when it reaches the head of the batch queue.

Setting Up Your Systems and Security

In order to maintain a secure environment, Remote Display uses SSH. The Remote Display Server, which runs on the remote host, allows only RFB (Remote Frame Buffer) connections from and to the remote host. No incoming access to the Server is allowed and the Server can only connect back to the Viewer over an established SSH connection. In addition, only one Viewer connection is allowed to the Server.

As Remote Display connects to systems, you are asked to enter your password. If you are allowed to use keyless ssh, you can simplify the connection process. You should check with your system administrator to confirm that this kind of connection is allowed and the ssh documentation for information on generating and storing key information.

Here are three things that must occur before the Client can connect to the remote host:

- If you use an LM_LICENSE_FILE environment variable to identify where your license is located, you must insure that this variable is read in on the remote host. This will be done automatically if the variable's definition is contained within one of the files read by the shell when Remote Display logs in.
- ssh must be available on all nonWindows systems being accessed.
- X Windows must be available on the remote system.

Session Profile Management

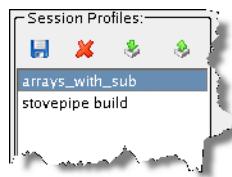
The Client lets you save the information you enter. At a later time, you can restore these settings by clicking on the profile's name in the Session Profile area.

The Client initially saves the information you first enter into a profile whose name is what you entered in the remote host area. Figure 41 on page 42 shows two saved profiles.

After you click on a profile, the Client writes this previously saved information into its text fields.

The four icons within the Session Profiles area are described in "Session Profiles Area" on page 45.

Figure 41: Session Profiles



If you make changes to the data in a text field, the Client automatically changes the information in the profile. If this is not what you want, click on the "Create" icon. The client then displays a dialog box into which you can type a new session profile name. The Client will write this existing data into a new profile instead of saving it to the original profile.

Sharing Profiles

Here's how you share a profile with others:

- 1 Select a profile.
- 2 Export it to a file.
- 3 Let others know what it is called and where it is located. They can then import the file after starting their own Remote Display setting.

Remote Display Server and Viewer

The Remote Display Server is started by the Client on the remote host. It also creates a virtual window on the remote host. The Server then sends the virtual window to the Viewer window running on your system. The Viewer is just another window running on the Client's system. You can interact with the Viewer window in the same way you interact with any window that runs directly on your system.

Behind the scenes, your interactions are sent to the Server, and the Server interacts with the virtual window running on the remote host. Changes made by this interaction are sent to the Viewer on your system. Performance depends on the load on the remote host and network latency.

The server looks for (in order) the following window managers on the remote host:

- icewm
- fvwm
- twm
- mwm

If you want to use another window manager or ensure a window manager is invoked first, use the **-wm *window_manager_name*** in the **Arguments for TotalView (or Arguments for MemoryScape)** text field. If you do this, the path of the window manager must be named in your PATH environment variable.

If you need to specify a font path on your remote host, use the **-fp *pathname*** in the **Arguments for TotalView (or Arguments for MemoryScape)** text field.

While Remote Display checks the obvious places, these places may not be obvious on some architectures.

To change the size of the Remote Display Viewer window created, use the **-geometry widthxheight** in the **Arguments for TotalView** (or **Arguments for MemoryScape**) text field. The default is 1024x768. To specify a path for the **rgb** file on the remote host, use **-co pathname** in the **Arguments for TotalView** (or **Arguments for MemoryScape**) text field. While Remote Display uses the default location, this may not be the same on architectures using a window manager on the remote host.

If you are running the Client on a Windows system, you'll see the following icons at the top of the window:

Figure 42: Local Data in a Stack Frame



From left to right, the commands associated with these icons are:

- Connection options
- Connection information
- Full Screen—this does not change the size of the Viewer window
- Request screen refresh
- Send Ctrl-Alt-Del
- Send Ctrl-Esc
- Send Ctrl key press and release
- Send Alt key press and release
- Disconnect

Batch Scripts

The actions that occur when you select PBS Pro or LoadLeveler within the **Submit job to Batch Queueing System** are defined in two files: **tv_PBS.csh** and **tv_LoadLever.csh**. If the actions defined in these scripts are not correct for your environment, you can either change one of these scripts or add a new script, which is the recommended way.

You must place the created script you create into *installation_dir/totalview_version/batch*. For example, you could place a new script file called **Run_Large.csh** into the *installation_dir/toolworks//batch* directory.

tv_PBS.csh Script

Here are the contents of the **tv_PBS.csh** script file:

```
#!/bin/csh -f
#
# Script to submit using PBS
#
# These are passed to batch scheduler:::
#
# account to be charged
```

Remote Display Commands

```
##PBS -A VEN012
#
# pass users environment to the job
##PBS -V
#
# name of the job
#PBS -N TotalView
#
# input and output are combined to standard
##PBS -o PBSPro_out.txt
##PBS -e PBSPro_err.txt
#
##PBS -l feature=xt3
#
#PBS -l walltime=1:00:00,nodes=2:ppn=1
#
#
# Do not remove the following:
TV_COMMAND
exit

#
# end of execution script
#
```

You can uncomment or change any line and you can add commands to this script. The only lines you cannot change are:

```
TV_COMMAND
exit
```

tv_LoadLeveler.csh Script

Here are the contents of the `tv_Loadleveler.csh` script file:

```
#!/bin/csh -f
# @ job_type = bluegene
#@ output = tv.out.$(jobid).$(stepid)
#@ error = tv.job.err.$(jobid).$(stepid)
#@ queue
TV_COMMAND
```

You can uncomment or change any line and you can add commands to this script. The only line you cannot change is:

```
TV_COMMAND
```

Remote Display Commands

The TotalView Remote Display Client lets you specify the information you need to launch TotalView on a remote system. After launching the debug session, the Remote Display Viewer Window appears and you can now

interact with the TotalView running on that remote server. You will find details on how this happens in "Using Remote Display" on page 35. The following sections tell you what the individual controls and commands do within the Client Window.

To see how you use the client, see "Sample Session" on page 37.

Session Profiles Area

This section of the Client window has two sections. The first has four icons. From left to right they are:

-  Creates a new profile using the current settings. If you do not use this button, the Client automatically writes the changes you've made back into the selected profile when you press the **Launch Debug Session** button at the bottom of the Client window.
-  Deletes the selected profile.
-  Imports a file. After selecting this command, the Client displays a file explorer window. Use this window to locate the profile to be loaded. After you import a file, it remains within your Client profile until you delete it.
-  Exports a file. After selecting this command, the Client displays a file explorer window. Use this window to locate a directory into which the Client will write the selected profile.

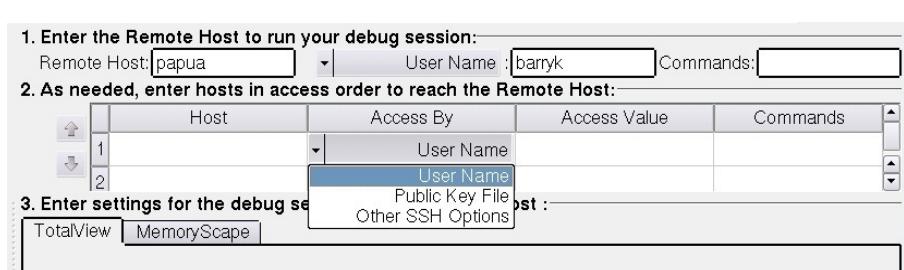
The second part names the profiles that you have either created or imported. After selecting a profile, its attributes are displayed in the right side of the window.

Remote Host and Routing Area

This top right portion of the Client window lets you:

- Name the remote host upon which TotalView will run.
- Indicate the method you will use to access the remote host. You can access it by providing your user name on the remote host, a public key file, or ssh options.
- Optionally select intermediate hosts (if they are needed), the method used to access the intermediate host, and the value needed to use this method.

Figure 43: Access By Options



1. Enter the Remote Host to run your debug session:

Remote Host: User Name: Commands:

2. As needed, enter hosts in access order to reach the Remote Host:

Host	Access By	Access Value	Commands
1	User Name		
2	User Name		

3. Enter settings for the debug session:

TotalView | MemoryScape |

User Name | Public Key File | Other SSH Options |

You enter information into these fields as follows:

Remote Display Commands

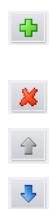
Remote Host	Type in the name of the machine upon which TotalView will execute. TotalView must be installed on that machine. If you do not have a direct path to the remote host, see "Host" on page 46
User Name	Type in your user name on the remote host.
Commands	Type in commands (in a comma-separated list) to execute on the remote host before TotalView or MemoryScape executes
Host	If you cannot directly reach the machine upon which TotalView will execute, enter an intermediate host name of the route the Client will take, the Access By method used to reach this remote host, and the information required by the Access By entry. If your network has a gateway machine, you would name it here in addition to other systems in the path to the remote host.
Commands	In the column following the Access Type, type in commands (in a comma-separated list) to execute when connected to the remote host, before connecting to the next host.

Access By and Access Value

Select one of these options:

- (1) **User Name** is what you enter into a shell command such as ssh to login to the host machine. You would enter this name in the **Access Value** field.
- (2) **Public Key File** tells the client that access information is entered into file entered into the Access Value field.
- (3) **Other SSH Options** will contain the ssh arguments needed to access the intermediate host. These are the same arguments you normally add to the ssh command.

Route management controls



- Adds an additional row into which you can enter a **Host**, **Access By** method, and an **Access Value**.
- Deletes a row from the table.
- Moves the selected row up one row.
- Moves the selected row down one row.

Product Area

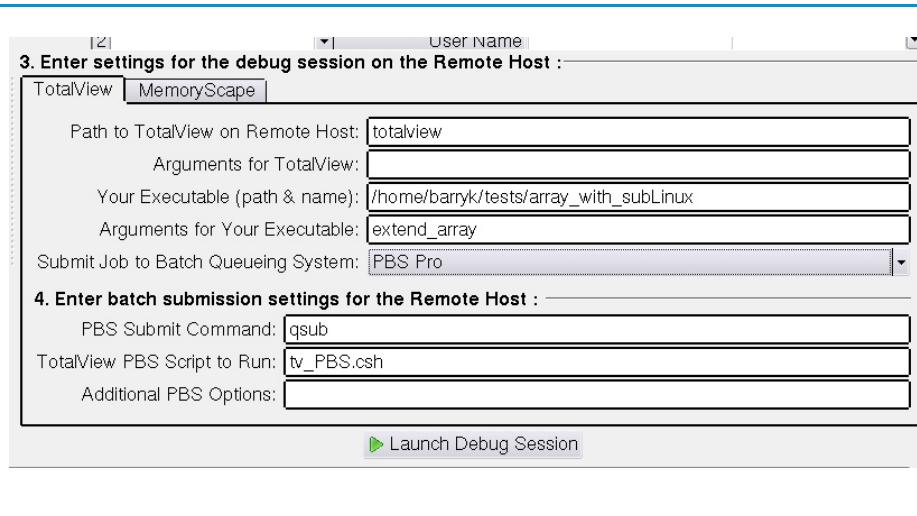
The product area within the Client window is where you enter information needed to start TotalView on the remote host.

TotalView Directory on Remote Host

Enter the directory name in which the executable resides. If you have changed the executable's pathname, you'll need to specify the new pathname.

Arguments for ... Enter TotalView command-line options.

Figure 44: Remote Host Information Area



Your Executable (path & name)

Enter either a complete or relative pathname to the program being debugged. "Relative" means relative to your home directory.

Arguments for Your Executable

If your program needs arguments, enter them in this field. If you were executing TotalView directly, these are the arguments that follow the `-a` command-line option.

Submit job to Batch Queuing System

This pull down list shows the queuing systems that you can select. This will be explained in the "Using the Submit to Batch Queuing System Entries" on page 47 topic.

Figure 45: Choosing a Batch Queuing System



Launch Debug Session

Pressing this button starts the process of creating the Remote Display Viewer. Typically, you will see an xterm window into which you can type passwords for each of the systems. All connections in the route are made using ssh.

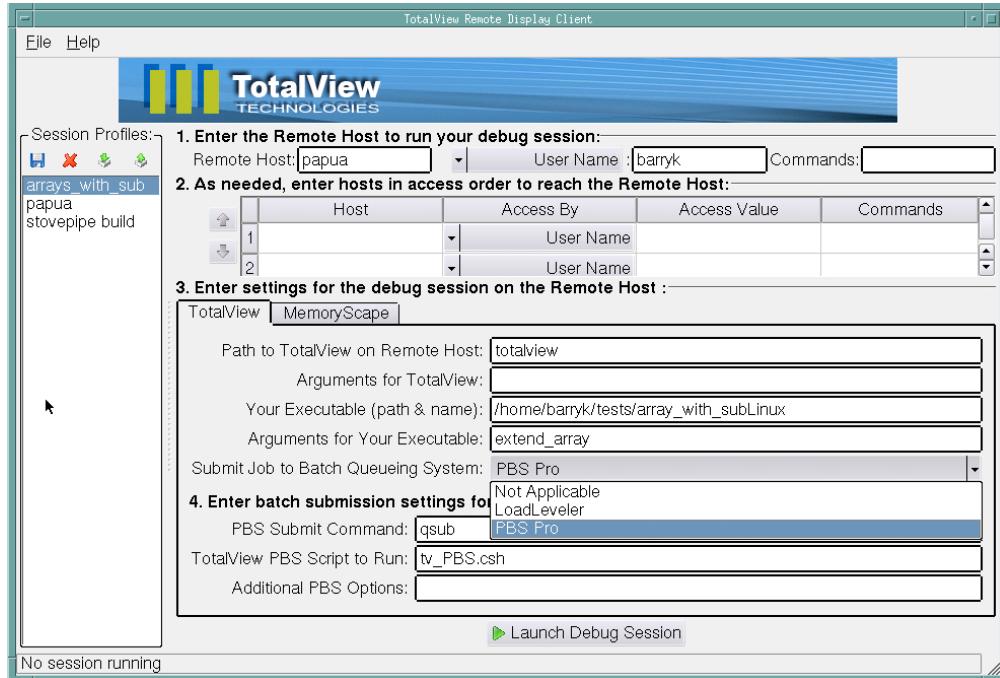
Using the Submit to Batch Queuing System Entries

If you are sending the debugging job to a batch queuing system, you need to select an entry in the **Submit job to Batch Queuing System** pull down list. After selecting an entry, the lower right portion adds a section into which

Remote Display Commands

you can enter information that is needed by the batch system. TFigure 46 on page 48 shows these changes:

Figure 46: Remote Display Client: Showing Batch Options



The items unique to this area of this area are the **PBS** or **LoadLeveler Submit** Command. The default values are **qsub** for PBS Pro and **lsubmit** for LoadLeveler.

PBS or LoadLeveler Script to run

The default values are **tv_PBS.csh** for PBS Pro and **tv_LoadLeveler.csh** for LoadLeveler. For more information, see the "Batch Scripts" on page 43 topic.

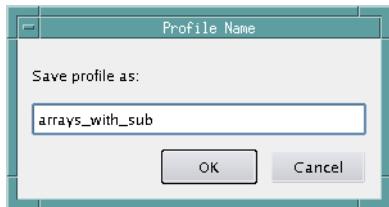
Additional PBS or LoadLeveler Options

If you need additional command-line options to either PBS or LoadLeveler, enter them here. What you enter here overrides the same setting in the script being run. For more information, see the "Batch Scripts" on page 43 topic.

File > Profile > Save

After selecting this command, the Client asks that you name the profile information it is about to create. The values contained within this profile are those currently being displayed in the Client window. After pressing OK, the Client saves the profile and places its name in the **Session Profiles** area. You do not need to save changes you make to the current profile as the Client automatically saves them. (See Figure 47 on page 49.)

Figure 47: Saving a Profile

**File > Profile > Delete**

Use this command to delete the currently selected profile. You need to confirm that you really want to delete the profile. (See Figure 48 on page 49.)

Figure 48: Deleting a Profile

**File > Profile > Import**

Select this command to tell the Client that it should import a profile previously written into a file. The Client responds by displaying a file explorer window so that you can select the profile being imported.

File > Profile > Export

Select this command to tell the Client that it should export the selected profile. The Client responds by displaying a file explorer window so that you can select a directory into which the Client writes the profile.

File > Exit

Select this command to shut down the Client Window. If sessions are open, the Client displays the following question:

Figure 49: Exit Dialog Box



Setting Up a Debugging Session



CHAPTER 4

This chapter explains how to set up a TotalView session. It also describes some of the most-used set-up commands and procedures. For information on setting up remote debugging, see Chapter 5, “Setting Up Remote Debugging Sessions,” on page 81. For information on setting up parallel debugging sessions, see Chapter 6, “Setting Up MPI Debugging Sessions,” on page 97 and Chapter 7, “Setting Up Parallel Debugging Sessions,” on page 131.

This chapter contains the following sections:

- “Compiling Programs” on page 51
- “Starting TotalView” on page 53
- “Exiting from TotalView” on page 58
- “Loading Programs” on page 58
- “Attaching to Processes” on page 61
- “Detaching from Processes” on page 62
- “Examining Core Files” on page 63
- “Viewing Process and Thread States” on page 66
- “Handling Signals” on page 69
- “Setting Search Paths” on page 71
- “Setting Preferences” on page 74

Compiling Programs

[Compilers and Platforms](#)
[Linking with the dbfork Library](#)

The first step in getting a program ready for debugging is to add your compiler's **-g** debugging command-line option. This option tells your compiler to generate symbol table debugging information; for example:

```
cc -g -o executable source_program
```

You can also debug programs that you did not compile using the **-g** option, or programs for which you do not have source code. For more information, see "Viewing the Assembler Version of Your Code" on page 171.

The following table presents some general considerations "Compilers and Platforms" in the *TotalView Reference Guide* contains additional considerations.

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually -g)	Generates debugging information in the symbol table.	Before debugging <i>any</i> program with TotalView.
Optimization option (usually -O)	Rearranges code to optimize your program's execution. <i><p></i> Some compilers won't let you use the -O option and the -g option at the same time. <i></p></i> <i><p></i> Even if your compiler lets you use the -O option, don't use it when debugging your program, since strange results often occur. <i></p></i>	After you finish debugging your program.
multi-process programming library (usually dbfork)	Uses special versions of the fork() and execve() system calls. <i><p></i> In some cases, you need to use the -lpthread option. <i></p></i> <i><p></i> For more information about dbfork, see "Linking with the dbfork Library" contained in the "Compilers and Platforms" Chapter of the <i>TotalView Reference Guide</i> . <i></p></i>	Before debugging a multi-process program that explicitly calls fork() or execve() . <i><p></i> See "Debugging Processes That Call the fork() Function" on page 361 and "Debugging Processes that Call the execve() Function" on page 361. <i></p></i>

Using File Extensions

When TotalView opens a file, it uses the file's extension to determine which programming language you used. If you are using an unusual extension, you can manually associate your extension with a programming language by setting the **TV::suffixes** variable in a startup file. For more information, see the "TotalView Variables" chapter in the *TotalView Reference Guide*.

Your installation may have its own guidelines for compiling programs. Your site administrator may have made a [..../Other/](#)

[customer_compiling.html](#) " target="_top">link to information located on your site.

Starting TotalView

[Setting Up Remote Debugging Sessions](#)

[Setting Up Parallel Debugging Sessions](#)

[Initializing TotalView](#)

[Loading Programs](#)

[Attaching to Processes](#)

[Examining Core Files](#)

[TotalView Command-Line Options](#)

TotalView can debug programs that run in many different computing environments and which use many different parallel processing modes and systems. This section looks at few of the ways you can start TotalView. See the “*TotalView Command Syntax*” chapter in the *TotalView Reference Guide* for more detailed information.

In most cases, the command for starting TotalView looks like the following:

totalview [executable [corefiles]] [options]

where *executable* is the name of the executable file to debug and *corefile* is the name of the core file to examine.

CLI: totalviewcli [executable [corefiles]] [options]

Your environment may require you to start TotalView in another way. For example, if you are debugging an MPI program, you must invoke TotalView on **mpirun**. For details, see Chapter 7, “*Setting Up Parallel Debugging Sessions*,” on page 131.

You can use the GUI and the CLI at the same time. Use the **Tools > Command Line** command to display the CLI’s window.

Your installation may have its own procedures and guidelines for running TotalView. Your site administrator may have made a [customer_running_tv.html](#) " target="_top">link to information located on your site.

The following examples show different ways of that you might begin debugging a program:

Starting TotalView

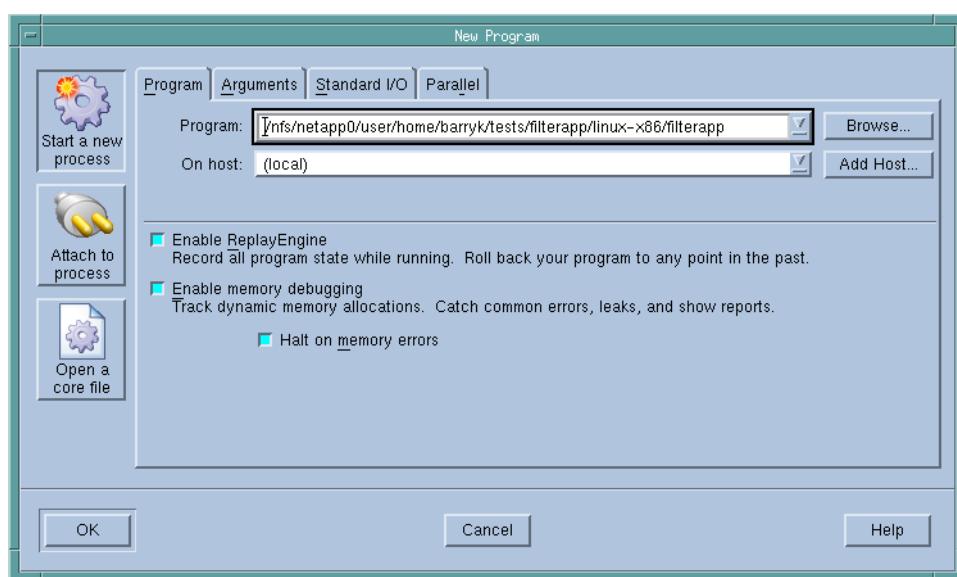
totalview Starts TotalView without loading a program or core file. Instead, TotalView displays its **File > New Program** dialog box. You can now fill in information that allows

Starting TotalView

TotalView to load your program. (See Figure 50 on page 54.)

CLI: `totalviewcli then dload executable`

Figure 50: File > New Program Dialog Box



Notice the two checkboxes on the Program tab. These checkboxes let you enable memory debugging and enable notifications—notifications tell TotalView to stop executing when memory events occur.

Starting on Mac OS X

If you installed TotalView on a Macintosh using the application bundle, you can click on the TotalView icon. If you've installed the **.dmg** version, you can start TotalView from an **xterm** by typing:

`install_path/TotalView.app/totalview`

where `install_path` is where TotalView is installed.

If the way TotalView was installed on your system was not installed without **procmod** permission, you will not be able to debug programs. If TotalView detects this problem, it displays a dialog box containing information describing what you should do.

Debugging a program

`totalview executable`

Starts TotalView and loads the *executable* program.

CLI: `totalviewcli executable`

If you installed TotalView on a Macintosh using the application bundle, you can drag your program's executable to the TotalView icon on your desktop.

Debugging a core file**Passing arguments to the program being debugged****Debugging a program that runs on another computer**

If you type an executable name, TotalView remembers that name and many of its arguments.

totalview executable corefiles

Starts TotalView and loads the *executable* program and the *corefile* core file.

CLI: `dattach -c corefiles -e executable`

The *corefiles* argument lets you name more than one core file that is associated with the same executable. In addition, you can use wild cards in the core file name.

totalview executable -a args

Starts TotalView and passes all the arguments following the **-a** option to the *executable* program. When you use the **-a** option, you must enter it as the last TotalView option on the command line.

CLI: `totalviewcli executable -a args`

If you don't use the **-a** option and you want to add arguments after TotalView loads your program, either add them using the Arguments tab within the **File > New Program** dialog box or use the **Process > Startup** command.

CLI: `dset ARGS_DEFAULT {value}`

totalview executable -remote hostname_or_address[:port]

Starts TotalView on your local host and the **tvdsrv** on a remote host. After TotalView begins executing, it loads the program specified by *executable* for remote debugging. You can specify a host name or a TCP/IP address. If you need to, you can also enter the TCP/IP port number.

CLI: `totalviewcli executable -r hostname_or_address[:port]`

Setting Up Remote Debugging Sessions

If TotalView fails to automatically load a remote executable, you may need to disable *autolaunching* for this connection and manually start the **tvdsrv**. (*Autolaunching* is the process of automatically launching **tvdsrv** processes.) You can disable autolaunching by adding the *hostname:portnumber* suffix to the name you type in the **Host** field of the **File > New Program** dialog box. As always, the *portnumber* is the TCP/IP port number on which TotalView server is communicating with TotalView. See "Setting Up and Starting the TotalView Server" on page 81 for more information.



Debugging an MPI Program

Using gnu_debuglink Files

TotalView Individual does not allow remote debugging.

totalview executable

(method 1) In many cases, you can start an MPI program in much the same way as you would start any other program. However, you will need to select the **Parallel** tab within the **File > New Programs** dialog box, and select the MPI version in addition to other options.

mpirun -np count -tv executable

(method 2) The MPI **mpirun** command starts the TotalView pointed to by the **TOTALVIEW** environment variable. TotalView then starts your program. This program will run using **count** processes.

totalview executable

If you have prepared a **gnu_debuglink** file, TotalView can access this information. For more information, see "Using **gnu_debuglink** Files" within the *Compilers and Platforms* chapter of the *TotalView Reference Guide*.

Here's where you can find more information:

- Debugging parallel programs such as MPI, PVM, or UPC, see Chapter 7, "Setting Up Parallel Debugging Sessions," on page 131.
- The **totalview** command, see "TotalView Command Syntax" in the *TotalView Reference Guide*.
- Remote debugging, see "Setting Up and Starting the TotalView Server" on page 81 and "TotalView Debugger Server (tvdsrv) Command Syntax" in the *TotalView Reference Guide*.

Initializing TotalView

Starting TotalView

TotalView Variables

Saving Action Points to a File

When TotalView begins executing, it reads initialization and startup information from a number of files. The two most common are initialization files that you create and preference files that TotalView creates.

An *initialization file* is a place where you can store CLI functions, set variables, and execute actions. TotalView interprets the information in this file whenever it begins executing. This file, which you must name **tvdr**, resides in the **.totalview** subdirectory contained in your home directory. TotalView creates this directory for you the first time it executes.

TotalView can actually read more than one initialization file. You can place these files in your installation directory, the **.totalview** subdirectory, the directory in which you invoke TotalView, or the directory in which the program resides. If an initialization file is present in one or all of these places, TotalView reads and executes each. Only the initialization file in your **.totalview** directory has the name **tvdr**. The other initialization files have the name **.tvdr**. That is, a dot precedes the file name.



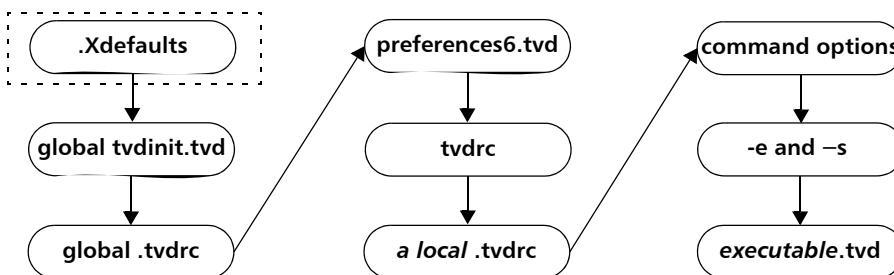
Before Version 6.0, you placed your personal `.tvdr`c file in your home directory. If you do not move this file to the `.totalview` directory, TotalView will still find it. However, if you also have a `tvdr`c file in the `.totalview` directory, TotalView ignores the `.tvdr`c file in your home directory.

TotalView automatically writes your *preferences* file to your `.totalview` subdirectory. Its name is `preferences6.tvd`. Do not modify this file as TotalView overwrites it when it saves your preferences.

If you add the `-s filename` option to either the `totalview` or `totalviewcli` shell command, TotalView executes the CLI commands contained in *filename*. This startup file executes after a `tvdr`c file executes. The `-s` option lets you, for example, initialize the debugging state of your program, run the program you're debugging until it reaches some point where you're ready to begin debugging, and even create a shell command that starts the CLI.

The following figureFigure 51 shows the order in which TotalView executes initialization and startup files.

Figure 51: Startup and Initialization Sequence



The `.Xdefaults` file, which is actually read by the server when you start X Windows, is only used by the GUI. The CLI ignores it.

The `tvdinit.tvd` file resides in the TotalView `lib` directory. It contains startup macros that TotalView requires. Do not edit this file. Instead, if you want to globally set a variable or define or run a CLI macro, create a file named `.tvdr`c and place it in the TotalView `lib` directory.

As part of the initialization process, TotalView exports three environment variables into your environment: `LM_LICENSE_FILE`, `TVROOT`, and either `SHLIB_PATH` or `LD_LIBRARY_PATH`.

If you have saved an action point file to the same subdirectory as your program, TotalView automatically reads the information in this file when it loads your program.

You can also invoke scripts by naming them in the `TV::process_load_callbacks` list. For information on using this variable, see the "Variables" chapter of the *TotalView Reference Guide*.

If you are debugging multi-process programs that run on more than one computer, TotalView caches library information in the `.totalview` subdirectory. If you want to move this cache to another location, set

TV::library_cache_directory to this location. TotalView can share the files in this cache directory among users.

Exiting from TotalView

Starting TotalView

To exit from TotalView, select **File > Exit**. You can select this command in the Root, Process, and Variable Windows. After selecting this command, TotalView displays the dialog box shown in Figure 52 on page 58.[following dialog box](#):

Figure 52: **File > Exit**
Dialog Box



Select **Yes** to exit. As TotalView exits, it kills all programs and processes that it started. However, programs and processes that TotalView did not start continue to execute.



If you have a CLI window open, TotalView also closes this window. Similarly, if you type **exit** in the CLI, the CLI closes GUI windows. If you type **exit** in the CLI and you have a GUI window open, TotalView still displays this dialog box.

CLI: **exit**

If you have both the CLI and the GUI open and only want to exit from the CLI, type **Ctrl+D**.

Loading Programs

Starting TotalView

Attaching to Processes

Detaching from Processes

TotalView can debug programs on local and remote hosts, and programs that you access over networks and serial lines. The **File > New Program**

command, which is located in the Root and Process Windows, loads local and remote programs, core files, and processes that are already running.

When entering a program's name, you can use a full or relative path name in the **Executable** and **Core File** fields. If you enter a file name, TotalView searches for it in the list of directories named using the **File > Search Path** command or listed in your **PATH** environment variable.

CLI: dset EXECUTABLE_PATH



Your license limits the number of programs you can load. For example, TotalView Individual limits you to sixteen processes or threads.

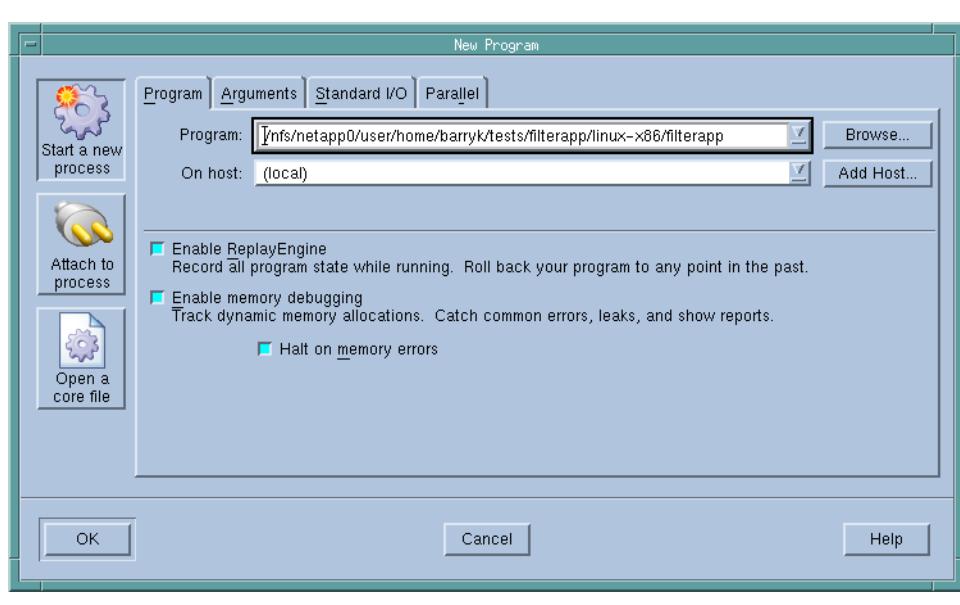
Loading Programs Using the GUI

The Program page within the **File > New Program** dialog box lets you load programs in three ways by selecting commands in the top left area of the **Program** tab. You can:

■ Start a new process

Type the path name in the **Program** area within the **Program** tab. (See Figure 53.)

Figure 53: Start a New Process



You can either type the name of your program in the **Program** area or press the **Browse** button to look for the file. Because TotalView remembers programs you've previously debugged, you may find your program in this pulldown list.

If the program is to be executed on another computer, you can name that computer by selecting the computer's name in the host area. If that computer isn't on the displayed list, select **Add host** (which is an entry in the list on the right) and enter its name in the displayed dialog box.

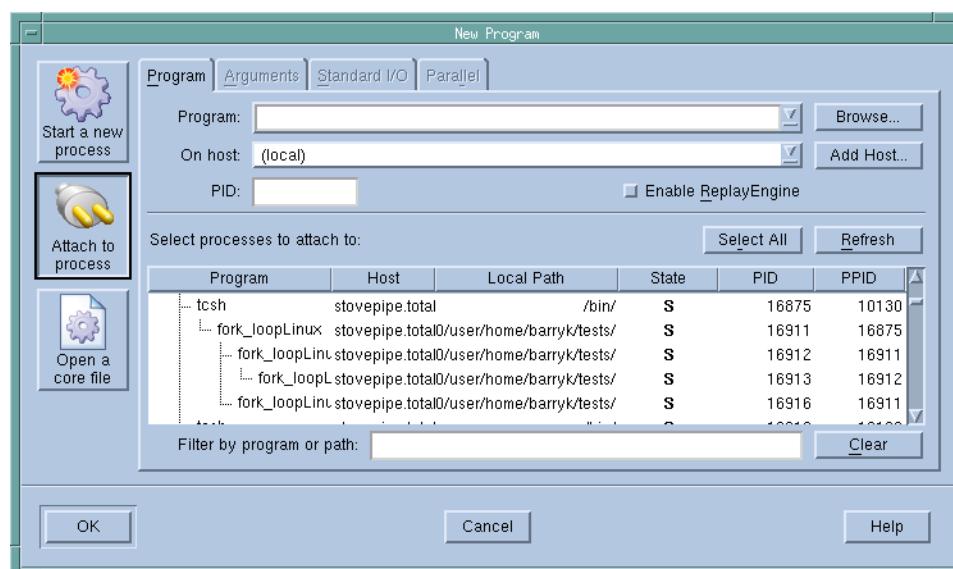


If TotalView supports your program's parallel process runtime library (for example, MPI, PVM, or UPC), it automatically connects to remote hosts. For more information, see Chapter 7, "Setting Up Parallel Debugging Sessions," on page 131.

■ Attach to process

Selecting this entry from the pulldown list on the left adds a list of processes running on a machine to which you can attach. (See Figure 54 on page 60.)

Figure 54: Attach to an Existing Process



This new information at the bottom of the window is a list of the processes executing on the selected host. To attach to a program, simply click on the program's name, then press the **OK** button.

You can select processes on other hosts by changing the information in the **on host** area.



You cannot attach to processes running on another host if you are using TotalView Individual.

For more information, see "Attaching to Processes" on page 61.

■ Open a core file

Selecting this entry from the pulldown list changes the information being displayed in the dialog box so that you can open a core file. (See Figure 55.)

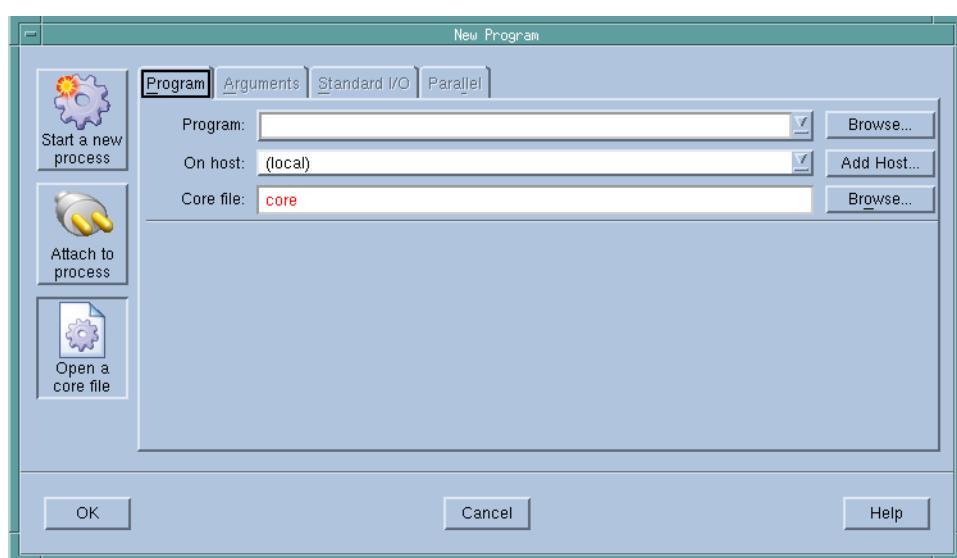
The dialog box now contains a text field and browse button that you can use to name the core file, which is usually named core (of course). TotalView also requires that you enter the name of program that dumped the core file.

For more information, see "Examining Core Files" on page 63.

Loading Programs Using the CLI

When using the CLI, you can load programs in a number of ways. Here are some of them:

Figure 55: Open a Core File



Start a new process

`dload -e executable`

Open a core file

`dattach -c corefile -e executable`

Load a program using its process ID

`dattach executable pid`

Load a program on a remote computer

`dload executable -r hostname`

You can type the computer's name (for example, `gandalf.totalviewtech.com` or an IP address).

Load a poe program

`dload -mpi POE -np 2 -nodes \
-starter_args "hfile=~/my_hosts"`

Attaching to Processes

Seeing Attached Process States

Starting TotalView

Using the Root Window

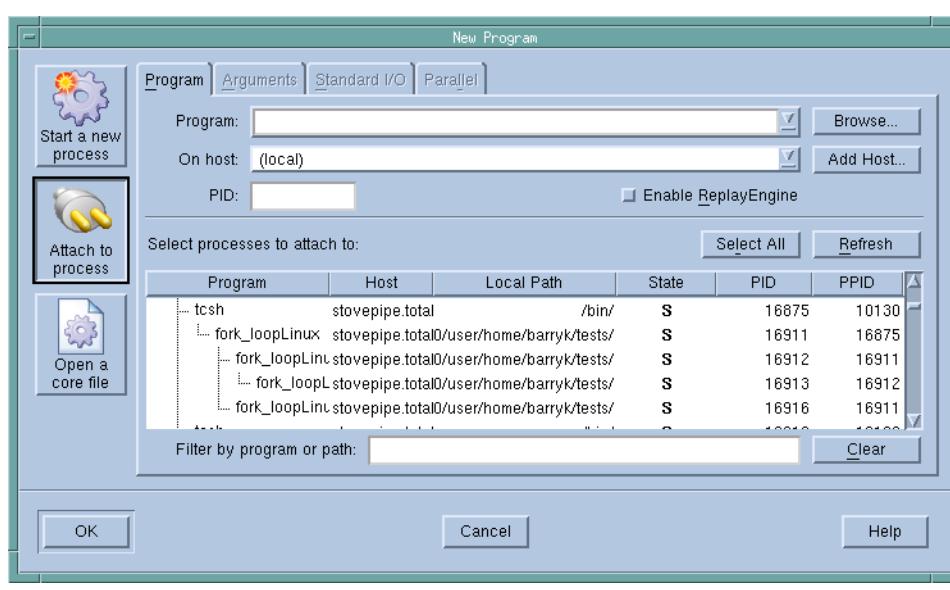
Loading Programs

If a program you're testing is hung or looping (or misbehaving in some other way), you can attach to it while it is running. TotalView lets you attach to single and multi-process programs, and these programs can be running remotely.

To attach to a process, select the **Attach to an existing process** item within the Program Page in the **File > New Program** command. (See Figure 56.)

CLI: `dattach executable pid`

Figure 56: Attaching to an existing process



When you exit from TotalView, TotalView kills all programs and processes that it started. However, programs and processes that were executing before you brought them under TotalView's control continue to execute.

While you must link programs that use `fork()` and `execve()` with the TotalView dbfork library so that TotalView can automatically attach to them when your program creates them, programs that you attach to need not be linked with this library.



You cannot attach to processes running on another host if you are using TotalView Individual.

Detaching from Processes

You can either detach from a group of processes or detach from one process.

Use the **Group > Detach** command to remove attached processes within a control group. As TotalView executes this command, it eliminates all of the state information related to these processes. If TotalView didn't start a process, it continues executing in its normal run-time environment.

[Process > Detach Command](#)

[ddetach Command](#)

[Thread > Continuation Signal Command](#)

To detach from processes that TotalView did not create:

- 1 (Optional) After opening a Process Window on the process, select the **Thread > Continuation Signal** command to display the following dialog box.(See Figure 57.)

CLI: `TV::thread.`
The examples at the end of `TV::thread` discussion show setting a signal.

Figure 57: Thread > Continuation Signal Dialog Box



Choose the signal that TotalView sends to the process when it detaches from the process. For example, to detach from a process and leave it stopped, set the continuation signal to **SIGSTOP**.

- 2 Select **OK**.
- 3 Select the **Process > Detach** command.

CLI: `ddetach`

When you detach from a process, TotalView removes all breakpoints that you have set in it.

Examining Core Files

[Loading Programs](#)

[Using the Process Window](#)

If a process encounters a serious error and dumps a core file, you can look at this file using one of the following methods:

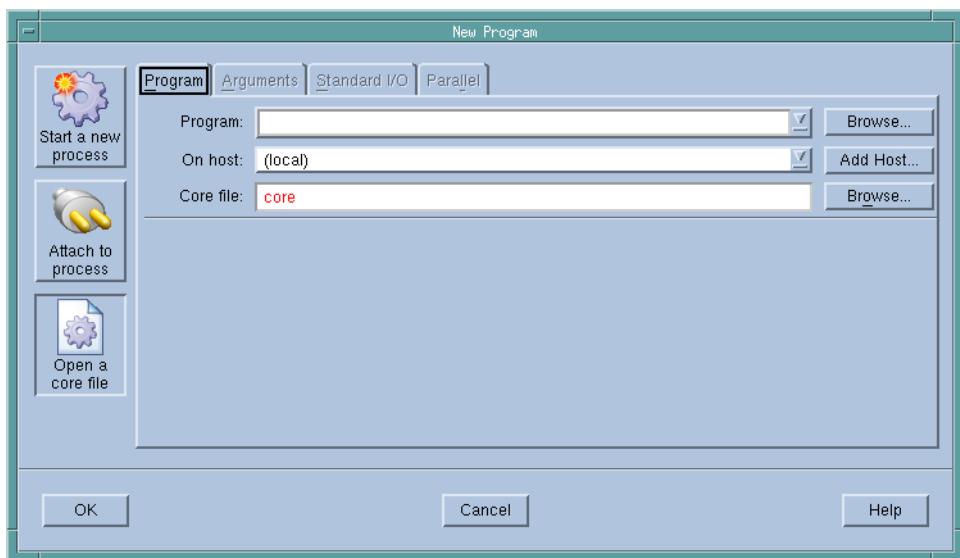
- Start TotalView as follows:
`totalview filename corefile [options]`

CLI: `totalviewcli filename corefile [options]`

- Select the **File > New Program** command from the Root Window and then select **Open a core file** from the list on the left side of the window. You will now need to type the program and core file's name (See Figure 58.) .

```
CLI: dattach -c corefile -e executable
```

Figure 58: Open a Core File



If the program and corefile reside on another system, you will, of course, need to name that system in the host area.

If your operating system can create multi-threaded core files (and most can), TotalView can examine the thread in which the problem occurred. It can also show you information about other threads in your program.

The Process Window displays the core file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when it dumped core. The title bar of the Process Window names the signal that caused the core dump. The right arrow in the line number area of the Source Pane indicates the value of the program counter (PC) when the process encountered the error.

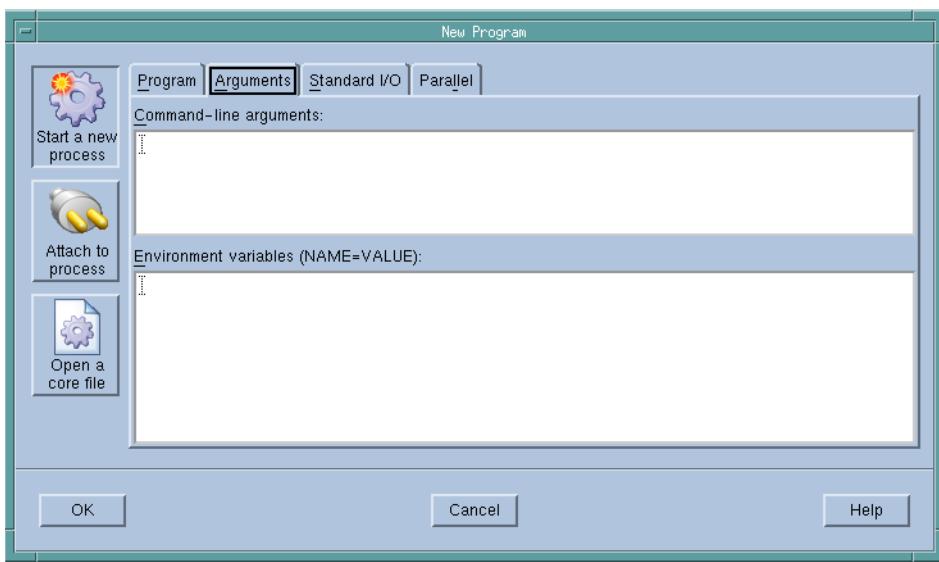
You can examine the state of all variables at the time the error occurred. Chapter 14, "Examining and Changing Data," on page 279 contains more information.

If you start a process while you're examining a core file, TotalView stops using the core file and switches to this new process.

Setting Command-line Arguments and Environment Variables

Because you are loading the program from within TotalView, you will not have entered the command-line arguments that the program needs. The Arguments page allows you to enter what ever is needed. (See Figure 59.)

Figure 59: Setting Command-Line Options and Environment Variables



Either separate each argument with a space or place each one on a separate line. If an argument contains spaces, enclose the entire argument in double-quotation marks.

When TotalView begins executing, it reads in your environment variable and, when your program begins executing, will ensure that your program has access to these variables. Use the Environment variables area to add additional environment variables or to override values of existing variables.



TotalView does not display the variables that were passed to it when you started your debugging session. Instead, this area of this tabbed page just displays the variables you added using this command.

The format for specifying an environment variable is *name=value*. For example, the following definition creates an environment variable named **DISPLAY** whose value is **enterprise:0.0**:

DISPLAY=enterprise:0.0

You can also enter this information using the Process Window's **Process > Startup Parameters** command.

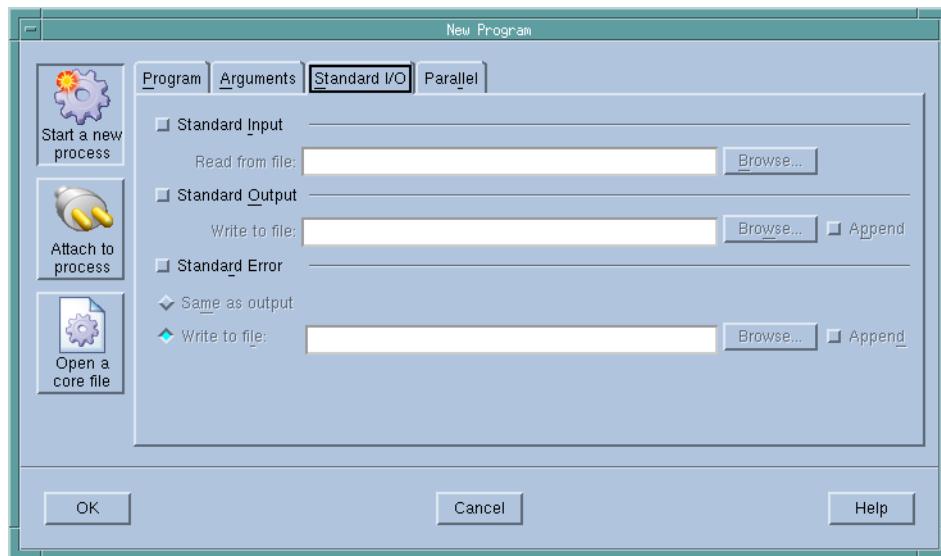
Altering Standard I/O

Use the controls within the Standard I/O page to alter standard input, output, and error. In all cases, you name the file to which TotalView will write or

Viewing Process and Thread States

from which TotalView will read information. Other controls tell TotalView that it should append to an existing file if one exists instead of overwriting it and if it should merge standard out and standard error to the same stream. (See Figure 60.)

Figure 60: Resetting Standard I/O



You can also enter this information using the Process Window's **Process > Startup Parameters** command.

Viewing Process and Thread States

[Seeing Attached Process States](#)

[Seeing Unattached Process States](#)

[Using the Root Window](#)

[Using the Process Window](#)

Process and thread states are displayed in the following:

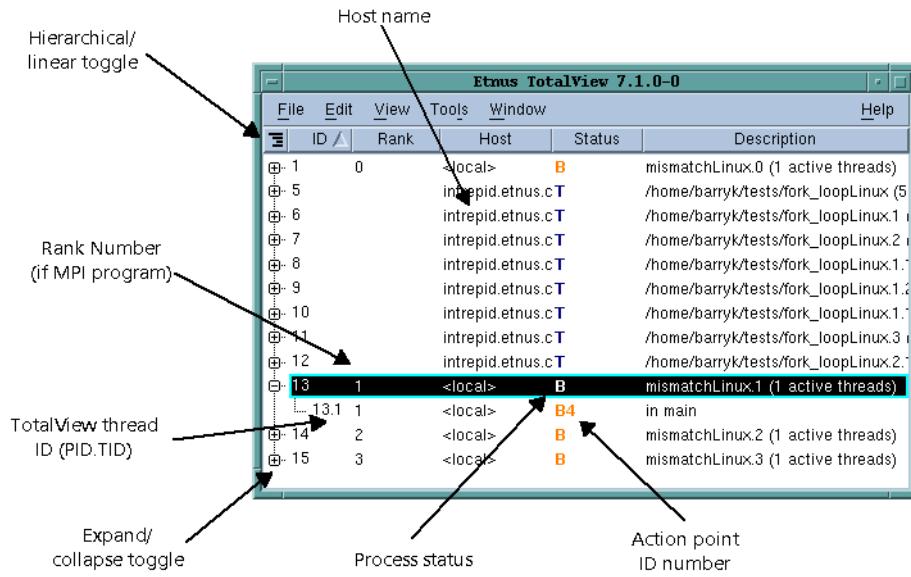
- The Root Window.
- The information within the **File > New Program** dialog box.
- The process and thread status bars of the Process Window.
- The Threads tab of the Process Window.

The following figure on the next page shows TotalView displaying process state information in the Attach to process page. (See Figure 61.)

CLI: dstatus and dptsets

When you use either of these commands, TotalView also displays state information.

Figure 61: Root Window Showing Process and Thread Status



The **Status** of a process includes the process location, the process ID, and the state of the process. (These characters are explained in "Seeing Attached Process States" on page 68.)

If you need to attach to a process that is not yet being debugged, open the **File > New Program** dialog box and select **Attach to an existing process** from the left pulldown list. TotalView will then show all processes associated with your username. Notice that some of the processes will be dim (drawn in a lighter font). This indicates either you cannot attach to the process or you're already attached to it.

Notice that the status bars in the Process Window also display status information. (See Figure 62.)

Figure 62: Process and Thread Labels in the Process Window

Process 2 (29316@intrepid.ethnus.com): fork_loopLINUX (Stopped)
Thread 2.1 (29316): (Stopped)



If the thread ID that TotalView assigns is the same as the operating system thread ID your, TotalView only displays ID. If you are debugging an MPI program, TotalView displays the thread's rank number.

Seeing Attached Process States

Seeing Unattached Process States

TotalView uses the letters shown in the following table to indicate process and thread state. (These letters are in the **Status** column in the Root Window, as the figure in the previous section shows.)

State Code	State Description
blank	Exited or never created
B	At breakpoint
E	Error reason
H	Held
K	In kernel
L	(cell only) Loose—Indicates slave SPU threads that are not held and not currently bound to PPU threads
M	Mixed
R	Running
T	Stopped reason
W	At watchpoint

The error state usually indicates that your program received a fatal signal, such as **SIGSEGV**, **SIGBUS**, or **SIGFPE**, from the operating system. See "Handling Signals" on page 69 for information on controlling how TotalView handles signals that your program receives.

CLI: The CLI prints out a word indicating the state; for example, "breakpoint."

Seeing Unattached Process States

Seeing Attached Process States

TotalView derives the state information for a process displayed in the **File > New Program** dialog box's **Attach to an existing process** state from the operating system. The state characters TotalView uses to summarize the state of an unattached process do not necessarily match those used by the operating system. The following table describes the state indicators that TotalView displays:

State Code	State Description
I	Idle
R	Running
S	Sleeping
T	Stopped
Z	Zombie (no apparent owner)

Handling Signals

[Thread > Continuation Signal Command](#)

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. The following table shows how TotalView handles UNIX signals if you do not tell it how to handle them:

Signals that TotalView Passes Back to Your Program		Signals that TotalView Treats as Errors	
SIGHUP	SIGIO	SIGILL	SIGPIPE
SIGINT	SIGIO	SIGTRAP	SIGTERM
SIGQUIT	SIGPROF	SIGIOT	SIGTSTP
SIGKILL	SIGWINCH	SIGEMT	SIGTTIN
SIGALRM	SIGLOST	SIGFPE	SIGTTOU
SIGURG	SIGUSR1	SIGBUS	SIGXCPU
SIGCONT	SIGUSR2	SIGSEGV	SIGXFSZ
SIGCHLD		SIGSYS	



TotalView uses the **SIGTRAP** and **SIGSTOP** signals internally. If a process receives either of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. You cannot alter the way TotalView uses these signals.

On some systems, hardware registers affect how TotalView and your program handle signals such as **SIGFPE**. For more information, see “*Interpreting the Status and Control Registers*” on page 250 of this manual and the “Architectures” chapter in the *TotalView Reference Guide*.



On an SGI computer, setting the **TRAP_FPE** environment variable to any value indicates that your program traps underflow errors. If you set this variable, however, you also need to use the controls in the **File > Signals** Dialog Box to indicate what TotalView should do with **SIGFPE** errors. (In most cases, you set **SIGFPE** to **Resend**.)

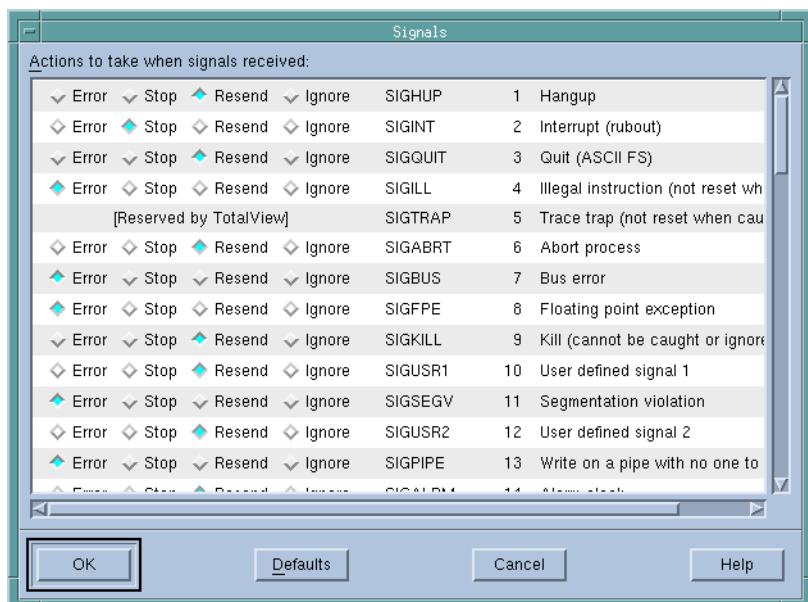
You can change the signal handling mode using the **File > Signals** command. (See Figure 63.)

CLI: dset TV::signal_handling_mode

The signal names and numbers that TotalView displays are platform-specific. That is, what you see in this box depends on the computer and operating system in which your program is executing.

You can change the default way in which TotalView handles a signal by setting the **TV::signal_handling_mode** variable in a **.tvrc** startup file. For more information, see Chapter 4 of the “*TotalView Reference Guide*.”

Figure 63: File > Signals Dialog Box



When your program receives a signal, TotalView stops all related processes. If you don't want this behavior, clear the **Stop control group on error signal** check box on the Options Page of the **File > Preferences** Dialog Box.

CLI: `dset TV::warn_step_throw`

When your program encounters an error signal, TotalView opens or raises the Process Window. Clearing the **Open process window on error signal** check box, also found on the Options Page in the **File > Preferences** Dialog Box, tells TotalView not to open or raise windows.

CLI: `dset TV::GUI::pop_on_error`

If processes in a multi-process program encounter an error, TotalView only opens a Process Window for the first process that encounters an error. (If it did it for all of them, TotalView would quickly fill up your screen with Process Windows.)

If you select the **Open process window at breakpoint** check box on the **File > Preferences** Action Points Page, TotalView opens or raises the Process Window when your program reaches a breakpoint.

CLI: `dset TV::GUI::pop_at_breakpoint`

Make your changes by selecting one of the radio buttons described in the following table.

Button	Description
Error	Stops the process, places it in the <i>error</i> state, and displays an error in the title bar of the Process Window. If you have also selected the Stop control group on error signal check box, TotalView also stops all related processes. Select this button for severe error conditions, such as SIGSEGV and SIGBUS .
Stop	Stops the process and places it in the <i>stopped</i> state. Select this button if you want TotalView to handle this signal as it would a SIGSTOP signal.
Resend	Sends the signal back to the process. This setting lets you test your program's signal handling routines. TotalView sets the SIGKILL and SIGHUP signals to Resend since most programs have handlers to handle program termination.
Ignore	Discards the signal and continues the process. The process does not know that something raised a signal.



*Do not use Ignore for fatal signals such as **SIGSEGV** and **SIGBUS**. If you do, TotalView can get caught in a signal/resignal loop with your program; the signal immediately reoccurs because the failing instruction repeatedly re-executes.*

Setting Search Paths

Starting TotalView

EXECUTABLE_PATH Variable

If your source code, executable, and object files reside in different directories, set search paths for these directories with the **File > Search Path** command. You do not need to use this command if these directories are already named in your environment's **PATH** variable.

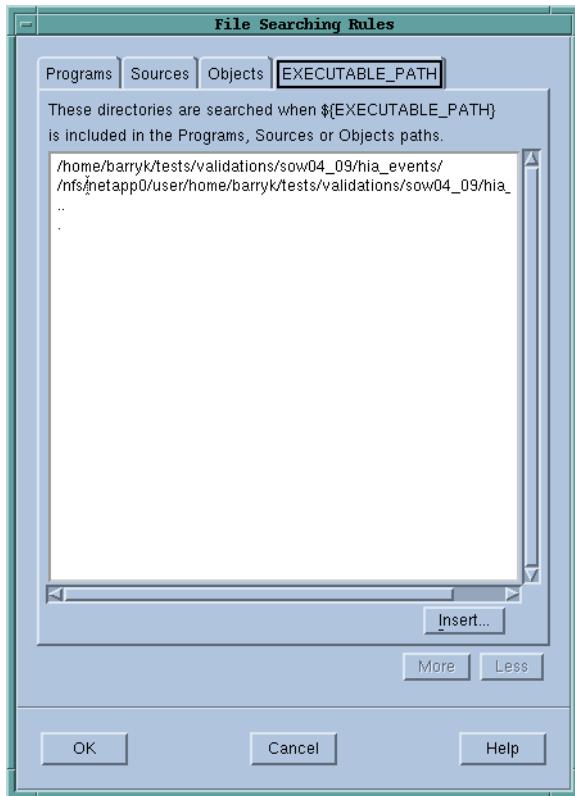
CLI: `dset EXECUTABLE_PATH`

These search paths apply to *all* processes that you're debugging. (See Figure 64 on page 72.)

TotalView searches the following directories in order:

- 1 The current working directory (.) and the directories you specify with the **File > Search Path** command, in the exact order you enter them.
- 2 The directory name hint. This is the directory that is within the debugging information generated by your compiler.
- 3 If you entered a full path name for the executable when you started TotalView, TotalView searches this directory.

Figure 64: File > Search Path Dialog Box



4 If your executable is a symbolic link, TotalView looks in the directory in which your executable actually resides for the new file.

Since you can have multiple levels of symbolic links, TotalView continues following links until it finds the actual file. After it finds the current executable, it looks in its directory for your file. If the file isn't there, TotalView backs up the chain of links until either it finds the file or determines that the file can't be found.

5 The directories specified in your **PATH** environment variable.

6 The **src** directory within your TotalView installation directory.

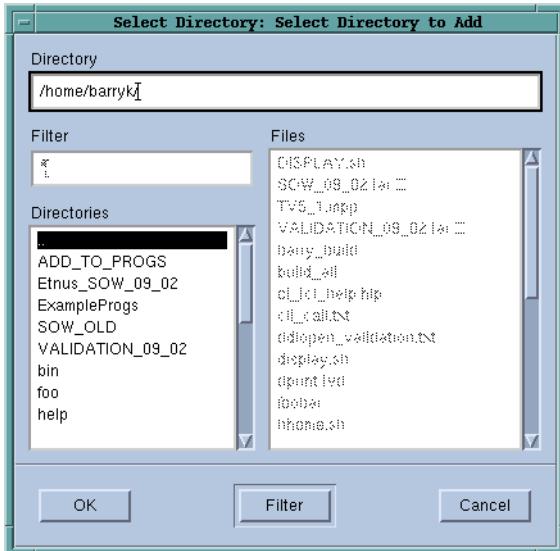
The simplest way to enter a search path is select the **EXECUTABLE_PATH** tab, then type an entry or press **Insert** and use the displayed dialog box to find the directory.

When you enter directories into this dialog box, you must enter them in the order you want them searched, and you must enter each on its own line.

You can enter directories in the following ways:

- Type path names directly.
- Cut and paste directory information.
- Click the **Insert** button to display the **Select Directory** dialog box that lets you browse through the file system, interactively selecting directories.
(See Figure 65 on page 73.)[Here is the dialog box:](#)

Figure 65: Select Directory Dialog Box



The current working directory (.) in the **File > Search Path** Dialog Box is the first directory listed in the window. TotalView interprets relative path names as being *relative* to the current working directory.

If you remove the current working directory from this list, TotalView reinserts it at the top of the list.

After you change this list of directories, TotalView again searches for the source file of the routine being displayed in the Process Window.

You can also specify search directories using the **EXECUTABLE_PATH** environment variable.

TotalView search path is not usually passed to other processes. For example, it does not affect the **PATH** of a starter process such as **poe**. Suppose that **.** is in your TotalView path, but it is not in your **PATH** environment variable. In addition, the executable named **prog_name** is listed in your **PWD** environment variable. In this case, the following command works:

```
totalview prog_name
```

However, the following command does not:

```
totalview poe -a prog_name
```

You will find a complete description of how to use this dialog box in the help.

Setting Startup Parameters

After you load a program, you may want to change a program's command-line arguments and environment variables or change the way standard input, output, and error. Do this using the **Process > Startup Parameters** command. The displayed dialog box is nearly identical to that displayed when you use the **File > New Program** command, differing in that the dialog box doesn't have a **Program** tab.

For information on this dialog box's tabs, see "*Setting Command-line Arguments and Environment Variables*" on page 65 and "*Altering Standard I/O*" on page 65.

If you are using the CLI, you can set default command line arguments by setting the **ARGS_DEFAULT** variable.

Also, the **drun** and **drerun** commands let you reset stdin, stdout, and stderr.

Setting Preferences

[File > Preferences](#)

The **File > Preferences** command lets you tailor many TotalView behaviors. This section contains an overview of these preferences. See the online Help for detailed explanations.

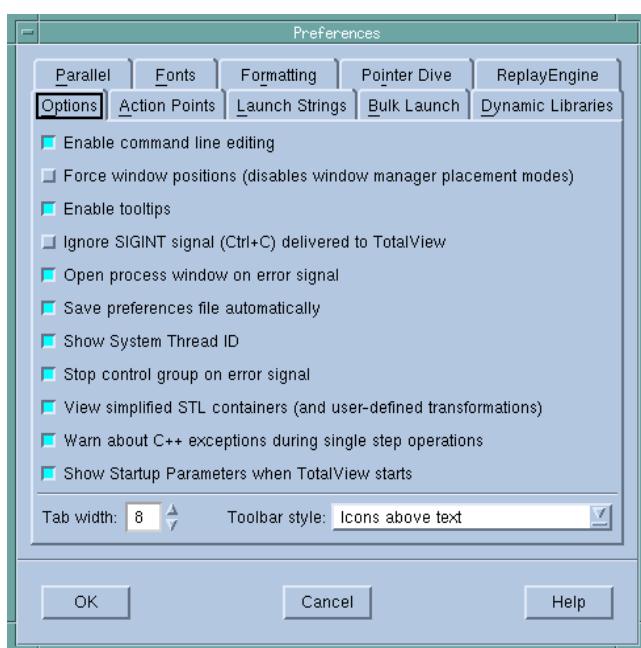
Some settings, such as the prefixes and suffixes examined before loading dynamic libraries, can differ between operating systems. If they can differ, TotalView can store a unique value for each. TotalView does this transparently, which means that you only see an operating system's values when you are running TotalView on that operating system. For example, if you set a server launch string on an SGI computer, it does not affect the value stored for other systems. Generally, this occurs for server launch strings and dynamic library paths.

Every preference has a variable that you can set using the CLI. These variables are described in the "*Variables*" chapter of the *TotalView Reference Guide*.

The rest of this section is an overview of these preferences.

Options This page contains check boxes that are either general in nature or that influence different parts of the system. See the online Help for information on using these check boxes. (See Figure 66.)

Figure 66: File > Preferences Dialog Box: Options Page

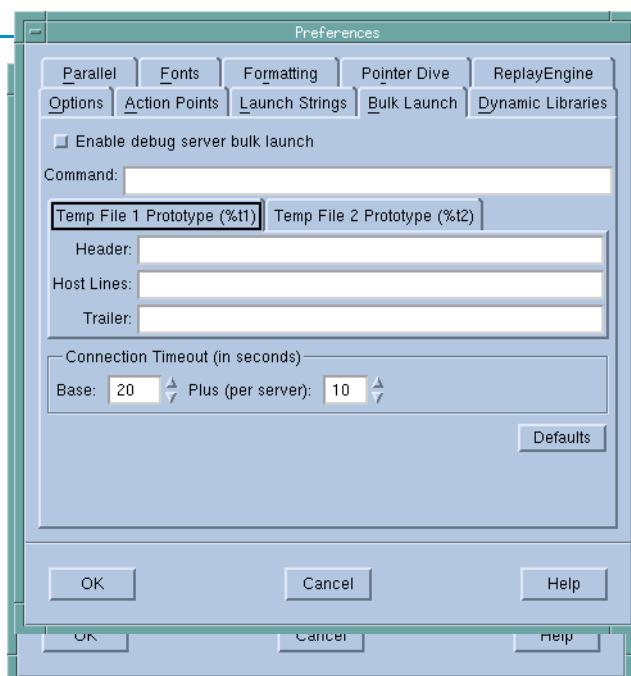


Action Points

The commands on this page indicate whether TotalView should stop anything else when it encounters an action point, the scope of the action point, automatic saving and loading of action points, and if TotalView should open a Process Window for the process encountering a breakpoint.

(See Figure 67.)

Figure 67: File > Preferences Dialog Box: Action Points Page

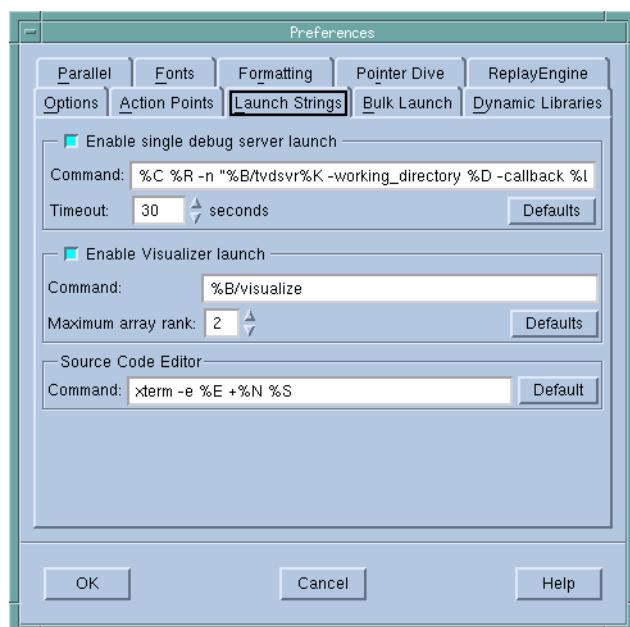


Setting Preferences

Launch Strings

The three areas of this page let you set the launch string that TotalView uses when it launches the **tvdsvr** remote debugging server, the Visualizer, and a source code editor. The values you initially see in the page are default values that TotalView supplies. (See Figure 68.)

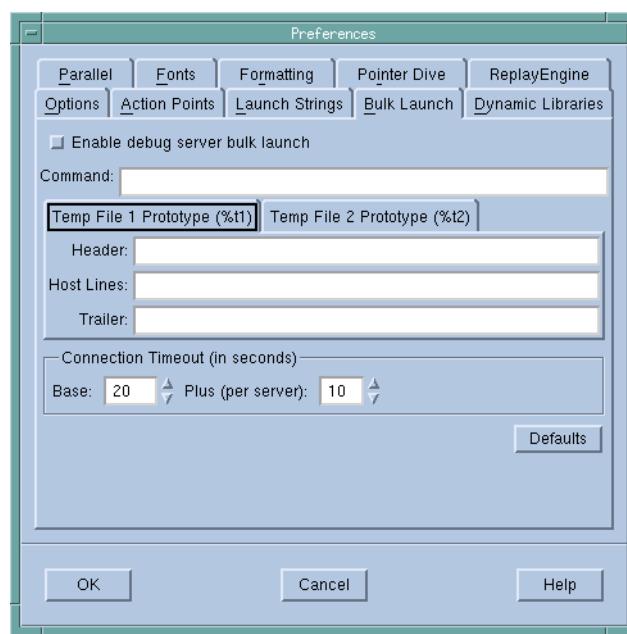
Figure 68: File > Preferences Dialog Box: Launch Strings Page



Bulk Launch

The fields and commands on this page configure the TotalView bulk launch system. (The bulk launch system launches groups of processes simultaneously.) See Chapter 4 for more information. (See Figure 69.)

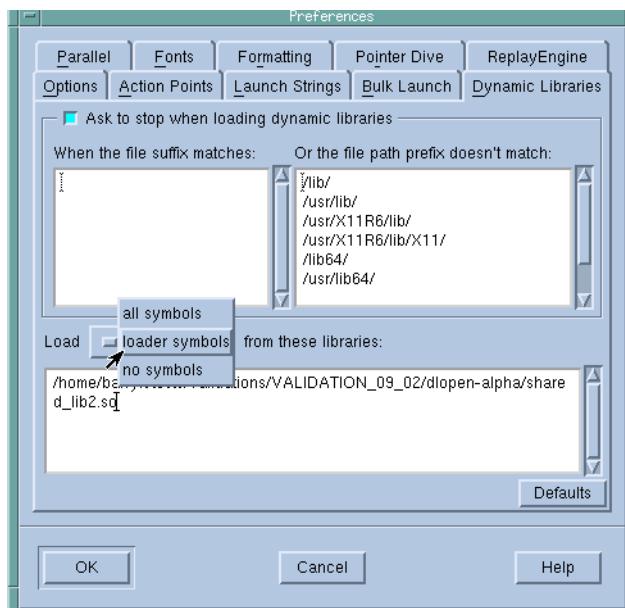
Figure 69: File > Preferences Dialog Box: Bulk Launch Page



Dynamic Libraries

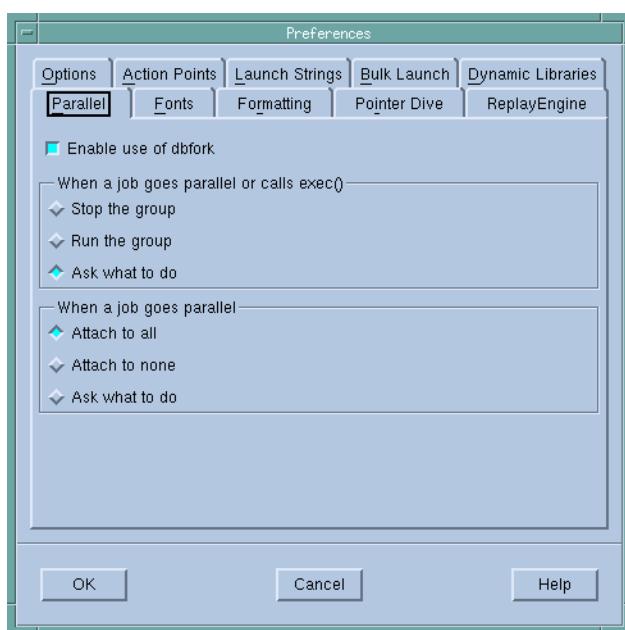
When debugging large programs, you can sometimes increase performance by telling TotalView that it should load all process debugging symbols. This page lets you control which symbols are added to TotalView when it loads a dynamic library, and how many of a library's symbols are read in. (See Figure 70.)

Figure 70: File > Preferences Dialog Box: Dynamic Libraries Page

**Parallel**

This options on this page let you control if TotalView will stop or continue executing when a processes creates a thread or goes parallel. By telling your program to stop, you can set breakpoints and examine code before execution begins. (See Figure 71.)

Figure 71: File > Preferences Dialog Box: Parallel Page

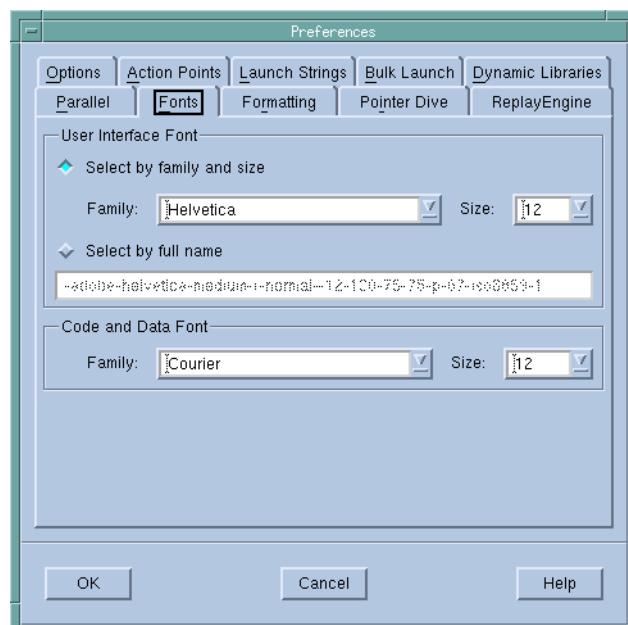


Setting Preferences

Fonts

The options on this page lets you specify the fonts TotalView uses in the user interface and how TotalView displays your code. (See Figure 72.)

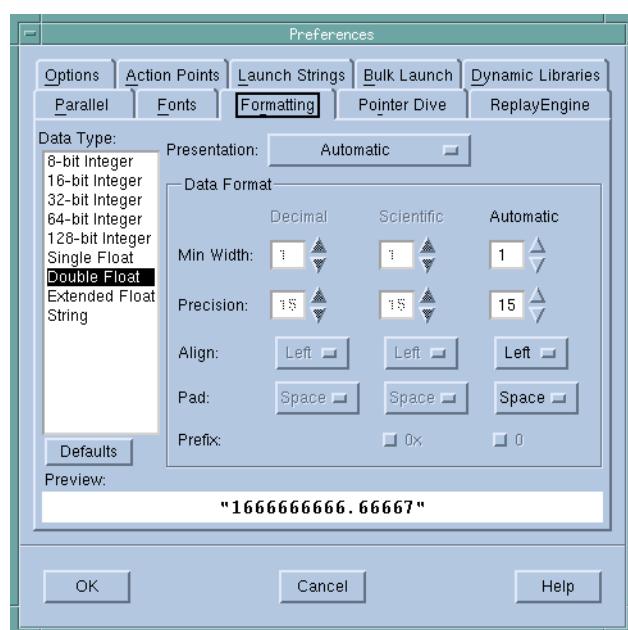
Figure 72: File > Preferences Dialog Box: Fonts Page



Formatting

The options on this page control how TotalView displays your program's variables. (See Figure 73.)

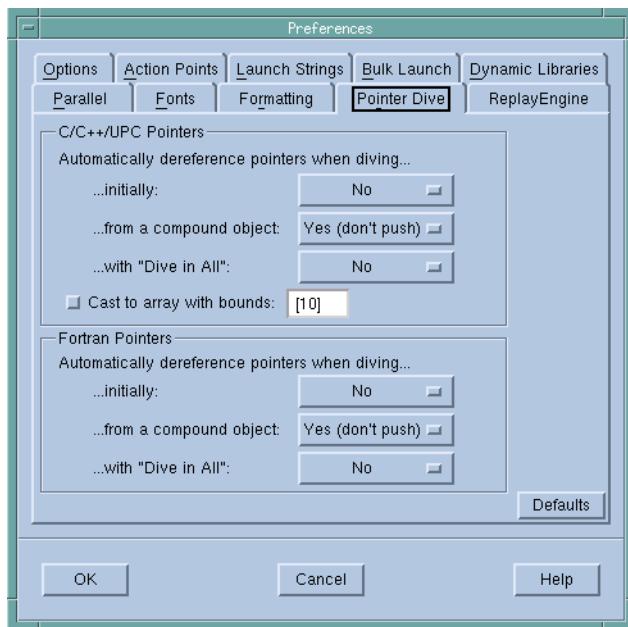
Figure 73: File > Preferences Dialog Box: Formatting Page



Pointer Dive

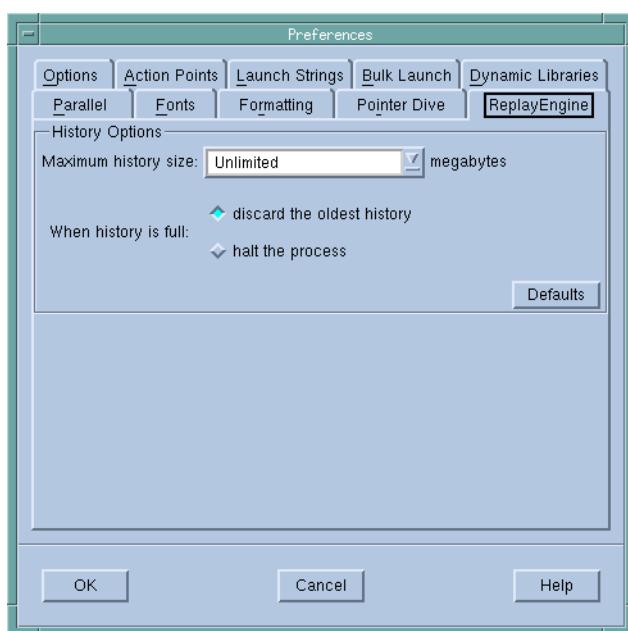
The options on this page control how TotalView dereferences pointers and how it casts pointers to arrays are cast (See Figure 74.)

Figure 74: File > Preferences Dialog Box: Pointer Dive Page

**ReplayEngine**

The options on this page control how ReplayEngine handles recorded history. (See Figure 75.)

Figure 75: File > Preferences Dialog Box: ReplayEngine



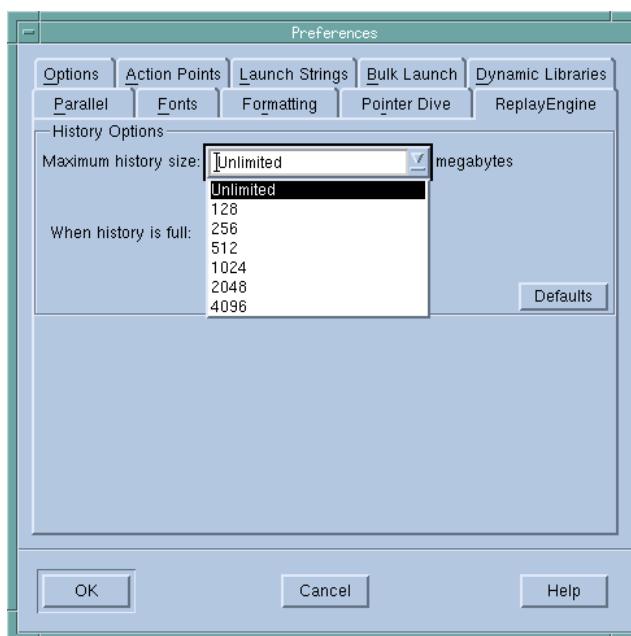
The **Maximum history size** option sets the size in megabytes for ReplayEngine's history buffer. The default value, Unlimited, means ReplayEngine will use as much memory as is available to save recorded history. You can

Setting Preferences

enter a new value into the text field or select from a drop-down list, as seen in Figure 76.

The second option on the ReplayEngine preference page defines the tool's behavior when the history buffer is full. By default, the oldest history will be discarded so that recording can continue. You can change that so that the recording process will simply stop when the buffer is full.

Figure 76: File > Preferences
Dialog Box:
ReplayEngine History
Option



Setting Preferences, Options, and X Resources

Setting Preferences

TotalView Variables

In most cases, preferences are the best way to set many features and characteristics. In some cases, you need have more control. When these situations occur, you can the preferences and other TotalView attributes using variables and command-line options.

Older versions of TotalView did not have a preference system. Instead, you needed to set values in your `.Xdefaults` file or using a command-line option. For example, setting `totalview*autoLoadBreakpoints` to true tells TotalView to automatically load an executable's breakpoint file when it loads an executable. Because you can also set this option as a preference and set it using the CLI `dset` command, this X resource has been *deprecated*.



Deprecated means that while the feature still exists in the current release, there's no guarantee that it will continue to work at a later time. We have deprecated all "totalview" X default options. TotalView still fully supports Visualizer resources. Information on these Visualizer settings is at <http://www.totalviewtech.com/Documentation/xresources/XResources.pdf>.

Similarly, documentation for earlier releases told you how to use a command-line option to tell TotalView to automatically load breakpoints, and there were two different command-line options to perform this action. While these methods still work, they are also deprecated.

In some cases, you might set a state for one session or you might override one of your preferences. (A *preference* indicates a behavior that you want to occur in all of your TotalView sessions.) This is the function of the command-line options described in “*TotalView Command Syntax*” in the *TotalView Reference Guide*.

For example, you can use the **-bg** command-line option to set the background color for debugger windows in the session just being invoked. TotalView does not remember changes to its default behavior that you make using command-line options. You have to set them again when you start a new session.

Setting Preferences

Setting Up Remote Debugging Sessions



This chapter explains how to set up TotalView remote debugging sessions.

This chapter contains the following sections:

- “*Setting Up and Starting the TotalView Server*” on page 81
- “*Starting the TotalView Server Manually*” on page 86
- “*Disabling Autolaunch*” on page 91

You cannot debug remote processes using TotalView Individual.



5

Setting Up and Starting the TotalView Server

Debugging a remote process with TotalView is only slightly different than debugging a native process. The following are the primary differences:

- TotalView needs to work with a process that will be running on remote computers. This remote process, which TotalView usually launches, is called the **tvdsrv**.
- TotalView performance depends on your network’s performance. If the network is overloaded, debugging can be slow.

TotalView can automatically launch **tvdsrv** in one of the following ways:

- It can independently launch a **tvdsrv** on each remote host. This is called *single-process server launch*.
- It can launch all remote processes at the same time. This is called *bulk server launch*.

Setting Up and Starting the TotalView Server

Because TotalView can automatically launch **tvdsrv**, you usually do not need to do anything special for programs that launch remote processes. When using TotalView, it doesn't matter whether a process is local or remote.



Some parallel programs—MPI programs, for example—make use of a starter program such as poe or mpirun. This program creates all the parallel jobs on your nodes. TotalView lets you start these programs in two ways. One requires that the starter program be under TotalView control, and the other does not. In the first case, you will enter the name of the starter program on the command line. In the other, you will enter program information into the File > New Program or Process > Startup Parameter dialog boxes. Programs started using these dialog boxes do not use the information you set for single-process and bulk server launching.

In general, when you are debugging programs remotely, the architecture of the remote machine must be compatible with that of the machine upon which you are running TotalView. For example, you cannot perform remote debugging on a 64-bit Linux system if you launch TotalView from a 32-bit Linux system. In addition, the operating systems must also be compatible.

However, TotalView supports several forms of heterogeneous debugging, where the operating system and/or architecture differ. For example, from a Linux x86-64 session you can debug remote processes on Linux Cell.

This table shows the supported combinations:

Host System	Target System
Linux x86-64	Linux x86 Linux x86-64 Linux Power 32 Linux Power 64 / Cell SiCortex Cray XT
Linux x86	Linux x86 Linux Power 32 Linux Power 64 / Cell
Linux Power 64 (including Linux Cell)	Linux Power 32 Linux Power 64 / Cell Blue Gene
SiCortex	Linux x86-64
	Linux MIPS 64

You must install TotalView for each host and target platform combination being debugged.



The path to TotalView must be identical on the local and all remote systems. If they are not, TotalView will not find the tvdsrv program.

TotalView assumes that you will launch **tvdsrv** using **rsh**. If **rsh** is unavailable, you should set the **TVDSVRLAUNCHCMD** environment variable to the command that you use to remotely access the remote system. In most cases, this will be **ssh**.



If the default single-process server launch procedure meets your needs and you're not experiencing any problems accessing remote processes from TotalView, you probably do not need the information in this chapter. If you do experience a problem launching the server, check that the `tvdsrv` process is in your path.

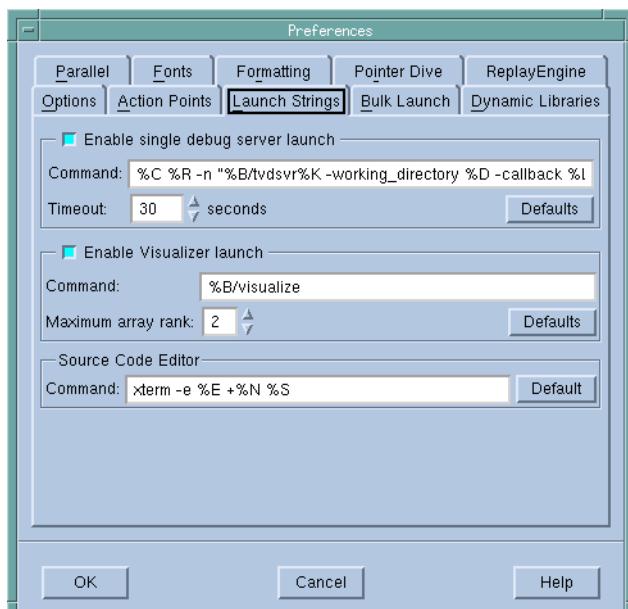
This section contains the following information:

- "Setting Single-Process Server Launch Options" on page 83
- "Setting Bulk Launch Window Options" on page 84
- "Starting the TotalView Server Manually" on page 86
- "Using the Single-Process Server Launch Command" on page 87
- "Bulk Server Launch Setting on an SGI Computer" on page 89
- "Setting Bulk Server Launch on an HP Alpha Computer" on page 90
- "Setting Bulk Server Launch on a Cray XT Series Computer" on page 90
- "Setting Bulk Server Launch on an IBM RS/6000 AIX Computer" on page 91
- "Disabling Autolaunch" on page 91
- "Changing the Remote Shell Command" on page 92
- "Changing Arguments" on page 92
- "Autolaunching Sequence" on page 93

Setting Single-Process Server Launch Options

The **Enable single debug server launch** check box in the Launch Strings Page of the **File > Preferences** Dialog Box lets you disable autolaunching, change the command that TotalView uses when it launches remote servers, and alter the amount of time TotalView waits when establishing connections to a `tvdsrv` process. (The **Enable Visualizer launch** and **Source Code Editor** areas are not used when setting launch options.) See Figure 75.

Figure 75: File > Preferences:
Launch Strings Page



Enable single debug server launch

If you select this check box, TotalView independently launches the **tvdsrv** on each remote system.

CLI: `dset TV::server_launch_enabled`



Even if you have enabled bulk server launch, you probably also want to enable this option. TotalView uses this launch string after you start TotalView and when you name a host in the File > New Program Dialog Box or have used the –remote command-line option. You only want to disable single server launch when it can't work.

Command

Enter the command that TotalView will use when it independently launches **tvdsrv**. For information on this command and its options, see “*Using the Single-Process Server Launch Command*” on page 87.

CLI: `dset TV::server_launch_string`

Timeout

After TotalView automatically launches the **tvdsrv** process, it waits 30 seconds for it to respond. If the connection isn’t made in this time, TotalView times out. You can change the length of time by entering a value from 1 to 3600 seconds (1 hour).

CLI: `dset TV::server_launch_timeout`

If you notice that TotalView fails to launch **tvdsrv** (as shown in the **xterm** window from which you started TotalView) before the timeout expires, click **Yes** in the **Question** Dialog Box that appears.

Defaults

If you make a mistake or decide you want to go back to default settings, click the **Defaults** button.

Clicking the **Defaults** button also discards all changes you made using a CLI variable. TotalView doesn’t immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Setting Bulk Launch Window Options

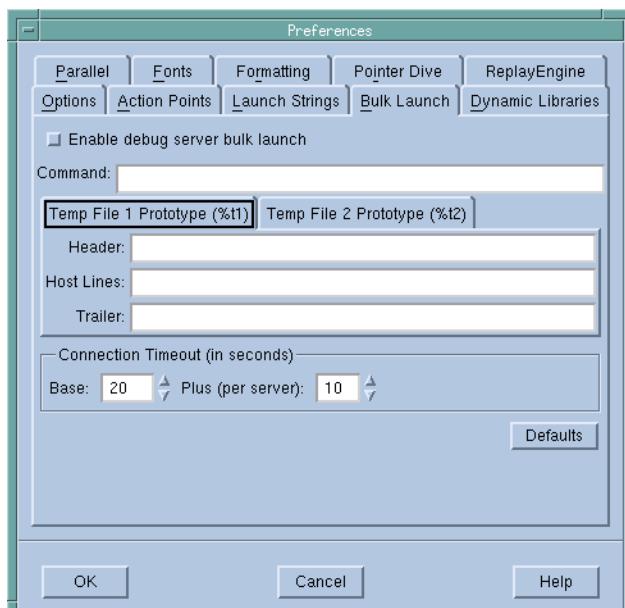
The fields in the **File > Preferences** Bulk Launch Page let you change the bulk launch command, disable bulk launch, and alter connection timeouts that TotalView uses when it launches **tvdsrv** programs. (See Figure 76.)

CLI: `dset TV::bulk_launch_enabled`

Enable debug server bulk launch

If you select this check box, TotalView uses its bulk launch procedure when launching the **tvdsrv**. By de-

Figure 76: File > Preferences:
Bulk Launch Page



fault, bulk launch is disabled; that is, TotalView uses its single-server launch procedure.

Command

If you enable bulk launch, TotalView uses this command to launch **tvdsrv**. For information on this command and its options, see "Bulk Server Launch Setting on an SGI Computer" on page 89 and "Setting Bulk Server Launch on an IBM RS/6000 AIX Computer" on page 91.

CLI: dset TV::bulk_launch_string

Temp File 1 Prototype

Temp File 2 Prototype

Both of these tab pages have three fields. These fields let you specify the contents of temporary files that the bulk launch operation uses. For information on these fields, see "*TotalView Debugger Server (tvdsrv) Command Syntax*" in the *TotalView Reference Guide*.

CLI: dset TV::bulk_launch_tmpfile1_header_line
dset TV::bulk_launch_tmpfile1_host_lines
dset TV::bulk_launch_tmpfile1_trailer_line
dset TV::bulk_launch_tmpfile2_header_line
dset TV::bulk_launch_tmpfile2_host_lines
dset TV::bulk_launch_tmpfile2_trailer_line

Connection Timeout (in seconds)

After TotalView launches **tvdsrv** processes, it waits 20 seconds for responses from the process (the **Base** time) plus 10 seconds for each server process being

launched. If a connection is not made in this time, TotalView times out.

A **Base** timeout value can range from 1 to 3600 seconds (1 hour). The incremental **Plus** value is from 1 to 360 seconds (6 minutes). See the online Help for information on setting these values.

```
CLI: dset TV::bulk_launch_base_timeout  
      dset TV::bulk_launch_incr_timeout
```

If you notice that TotalView fails to launch **tvdsrv** (as shown in the **xterm** window from which you started TotalView) before the timeout expires, select **Yes** in the **Question Dialog Box** that appears.

Defaults

If you make a mistake or decide you want to go back to TotalView's default settings, click the **Defaults** button.

Clicking **Defaults** also throws away any changes you made using a CLI variable. TotalView doesn't immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Starting the TotalView Server Manually

If TotalView can't automatically launch **tvdsrv**, you can start it manually.



You cannot debug remote processes using TotalView Individual.

If you use a "*hostname:portnumber*" qualifier when opening a remote process, TotalView does not launch a debugger server.

Here are the steps for manually starting **tvdsrv**:

- 1 Make sure that both bulk launch and single server launch are disabled. To disable bulk launch, click the Bulk Launch Tab in the **File > Preferences** Dialog Box. (You can select this command from the Root Window or the Process Window.) The dialog box shown on the next page appears.
- 2 Clear the **Enable debug server bulk launch** check box in the Bulk Launch Tab to disable autolaunching and then select **OK**.

```
CLI: dset TV::bulk_launch_enabled
```

- 3 Select the Server Launch Tab and clear the **Enable single debug server launch** check box.

```
CLI: dset TV::server_launch_enabled
```

- 4 Log in to the remote computer and start **tvdsrv**:

```
tvdsrv -server
```

If you don't (or can't) use the default port number (4142), you will need to use the **-port** or **-search_port** options. For details, see "TotalView Debugger Server (**tvdsrv**) Command Syntax" in the *TotalView Reference Guide*.

After printing the port number and the assigned password, the server begins listening for connections. Be sure to remember the password—you need to enter it in step 5.



*Using the **-server** option is not secure, other users could connect to your **tvdsrv** process and use your UNIX UID. Consequently, this command-line option must be explicitly enabled. (Your system administrator usually does this.) For details, see **-server** in the "TotalView Command Syntax" chapter of the *TotalView Reference Guide*.*

- 5 From the Root Window, select the **File > New Program** command. Type the program's name in the **Program** field and the *hostname:portnumber* in the **on host** field and then select **OK**.

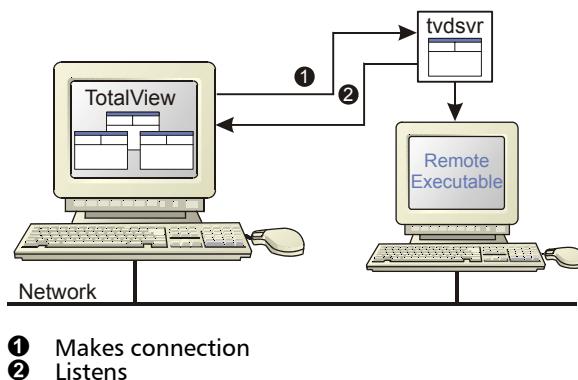
```
CLI: dload executable -r hostname
```

- 6 TotalView tries to connect to **tvdsrv**.

- 7 When TotalView prompts you for the password, enter the password that **tvdsrv** displayed in step 4.

The following figure summarizes the steps for starting **tvdsrv** manually.

Figure 77: Manual Launching of Debugger Server



Using the Single-Process Server Launch Command

The following is the default command string that TotalView uses when it automatically launches TotalView server for a single process:

Starting the TotalView Server Manually

```
%C %R -n "%B/tvdsrv –working_directory %D –callback %L \
–set_pw %P –verbosity %V %F"
```

where:

%C	Expands to the name of the server launch command to use, which is the value of TV::launch_command . On most platforms, this is rsh . On HP computers, it is remsh . On SiCortex, it is ssh - x . If the TVDSVRLAUNCHCMD environment variable exists, TV::launch_command is initialized to its value.
%R	Expands to the host name of the remote computer that you specified in the File > New Program or dload commands.
%B	Expands to the bin directory in which tvdsrv is installed.
-n	Tells the remote shell to read standard input from /dev/null ; that is, the process immediately receives an EOF (End-Of-File) signal.
–working_directory %D	Makes %D the directory to which TotalView connects. %D expands to the absolute path name of the directory. When you use this option, the host computer and the target computer must mount identical file systems. That is, the path name of the directory to which TotalView connects must be identical on host and target computers. After changing to this directory, the shell invokes the tvdsrv command. You must make sure that the tvdsrv directory is in your path on the remote computer.
–callback %L	Establishes a connection from tvdsrv to TotalView. %L expands to the host name and TCP/IP port number (<i>hostname:portnumber</i>) on which TotalView is listening for connections from tvdsrv .
–set_pw %P	Sets a 64-bit password. TotalView must supply this password when tvdsrv establishes a connection with it. TotalView expands %P to the password that it automatically generates. For more information on this password, see "TotalView Debugger Server (tvdsrv) Command Syntax" in the <i>TotalView Reference Guide</i> .
–verbosity %V	Sets the verbosity level of the tvdsrv . %V expands to the current verbosity setting. For information on verbosity, see the "Variables" chapter within the <i>TotalView Reference Guide</i> .
%F	Contains the tracer configuration flags that need to be sent to tvdsrv processes. These are system-specific startup options that the tvdsrv process needs.

You can also use the %H option with this command. See "Bulk Server Launch Setting on an SGI Computers" on page 89 for more information.

For information on the complete syntax of the **tvdsrv** command, see "TotalView Debugger Server (**tvdsrv**) Command Syntax" in the *TotalView Reference Guide*.

Bulk Server Launch Setting on an SGI Computers

On SGI MIPS and SGI ICE computers, the bulk server launch string is as follows:

```
array tvdsrv –working_directory %D –callback_host %H \
–callback_ports %L –set_pws %P –verbosity %V %F
```

where:

–working_directory %D

Makes %D the directory to which TotalView connects. TotalView expands %D to this directory's absolute path name.

When you use this option, the host computer and the target computer must mount identical file systems. That is, the path name of the directory to which TotalView connects must be identical on the host and target computers.

After performing this operation, **tvdsrv** starts executing.

–callback_host %H

Names the host upon which TotalView makes this callback. TotalView expands %H to the host name of the computer on which TotalView is running.

–callback_ports %L

Names the ports on the host computers that TotalView uses for callbacks. TotalView expands %L to a comma-separated list of host names and TCP/IP port numbers (*hostname:portnumber,hostname:portnumbe...*) on which TotalView is listening for connections.

–set_pws %P

Sets 64-bit passwords. TotalView must supply these passwords when **tvdsrv** establishes the connection with it. %P expands to a comma-separated list of 64-bit passwords that TotalView automatically generates. For more information, see "TotalView Debugger Server (**tvdsrv**) Command Syntax" in the *TotalView Reference Guide*.

–verbosity %V

Sets the **tvdsrv** verbosity level. TotalView expands %V to the current verbosity setting. For information on verbosity, see the "Variables" chapter within the *TotalView Reference Guide*.

You must enable the use of the **array** command by **tvdsrv** by adding the following information to the */usr/lib/array/arrayd.conf* file:

```
#  
# Command that allow invocation of the TotalView  
# Debugger server when performing a Bulk Server Launch.
```

```
#  
command tvdsvr  
    invoke /opt/totalview/bin/tvdsvr %ALLARGS  
    user %USER  
    group %GROUP  
    project %PROJECT
```

If your code is not in `/opt/totalview/bin`, you will need to change this information. For information on the syntax of the `tvdsvr` command, see “*TotalView Debugger Server (tvdsvr) Command Syntax*” in the *TotalView Reference Guide*.

Setting Bulk Server Launch on an HP Alpha Computer

The following is the bulk launch string on an HP Alpha computer:

```
prun -T %Z %B/tvdsvr –callback_host %H \  
    –callback_ports %L –set_pws %P \  
    –verbosity %V –working_directory %D
```

Information on the options and expansion symbols is in the “*TotalView Debugger Server (tvdsvr) Syntax*” chapter of the *TotalView Reference Guide*.

Setting Bulk Server Launch on a Cray XT Series Computer

The following is the bulk server launch string for Cray XT series computers:

```
svrlaunch %B/tvdsvrmain%K -verbosity %V %F %H \  
    %t1 %I %K
```

where the options unique to this command are:

%B	The bin directory where <code>tvdsvr</code> resides.
%K	The number of servers that TotalView launches.
<code>-verbosity %V</code>	Sets the verbosity level of the <code>tvdsvr</code> . <code>%V</code> expands to the current verbosity setting. For information on verbosity, see the “ <i>Variables</i> ” chapter within the <i>TotalView Reference Guide</i> .
%F	Contains the “tracer configuration flags” that need to be sent to <code>tvdsvr</code> processes. These are system-specific startup options that the <code>tvdsvr</code> process needs.
%H	Expands to the host name of the machine upon which TotalView is running.
%t1	A temporary file created by TotalView that contains a list of the hosts on which <code>tvdsvr</code> runs. This is the information you enter in the Temp File 1 Prototype field on the Bulk Launch Page.
%I	Expands to the pid of the MPI starter process. For example, it can contain <code>mpirun</code> , <code>aprun</code> , etc. It can also be the process to which you manually attach. If no pid is available, <code>%I</code> expands to 0.

Setting Bulk Server Launch on an IBM RS/6000 AIX Computer

The following is the bulk server launch string on an IBM RS/6000 AIX computer:

```
%C %H -n "poe -pgmmodel mpmd -resd no -tasks_per_node 1\
-procs %N -hostfile %t1 -cmdfile %t2 %F"
```

where the options unique to this command are:

- | | |
|-----|---|
| %N | The number of servers that TotalView launches. |
| %t1 | A temporary file created by TotalView that contains a list of the hosts on which tvdsrv runs. This is the information you enter in the Temp File 1 Prototype field on the Bulk Launch Page. TotalView generates this information by expanding the %R symbol. This is the information you enter in the Temp File 2 Prototype field on the Bulk Launch Page. |
| %t2 | A file that contains the commands to start the tvdsrv processes on each computer. TotalView creates these lines by expanding the following template: |

```
tvdsrv -working_directory %D \
-callback %L -set_pw %P \
-verbosity %V
```

Information on the options and expansion symbols is in the "*TotalView Debugger Server (tvdsrv) Syntax*" chapter of the *TotalView Reference Guide*.

Disabling Autolaunch

If after changing autolaunching options, TotalView still can't automatically start **tvdsrv**, you must disable autolaunching and start **tvdsrv** manually. Before trying to manually start the server, you must clear the **Enable single debug server launch** check box on the Launch Strings Page of the **File > Preferences** Dialog Box.

CLI: dset TV::server_launch_enabled

You can use the procedure described in "Setting Up and Starting the TotalView Server" on page 81 to get the program started. You will also need to enter a host name and port number in the bottom section of the **File > New Program** Dialog Box. This disables autolaunching for the current connection.



If you disable autolaunching, you must start tvdsrv before you load a remote executable or attach to a remote process.

Changing the Remote Shell Command

Some environments require you to create your own autolaunching command. You might do this, for example, if your remote shell command doesn't provide the security that your site requires.

If you create your own autolaunching command, you must use the **tvdsrv -callback** and **-set_pw** command-line options.

If you're not sure whether **rsh** (or **remsh** on HP computers) works at your site, try typing "**rsh hostname**" (or "**remsh hostname**") from an **xterm** window, where *hostname* is the name of the host on which you want to invoke the remote process. If the process doesn't just run and instead this command prompts you for a password, you must add the host name of the host computer to your **.rhosts** file on the target computer.

For example, you can use the following combination of the **echo** and **telnet** commands:

```
echo %D %L %P %V; telnet %R
```

After **telnet** establishes a connection to the remote host, you can use the **cd** and **tvdsrv** commands directly, using the values of **%D**, **%L**, **%P**, and **%V** that were displayed by the **echo** command; for example:

```
cd directory
tvdsrv -callback hostname:portnumber -set_pw password
```

If your computer doesn't have a command for invoking a remote process, TotalView can't autolaunch the **tvdsrv** and you must disable both single server and bulk server launches.

For information on the **rsh** and **remsh** commands, see the manual page supplied with your operating system.

Changing Arguments

You can also change the command-line arguments passed to **rsh** (or whatever command you use to invoke the remote process).

For example, if the host computer doesn't mount the same file systems as your target computer, **tvdsrv** might need to use a different path to access the executable being debugged. If this is the case, you can change **%D** to the directory used on the target computer.

If the remote executable reads from standard input, you cannot use the **-n** option with your remote shell command because the remote executable receives an EOF immediately on standard input. If you omit the **-n** command-line option, the remote executable reads standard input from the **xterm** in which you started TotalView. This means that you should invoke **tvdsrv** from another **xterm** window if your remote program reads from standard input. The following is an example:

```
%C %R "xterm -display hostname:0 -e tvdsrv \
-callback %L -working_directory %D -set_pw %P \
-verbosity %V"
```

Each time TotalView launches **tvdsrv**, a new **xterm** appears on your screen to handle standard input and output for the remote program.

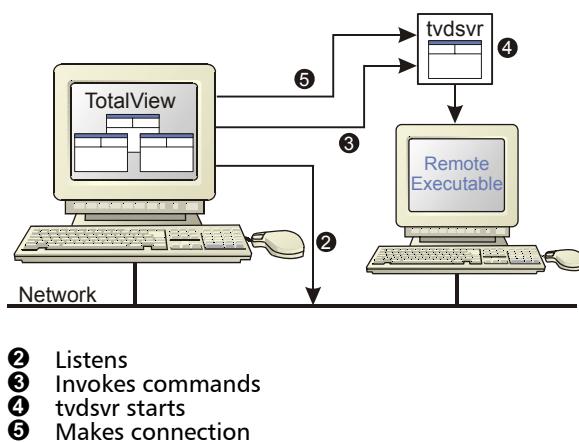
Autolaunching Sequence

This section describes the sequence of actions involved in autolaunching. You can skip this section if you aren't having any problems or if you aren't curious.

- 1** With the **File > New Program** or **dload** commands, you specify the host name of the computer on which you want to debug a remote process, as described in "Setting Up and Starting the TotalView Server" on page 81.
- 2** TotalView begins listening for incoming connections.
- 3** TotalView launches the **tvdsrv** process with the server launch command. (See "Using the Single-Process Server Launch Command" on page 87 for more information.)
- 4** The **tvdsrv** process starts on the remote computer.
- 5** The **tvdsrv** process establishes a connection with TotalView.

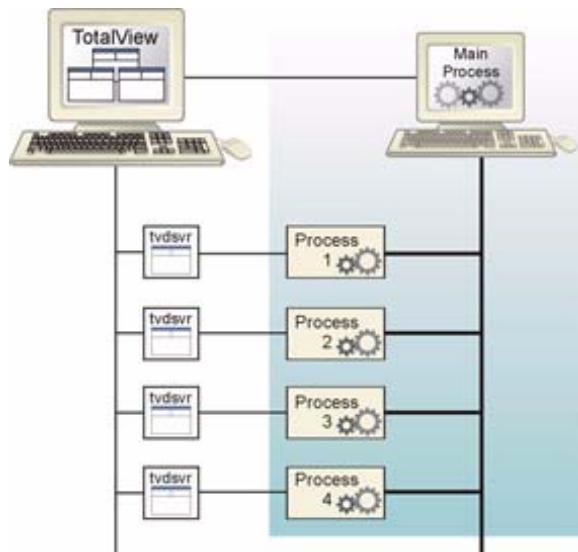
The following figure summarizes these actions if your program is launching one server. The numbers within this figure refer to the numbers in the preceding procedure.

Figure 78: Launching **tvdsrv**



If you have more than one server process, the following figure shows what your environment might look like:

Figure 79: Multiple `tvdsrv` Processes



Debugging Over a Serial Line

TotalView lets you debug programs over a serial line as well as TCP/IP sockets. However, if a network connection exists, you probably want to use it to improve performance.

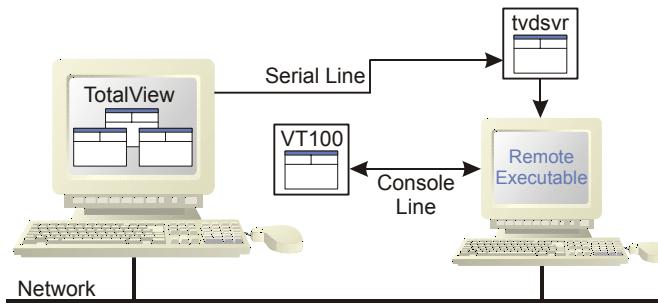
You need two connections to the target computer: one for the console and the other for TotalView. Do not try to use one serial line because TotalView cannot share a serial line with the console.

The following figure illustrates a TotalView session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a `tvdsrv` running on the target host. A VT100 terminal is connected to the target host's console line, which lets you type commands on the target host. (See Figure 80.)

This section contains the following topics:

- "Starting the TotalView Debugger Server" on page 95
- "Using the New Program Window" on page 95

Figure 80: Debugging Session
Over a Serial Line



Starting the TotalView Debugger Server

To start a debugging session over a serial line from the command line, you must first start the **tvdsrv**.

Using the console connected to the target computer, start **tvdsrv** and enter the name of the serial port device on the target computer. Use the following syntax:

```
tvdsrv -serial device[:baud=num]
```

where:

device The name of the serial line device.

num The serial line's baud rate. If you omit the baud rate, TotalView uses a default value of **38400**.

For example:

```
tvdsrv -serial /dev/com1:baud=38400
```

After it starts, **tvdsrv** waits for TotalView to establish a connection.

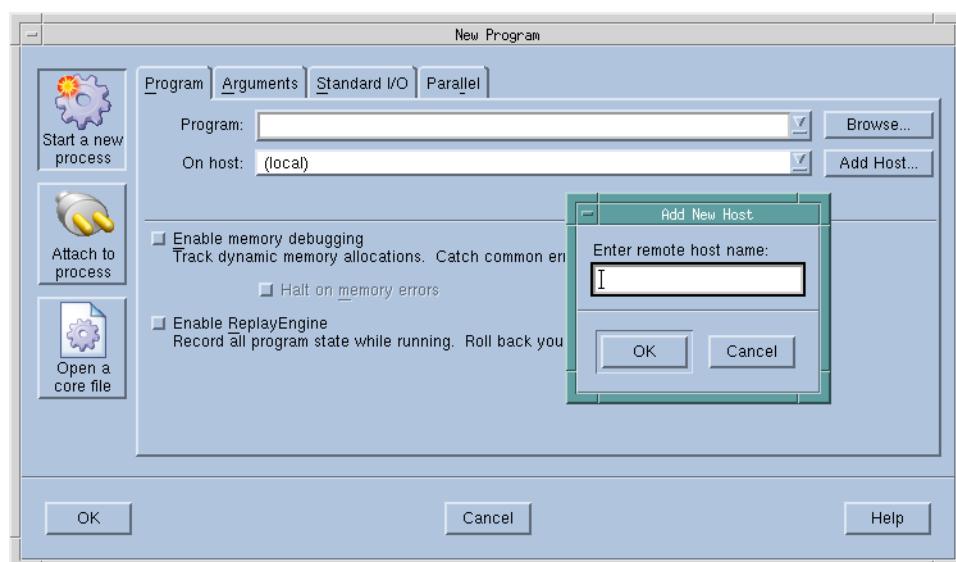
Using the New Program Window

The following procedure starts a debugging session over a serial line when you're already in TotalView:

- 1 Start the **tvdsrv** (see "Starting the TotalView Debugger Server" on page 95).
 - 2 Select the **File > New Program** command. In the pulldown list on the right, select **Add new host**. TotalView displays the dialog box shown in Figure 81.
 - 3 Type the name of the program in the **Program** field.
 - 4 Type the name of the serial line device in the **Serial Line** field. For example, you might type **/dev/ttyS0:serial**, where **/dev/ttyS0** is the device file representing the serial port.
- Select **OK**.

Debugging Over a Serial Line

Figure 81: Adding New Host



Setting Up MPI Debugging Sessions



This chapter explains how to set up TotalView MPI debugging sessions. You will find information on starting up other kinds of parallel programs in Chapter 7, “*Setting Up Parallel Debugging Sessions*,” on page 131.



If you are using TotalView Individual, all of your MPI processes must execute on the computer on which you installed TotalView. In addition, TotalView Individual limits you to no more than 16 processes and threads.

CHAPTER 6

This chapter describes the following MPI systems:

- “Debugging MPI Programs” on page 98
- “Debugging MPICH Applications” on page 100
- “Debugging MPICH2 Applications” on page 104
- “Debugging Cray MPI Applications” on page 112
- “Debugging HP Tru64 Alpha MPI Applications” on page 112
- “Debugging HP MPI Applications” on page 113
- “Debugging IBM MPI Parallel Environment (PE) Applications” on page 115
- “Debugging IBM Blue Gene Applications” on page 118
- “Debugging LAM/MPI Applications” on page 119
- “Debugging QSW RMS Applications” on page 120
- “Debugging SiCortex MPI Applications” on page 121
- “Debugging SGI MPI Applications” on page 121
- “Debugging Sun MPI Applications” on page 123

This chapter also describes debugger features that you can use with most parallel models:

- If you’re using a messaging system, TotalView displays this information visually as a message queue graph and textually in a Message Queue Window. For more information, see “*Displaying the Message Queue Graph Window*” on page 107 and “*Displaying the Message Queue*” on page 109.

- TotalView lets you decide which process you want it to attach to. See "Attaching to Processes" on page 124.
- See "Debugging Parallel Applications Tips" on page 124 for hints on how to approach debugging parallel programs.

Debugging MPI Programs

Starting MPI Programs

MPI programs use a starter program such as mpirun to start your program. TotalView lets you start these MPI programs in two ways. One requires that the starter program be under TotalView control, and the other does not. In the first case, you will enter the name of the starter program on the command line. In the other, you will enter program information into the **File > New Program** or **Process > Startup Parameter** dialog boxes.

Programs started using these dialog boxes do not use the information you set for single-process and bulk server launching. In addition, you cannot use the Attach Subset command when entering information using these dialog boxes.

Starting MPI programs using the dialog boxes is the recommended method. This method is described in the next section. Starting using a started program is described in various places throughout this chapter.

Starting MPI Programs Using File > New Program

In many cases, the way in which you invoke an MPI program within TotalView control differs little from discipline to discipline. If you invoke TotalView from the command line without an argument, TotalView displays its **File > New Program** dialog box. (See Figure 82 on page 99.)

After entering program's name (**Start a new process** should be selected by default), select the Parallel tab. (See Figure 83 on page 99.)

You can now select the **Parallel** system, the number of **Tasks**, and **Nodes**. If there are additional arguments that need to be sent to the starter process, type them within the **Additional starter arguments** area. These arguments are ones that are sent to a starter process such as mpirun or poe. They are not arguments sent to your program.

If you need to add and initialize environment variables and command-line options, select the Arguments tab and enter them.

In most cases, TotalView will remember what you type between invocations of TotalView. For example, suppose you were debugging a program called

Figure 82: File > New Program Dialog Box

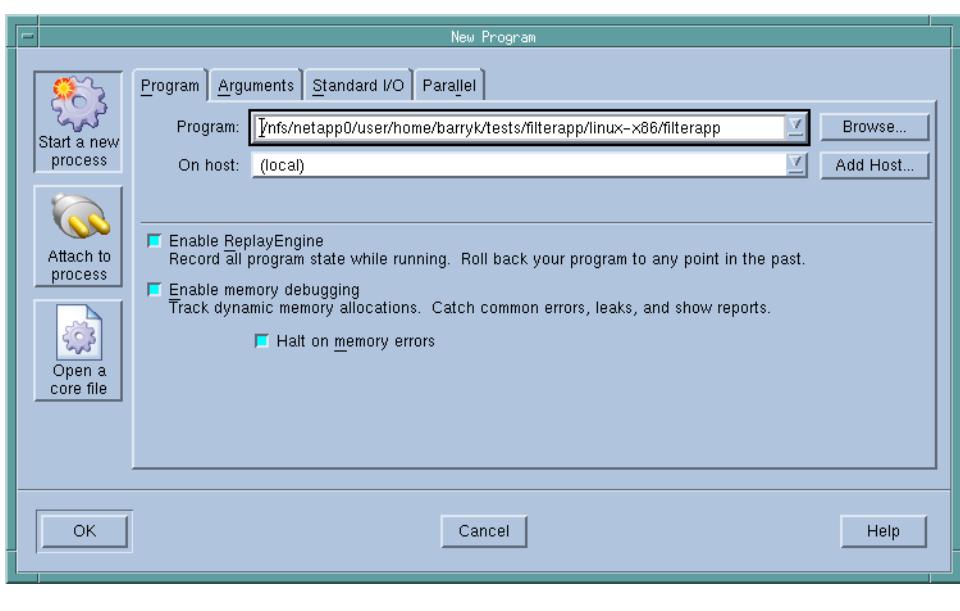
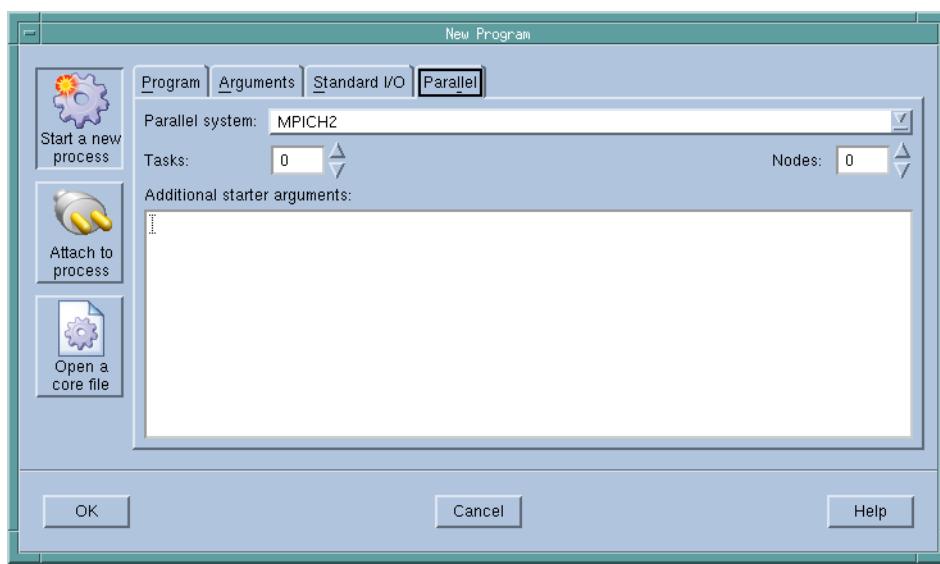


Figure 83: File > New Program Dialog Box: Parallel Tab



my_foo and set it up using these controls. The next time you start TotalView, you can use the following command:

```
totalview my_foo
```

TotalView will remember what you entered so there is no need to respecify these options.

Debugging MPICH Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.3 or later on a homogenous collection of computers. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at www.mcs.anl.gov/mpi. (We strongly urge that you use a later version of MPICH. The *TotalView Platforms and Systems Requirements* document has information on versions that work with TotalView.)

The MPICH library should use the `ch_p4`, `ch_p4mpd`, `ch_shmem`, `ch_ifshmem`, or `ch_mpl` devices.

- For networks of workstations, the default MPICH library is `ch_p4`.
- For shared-memory SMP computers, use `ch_shmem`.
- On an IBM SP computer, use the `ch_mpl` device.

The MPICH source distribution includes all of these devices. Choose the device that best fits your environment when you configure and build MPICH.



When configuring MPICH, you must ensure that the MPICH library maintains all of the information that TotalView requires. This means that you must use the `--enable-debug` option with the MPICH `configure` command. (Versions earlier than 1.2 used the `--debug` option.) In addition, the TotalView Release Notes contains information on patching your MPICH version 1.2.3 distribution.

For more information, see:

- "Starting TotalView on an MPICH Job" on page 100
- "Attaching to an MPICH Job" on page 102
- "Using MPICH P4 procgroup Files" on page 103

Starting TotalView on an MPICH Job

Before you can bring an MPICH job under TotalView's control, both TotalView and the `tvdsvr` must be in your path. You can do this in a login or shell startup script.

For version 1.1.2, the following command-line syntax starts a job under TotalView control:

`mpirun [MPICH-arguments] -tv program [program-arguments]`

For example:

`mpirun -np 4 -tv sendrecv`

The MPICH **mpirun** command obtains information from the **TOTALVIEW** environment variable and then uses this information when it starts the first process in the parallel job.

For Version 1.2.4, the syntax changes to the following:

```
mpirun -dbg=totalview [ other_mpich-args ] program [ program-args ]
```

For example:

```
mpirun -dbg=totalview -np 4 sendrecv
```

In this case, **mpirun** obtains the information it needs from the **-dbg** command-line option.

In other contexts, setting this environment variable means that you can use TotalView different versions or pass command-line options to TotalView.

For example, the following is the C shell command that sets the **TOTALVIEW** environment variable so that **mpirun** passes the **-no_stop_all** option to TotalView:

```
setenv TOTALVIEW "totalview -no_stop_all"
```

TotalView begins by starting the first process of your job, the master process, under its control. You can then set breakpoints and begin debugging your code.

On the IBM SP computer with the **ch_mpl** device, the **mpirun** command uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and its **-tv** option) command to start an MPICH job, the way you start MPICH differs. For details on using TotalView with **poe**, see "Starting TotalView on a PE Program" on page 116.

Starting TotalView using the **ch_p4mpd** device is similar to starting TotalView using **poe** on an IBM computer or other methods you might use on Sun and HP platforms. In general, you start TotalView using the **totalview** command, with the following syntax;

```
totalview mpirun [ totalview_args ] -a [ mpich-args ] program [ program-args ]
```

CLI: totalviewcli mpirun [totalview_args] \ -a [mpich-args] program [program-args]
--

As your program executes, TotalView automatically acquires the processes that are part of your parallel job as your program creates them. Before TotalView begins to acquire them, it asks if you want to stop the spawned processes. If you click **Yes**, you can stop processes as they are initialized. This lets you check their states or set breakpoints that are unique to the process. TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. Consequently, you don't have to stop them just to set these breakpoints.

If you're using the GUI, TotalView updates the Root Window to show these newly acquired processes. For more information, see "Attaching to Processes" on page 124.

Attaching to an MPICH Job

TotalView lets you to attach to an MPICH application even if it was not started under TotalView control.

To attach to an MPICH application:

1 Start TotalView.

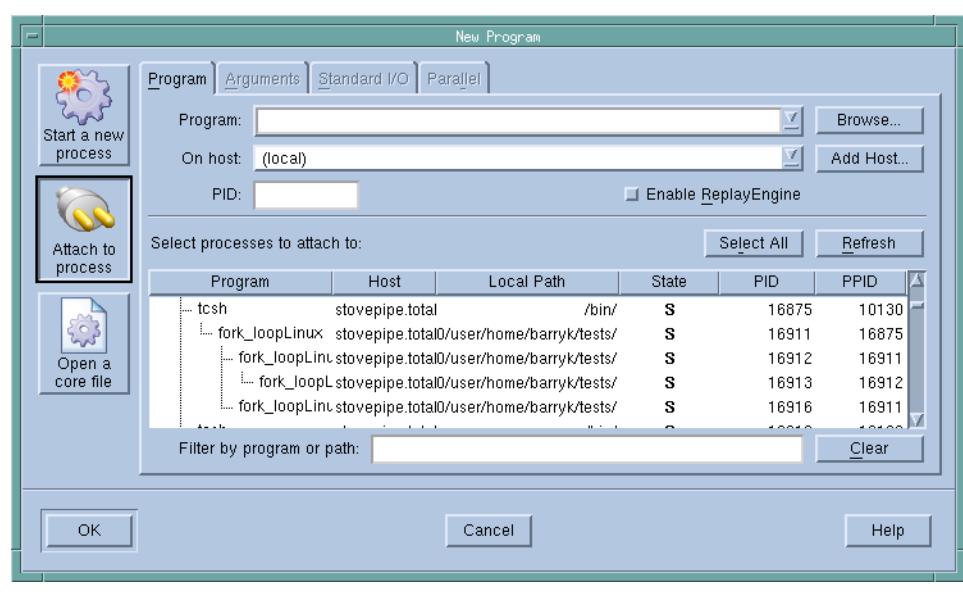
Select **Attach to an existing process** from within the **File > New Program** dialog box. TotalView updates the dialog box so that it displays the processes that are not yet owned.

2 Attach to the first MPICH process in your workstation cluster by diving into it.

CLI: `dattach executable pid`

3 On an IBM SP with the **ch_mpi** device, attach to the **poe** process that started your job. For details, see "Starting TotalView on a PE Program" on page 116. Figure 84 shows this information.

Figure 84: File > New Program: Attach to an Existing Process



Normally, the first MPICH process is the highest process with the correct program name in the process list. Other instances of the same executable can be:

- The **p4** listener processes if MPICH was configured with **ch_p4**.
- Additional slave processes if MPICH was configured with **ch_shmem** or **ch_ifshmem**.
- Additional slave processes if MPICH was configured with **ch_p4** and has a file that places multiple processes on the same computer.

- 4** After you attach to your program's processes, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

As an alternative, you can use the **Group > Attach Subset** command to predefine what TotalView should do. For more information, see "Attaching to Processes" on page 124.



If you are using TotalView Individual, all of your MPI processes must execute on the computer on which you installed TotalView.

In some situations, the processes you expect to see might not exist (for example, they may crash or exit). TotalView acquires all the processes it can and then warns you if it can not attach to some of them. If you attempt to dive into a process that no longer exists (for example, using a message queue display), TotalView tells you that the process no longer exists.

Using MPICH P4 procgroup Files

If you're using MPICH with a P4 **procgroup** file (by using the **-p4pg** option), you must use the *same* absolute path name in your **procgroup** file and on the **mpirun** command line. For example, if your **procgroup** file contains a different path name than that used in the **mpirun** command, even though this name resolves to the same executable, TotalView assumes that it is a different executable, which causes debugging problems.

The following example uses the same absolute path name on the TotalView command line and in the **procgroup** file:

```
% cat p4group
local 1 /users/smith/mympichexe
bigiron 2 /users/smith/mympichexe
% mpirun -p4pg p4group -tv /users/smith/mympichexe
```

In this example, TotalView does the following:

- 1** Reads the symbols from **mympichexe** only once.
- 2** Places MPICH processes in the same TotalView share group.
- 3** Names the processes **mympichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

If TotalView assigns names such as **mympichexe<mympichexe>.0**, a problem occurred and you need to compare the contents of your **procgroup** file and **mpirun** command line.

Debugging MPICH2 Applications



You should be using MPICH2 version 1.0.5p4 or higher. Earlier versions had problems that prevented TotalView from attaching to all the processes or viewing message queue data.

Downloading and Configuring MPICH2

You can download the current MPICH2 version from:

<http://www-unix.mcs.anl.gov/mpi/mpich/>

If you wish to use all of the TotalView MPI features, you must configure MPICH2. Do this by adding the following to the **configure** script that is within the downloaded information:

`- -enable-debuginfo - -enable-totalview`

The **configure** script looks for the following file:

`python2.x/config/Makefile`

It fails if the file is not there.

The next steps are:

1 Run **make**

2 Run **make install**

This places the binaries and libraries in the directory specified by the optional `- -prefix` option.

3 Set the PATH and LD_LIBRARY_PATH to point to the MPICH2 **bin** and **lib** directories.

Starting the mpd Daemon

You must start the **mpd** daemon using the **mpdboot** command. For example:

`mpdboot -n 4 -f hostfile`

where:

-n 4 Indicates the number of hosts upon which you wish to run the daemon. In this example, the daemon runs on four hosts

-f hostfile Is a list of the hosts upon which the application will run. In this example, a file named **hostfile** contains this list.

You are now ready to start debugging your application.



TotalView only supports jobs run using MPD. Using other daemons such as SMPD is not supported.

Starting TotalView Debugging on an MPICH2 Job



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

TotalView lets you start an MPICH2 job in one of the following ways:

```
mpiexec mpi-args -tv program -a program-args
```

This command tells MPI to start TotalView. You will need to set the TOTALVIEW environment variable to where TotalView is located in your file system when you start a program using `mpiexec`. For example:

```
setenv TOTALVIEW \
/opt/totalview/bin/totalview
```

This method of starting TotalView does not let you restart your program without exiting TotalView and you will not be able to attach to a running MPI job.

```
totalview python -a `which mpiexec` \
-tvsu mpiexec-args program program-args
```

This command lets you restart your MPICH2 job. It also lets you attach to a running MPICH2 job by using the **Attach to process** options within the **File > New Program** dialog box. You need to be careful that you attach to the right instance of python as it is likely that a few instances are running. The one to which you want to attach has no attached children—child processes are indented with a line showing the connection to the parent.

You may not see sources to your program at first. If you do see the program, you can set breakpoints. In either case, press the **Go** button. Your process will start and TotalView displays a dialog box when your program goes parallel that allows you to stop execution. (This is the default behavior. You can change it using the options within **File > Preferences > Parallel** page.

You will also need to set the TOTALVIEW environment variable as indicated in the previous method.

Starting MPI Issues



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

MPI Rank Display

If you can't successfully start TotalView on MPI programs, check the following:

- Can you successfully start MPICH programs without TotalView?
The MPICH code contains some useful scripts that let you verify that you can start remote processes on all of the computers in your computers file. (See **tstmachines** in **mpich/util**.)
- You won't get a message queue display if you get the following warning:
The symbols and types in the MPICH library used by TotalView to extract the message queues are not as expected in the image <your image name>. This is probably an MPICH version or configuration problem.
You need to check that you are using MPICH Version 1.1.0 or later and that you have configured it with the **-debug** option. (You can check this by looking in the **config.status** file at the root of the MPICH directory tree.)
- Does the TotalView Server (**tvdsrv**) fail to start?
tvdsrv must be in your **PATH** when you log in. Remember that TotalView uses **rsh** to start the server, and that this command doesn't pass your current environment to remotely started processes.
- Make sure you have the correct MPI version and have applied all required patches. See the *TotalView Release Notes* for up-to-date information.
- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal. You can see this behavior when you use the **Group > Kill** command as the first step in restarting an MPICH job.

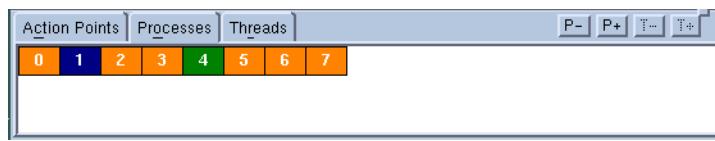
CLI: **dfocus g ddelete**

If TotalView exits and terminates abnormally with a **Killed** message, try setting the **TV::ignore_control_c** variable to true.

MPI Rank Display

The Processes/Ranks Tab at the bottom of the Process Window contains a grid that displays the status of each rank. For example, in Figure 85, six ranks are at a breakpoint, one is running, and one is stopped.

Figure 85: Ranks Tab

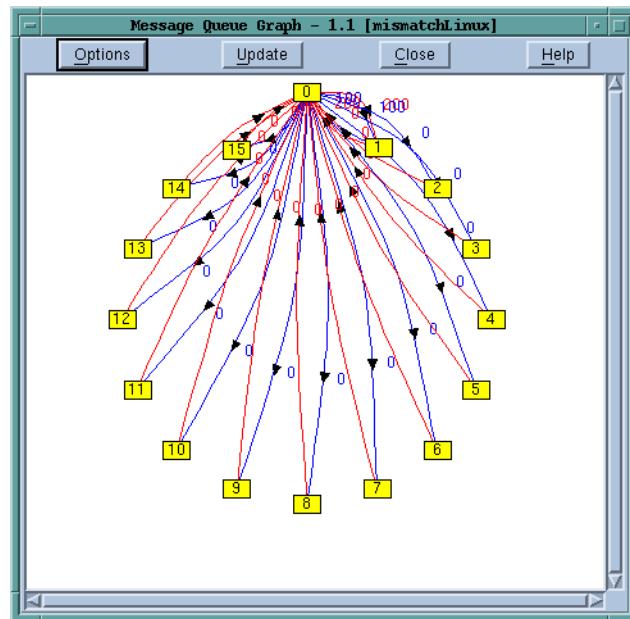


For more information, see "Using the Processes/Ranks Tab" on page 229.

Displaying the Message Queue Graph Window

TotalView can graphically display your MPI program's message queue state. If you select the Process Window Tools > Message Queue Graph command, TotalView displays a window that shows a graph of the current message queue state. (See Figure 86.)

Figure 86: Tools > Message Queue Graph Window



If you want to restrict what TotalView displays, you can select the **Options** button. This is shown in Figure 87 on page 108.

Using commands and controls within this window, you can either alter the way in which TotalView displays ranks within this window—for example, as a grid or in a circle.

Using the commands within the **Cycle Detection** tab tells TotalView to let you know about cycles in your messages. This is a quick and efficient way to detect when messages are blocking one another and causing deadlocks.

Perhaps the most used of these tabs is **Filter**. (See Figure 88 on page 108.)

The button colors used for selecting messages are the same as those used to draw the lines and arrows in the **Message Queue Graph** Window, as follows:

- **Green**: Pending Sends
- **Blue**: Pending Receives
- **Red**: Unexpected Messages

Displaying the Message Queue Graph Window

Figure 87: Tools > Message Queue Graph Options Window

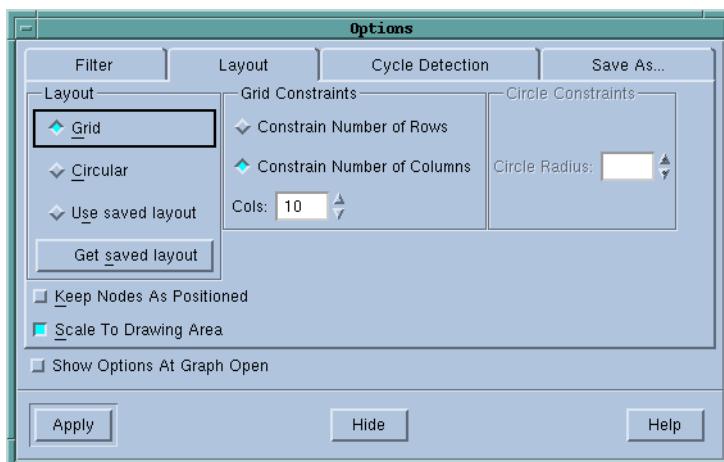
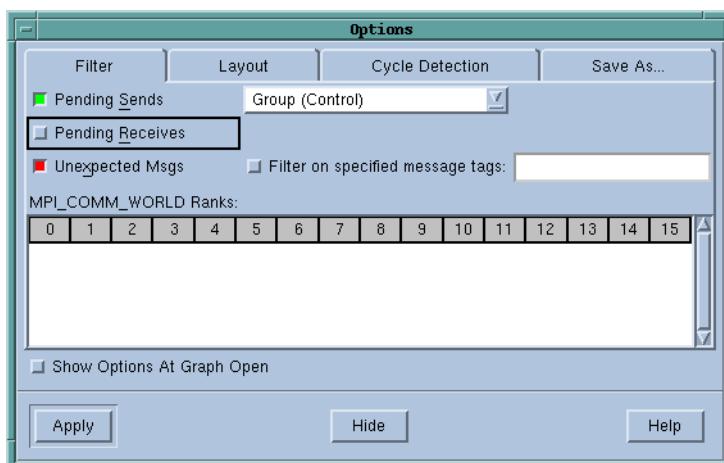


Figure 88: Tools > Message Queue Graph Options. Filter Tab



You can directly select which ranks you want displayed in the lower part of the window. The **Filter on specified message tags** area lets you name which tags should be used as filters. Finally, you can select a group or a communicator in the group pulldown. If you have created your own communicators and groups, they will appear here.

Changes made within the **Options** dialog box do not occur until after you click the **Apply** button. The graph window will then change to reflect your changes.

The message queue graph shows your program's state at a particular instant. Selecting the **Update** button tells TotalView to fetch new information and redraw the graph.

The numbers in the boxes within the **Message Queue Graph** Window indicate the MPI message source or destination process rank. Diving on a box tells TotalView to open a Process Window for that process.

The numbers next to the arrows indicate the MPI message tags that existed when TotalView created the graph. Diving on an arrow tells TotalView to dis-

play its **Tools > Message Queue** Window, which has detailed information about the messages. If TotalView has not attached to a process, it displays this information in a grey box.

You can use the **Message Queue Graph** Window in many ways, including the following:

- Pending messages often indicate that a process can't keep up with the amount of work it is expected to perform. These messages indicate places where you may be able to improve your program's efficiency.
- Unexpected messages can indicate that something is wrong with your program because the receiving process doesn't know how to process the message. The red lines indicate unexpected messages.
- After a while, the shape of the graph tends to tell you something about how your program is executing. If something doesn't look right, you might want to determine why.
- You can change the shape of the graph by dragging nodes or arrows. This is often useful when you're comparing sets of nodes and their messages with one another. Ordinarily, TotalView doesn't remember the places to which you have dragged the nodes and arrows. This means that if you select the **Update** button after you arrange the graph, your changes are lost. However, if you select **Keep nodes as positioned** from the **Options** dialog box, updating the window does not change node positioning.

Displaying the Message Queue

The **Tools > Message Queue** Window displays your MPI program's message queue state textually. This can be useful when you need to find out why a deadlock occurred.

The MPI versions for which we display the message queue are described in our platforms guide. This document is contained within the online help and is also available on our web site at <http://www.totalviewtech.com/Documentation/>

For more information, see:

- "About the Message Queue Display" on page 109
- "Using Message Operations" on page 110

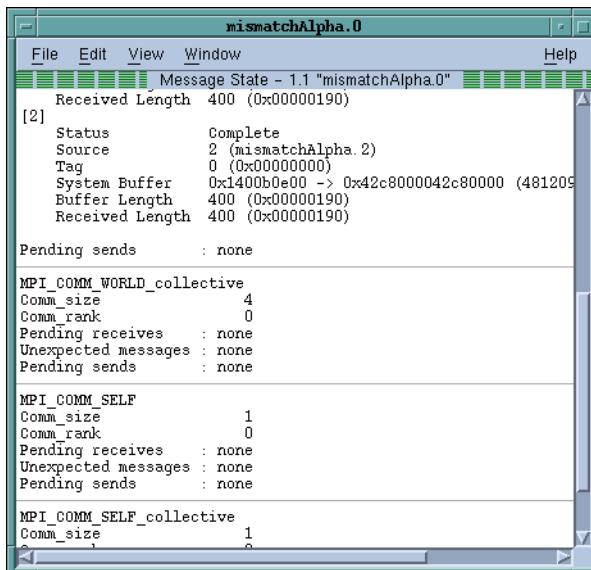
About the Message Queue Display

After an MPI process returns from the call to **MPI_Init()**, you can display the internal state of the MPI library by selecting the **Tools > Message Queue** command. (See Figure 89 on page 110.)

This window displays the state of the process's MPI communicators. If user-visible communicators are implemented as two internal communica-

Displaying the Message Queue

Figure 89: Message Queue Window



For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has an index value displayed in brackets ([n]). The online Help for this window contains a description of the fields that you can display.

You cannot edit any of the fields in the Message Queue Window.



The contents of the Message Queue Window are only valid when a process is stopped.

Using Message Operations

For each communicator, TotalView displays a list of pending receive operations, pending unexpected messages, and pending send operations. Each operation has an index value displayed in brackets ([n]). The online Help for this window contains a description of the fields that you can display.

For more information, see:

- "Diving on MPI Processes" on page 110
- "Diving on MPI Buffers" on page 111
- "About Pending Receive Operations" on page 111
- "About Unexpected Messages" on page 111
- "About Pending Send Operations" on page 112

Diving on MPI Processes

To display more detail, you can dive into fields in the Message Queue Window. When you dive into a process field, TotalView does one of the following:

- Raises its Process Window if it exists.
- Sets the focus to an existing Process Window on the requested process.
- Creates a new Process Window for the process if a Process Window doesn't exist.

Diving on MPI Buffers

When you dive into the buffer fields, TotalView opens a Variable Window. It also guesses what the correct format for the data should be based on the buffer length and the data alignment. You can edit the **Type** field within the Variable Window, if necessary.

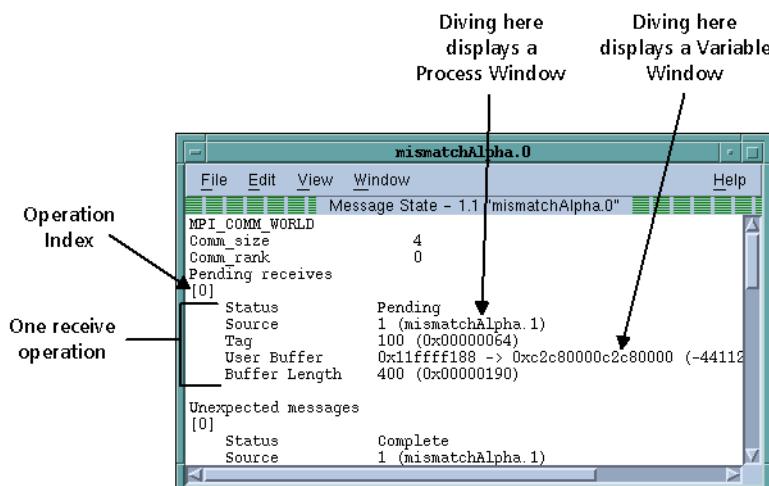


TotalView doesn't use the MPI data type to set the buffer type.

About Pending Receive Operations

TotalView displays each pending receive operation in the **Pending receives** list. Figure 90 shows an example of an MPICH pending receive operation.

Figure 90: Message Queue Window Showing Pending Receive Operation



*TotalView displays all receive operations maintained by the IBM MPI library. Set the environment variable **MP_EUIDEVELOP** to the value **DEBUG** if you want blocking operations to be visible; otherwise, the library only maintains nonblocking operations. For more details on the **MP_EUIDEVELOP** environment variable, see the IBM Parallel Environment Operations and Use manual.*

About Unexpected Messages

The **Unexpected messages** portion of the **Message Queue** Window shows information for retrieved and enqueued messages that are not yet matched with a receive operation.

Some MPI libraries, such as MPICH, only retrieve messages that have already been received as a side effect of calls to functions such as **MPI_Recv()** or **MPI_Iprobe()**. (In other words, while some versions of MPI may know about the message, the message may not yet be in a queue.) This means that TotalView can't list a message until after the destination process makes a call that retrieves it.

About Pending Send Operations

TotalView displays each pending send operation in the **Pending sends** list.

MPICH does not normally keep information about pending send operations. If you want to see them, start your program under TotalView control and use the **mpirun -ksq** or **-KeepSendQueue** command.

Depending on the device for which MPICH was configured, blocking send operations may or may not be visible. However, if TotalView doesn't display them, you can see that these operations occurred because the call is in the stack backtrace.

If you attach to an MPI program that isn't maintaining send queue information, TotalView displays the following message:

Pending sends : no information available

Debugging Cray MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

Specific information on debugging Cray MPI applications is located in our discussion of running TotalView on Cray platforms. See "Debugging Cray XT Applications" on page 144 for information.

Debugging HP Tru64 Alpha MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

To use TotalView with HP Tru64 Alpha MPI applications, you must use HP Tru64 Alpha MPI version 1.7 or later.

Starting TotalView on an HP Alpha MPI Job

In most cases, you start an HP Alpha MPI program by using the **dmpirun** command. The command for starting an MPI program under TotalView control is similar; it uses the following syntax:

```
{ totalview | totalviewcli } dmpirun -a dmpirun-command-line
```

This command invokes TotalView and tells it to show you the code for the main program in **dmpirun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

CLI: **dfocus p dgo**

The **dmpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them.



Problems can occur if you rerun HP Alpha MPI programs that are under TotalView control because resource allocation issues exist within HP Alpha MPI. The HP Alpha MPI documentation contains information on using mpiclean to clean up the MPI system state.

Attaching to an HP Alpha MPI Job

To attach to a running HP Alpha MPI job, attach to the **dmpirun** process that started the job. The procedure for attaching to a **dmpirun** process is the same as the procedure for attaching to other processes. For details, see "Attaching to Processes" on page 61. You can also use the **Group > Attach Subset** command which is discussed in "Attaching to Processes" on page 124.

After you attach to the **dmpirun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

Debugging HP MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

You can debug HP MPI applications on a PA-RISC 1.1 or 2.0 processor. To use TotalView with HP MPI applications, you must use HP MPI versions 1.6 or 1.7.

Starting TotalView on an HP MPI Job

TotalView lets you start an MPI program in one of the following ways:

`{ totalview | totalviewcli } program -a mpi-arguments`

This command tells TotalView to start the MPI process. TotalView then shows you the machine code for the HP MPI `mpirun` executable.

CLI: dfocus p dgo

`mpirun mpi-arguments -tv program`

This command tells MPI to start TotalView. You will need to set the TOTALVIEW environment variable to where TotalView is located in your file system when you start a program using `mpirun`. For example:

```
setenv TOTALVIEW \
/opt/totalview/bin/totalview
```

`mpirun mpi-arguments -tv -f startup_file`

This command tells MPI to start TotalView and then start the MPI processes as they are defined in the `startup_file` script. This file names the processes that MPI starts. Typically, this file has contents that are similar to:

```
-h aurora -np 8 /path/to/program
-h borealis -np 8 /path/to/program1
```

Your HP MPI documentation describes the contents of this startup file. These contents include the remote host name, environment variables, number of processes, programs, and so on. As is described in the previous example, you must set the TOTALVIEW environment variable.

Just before `mpirun` starts your MPI processes, TotalView acquires them and asks if you want to stop the processes before they start executing. If you answer **yes**, TotalView halts them before they enter the `main()` routine. You can then create breakpoints.

Attaching to an HP MPI Job

To attach to a running HP MPI job, attach to the HP MPI `mpirun` process that started the job. The procedure for attaching to an `mpirun` process is the same as the procedure for attaching to any other process. For details, see "Attaching to Processes" on page 61.

After TotalView attaches to the HP MPI `mpirun` process, it displays the same dialog box as it does with MPICH. (See step 4 on page 103 of "Attaching to an MPICH Job" on page 102.)

Debugging IBM MPI Parallel Environment (PE) Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView's ability to automatically acquire processes, you must be using release 3.1 or later of the Parallel Environment for AIX.

Topics in this section are:

- "Preparing to Debug a PE Application" on page 115
- "Starting TotalView on a PE Program" on page 116
- "Setting Breakpoints" on page 116
- "Starting Parallel Tasks" on page 117
- "Attaching to a PE Job" on page 117

Preparing to Debug a PE Application

The following sections describe what you must do before TotalView can debug a PE application.

Using Switch-Based Communications

If you're using switch-based communications (either *IP over the switch* or *user space*) on an SP computer, you must configure your PE debugging session so that TotalView can use *IP over the switch* for communicating with the TotalView Server (`tvdsrv`). Do this by setting the `-adapter_use` option to `shared` and the `-cpu_use` option to `multiple`, as follows:

- If you're using a PE host file, add `shared multiple` after all host names or pool IDs in the host file.
- Always use the following arguments on the `poe` command line:
`-adapter_use shared -cpu_use multiple`

If you don't want to set these arguments on the `poe` command line, set the following environment variables before starting `poe`:

```
setenv MP_ADAPTER_USE shared
setenv MP_CPU_USE multiple
```

When using *IP over the switch*, the default is usually `shared adapter use` and `multiple cpu use`; we recommend that you set them explicitly using one of these techniques. You must run TotalView on an SP or SP2 node. Since TotalView will be using *IP over the switch* in this case, you cannot run TotalView on an RS/6000 workstation.

Performing a Remote Login

You must be able to perform a remote login using the **rsh** command. You also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Server by using the **rsh** command with the switch host name of the node.

Setting Timeouts

If you receive communications timeouts, you can set the value of the **MP_TIMEOUT** environment variable; for example:

```
setenv MP_TIMEOUT 1200
```

If this variable isn't set, TotalView uses a **timeout** value of 600 seconds.

Starting TotalView on a PE Program

The following is the syntax for running Parallel Environment (PE) programs from the command line:

```
program [ arguments ] [ pe_arguments ]
```

You can also use the **poe** command to run programs as follows:

```
poe program [ arguments ] [ pe_arguments ]
```

If, however, you start TotalView on a PE application, you must start **poe** as TotalView's target using the following syntax:

```
{ totalview | totalviewcli } poe -a program [ arguments ] [ PE_arguments ]
```

For example:

```
totalview poe -a sendrecv 500 -rmpool 1
```

Setting Breakpoints

After TotalView is running, start the **poe** process using the **Process > Go** command.

CLI: dfocus p dgo

TotalView responds by displaying a dialog box—in the CLI, it prints a question—that asks if you want to stop the parallel tasks.

If you want to set breakpoints in your code before they begin executing, answer **Yes**. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. You can now set breakpoints and control parallel tasks in the same way as any process controlled by TotalView.

If you have already set and saved breakpoints with the **Action Point > Save All** command, and you want to reload the file, answer **No**. After TotalView loads these saved breakpoints, the parallel tasks begin executing.

```
CLI: dactions -save filename
      dactions -load filename
```

Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks with the Process Window **Group > Go** command.

```
CLI: dfocus G dgo
      Abbreviation: G
```



No parallel tasks reach the first line of code in your main routine until all parallel tasks start.

Be very cautious in placing breakpoints at or before a line that calls **MPI_Init()** or **MPL_Init()** because timeouts can occur while your program is being initialized. After you allow the parallel processes to proceed into the **MPI_Init()** or **MPL_Init()** call, allow all of the parallel processes to proceed through it within a short time. For more information on this, see "Avoid unwanted timeouts" on page 129.

Attaching to a PE Job

To take full advantage of TotalView's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on all of its nodes. In this way, TotalView acquires the processes you want to debug.

Attaching from a Node Running **poe**

To attach TotalView to **poe** from the node running **poe**:

- 1 Start TotalView in the directory of the debug target.
If you can't start TotalView in the debug target directory, you can start TotalView by editing the **tvdsrv** command line before attaching to **poe**. See "Using the Single-Process Server Launch Command" on page 87.
- 2 In the **File > New Program** dialog box, select **Attach to an existing process**, then find the **poe** process list, and attach to it by diving into it. When necessary, TotalView launches **tvdsvrs**. TotalView also updates the Root Window and opens a Process Window for the **poe** process.

```
CLI: dattach poe pid
```

- 3 Locate the process you want to debug and dive on it. TotalView responds by opening a Process Window for it. If your source code files are not displayed in the Source Pane, you might not have told TotalView where these files reside. You can fix this by invoking the **File > Search Path** command to add directories to your search path.

Attaching from a Node Not Running poe

The procedure for attaching TotalView to **poe** from a node that is not running **poe** is essentially the same as the procedure for attaching from a node that is running **poe**. Since you did not run TotalView from the node running **poe** (the startup node), you won't be able to see **poe** on the process list in the Root Window and you won't be able to start it by diving into it.

To place **poe** in this list:

- 1 Connect TotalView to the startup node. For details, see "Setting Up and Starting the TotalView Server" on page 81 and "Attaching to Processes" on page 61.
- 2 Select the **File > New Program** dialog box, and select **Attach to an existing process**.
- 3 Look for the process named **poe** and continue as if attaching from a node that is running **poe**.

```
CLI: dattach -r hostname poe poe-pid
```

Debugging IBM Blue Gene Applications

While the way in which you debug IBM Blue Gene MPI programs is identical to the way in which you debug these programs on other platforms, starting TotalView on your program differs slightly. Unfortunately, each machine is configured differently so you'll need to find information in IBM's documentation or in documentation created at your site.

Nevertheless, the remainder of this section will present some hints.

In general, you will either launch **mpirun** under debugger control or start TotalView and attach to an already running **mpirun**. For example:

```
{ totalview | totalviewcli } mpirun -a mpirun-command-line
```

TotalView tells **mpirun** to launch TotalView Debug Servers on each Blue Gene I/O nodes.

Because I/O nodes cannot resolve network names, TotalView must pass the address of the front-end node interface to the servers on the I/O nodes. This is usually not the same interface that is generally used to connect to

the front-end node. TotalView assumes that the address can be resolved by using a name that is:

`front-end-hostname-io`.

For example, if the hostname of the front-end is `fred`, the servers will connect to `fred-io`.



The systems at the IBM Blue Gene Capacity on Demand follow this convention. If you are executing programs there, you will not need to set the TotalView variables described in the rest of this section.

If the front-end cannot resolve this name, you must supply the name of the interface using the `-local_interface` command-line option or by setting the `bluegene_io_interface` TotalView variable. (This variable is described in the Chapter 4 of the *TotalView Reference Guide*.)

Because the same version of TotalView must be able to debug both Power-Linux programs (for example, `mpirun`) and Blue Gene programs, TotalView uses a Blue Gene-specific server launch string. You can define this launch string by setting the `bluegene_server_launch_string` TotalView variable or command-line option.



You must set this variable in a tvdrc file. This differs from other TotalView launch strings, which you can set using the File > Preferences Dialog Box.

The default value for the `bluegene_server_launch_string` variable is:

`-callback %L -set_pw %P -verbosity %V %F`

In this string, `%L` is the address of the front-end node interface used by the servers. The other substitution arguments have the same meaning as they do in a normal server launch string. These substitution arguments are discussed in Chapter 7 of the *TotalView Reference Guide*.

Debugging LAM/MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

The following is a description of the LAM/MPI implementation of the MPI standard. Here are the first two paragraphs of Chapter 2 of the "LAM/MPI User's Guide". You can find this document by going to the LAM documentation page, which is: <http://www.lam-mpi.org/using/docs/>.

"LAM/MPI is a high-performance, freely available, open source implementation of the MPI standard that is researched, developed, and maintained at the Open Systems Lab at Indiana University. LAM/MPI supports all of

the MPI-1 Standard and much of the MPI-2 standard. More information about LAM/MPI, including all the source code and documentation, is available from the main LAM/MPI web site.

"LAM/MPI is not only a library that implements the mandated MPI API, but also the LAM run-time environment: a user-level, daemon-based run-time environment that provides many of the services required by MPI programs. Both major components of the LAM/MPI package are designed as component frameworks—extensible with small modules that are selectable (and configurable) at run-time. ...

You debug a LAM/MPI program in a similar way to how you debug most MPI programs. Use the following syntax if TotalView is in your path:

```
mpirun -tv args prog prog_args
```

As an alternative, you can invoke TotalView on **mpirun**:

```
totalview mpirun -a prog prog_args
```

The LAM/MPI User's Guide discusses how to use TotalView to debug LAM/MPI programs.

Debugging QSW RMS Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

TotalView supports automatic process acquisition on AlphaServer SC systems and 32-bit Red Hat Linux systems that use Quadrics RMS resource management system with the QSW switch technology.



Message queue display is only supported if you are running version 1, patch 2 or later, of AlphaServer SC.

Starting TotalView on an RMS Job

To start a parallel job under TotalView control, use TotalView as if you were debugging **prun**:

```
{ totalview | totalviewcli } prun -a prun-command-line
```

TotalView starts and shows you the machine code for RMS **prun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

```
CLI: dfocus p dgo
```

The RMS **prun** command executes and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** process the same way you attach to other processes. For details on attaching to processes, see "Attaching to Processes" on page 61.

After you attach to the RMS **prun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPI processes.

As an alternative, you can use the **Group > Attach Subset** command to pre-define what TotalView should do. For more information, see "Attaching to Processes" on page 124.

Debugging SiCortex MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98

Specific information on debugging SiCortex MPI applications is located in our discussion of running TotalView on SiCortex platforms. See "Debugging SiCortex Applications" on page 147 for information.

Debugging SGI MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

TotalView can acquire processes started by SGI MPI applications. This MPI is part of the Message Passing Toolkit (MPT) 1.3 and 1.4 packages. TotalView can display the Message Queue Graph Window for these releases. See "Displaying the Message Queue Graph Window" on page 107 for message queue display.

Starting TotalView on an SGI MPI Job

You normally start SGI MPI programs by using the **mpirun** command. You use a similar command to start an MPI program under debugger control, as follows:

```
{ totalview | totalviewcli } mpirun -a mpirun-command-line
```

This invokes TotalView and tells it to show you the machine code for **mpirun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

CLI: dfocus p dgo

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **Yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message that shows the name of the array and the value of the array services handle (**ash**) to which it is attaching.

Attaching to an SGI MPI Job

To attach to a running SGI MPI program, attach to the SGI MPI **mpirun** process that started the program. The procedure for attaching to an **mpirun** process is the same as the procedure for attaching to any other process. For details, see "Attaching to Processes" on page 61.

After you attach to the **mpirun** process, TotalView asks if you also want to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

If you choose **Yes**, TotalView starts the server processes and acquires all MPICH processes.

As an alternative, you can use the **Group > Attach Subset** command to pre-define what TotalView will do. For more information, see "Attaching to Processes" on page 124.

Using ReplayEngine with SGI MPI

SGI MPI uses the **xpmem** module to map memory from one MPI process to another during job startup. Memory mapping is enabled by default. The size of this mapped memory can be quite large, and can have a negative effect on TotalView's ReplayEngine performance. Therefore, we have limited

mapped memory by default for the **xpmem** module if Replay is enabled. The environment variable, **MPI_MEMMAP_OFF**, is set to 1 in the TotalView file **parallel_support.tvd** by adding the variable to the **replay_env**: specification as follows: **replay_env: MPI_MEMMAP_OFF=1**.

If full memory mapping is required, you can set the startup environment variable in the Startup Parameters dialog window (in the Arguments tab). Add the following to the environment variables: **MPI_MEMMAP_OFF=0**.

Be aware that the default mapped memory size may prove to be too large for ReplayEngine to deal with, and it could be quite slow. You can limit the size of the mapped heap area by using the **MPI_MAPPED_HEAP_SIZE** environment variable documented in the SGI documentation. After turning off **MEMMAP_OFF** as described above, you can set the size (in bytes) in the TotalView startup parameters.

For example:

MPI_MAPPED_HEAP_SIZE=1048576



SGI has a patch for an MPT/XPMEM issue. Without this patch, XPMEM can crash the system if ReplayEngine is turned on. To get the XPMEM fix for the munmap problem, either upgrade to ProPack 6 SP 4 or install SGI patch 10570 on top of ProPack 6 SP 3.

Debugging Sun MPI Applications



In many cases, you can bypass the procedure described in this section. For more information, see "Debugging MPI Programs" on page 98.

TotalView can debug a Sun MPI program and can display Sun MPI message queues. This section describes how to perform *job startup* and *job attach* operations.

To start a Sun MPI application, use the following procedure.

1 Type the following command:

totalview mprun [totalview_args] -a [mpi_args]

For example:

totalview mprun -g blue -a -np 4 /usr/bin/mpi/conn.x

CLI: totalviewcli mprun [totalview_args] -a [mpi_args]

When the TotalView Process Window appears, select the **Go** button.

CLI: dfocus p dgo

TotalView may display a dialog box with the following text:

Process mprun is a parallel job. Do you want to stop the job now?

- 2 If you compiled using the **-g** option, click **Yes** to tell TotalView to open a Process Window that shows your source. All processes are halted.

Attaching to a Sun MPI Job

To attach to an already running **mprun** job:

- 1 Find the host name and process identifier (PID) of the **mprun** job by typing **mpps -b**. For more information, see the **mpps(1M)** manual page.

The following is sample output from this command:

JOBNAME	MPRUN_PID	MPRUN_HOST
cre.99	12345	hpc-u2-9
cre.100	12601	hpc-u2-8

- 2 After selecting **File > New Program**, type **mprun** in the **Executable** field and type the PID in the **Process ID** field.

CLI: **dattach mprun mprun-pid**
For example:
dattach mprun 12601

- 3 If TotalView is running on a different node than the **mprun** job, enter the host name in the **Remote Host** field.

CLI: **dattach -r host-name mprun mprun-pid**

Debugging Parallel Applications Tips

This section contains information about debugging parallel programs:

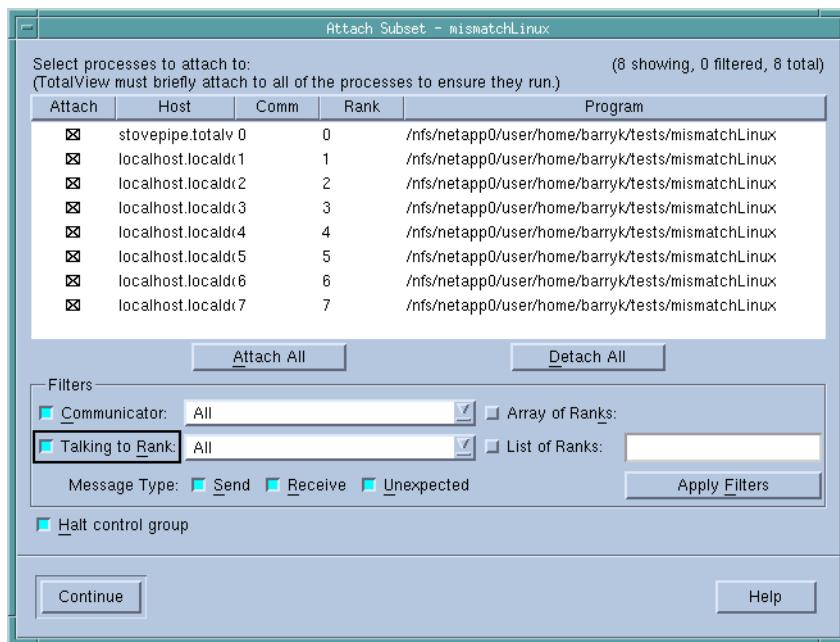
- “*Attaching to Processes*” on page 124
- “*Parallel Debugging Tips*” on page 127
- “*MPICH Debugging Tips*” on page 129
- “*IBM PE Debugging Tips*” on page 129

Attaching to Processes

In a typical multi-process job, you’re interested in what’s occurring in some of your processes and not as much interested in others. By default, TotalView tries to attach to all of the processes that your program starts. If there are a lot of processes, there can be considerable overhead involved in opening and communicating with the jobs.

You can minimize this overhead by using the **Group > Attach Subset** command, which displays the dialog box shown in Figure 91.

Figure 91: Group > Attach Subset Dialog Box



TotalView lets you start MPI jobs in two ways. One requires that the starter program be under TotalView control and have special instrumentation for TotalView while the other does not. In the first case, you will enter the name of the starter program on the command line. The other requires that you enter information into the File > New Program or Process > Startup Parameters dialog boxes. The **Attach Subset** command is only available if you directly name a starter program on the command line.

Selecting boxes on the left side of the list tells TotalView which processes it should attach to. Although your program will launch all of these processes, TotalView only attaches to the processes that you have selected.

The controls under the **All** and the **None** buttons let you limit which processes TotalView automatically attaches to, as follows:

- The **Communicator** control specifies that the processes must be involved with the communicators that you select. For example, if something goes wrong that involves a communicator, selecting it from the list tells TotalView to only attach to the processes that use that communicator.
- The **Talking to Rank** control further limits the processes to those that you name here. Most of the entries in this list are just the process numbers. In most cases, you would select **All** or **MPI_ANY_SOURCE**.
- The three checkboxes in the **Message Type** area add yet another qualifier. Checking a box tells TotalView to only display communicators that are involved with a **Send**, **Receive**, or **Unexpected** message.

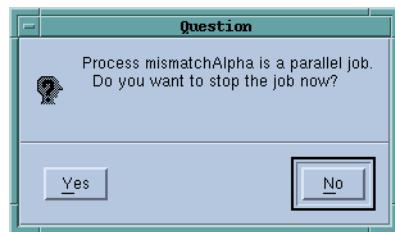
After you find the problem, you can detach from these nodes by selecting **None**. In most cases, use the **All** button to select all the check boxes, then clear the ones that you're not interested in.

Debugging Parallel Applications Tips

Many applications place values that indicate the rank in a variable so that the program can refer to them as they are needed. If you do this, you can display the variable in a Variable Window and then select the **Tools > Attach Subset (Array of Ranks)** command to display this dialog box.

You can use the **Group > Attach Subset** command at any time, but you would probably use it immediately before TotalView launches processes. Unless you have set preferences otherwise, TotalView stops and asks if you want it to stop your processes. When selected, the **Halt control group** check box also tells TotalView to stop a process just before it begins executing. (See Figure 92.)

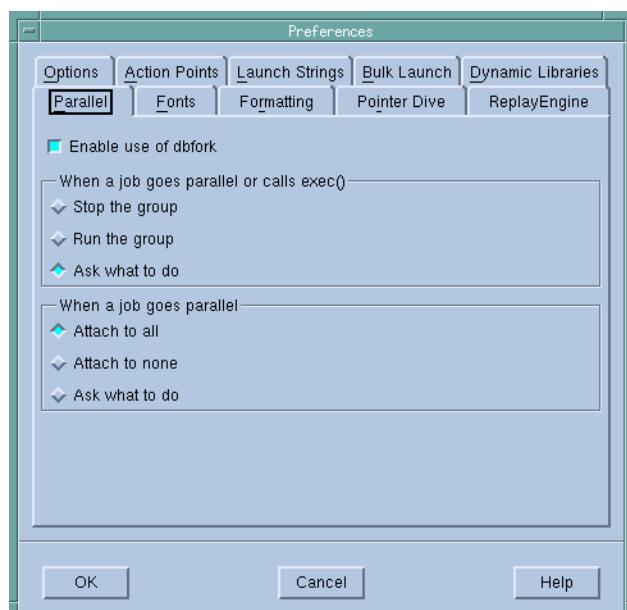
Figure 92: Stop Before Going Parallel Question Box



If you click Yes, when the job stops the starter process should be at a "magic breakpoint." These are set by TotalView behind the scene, and usually not visible. The other processes may or may not be at a "magic breakpoint."

The commands on the Parallel Page in the **File > Preferences** Dialog Box let you control what TotalView does when your program goes parallel. (See Figure 93.)

Figure 93: File > Preferences: Parallel Page





TotalView only displays the preceding question box when you directly name a starter program on the command line.

The radio button in the **When a job goes parallel or calls exec()** area lets TotalView:

- **Stop the group:** Stop the control group immediately after the processes are created.
- **Run the group:** Allow all newly created processes in the control group to run freely.
- **Ask what to do:** Ask what should occur. If you select this option, TotalView asks if it should start the created processes.

CLI: `dset TV::parallel_stop`

The radio buttons in the **When a job goes parallel** area let TotalView:

- **Attach to all:** Automatically attach to all processes when they begin executing.
- **Attach to none:** Does not attach to any created process when it begins executing.
- **Ask what to do:** Asks what should occur. If you select this option, TotalView opens the same dialog box that is displayed when you select **Group > Attach Subset**. TotalView then attaches to the processes that you have selected. This dialog box isn't displayed when you set the preference. Instead, it controls what happens when your program creates parallel processes.

CLI: `dset TV::parallel_attach`

Parallel Debugging Tips

The following tips are useful for debugging most parallel programs:

■ Setting Breakpoint behavior

When you're debugging message-passing and other multi-process programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multi-process program hits a breakpoint, TotalView stops all the other processes.

To change the default stopping action of breakpoints and barrier breakpoints, you can set debugger preferences. The online Help contains information on these preference. These preferences tell TotalView whether to continue to run when a process or thread hits the breakpoint.

These options only affect the default behavior. You can choose a behavior for a breakpoint by setting the breakpoint properties in the **File > Preferences** Action Points Page. See "Setting Breakpoints for Multiple Processes" on page 359.

■ Synchronizing Processes

TotalView has two features that make it easier to get all of the processes in a multi-process program synchronized and executing a line of code.

Process barrier breakpoints and the process hold/release features work together to help you control the execution of your processes. See "Setting Barrier Points" on page 362.

The Process Window **Group > Run To** command is a special stepping command. It lets you run a group of processes to a selected source line or instruction. See "Stepping (Part I)" on page 253.

■ Using group commands

Group commands are often more useful than process commands.

It is often more useful to use the **Group > Go** command to restart the whole application instead of the **Process > Go** command.

CLI:	<code>dfocus g dgo</code>
Abbreviation:	G

You would then use the **Group > Halt** command instead of **Process > Halt** to stop execution.

CLI:	<code>dfocus g dhalt</code>
Abbreviation:	H

The group-level single-stepping commands such as **Group > Step** and **Group > Next** let you single-step a group of processes in a parallel. See "Stepping (Part I)" on page 253.

CLI:	<code>dfocus g dstep</code>
Abbreviation:	S
	<code>dfocus g dnext</code>
	Abbreviation: N

■ Stepping at Process-level

If you use a process-level single-stepping command in a multi-process program, TotalView may appear to hang (it continuously displays the watch cursor). If you single-step a process over a statement that can't complete without allowing another process to run, and that process is stopped, the stepping process appears to hang. This can occur, for example, when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by selecting **Cancel** in the **Waiting for Command to Complete** window that TotalView displays. As an alternative, consider using a group-level single-step command.

CLI:	Type <code>Ctrl+C</code>
------	--------------------------



TotalView Technologies receives many bug reports about processes being hung. In almost all cases, the reason is that one process is waiting for another. Using the Group debugging commands almost always solves this problem.

■ Determining which processes and threads are executing

The Root Window helps you determine where various processes and threads are executing. When you select a line of code in the Process Window, the Root Window updates to show which processes and threads are executing that line.

■ Viewing variable values

You can view the value of a variable that is replicated across multiple processes or multiple threads in a single Variable Window. See "Displaying a Variable in all Processes or Threads" on page 345.

■ Restarting from within TotalView

You can restart a parallel program at any time. If your program runs past the point you want to examine, you can kill the program by selecting the **Group > Kill** command. This command kills the master process and all the slave processes. Restarting the master process (for example, **mpirun** or **poe**) recreates all of the slave processes. Start up is faster when you do this because TotalView doesn't need to reread the symbol tables or restart its **tvdsvr** processes, since they are already running.

```
CLI: dfocus g dkill
```

MPICH Debugging Tips

The following debugging tips apply only to MPICH:

■ Passing options to mpirun

You can pass options to TotalView using the MPICH **mpirun** command. To pass options to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **-no_stop_all** option, as in the following C shell example:

```
setenv TOTALVIEW "totalview -no_stop_all"
```

■ Using ch_p4

If you start remote processes with MPICH/**ch_p4**, you may need to change the way TotalView starts its servers.

By default, TotalView uses **rsh** to start its remote server processes. This is the same behavior as **ch_p4** uses. If you configure **ch_p4** to use a different start-up mechanism from another process, you probably also need to change the way that TotalView starts the servers.

For more information about **tvdsvr** and **rsh**, see "Setting Single-Process Server Launch Options" on page 83. For more information about **rsh**, see "Using the Single-Process Server Launch Command" on page 87.

IBM PE Debugging Tips

The following debugging tips apply only to IBM MPI (PE):

■ Avoid unwanted timeouts

Timeouts can occur if you place breakpoints that stop other processes too soon after calling **MPI_Init()** or **MPL_Init()**. If you create "stop all" breakpoints, the first process that gets to the breakpoint stops all the

other parallel processes that have not yet arrived at the breakpoint. This can cause a timeout.

To turn the option off, select the Process Window **Action Point > Properties** command while the line with the stop symbol is selected. After the **Properties** Dialog Box appears, select the **Process** button in the **When Hit, Stop** area, and also select the **Plant in share group** button.

```
CLI: dbarrier location -stop_when_hit process
```

■ Control the poe process

Even though the **poe** process continues under debugger control, do not attempt to start, stop, or otherwise interact with it. Your parallel tasks require that **poe** continues to run. For this reason, if **poe** is stopped, TotalView automatically continues it when you continue any parallel task.

■ Avoid slow processes due to node saturation

If you try to debug a PE program in which more than three parallel tasks run on a single node, the parallel tasks on each node can run noticeably slower than they would run if you were not debugging them.

In general, the number of processes running on a node should be the same as the number of processors in the node.

This becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks does not progress. This is because PE uses the **SIGALRM** signal to implement communications operations, and AIX requires that debuggers must intercept all signals. As the number of parallel tasks on a node increases, TotalView becomes saturated and can't keep up with the **SIGALRM** signals being sent, thus slowing the tasks.

Setting Up Parallel Debugging Sessions



This chapter explains how to set up TotalView parallel debugging sessions for applications that use the parallel execution models that TotalView supports and which do not use MPI.



If you are using TotalView Individual, all of your program's processes must execute on the computer on which you installed TotalView. In addition, TotalView Individual limits you to no more than 16 processes and threads.

This chapter contains the following topics:

- “Debugging OpenMP Applications” on page 132
- “Using SLURM” on page 137
- “Debugging IBM Cell Broadband Engine Programs” on page 138
- “Debugging Cray XT Applications” on page 144
- “Debugging Global Arrays Applications” on page 148
- “Debugging PVM (Parallel Virtual Machine) and DPVM Applications” on page 151
- “Debugging Shared Memory (SHMEM) Code” on page 156
- “Debugging UPC Programs” on page 158

This chapter also describes TotalView features that you can use with most parallel models:

- TotalView lets you decide which process you want it to attach to. See “Attaching to Processes” on page 124.
- See “Debugging Parallel Applications Tips” on page 124 for hints on how to approach debugging parallel programs.

Debugging OpenMP Applications

TotalView supports many OpenMP C and Fortran compilers. Supported compilers and architectures are listed in the *TotalView Platforms and Systems Requirements* document, which is on our Web site.

The following are some of the features that TotalView supports:

- Source-level debugging of the original OpenMP code.
- The ability to plant breakpoints throughout the OpenMP code, including lines that are executed in parallel.
- Visibility of OpenMP worker threads.
- Access to **SHARED** and **PRIVATE** variables in OpenMP PARALLEL code.
- A stack-back link token in worker threads' stacks so that you can find their master stack.
- Access to OMP THREADPRIVATE data in code compiled by the IBM and Guide, SGI IRIX, and HP Alpha compilers.

The code examples used in this section are included in the TotalView distribution in the `examples/omp_simplef` file.



On the SGI IRIX platform, you must use the MIPSpro 7.3 compiler or later to debug OpenMP.

Topics in this section are:

- "Debugging OpenMP Programs" on page 132
- "Viewing OpenMP Private and Shared Variables" on page 134
- "Viewing OpenMP THREADPRIVATE Common Blocks" on page 136
- "Viewing the OpenMP Stack Parent Token Line" on page 137

Debugging OpenMP Programs

The way in which you debug OpenMP code is similar to the way you debug multi-threaded code. The major differences are related to the way the OpenMP compiler alters your code. These alterations include:

- The most visible transformation is *outlining*. The compiler pulls the body of a parallel region out of the original routine and places it in an *outlined routine*. In some cases, the compiler generates multiple outlined routines from a single parallel region. This allows multiple threads to execute the parallel region.
The outlined routine's name is based on the original routine's name. In most cases, the compiler adds a numeric suffix.
- The compiler inserts calls to the OpenMP runtime library.

- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables reside in the master thread's original routine, and private variables reside in the outlined routine.
- The master thread creates threads to share the workload. As the master thread begins to execute a parallel region in the OpenMP code, it creates the worker threads, dispatches them to the outlined routine, and then calls the outlined routine itself.

About TotalView OpenMP Features

TotalView interprets the changes that the OpenMP compiler makes to your code so that it can display your program in a coherent way. Here are some things you should know:

- The compiler can generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions.
- You can't single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and let the process run to it. After execution reaches the parallel region, you can single step in it.
- OpenMP programs are multi-threaded programs, so the rules for debugging multi-threaded programs apply.

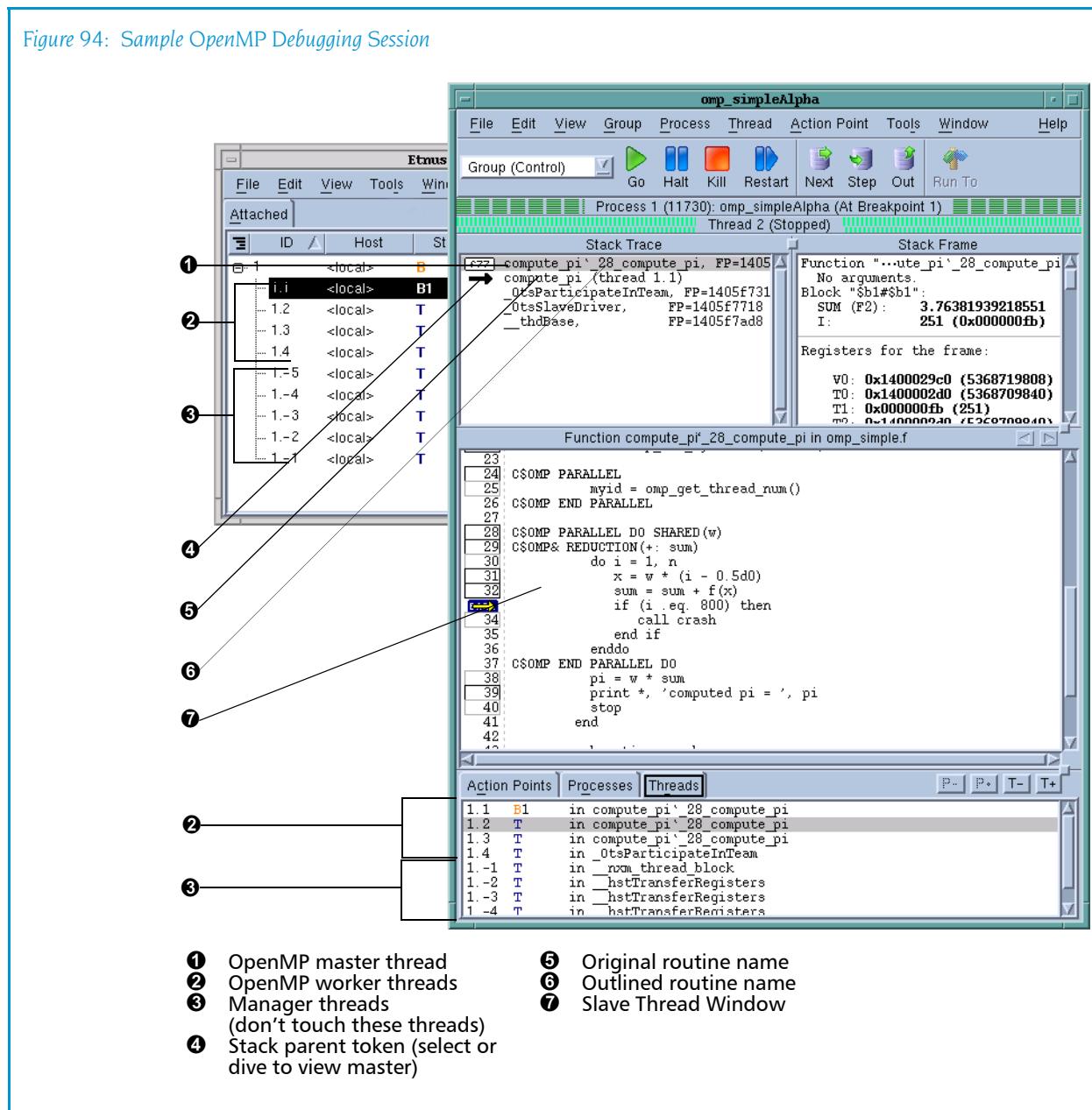
Figure 94 on page 134 shows a sample OpenMP debugging session.

About OpenMP Platform Differences

In general, TotalView smooths out the differences that occur when you execute OpenMP platforms on different platforms. The following list discusses these differences:

- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.
- When you stop the OpenMP worker threads in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
 - Outlined routine called from the special stack parent token line.
 - The OpenMP runtime library called from.
 - The original routine (containing the parallel region).
- On HP Alpha Tru64 UNIX, the system manager threads have a negative thread ID; since these threads are created by OpenMP they are not part of the code you wrote. Consequently, you should never manipulate them.
- On HP Alpha Tru64 UNIX and on the Guide compilers, the OpenMP threads are implemented by the compiler as **pthreads**. On SGI IRIX, they are implemented as **sprocs**. TotalView shows the threads' logical and/or system thread ID, not the OpenMP thread number.
- SGI OpenMP uses the **SIGTERM** signal to terminate threads. Because TotalView stops a process when the process receives a **SIGTERM**, the OpenMP process doesn't terminate. If you want the OpenMP process to terminate instead of stop, set the default action for the **SIGTERM** signal to **Resend**.

Figure 94: Sample OpenMP Debugging Session



Viewing OpenMP Private and Shared Variables

TotalView lets you view both OpenMP private and shared variables.

The compiler maintains OpenMP private variables in the outlined routine, and treats them like local variables. See “*Displaying Local Variables and Registers*” on page 292. In contrast, the compiler maintains OpenMP shared variables in the master thread’s original routine stack frame. However, Guide compilers pass shared variables to the outlined routine as parameter references.

TotalView lets you display shared variables through a Process Window focused on the OpenMP master thread, or through one of the OpenMP worker threads.

To see these variables, you must:

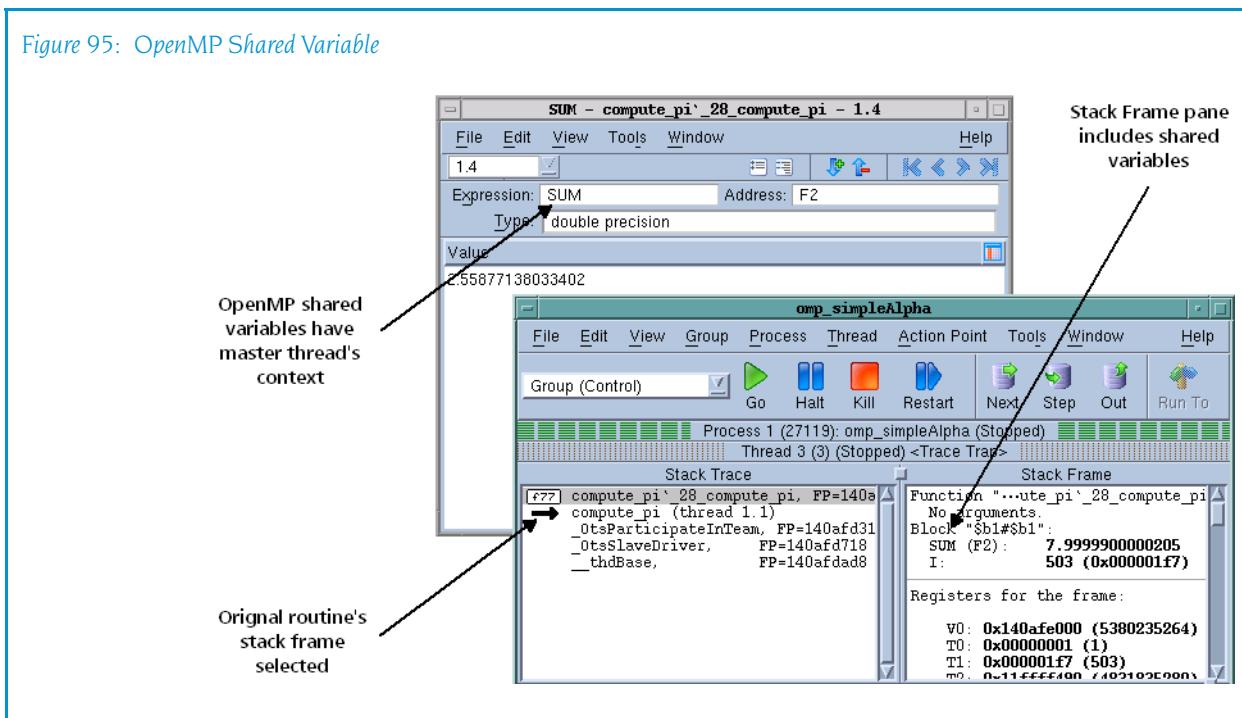
- 1 Select the outlined routine in the Stack Trace Pane, or select the original routine stack frame in the OpenMP master thread.
- 2 Dive on the variable name, or select the **View > Lookup Variable** command. When prompted, enter the variable name.

CLI: `dprint`

You will need to set your focus to the OpenMP master thread first.

TotalView opens a Variable Window that displays the value of the OpenMP shared variable, as shown in Figure 95.

Figure 95: OpenMP Shared Variable



Shared variables reside in the OpenMP master thread's stack. When displaying shared variables in OpenMP worker threads, TotalView uses the stack context of the OpenMP master thread to find the shared variable. TotalView uses the OpenMP master thread's context when displaying the shared variable in a Variable Window.

You can also view OpenMP shared variables in the Stack Frame Pane by selecting either of the following:

- Original routine stack frame in the OpenMP master thread.
- Stack parent token line in the Stack Trace Pane of OpenMP worker threads, as shown in the following figure.

Viewing OpenMP **THREADPRIVATE** Common Blocks

The HP Alpha Tru64 UNIX OpenMP and SGI IRIX compilers implement OpenMP **THREADPRIVATE** common blocks by using the thread local storage system facility. This facility stores a variable declared in OpenMP **THREADPRIVATE** common blocks at different memory locations in each thread in an OpenMP process. This allows the variable to have different values in each thread. In contrast, the IBM and Guide compilers use the pthread key facility.

When you use SGI compilers, the compiler maps **THREADPRIVATE** variables to the same virtual address. However, they have different physical addresses.

To view a variable in an OpenMP **THREADPRIVATE** common block, or the OpenMP **THREADPRIVATE** common block:

- 1 In the Threads Tab of the Process Window, select the thread that contains the private copy of the variable or common block you want to view.
- 2 In the Stack Trace Pane of the Process Window, select the stack frame that lets you access the OpenMP **THREADPRIVATE** common block variable. You can select either the outlined routine or the original routine for an OpenMP master thread. You must, however, select the outlined routine for an OpenMP worker thread.
- 3 From the Process Window, dive on the variable name or common block name, or select the **View > Lookup Variable** command. When prompted, enter the name of the variable or common block. You may need to append an underscore character (_) after the common block name.

CLI: `dprint`

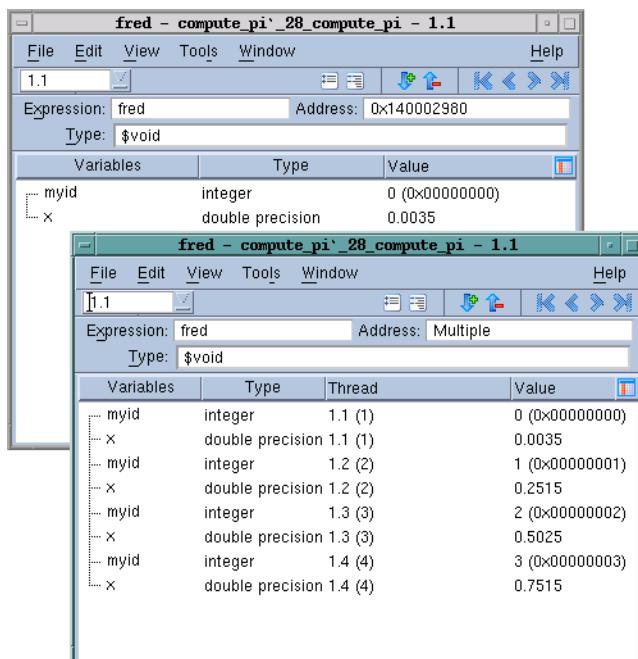
TotalView opens a Variable Window that displays the value of the variable or common block for the selected thread.

See “*Displaying Variables*” on page 283 for more information on displaying variables.

- 4 To view OpenMP **THREADPRIVATE** common blocks or variables across all threads, use the Variable Window’s **Show across > Threads** command. See “*Displaying a Variable in all Processes or Threads*” on page 345.

The following figure shows Variable Windows displaying OpenMP **THREADPRIVATE** common blocks. Because the Variable Window has the same thread context as the Process Window from which it was created, the title bar patterns for the same thread match. TotalView displays the values of the common block across all threads when you use the **View > Show Across > Threads** command. (See Figure 96 on page 137.)

Figure 96: OpenMP
THREADPRIVATE
Common Block Variables

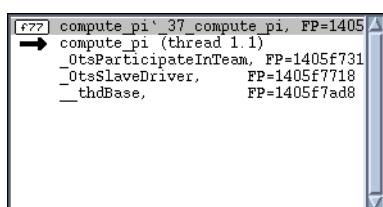


Viewing the OpenMP Stack Parent Token Line

TotalView inserts a special stack parent token line in the Stack Trace Pane of OpenMP worker threads when they are stopped in an outlined routine.

When you select or dive on the stack parent token line, the Process Window switches to the OpenMP master thread, allowing you to see the stack context of the OpenMP worker thread's routine. (See Figure 97.)

Figure 97: OpenMP Stack Parent Token Line



This stack context includes the OpenMP shared variables.

Using SLURM

Beginning at Version 7.1, TotalView supports the SLURM resource manager. Here is some information copied from the SLURM website (<http://www.llnl.gov/linux/slurm/>).

SLURM is an open-source resource manager designed for Linux clusters of all sizes. It provides three key functions. First it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work.

SLURM is not a sophisticated batch system, but it does provide an Applications Programming Interface (API) for integration with external schedulers such as the Maui Scheduler. While other resource managers do exist, SLURM is unique in several respects:

- Its source code is freely available under the GNU General Public License.
- It is designed to operate in a heterogeneous cluster with up to thousands of nodes.
- It is portable; written in C with a GNU autoconf configuration engine. While initially written for Linux, other UNIX-like operating systems should be easy porting targets. A plugin mechanism exists to support various interconnects, authentication mechanisms, schedulers, etc.
- SLURM is highly tolerant of system failures, including failure of the node executing its control functions.
- It is simple enough for the motivated end user to understand its source and add functionality.

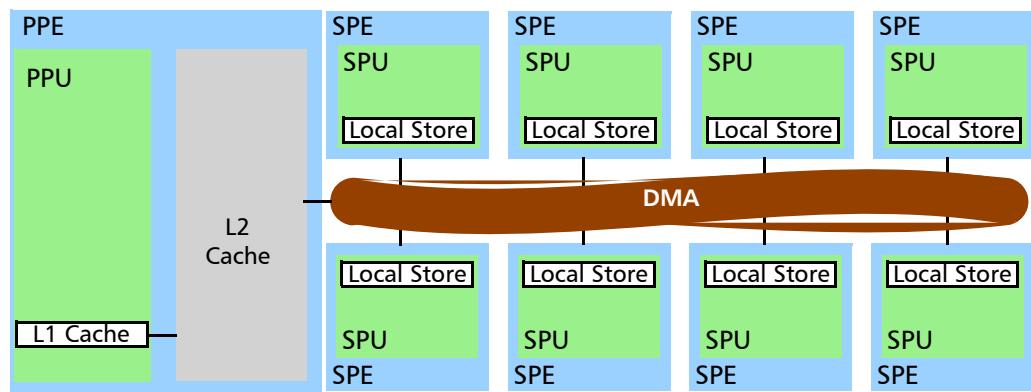
Debugging IBM Cell Broadband Engine Programs

The IBM Cell Broadband Engine is a heterogeneous computer having a PPE (PowerPC Processor Element) and eight SPEs (Synergistic Processor Elements). Despite being a heterogeneous computer, the way you debug Cell programs is nearly identical to the way you use TotalView to debug programs running on other architectures. (See Figure 98 on page 139.)

Of course, the way in which programs are loaded and execute mean there are a few differences. For example, when a context is created on an SPU (Synergistic Processor Unit), this context is not initialized; instead, resources are simply allocated. This empty context is visible, but there are no stack traces or source displays. There is no equivalent to this on other architectures that TotalView supports. At a later time, the PPU (PowerPC Processor Unit) will load an SPU image into this context and tell it to run.

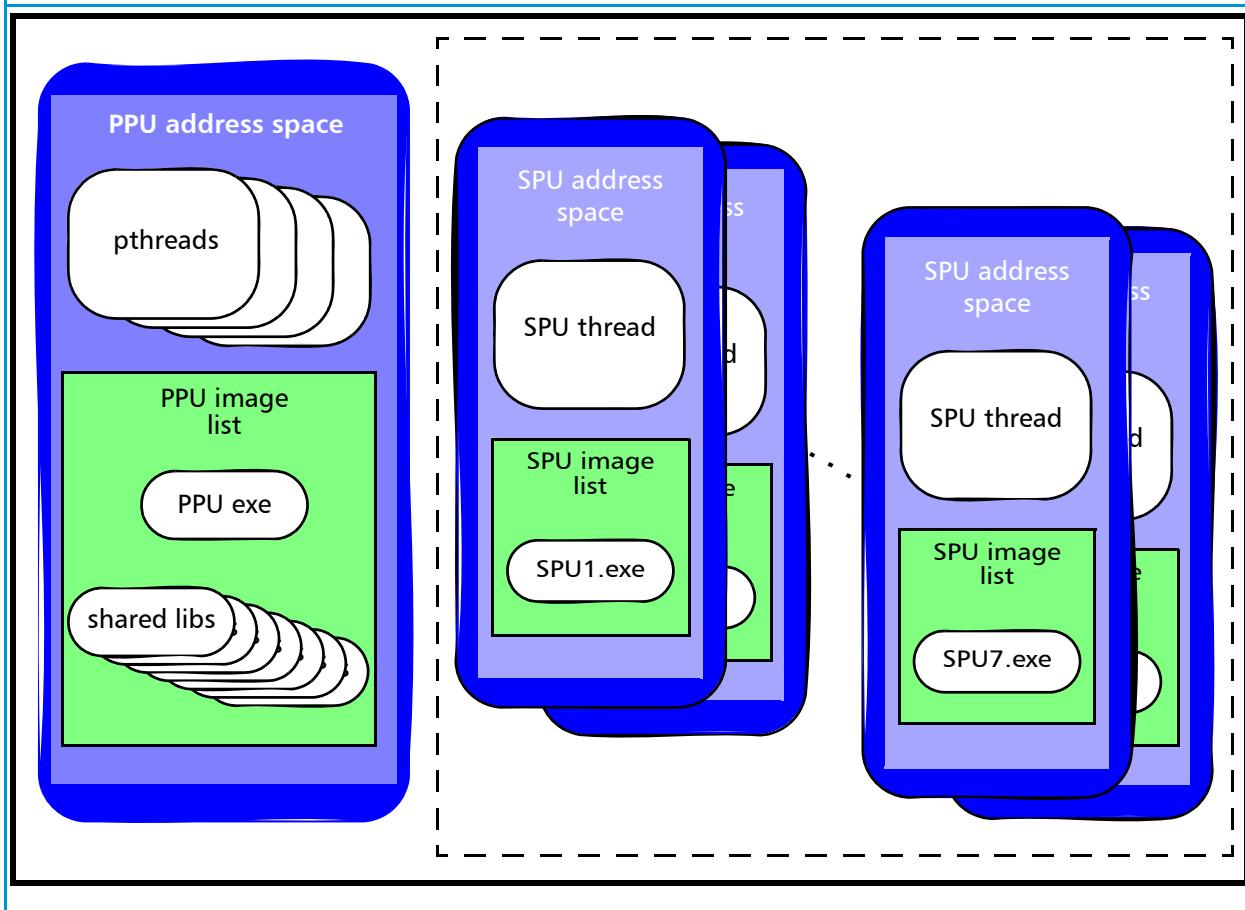
In all cases, when you focus on a PPU thread, only the PPU address space is visible. Similarly, when you focus on an SPU thread, the address space of that SPU thread is visible.

Figure 98: Cell Architecture



TotalView looks at the executing program in a slightly different way. The following illustration shows how TotalView models the processes running on the Cell.

Figure 99: A Cell Process



The TotalView separates the PPU address space from that of each SPU. The PPU has its own image. In addition, it uses pthreads to launch SPU contexts. TotalView manages each context individually. This structuring lets you see

the PPU and each SPU separately. For example, you can switch from one to another by diving on it in the Root Window.

The PPU

When debugging Cell programs, you need to be aware of the two different architectures as they have different behaviors. The PPU is a standard IBM Linux PowerPC. You interact with programs running on the PPU in exactly the same way as you interact with standard Linux PowerPC programs.

As it is typical to run multiple Cell machines together, each PPU process running the same program is placed in one share group. If more than one executable is being run, each set of identical programs is placed in its own share group. All of these share groups are contained within one control group. The way in which TotalView groups processes and threads is identical to how it groups them on all other architectures.

The SPU

The programs running on the SPU are handled in a slightly different way. On other architectures, a share group only contains processes. On the Cell, TotalView places SPU threads that are running the same executable into separate share groups. That is, on SPUs, a share group contains threads, not processes. SPU share groups can contain SPU threads running on other Cells, and these Cells can be within the same blade or within blades in a cluster.

Cell Programming

While there are several ways to program the Cell, the primary method uses the front-end PPU to load programs into the SPUs.

The SPU image can be embedded within the same executable that contains the PPU executable or it can be contained within shared libraries.

PPU and SPU Executable Organization

Typically, a Cell executable file is usually organized in one of the following ways:

- The PPU executable file contains all code that runs on the PPU and SPUs. That is, the SPU executables are embedded within a PPU executable file or shared library.
- The PPU executable file only contains the code that runs on the PPU. The SPU executable file opened at runtime by a PPU thread using a **libspe**/ **libspe2** function. The opened file is then loaded into the SPU using the same functions that load an embedded SPU image.
- A combination of the previous two methods.

In general, executing an SPU thread is a multi-step process. The PPU begins by create an SPE context. This context can be thought of as being a thread that is not running: it has an address space, registers, MFC state, and so on.) This thread has all memory, registers, and the like set to zero.

The second step is to load the executable into the SPE context. The final step is to run the SPU executable.

PPU and SPU Executable Naming

If your SPU programs are embedded into your PPU program, TotalView names the SPU program in the following way:

ppu.exe(spu.exe@offset)

ppu.exe is the name of the PPU image containing the SPU executable and *offset* is the file offset within *ppu.exe*. *spu.exe* is the name of the SPU executable.

If, however, the SPU executable is opened at runtime using **libspe**, the SPU name is shown as:

spu.exe

Thread IDs

Another difference is the TotalView TID (Thread ID). An SPU thread has a negative number. This differentiates it from the positive TIDs used for PPU threads. Negative TIDs on other architectures have a different meaning. On these other architectures, a negative TID represents a manager thread that is performing an activity for which you have little interest. (See Figure 100.)

Figure 100: Root Window for a Cell Program

ID	Rank	Host	Status	Description
1	<local>	B	euler_tuned_multi_spe (8 active t	
1.1	<local>	B3	in main	
1.3	<local>	T	in syscall	
1.4	<local>	T	in syscall	
1.5	<local>	T	in syscall	
1.-1	<local>	T	in _exit	
1.-2	<local>	T	in process_buffer	
1.-3	<local>	T	in process_buffer	
1.-4	<local>	T	in main	

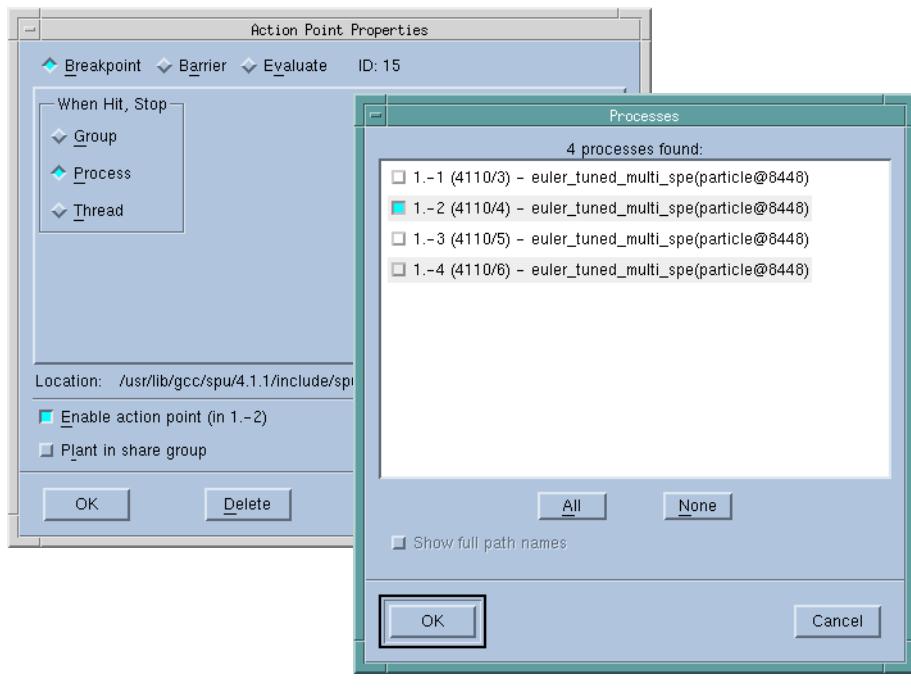
All SPU threads can be run synchronously or asynchronously. Like all threads within TotalView, you can use the CLI to place SPU threads in a named group and control these named groups separately.

Breakpoints

An SPU thread share group shares one characteristic of a process share group: breakpoints can be shared across the threads. That is, when you plant a breakpoint in one thread in the share group, TotalView can plant it in all members. (See Figure 101 on page 142.))

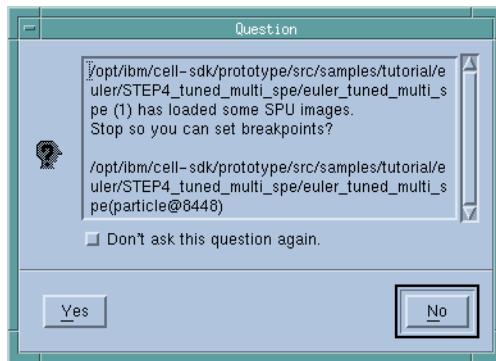
Breakpoints can be in a pending state; that is, if TotalView cannot assign a breakpoint to an address because an SPU executable is not yet loaded, TotalView will wait until it can set the breakpoint.

Figure 101: Action Point Properties Dialog Box



When the PPU thread loads an SPU executable into an SPU context, TotalView stops execution and asks if you want to set a breakpoint. Figure 102 is an example of the question box that you will see.

Figure 102: Stop to Set Breakpoints Question



If you would like to control whether this question is asked, use the following CLI variables:

- TV::ask_on_cell_spu_image_load
- TV::cell_spu_image_ignore_regexp
- TV::cell_spu_images_stop_regexp

These variables are described in Chapter 4, "TotalView Variables", in the *TotalView Reference Guide*.

To create a breakpoint in SPU threads, select **Yes**. Then select an SPU thread. You can now navigate within this code and set breakpoints.

When you exit from your program or when you manually save breakpoints, TotalView writes breakpoints for each SPU thread share group. The next time your program loads an SPU executable into an SPU context, these breakpoints are read back in.

Within the CLI your focus determines if the breakpoint is shared on unshared. For example::

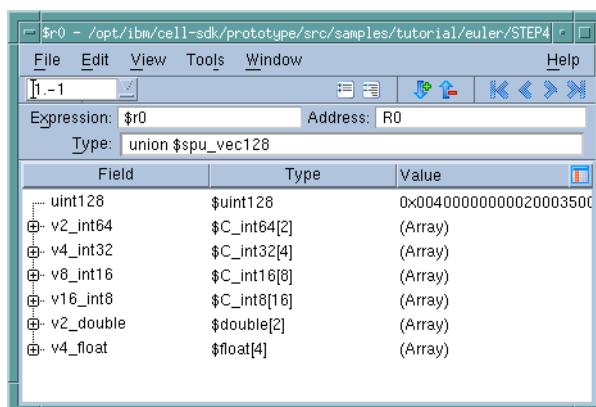
Focus	Type of Breakpoint
dfocus t 1.-1 dbbreakpoint main	Unshared breakpoint
dfocus p 1.-1 dbbreakpoint main	Unshared breakpoint
dfocus d 1.-1 dbbreakpoint main	Shared breakpoint
dfocus g 1.-1 dbbreakpoint main	Shared breakpoint

Registers, Unions, and Casting

SPU registers are 128-bits wide. In most cases, you'll be loading data into the registers in different ways. For example, you might be using the SPU as a vector processor and be loading 4 32-bit values into a register.

TotalView defines a union that helps you see this data in seven different ways, as is shown in the following figure.

Figure 103: Register Union



This picture shows how TotalView displays register R0. TotalView defines the data type of this register as union **\$spu_vec128**. If you dive (double-click) on a member in the union, TotalView shows the contents of the register as if they were defined in this way. For example, diving in **v4_float** tells TotalView to display this information as an array of four floating point numbers.

After diving on a union member, the displayed values can be edited and stored back into memory.

If this union doesn't describe your data, you can tell the TotalView to display it in the way you want by altering the information in this window's **Type** field. However, casting the value into a data type smaller than 128-bits pulls the value from the preferred slot of the register that is appropriate for the type's size.

Debugging Cray XT Applications

The Cray XT Series is supported by the TotalView x86_64 distribution. This section describes running applications on Cray XT Catamount. You should be familiar with this information when running applications on a Cray XT CNL. The primary difference between the two, and it is a big difference, is that Cray XT CNL uses `aprun` to launch programs rather than `yod`.

Cray XT Catamount

On the Cray XT Catamount, all jobs running on compute nodes are started with the `yod` starter program. These jobs do not have to be MPI jobs. Debugging a program started with `yod` is similar to debugging any program using a starter program. In general, you would type:

```
totalview totalview_args yod -a yod_args
```

For example:

```
totalview yod -a -np 4 ./my_prog
```

Here are some things you should know:

- `tvdsrv_rs` processes are started for your compute nodes. (This is a process started by TotalView on a remote node that communicates back with TotalView. For more information on this server, see Chapter 4.) `yod` will then pass information to TotalView, which will then start the servers. If this does not occur, consult your `yod` documentation.
- There may be more than one `tvdsrv_rs` process. TotalView will create one `tvdsrv_rs` process for each `RS_DBG_CLIENTS_PER_SERVER` or 64 compute nodes.
- To attach to a running program, attach to the instance of `yod` that is controlling it using normal TotalView mechanisms. TotalView will automatically attach to all compute node tasks that are part of the job.
- TotalView cannot know how many compute nodes are available, so each server assumes that it will be serving 64 compute nodes, and asks for a 64-node license. You can override this default by using the `-nodes_allowed tvdsrv` command-line option.

If you wish to use a small license (that is, a license for less than 64 processors), you must use the `-nodes_allowed tvdsrv` command-line option. The argument to this option specifies how many nodes the server supports and how many licenses it needs. Because this is a `tvdsrv_rs` command-line option, you must add it into the server launch string.

You can also use the `-nodes_allowed` server launch string option along with the `RS_DBG_CLIENTS_PER_SERVER` environment variable to increase the number of compute nodes each server will serve (and the number of Cray licences it asks for). However, we do not recommend that you set this server launch string option to a value greater than 256. (Note that you only need set this variable if `RS_DBG_CLIENTS_PER_SERVER` is greater than 64.)

For information on setting server launch strings, see "Setting Up and Starting the TotalView Server" on page 81.



While debugging, you must also have the FlexLM license server running. TotalView uses this server to verify that you are using licensed software. However, this server is not related to the servers that TotalView launches when you are debugging your program.

Configuring TotalView

When configuring your Cray XT system for use with the TotalView, you must:

- Mount user home directories on all service nodes that you will use while debugging.
- (Optional) Enable passwordless **ssh** on all service nodes that you will use while debugging. In most cases, your system administrator will have enabled your system so that it uses **ssh** instead of **rsh**. If passwordless **ssh** is not enabled, you will be asked to enter a password each time a process is launched on a node.
- (Optional) Automatically set **RS_DBG_CLIENTS_PER_SERVER** and **-nodes_allowed**.

On Cray XT systems setting a **-nodes_allowed** command-line option to 64 will not work. Instead, you should configure TotalView to use **RS_DBG_CLIENTS_PER_SERVER** and **-nodes_allowed** to make best use of the cluster and TotalView licenses.

TotalView administrators can set installation preferences by editing (or creating) the **\$TVROOT/linux-x86-64/lib/.tvdr**c file. Here are two simple scenarios

- If you only have one TotalView license and that license is for less than 64 processors then a server launch string like this would be best:

```
dset -set_as_default TV::server_launch_string \
    { %C %R -n "%B/tvdsrv%K -working_directory %D \
    -callback %L \
    -nodes_allowed maximum_processor_license \
    -set_pw %P -verbosity %V %F" }
```

where *maximum_processor_license* is the processor count for your TotalView license.

- If you will be running TotalView on a cluster where the ratio of service nodes to compute nodes is less than 1:64, you should use a server launch string. For example:

```
dset -set_as_default TV::server_launch_string \
    { %C %R -n "%B/tvdsrv%K -working_directory %D \
    -callback %L \
    -nodes_allowed ${RS_DBG_CLIENTS_PER_SERVER-64} \
    -set_pw %P -verbosity %V %F" }
```

You will need to set the **RS_DBG_CLIENTS_PER_SERVER** environment variable before submitting all jobs where a service node-to-compute node ratio of 1:64 is not possible. You should also set

RS_DBG_CLIENTS_PER_SERVER to the number of compute nodes served by each service node. For example, if you have a service-to-node ratio of 1:128, set this variable to 128.



The TV::server_launch_string variable is used for both Cray XT3 and Linux x86-64. This means that if you will also be using this TotalView installation on other linux-x86-64 machines, you should not set the TV::server_launch_string variable in your global .tvdrc.

Using TotalView

As part of launching an application on a compute node, TotalView will launch a server program on your login node using ssh. As with any **ssh** session, authentication is required. We recommend that you enable **ssh** without a pass phrase.

TotalView is typically run interactively. If your site has not designated any compute nodes for interactive processing, use the **PBS Pro qsub -I** interactive mode. This mode is described in the *Cray XT3 Programming Environment User's Guide*.

If TotalView is installed on your system, use the following command to load it into your user environment:

```
module load xt-totalview
```

You can now use the following command to start the CLI:

```
totalviewcli yod [-a argument_list] application_name
```

Here's the command that starts the GUI:

```
totalviewcli yod [-a argument_list] application_name
```

The following example shows how you can debug a program named **a.out**:

```
% qsub -I -l size=4
qsub: waiting for job 14448.nid00003 to start
qsub: job 14448.nid00003 ready

DISPLAY is user1:0.0
Linux perch 2.4.21-0-sles9-ss-lustre #2 Fri Apr 29
17:14:15 PDT 2005 x86_64 x86_64 x86_64 GNU/Linux
/ufs/home/users/user1

% module load xt-totalview
% cd working_directory
% totalview yod -a -sz 4 a.out
```

Cray XT CNL

Cray XT CNL applications are similar to those on Cray XT Catamount. The primary difference is that CNL applications are launched using **aprun** rather than **yod**.

Most, perhaps all, programs are launched using a batch queueing system such as PBS, Moab, and so on. While this is independent from TotalView, you will need to do queue up for an interactive session. For example:

```
qsub -I -sz=size
```

Here is an example of how you would start a Cray XT CNL debugging session:

```
totalview aprun -a -n4 a.out
```

TotalView is not able to stop your program before it calls **MPI_Init()**. While this is typically at the beginning of **main()**, the actual location depends on how you've written the program. This means that if you set a breakpoint before the **MPI_Init()** call, TotalView ignores it because the statement upon which you set the breakpoint will have already executed.

Debugging SiCortex Applications

TotalView runs as a cross-debugger within the SiCortex-MIPS Linux environment. The SiCortex version of TotalView is a 64-bit application and must run on a x86-64 system running a 64-bit kernel. All debugging must use the remote features of TotalView.

Features that are not implemented in this version are:

- Memory debugging
- OpenMP
- SHMEM
- PVM
- Debugging 32-bit executables
- Watchpoints
- Compiled EVAL points

Installation Notes

The default location into which the TotalView installer places TotalView binaries is

/opt/toolworks. In addition, you must use this same path when you install it into the file system that will be mounted on the MIPS nodes.

For more information on installing TotalView, consult the "TotalView Installation Guide," which is located at <http://www.totalviewtech.com/Documentation/>.

Using TotalView on SiCortex

TotalView must be able to execute a command on the target system from the development host. By default, this version uses the **ssh -x** command. However, we suggest that set up **ssh** so that lets you execute commands without requiring a password.

Your program's executable file must be visible from both the development host and the target system. Do this by placing the executables in a directory that is visible on both machines through the same path. Having the

executable visible in separate directories that are accessed through the same path on both machines will also work. However, if you're using the cross development compilers, you'll probably want the same copy of the executable visible from both machines anyway.

The SiCortex version of the TotalView does not use the same naming conventions as is used on other versions. Instead, the command has an **sc** prefix. For example, use the **sctv8** or **sctotalview** commands instead of **tv8** or **totalview8** to invoke the GUI version. Use **sctv8cli** or **sctotalviewcli** instead of **tv8cli** or **totalviewcli** to invoke the CLI.

The commands are located within the following directory:

install_path/toolworks/totalview.version/bin

If this directory is in your PATH, here's how to start the TotalView GUI:

sctv8 -r remote_host executable_path -a arguments

When you use the **--remote** command-line option and the program's name is **srun**, TotalView makes a local copy of **srun** using **scp**. It also substitutes the name of the local copy for the program name. This local copy is deleted when TotalView exits.

MPI Debugging

TotalView normally requires that the executable be visible through the same path on the host machine and target machines. In addition, TotalView must debug the MIPS version of **srun**. TotalView will not let you debug programs invoked using the x86-64 version of **srun**. We recommend that you copy the MIPS version of **srun** into the directory with your executables, then invoke TotalView as follows:

sctv8 -r SiCortex_node ./srun -a srun_arguments

You can also run MPI programs by entering information into the **File > New Program** dialog box. Within the **Parallel** tab, select **SiCortex** within the pull-down list. (This is the preferred way to start MPI programs from within TotalView.) When using this dialog box, you must be sure that TotalView finds the MIPS version of **srun**.

Debugging Global Arrays Applications

The following paragraphs, which are copies from the Global Arrays home site (<http://www.emsl.pnl.gov/docs/global/ga.html>), describe the global arrays environment:

The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures (called "global arrays"). From the user perspective, a global array can be used as if

it was stored in shared memory. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.

GA divides logically shared data structures into "local" and "remote" portions. It recognizes variable data transfer costs required to access the data depending on the proximity attributes. A local portion of the shared memory is assumed to be faster to access and the remainder (remote portion) is considered slower to access. These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located. In addition, any processes can access a local portion of the shared data directly/in-place like any other data in process local memory. Access to other portions of the shared data must be done through the GA library calls.

GA was designed to complement rather than substitute for the message-passing model, and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.

TotalView supports Global Arrays on the Intel IA-64 platform. You debug a Global Arrays program in basically the same way that you debug any other multi-process program. The one difference is that you will use the **Tools > Global Arrays** command to display information about your global data.

The global arrays environment has a few unique attributes. Using TotalView, you can:

- Display a list of a program's global arrays.
- Dive from this list of global variables to see the contents of a global array in C or Fortran format.
- Cast the data so that TotalView interprets data as a global array handle. This means that TotalView displays the information as a global array. Specifically, casting to **\$GA** forces the Fortran interpretation; casting to **\$ga** forces the C interpretation; and casting to **\$Ga** tells TotalView to use the language in the current context.

Within a Variable Window, the commands that operate on a local array, such as slicing, filtering, obtaining statistics, and visualization, also operate on global arrays.

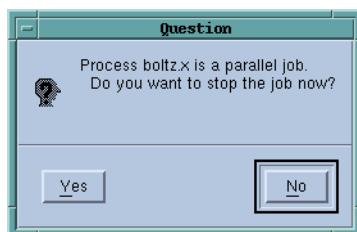
The command you use to start TotalView depends on your operating system. For example, the following command starts TotalView on a program that is invoked using **prun** and which uses three processes:

```
totalview prun -a -N 3 boltz.x
```

Before your program starts parallel execution, TotalView asks if you want to stop the job. (See Figure 104 on page 150.)

Choose **Yes** if you want to set breakpoints or inspect the program before it begins execution.

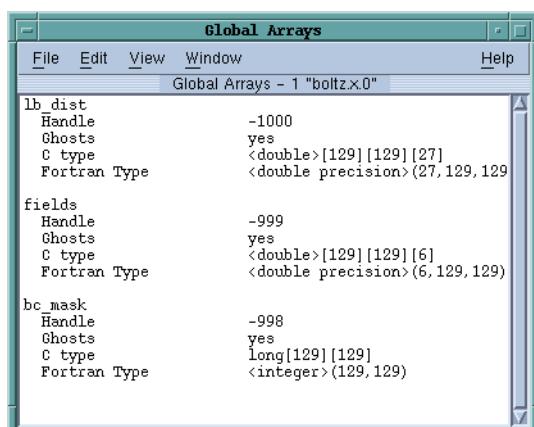
Figure 104: Question Window for Global Arrays Program



After your program hits a breakpoint, use the **Tools > Global Arrays** command to begin inspecting your program's global arrays. TotalView displays the following window.

CLI: dga

Figure 105: Tools > Global Arrays Window



The arrays named in this window are displayed using their C and Fortran type names. Diving on the line that contains the type definition tells TotalView to display Variable Windows that contains information about that array.

After TotalView displays this information, you can use other standard commands and operations on the array. For example, you can use the slice and filter operations and the commands that visualize, obtain statistics, and show the nodes from which the data was obtained.

If you inadvertently dive on a global array variable from the Process Window, TotalView does not know that it is a component of a global array. If, however, you do dive on the variable, you can cast the variable into a global array using either **\$ga** for a C Language cast or **\$GA** for a Fortran cast.

Debugging PVM (Parallel Virtual Machine) and DPVM Applications

You can debug applications that use the Parallel Virtual Machine (PVM) library or the HP Alpha Tru64 UNIX Parallel Virtual Machine (DPVM) library with TotalView on some platforms. TotalView supports ORNL PVM Version 3.4.4 on all platforms and DPVM Version 1.9 or later on the HP Alpha platform.



See the TotalView Platforms document for the most up-to-date information regarding your PVM or DPVM software.

For tips on debugging parallel applications, see "Debugging Parallel Applications Tips" on page 124.

Topics in this section are:

- "Supporting Multiple Sessions" on page 151
- "Setting Up ORNL PVM Debugging" on page 152
- "Starting an ORNL PVM Session" on page 152
- "Starting a DPVM Session" on page 153
- "Automatically Acquiring PVM/DPVM Processes" on page 154
- "Attaching to PVM/DPVM Tasks" on page 155

Supporting Multiple Sessions

When you debug a PVM or DPVM application, TotalView becomes a PVM tasker. This lets it establish a debugging context for your session. You can do the following:

- You can run TotalView PVM or DPVM debugging session for a user and for an architecture; that is, different users can't interfere with each other on the same computer or same computer architecture.

One user can start TotalView to debug the same PVM or DPVM application on different computer architectures. However, a single user can't have multiple instances of TotalView debugging the same PVM or DPVM session on a single computer architecture.

For example, if you start a PVM session on Sun 5 and HP Alpha computers. You must start two TotalView sessions: one on the Sun 5 computer to debug the Sun 5 portion of the PVM session, and one on the HP Alpha computer to debug the HP Alpha portion of the PVM session. These two TotalView sessions are separate and don't interfere with one another.

- In one TotalView session, you can run either a PVM application or a DPVM application, but not both. However, if you run TotalView on an HP Alpha, you can have two TotalView sessions: one debugging PVM and one debugging DPVM.

Setting Up ORNL PVM Debugging

To enable PVM, create a symbolic link from the PVM **bin** directory (which is usually `$HOME/pvm3/bin/$PVM_ARCH/tvdsrv`) to the TotalView Server (**tvdsrv**). With this link in place, TotalView invokes **pvm_spawn()** to spawn the **tvdsrv** tasks.

For example, if **tvdsrv** is installed in the `/opt/totalview/bin` directory, enter the following command:

```
ln -s /opt/totalview/bin/tvdsrv \
      $HOME/pvm3/bin/$PVM_ARCH/tvdsrv
```

If the symbolic link doesn't exist, TotalView can't spawn **tvdsrv**. If TotalView can't spawn **tvdsrv**, it displays the following error:

```
Error spawning TotalView Debugger Server: No such file
```

Starting an ORNL PVM Session

Start the ORNL PVM daemon process before you start TotalView. See the ORNL PVM documentation for information about the PVM daemon process and console program. The procedure for starting an ORNL PVM application is as follows:

- 1 Use the **pvm** command to start a PVM console session—this command starts the PVM daemon.

If PVM isn't running when you start TotalView (with PVM support enabled), TotalView exits with the following message:

```
Fatal error: Error enrolling as PVM task:
pvm error
```

- 2 If your application uses groups, start the **pvmgs** process before starting TotalView.

PVM groups are unrelated to TotalView process groups. For information about TotalView process groups, see "Examining Groups" on page 235.

- 3 You can use the **-pvm** command-line option to the **totalview** command. As an alternative, you can set the **TV::pvm** variable in a startup file.

The command-line options override the CLI variable. For more information, see "TotalView Command Syntax" in the *TotalView Reference Guide*.

- 4 Set the TotalView directory search path to include the PVM directories.

This directory list must include those needed to find both executable and source files. The directories you use can vary, but should always contain the current directory and your home directory.

You can set the directory search path using either the **EXECUTABLE_PATH** variable or the **File > Search Path** command. See "Setting Search Paths" on page 71 for more information.

For example, to debug the PVM examples, you can place the following directories in your search path:

```
.
$HOME
$PVM_ROOT/xep
$PVM_ROOT/xep/$PVM_ARCH
$PVM_ROOT/src
```

```
$PVM_ROOT/src/$PVM_ARCH
$PVM_ROOT/bin/$PVM_ARCH
$PVM_ROOT/examples
$PVM_ROOT/examples/$PVM_ARCH
$PVM_ROOT/gexamples
$PVM_ROOT/gexamples/$PVM_ARCH
```

- 5** Verify that the action taken by TotalView for the **SIGTERM** signal is appropriate. (You can examine the current action by using the Process Window **File > Signals** command. See "Handling Signals" on page 69 for more information.)

PVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the process is not terminated. If you want the PVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

TotalView will automatically acquire your application's PVM processes. For more information, see "Automatically Acquiring PVM/DPVM Processes" on page 154.

Starting a DPVM Session

Starting a DPVM debugging session is similar to starting any other TotalView debugging session. The only additional requirement is that you must start the DPVM daemon before you start TotalView. See the DPVM documentation for information about the DPVM daemon and its console program. The procedure for starting an DPVM application is as follows

- 1** Use the **dpvm** command to start a DPVM console session; starting the session also starts the DPVM daemon.
If DPVM isn't running when you start TotalView (with DPVM support enabled), TotalView displays the following error message before it exits:
Fatal error: Error enrolling as DPVM task: dpvm error
- 2** Enable DPVM support either by using the **TV::dpvm** CLI variable or by using the **-dpvm** command-line option to the **totalview** command.
The command-line options override the **TV:dpvm** command variable. For more information on the **totalview** command, see "TotalView Command Syntax" in the *TotalView Reference Guide*.
- 3** Verify that the default action taken by TotalView for the **SIGTERM** signal is appropriate. (You can examine the default actions with the Process Window **File > Signals** command in TotalView. See "Handling Signals" on page 69 for more information.)

DPVM uses the **SIGTERM** signal to terminate processes. Because TotalView stops a process when the process receives a **SIGTERM**, the process is not terminated. If you want the DPVM process to terminate, set the action for the **SIGTERM** signal to **Resend**.

If you enable PVM support using the **TV::pvm** variable and you need to use DPVM, you must use both **-no_pvm** and **-dpvm** command-line options when you start TotalView. Similarly, when enabling DPVM support us the **TV::dpvm** variable, you must use the **-no_dpvm** and **-pvm** command-line options.



You cannot use CLI variables to start both PVM and DPVM.

Automatically Acquiring PVM/DPVM Processes

When you start TotalView as part of a PVM or DPVM debugging session, it takes the following actions:

- TotalView makes sure that no other PVM or DPVM taskers are running. If TotalView finds a tasker on a host that it is debugging, it displays the following message and then exits:

**Fatal error: A PVM tasker is already running
on host '*host*'**

- TotalView finds all the hosts in the PVM or DPVM configuration. Using the **pvm_spawn()** call, TotalView starts a TotalView Server (**tvdsrv**) on each remote host that has the same architecture type as the host TotalView is running on. It tells you it has started a debugger server by displaying the following message:

**Spawning TotalView Debugger Server onto PVM
host '*host*'**

If you add a host with a compatible computer architecture to your PVM or DPVM debugging session after you start TotalView, it automatically starts a debugger server on that host.

After all debugger servers are running, TotalView intercepts every PVM or DPVM task created with the **pvm_spawn()** call on hosts that are part of the debugging session. If a PVM or DPVM task is created on a host with a different computer architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, the following actions occur:

- 1 TotalView reads the symbol table of the spawned executable.
- 2 If a saved breakpoint file for the executable exists and you have enabled automatic loading of breakpoints, TotalView loads breakpoints for the process.
- 3 TotalView asks if you want to stop the process before it enters the **main()** routine.

If you answer **Yes**, TotalView stops the process before it enters **main()** (that is, before it executes any user code). This allows you to set breakpoints in the spawned process before any user code executes. On most computers, TotalView stops a process in the **start()** routine of the **crt0.o** module if it is statically linked. If the process is dynamically linked, TotalView stops it just after it finishes running the dynamic linker. Because the Process Window displays assembler instructions, you need to use the **View > Lookup Function** command to display the source code for **main()**.

CLI: dlist *function-name*

For more information on this command, see "Finding the Source Code for Functions" on page 225.

Attaching to PVM/DPVM Tasks

You can attach to a PVM or DPVM task if the following are true:

- The computer architecture on which the task is running is the same as the computer architecture upon which TotalView is running.
- The task must be created. (This is indicated when flag 4 is set in the PVM Tasks and Configuration Window.)
- The task must not be a PVM tasker. If flag 400 is clear in the PVM Tasks and Configuration Window, the process is a tasker.
- The executable name must be known. If the executable name is listed as a dash (-), TotalView cannot determine the name of the executable. (This can occur if a task was not created with the `pvm_spawn()` call.)

To attach to a PVM or DPVM task:

- 1 Select the **Tools > PVM Tasks** command from the Root Window.
TotalView responds the PVM Tasks Window. (See Figure 106.)

Figure 106: PVM Tasks and Configuration Window

The screenshot shows the 'PVM Tasks' window with the following data:

PVM Tasks And Configuration					
HOST	TID	PTID	PID	FLAG	EXECUTABLE
rsmpl	40008	0	31660	404	-
rsmpl	40009	0	25790	4	-
rsmpl	4000a	40009	37828	4	mtile
albacore	80002	40009	641	6	mtile
albacore	80001	40003	2602	6	mtile

HOST	DTID	ARCH	SPEED
albacore	80000	SUN4SOL2	1000
rsmpl	40000	AIX46K	1000

Annotations point to specific columns and rows:

- Tasks**: Points to the first two columns of the main table.
- Hosts**: Points to the first column of the bottom table.
- Daemon TID**: Points to the DTID column of the bottom table.
- Machine Architecture**: Points to the ARCH column of the bottom table.
- Task ID (TID)**: Points to the TID column of the main table.
- Parent TID**: Points to the PTID column of the main table.
- UNIX Process ID (PID)**: Points to the PID column of the main table.

This window displays current information about PVM tasks and hosts—TotalView automatically updates this information as it receives events from PVM.

Since PVM doesn't always generate an event that allows TotalView to update this window, use the **Window > Update** command to ensure that you are seeing the most current information.

For example, you can attach to the tasks named **xep** and **mtile** in the preceding figure because flag 4 is set. In contrast, you cannot attach to the - (dash) executables and **tvdsrv**, because flag 400 is set.

- 2 Dive on a task entry that meets the criteria for attaching to tasks.
TotalView attaches to the task.

- 3 If the task to which you attached has related tasks that can be debugged, TotalView asks if you want to attach to these related tasks. If you answer **Yes**, TotalView attaches to them. If you answer **No**, it only attaches to the task you dove on.

After attaching to a task, TotalView looks for attached tasks that are related to this task; if there are related tasks, TotalView places them in the same control group. If TotalView is already attached to a task you dove on, it simply opens and raises the Process Window for the task.

About Reserved Message Tags

TotalView uses PVM message tags in the range 0xDEB0 through 0xDEBF to communicate with PVM daemons and the TotalView Server. Avoid sending messages that use these reserved tags.

Cleaning Up Processes

The **pvmgs** process registers its task ID in the PVM database. If the **pvmgs** process terminates, the **pvm_joingroup()** routine hangs because PVM won't clean up the database. If this happens, you must manually terminate the program and then restart the PVM daemon.

TotalView attempts to clean up the **tvdsvr** processes that also act as taskers. If some of these processes do not terminate, you must manually terminate them.

Debugging Shared Memory (SHMEM) Code

TotalView supports programs using the distributed memory access Shared Memory (SHMEM) library on Quadrics RMS systems and SGI Altix and IRIX systems. The SHMEM library allows processes to read and write data stored in the memory of other processes. This library also provided collective operations.

Debugging a SHMEM RMS or SGI Altix program is no different than debugging any other program that uses a starter program. For example:

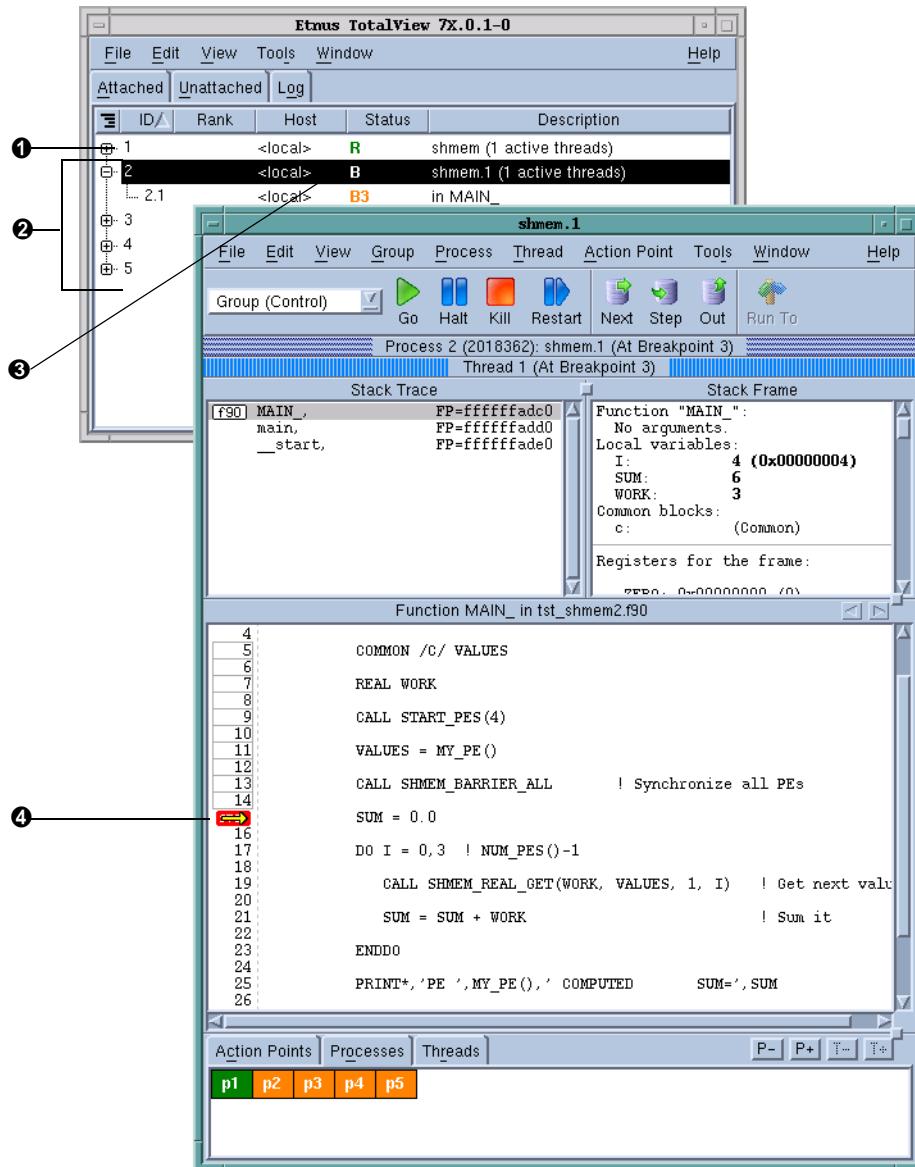
```
totalview srun -a my_program
```

Debugging a SHMEM program running on SGI IRIX requires the following:

- 1 Link the SHMEM program with the dbfork library. See "*Linking with the dbfork Library*" in the "*Compilers and Platforms*" chapter of the *TotalView Reference Guide*.
- 2 Start TotalView on your program. (See Chapter 4, "*Setting Up a Debugging Session*," on page 51.)

- 3 Set at least one breakpoint after the call to the `start_pes()` SHMEm routine. (See Figure 107.)

Figure 107: SHMEm Sample Session



- ① SHMEm starter process
- ② SHMEm worker processes
- ③ Select a worker process in the Root Window
- ④ Set a breakpoint after the call to `start_pes()`



You cannot single-step over the call to `start_pes()`. Also, the call to `start_pes()` creates new worker processes that return from the `start_pes()` call and execute the remainder of your program. The original process never returns from `start_pes()`, but instead stays in that routine, waiting for the worker processes it created to terminate.

Debugging UPC Programs

TotalView lets you debug UPC programs that were compiled using the HP Compaq Alpha UPC 2.0 and the Intrepid (SGI gcc UPC) compilers. This section only discusses the UPC-specific features of TotalView. It is not an introduction to the UPC Language. For an introduction to the UPC language, go to <http://www.gwu.edu/~upc>.



When debugging UPC code, TotalView requires help from a UPC assistant library that your compiler vendor provides. You need to include the location of this library in your LD_LIBRARY_PATH environment variable. TotalView Technologies also provides assistants that you can use. You can find these assistants at <http://www.totalview-tech.com/Products/TotalView/developers/index.html>.

Topics in this section are:

- "Invoking TotalView" on page 158
- "Viewing Shared Objects" on page 158
- "Displaying Pointer to Shared Variables" on page 160

Invoking TotalView

The way in which you invoke TotalView on a UPC program is straight-forward. However, this procedure depends on the computer upon which the program is executing:

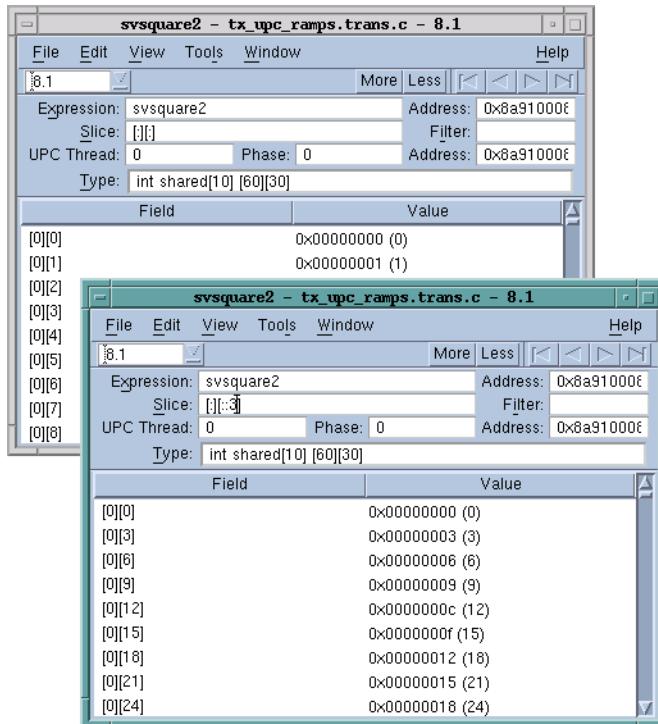
- When running on an SGI system using the gcc UPC compiler, invoke TotalView on your UPC program in the same way as you would invoke it on most other programs; for example:
`totalview prog_upc -a prog_upc_args`
- When running on Linux and HP Compaq SC computers, debug UPC code in the same way that you would debug other kinds of parallel code; for example:
`totalview prun -a -n node_count prog_upc prog_upc_args`

Viewing Shared Objects

TotalView displays UPC shared objects, and fetches data from the UPC thread with which it has an affinity. For example, TotalView always fetches shared scalar variables from thread 0.

The upper-left screen in Figure 108 on page 159 displays elements of a large shared array. You can manipulate and examine shared arrays the same as any other array. For example, you can slice, filter, obtain statistical information, and so on. (For more information on displaying array data, see Chapter 15, "Examining Arrays," on page 333.) The lower-right screen shows a slice of this array.

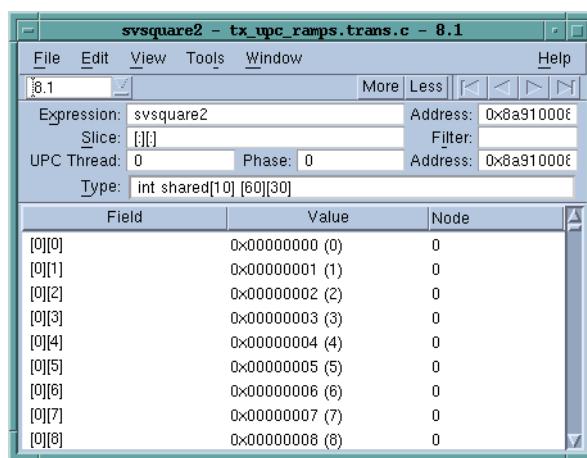
Figure 108: A Sliced UPC Array



In this figure, TotalView displays the value of a pointer-to-shared variable whose target is the array in the **Shared Address** area. As usual, the address in the process appears in the top left of the display.

Since the array is shared, it has an additional property: the element's affinity. You can display this information if you right-click your mouse on the header and tell TotalView to display Nodes.(See Figure 109.)

Figure 109: UPC Variable Window Showing Nodes

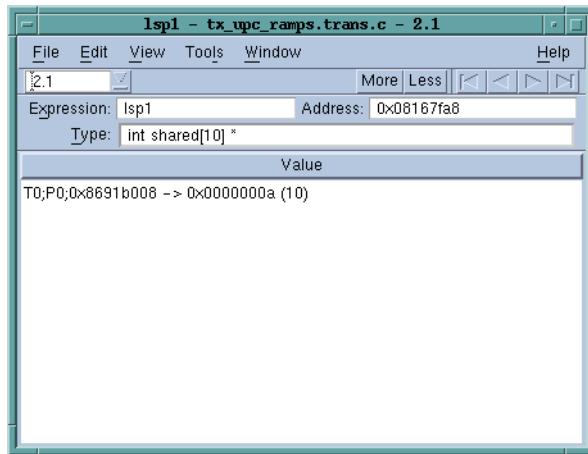


You can also use the **Tools > Visualize Distribution** command to visualize this array. For more information on visualization, see "Visualizing Array Data" on page 181.

Displaying Pointer to Shared Variables

TotalView understands pointer-to-shared data and displays the components of the data, as well as the target of the pointer to shared variables. For example, Figure 110 shows this data being displayed:

Figure 110: A Pointer to a Shared Variable



In this figure, notice the following:

- Because the **Type** field shows the full type name, TotalView is telling you that this is a pointer to a shared **int** with a block size of 10.
- In this figure, TotalView also displays the **upc_threadof ("T0")**, the **upc_phaseof ("P0")**, and the **upc_addrfield (0x0x10010ec4)** components of this variable.

In the same way that TotalView normally shows the target of a pointer variable, it also shows the target of a UPC pointer variable. When dereferencing a UPC pointer, TotalView fetches the target of the pointer from the UPC thread with which the pointer has affinity.

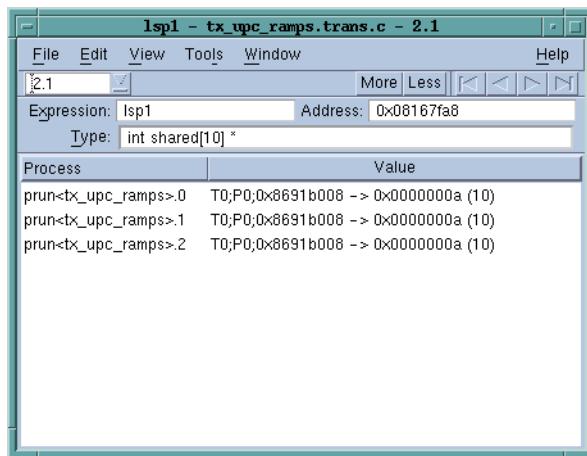
You can update the pointer by selecting the pointer value and editing the thread, phase, or address values. If the phase is corrupt, you'll see something like the following in the **Value** area:

```
T0;P6;0x3ffc0003b00 <Bad phase [max 4]> ->
0xc0003c80 (-1073726336)
```

In this example, the pointer is invalid because the phase is outside the legal range. TotalView displays a similar message if the thread is invalid.

Since the pointer itself is not shared, you can use the **TView > Show Across** commands to display the value from each of the UPC threads. (See Figure 111 on page 161.)

Figure 111: Pointer to a Shared Variable



Part III: Using the GUI



The two chapters in this part of the users guide contains information about using the TotalView GUI.

Chapter 8: Using TotalView Windows

Describes using the mouse and the fundamental TotalView windows.

Chapter 9: Visualizing Programs and Data

Some TotalView commands and tools are only useful if you're using the GUI. Here you will find information on the Call Graph and Visualizer.

Using TotalView Windows



c
h
a
p
t
e
r

8

This chapter introduces you to the most important TotalView windows and the mechanics of using the GUI. The topics in this chapter are as follows:

- “*Using Mouse Buttons*” on page 165
- “*Using the Root Window*” on page 166
- “*Using the Process Window*” on page 169
- “*Viewing the Assembler Version of Your Code*” on page 171
- “*Diving into Objects*” on page 173
- “*Resizing and Positioning Windows and Dialog Boxes*” on page 175
- “*Editing Text*” on page 176
- “*Saving the Contents of Windows*” on page 177

Using Mouse Buttons

TotalView uses the buttons on your three-button mouse as follows:

Button	Action	Purpose	How to Use It
Left	Select	Selects or edits object. Scrolls in windows and panes.	Move the cursor over the object and click the button.
Middle	Paste	Writes information previously copied or cut into the clipboard.	Move the cursor to where you will be inserting the information and click the button. Not all windows support pasting.

Button	Action	Purpose	How to Use It
	Dive	Displays more information or replaces window contents.	Move the cursor over an object, then click the middle-mouse button.
Right	Context menu	Displays a menu with commonly used commands.	Move the cursor over an object and click the button. Most windows and panes have context menus; dialog boxes do not have context menus.

In most cases, a single-click selects what's under the cursor and a double-click dives on the object. However, if the field is editable, TotalView goes into its edit mode, in which you can alter the selected item's value.

In some places such as the Stack Trace Pane, selecting a line tells TotalView to perform an action. In this pane, TotalView dives on the selected routine. (In this case, *diving* means that TotalView finds the selected routine and shows it in the Source Pane.)

In the line number area of the Source Pane, a left mouse click sets a breakpoint at that line. TotalView shows you that it has set a breakpoint by displaying a  icon instead of a line number.

Selecting the  icon a second time deletes the breakpoint. If you change any of the breakpoint's properties or if you've created an eval point (indicated by an  icon), selecting the icon disables it. For more information on breakpoints and eval points, see Chapter 16, "Setting Action Points," on page 349.

Using the Root Window

The Root Window appears when you start TotalView. If you type a program name immediately after the **totalview** command, TotalView also opens a Process Window that contains the program's source code. If you do not enter a program name when you start TotalView, TotalView also displays its **File > New Program** dialog box. Use this dialog box to enter a program's name and information needed to execute that program.

The Root Window displays a list of all the processes and threads being debugged. Initially—that is, before your program begins executing—the Root Window just contains the name of the program being debugged. As your program creates processes and threads, TotalView adds them to this list. Associated with each is a name, location (if a remote process), process ID, status, and a list of executing threads for each process. It also shows the thread ID, status, and the routine being executed in each thread.

Figure 112 shows the Root Window for an executing multi-threaded multi-process program.

Figure 112: Root Window

The screenshot shows the Etnus TotalView 7.1.0-0 interface with the title bar "Etnus TotalView 7.1.0-0". The menu bar includes File, Edit, View, Tools, Window, and Help. The main window displays a table with columns: ID, Rank, Host, Status, and Description. The table lists 15 entries, with row 13 expanded to show sub-entries 13.1 through 13.3. The "Host" column indicates the host for each process, and the "Status" column shows the status of each thread.

	ID	Rank	Host	Status	Description
⊕- 1	0	<local>	B	mismatchLinux.0 (1 active threads)	
⊕- 5		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux (5	
⊕- 6		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 7		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.2 (
⊕- 8		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 9		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 10		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 11		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.3 (
⊕- 12		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.2 (
⊕- 13	1	<local>	B	mismatchLinux.1 (1 active threads)	
	13.1	1	<local>	B4	in main
⊕- 14	2	<local>	B	mismatchLinux.2 (1 active threads)	
⊕- 15	3	<local>	B	mismatchLinux.3 (1 active threads)	

When debugging a remote process, TotalView displays the host name on which the process is running within the Process Window and in the Root Window. In Figure 113, the processes are running on **intrepid** and on **localhost**. This figure also describes the contents of the columns in this window.

Figure 113: Root Window
Showing Two Host
Computers

This screenshot is similar to Figure 112 but includes callout arrows pointing to specific columns and rows to explain their meaning. The columns are labeled as follows:

- Hierarchical/linear toggle
- Host name
- Rank Number (if MPI program)
- TotalView thread ID (PID.TID)
- Expand/collapse toggle
- Process status
- Action point ID number

The table structure is identical to Figure 112, showing a list of processes and threads across multiple hosts.

	ID	Rank	Host	Status	Description
⊕- 1	0	<local>	B	mismatchLinux.0 (1 active threads)	
⊕- 5		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux (5	
⊕- 6		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 7		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.2 (
⊕- 8		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 9		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 10		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.1 (
⊕- 11		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.3 (
⊕- 12		intrepid.ethnus.c	T	/home/barryk/tests/fork_loopLinux.2 (
⊕- 13	1	<local>	B	mismatchLinux.1 (1 active threads)	
	13.1	1	<local>	B4	in main
⊕- 14	2	<local>	B	mismatchLinux.2 (1 active threads)	
⊕- 15	3	<local>	B	mismatchLinux.3 (1 active threads)	

When you dive on a line in this window, TotalView displays the source for that process or thread in a Process Window.

Using the Root Window

TotalView can display process and thread data linearly and hierarchically. (See Figure 114.)

Figure 114: Two Views of the Root Window

The figure shows two windows of the Etnus TotalView 7.1.0-0 application side-by-side. Both windows have a title bar 'Etnus TotalView 7.1.0-0' and a menu bar with File, Edit, View, Tools, Window, and Help.

The left window displays data in a hierarchical view. The columns are ID / Rank, Host, Status, and Description. The data includes:

ID / Rank	Host	Status	Description
1	<local>	B	mismatchLinux.0 (1 active threads)
5	intrepid.etnus.cT		/home/barryk/tests/fork_looplLinux (
5.1	intrepid.etnus.cT		in __select
5.2	intrepid.etnus.cT		in __select
5.3	intrepid.etnus.cT		in __select
5.4	intrepid.etnus.cT		in __select
6	intrepid.etnus.cT		/home/barryk/tests/fork_looplLinux.1
7	intrepid.etnus.cT		/home/barryk/tests/fork_looplLinux.2
8	intrepid.etnus.cT		/home/barryk/tests/fork_looplLinux

The right window displays data in a linear view. The columns are ID / Rank, Host, Status, and Description. The data includes:

ID / Rank	Host	Status	Description
1.1	<local>	B4	in main
5.1	intrepid.etnus.cT		in __select
5.2	intrepid.etnus.cT		in __select
5.3	intrepid.etnus.cT		in __select
5.4	intrepid.etnus.cT		in __select
6.1	intrepid.etnus.cT		in __select
6.2	intrepid.etnus.cT		in __select
6.3	intrepid.etnus.cT		in __select
6.4	intrepid.etnus.cT		in __select
7.1	intrepid.etnus.cT		in __select
7.2	intrepid.etnus.cT		in __select
7.3	intrepid.etnus.cT		in __select
7.4	intrepid.etnus.cT		in __select
8.1	intrepid.etnus.cT		in __select

Annotations in the figure point to the hierarchy toggle button (a small icon with a plus sign) in the top-left corner of both windows, and to specific rows in the hierarchical view table to indicate the difference between the two modes.

Selecting the hierarchy toggle button () changes the view from linear to hierarchical view. When data is being displayed hierarchically, you can perform the following additional operations:

- Selectively display information using the + or – indicators. The **View > Expand All** and **View > Compress All** commands let you open and close all of this window's hierarchies.
- Sort a column by clicking on a column header.

The hierarchical view lets you group similar information. For example, if you sort the information by clicking the **Status** header, TotalView groups all attached processes by their status. This lets you see, for example, which threads are held, at a breakpoint, and so on. When information is aggregated (that is grouped) like this, you can also display information selectively. (See Figure 115 on page 169.)

TotalView displays all of your program's processes and threads. You can change this using the following commands:

- **View > Display Managers:** When multi-process and multi-threaded programs run, the operating system often creates threads whose sole function is to manage your program's processes. In addition, many HPC programs use a starter process. Usually, you are not interested in these threads and processes. This command lets you remove them from the

Figure 115: Sorted and Aggregated Root Window

ID	Rank	Host	Status	Description
1.1.0	<local>	B4	in main	Breakpoint (4 active threads)
13.1.1	<local>	B4	in main	
14.1.2	<local>	B4	in main	
15.1.3	<local>	B4	in main	
T			Stopped (32 active threads)	
5.1	intrepid.etnus.cT		in __select	
5.2	intrepid.etnus.cT		in __select	
5.3	intrepid.etnus.cT		in __select	
5.4	intrepid.etnus.cT		in __select	
6.1	intrepid.etnus.cT		in __select	
6.2	intrepid.etnus.cT		in __select	
6.3	intrepid.etnus.cT		in __select	
6.4	intrepid.etnus.cT		in __select	

display. In most cases, this is what you want to do as you will not be debugging these processes or threads.



If you are running TotalView Team, a manager process uses a token in exactly the same way a user process. For example, if you are running a 32 process MPI job that is invoked using `mpirun`, you will need 33 tokens.

- **View > Display Exited Threads:** Tracking when processes stop and start executing in a multi-process, multi-threaded environment can be challenging. Selecting this command tells TotalView to display threads after they've exited. While this clutters your display with information about threads that are no longer executing, it can sometimes be helpful in trying to track down some problems. You probably don't want to see these threads in the listing. However, you can tell TotalView to show them at anytime. That is, TotalView remembers them so that toggling this command shows this information.

Using the Process Window

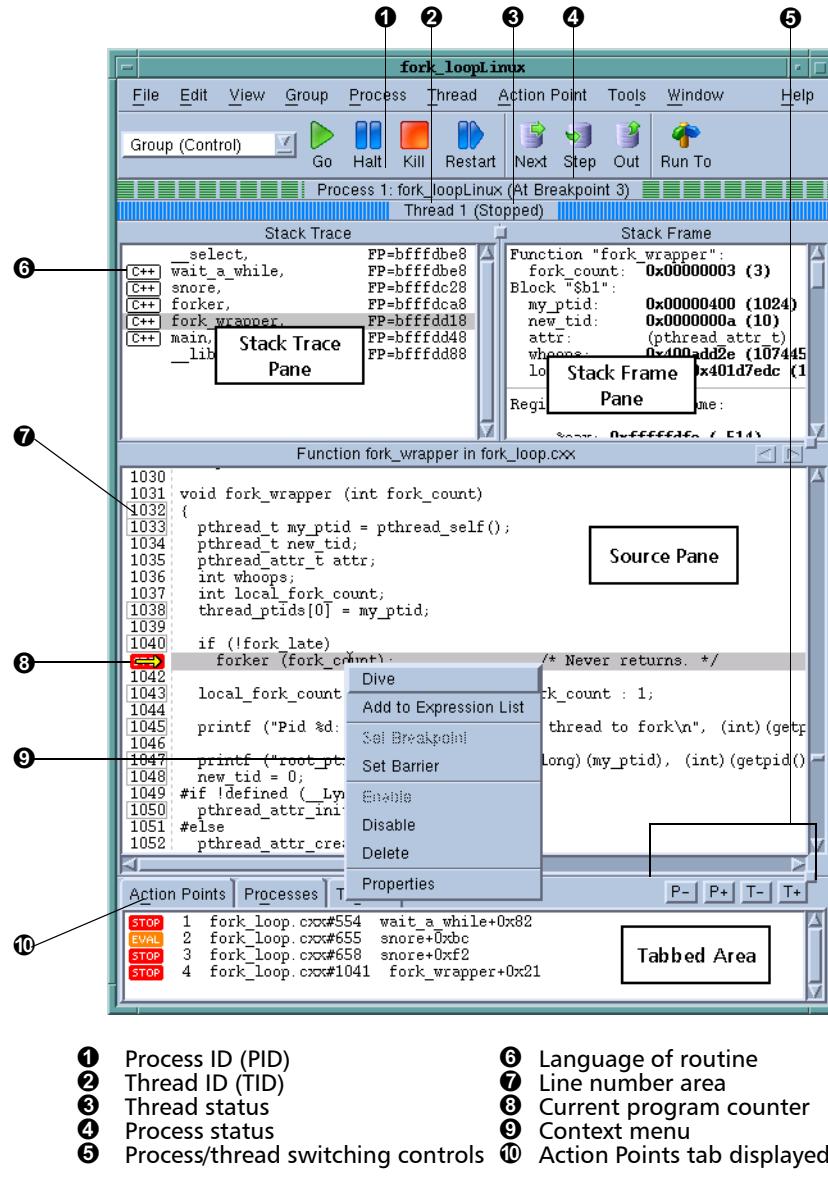
The *Process Window* (see Figure 116 on page 170) contains the code for the process or thread that you're debugging, as well as other related information. This window contains *panes* of information. The large scrolling list in the middle of the Process Window is the Source Pane. (The contents of these panes are discussed later in this section.)

As you examine the Process Window, notice the following:

- The thread ID shown in the Root Window and in the process's Threads Tab with the Tabs Pane is the logical thread ID (TID) assigned by TotalView and the system-assigned thread ID (SYSTID). On systems such as HP Alpha Tru64 UNIX, where the TID and SYSTID values are the same, TotalView displays only the TID value.

Using the Process Window

Figure 116: A Process Window



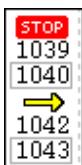
In other windows, TotalView uses the *pid.tid* value to identify a process's threads.

The Threads Tab shows the threads that currently exist in a process. When you select a different thread in this list, TotalView updates the Stack Trace, Stack Frame, and Source Panes to show the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window that displays information for that thread.

- The Stack Trace Pane shows the call stack of routines that the selected thread is executing. You can move up and down the call stack by clicking on the routine's name (stack frame). When you select a different stack frame, TotalView updates the Stack Frame and Source Panes to show the information about the routine you just selected.
- The Stack Frame Pane displays all of a routine's parameters, its local variables, and the registers for the selected stack frame.

- The information displayed in the Stack Trace and Stack Frame Panes reflects the state of a process when it was last stopped. Consequently, the information that they display is not up-to-date while a thread is running.
- The left margin of the *Source Pane* displays line numbers and action point icons. You can place a breakpoint at any line whose line number is contained within a box. The box indicates that executable code was created by the source code.
When you place a breakpoint on a line, TotalView places a **STOP** icon over the line number. An arrow over the line number shows the current location of the program counter (PC) in the selected stack frame. (See Figure 117.)

Figure 117: Line Numbers with Stop Icon and PC Arrow



Each thread has its own unique program counter (PC). When you stop a multi-process or multi-threaded program, the routine displayed in the Stack Trace Pane for a thread depends on the thread's PC. Because threads execute asynchronously, threads are stopped at different places. (When your thread hits a breakpoint, the default is to stop all the other threads in the process as well.)

- The tabbed area at the bottom contains a set of tabs whose information you can hide or display as you need it. In addition, the P+, P-, T+, and T- buttons within this area allow you to change the Process Window's context by moving to another process or thread.

The *Action Points Tab* with the Tabs Pane shows the list of breakpoints, eval points, and watchpoints for the process. The *Processes/Ranks tab* displays a grid of all of your program's processes. The grid's element's shows process status and indicates the selected group. Selecting a process switches the context to the first thread in that process.

The *Threads Tab* shows each thread and information about the thread. Selecting a process switches the context to the that thread.

Viewing the Assembler Version of Your Code

You can display your program in source or assembler. You can use the following commands:

Source code (Default)

Select the **View > Source As > Source** command.

Assembler code Select the **View > Source As > Assembler** command.

Viewing the Assembler Version of Your Code

Both Source and assembler

Select the **View > Source As > Both** command.

The Source Pane divides into two parts. The left pane contains the program's source code and the right pane contains the assembler version of this code. You can set breakpoints in either of these panes. Setting an action point at the first instruction after a source statement is the same as setting it at that source statement.

The commands in the following table tell TotalView to display your assembler code by using symbolic or absolute addresses:

Command	Display
View > Assembler > By Address	Absolute addresses for locations and references (default)
View > Assembler > Symbolically	Symbolic addresses (function names and offsets) for locations and references



You can also display assembler instructions in a Variable Window. For more information, see "Displaying Machine Instructions" on page 297.

The following three figures illustrate the different ways TotalView can display assembler code. In the following figure, the second column (the one to the right of the line numbers) shows the absolute address location. The fourth column shows references using absolute addresses.

Figure 118: Address Only
(Absolute Addresses)

```
Function forker in fork_loop.cxx
0x0804a4f7:    0x0c
0x0804a4f8:    0x6a push    $0
0x0804a4f9:    0x00
0x0804a4fa:    0xe8 call    snore(void*)
0x0804a4fb:    0x35
0x0804a4fc:    0xf3
0x0804a4fd:    0xff
0x0804a4fe:    0xff
0x0804a4ff:    0x83 addl    $16,%esp
0x0804a500:    0xc4
0x0804a501:    0x10
1026 0x0804a502:    0xc9 leave
0x0804a503:    0xc3 ret
1032 0x0804a504:    0xb5 pushl   %ebp
fork_wrapper(int): 0x89 movl    %esp,%ebp
0x0804a505:    0xe5
0x0804a506:    0x83 subl    $88,%esp
0x0804a507:    0xec
0x0804a508:    0x58
0x0804a509:    0x8e
1033 0x0804a50a:    0x88 call    pthread_self@GLIBC_2.0
0x0804a50b:    0x29
0x0804a50c:    0xe8
0x0804a50d:    0xff
```

The following figure shows information symbolically. The second column shows locations using functions and offsets. (See Figure 119 on page 173.)

The final assembler figure shows the split Source Pane, with one side showing the program's source code and the other showing the assembler version. In this example, the assembler is shown symbolically. How it is shown depends on whether you've selected **View > Assembler > By Address** or **View > Assembler > Symbolically**. (See Figure 120 on page 173.)

Figure 119: Assembly Only
(Symbolic Addresses)

```

Function forker in fork_loop.cxx
1026:    forker(long)+0x5b3:      0x0c
1027:    forker(long)+0x5b4:      0x6a  push   $0
1028:    forker(long)+0x5b5:      0x00
1029:    forker(long)+0x5b6:      0xe8  call    snore(void*)
1030:    forker(long)+0x5b7:      0x35
1031:    forker(long)+0x5b8:      0xf3
1032:    forker(long)+0x5b9:      0xff
1033:    forker(long)+0x5ba:      0xff
1034:    forker(long)+0x5bb:      0x83  addl   $16, %esp
1035:    forker(long)+0x5bc:      0xc4
1036:    forker(long)+0x5bd:      0x10
1037:    forker(long)+0x5be:      0xc9  leave
1038:    forker(long)+0x5bf:      0xc3  ret
1039:    fork_wrapper(int):       0x55  pushl  %ebp
1040:    fork_wrapper(int)+0x01:  0x89  movl   %esp, %ebp
1041:    fork_wrapper(int)+0x02:  0xe5
1042:    fork_wrapper(int)+0x03:  0x83  subl   $88, %esp
1043:    fork_wrapper(int)+0x04:  0xec
1044:    fork_wrapper(int)+0x05:  0x58
1045:    fork_wrapper(int)+0x06:  0xe8  call    pthread_self@GLIBC_2.0
1046:    fork_wrapper(int)+0x07:  0x29
1047:    fork_wrapper(int)+0x08:  0xe8
1048:    fork_wrapper(int)+0x09:  0xff

```

Figure 120: Both Source and Assembler (Symbolic Addresses)

```

Function forker in fork_loop.cxx
1015:    ) /* if */
1016:    ) /* for */
1017:    STOP
1018:    if (failures)
1019:        sleep_for_seco
1020:    } while (failures)
1021:    /* if */
1022:    if (!please_shut_up)
1023:        printf ("Pid %d: Sleep
1024:        fflush (stdout);
1025:        snore (0);
1026:    } /* fork */
1027:    /* Spin a second thread. i
1028: ****
1029:    /* Spin a second thread. i
1030:
1031: void fork_wrapper (int for
1032:
1033:     pthread_t my_ptid = pthr
1034:     pthread_t new_tid;
1035:     pthread_attr_t attr;
1036:     int whoops;
1037:     int local_fork_count;

```



When TotalView displays instructions, the arguments are almost always in the following order: "source,destination". On Linux-x86 and Linux x86-64 platforms, this can be confusing as the order indicated in AMD and Intel technical literature indicates that the order is usually "destination,source". The order in which TotalView displays this information conforms to the GNU assembler. This ordering is usually only an issue when you are examining a core dump.

Diving into Objects



Diving, which is clicking your middle mouse button on something in a TotalView window, is one of TotalView's more distinguishing features.

In some cases, single-clicking performs a dive. For example, single-clicking on a function name in the Stack Trace Pane tells TotalView to dive into the function. In other cases, double-clicking does the same thing.

Diving on processes and threads in the Root Window is the quickest way to display a Process Window that contains information about what you're diving on. The procedure is simple: dive on a process or thread and TotalView takes care of the rest. Another example is diving on variables in the Process Window, which tells TotalView to display information about the variable in a Variable Window.

The following table describes typical diving operations:

Items you dive on:	Information Displayed:
Process or thread	When you dive on a thread in the Root Window, TotalView finds or opens a Process Window for that process. If it doesn't find a matching window, TotalView replaces the contents of an existing window and shows you the selected process.
Variable	The variable displays in a Variable Window.
Expression List Variable	Same as diving on a variable in the Source Pane: the variable displays in a Variable Window.
Routine in the Stack Trace Pane	The stack frame and source code for the routine appear in a Process Window.
Array element, structure element, or referenced memory area	The contents of the element or memory area replace the contents that were in the Variable Window. This is known as a <i>nested</i> dive.
Pointer	TotalView dereferences the pointer and shows the result in a separate Variable Window. Given the nature of pointers, you may need to cast the result into the logical data type.
Subroutine	The source code for the routine replaces the current contents of the Source Pane. When this occurs TotalView places a right angle bracket (>) in the process's title. Every time it dives, it adds another angle bracket. See the figure that follows this table
Variable Window	A routine must be compiled with source-line information (usually, with the <code>-g</code> option) for you to dive into it and see source code. If the subroutine wasn't compiled with this information, TotalView displays the routine's assembler code.
Expression List Window	TotalView replaces the contents of the Variable Window with information about the variable or element you're diving on.
	TotalView displays information about the variable in a separate Variable Window.

Figure 121: Nested Dive

Function >>>`pthread_mutex_lock`



Diving on a struct or class member that is out of scope does not work.

TotalView tries to reuse windows. For example, if you dive on a variable and that variable is already being displayed in a window, TotalView pops the

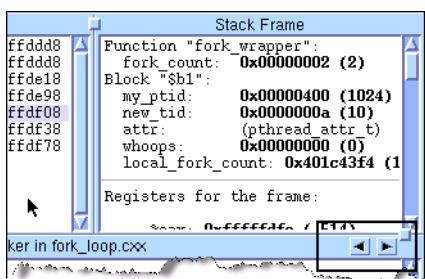
window to the top of the display. If you want the information to appear in a separate window, use the **View > Dive in New Window** command.



*Diving on a process or a thread might not create a new window if TotalView determines that it can reuse a Process Window. If you really want to see information in two windows, use the Process Window **Window > Duplicate** command.*

When you dive into functions in the Process Window, or when you are chasing pointers or following structure elements in the Variable Window, you can move back and forth between your selections by using the *forward* and *backward* buttons. The boxed area of the following figure shows the location of these two controls.

Figure 122: Backward and Forward Buttons



For additional information about displaying variable contents, see "Diving in Variable Windows" on page 298.

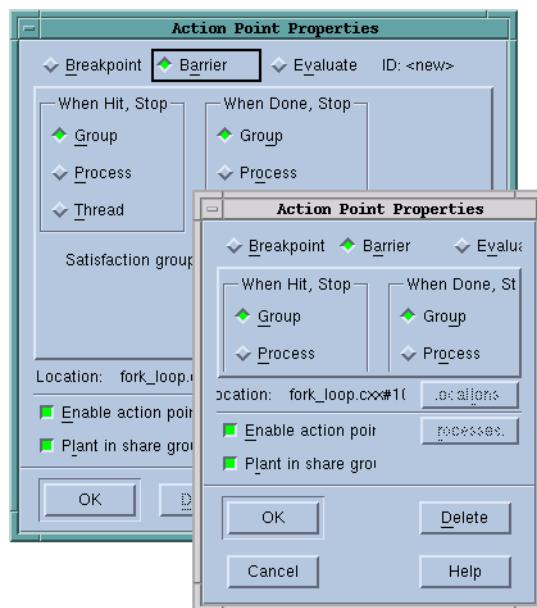
You can also use the following additional windowing commands:

- **Window > Duplicate:** (Variable and Expression List Windows) Creates a duplicate copy of the current Variable Window.
- **File > Close:** Closes an open window.
- **File > Close Relatives:** Closes windows that are related to the current window. The current window isn't closed.
- **File > Close Similar:** Closes the current window and all windows similar to it.

Resizing and Positioning Windows and Dialog Boxes

You can resize most TotalView windows and dialog boxes. While TotalView tries to do the right thing, you can push things to the point where shrinking doesn't work very well. Figure 123 on page 176 shows a before-and-after look in which a dialog box was made too small.

Figure 123: Resizing (and Its Consequences)



Many programmers like to have their windows always appear in the same position in each session. The following two commands can help:

- **Window > Memorize:** Tells TotalView to remember the position of the current window. The next time you bring up this window, it'll be in this position.
- **Window > Memorize All:** Tells TotalView to remember the positions of most windows. The next time you bring up any of the windows displayed when you used this command, it will be in the same position.

Most modern window managers such as KDE or Gnome do an excellent job managing window position. If you are using an older window manager such as **twm** or **mwm**, you may want to select the **Force window positions (disables window manager placement modes)** check box option located on the **Options Page** of the **File > Preferences** Dialog Box. This tells TotalView to manage a window's position and size. If it isn't selected, TotalView only manages a window's size.

Editing Text

The TotalView field editor lets you change the values of fields in windows or change text fields in dialog boxes.

To edit text:

- 1 Click the left mouse button to select the text you want to change. If you can edit the selected text, TotalView will display an editing cursor.

2 Edit the text and press Return.

Like other Motif-based applications, you can use your mouse to copy and paste text in TotalView and to other X Windows applications by using your mouse buttons.

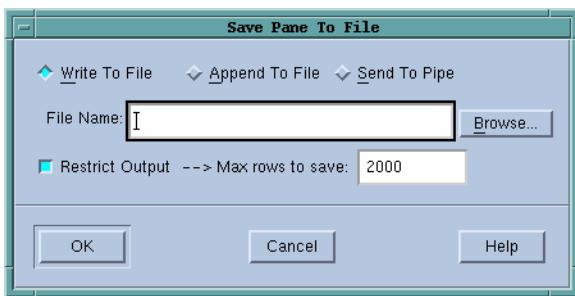
You can also manipulate text by using **Edit > Copy**, **Edit > Cut**, **Edit > Paste**, and **Edit > Delete**. If you haven't yet pressed the Return key to confirm your change, you can use the **Edit > Undo** command to restore information.

Usually TotalView dives when you click your middle-mouse button on something. However, if TotalView is displaying an editing cursor, clicking your middle-mouse button pastes text.

Saving the Contents of Windows

You can write an ASCII equivalent to most pages and panes by using the **File > Save Pane** command. This command also lets you pipe data to UNIX shell commands. (See Figure 124.)

Figure 124: *File > Save Pane* Dialog Box



If the window or pane contains a lot of data, you can use the **Restrict Output** option to limit how much information TotalView writes or sends. For example, you might not want to write a 100 x 100 x 10,000 array to disk. If this option is checked (the default), TotalView only sends the indicated number of lines. You can, of course, change the amount indicated here.

When piping information, TotalView sends what you've typed to **/bin/sh**. This means that you can enter a series of shell commands. For example, the following is a command that ignores the top five lines of output, compares the current ASCII text to an existing file, and writes the differences to another file:

```
| tail +5 | diff - file > file.diff
```

Saving the Contents of Windows

Visualizing Programs and Data



TotalView provides a set of tools that let you visualize arrays and your program's activity. This chapter describes:

- "Displaying Call Graphs" on page 179
- "Visualizing Array Data" on page 181

If you are running an MPI program, "Displaying the Message Queue Graph Window" on page 107 describes another visualization tool.

CHAPTER 9

Displaying Call Graphs

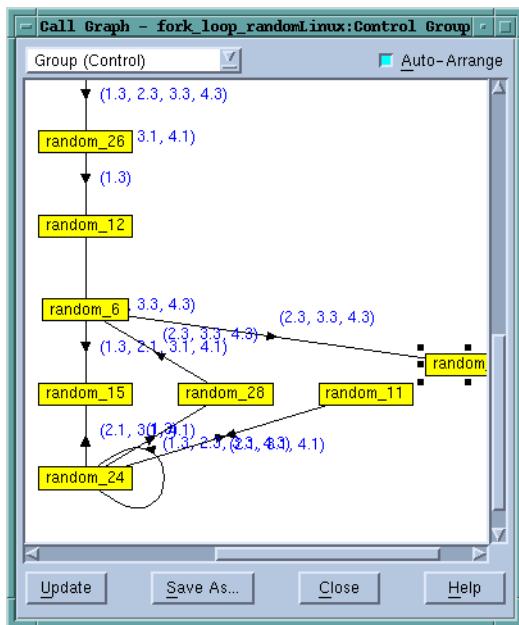
Debugging is an art, not a science. Debugging often means having the intuition to make guesses about what a program is doing and where to look for what is causing the problem. Locating a problem is often 90% or more of the effort. The call graph can help you understand what your program is doing so that you can begin to understand how your program is executing.

The call graph is a diagram that shows all the currently active routines. These routines are linked by arrows indicating that one routine is called by another. The call graph is a *dynamic* call graph in that it displays the call graph at the time when TotalView creates it. The **Update** button tells TotalView to recreate this display.

To display a call graph, choose the **Tools > Call Graph** command in the Process Window. (A sample call graph is shown on the next page.)

You can tell TotalView to display a call graph for the processes and threads specified with the controls at the top of this window. If you don't touch these controls, TotalView displays a call graph for the group defined in the toolbar of your Process Window. If TotalView is displaying the call graph for

Figure 125: Tools > Call Graph Dialog Box



a multi-process or multi-threaded program, numbers next to the arrows indicate which threads have a routine on their call stack.

If you dive on a routine within the call graph, TotalView creates a group called **call_graph**. This group contains all of the threads that have the routine you dived on in its call stack. If you look at the Process Window's Processes tab, you'll see that the **call_graph** set is selected in the scope pull-down. In addition, the context of the Process Window changes to the first thread in the set.

As you begin to understand your program, you will see that it has a rhythm and a dynamic that is reflected in this diagram. As you examine and understand this structure, you will sometimes see things that don't look right—which is a subjective response to how your program is operating. These places are often where you want to begin looking for problems.

By diving on a routine that doesn't look right, you'll isolate the processes into their own group so that you can find out what is occurring there. Every time you dive on a routine, TotalView overwrites the group. If you want to preserve the group, use the **Groups > Custom Groups** command to make a copy of the group.

Looking at the call graph can also tell you where bottlenecks are occurring. For example, if one routine is used by many other routines, and that routine controls a shared resource, this thread might be negatively affecting performance.

Visualizing Array Data

The TotalView Visualizer creates graphic images of your program's array data. Topics in this section are:

- "Command Summary" on page 181
- "How the Visualizer Works" on page 182
- "Viewing Data Types in the Visualizer" on page 182
- "Visualizing Data Manually" on page 183
- "Using the Visualizer" on page 184
- "Using the Graph Window" on page 186
- "Using the Surface Window" on page 187
- "Visualizing Data Programmatically" on page 192
- "Launching the Visualizer from the Command Line" on page 193
- "Configuring TotalView to Launch the Visualizer" on page 193



The Visualizer isn't available on Linux Alpha and 32-bit SGI Irix. It's available on all other platforms. At release 7.0.1, the Visualizer was re-engineered. If you are using an older release, you'll need to see the documentation for that release.

Command Summary

While the remainder of this chapter describes the Visualizer, you can bet by with the information described in the following table. This table summarizes most of what you need to know when using the Visualizer.

Action	Click or Press	
Camera mode	Actor mode	
Rotate camera around focal point (surface only)	Rotate actor around focal point (surface only)	Left mouse button
Zoom	Scale	Right mouse button
Pan	Translate	Middle mouse button or Shift-left mouse button
Other Functions		
Pick (show value)		p
Camera mode: mouse events affect the camera position and focal point. (The axis moves and you don't.)		c
Actor mode: mouse events affect the actor that is under the mouse pointer.Joystick-mode. (You move and the axis doesn't.)		a
Joystick mode: motion occurs continuously while a mouse button is pressed		j
Trackball mode: motions only occurs when the mouse button is presses and the mouse pointer moves.		t
Wireframe view		w
Surface view		s

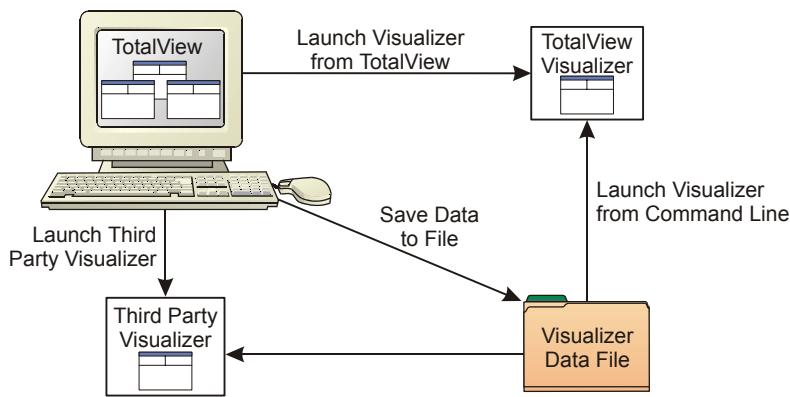
Action	Click or Press
Reset	r
Initialize	I
Exit or Quit	Ctrl-Q

How the Visualizer Works

The Visualizer is a stand-alone program to which TotalView sends information. Because it is separate, you can use it in more than one way; for example:

- When you launch it from within TotalView, you can see your program's data as you are debugging your program.
- If you save the data that would be sent to the Visualizer, you can view it later by invoking the Visualizer from the command line. (See Figure 126.)

Figure 126: TotalView Visualizer Relationships



- Because TotalView is sending a data stream to the Visualizer, you can even replace our Visualizer with any tool that can read this data.



The online Help contains information on adapting a third-party visualizer so that it can be used with TotalView. This is also on our web site at http://www.totalviewtech.com/Documentation/latest/html/User_Guide/AdaptingaThirdPartyVisualizer.html.

Viewing Data Types in the Visualizer

The data selected for visualization is called a *dataset*. TotalView treats stack variables at different recursion levels or call paths as different datasets.

TotalView can visualize one- and two-dimensional arrays of integer or floating-point data. If an array has more than two dimensions, you can visualize part of it using an array slice that creates a subarray having fewer dimensions. Figure 127 on page 183 shows a three-dimensional variable sliced so that one of the dimensions is invariant.

Figure 127: A Three-Dimensional Array Sliced into Two Dimensions

Field	Value
(1,1,1)	1.00088
(2,1,1)	1.75176
(3,1,1)	2.50264
(4,1,1)	3.25352
(5,1,1)	4.0044
(6,1,1)	4.75528
(7,1,1)	5.50616
(8,1,1)	6.25704
(9,1,1)	7.00792
(10,1,1)	7.7588

Viewing Data

Different datasets can require different views to display their data. For example, a graph is more suitable for displaying one-dimensional datasets or two-dimensional datasets if one of the dimensions has a small extent. However, a surface view is better for displaying a two-dimensional dataset.

When TotalView launches the Visualizer, one of the following actions occurs:

- If the Visualizer is displaying the dataset, it raises the dataset's window to the top of the desktop. If you had minimized the window, the Visualizer restores it.
- If you previously visualized a dataset but you've killed its window, the Visualizer creates a new window using the most recent visualization method.
- If you haven't visualized the dataset, the Visualizer chooses an appropriate method. If you don't want the Visualizer to choose, disable this feature by using the **Options > Auto Visualize** command in the Visualizer Directory Window.

Visualizing Data Manually

Before you can visualize an array, you must:

- Open a Variable Window for the array's data.
- Stop program execution when the array's values are set to what you want them to be when they are visualized.

You can restrict the data being visualized by editing the **Slice** field. For example, editing the **Slice** field limits the amount of data being visualized. (See "Displaying Array Slices" on page 334.) Limiting the amount of data increases the speed of the Visualizer.

After selecting the Variable Window **Tools > Visualize** command, the Visualizer creates its window. The data sent to the Visualizer isn't automatically updated as you step through your program. Instead, you must explicitly update the display by selecting the **Tools > Visualize** again.

TotalView can visualize variables across threads or processes. (See "Visualizing a "Show Across" Variable Window" on page 347.) When you visualize this display of the "Show Across" information, the Visualizer uses the process or thread index as one dimension. This means that you can only visualize scalar or vector information. If you do not want the process or thread index to be a dimension, do not use a **Show Across** command.

Using the Visualizer

The Visualizer uses two types of windows:

■ Dataset Window

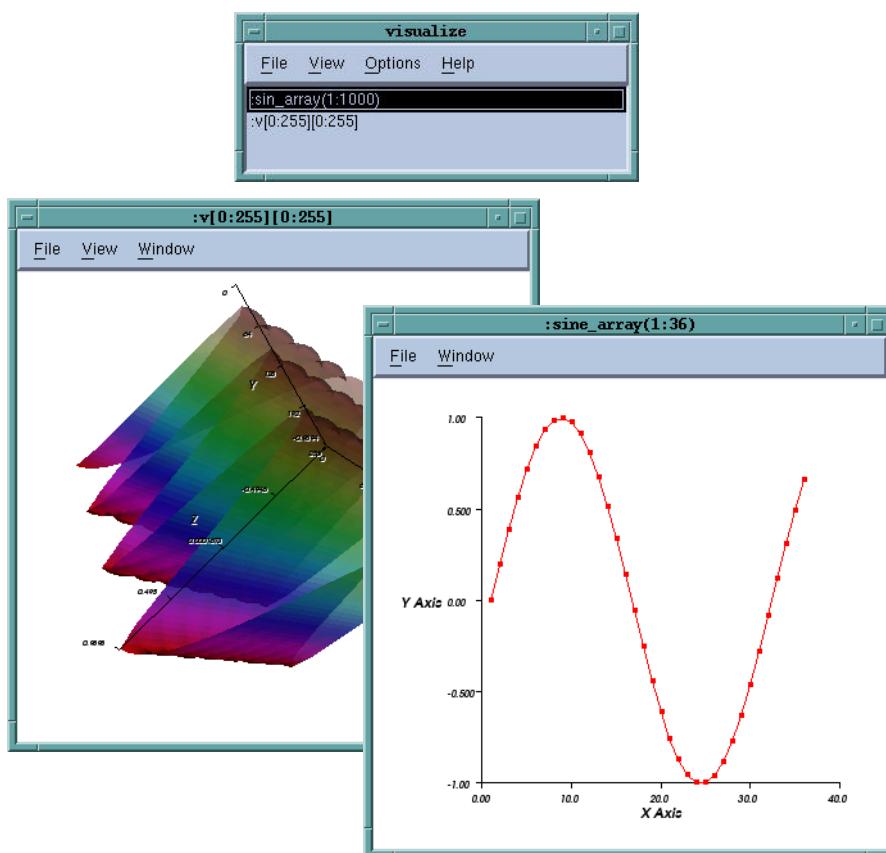
This window contains the datasets that you can visualize. Use this window to set global options and to create views of your datasets. Commands in this window let you obtain different views of the same data by allowing you to open more than one View Window.

■ View Window

These are the windows that display your data. The commands in a View Window let you set viewing options and change the way the Visualizer displays your data.

In Figure 128, the top window is a Dataset Window. The two remaining windows show a surface and a graph view.

Figure 128: Sample Visualizer Windows



Using Dataset Window Commands

The Dataset Window shows you the datasets you can display. Double-clicking upon a dataset tells the Visualizer to display it.

The **View** menu lets you select **Graph** or **Surface** visualization. Whenever TotalView sends a new dataset to the Visualizer, the Visualizer updates its dataset list. To delete a dataset from the list, click on it, display the File menu, and then select Delete. (It's usually easier to just close the Visualizer.)

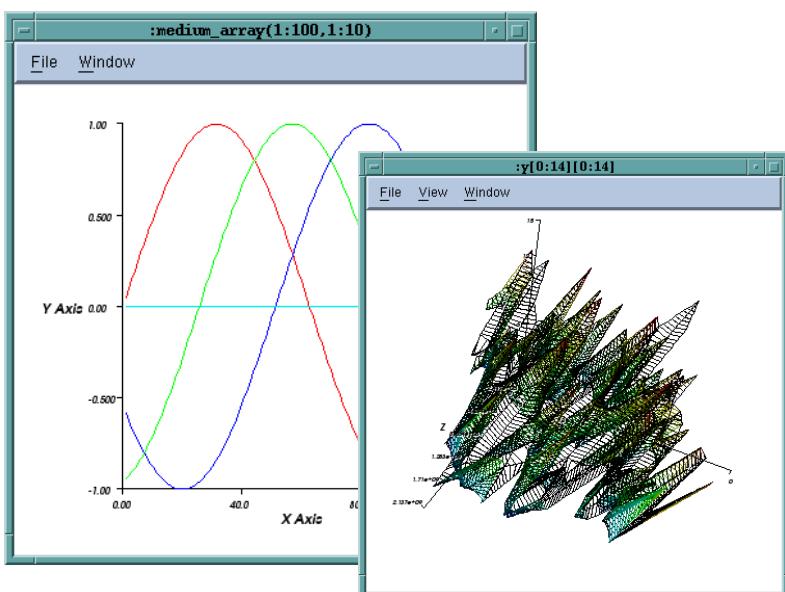
The following commands are in the Dataset Window menu bar:

- File > Delete** Deletes the currently selected dataset. It removes the dataset from the dataset list and *destroys* the View Windows that displays it.
- File > Exit** Closes all windows and exits the Visualizer.
- View > Graph** Creates a new Graph Window; see "Using the Graph Window" on page 186.
- View > Surface** Creates a new Surface Window; see "Using the Surface Window" on page 187.
- Options > Auto Visualize** This item is a toggle; when enabled, the Visualizer automatically visualizes new datasets as they are read. Typically, this option is left on. If, however, you have large datasets and you want to configure how the Visualizer displays the graph, you should disable this option.

Using View Window Commands

View Windows display graphic images of your data. Figure 129 shows a graph view and a surface view. The View Window's title is the text that appears in the Dataset Window.

Figure 129: Graph and Surface Visualizer Windows



The View Window menu commands are as follows:

File > Close Closes the View Window.

File > Dataset Raises the Dataset Window to the front of the desktop. If you minimized the Dataset Window, the Visualizer restores it.

File > Delete Deletes the View Window dataset from the dataset list. This also destroys other View Windows that view the dataset.

File > Options Pops up a window of viewing options.

Window > Duplicate Base Window Creates a new View Window that has the same visualization method and dataset as the current View Window.

The drawing area displays the image of your data. You can interact with the drawing area to alter the view of your data. For example, if the Visualizer is showing a surface, you can rotate the surface to view it from different angles. You can also get the value and indices of the dataset element nearest the cursor by clicking on it and typing "P". A pop-up window displays the information. These operations are discussed in "*Using the Graph Window*" on page 186 and "*Using the Surface Window*" on page 187.

Using the Graph Window

The Graph Window displays a two-dimensional graph of one- or two-dimensional datasets. If the dataset is two-dimensional, the Visualizer displays multiple graphs. When you first create a Graph Window on a two-dimensional dataset, the Visualizer uses the dimension with the larger number of elements for the X axis. It then draws a separate graph for each subarray that has the smaller number of elements. If you don't like this choice, you can transpose the data by selecting a checkbox within the **File > Options** Dialog Box.



You probably don't want to use a graph to visualize two-dimensional datasets with large extents in both dimensions as the display can be very cluttered. If you try, the Visualizer only shows the first ten.

You can display graphs with points for each element of the dataset, with lines connecting dataset elements, or with both lines and points. (See Figure 130 on page 187.)

If the Visualizer is displaying more than one graph, each is displayed in a different color. The X axis of the graph is annotated with the indices of the long dimension. The Y axis shows you the data value.

Displaying Graph Views

The **File > Options** Dialog Box lets you control how the Visualizer displays the graph. (A different dialog box appears if the Visualizer is displaying a surface view. See Figure 131 on page 187.)

Figure 130: Visualizer Graph View Window

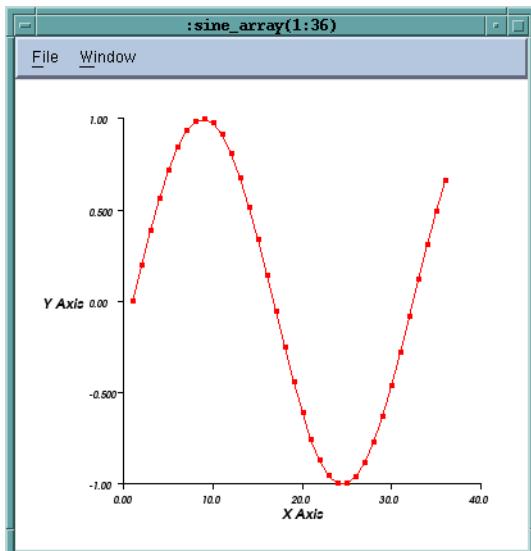
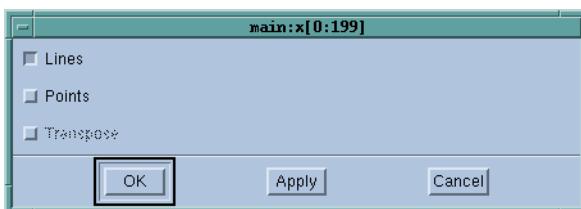


Figure 131: Graph Options Dialog Box



The following describes the meanings of these check boxes:

- | | |
|------------------|--|
| Lines | When set, the Visualizer displays lines connecting dataset elements. |
| Points | When set, the Visualizer displays points (markers) for dataset elements. |
| Transpose | When set, the Visualizer inverts which axis is held constant when the Visualizer generates a graph of a two-dimensional graph. If you are not graphing a two-dimensional object, the Visualizer grays out this checkbox. |

Figure 132 on page 188 shows a sine wave displayed in three different ways:

To see the value of a dataset's element, place your cursor near a graph marker, and type "P". The Visualizer responds by displaying its value. The bottom graph in Figure 132 on page 188 shows the value of a data point.

Using the Surface Window

The Surface Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (X and Y axes) of the display. Figure 133 on page 188 shows a surface view

Visualizing Array Data

Figure 132: Sine wave
Displayed in Three Ways

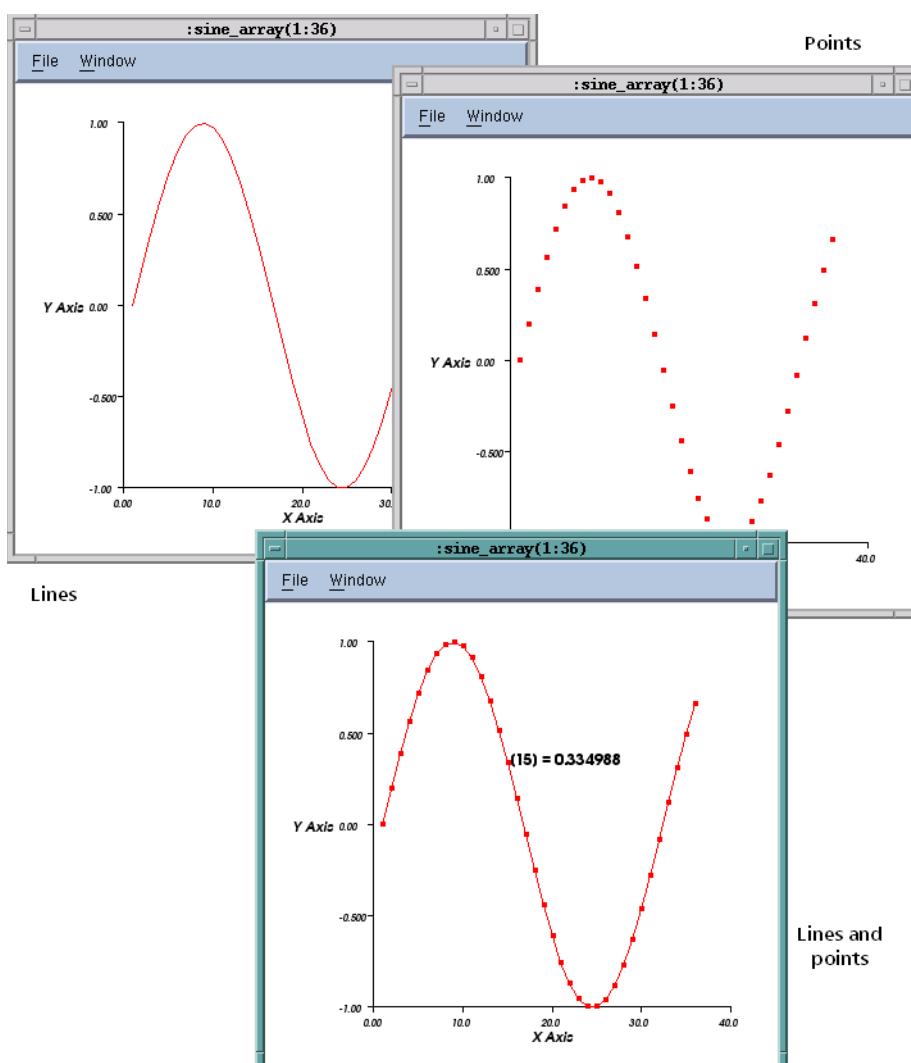
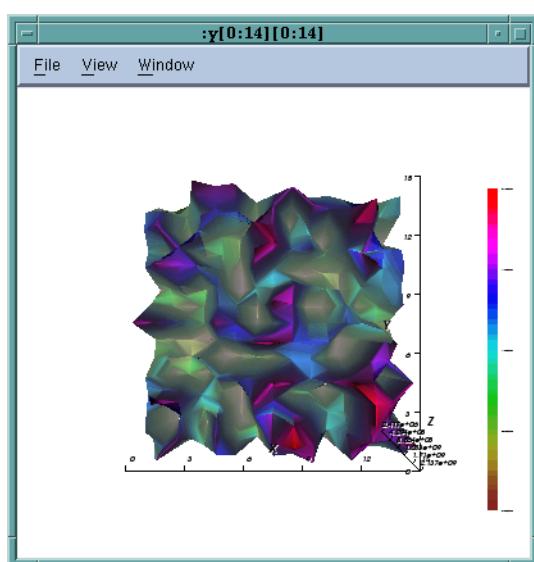
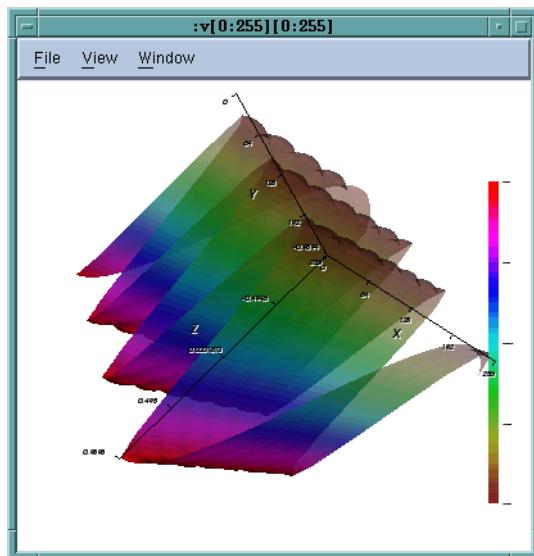


Figure 133: A Surface View



The following figure shows a three-dimensional surface that maps element values to the height (Z axis).

Figure 134: A Surface View of a Sine Wave



Displaying Surface Views

The Surface Window File > Options command (see Figure 135) lets you control how the Visualizer displays the graph. (A different dialog box appears if the Visualizer is displaying a Graph View.)

Figure 135: Surface Options Dialog Box



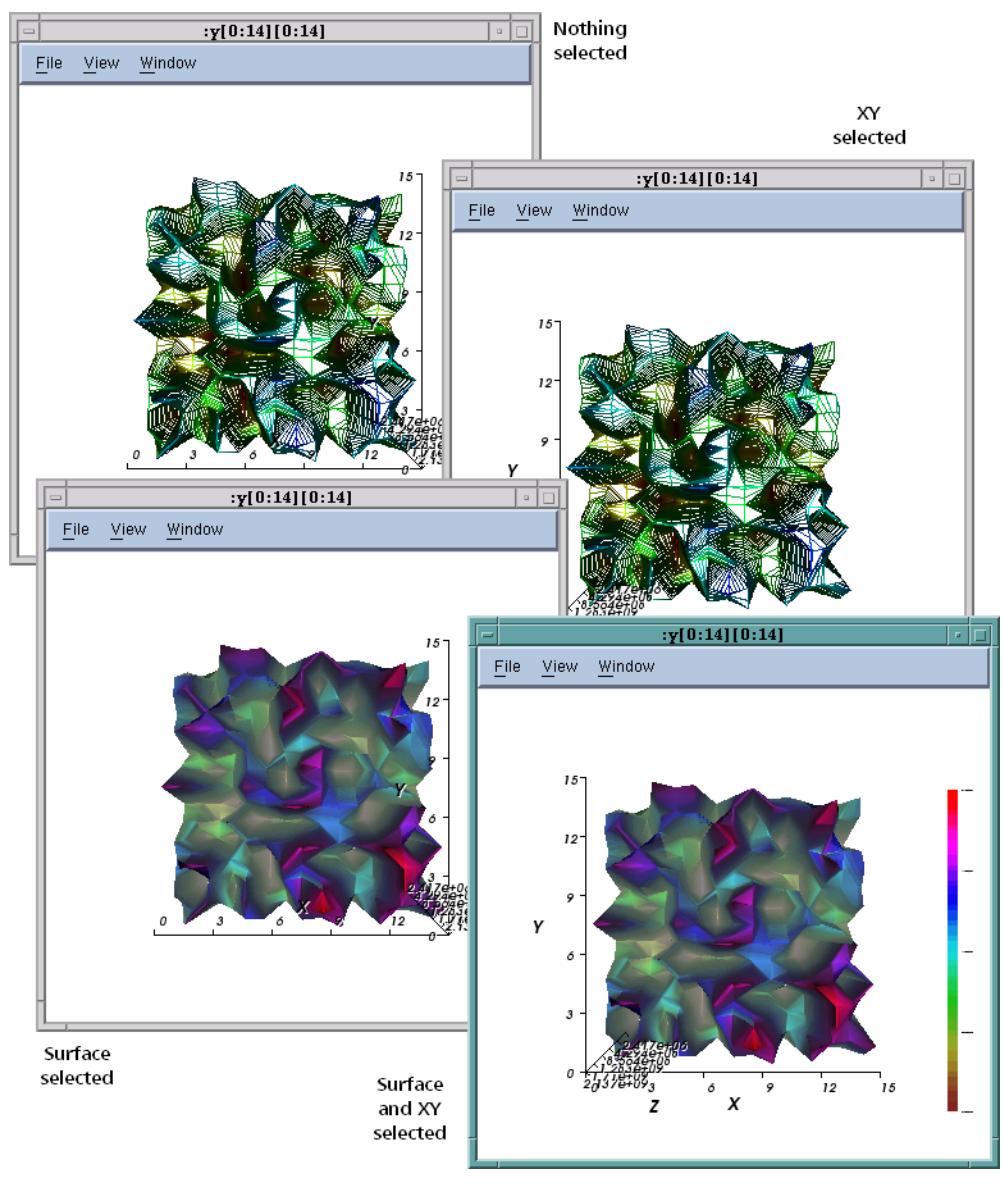
The following describes the meaning of these check boxes:

- | | |
|--------------------|---|
| Surface | If this option is set, the Visualizer displays the array's data as a three-dimensional surface. If you don't set this option, the Visualizer displays the surface as a grid. |
| XY | If this option is set, the Visualizer reorients the view's XY axes. The Z axis is perpendicular to the display. |
| Auto Reduce | If this option is set, the Visualizer derives the displayed surface by averaging over neighboring elements in the original dataset. This speeds up visualization by reducing the resolution of the surface. Clear this option if you want to accurately visualize all dataset elements. |

The **Auto Reduce** option lets you choose between viewing all your data points—which takes longer to appear in the display—or viewing the averaging of data over a number of nearby points.

Figure 136 shows four different views of the same data, differing only in the selection of Surface and XY options.

Figure 136: Four Surface Views



You can reset the viewing parameters to those used when you first invoked the Visualizer by selecting the **View > Initialize View** command, which restores all translation, rotation, and scaling to their initial state, and enlarges the display area slightly.

Manipulating Surface Data

The actions that the Visualizer performs differ when you are in actor mode instead of camera mode, which is the default. In camera mode, the Visualizer pretends to be moving a "camera" that is showing you the view. In actor mode, the Visualizer assumes that changes to the way you are seeing the object are due to you changing your position. The difference between these is subtle. In some circumstances, actions such as pan and zoom in camera mode can also add a slight rotation to the object.

From within TotalView, you can only see one array at a time. However, if you combine multiple datasets and visualize them externally, the differences between camera and actor mode can help differentiate the objects.

The following table defines general commands that you can use while displaying a surface view. Command letters can be typed in either upper- or lower-case.

Action	Press
Pick (show value): The Visualizer shows the value of the data point underneath the cursor.	p
Camera mode: Mouse events affect the camera position and focal point. (Axes moves, and you don't.)	c
Actor mode: Mouse events affect the actor that is under the mouse pointer. (You move, not the axes.)	a
Joystick mode: Motion occurs continuously while you are pressing a mouse button.	j
Trackball mode: Motions only occurs when you press the mouse button and you move the mouse pointer.	t
Wireframe view: The Visualizer displays the surface as a mesh. (This is the same as not checking the Surface option.)	w
Surface view: The Visualizer displays the surface as a solid. (This is the same as having checked the Surface option.)	s
Reset: Removes some of the changes you've made to the way the Visualizer displays an object.	r
Initialize: Restores the object to what it was before you interacted with the Visualizer. As this is a menubar accelerator, the window must have focus.	i
Exit or Quit: Close the Visualizer or	Ctrl-Q

The following table defines the actions you can perform using your mouse:

Action	Click or Press
Camera mode	Actor mode
Rotate camera around focal point (surface only)	Rotate actor around focal point (surface only)
Zoom : you appear to get closer to the object.	Scale : the object appears to get larger
Pan : you move the "camera". For example, moving the camera up means the object moves down.	Translate : The object moves in the direction you pull it.

Visualizing Data Programmatically

The `$visualize` function lets you visualize data from within eval points and the **Tools > Evaluate** Window. Because you can enter more than one `$visualize` function within an eval point or Evaluate Window, you can simultaneous visualize multiple variables.

If you enter the `$visualize` function in an eval point, TotalView interprets rather than compiles the expression, which can greatly decrease performance. See “*Defining Eval Points and Conditional Breakpoints*” on page 366 for information about compiled and interpreted expressions.

Using the `$visualize` function in an eval point lets you animate the changes that occur in your data, because the Visualizer updates the array’s display every time TotalView reaches the eval point. Here is this function’s syntax:

```
$visualize ( array [, slice_string ])
```

The `array` argument names the dataset being visualized. The optional `slice_string` argument is a quoted string that defines a constant slice expression that modifies the `array` parameter’s dataset. In Fortran, you can use either a single (‘) or double-quotation mark (“). You must use a double-quotation mark if your language is C or C++.

The following examples show how you can use this function. Notice that the array’s dimension ordering differs between C/C++ and Fortran.

C and C++	<code>\$visualize (my_array);</code> <code>\$visualize (my_array, "[:2][10:15]");</code> <code>\$visualize (my_array, "[12][:]);</code>
Fortran	<code>\$visualize (my_array)</code> <code>\$visualize (my_array, '(11:16,:2)')</code> <code>\$visualize (my_array, '(:,13)')</code>

The first example in each programming language group visualizes the entire array. The second example selects every second element in the array’s major dimension; it also clips the minor dimension to all elements in the range. The third example reduces the dataset to a single dimension by selecting one subarray.

You may need to cast your data so that TotalView knows what the array’s dimensions are. For example, here is a the C function that passes a two-dimensional array parameter that does not specify the major dimension’s extent.

```
void my_procedure (double my_array[] [32])
{ /* procedure body */ }
```

You would need to cast this before TotalView can visualize it. For example:

```
$visualize (* (double[32] [32])*my_array);
```

Sometimes, it’s hard to know what to specify. You can quickly refine array and slice arguments, for example, by entering the `$visualize` function into the **Tools > Evaluate** Dialog Box. When you select the **Evaluate** button, you

quickly see the result. You can even use this technique to display several arrays simultaneously.

Launching the Visualizer from the Command Line

To start the Visualizer from the shell, use the following syntax:

```
visualize [ -file filename | -persist ]
```

where:

- file *filename* Reads data from *filename* instead of reading from standard input. For information on creating this file, see "Setting the Visualizer Launch Command" on page 194.
- persist Continues to run after encountering an EOF (End-of-File) on standard input. If you don't use this option, the Visualizer exits as soon as it reads all of the data.

By default, the Visualizer reads its datasets from standard input and exits when it reads an EOF. When started by TotalView, the Visualizer reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input, invoke it by using the **-persist** option.

If you want to read data from a file, invoke the Visualizer with the **-file** option:

```
visualize -file my_data_set_file
```

The Visualizer reads all the datasets in the file. This means that the images you see represent the last versions of the datasets in the file.

The Visualizer supports the generic X toolkit command-line options. For example, you can start the Visualizer with the Directory Window minimized by using the **-iconic** option. Your system manual page for the X server or the *X Window System User's Guide* by O'Reilly & Associates lists the generic X command-line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **-xrm resource_setting** option. Using X resources to modify the default behavior of TotalView or the Visualizer is described in greater detail on our Web site at <http://www.totalviewtech.com/Documentation/xresources/XResources.pdf>.

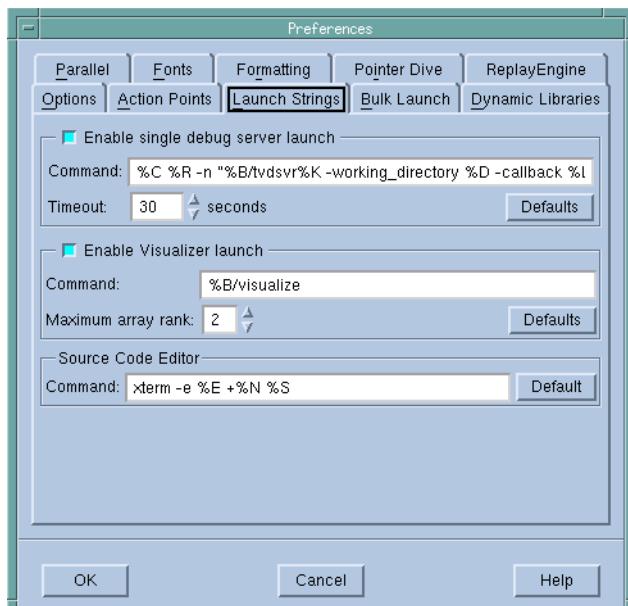
Configuring TotalView to Launch the Visualizer

TotalView launches the Visualizer when you select the **Tools > Visualize** command from the Variable Window. It also launches it if or when you use a **\$visualize** function in an eval point and the **Tools > Evaluate** Dialog Box.

TotalView lets you set a preference that disables visualization. This lets you turn off visualization when your program executes code that contains eval points, without having to individually disable all the eval points.

To change the Visualizer launch options interactively, select **File > Preferences**, and then select the Launch Strings Tab. (See Figure 137 on page 194.)

Figure 137: File > Preferences
Launch Strings Page



The changes you can make using these preferences are:

- Customize the command that TotalView uses to start a visualizer by entering the visualizer's start up command in the **Command** edit box.
- Change the autolaunching option. If you want to disable visualization, clear the **Enable Visualizer launch** check box.
- Change the maximum permissible rank. Edit the value in the **Maximum array rank** field to save the data exported from TotalView or display it in a different visualizer. A rank's value can range from **1** to **16**.
Setting the maximum permissible rank to either **1** or **2** (the default is **2**) ensures that the Visualizer can use your data—the Visualizer displays only two dimensions of data. This limit doesn't apply to data saved in files or to third-party visualizers that can display more than two dimensions of data.
- Clicking the **Defaults** button returns all values to their default values. This reverts options to their default values even if you have used X resources to change them.

If you disable visualization while the Visualizer is running, TotalView closes its connection to the Visualizer. If you reenable visualization, TotalView launches a new Visualizer process the next time you visualize something.

Setting the Visualizer Launch Command

You can change the shell command that TotalView uses to launch the Visualizer by editing the Visualizer launch command. (In most cases, the only reason you'd do this is if you're having path problems or you're running a

different visualizer.) You can also change what's entered here so that you can view this information at another time; for example:

```
cat > your_file
```

Later, you can visualize this information by typing either:

```
visualize -persist < your_file  
visualize -file your_file
```

You can preset the Visualizer launch options by setting X resources. These resources are described on our web site. For more information, go to <http://www.totalviewtech.com/Documentation/>.

Visualizing Array Data

Part IV: Using the CLI



The chapters in this part of the book deal exclusively with the CLI. Most CLI commands must have a process/thread focus for what they do. See Chapter 13: "Using Groups, Processes, and Threads" on page 251 for more information.

Chapter 11: Seeing the CLI at Work

While you can use the CLI as a stand-alone debugger, using the GUI is usually easier. You will most-often use the CLI when you need to debug programs using very communication liens or when you need to create debugging functions that are unique to your program. This chapter presents a few Tcl macros in which CLI commands are embedded.

Most of these examples are simple. They are designed to give you a feel for what you can do.

Chapter 10: Using the CLI

You can use CLI commands without knowing much about Tcl, which is the approach taken in this chapter. This chapter tells you how to enter CLI commands and how the CLI and TotalView interact with one another when used in a non-graphical way.

Using the CLI



The two components of the Command Line Interface (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that lets you debug your program. This chapter looks at how these components interact, and describes how you specify processes, groups, and threads.

This chapter emphasizes interactive use of the CLI rather than using the CLI as a programming language because many of its concepts are easier to understand in an interactive framework. However, everything in this chapter can be used in both environments.

This chapter contains the following sections:

- ["About the Tcl and the CLI" on page 199](#)
- ["Starting the CLI" on page 201](#)
- ["About CLI Output" on page 205](#)
- ["Using Command Arguments" on page 206](#)
- ["Using Namespaces" on page 207](#)
- ["About the CLI Prompt" on page 207](#)
- ["Using Built-in and Group Aliases" on page 208](#)
- ["How Parallelism Affects Behavior" on page 209](#)
- ["Controlling Program Execution" on page 210](#)

About the Tcl and the CLI

The CLI is built in version 8.0 of Tcl, so TotalView CLI commands are built into Tcl. This means that the CLI is not a library of commands that you can bring into other implementations of Tcl. Because the Tcl you are running is

the standard 8.0 version, the CLI supports all libraries and operations that run using version 8.0 of Tcl.

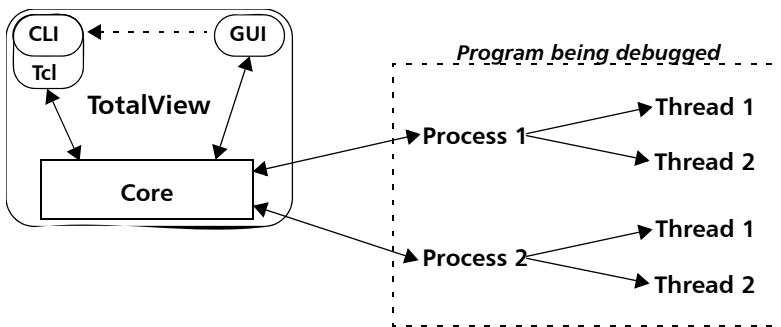
Integrating CLI commands into Tcl makes them intrinsic Tcl commands. This lets you enter and execute all CLI commands in exactly the same way as you enter and execute built-in Tcl commands. As CLI commands are also Tcl commands, you can embed Tcl primitives and functions in CLI commands, and embed CLI commands in sequences of Tcl commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, and then use a CLI command that operates on the elements of this list. You can also create a Tcl function that dynamically builds the arguments that a process uses when it begins executing.

About The CLI and TotalView

The following figure illustrates the relationship between the CLI, the GUI, the TotalView core, and your program:

Figure 138: The CLI, GUI and TotalView



The CLI and GUI are components that communicate with the TotalView core, which is what actually does the work. In this figure, the dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI. The reverse isn't true: you can't invoke the GUI from the CLI.

In turn, the TotalView core communicates with the processes that make up your program, receives information back from these processes, and passes information back to the component that sent the request. If the GUI is also active, the core also updates the GUI's windows. For example, stepping your program from within the CLI changes the PC in the Process Window, updates data values, and so on.

Using the CLI Interface

You interact with the CLI by entering a CLI or Tcl command. (Entering a Tcl command does exactly the same thing in the CLI as it does when interacting with a Tcl interpreter.) Typically, the effect of executing a CLI command is one or more of the following:

- The CLI displays information about your program.
- A change takes place in your program's state.
- A change takes place in the information that the CLI maintains about your program.

After the CLI executes your command, it displays a prompt. Although CLI commands are executed sequentially, commands executed by your program might not be. For example, the CLI doesn't require that your program be stopped when it prompts for and performs commands. It only requires that the last CLI command be complete before it can begin executing the next one. In many cases, the processes and threads being debugged continue to execute after the CLI finished doing what you asked it to do.

If you need to stop an executing command or Tcl macro, press **Ctrl+C** while the command is executing. If the CLI is displaying its prompt, typing **Ctrl+C** stops executing processes.

Because actions are occurring constantly, state information and other kinds of messages that the CLI displays are usually mixed in with the commands that you type. You might want to limit the amount of information TotalView displays by setting the **VERBOSE** variable to **WARNING** or **ERROR**. (For more information, see the "Variables" chapter in the *TotalView Reference Guide*.)

Starting the CLI

You can start the CLI in one of the following ways:

- You can start the CLI from the GUI by selecting the **Tools > Command Line** command in the Root or Process Windows. After selecting this command, TotalView opens a window into which you can enter CLI commands.
- You can start the CLI directly from a shell prompt by typing **totalviewcli**. (This assumes that the TotalView binary directory is in your path.)

Figure 139 on page 202 is a snapshot of a CLI window that shows part of a program being debugged.

If you have problems entering and editing commands, it might be because you invoked the CLI from a shell or process that manipulates your **stty** settings.

Figure 139: CLI xterm Window

```

TotalView Command Line Input
d1.◇ s
81 >     denorms(i) = x'00000001'
d1.◇ s
820> 40  continue
d1.◇ dlist -n 6
79
800      do 40 i = 1, 500
81       denorms(i) = x'00000001'
820> 40  continue
83      do 42 i = 500, 1000
84       denorms(i) = x'80000001'
d1.◇ dstatus
1          (4656)  Breakpoint [arraysLINUX]
1.1        (4656/4656) Breakpoint PC=0x08048fa8, [arrays,F#82]
d1.◇ duhere
> 0 MAIN_          PC=0x08048fa8, FP=0xbffffdaa8 [arrays,F#82]
  1 main           PC=0x0804909e, FP=0xbffffdac8 [/nfs/fs/u3/home/barryk/Examp
1eProgs/arraysLINUX]
  2 __libc_start_main PC=0x40065647, FP=0xbffffdb08 [.,sysdeps/generic/libc-sta
rt.c#129]
d1.◇ dup
  1 main           PC=0x0804909e, FP=0xbffffdac8 [/nfs/fs/u3/home/barryk/Examp
1eProgs/arraysLINUX]
d1.◇ ■

```

You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option isn't available, you have to change values individually.)

If you start the CLI with the **totalviewcli** command, you can use all of the command-line options that you can use when starting TotalView, except those that have to do with the GUI. (In some cases, TotalView displays an error message if you try. In others, it just ignores what you did.)

Startup Example

The following is a very small CLI script:

```

#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg

```

This script begins by loading and interpreting the **make_actions.tcl** file, which was described in Chapter 11, "Seeing the CLI at Work," on page 213. It then loads the **fork_loop** executable, sets its default startup arguments, and steps one source-level statement.

If you stored this in a file named **fork_loop.tvd**, you can tell TotalView to start the CLI and execute this file by entering the following command:

```
totalviewcli-s fork_loop.tvd
```

Information on command-line options is in the "TotalView Command Syntax" chapter of the *TotalView Reference Guide*.

The following example places a similar set of commands in a file that you invoke from the shell:

```

#!/bin/sh
# Next line exec. by shell, but ignored by Tcl because: \
exec totalviewcli-s "$0" "$@"

```

```

#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg

```

The only real difference between the last two examples is the first few lines in the file. In this second example, the shell ignores the backslash continuation character; Tcl processes it. This means that the shell executes the **exec** command while Tcl will ignore it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute your program from within the CLI, enter a **dload** command followed by the **drun** command.



If your program is launched from a starter program such as srun or yod, use the drerun command rather than drun to start your program. If you use drun, default arguments to the process are suppressed; drerun passes them on.

The following example uses the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. Since this was not the first time the file was run, breakpoints exist from a previous session.



In this listing, the CLI prompt is "d1.<>". The information preceding the greater-than symbol (>) symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in "About the CLI Prompt" on page 207.

```

% totalviewcli
d1.<> dload arraysAlpha      #load the arraysAlpha program
1
d1.<> dactions              # Show the action points
No matching breakpoints were found
d1.<> dlist -n 10 75
75          real16_array (i, j) = 4.093215 * j+2
76 #endif
77 26    continue
78 27    continue
79
80 do 40 i = 1, 500
81     denorms(i) = x'00000001'
82 40 continue
83 do 42 i = 500, 1000
84     denorms(i) = x'80000001'
d1.<> dbreak 80                # Add two action points
1
d1.<> dbreak 83
2
d1.<> drun                   # Run the program to the action point

```

This two-step operation of loading and running lets you set action points before execution begins. It also means that you can execute a program more than once. At a later time, you can use the **drerun** command to restart your program, perhaps sending it new command-line arguments. In contrast, reentering the **dload** command tells the CLI to reload the program into memory (for example, after editing and recompiling the program). The **dload** command always creates new processes. This means that you get a new process each time the CLI executes it. The CLI does not, however, remove older ones.

The **dkill** command terminates one or more processes of a program started by using a **dload**, **drun**, or **drerun** command. The following example continues where the previous example left off:

```

d1.<> dkill          # kills process
d1.<> drun           # runs program from start
d1.<> dlist -e -n 3   # shows lines about current spot
    79
    80@>      do 40 i = 1, 500
    81          denorms(i) = x'00000001'
d1.<> dwhat master_array # Tell me about master_array
In thread 1.1:
Name: master_array; Type: integer(100);
Size: 400 bytes; Addr: 0x140821310
Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
(Scope class: Any)
Address class: proc_static_var
(Routine static variable)
d1.<> dgo            # Start program running
d1.<> dwhat denorms   # Tell me about denorms
In thread 1.1:
Name: denorms; Type: <void>; Size: 8 bytes;
Addr: 0x1408214b8
Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
(Scope class: Any)
Address class: proc_static_var
(Routine static variable)
d1.<> dprint denorms(0) # Show me what is stored
denorms(0) = 0x0000000000000001 (1)
d1.<>
```

Because information is interleaved, you may not realize that the prompt has appeared. It is always safe to use the Enter key to have the CLI redisplay its prompt. If a prompt isn't displayed after you press Enter, you know that the CLI is still executing.

About CLI Output

A CLI command can either print its output to a window or return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you won't care about the difference between *printing* and *returning-and-printing*. Either way, the CLI displays information in your window. And, in both cases, printed output is fed through a simple *more* processor. (This is discussed in more detail in the next section.)

In the following two cases, it matters whether the CLI directly prints output or returns and then prints it:

- When the Tcl interpreter executes a list of commands, the CLI only prints the information returned from the last command. It doesn't show information returned by other commands.
- You can only assign the output of a command to a variable if the CLI returns a command's output. You can't assign output that the interpreter prints directly to a variable, or otherwise manipulate it, unless you save it using the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed—unless, of course, the **dload** command appears in the middle of a list of commands; for example:

```
{dload test_program;dstatus}
```

In this example, the CLI doesn't display the ID of the loaded program, since the **dload** command was not the last command.

When information is returned, you can assign it to a variable. For example, the next command assigns the ID of a newly created process to a variable:

```
set pid [dload test_program]
```

Because you can't assign the output of the **help** command to a variable, the following doesn't work:

```
set htext [help]
```

This statement assigns an empty string to **htext** because the **help** command doesn't return text. It just prints it.

To save the output of a command that prints its output, use the **capture** command. For example, the following example writes the **help** command's output into a variable:

```
set htext [capture help]
```



You can only capture the output from commands. You can't capture the informational messages displayed by the CLI that describe process state. If you are using the GUI, TotalView also writes this information to the Log Window. You can display this information by using the Tools > Event Log command.

'more' Processing

When the CLI displays output, it sends data through a simple *more*-like process. This prevents data from scrolling off the screen before you view it.

After you see the **MORE** prompt, press Enter to see the next screen of data. If you type **q** (followed by pressing the Enter key), the CLI discards any data it hasn't yet displayed.

You can control the number of lines displayed between prompts by using the **dset** command to set the **LINES_PER_SCREEN** CLI variable. (For more information, see the *TotalView Reference Guide*.)

Using Command Arguments

The default command arguments for a process are stored in the **ARGS(*num*)** variable, where *num* is the CLI ID for the process. If you don't set the **ARGS(*num*)** variable for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. TotalView sets the **ARGS_DEFAULT** variable when you use the **-a** option when starting the CLI or the GUI.



*The **-a** option tells TotalView to pass everything that follows on the command line to the program.*

For example:

```
totalviewcli -a argument-1, argument-2, ...
```

To set (or clear) the default arguments for a process, you can use the **dset** command to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, the following clears the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained in **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable; for example:

```
dunset ARGS_DEFAULT
```

All commands (except the **drun** command) that can create a process—including the **dgo**, **drerun**, **dcont**, **dstep**, and **dnext** commands—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

Using Namespaces

CLI interactive commands exist in the primary Tcl namespace (::). Some of the TotalView state variables also reside in this namespace. Seldom-used functions and functions that are not primarily used interactively reside in other namespaces. These namespaces also contain most TotalView state variables. (The variables that appear in other namespaces are usually related to TotalView preferences.) TotalView uses the following namespaces:

- | | |
|-----------|---|
| TV:: | Contains commands and variables that you use when creating functions. They can be used interactively, but this is not their primary role. |
| TV::GUI:: | Contains state variables that define and describe properties of the user interface, such as window placement and color. |

If you discover other namespaces beginning with **TV**, you have found a namespace that contains private functions and variables. These objects can (and will) disappear, so don't use them. Also, don't create namespaces that begin with **TV**, since you can cause problems by interfering with built-in functions and variables.

The CLI **dset** command lets you set the value of these variables. You can have the CLI display a list of these variables by specifying the namespace; for example:

```
dset TV::
```

You can use wildcards with this command. For example, **dset TV::au*** displays all variables that begin with "au".

About the CLI Prompt

The appearance of the CLI prompt lets you know that the CLI is ready to accept a command. This prompt lists the current focus, and then displays a greater-than symbol (>) and a blank space. (The *current focus* is the processes and threads to which the next command applies.) For example:

- | | |
|--------------------|--|
| d1.<> | The current focus is the default set for each command, focusing on the first user thread in process 1. |
| g2.3> | The current focus is process 2, thread 3; commands act on the entire group. |

t1.7>	The current focus is thread 7 of process 1.
gW3.>	The current focus is all worker threads in the control group that contains process 3.
p3/3	The current focus is all processes in process 3, group 3.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable; for example:

```
dset PROMPT "Kill this bug! >"
```

Using Built-in and Group Aliases

Many CLI commands have an alias that let you abbreviate the command's name. (An alias is one or more characters that Tcl interprets as a command or command argument.)



*The **alias** command, which is described in the TotalView Reference Guide, lets you create your own aliases.*

For example, the following command tells the CLI to halt the current group:

```
dfocus g dhalt
```

Using an abbreviation is easier. The following command does the same thing:

```
f g h
```

You often type less-used commands in full, but some commands are almost always abbreviated. These commands include **dbreak (b)**, **ddown (d)**, **dfocus (f)**, **dgo (g)**, **dlist (l)**, **dnext (n)**, **dprint (p)**, **dstep (s)**, and **dup (u)**.

The CLI also includes uppercase group versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is **s**; in contrast, **S** is the alias for **dfocus g dstep**. (The first command tells the CLI to step the process. The second steps the control group.)

Group aliases differ from the group-level command that you type interactively, as follows:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

How Parallelism Affects Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

■ Initial process

A preexisting process from the normal run-time environment (that is, created outside TotalView) or one that was created as TotalView loaded the program.

■ Spawned process

A new process created by a process executing under CLI control.

TotalView assigns an integer value to each individual process and thread under its control. This *process/thread identifier* can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; in contrast, thread numbers are only unique while the process exists.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

Thread 1 of process 1
Thread 2 of process 1
Thread 1 of process 2
Thread 2 of process 2

You identify the four threads as follows:

1.1—Thread 1 of process 1
1.2—Thread 2 of process 1
2.1—Thread 1 of process 2
2.2—Thread 2 of process 2

Types of IDs

Multi-threaded, multi-process, and distributed programs contain a variety of IDs. The following types are used in the CLI and the GUI:

System PID This is the process ID and is generally called the PID.

System TID This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no obvious meaning. On other systems, they start at 1 and are incremented by 1 for each thread.

TotalView thread ID

This is usually identical to the system TID. On some systems (such as AIX) where the threads have no obvious meaning), TotalView uses its own IDs.

pthread ID

This is the ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

Debugger PID

This is an ID created by TotalView that lets it identify processes. It is a sequentially numbered value beginning at 1 that is incremented for each new process. If the target process is killed and restarted (that is, you use the `dkill` and `drun` commands), TotalView PID doesn't change. The system PID changes, since the operating system has created a new target process.

Controlling Program Execution

Knowing what's going on and where your program is executing is simple in a serial debugging environment. Your program is either stopped or running. When it is running, an event such as arriving at a breakpoint can occur. This event tells TotalView to stop the program. Sometime later, you tell the serial program to continue executing. Multi-process and multi-threaded programs are more complicated. Each thread and each process has its own execution state. When a thread (or set of threads) triggers a breakpoint, TotalView must decide what it should do about other threads and processes because it may need to stop some and let others continue to run.

Advancing Program Execution

Debugging begins by entering a `dload` or `dattach` command. If you use the `dload` command, you must use the `drun` (or perhaps `drerun` if there's a starter program) command to start the program executing. These three commands work at the process level and you can't use them to start individual threads. (This is also true for the `dkill` command.)

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a P/T set. (P/T sets are discussed in Chapter 2, "About Threads, Processes, and Groups," on page 15 and Chapter 13, "Using Groups, Processes, and Threads," on page 251.) Because the set doesn't have to include all processes and threads, you can cause some processes to be executed while holding others back. You can also advance program execution by increments, *stepping* the program forward, and you can define the size of the increment. For example, `dnext 3` executes the next three statements, and then pauses what you've been stepping.

Typically, debugging a program means that you have the program run, and then you stop it and examine its state. In this sense, a debugger can be thought of as tool that lets you alter a program's state in a controlled way. And debugging is the process of stopping a process to examine its state. However, the term *stop* has a slightly different meaning in a multi-process, multi-threaded program; in these programs, *stopping* means that the CLI holds one or more threads at a location until you enter a command that tells them to start executing again.

For more information, see Chapter 12, "Debugging Programs," on page 223.

Using Action Points

Action points tell the CLI to stop a program's execution. You can specify the following types of action points:

- A *breakpoint* (see **dbreak** in the *TotalView Reference Guide*) stops the process when the program reaches a location in the source code.
- A *watchpoint* (see **dwatch** in the *TotalView Reference Guide*) stops the process when the value of a variable is changed.
- A *barrier point* (see **dbarrier** in the *TotalView Reference Guide*), as its name suggests, effectively prevents processes from proceeding beyond a point until all other related processes arrive. This gives you a method for synchronizing the activities of processes. (You can only set a barrier POINT on processes; you can't set them on individual threads.)
- An *eval point* (see **dbreak** in the *TotalView Reference Guide*) lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. eval points typically do not stop the process; instead, they perform an action. In most cases, an eval point stops the process when some condition that you specify is met.

For extensive information on action points, see "Setting Action Points" on page 349.



Each action point is associated with an *action point identifier*. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1; the second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and the GUI only let you assign one action point to a source code line, but you can make this action point as complex as you need it to be.

Seeing the CLI at Work



The CLI is a command-line debugger that is completely integrated with TotalView. You can use it and never use the TotalView GUI, or you can use it and the GUI simultaneously. Because the CLI is embedded in a Tcl interpreter, you can also create debugging functions that exactly meet your needs. When you do this, you can use these functions in the same way that you use TotalView's built-in CLI commands.

This chapter contains macros that show how the CLI programmatically interacts with your program and with TotalView. Reading examples without bothering too much with details gives you an appreciation for what the CLI can do and how you can use it. With a basic knowledge of Tcl, you can make full use of all CLI features.

In each macro in this chapter, all Tcl commands that are unique to the CLI are displayed in bold. These macros perform the following tasks:

- “*Setting the CLI EXECUTABLE_PATH Variable*” on page 214
- “*Initializing an Array Slice*” on page 215
- “*Printing an Array Slice*” on page 215
- “*Writing an Array Variable to a File*” on page 217
- “*Automatically Setting Breakpoints*” on page 218

Setting the CLI EXECUTABLE_PATH Variable

The following macro recursively descends through all directories, starting at a location that you enter. (This is indicated by the *root* argument.) The macro ignores directories named in the *filter* argument. The result is set as the value of the CLI EXECUTABLE_PATH state variable.

```
# Usage:
#
# rpath [root] [filter]
#
# If root is not specified, start at the current
# directory. filter is a regular expression that removes
# unwanted entries. If it is not specified, the macro
# automatically filters out CVS/RCS/SCCS directories.
#
# The search path is set to the result.

proc rpath {{root ".} {filter "/(CVS|RCS|SCCS) (/|$)"} {

    # Invoke the UNIX find command to recursively obtain
    # a list of all directory names below "root".
    set find [split [exec find $root -type d -print] \n]

    set npath ""

    # Filter out unwanted directories.
    foreach path $find {
        if {! [regexp $filter $path]} {
            append npath ":"}
            append npath $path
        }
    }

    # Tell TotalView to use it.
    dset EXECUTABLE_PATH $npath
}
```

In this macro, the last statement sets the EXECUTABLE_PATH state variable. This is the only statement that is unique to the CLI. All other statements are standard Tcl.

The **dset** command, like most interactive CLI commands, begins with the letter **d**. (The **dset** command is only used in assigning values to CLI state variables. In contrast, values are assigned to Tcl variables by using the standard Tcl **set** command.)

Initializing an Array Slice

The following macro initializes an array slice to a constant value:

```
array_set (var lower_bound upper_bound val) {
    for {set i $lower_bound} {$i <= $upper_bound} {incr i} {
        dassign $var\($i) $val
    }
}
```

The CLI **dassign** command assigns a value to a variable. In this case, it is setting the value of an array element. Use this function as follows:

```
d1.<> dprint list3
list3 = {
    (1) = 1 (0x00000001)
    (2) = 2 (0x00000001)
    (3) = 3 (0x00000001)
}
d1.<> array_set list 2 3 99
d1.<> dprint list3
list3 = {
    (1) = 1 (0x00000001)
    (2) = 99 (0x00000063)
    (3) = 99 (0x00000063)
}
```

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like others shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslice {anArray i1 i2 j1 j2 {i3 1} {j3 1} \
               {width 20}} {
    for {set i $i1} {$i <= $i2} {incr i $i3} {
        set row_out ""
        for {set j $j1} {$j <= $j2} {incr j $j3} {
            set ij [capture dprint $anArray\($i,$j\)]
            set ij [string range $ij \
                     [expr [string first "=" $ij] + 1] end]
            set ij [string trimright $ij]
            if {[string first "-" $ij] == 1} {
                set ij [string range $ij 1 end]}
            append ij " "
            append row_out " " \
```

```

        [string range $ij 0 $width] " "
    }
    puts $row_out
}
}

```



The CLI's **dprint** command lets you specify a slice. For example, you can type:
dprint a(1:4,1:4).

After invoking this macro, the CLI prints a two-dimensional slice ($i_1:i_2:i_3$, $j_1:j_2:j_3$) of a Fortran array to a numeric field whose width is specified by the **width** argument. This width doesn't include a leading minus sign (-).

All but one line is standard Tcl. This line uses the **dprint** command to obtain the value of one array element. This element's value is then captured into a variable. The CLI **capture** command allows a value that is normally printed to be sent to a variable. For information on the difference between values being displayed and values being returned, see "About CLI Output" on page 205.

The following shows how this macro is used:

```

d1.<> pf2Dslice a 1 4 1 4
      0.841470956802 0.909297406673 0.141120001673-
      0.756802499294
      0.909297406673-0.756802499294-0.279415488243
      0.989358246326
      0.141120001673-0.279415488243 0.412118494510-
      0.536572933197
      -0.756802499294 0.989358246326-0.536572933197-
      0.287903308868
d1.<> pf2Dslice a 1 4 1 4 1 1 17
      0.841470956802 0.909297406673 0.141120001673-
      0.756802499294
      0.909297406673-0.756802499294-0.279415488243
      0.989358246326
      0.141120001673-0.279415488243 0.412118494510-
      0.536572933197
      -0.756802499294 0.989358246326-0.536572933197-
      0.287903308868
d1.<> pf2Dslice a 1 4 1 4 2 2 10
      0.84147095 0.14112000
      0.14112000 0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
      -0.75680249 0.98935824
      0.98935824-0.28790330
d1.<>

```

Writing an Array Variable to a File

It often occurs that you want to save the value of an array so that you can analyze its results at a later time. The following macro writes array values to a file:

```
proc save_to_file {var fname} {
    set values [capture dprint $var]
    set f [open $fname w]

    puts $f $values
    close $f
}
```

The following example shows how you might use this macro. Using the **exec** command tells the shell's **cat** command to display the file that was just written.

```
d1.<> dprint list3
list3 = {
    (1) = 1 (0x00000001)
    (2) = 2 (0x00000002)
    (3) = 3 (0x00000003)
}
d1.<> save_to_file list3 foo
d1.<> exec cat foo
list3 = {
    (1) = 1 (0x00000001)
    (2) = 2 (0x00000002)
    (3) = 3 (0x00000003)
}
d1.<>
```

Automatically Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems occurs. The following CLI macro parses comments that you can include in a source file and, depending on the comment's text, sets a breakpoint or an eval point.

Following this macro is an excerpt from a program that uses it.

```
# make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions {{filename ""}} {

    if {$filename == ""} {
        puts "You need to specify a filename"
        error "No filename"
    }

    # Open the program's source file and initialize a
    # few variables.
    set fname [set filename]
    set fsource [open $fname r]
    set lineno 0
    set incomment 0

    # Look for "signals" that indicate the type of
    # action point; they are buried in the comments.
    while {[gets $fsource line] !=-1} {
        incr lineno
        set bpline $lineno

        # Look for a one-line eval point. The
        # format is ... /* EVAL: some_text */.
        # The text after EVAL and before the "*/" in
        # the comment is assigned to "code".
        if [regexp "/\\/* EVAL: *(.*)\\/*/" $line all code] {
            dbreak $fname#$bpline -e $code
            continue
        }

        # Look for a multiline eval point.
        if [regexp "/\\/* EVAL: *(.*)" $line all code] {
            # Append lines to "code".
            while {[gets $fsource interiorline] !=-1} {
                incr lineno

                # Tabs will confuse dbreak.
                regsub-all \t $interiorline \
                    " " interiorline
            }
        }
    }
}
```

```

        # If "/*" is found, add the text to "code",
        # then leave the loop. Otherwise, add the
        # text, and continue looping.
        if [regexp "(.*)\\/*/" $interiorline \
            all interiorcode]{
            append code \n $interiorcode
            break
        } else {
            append code \n $interiorline
        }
    }
dbreak $fname\#$bpline -e $code
continue
}
# Look for a breakpoint.
if [regexp "/\\* STOP: .*" $line] {
    dbreak $fname\#$bpline
    continue
}
# Look for a command to be executed by Tcl.
if [regexp "/\\* *CMD: *(.*)\\/*/" $line all cmd] {
    puts "CMD: [set cmd]"
    eval $cmd
}
close $fsource
}

```

The only similarity between this macro and the previous three is that almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command that set eval points and breakpoints.

The following excerpt from a larger program shows how to embed comments in a source file that is read by the **make_actions** macro:

```

...
struct struct_bit_fields_only {
    unsigned f3 : 3;
    unsigned f4 : 4;
    unsigned f5 : 5;
    unsigned f20 : 20;
    unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
...
int main()
{
    struct struct_bit_fields_only *lbfop = &sbfo;
...
    int i;
    int j;
    sbfo.f3 = 3;
    sbfo.f4 = 4;
    sbfo.f5 = 5;
    sbfo.f20 = 20;
    sbfo.f32 = 32;
...

```

```
/* TEST: Check to see if we can access all the
   values */
i=i;      /* STOP: */
i=1;      /* EVAL: if (sbfo.f3 != 3) $stop; */
i=2;      /* EVAL: if (sbfo.f4 != 4) $stop; */
i=3;      /* EVAL: if (sbfo.f5 != 5) $stop; */
...
return 0;
}
```

The `make_actions` macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with `/* STOP`, `/* EVAL`, and `/* CMD`. After parsing the comment, it sets a breakpoint at a *stop* line or an eval point at an *eval* line, or executes a command at a *cmd* line.

Using eval points can be confusing because eval point syntax differs from that of Tcl. In this example, the `$stop` function is built into the CLI. Stated differently, you can end up with Tcl code that also contains C, C++, Fortran, and TotalView functions, variables, and statements. Fortunately, you only use this kind of mixture in a few places and you'll know what you're doing.

Part V: Debugging



The chapters in this part of the *TotalView Users Guide* describe how you actually go about debugging your programs. The preceding chapters describe, for the most part, what you need to do before you get started with TotalView. In contrast, the chapters in this part are what TotalView is really about.

Chapter 12: Debugging Programs

Read this chapter to help you find your way around your program. This chapter describes ways to step your program's execution, and how to halt, terminate, and restart your program.

Chapter 13: Using Groups, Processes, and Threads

The stepping information in Chapter 10 describes the commands and the different types of stepping. In a multi-process, multi-threaded program, you may need to finely control what is executing. This chapter tells you how to do this.

Chapter 14: Examining and Changing Data

As your program executes, you will want to examine what the value stored in a variable is. This chapter tells you how.

Chapter 15: Examining Arrays

Displaying information in arrays presents special problems. This chapter tells how TotalView solves these problems.

Chapter 16: Setting Action Points

Action points let you control how your programs execute and what happens when your program reaches statements that you define as important. Action points also let you monitor changes to a variable's value.

Chapter 17: Evaluating Expressions

Many TotalView operations such as displaying variables are actually operating upon expressions. Here's where you'll find details of what TotalView does. This information is not just for advanced users.

Debugging Programs



This chapter explains how to perform basic debugging tasks with TotalView.

This chapter contains the following sections:

- “*Searching and Looking For Program Elements*” on page 223
- “*Editing Source Text*” on page 227
- “*Manipulating Processes and Threads*” on page 228
- “*Using Stepping Commands*” on page 239
- “*Executing to a Selected Line*” on page 241
- “*Executing Out of a Function*” on page 241
- “*Continuing with a Specific Signal*” on page 242
- “*Killing (Deleting) Programs*” on page 243
- “*Restarting Programs*” on page 243
- “*Checkpointing*” on page 244
- “*Fine-Tuning Shared Library Use*” on page 245
- “*Setting the Program Counter*” on page 248
- “*Interpreting the Status and Control Registers*” on page 250

CHAPTER
12

Searching and Looking For Program Elements

TotalView provides several ways for you to navigate and find information in your source file.

Topics in this section are:

- “*Searching for Text*” on page 224
- “*Looking for Functions and Variables*” on page 224
- “*Finding the Source Code for Functions*” on page 225

- "Finding the Source Code for Files" on page 227
- "Resetting the Stack Frame" on page 227

Searching for Text

You can search for text strings in most windows using the **Edit > Find** command, which displays the following dialog box.

Figure 140: **Edit > Find** Dialog Box



Controls in this dialog box let you:

- Perform case-sensitive searches.
- Continue searching from the beginning of the file if the string isn't found in the region beginning at the currently selected line and ending at the last line of the file.
- Keep the dialog box displayed.
- Tell TotalView to search towards the bottom of the file (**Down**) or the top (**Up**).

After you have found a string, you can find another instance of it by using the **Edit > Find Again** command.

If you searched for the same string previously, you can select it from the pulldown list on the right side of the **Find** text box.

Looking for Functions and Variables

Having TotalView locate a variable or a function is usually easier than scrolling through your sources to look for it. Do this with the **View > Lookup Function** and **View > Lookup Variable** commands. (See Figure 141 on page 225.)

CLI: `dprint variable`

If TotalView doesn't find the name and it can find something similar, it displays a dialog box that contains the names of functions that might match. (See Figure 142 on page 225.)

If the one you want is listed, click on its name and then choose **OK** to display it in the Source Pane.

Figure 141: View > Lookup Variable Dialog Box

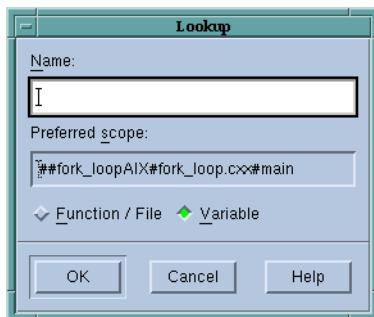
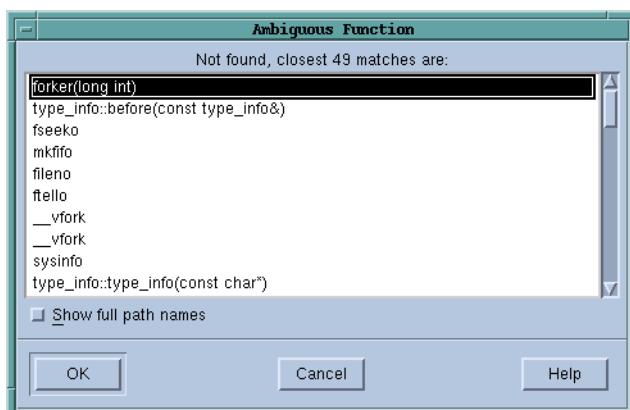


Figure 142: Ambiguous Function Dialog Box



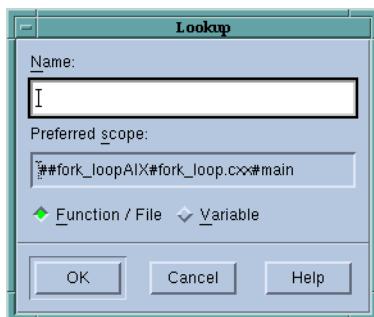
Finding the Source Code for Functions

Use the **File > Open Source** command to search for a function's declaration.

CLI: `dlist function-name`

This command tells TotalView to display the following dialog box:

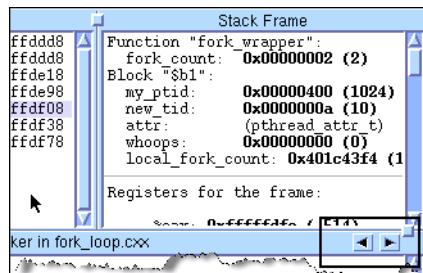
Figure 143: View > Lookup Function Dialog Box



After locating your function, TotalView displays it in the Source Pane. If you didn't compile the function using the `-g` command-line option, TotalView displays disassembled machine code.

When you want to return to the previous contents of the Source Pane, use the Backward button located in the upper-right corner of the Source Pane and just below the Stack Frame Pane. In Figure 144, a rectangle surrounds this button.

Figure 144: Undive/Dive Controls



You can also use the **View > Reset** command to discard the dive stack so that the Source Pane is displaying the PC it displayed when you last stopped execution.

Another method of locating a function's source code is to dive into a source statement in the Source Pane that shows the function being called. After diving, you see the source.

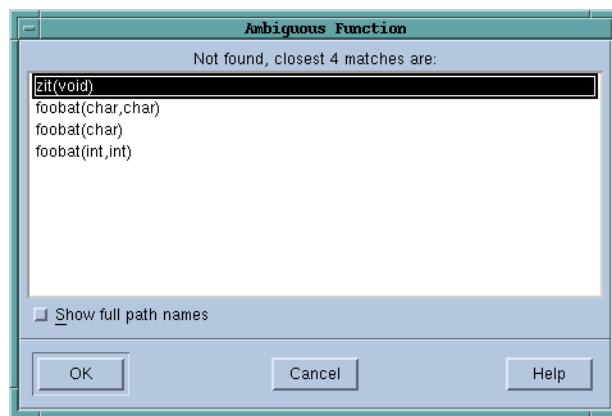
Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you might have specified the name of:

- A static function, and your program contains different versions of it.
- A member function in a C++ program, and multiple classes have a member function with that name.
- An overloaded function or a template function.

The following figure shows the dialog box that TotalView displays when it encounters an ambiguous function name. You can resolve the ambiguity by clicking the function name.

Figure 145: Ambiguous Function Dialog Box

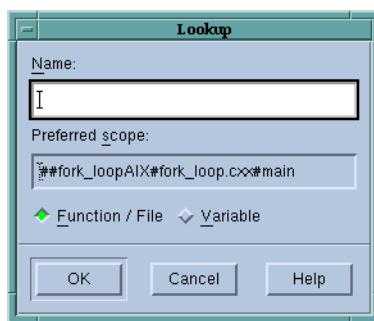


If the name being displayed isn't enough to identify which name you need to select, select the **Show full path names** check box to display additional information.

Finding the Source Code for Files

You can display a file's source code by selecting the **View > Lookup Function** command and entering the file name in the dialog box shown in the following figure.

Figure 146: View > Lookup Function Dialog Box



If a header file contains source lines that produce executable code, you can display the file's code by typing the file name here.

Resetting the Stack Frame

After moving around your source code to look at what's happening in different places, you can return to the executing line of code for the current stack frame by selecting the **View > Reset** command. This command places the PC arrow on the screen.

This command is also useful when you want to undo the effect of scrolling, or when you move to different locations using, for example, the **View > Lookup Function** command.

If the program hasn't started running, the **View > Reset** command displays the first executable line in your main program. This is useful when you are looking at your source code and want to get back to the first statement that your program executes.

Editing Source Text

Use the **File > Edit Source** command to examine the current routine in a text editor. TotalView uses an *editor launch string* to determine how to start your edi-

tor. TotalView expands this string into a command that TotalView sends to the `sh` shell.

The default editor is `vi`. However, TotalView uses the editor named in an `EDITOR` environment variable, or the editor you name in the Source Code Editor field of the **File > Preferences** Launch Strings Page. The online Help for this page contains information on setting this preference.

Manipulating Processes and Threads

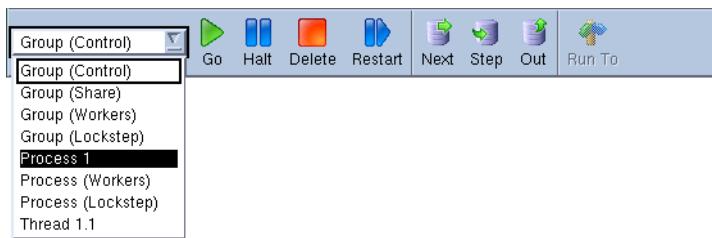
Topics discussed in this section are:

- “*Using the Toolbar to Select a Target*” on page 228
- “*Stopping Processes and Threads*” on page 229
- “*Updating Process Information*” on page 231
- “*Holding and Releasing Processes and Threads*” on page 231
- “*Using Barrier Points*” on page 233
- “*Holding Problems*” on page 234
- “*Examining Groups*” on page 235
- “*Placing Processes in Groups*” on page 236
- “*Placing Processes in Groups*” on page 236
- “*Starting Processes and Threads*” on page 236
- “*Creating a Process Without Starting It*” on page 237
- “*Creating a Process by Single-Stepping*” on page 237
- “*Stepping and Setting Breakpoints*” on page 237

Using the Toolbar to Select a Target

The Process Window toolbar has three sets of buttons. The first set is a single pulldown list. It defines the *focus* of the command selected in the second set of the toolbar. The third set changes the process and thread being displayed. The following figure shows this toolbar.

Figure 147: The Toolbar



When you are doing something to a multi-process, multi-threaded program, TotalView needs to know which processes and threads it should act on. In the CLI, you specify this target using the `dfocus` command. When using the

GUI, you specify the focus using the scope pulldown. For example, if you select a thread, and then select the **Step** button, TotalView steps the current thread. In contrast, if you select **Process (Workers)** and then select the **Go** button, TotalView tells all the processes that are in the same workers group as the current thread to start executing. (This thread is called the *thread of interest*.)



Chapter 13, "Using Groups, Processes, and Threads," on page 251 describes how TotalView manages processes and threads. While TotalView gives you the ability to control the precision your application requires, most applications do not need this level of interaction. In almost all cases, using the controls in the toolbar gives you all the control you need.

Stopping Processes and Threads

To stop a group, process, or thread, select a **Halt** command from the **Group**, **Process**, or **Thread** pulldown menu on the toolbar.

CLI: `dhalt`

Halts a group, process, or thread. Setting the focus changes the scope.

The three **Halt** commands differ in the scope of what they halt. In all cases, TotalView uses the current thread, which is called the thread of interest or TOI, to determine what else it will halt. For example, selecting **Process > Halt** tells TotalView to determine the process in which the TOI is running. It then halts this process. Similarly, if you select **Group > Halt**, TotalView determines what processes are in the group in which the current thread participates in. It then stops all of these processes.



For more information on the Thread of Interest, see "Defining the GOI, POI, and TOI" on page 251.

When you select the **Halt** button in the toolbar instead of the commands in the menubar, TotalView decides what it should stop based on what is set in the two toolbar pulldown lists.

After entering a **Halt** command, TotalView updates any windows that can be updated. When you restart the process, execution continues from the point where TotalView stopped the process.

Using the Processes/Ranks Tab

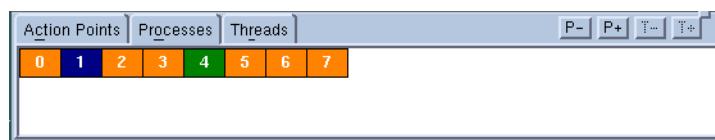
The Processes Tab, which is called a Ranks Tab if you are running an MPI program, contains a grid. Each block in the grid represents one process.

The color that TotalView uses to display a process indicates the process's state, as follows:

Color	Meaning
Blue	Stopped; usually due to another process or thread hitting a breakpoint.
Orange	At breakpoint.
Green	All threads in the process are running or can run.
Red	The Error state. Signals such as SIGSEGV , SIGBUS , and SIGFPE can indicate an error in your program.
Gray	The process has not begun running.

The following figure shows a tab with processes in three different states (the differences are hard to see in the printed manual):

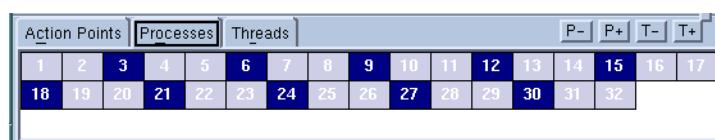
Figure 148: The Processes Tab



If you select a group by using the Process Window's group selector pull-down (see "Using the Toolbar to Select a Target" on page 228 for information), TotalView dims the blocks for processes not in the group. (See Figure 149.)

CLI: dptsets

Figure 149: The Processes Tab: Showing Group Selection



If you click on a block, the context within the Process Window changes to the first thread in that process.

CLI: dfocus

Clicking on the **P+** and **P-** buttons in the tab bar changes the process being displayed within the Process Window.

Using the Threads Tab

The Threads Tab displays information about the state of your threads. Clicking on a thread tells TotalView to shift the focus within the Process Window to that thread. (See Figure 150 on page 231.)

Clicking on the **T+** and **T-** buttons in the tab bar changes the thread being displayed within the Process Window.

Figure 150: The Threads Tab

Action Points	Processes	Threads	P-	P+	T-	T+
1.1	T	in __select				
1.2	T	in __select				
1.3	T	in __select				
1.4	T	in __select				
1.-1	T	in __poll				

Updating Process Information

Normally, TotalView only updates information when the thread being executed stops executing. You can force TotalView to update a window by using the **Window > Update** command. You need to use this command if you want to see what a variable's value is while your program is executing.



When you use this command, TotalView momentarily stops execution so that it can obtain the information that it needs. It then restarts the thread.

Holding and Releasing Processes and Threads

Many times when you are running a multi-process or multi-threaded program, you want to synchronize execution to the same place. You can do this manually using a *hold* command, or automatically by setting a barrier point.

When a process or a thread is *held*, any command that it receives that tells it to execute are ignored. For example, assume that you place a hold on a process in a control group that contains three processes. After you select **Group > Go**, two of the three processes resume executing. The held process ignores the **Go** command.

At a later time, you will want to run what is being held. Do this using a **Release** command. When you release a process or a thread, you are telling it that it can run. But you still need to tell it to execute, which means that it is waiting to receive an execution command, such as **Go**, **Out**, or **Step**.

Manually holding and releasing processes and threads is useful in the following instances:

- When you need to run a subset of the processes and threads. You can manually hold all but the ones you want to run.
- When a process or thread is held at a barrier point and you want to run it without first running all the other processes or threads in the group to that barrier. In this case, you release the process or the thread manually, and then run it.

TotalView can also hold a process or thread if it stops at a barrier breakpoint. You can manually release a process or thread being held at a barrier breakpoint. See "Setting Barrier Points" on page 362 for more information on manually holding and releasing barrier breakpoint.

When TotalView is holding a process, the Root and Process Windows display a held indicator, which is the uppercase letter **H**. When TotalView is holding a thread, it displays a lowercase **h**.

You can hold or release a thread, process, or group of processes in one of the following ways:

- You can hold a group of processes using the **Group > Hold** command.
- You can release a group of processes using the **Group > Release** command.
- You can toggle the hold/release state of a process by selecting and clearing the **Process > Hold** command.
- You can toggle the hold/release state of a thread by selecting and clearing the **Thread > Hold** command.

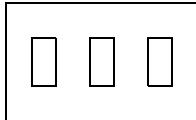
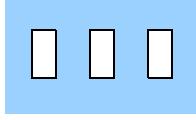
CLI: dhold and dunhold
Setting the focus changes the scope.

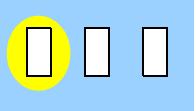
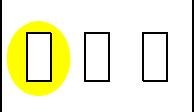
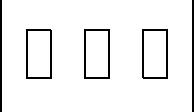
If a process or a thread is running when you use a hold or release command, TotalView stops the process or thread, and then holds it. TotalView lets you hold and release processes independently from threads.

The Process pulldown menu contains a **Hold Threads** and a **Release Threads** command. Although they appear to do the same thing, they are used in slightly different ways. When you use the **Hold Threads** commands on a multi-threaded process, you place a hold on all threads. This is seldom what you want as you really do want something to run. After selecting this command, go to the thread that you want to run and then clear the **Thread > Hold** command so that TotalView lets it run. This may appear awkward, but it is actually an easy way to select one or two threads when your program has a lot of threads. You can verify that you're doing the right thing by looking at the thread's status in the Root Window.

CLI: dhold –thread
dhold –process
dunhold –thread

The following set of drawings presents examples of using hold commands:

Held/Release State	What Can Be Run Using Process > Go
	This figure shows a process with three threads. Before you do anything, all threads in the process can be run.
	Select the Process > Hold toggle. The blue indicates that you held the process. (Or, at least its in blue if you are viewing this online.) Nothing runs when you select Process > Go .

Held/Release State	What Can Be Run Using Process > Go
	<p>Go to the Threads menu. The button next to the Hold command isn't selected. This is because the <i>thread hold</i> state is independent from the <i>process hold</i> state.</p> <p>Select it. The circle indicates that thread 1 is held. At this time, there are two different holds on thread 1. One is at the process level; the other is at thread level.</p> <p>Nothing will run when you select Process > Go.</p>
	<p>Select the Process > Hold command.</p> <p>Select Process > Go. The second and third threads run.</p>
	<p>Select Process > Release Threads. This releases the hold placed on the first thread by the Thread > Hold command.</p> <p>After you select Process > Go, all threads run.</p>

Using Barrier Points

Because threads and processes are often executing different instructions, keeping threads and processes together is difficult. The best strategy is to define places where the program can run freely and places where you need control. This is where barrier points come in.

To keep things simple, this section only discusses multi-process programs. You can do the same types of operations when debugging multi-threaded programs.

Why breakpoints don't work (part 1)

If you set a breakpoint that stops all processes when it is hit and you let your processes run using the **Group > Go** command, you can get lucky and all of your threads will be at the breakpoint. What's more likely is that some processes won't have reached the breakpoint and TotalView will stop them wherever they happen to be. To get your processes synchronized, you need to find out which ones didn't get there and then individually get them to the breakpoint using the **Process > Go** command. You can't use the **Group > Go** command since this also runs the processes stopped at the breakpoint.

Why breakpoints don't work (part 2)

If you set the breakpoint's property so that only the process hitting the breakpoint stops, you have a better chance of getting processes there. However, you should not have other breakpoints between where the program is currently at and this breakpoint. If processes hit these breakpoints, you are once again left running individual processes to the breakpoint.

Why single stepping doesn't work

Single stepping is just too tedious if you have a long way to go to get to your synchronization point, and stepping just won't work if your processes don't execute exactly the same code.

Why Barrier points work

If you use a barrier point, you can use the **Group > Go** command as many times as it takes to get all of your processes to the barrier, and you won't have to worry about a process running past the barrier.

The Root Window shows you which processes have hit the barrier. It marks all held processes with the letter **H** (meaning hold) in the column immediately to the right of the state codes. When all processes reach the barrier, TotalView removes all holds.

Holding Problems

Creating a barrier point tells TotalView to *hold* a process when it reaches the barrier. Other processes that can reach the barrier but aren't yet at it continue executing. One-by-one, processes reach the barrier and, when they do, TotalView holds them.

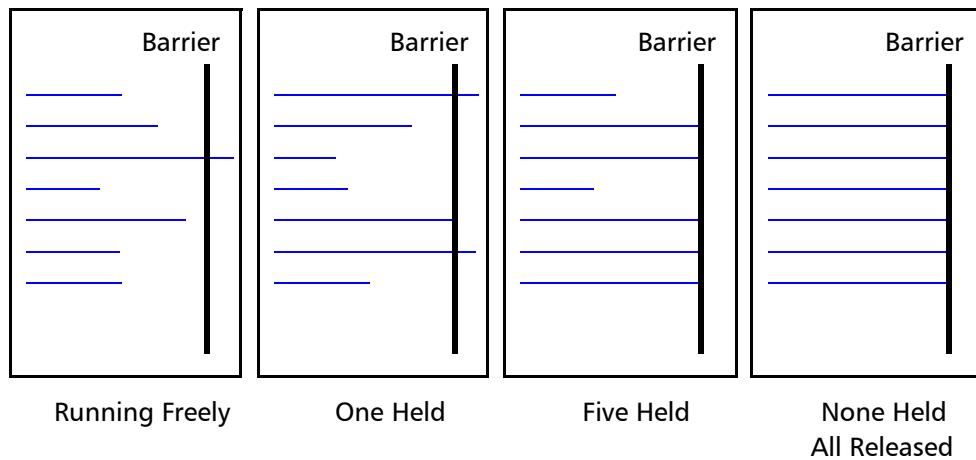
When a process is *held*, it ignores commands that tell it to execute. This means, for example, that you can't tell it to go or to step. If, for some reason, you want the process to execute, you can manually release it using either the **Group > Release** or **Process > Release Threads** command.

When all processes that share a barrier reach it, TotalView changes their state from *held* to *released*, which means that they no longer ignore a command that tells it to begin executing.

The following figure shows seven processes that are sharing the same barrier. (Processes that aren't affected by the barrier aren't shown.)

- First block: All seven processes are running freely.
- Second block: One process hits the barrier and is held. Six processes are executing.
- Third block: Five of the processes have now hit the barrier and are being held. Two are executing.
- Fourth block: All processes have hit the barrier. Because TotalView isn't waiting for anything else to reach the barrier, it changes the processes' states to *released*. Although the processes are released, none are executing. (See Figure 151.)

Figure 151: Running To Barriers



Examining Groups

When you debug a multi-process program, TotalView adds processes to both a control and a share group as the process starts. These groups are not related to either UNIX process groups or PVM groups. (See Chapter 2, "About Threads, Processes, and Groups," on page 15 for information on groups.)

Because a program can have more than one control group and more than one share group, TotalView decides where to place a process based on the type of system call—which can either be `fork()` or `execve()`—that created or changed the process. The two types of process groups are:

Control Group	The parent process and all related processes. A control group includes children that a process forks (processes that share the same source code as the parent). It also includes forked children that subsequently call a function such as <code>execve()</code> . That is, a control group can contain processes that don't share the same source code as the parent.
	Control groups also include processes created in parallel programming disciplines like MPI.
Share Group	The set of processes in a control group that shares the same source code. Members of the same share group share action points.

See Chapter 13, "Using Groups, Processes, and Threads," on page 251 for a complete discussion of groups.



TotalView automatically creates share groups when your processes fork children that call the `execve()` function, or when your program creates processes that use the same code as some parallel programming models such as MPI do.

TotalView names processes according to the name of the source program, using the following naming rules:

- TotalView names the parent process after the source program.
- The name for forked child processes differs from the parent in that TotalView appends a numeric suffix (`.n`). If you're running an MPI program, the numeric suffix is the process's rank in `COMM_WORLD`.
- If a child process calls the `execve()` function after it is forked, TotalView places a new executable name in angle brackets (`<>`).

In the following figure, assume that the **generate** process doesn't fork any children, and that the **filter** process forks two child processes. Later, the first child forks another child, and then calls the `execve()` function to execute the **expr** program. In this figure, the middle column shows the names that TotalView uses. (See Figure 152 on page 236.)

Figure 152: Control and Share Groups Example

	Process Groups	Process Names	Relationship
Control Group 1	<pre> graph TD CG1[Control Group 1] --- SG1[Share Group 1] CG1 --- SG2[Share Group 2] SG1 --- filter1[filter] SG1 --- filter1_1[filter.1] SG1 --- filter1_2[filter.2] SG2 --- filter2[filter<expr>.1.1] </pre>	<pre> graph TD P1[filter] --- P1_1[filter.1] P1 --- P1_2[filter.2] P1 --- P1_3[filter<expr>.1.1] </pre>	parent process #1 child process #1 child process #2 grandchild process #1
Control Group 2	<pre> graph TD CG2[Control Group 2] --- SG3[Share Group 3] SG3 --- generate[generate] </pre>	<pre> graph TD P2[generate] </pre>	parent process #2

Placing Processes in Groups

TotalView uses your executable's name to determine the share group that the program belongs to. If the path names are identical, TotalView assumes that they are the same program. If the path names differ, TotalView assumes that they are different, even if the file name in the path name is the same, and places them in different share groups.

Starting Processes and Threads

To start a process, select a **Go** command from the **Group**, **Process**, or **Thread** pulldown menus.

After you select a **Go** command, TotalView decides what to run based on the current thread. It uses this thread, which is called the Thread of Interest (TOI), to decide what other threads it should run. For example, if you select **Group > Go**, TotalView continues all threads in the current group that are associated with this thread.

CLI: `dfocus g dgo`
Abbreviation: G

The commands you will use most often are **Group > Go** and **Process > Go**. The **Group > Go** command creates and starts the current process and all other processes in the multi-process program. There are some limitations, however. TotalView only resumes a process if the following are true:

- The process is not being held.
- The process is already exists and is stopped.
- The process is at a breakpoint.

Using a **Group > Go** command on a process that's already running starts the other members of the process's control group.

CLI: `dgo`

If the process hasn't yet been created, **Go** commands creates and starts it. *Starting* a process means that all threads in the process resume executing unless you are individually holding a thread.



TotalView disables the **Thread > Go** command if asynchronous thread control is not available. If you enter a thread-level command in the CLI when asynchronous thread controls aren't available, TotalView tries to perform an equivalent action. For example, it continues a process instead of a thread.

For a single-process program, the **Process > Go** and **Group > Go** commands are equivalent. For a single-threaded process, the **Process > Go** and **Thread > Go** commands are equivalent.

Creating a Process Without Starting It

The **Process > Create** command creates a process and stops it before the first statement in your program executes. If you link a program with shared libraries, TotalView allows the dynamic loader to map into these libraries. Creating a process without starting it is useful when you need to do the following:

- Create watchpoints or change the values of global variables after a process is created, but before it runs.
- Debug C++ static constructor code.

CLI: `dstepi`

While there is no CLI equivalent to the **Process > Create** command, executing the `dstepi` command produces the same effect.

Creating a Process by Single-Stepping

The TotalView single-stepping commands lets you create a process and run it to the beginning of your programs. The single-stepping commands available from the **Process** menu are as shown in the following table:

GUI command	CLI command	Creates the process and ...
Process > Step	<code>dfocus p dstep</code>	Runs it to the first line of the <code>main()</code> routine.
Process > Next	<code>dfocus p dnnext</code>	Runs it to the first line of the <code>main()</code> routine; this is the same as Process > Step .
Process > Step Instruction	<code>dfocus p dstepi</code>	Stops it before any of your program executes.
Process > Next Instruction	<code>dfocus p dnnexti</code>	Runs it to the first line of the <code>main()</code> routine. This is the same as Process > Step .

If a group-level or thread-level stepping command creates a process, the behavior is the same as if it were a process-level command.



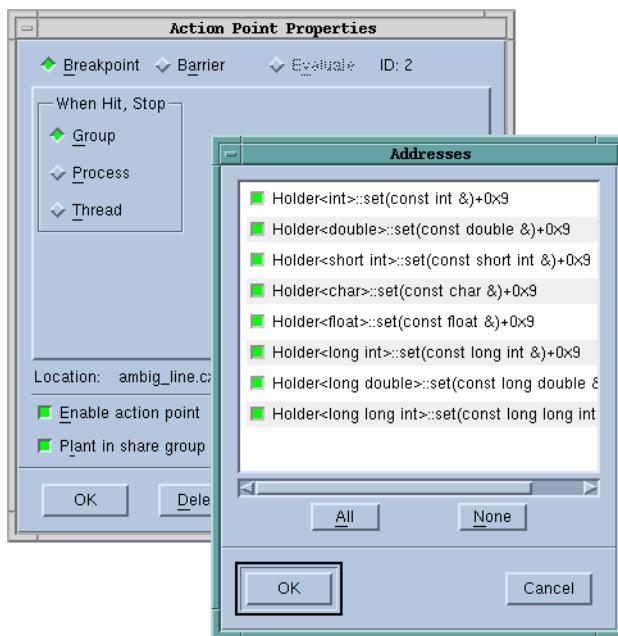
Chapter 13, "Using Groups, Processes, and Threads," on page 251 contains a detailed discussion of setting the focus for stepping commands.

Stepping and Setting Breakpoints

Several of the single-stepping commands require that you select a source line or machine instruction in the Source Pane. To select a source line, place the cursor over the line and click your left mouse button. If you select

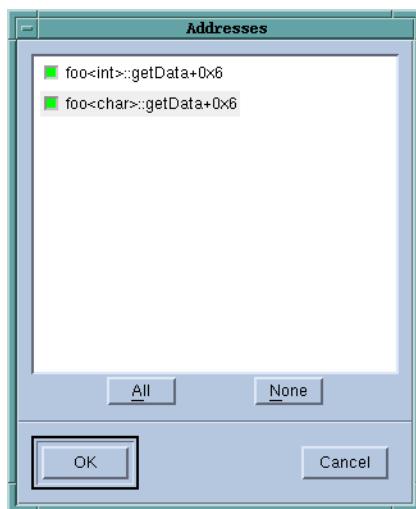
a source line that has more than one instantiation, TotalView will try to do the right thing. For example, if you select a line within a template so you can set a breakpoint on it, you'll actually set a breakpoint on all of the template's instantiations. If this isn't what you want, select the **Location** button in the **Action Point > Properties** Dialog Box to change which instantiations will have a breakpoint. (See "Setting Breakpoints and Barriers" on page 351.) (See Figure 153.)

Figure 153: Action Point and Addresses Dialog Boxes



Similarly, if TotalView cannot figure out which instantiation to set a breakpoint at, it will display its **Address** Dialog Box. (See Figure 154.)

Figure 154: Ambiguous Address Dialog Box



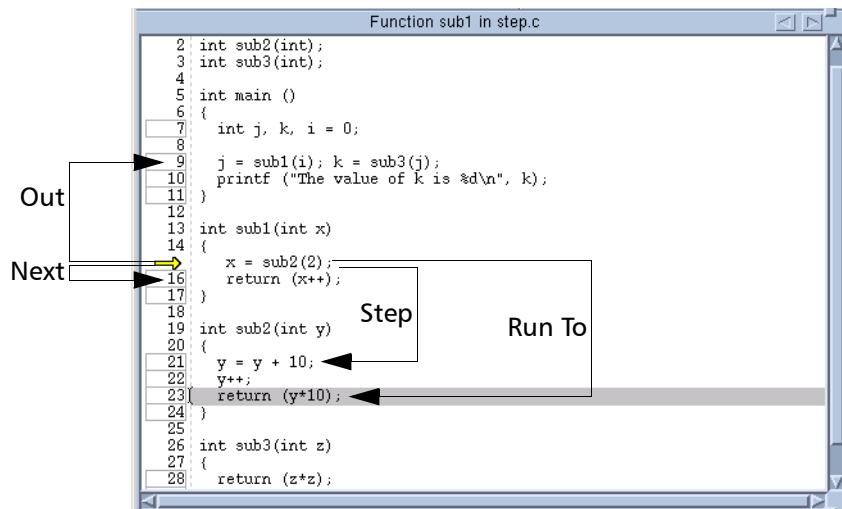
Using Stepping Commands

While different programs have different requirements, the most common stepping mode is to set group focus to **Control** and the target to **Process** or **Group**. You can now select stepping commands from the **Process** or **Group** menus or use commands in the toolbar.

```
CLI: dfocus g
      dfocus p
```

The following figure illustrates stepping commands.

Figure 155: Stepping Illustrated



The arrow indicates that the PC is at line 15. The four stepping commands do the following:

- **Next** executes line 15. After stepping, the PC is at line 16.
- **Step** moves into the **sub2()** function. The PC is at line 21.
- **Run To** executes all lines until the PC reaches the selected line, which is line 23.
- **Out** executes all statements within **sub1()** and exits from the function. The PC is at line 9. If you now execute a Step command, TotalView steps into **sub3()**.

Remember the following things about single-stepping commands:

- To cancel a single-step command, select **Group > Halt** or **Process > Halt**.

```
CLI: dhalt
```

- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you enter a source-line stepping command and the primary thread is executing in a function that has no source-line information, TotalView performs an assembler-level stepping command.
- When TotalView steps through your code, it steps one line at-a-time. This means that if you have more than one statement on a line, a step instruction executes all of the instructions on that line.

Stepping into Function Calls

The stepping commands execute one line in your program. If you are using the CLI, you can use a numeric argument that indicates how many source lines TotalView steps. For example, here's the CLI instruction for stepping three lines:

```
dstep 3
```

If the source line or instruction contains a function call, TotalView steps into it. If TotalView can't find the source code and the function was compiled with **-g**, it displays the function's machine instructions.

You might not realize that your program is calling a function. For example, if you overloaded an operator, you'll step into the code that defines the overloaded operator.



*If the function being stepped into wasn't compiled with the **-g** command-line option, TotalView always steps over the function.*

The GUI has eight **Step** commands and eight **Step Instruction** commands. These commands are located on the **Group**, **Process**, and **Thread** pull-downs. The difference between them is the focus.

```
CLI: dfocus ... dstep  
      dfocus ... dstepl
```

Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call.

The GUI has eight **Next** commands that execute a single source line while stepping over functions, and eight **Next Instruction** commands that execute a single machine instruction while stepping over functions. These commands are on the **Group**, **Process**, and **Thread** menus.

```
CLI: dfocus ... dnnext  
      dfocus ... dnnexti
```

If the PC is in assembler code—this can happen, for example, if you halt your program while it's executing in a library—a **Next** operation executes the next instruction. If you want to execute out of the assembler code so you're back in your code, select the **Out** command. You might need to select **Out** a couple of times until you're back to where you want to be.

Executing to a Selected Line

If you don't need to stop execution every time execution reaches a specific line, you can tell TotalView to run your program to a selected line or machine instruction. After selecting the line on which you want the program to stop, invoke one of the eight **Run To** commands defined within the GUI. These commands are on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... duntil

Executing to a selected line is discussed in greater depth in Chapter 13, "Using Groups, Processes, and Threads," on page 251.

If your program reaches a breakpoint while running to a selected line, TotalView stops at that breakpoint.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane. When you do this, TotalView determines where to stop execution by looking at the following:

- The frame pointer (FP) of the selected stack frame.
- The selected source line or instruction.

CLI: dup and ddown

Executing Out of a Function

You can step your program out of a function by using the **Out** commands. The eight **Out** commands in the GUI are located on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... dout

Continuing with a Specific Signal

If the source line that is the *goal* of the **Out** operation has more than one statement, TotalView will stop execution just after the routine from which it just emerged. For example, suppose that the following is your source line:

```
routine1; routine2;
```

Suppose you step into **routine1**, then use an **Out** command. While the PC arrow in the Source Pane still points to this same source line, the actual PC is just after **routine1**. This means that if you use a step command, you will step into **routine2**.

The PC arrow does not move when the source line only has one statement on it. The internal PC does, of course, change.

You can also return out of several functions at once, by selecting the routine in the Stack Trace Pane that you want to go to, and then selecting an **Out** command.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane to indicate which instance you are running out of.

Continuing with a Specific Signal

Letting your program continue after sending it a signal is useful when your program contains a signal handler. Here's how you tell TotalView to do this:

- 1 Select the Process Window's **Thread > Continuation Signal** command.

Figure 156: Thread > Continuation Signal Dialog Box



- 2 Select the signal to be sent to the thread and then select **OK**.

The continuation signal is set for the thread contained in the current Process Window. If the operating system can deliver multi-threaded signals, you can set a separate continuation signal for each thread. If it can't, this command clears continuation signals set for other threads in the process.

- 3 Continue execution of your program with commands such as **Process > Go, Step, Next, or Detach**.

TotalView continues the threads and sends the specified signals to your process.



*To clear the continuation signal, select **signal 0** from this dialog box.*

You can change the way TotalView handles a signal by setting the `TV::signal_handling_mode` variable in a `.tvdrcc` startup file. For more information, see Chapter 4 of the "TotalView Reference Guide."

Killing (Deleting) Programs

To kill (or delete) all the processes in a control group, use the **Group > Kill** command. The next time you start the program, for example, by using the **Process > Go** command, TotalView creates and starts a fresh master process.

CLI: `dfocus g dkill`

Restarting Programs

You can use the **Group > Restart** command to restart a program that is running or one that is stopped but hasn't exited.

CLI: `drerun`

If the process is part of a multi-process program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

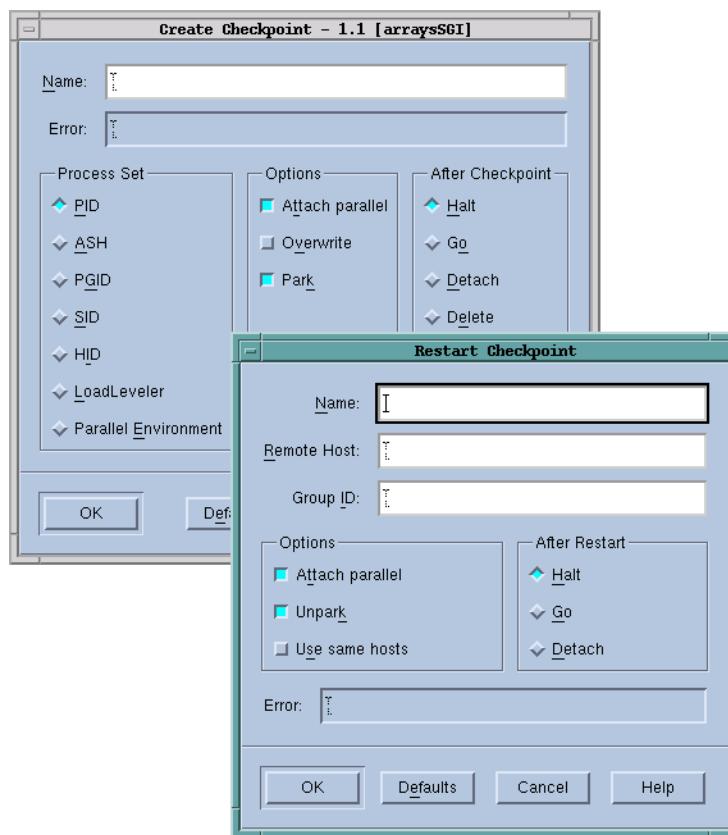
The **Group > Restart** command is equivalent to the **Group > Kill** command followed by the **Process > Go** command.

Checkpointing

On SGI IRIX and IBM RS/6000 platforms, you can save the state of selected processes and then use this saved information to restart the processes from the position where they were saved. For more information, see the Process Window Tools > Create Checkpoint and Tools > Restart Checkpoint commands in the online Help. (See Figure 157.)

CLI: dcheckpoint
drestart

Figure 157: Create Checkpoint and Restart Checkpoint Dialog Boxes



Fine-Tuning Shared Library Use

When TotalView encounters a reference to a shared library, it normally reads all of that library's symbols. In some cases, you might need to explicitly read in this library's information before TotalView automatically reads it.

On the other hand, you may not want TotalView to read and process a library's loader and debugging symbols. In most cases, reading these symbols occurs quickly. However, if your program uses large libraries, you can increase performance by telling TotalView not to read these symbols.

For more information, see "Preloading Shared Libraries" on page 245 and "Controlling Which Symbols TotalView Reads" on page 246.

Preloading Shared Libraries

As your program executes, it can call the `dlopen()` function to access code contained in shared libraries. In some cases, you might need to do something from within TotalView that requires you to preload library information. For example, you might need to refer to one of a library's functions in an eval point or in a **Tools > Evaluate** command. If you use the function's name before TotalView reads the dynamic library, TotalView displays an error message.

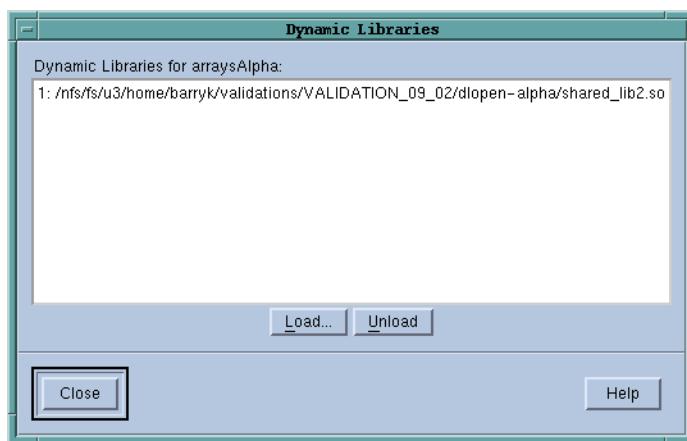
Use the **Tools > Debugger Loaded Libraries** command to tell the debugger to open a library.

CLI: `ddlopen`

This CLI command gives you additional ways to control how a library's symbols are used.

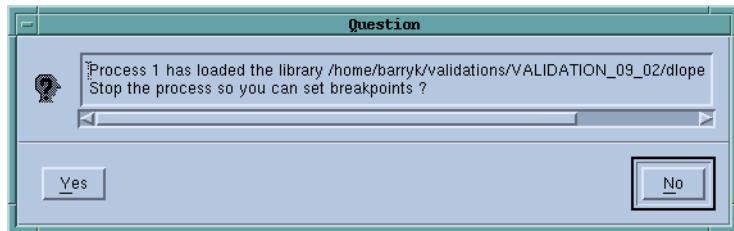
After selecting this command, TotalView displays the following dialog box:

Figure 158: Tools > Debugger Loaded Libraries Dialog Box



Selecting the **Load** button tells TotalView to display a file explorer dialog box that lets you navigate through your computer's file system to locate the library. After selecting a library, TotalView reads it and displays a question box that lets you stop execution to set a breakpoint:

Figure 159: Stopping to Set a Breakpoint Question Box



TotalView might not read in information symbol and debugging information when you use this command. See "Controlling Which Symbols TotalView Reads" on page 246 for more information.

Controlling Which Symbols TotalView Reads

When debugging large programs with large libraries, reading and parsing symbols can impact performance. This section describes how you can minimize the impact that reading this information has on your debugging session.



Using the preference settings and variables described in this section always slow down performance. However, for most programs, even large ones, the difference is often inconsequential. If, however, you are debugging a very large program with large libraries, significant performance improvements can occur.

A shared library contains, among other things, loader and debugging symbols. Typically, loader symbols are read quite quickly. Debugging symbols can require considerable processing. The default behavior is to read all symbols. You can change this behavior by telling TotalView to only read in loader symbols or even that it should not read in any symbols.



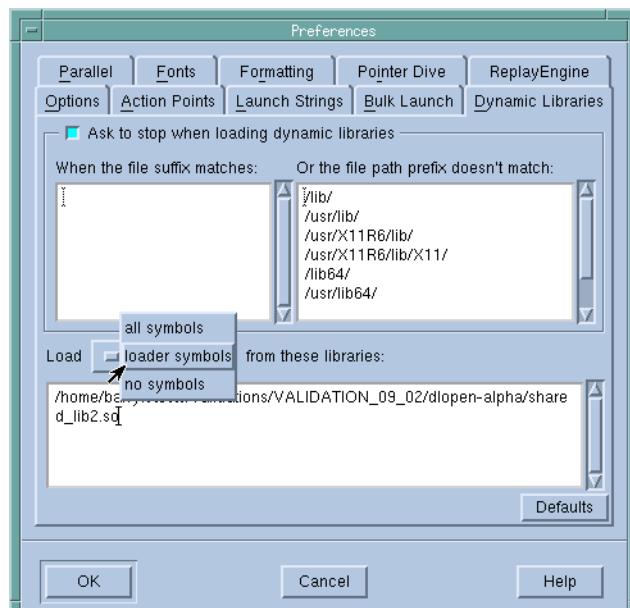
Saying "TotalView reads all symbols" isn't quite true as TotalView often just reads in loader symbols for some libraries. For example, it only reads in loader symbols if the library resides in the `/usr/lib` directory. (These libraries are typically those provided with the operating system.) You can override this behavior by adding a library name to the All Symbols list that is described in the next section.

Specifying Which Libraries are Read

After invoking the **File > Preferences** command, select the Dynamic Libraries Page. (See Figure 160 on page 247.)

The lower portion of this page lets you enter the names of libraries for which you need to manage the information that TotalView reads.

Figure 160: File >
Preferences: Dynamic
Libraries Page



When you enter a library name, you can use the * (asterisk) and ? (question mark) wildcard characters. These characters have their standard meaning. Placing entries into these areas does the following:

all symbols

This is the default operation. You only need to enter a library name here if it would be excluded by a wildcard in the **loader symbols** and **no symbols** areas.

loader symbols

TotalView reads loader symbols from these libraries. If your program uses a number of large shared libraries that you will not be debugging, you might set this to asterisk (*). You then enter the names of DLLs that you need to debug in the **all symbols** area.

no symbols

Normally, you wouldn't put anything on this list since TotalView might not be able to create a backtrace through a library if it doesn't have these symbols. However, you can increase performance if you place the names of your largest libraries here.

When reading a library, TotalView looks at these lists in the following order:

- 1 **all symbols**
- 2 **loader symbols**
- 3 **no symbols**

If a library is found in more than one area, it does the first thing it is told to do and ignores any other requests. For example, after TotalView reads a

library's symbols, it cannot honor a request to not load in symbols, so it ignores a request to not read them.

```
CLI: dset TV::dll_read_all_symbols  
      dset TV::dll_read_loader_symbols_only  
      dset TV::dll_read_no_symbols
```

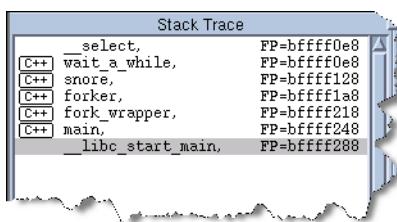
See the online Help for additional information.

If your program stops in a library that has not already had its symbols read, TotalView reads the library's symbols. For example, if your program SEGVs in a library, TotalView reads the symbols from that library before it reports the error. In all cases, however, TotalView always reads the loader symbols for shared system libraries.

Reading Excluded Information

While you are debugging your program, you might find that you do need the symbol information that you told TotalView that it shouldn't read. Tell TotalView to read them by right-clicking your mouse in the Stack Trace Pane and then selecting the **Load All Symbols in Stack** command from the context menu. (See Figure 161.)

Figure 161: Load All Symbols in Stack Context menu



After selecting this command, TotalView examines all active stack frames and, if it finds unread libraries in any frame, TotalView reads them.

```
CLI: TV::read_symbols  
This CLI command also gives you finer control over how TotalView  
reads in library information.
```

Setting the Program Counter

TotalView lets you resume execution at a different statement than the one at which it stopped execution by resetting the value of the program counter (PC). For example, you might want to skip over some code, execute

some code again after changing certain variables, or restart a thread that is in an error state.

Setting the PC can be crucial when you want to restart a thread that is in an error state. Although the PC symbol in the line number area points to the source statement that caused the error, the PC actually points to the failed machine instruction in the source statement. You need to explicitly reset the PC to the correct instruction. (You can verify the actual location of the PC before and after resetting it by displaying it in the Stack Frame Pane, or displaying both source and assembler code in the Source Pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line or a selected instruction. When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is all you need to do.

To set the PC to a selected line:

- 1 If you need to set the PC to a location somewhere in a line of source code, select the **View > Source As > Both** command.
TotalView responds by displaying assembler code.
- 2 Select the source line or instruction in the Source Pane.
TotalView highlights the line.
- 3 Select the **Thread > Set PC** command.
TotalView asks for confirmation, resets the PC, and moves the PC symbol to the selected line.

When you select a line and ask TotalView to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement in the currently selected stack frame. If the currently selected stack frame is not the top stack frame, TotalView asks if it can unwind the stack:

This frame is buried. Should we attempt to unwind the stack?

If you select **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame. If you select **No**, TotalView sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you can't assume that the stack and registers have the right values, selecting **No** is almost always the wrong thing to do.

Interpreting the Status and Control Registers

The Stack Frame Pane in the Process Window lists the contents of CPU registers for the selected frame—you might need to scroll past the stack local variables to see them.

CLI: `dprint register`

You must quote the initial \$ character in the register name; for example, `dprint \$r1`.

For your convenience, TotalView displays the bit settings of many CPU registers symbolically. For example, TotalView symbolically displays registers that control rounding and exception enable modes. You can edit the values of these registers and then resume program execution. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are displayed vary from platform to platform, see "Architectures" in the *TotalView Reference Guide* for information on how TotalView displays this information on your CPU. For general information on editing the value of variables (including registers), see "Displaying Areas of Memory" on page 296. To learn about the meaning of these registers, you see the documentation for your CPU.

Using Groups, Processes, and Threads



CHAPTER 13

The specifics of how multi-process, multi-threaded programs execute differ greatly from platform to platform and environment to environment, but all share some general characteristics. This chapter discusses the TotalView process/thread model. It also describes the way in which you tell the GUI and the CLI what processes and threads to direct a command to.

This chapter contains the following sections:

- “*Defining the GOI, POI, and TOI*” on page 251
- “*Setting a Breakpoint*” on page 252
- “*Stepping (Part I)*” on page 253
- “*Using P/T Set Controls*” on page 256
- “*Setting Process and Thread Focus*” on page 257
- “*Setting Group Focus*” on page 262
- “*Stepping (Part II): Examples*” on page 273
- “*Using P/T Set Operators*” on page 275
- “*Creating Custom Groups*” on page 276

Defining the GOI, POI, and TOI

This chapter consistently uses the following three related acronyms:

- GOI—Group of Interest
- POI—Process of Interest
- TOI—Thread of Interest

These terms are important in the TotalView process/thread model because TotalView must determine the scope of what it does when it executes a command. For example, Chapter 2, “*About Threads, Processes, and Groups*”

introduced the types of groups contained with TotalView. That chapter ignored what happens when you execute a TotalView command on a group. For example, what does “stepping a group” actually mean? What happens to processes and threads that aren’t in this group?

Associated with these three terms is a fourth term: *arena*. The *arena* is the collection of processes, threads, and groups that are affected by a debugging command. This collection is called an *arena list*.

In the GUI, the arena is most often set using the pulldown list in the toolbar. You can set the arena using commands in the menubar. For example, there are eight *next* commands. The difference between them is the arena; that is, the difference between the *next* commands is the processes and threads that are the target of what the *next* command runs.

When TotalView executes any action command, the arena decides the scope of what can run. It doesn’t, however, determine what does run. Depending on the command, TotalView determines the TOI, POI, or GOI, and then executes the command’s action on that thread, process, or group. For example, suppose TotalView steps the current control group:

- TotalView needs to know what the TOI is so that it can determine what threads are in the lockstep group—TotalView only lets you step a lockstep group.
- The lockstep group is part of a share group.
- This share group is also contained in a control group.

By knowing what the TOI is, the GUI also knows what the GOI is. This is important because, while TotalView knows what it will step (the threads in the lockstep group), it also knows what it will allow to run freely while it is stepping these threads. In the CLI, the P/T set determines the TOI.

Setting a Breakpoint

You can set breakpoints in your program by selecting the boxed line numbers in the Source Code pane of a Process window. A boxed line number indicates that the line generates executable code. A **STOP** icon masking a line number indicates that a breakpoint is set on the line. Selecting the **STOP** icon clears the breakpoint.

When a program reaches a breakpoint, it stops. You can let the program resume execution in any of the following ways:

- Use the single-step commands described in “*Using Stepping Commands*” on page 239.
- Use the set program counter command to resume program execution at a specific source line, machine instruction, or absolute hexadecimal value. See “*Setting the Program Counter*” on page 248.

- Set breakpoints at lines you choose, and let your program execute to that breakpoint. See "Setting Breakpoints and Barriers" on page 351.
- Set conditional breakpoints that cause a program to stop after it evaluates a condition that you define, for example "stop when a value is less than eight. See "Setting Eval Points" on page 367.

TotalView provides additional features for working with breakpoints, process barrier breakpoints, and eval points. For more information, see Chapter 16, "Setting Action Points," on page 349.

Stepping (Part I)

You can use TotalView stepping commands to:

- Execute one source line or machine instruction at a time; for example, **Process > Step** in the GUI and **dstep** in the CLI.

CLI: dstep

- Run to a selected line, which acts like a temporary breakpoint; for example, **Process > Run To**.

CLI: duntil

- Run until a function call returns; for example, **Process > Out**.

CLI: dout

In all cases, stepping commands operate on the Thread of Interest (TOI). In the GUI, the TOI is the selected thread in the current Process Window. In the CLI, the TOI is the thread that TotalView uses to determine the scope of the stepping operation.

On all platforms except SPARC Solaris, TotalView uses *smart* single-stepping to speed up stepping of one-line statements that contain loops and conditions, such as Fortran 90 array assignment statements. *Smart stepping* occurs when TotalView realizes that it doesn't need to step through an instruction. For example, assume that you have the following statements:

```
integer iarray (1000,1000,1000)
iarray = 0
```

These two statements define one billion scalar assignments. If your computer steps every instruction, you will probably never get past this statement. *Smart stepping* means that TotalView single-steps through the assignment statement at a speed that is very close to your computer's native speed.

Other topics in this section are:

- "Understanding Group Widths" on page 254
- "Understanding Process Width" on page 254
- "Understanding Thread Width" on page 255
- "Using Run To and duntil Commands" on page 255

Understanding Group Widths

TotalView behavior when stepping at group width depends on whether the Group of Interest (GOI) is a process group or a thread group. In the following lists, *goal* means the place at which things should stop executing. For example, if you selected a *step* command, the goal is the next line. If you selected a *run to* command, the goal is the selected line.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView examines the group, and identifies which of its processes has a thread stopped at the same location as the TOI (a *matching* process). TotalView runs these matching processes until one of its threads arrives at the goal. When this happens, TotalView stops the thread's process. The command finishes when it has stopped all of these *matching* processes.
- *Thread group*—TotalView runs all processes in the control group. However, as each thread arrives at the goal, TotalView only stops that thread; the rest of the threads in the same process continue executing. The command finishes when all threads in the GOI arrive at the goal. When the command finishes, TotalView stops all processes in the control group. TotalView doesn't wait for threads that are not in the same share group as the TOI, since they are executing different code and can never arrive at the goal.

Understanding Process Width

TotalView behavior when stepping at process width (which is the default) depends on whether the Group of Interest (GOI) is a process group or a thread group.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView runs all threads in the process, and execution continues until the TOI arrives at its goal, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal does TotalView stop the other threads in the process.
- *Thread group*—TotalView lets all threads in the GOI and all manager threads run. As each member of the GOI arrives at the goal, TotalView stops it; the rest of the threads continue executing. The command finishes when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

Understanding Thread Width

When TotalView performs a stepping command, it decides what it steps based on the *width*. Using the toolbar, you specify width using the left-most pulldown. This pulldown has three items: **Group**, **Process**, and **Thread**.

Stepping at thread width tells TotalView to only run that thread. It does not step other threads. In contrast, process width tells TotalView to run all threads in the process that are allowed to run while the TOI is stepped. While TotalView is stepping the thread, manager threads run freely.

Stepping a thread isn't the same as stepping a thread's process, because a process can have more than one thread.



Thread-stepping is not implemented on Sun platforms. On SGI platforms, thread-stepping is not available with pthread programs. If, however, your program's parallelism is based on SGI's sprocs, thread-stepping is available.

Thread-level single-step operations can fail to complete if the TOI needs to synchronize with a thread that isn't running. For example, if the TOI requires a lock that another held thread owns, and steps over a call that tries to acquire the lock, the primary thread can't continue successfully. You must allow the other thread to run in order to release the lock. In this case, you use process-width stepping instead.

Using Run To and duntil Commands

The **duntil** and **Run To** commands differ from other step commands when you apply them to a process group. (These commands tell TotalView to execute program statements *until* it reaches the selected statement.) When used with a process group, TotalView identifies all processes in the group that already have a thread stopped at the goal. These are the *matching* processes. TotalView then runs only nonmatching processes. Whenever a thread arrives at the goal, TotalView stops its process. The command finishes when it has stopped all members of the group. This lets you synchronize all the processes in a group in preparation for group-stepping them.

You need to know the following if you're running at process width:

Process group If the Thread of Interest (TOI) is already at the goal location, TotalView steps the TOI past the line before the process runs. This lets you use the **Run To** command repeatedly in loops.

Thread group If any thread in the process is already at the goal, TotalView temporarily holds it while other threads in the process run. After all threads in the thread group reach the goal, TotalView stops the process. This lets you synchronize the threads in the POI at a source line.

Using P/T Set Controls

If you're running at group width:

Process group TotalView examines each process in the process and share group to determine whether at least one thread is already at the goal. If a thread is at the goal, TotalView holds its process. Other processes are allowed to run. When at least one thread from each of these processes is held, the command completes. This lets you synchronize at least one thread in each of these processes at a source line. If you're running a control group, this synchronizes all processes in the share group.

Thread group TotalView examines all the threads in the thread group that are in the same share group as the TOI to determine whether a thread is already at the goal. If it is, TotalView holds it. Other threads are allowed to run. When all of the threads in the TOI's share group reach the goal, TotalView stops the TOI's *control* group and the command completes. This lets you synchronize thread group members. If you're running a workers group, this synchronizes all worker threads in the share group.

The process stops when the TOI and at least one thread from each process in the group or process being run reach the command stopping point. This lets you synchronize a group of processes and bring them to one location.

You can also run to a selected line in a nested stack frame, as follows:

- 1 Select a nested frame in the Stack Trace Pane.
- 2 Select a source line or instruction in the function.
- 3 Enter a **Run To** command.

TotalView executes the primary thread until it reaches the selected line in the selected stack frame.

Using P/T Set Controls

A few GUI windows have P/T set control elements. For example, the following figure shows the top portion of the Process Window.

Figure 162: The P/T Set Control in the Process Window



When you select a scope modifier, you are telling TotalView that when you press one of the remaining buttons on the toolbar, this element names the focus on which TotalView acts. For example, if you select a thread and then select **Step**, TotalView steps the current thread. If **Process (workers)** is selected and you select **Halt**, TotalView halts all processes associated with the current threads workers group. If you were running a multi-process program, other processes continue to execute.

The Processes/Ranks Tab shows you which processes or ranks are members of the group. For more information, see "Using the Toolbar to Select a Target" on page 228.

Setting Process and Thread Focus



The previous sections emphasize the GUI; this section and the ones that follow emphasize the CLI. Here you will find information on how to have full asynchronous debugging control over your program. Fortunately, having this level of control is seldom necessary. In other words, don't read the rest of this chapter unless you have to.

When TotalView executes a command, it must decide which processes and threads to act on. Most commands have a default set of threads and processes and, in most cases, you won't want to change the default. In the GUI, the default is the process and thread in the current Process Window. In the CLI, this default is indicated by the focus, which is shown in the CLI prompt.

There are times, however, when you need to change this default. This section begins a rather intensive look at how you tell TotalView what processes and threads to use as the target of a command.

Topics in this section are:

- "Understanding Process/Thread Sets" on page 257
- "Specifying Arenas" on page 259
- "Specifying Processes and Threads" on page 259

Understanding Process/Thread Sets

All TotalView commands operate on a set of processes and threads. This set is called a *Process/Thread (P/T) set*. The right-hand text box in windows that contain P/T set controls lets you construct these sets. In the CLI, you specify a P/T set as an argument to a command such as **dfocus**. If you're using the GUI, TotalView creates this list for you based on which Process Window has focus.

Unlike a serial debugger in which each command clearly applies to the only executing thread, TotalView can control and monitor many threads with their PCs at many different locations. The P/T set indicates the groups, processes, and threads that are the target of the CLI command. No limitation exists on the number of groups, processes, and threads in a set.

A P/T set is a list that contains one or more P/T identifiers. (The next section, “*Specifying Arenas*” on page 259, explains what a P/T identifier is.) Tcl lets you create lists in the following ways:

- You can enter these identifiers within braces (`{ }`).
- You can use Tcl commands that create and manipulate lists.

These lists are then used as arguments to a command. If you’re entering one element, you usually do not have to use the Tcl list syntax.

For example, the following list contains specifiers for process 2, thread 1, and process 3, thread 2:

```
{p2.1 p3.2}
```

If you do not explicitly specify a P/T set in the CLI, TotalView defines a target set for you. (In the GUI, the default set is determined by the current Process Window.) This set is displayed as the default CLI prompt. (For information on this prompt, see “*About the CLI Prompt*” on page 207.)

You can change the focus on which a command acts by using the **dfocus** command. If the CLI executes the **dfocus** command as a unique command, it changes the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

After TotalView executes this command, all commands that follow focus on process 2.



In the GUI, you set the focus by displaying a Process Window that contains this process. Do this by using the P+ and P- buttons in the tab bar at the bottom, by making a selection in the Processes/Ranks Tab, or by clicking on a process in the Root Window.

If you begin a command with **dfocus**, TotalView changes the target only for the command that follows. After the command executes, TotalView restores the *former* default. The following example shows both of these ways to use the **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then step the threads in this group twice:

```
dfocus g2  
dstep  
dstep
```

In contrast, if the current focus is process 1, thread 1, the following commands step group 2 and then step process 1, thread 1:

```
dfocus g2 dstep  
dstep
```

Some commands only operate at the process level; that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Specifying Arenas

A P/T identifier often indicates a number of groups, processes, and threads. For example, assume that two threads executing in process 2 are stopped at the same statement. This means that TotalView places the two stopped threads into lockstep groups. If the default focus is process 2, stepping this process actually steps both of these threads.

TotalView uses the term *arena* to define the processes and threads that are the target of an action. In this case, the arena has two threads. Many CLI commands can act on one or more arenas. For example, the following command has two arenas:

```
dfocus {p1 p2}
```

The two arenas are process 1 and process 2.

When there is an arena list, each arena in the list has its own GOI, POI, and TOI.

Specifying Processes and Threads

The previous sections described P/T sets as being lists; however, these discussions ignored what the individual elements of the list are. A better definition is that a P/T set is a list of arenas, where an *arena* consists of the processes, threads, and groups that are affected by a debugging command. Each *arena specifier* describes a single arena in which a command acts; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some CLI output commands, however, combine arenas and act on them as a single target.

An arena specifier includes a *width* and a TOI. (Widths are discussed later in this section.) In the P/T set, the TOI specifies a target thread, while the width specifies how many threads surrounding the thread of interest are affected.

Defining the Thread of Interest (TOI)

The TOI is specified as **p.t**, where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID). The **p.t** combination identifies the POI (Process of Interest) and TOI. The TOI is the primary thread affected by a command. This means that it is the primary focus for a TotalView command. For example, while the **dstep** command always steps the TOI, it can run the rest of the threads in the POI and step other processes in the group.

In addition to using numerical values, you can also use two special symbols:

- The less-than character (<) indicates the *lowest numbered worker thread* in a process, and is used instead of the TID value. If, however, the arena explicitly names a thread group, the < symbol means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which might not be thread 1; for example, the first and only user thread might be thread number 3 on HP Alpha systems.
- A dot (.) indicates the current set. Although you seldom use this symbol interactively, it can be useful in scripts.

About Process and Thread Widths

You can enter a P/T set in two ways. If you’re not manipulating groups, the format is as follows:

`[width_letter][pid][.tid]`



"*Specifying Groups in P/T Sets*" on page 263 extends this format to include groups. When using P/T sets, you can create sets with just width indicators or just group indicators, or both.

For example, `p2.3` indicates process 2, thread 3.

Although the syntax seems to indicate that you do not need to enter any element, TotalView requires that you enter at least one. Because TotalView tries to determine what it can do based on what you type, it tries to fill in what you omit. The only requirement is that when you use more than one element, you use them in the order shown here.

You can leave out parts of the P/T set if what you do enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be `<`. For more information, see "*Naming Incomplete Arenas*" on page 272.

The `width_letter` indicates which processes and threads are part of the arena. You can use the following letters:

t	<i>Thread width</i>
	A command’s target is the indicated thread.
p	<i>Process width</i>
	A command’s target is the process that contains the TOI.
g	<i>Group width</i>
	A command’s target is the group that contains the POI. This indicates control and share groups.
a	<i>All processes</i>
	A command’s target is all threads in the GOI that are in the POI.
d	<i>Default width</i>
	A command’s target depends on the default for each command. This is also the width to which the default focus is set. For example, the dstep command defaults to process width (run the process while stepping one thread), and the dwhere command defaults to thread width.

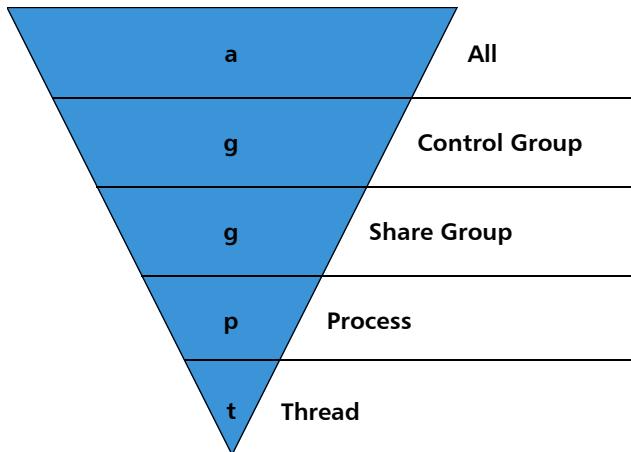
You must use lowercase letters to enter these widths.

Figure 163 on page 261 illustrates how these specifiers relate to one another.

The **g** specifier indicates control and share groups. This inverted triangle indicates that the arena focuses on a greater number of entities as you move from **Thread** level at the bottom to **All** level at the top.

As mentioned previously, the TOI specifies a target thread, while the width specifies how many threads surrounding the TOI are also affected. For

Figure 163: Width Specifiers



example, the **dstep** command always requires a TOI, but entering this command can do the following:

- Step just the TOI during the step operation (thread-level single-step).
- Step the TOI and step all threads in the process that contain the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC as the TOI (group-level single-step).

This list doesn't indicate what happens to other threads in your program when TotalView steps your thread. For more information, see "Stepping (Part II): Examples" on page 273.

To save a P/T set definition for later use, assign the specifiers to a Tcl variable; for example:

```
set myset {g2.3 t3.1}
dfocus $myset dgo
```

As the **dfocus** command returns its focus set, you can save this value for later use; for example:

```
set save_set [dfocus]
```

Specifier Examples

The following are some sample specifiers:

- | | |
|----------------|---|
| g2.3 | Select process 2, thread 3, and set the width to <i>group</i> . |
| t1.7 | Commands act only on thread 7 or process 1. |
| d1.< | Use the default set for each command, focusing on the first user thread in process 1. The less-than symbol (<) sets the TID to the first user thread. |

Setting Group Focus

TotalView has two types of groups: process groups and thread groups. Process groups only contain processes, and thread groups only contain threads. The threads in a thread group can be drawn from more than one process.

Topics in this section are:

- "Specifying Groups in P/T Sets" on page 263
- "About Arena Specifier Combinations" on page 264
- "'All' Does Not Always Mean 'All'" on page 267
- "Setting Groups" on page 268
- "Using the *g* Specifier: An Extended Example" on page 269
- "Merging Focuses" on page 271
- "Naming Incomplete Arenas" on page 272
- "Naming Lists with Inconsistent Widths" on page 273

TotalView has four predefined groups. Two of these only contain processes, while the other two only contain threads. TotalView also lets you create your own groups, and these groups can have elements that are processes and threads. The following are the predefined process groups:

■ Control Group

Contains the parent process and all related processes. A control group includes children that were forked (processes that share the same source code as the parent) and children that were forked but subsequently called the `execve()` function.

Assigning a new value to the `CGROUP (dpid)` variable for a process changes that process's control group. In addition, the `dgroups -add` command lets you add members to a group in the CLI. In the GUI, you use the **Group > Custom Groups** command.

■ Share Group

Contains all members of a control group that share the same executable. TotalView automatically places processes in share groups based on their control group and their executable.



You can't change a share group's members. However, the dynamically loaded libraries used by group members can be different.

The following are the predefined thread groups:

■ Workers Group

Contains all worker threads from all processes in the control group. The only threads not contained in a worker's group are your operating system's manager threads.

■ Lockstep Group

Contains every stopped thread in a share group that has the same PC. TotalView creates one lockstep group for every thread. For example, suppose two threads are stopped at the same PC. TotalView creates two lockstep groups. While each lockstep group has the same two members, they differ in that each has a different TOI. While there are some circumstances where this is important, you can usually ignore this distinction. That is, while two lockstep groups exist if two threads are stopped at the same PC, ignoring the second lockstep group is almost never harmful.

The group ID value for a lockstep group differs from the ID of other groups. Instead of having an automatically and transient integer ID, the lockstep group ID is **pid.tid**, where **pid.tid** identifies the thread with which it is associated. For example, the lockstep group for thread 2 in process 1 is **1.2**.

In general, if you're debugging a multi-process program, the control group and share group differ only when the program has children that it forked with by calling the **execve()** function.

Specifying Groups in P/T Sets

This section extends the arena specifier syntax to include groups.

If you do not include a group specifier, the default is the control group. For example, the CLI only displays a target group in the focus string if you set it to something other than the default value.



You most often use target group specifiers with the stepping commands, as they give these commands more control over what's being stepped.

Use the following format to add groups to an arena specifier:

`[width_letter][group_indicator][pid][.tid]`

This format adds the *group_indicator* to what was discussed in "Specifying Processes and Threads" on page 259.

In the description of this syntax, everything appears to be optional. But, while no single element is required, you must enter at least one element. TotalView determines other values based on the current focus.

TotalView lets you identify a group by using a letter, number, or name.

A Group Letter

You can name one of TotalView's predefined sets. Each set is identified by a letter. For example, the following command sets the focus to the workers group:

`dfocus W`

The following are the group letters. These letters are in uppercase:

C	<i>Control group</i> All processes in the control group.
D	<i>Default control group</i> All processes in the control group. The only difference between this specifier and the C specifier is that this letter tells the CLI not to display a group letter in the CLI prompt.
S	<i>Share group</i> The set of processes in the control group that have the same executable as the arena's TOI.
W	<i>Workers group</i> The set of all worker threads in the control group.
L	<i>Lockstep group</i> A set that contains all threads in the share group that have the same PC as the arena's TOI. If you step these threads as a group, they proceed in lockstep.

A Group Number

You can identify a group by the number TotalView assigns to it. The following example sets the focus to group 3:

`dfocus 3/`

The trailing slash tells TotalView that you are specifying a group number instead of a PID. The slash character is optional if you're using a *group_letter*. However, you must use it as a separator when entering a numeric group ID and a **pid.tid** pair. For example, the following example identifies process 2 in group 3:

`p3/2`

A Group Name

You can name a set that you define. You enter this name with slashes. The following example sets the focus to the set of threads contained in process 3 that are also contained in a group called **my_group**:

`dfocus p/my_group/3`

About Arena Specifier Combinations

The following table lists what's selected when you use arena and group specifiers to step your program:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."

Specifier	Specifies
gC	All threads in the Thread of Interest (TOI) control group.
gS	All threads in the TOI share group.
gW	All worker threads in the control group that contains the TOI.
gL	All threads in the same share group within the process that contains the TOI that have the same PC as the TOI.
pC	All threads in the control group of the Process of Interest (POI). This is the same as gC .
pS	All threads in the process that participate in the same share group as the TOI.
pW	All worker threads in the POI.
pL	All threads in the POI whose PC is the same as the TOI.
tC	Just the TOI. The t specifier overrides the group specifier. So, for example, tW and t both name the current thread.
tS	
tW	
tL	



*Stepping commands behave differently if the group being stepped is a process group rather than a thread group. For example, **aC** and **aS** perform the same action, but **aL** is different.*

If you don't add a PID or TID to your arena specifier, TotalView does it for you, taking the PID and TID from the current or default focus.

The following are some additional examples. These examples add PIDs and TIDs numbers to the raw specifier combinations listed in the previous table:

pW3	All worker threads in process 3.
pW3.<	All worker threads in process 3. The focus of this specifier is the same as the focus in the previous example.
gW3	All worker threads in the control group that contains process 3. The difference between this and pW3 is that pW3 restricts the focus to one of the processes in the control group.
gL3.2	All threads in the same <i>share</i> group as process 3 that are executing at the same PC as thread 2 in process 3. The reason this is a share group and not a control group is that different share groups can reside only in one control group.
/3	Specifies processes and threads in process 3. The arena width, POI, and TOI are inherited from the existing P/T set, so the exact meaning of this specifier depends on the previous context. While the slash is unnecessary because no group is indicated, it is syntactically correct.
g3.2/3	The 3.2 group ID is the name of the lockstep group for thread 3.2. This group includes all threads in the process 3 share group that are executing at the same PC as thread 2.

p3/3 Sets the process to process 3. The Group of Interest (GOI) is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act on all threads in process 3 that are also in group 3.

When you set the process using an explicit group, you might not be including all the threads you expect to be included. This is because commands must look at the TOI, POI, and GOI.



It is redundant to specify a thread width with an explicit group ID as this width means that the focus is on one thread.

In the following examples, the first argument to the **dfocus** command defines a temporary P/T set that the CLI command (the last term) operates on. The **dstatus** command lists information about processes and threads. These examples assume that the global focus was **d1.<** initially.

dfocus g dstatus

Displays the status of all threads in the control group.

dfocus gW dstatus

Displays the status of all worker threads in the control group.

dfocus p dstatus

Displays the status of all worker threads in the current focus process. The width here, as in the previous example, is process, and the (default) group is the control group. The intersection of this width and the default group creates a focus that is the same as in the previous example.

dfocus pW dstatus

Displays the status of all worker threads in the current focus process. The width is process level, and the target is the workers group.

The following example shows how the prompt changes as you change the focus. In particular, notice how the prompt changes when you use the **C** and the **D** group specifiers.

```
d1.<> f C  
dC1.<  
dC1.<> f D  
d1.<  
d1.<>
```

Two of these lines end with the less-than symbol (<). These lines aren't prompts. Instead, they are the value returned by TotalView when it executes the **dfocus** command.

'All' Does Not Always Mean 'All'

When you use stepping commands, TotalView determines the scope of what runs and what stops by looking at the TOI. This section looks at the differences in behavior when you use the **a** (all) arena specifier. The following table describes what runs when you use this arena:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."

The following are some combinations:

f aC dgo Runs everything. If you're using the **dgo** command, everything after the **a** is ignored: **a/aPizza/17.2, ac, aS, and aL** do the same thing. TotalView runs everything.

f aC duntil While everything runs, TotalView must wait until something reaches a goal. It really isn't obvious what this focus is. Since **C** is a process group, you might guess that all processes run until at least one thread in every participating process arrives at a goal. The reality is that since this goal must reside in the current share group, this command completes as soon as all processes in the TOI share group have at least one thread at the goal. Processes in other control groups run freely until this happens.

The TOI determines the goal. If there are other control groups, they do not participate in the goal.

f aS duntil This command does the same thing as the **f aC duntil** command because the goals for **f aC duntil** and **f aS duntil** are the same, and the processes that are in this scope are identical.

Although more than one share group can exist in a control group, these other share groups do not participate in the goal.

f aL duntil Although everything will run, it is not clear what should occur. **L** is a thread group, so you might expect that the **duntil** command will wait until all threads in all lockstep groups arrive at the goal. Instead, TotalView defines the set of threads that it allows to run to a goal as just those threads in the TOI's lockstep group. Although there are other lockstep groups, these lockstep groups do not participate in the goal. So, while the TOI's lockstep threads are progressing towards their goal, all threads that were previously stopped run freely.

f aW duntil Everything runs. TotalView waits until all members of the TOI workers group arrive at the goal.

Two broad distinctions between process and thread group behavior exist:

- When the focus is on a process group, TotalView waits until just one thread from each participating process arrives at the goal. The other threads just run.

When focus is on a thread group, every participating thread must arrive at the goal.

- When the focus is on a process group, TotalView steps a thread over the goal breakpoint and continues the process if it isn't the right thread.

When the focus is on a thread group, TotalView holds a thread even if it isn't the right thread. It also continues the rest of the process.

If your system doesn't support asynchronous thread control, TotalView treats thread specifiers as if they were process specifiers.

With this in mind, **f aL dstep** does not step all threads. Instead, it steps only the threads in the TOI's lockstep group. All other threads run freely until the stepping process for these lockstep threads completes.

Setting Groups

This section presents a series of examples that set and create groups.

You can use the following methods to indicate that thread 3 in process 2 is a worker thread:

dset WGROUP(2.3) \$WGROUP(2)

Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a nonzero value to **WGROUP** indicates that this is a worker group.)

dset WGROUP(2.3) 1

This is a simpler way of doing the same thing as the previous example.

dfocus 2.3 dworker 1

Adds the groups in the indicated focus to a workers group.

dset CGROUP(2) \$CGROUP(1)

dgroups -add -g \$CGROUP(1) 2

dfocus 1 dgroups -add 2

These three commands insert process 2 into the same control group as process 1.

dgroups -add -g \$WGROUP(2) 2.3

Adds process 2, thread 3 to the workers group associated with process 2.

dfocus tw2.3 dgroups -add

This is a simpler way of doing the same thing as the previous example.

The following are some additional examples:

```
dfocus g1 dgroups -add -new thread
```

Creates a new thread group that contains all the threads in all the processes in the control group associated with process 1.

```
set mygroup [dgroups -add -new thread  
$GROUP($SGROUP(2))]  
dgroups -remove -g $mygroup 2.3  
dfocus g$mygroup/2 dgo
```

These three commands define a new group that contains all the threads in the process 2 share group except for thread 2.3, and then continue that set of threads. The first command creates a new group that contains all the threads from the share group; the second removes thread 2.3; and the third runs the remaining threads.

Using the **g** Specifier: An Extended Example

The meaning of the **g** width specifier is sometimes not clear when it is coupled with a group scope specifier. Why have a **g** specifier when you have four other group specifiers? Stated in another way, isn't something like **gL** redundant?

The simplest answer, and the reason you most often use the **g** specifier, is that it forces the group when the default focus indicates something different from what you want it to be.

The following example shows this. The first step sets a breakpoint in a multi-threaded OMP program and execute the program until it hits the breakpoint.

```
d1.<> dbreak 35  
Loaded OpenMP support library libguidedb_3_8.so :  
KAP/Pro Toolset 3.8  
1  
d1.<> dcont  
Thread 1.1 has appeared  
Created process 1/37258, named "omp_prog"  
Thread 1.1 has exited  
Thread 1.1 has appeared  
Thread 1.2 has appeared  
Thread 1.3 has appeared  
Thread 1.1 hit breakpoint 1 at line 35 in  
".breakpoint_here"
```

The default focus is **d1.<**, which means that the CLI is at its default width, the POI is 1, and the TOI is the lowest numbered nonmanager thread.

Because the default width for the **dstatus** command is process, the CLI displays the status of all processes. Typing **dfocus p dstatus** produces the same output.

```
d1.<> dstatus  
1: 37258 Breakpoint [omp_prog]  
1.1: 37258.1 Breakpoint PC=0x1000acd0,  
[./omp_prog.f#35]
```

```

1.2: 37258.2 Stopped      PC=0xfffffffffffffff
1.3: 37258.3 Stopped      PC=0xd042c944
d1.<> dfocus p dstatus
1:      37258   Breakpoint  [omp_prog]
1.1: 37258.1 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]
1.2: 37258.2 Stopped      PC=0xfffffffffffffff
1.3: 37258.3 Stopped      PC=0xd042c944

```

The CLI displays the following when you ask for the status of the lockstep group. (The rest of this example uses the **f** abbreviation for **dfocus**, and **st** for **dstatus**.)

```

d1.<> f L st
1:      37258   Breakpoint  [omp_prog]
1.1: 37258.1 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]

```

This command tells the CLI to display the status of the threads in thread, which is the 1.1 lockstep group since this thread is the TOI. The **f L focus** command narrows the set so that the display only includes the threads in the process that are at the same PC as the TOI.



*By default, the **dstatus** command displays information at process width. This means that you don't need to type **f pL dstatus**.*

The **duntil** command runs thread 1.3 to the same line as thread 1.1. The **dstatus** command then displays the status of all the threads in the process:

```

d1.<> f t1.3 duntil 35
35@>           write(*,*)"i= ",i,
           "thread= ",omp_get_thread_num()
d1.<> f p dstatus
1:      37258   Breakpoint  [omp_prog]
1.1: 37258.1 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]
1.2: 37258.2 Stopped      PC=0xfffffffffffffff
1.3: 37258.3 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]

```

As expected, the CLI adds a thread to the lockstep group:

```

d1.<> f L dstatus
1:      37258   Breakpoint  [omp_prog]
1.1: 37258.1 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]
1.3: 37258.3 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]

```

The next set of commands begins by narrowing the width of the default focus to thread width—notice that the prompt changes—and then displays the contents of the lockstep group:

```

d1.<> f t
t1.<> f L dstatus
1:      37258   Breakpoint  [omp_prog]
1.1: 37258.1 Breakpoint  PC=0x1000acd0,
      [.omp_prog.f#35]

```

Although the lockstep group of the TOI has two threads, the current focus has only one thread, and that thread is, of course, part of the lockstep group. Consequently, the lockstep group *in the current focus* is just the one thread, even though this thread's lockstep group has two threads.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group of thread 1.1. as follows:

```
t1.<> f pL dstatus
1:      37258  Breakpoint  [omp_prog]
        1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]
        1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]

t1.<> f gL dstatus
1:      37258  Breakpoint  [omp_prog]
        1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]
        1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]
```



If the TOI is 1.1, **L** refers to group number 1.1, which is the lockstep group of thread 1.1.

Because this example only contains one process, the **pL** and **gL** specifiers produce the same result when used with the **dstatus** command. If, however, there were additional processes in the group, you only see them when you use the **gL** specifier.

Merging Focuses

When you specify more than one focus for a command, the CLI merges them together. In the following example, the focus indicated by the prompt—this focus is called the *outer* focus—controls the display. This example shows what happens when **dfocus** commands are strung together:

```
t1.<> f d
d1.<
d1.<> f tL dstatus
1:      37258  Breakpoint  [omp_prog]
        1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]

d1.<> f tL f p dstatus
1:      37258  Breakpoint  [omp_prog]
        1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]
        1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]

d1.<> f tL f p f D dstatus
1:      37258  Breakpoint  [omp_prog]
        1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                  [.omp_prog.f#35]
        1.2: 37258.2 Stopped    PC=0xffffffffffff
```

```
1.3: 37258.3 Breakpoint PC=0x1000acd0,  
      [./omp_prog.f#35]  
d1.<> f tL f p f D f L dstatus  
1: 37258 Breakpoint [omp_prog]  
1.1: 37258.1 Breakpoint PC=0x1000acd0,  
      [./omp_prog.f#35]  
1.3: 37258.3 Breakpoint PC=0x1000acd0,  
      [./omp_prog.f#35]
```

Stringing multiple focuses together might not produce the most readable result. In this case, it shows how one **dfocus** command can modify what another sees and acts on. The ultimate result is an arena that a command acts on. In these examples, the **dfocus** command tells the **dstatus** command what to display.

Naming Incomplete Arenas

In general, you do not need to completely specify an arena. TotalView provides values for any missing elements. TotalView either uses built-in default values or obtains them from the current focus. The following explains how TotalView fills in missing pieces:

- If you don't use a width, TotalView uses the width from the current focus.
- If you don't use a PID, TotalView uses the PID from the current focus.
- If you set the focus to a list, there is no longer a default arena. This means that you must explicitly name a width and a PID. You can, however, omit the TID. (If you omit the TID, TotalView defaults to the less-than symbol <.)

You can type a PID without typing a TID. If you omit the TID, TotalView uses the default <, where < indicates the lowest numbered worker thread in the process. If, however, the arena explicitly names a thread group, < means the lowest numbered member of the thread group.

TotalView doesn't use the TID from the current focus, since the TID is a process-relative value.

- A dot before or after the number specifies a process or a thread. For example, **1.** is clearly a PID, while **.7** is clearly a TID.

If you type a number without typing a dot, the CLI most often interprets the number as being a PID.

- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and don't specify a PID or TID, TotalView uses the PID and TID from the current focus.

If you use a letter as a group specifier, TotalView obtains the rest of the arena specifier from the default focus.

- You can use a group ID or tag followed by a /. TotalView obtains the rest of the arena from the default focus.

Focus merging can also influence how TotalView fills in missing specifiers. For more information, see "Merging Focuses" on page 271.

Naming Lists with Inconsistent Widths

TotalView lets you create lists that contain more than one width specifier. This can be very useful, but it can be confusing. Consider the following:

`{p2 t7 g3.4}`

This list is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should TotalView use this set of processes, groups, and threads?

In most cases, TotalView does what you would expect it to do: it iterates over the list and acts on each arena. If TotalView cannot interpret an inconsistent focus, it prints an error message.

Some commands work differently. Some use each arena's width to determine the number of threads on which it acts. This is exactly what the `dgo` command does. In contrast, the `dwhere` command creates a call graph for process-level arenas, and the `dstep` command runs all threads in the arena while stepping the TOI. TotalView may wait for threads in multiple processes for group-level arenas. The command description in the *TotalView Reference Guide* points out anything that you need to watch out for.

Stepping (Part II): Examples

The following are examples that use the CLI stepping commands:

■ Step a single thread

While the thread runs, no other threads run (except kernel manager threads).

Example: `dfocus t dstep`

■ Step a single thread while the process runs

A single thread runs into or through a critical region.

Example: `dfocus p dstep`

■ Step one thread in each process in the group

While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

Example: `dfocus g dstep`

■ Step all worker threads in the process while nonworker threads run

Worker threads run through a parallel region in lockstep.

Example: `dfocus pw dstep`

■ Step all workers in the share group

All processes in the share group participate. The nonworker threads run.

Example: `dfocus gw dstep`

■ **Step all threads that are at the same PC as the TOI**

TotalView selects threads from one process or the entire share group. This differs from the previous two items in that TotalView uses the set of threads are in lockstep with the TOI rather than using the workers group.

Example: **dfocus L dstep**

In the following examples, the default focus is set to **d1.<**.

dstep Steps the TOI while running all other threads in the process.

dfocus W dnext Runs the TOI and all other worker threads in the process to the next statement. Other threads in the process run freely.

dfocus W duntil 37

Runs all worker threads in the process to line 37.

dfocus L dnext Runs the TOI and all other stopped threads at the same PC to the next statement. Other threads in the process run freely. Threads that encounter a temporary breakpoint in the course of running to the next statement usually join the lockstep group.

dfocus gW duntil 37

Runs all worker threads in the share group to line 37. Other threads in the control group run freely.

UNW 37 Performs the same action as the previous command: runs all worker threads in the share group to line 37. This example uses the predefined **UNW** alias instead of the individual commands. That is, **UNW** is an alias for **dfocus gW duntil**.

SL Finds all threads in the share group that are at the same PC as the TOI and steps them all in one statement. This command is the built-in alias for **dfocus gL dstep**.

sl Finds all threads in the current process that are at the same PC as the TOI, and steps them all in one statement. This command is the built-in alias for **dfocus L dstep**.

Using P/T Set Operators

At times, you do not want all of one type of group or process to be in the focus set. TotalView lets you use the following three operators to manage your P/T sets:

- | Creates a union; that is, all members of two sets.
- Creates a difference; that is, all members of the first set that are not also members of the second set.
- & Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, the following creates a union of two P/T sets:

`p3 | L2`

You can, apply these operations repeatedly; for example:

`p2 | p3 & L2`

This statement creates an intersection between p3 and L2, and then creates a union between p2 and the results of the intersection operation. You can directly specify the order by using parentheses; for example:

`p2 | (p3 & pL2)`

Typically, these three operators are used with the following P/T set functions:

<code>breakpoint(ptset)</code>	Returns a list of all threads that are stopped at a breakpoint.
<code>comm(process, "comm_name")</code>	Returns a list containing the first thread in each process associated within a communicator within the named process. While <i>process</i> is a P/T set it is not expanded into a list of threads.
<code>error(ptset)</code>	Returns a list of all threads stopped due to an error.
<code>existent(ptset)</code>	Returns a list of all threads.
<code>held(ptset)</code>	Returns a list of all threads that are held.
<code>nonexistent(ptset)</code>	Returns a list of all processes that have exited or which, while loaded, have not yet been created.
<code>running(ptset)</code>	Returns a list of all running threads.
<code>stopped(ptset)</code>	Returns a list of all stopped threads.
<code>unheld(ptset)</code>	Returns a list of all threads that are not held.
<code>watchpoint(ptset)</code>	Returns a list of all threads that are stopped at a watch-point.

The way in which you specify the P/T set argument is the same as the way that you specify a P/T set for the **dfocus** command. For example, **watchpoint(L)** returns all threads in the current lockstep group. The only operator that differs is **comm**, whose argument is a process.

The dot operator (.), which indicates the current set, can be helpful when you are editing an existing set.

The following examples clarify how you use these operators and functions. The P/T set **a** (all) is the argument to these operators.

f {breakpoint(a) | watchpoint(a)} dstatus

Shows information about all threads that stopped at breakpoints and watchpoints. The **a** argument is the standard P/T set indicator for **all**.

f {stopped(a) - breakpoint(a)} dstatus

Shows information about all stopped threads that are not stopped at breakpoints.

f {. | breakpoint(a)} dstatus

Shows information about all threads in the current set, as well as all threads stopped at a breakpoint.

f {g.3 - p6} duntil 577

Runs thread 3 along with all other processes in the group to line 577. However, it does not run anything in process 6.

f {(\$PTSET) & p123}

Uses just process 123 in the current P/T set.

Creating Custom Groups

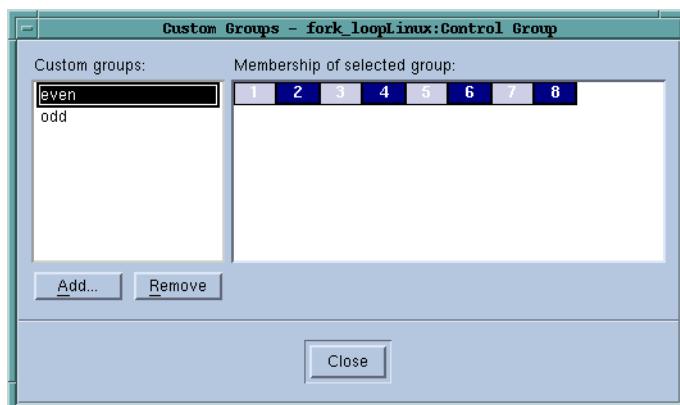
Debugging a multi-process or multi-threaded program most often focuses on running the program in one of two ways: either you run everything or run one or two things. Figuring out what you should be running, however, is a substantial part of the art of debugging. You can make things easier on yourself if you divide your program into groups, and then control these groups separately. When you need to do this, use the **Groups > Custom Groups** Dialog Box. (See Figure 164 on page 277.) This dialog box also lets you alter a group's contents as well as delete the group.



You can only manage process groups with this dialog box. Thread groups can only be managed using the CLI. In addition, the groups you create must reside within one control group.

When you first display this dialog box, TotalView also displays a second dialog box. Use this dialog box to enter the group's name.

Figure 164: Group > Custom Groups Dialog Box



The right side of this dialog box contains a box. Each represents one of your processes. The initial color represents the process's state. (This just helps you coordinate within the display in the Process Window's Processes/Ranks Tab.) You can now create a group using your mouse by clicking on blocks as follows:

- **Left-click on a box:** Selects a box. No other box is selected. If other boxes are selected, they will no longer be selected.
- **Shift-left-click and drag:** select a group of contiguous boxes.
- **Control-left-click on a box:** Adds a box to the current selection.

Edit an existing group in the same way. After making the group active by clicking on its name on the left, click within the right to make changes. (In most cases, you'll be using a control-left-click.)

If you've changed a group and then select **Add** or **Close**, TotalView asks if you want to save the changed group.

If you click **Add** when a group is selected, TotalView creates a group with the same members as that group.

Finally, you can delete a group by selecting its name on the left, then press the **Remove** button.

Examining and Changing Data



CHAPTER
14

This chapter explains how to examine and change data as you debug your program.

This chapter contains the following sections:

- “*Changing How Data is Displayed*” on page 279
- “*Displaying Variables*” on page 283
- “*Diving in Variable Windows*” on page 298
- “*Viewing a List of Variables*” on page 303
- “*Changing the Values of Variables*” on page 310
- “*Changing a Variable’s Data Type*” on page 311
- “*Changing the Address of Variables*” on page 321
- “*Displaying C++ Types*” on page 322
- “*Displaying Fortran Types*” on page 324
- “*Displaying Thread Objects*” on page 329
- “*Scoping and Symbol Names*” on page 330

This chapter does not discuss array data. For that information, see Chapter 15, “*Examining Arrays*,” on page 333.

Changing How Data is Displayed

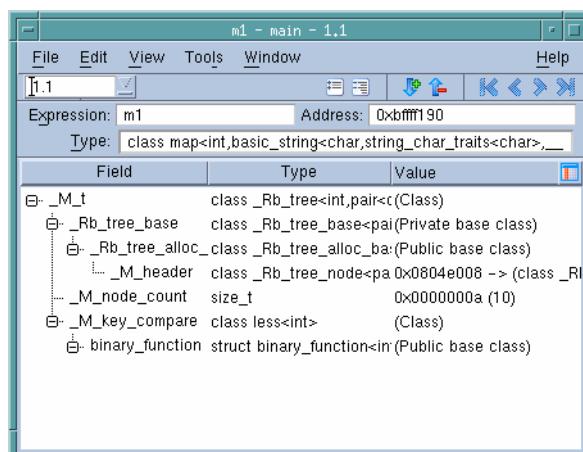
When a debugger displays a variable, it relies on the definitions of the data used by your compiler. The following two sections show how you can change the way TotalView displays this information:

- “*Displaying STL Variables*” on page 280
- “*Changing Size and Precision*” on page 282

Displaying STL Variables

The C++ STL (Standard Template Library) greatly simplifies the way in which you can access data. Since it offers standard and prepackaged ways to organize data, you do not have to be concerned with the mechanics of the access method. The disadvantage to using the STL while debugging is that the information debuggers display is organized according to the compiler's view of the data, rather than the STL's logical view. For example, here is how your compiler sees a map compiled using the GNU C++ compiler (gcc):

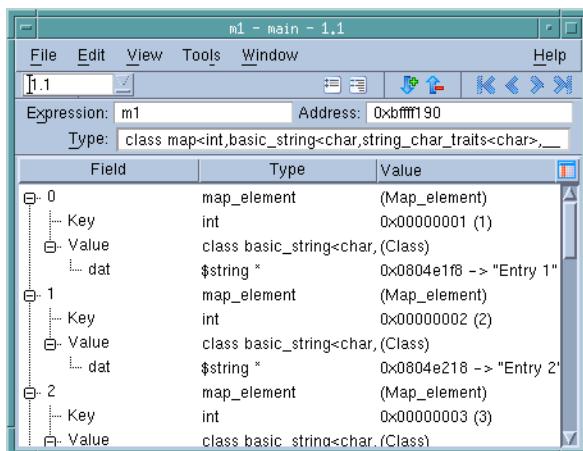
Figure 165: An Untransformed Map



Most of the information is generated by the STL template and, in most cases, is not interesting. In addition, the STL does not aggregate the information in a useful way.

STLView solves these problems by rearranging (that is, *transforming*) the data so that you can easily examine it. For example, here is the transformed map.

Figure 166: A Transformed Map

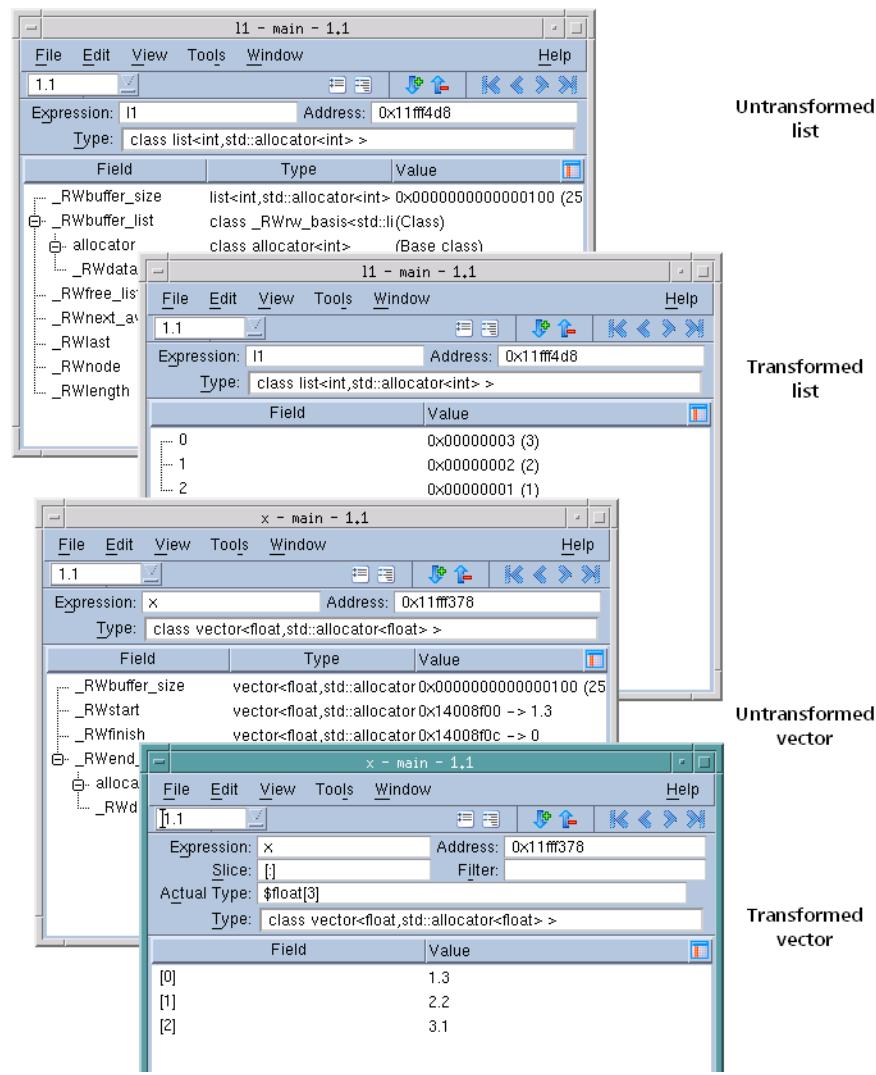


Using native and GCC compilers on IBM AIX, IRIX/MIPS, HP Tru64 Alpha, and Sun Solaris, TotalView can transform STL strings, vectors, lists, and

maps. TotalView can also transform these STL types if you are using GCC and Intel Version 7 and 8 C++ 32-bit compiler running on the Red Hat x86 platform. The *TotalView Platforms and System Requirements Guide* names the compilers for which TotalView transforms STL data types.

The following figure shows an untransformed and transformed list and vector.

Figure 167: List and Vector Transformations



You can create transformations for other STL containers. See the "TotalView Reference Guide" for more information.

By default, TotalView transforms STL types. If you need to look at the untransformed data structures, clear the **View simplified STL containers (and user-defined transformations)** checkbox on the Options Page of the **File > Preference** Dialog Box.

```
CLI: dset TV::ttf { true | false }
```

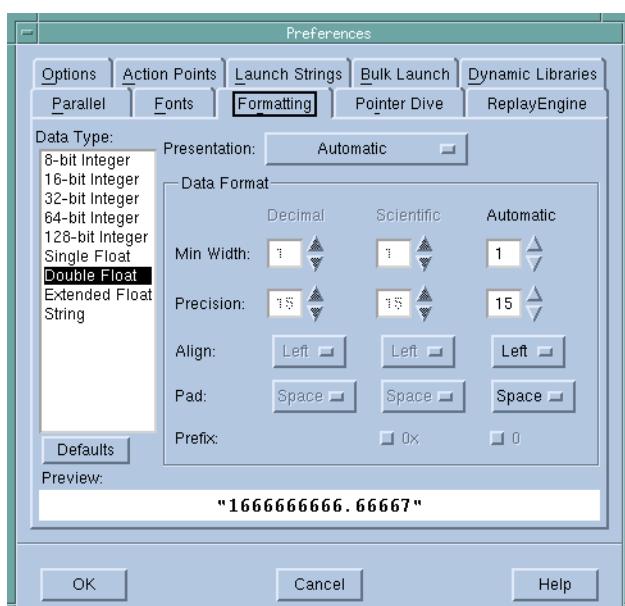
Changing How Data is Displayed

Following pointers in an STL data structure to retrieve values can be time-consuming. By default, TotalView only follows 500 pointers. You can change this by altering the value of the `TV::ttf_max_length` variable.

Changing Size and Precision

If the defaults formats that TotalView uses to display a variable's value doesn't meet your needs, you can use the **Formatting** Page of the **File > Preferences** Dialog Box to indicate how precise you want the simple data types to be.

Figure 168: File > Preferences
Formatting Page



After selecting one of the data types listed on the left side of the Formatting Page, you can set how many character positions a value uses when TotalView displays it (**Min Width**) and how many numbers to display to the right of the decimal place (**Precision**). You can also tell TotalView how to align the value in the **Min Width** area, and if it should pad numbers with zeros or spaces.

Although the way in which these controls relate and interrelate may appear to be complex, the **Preview** area shows you the result of a change. Play with the controls for a minute or so to see what each control does. You may need to set the **Min Width** value to a larger number than you need it to be to see the results of a change. For example, if the **Min Width** value doesn't allow a number to be justified, it could appear that nothing is happening.

CLI: You can set these properties from within the CLI. To obtain a list of variables that you can set, type "dset TV::data_format*".

Displaying Variables

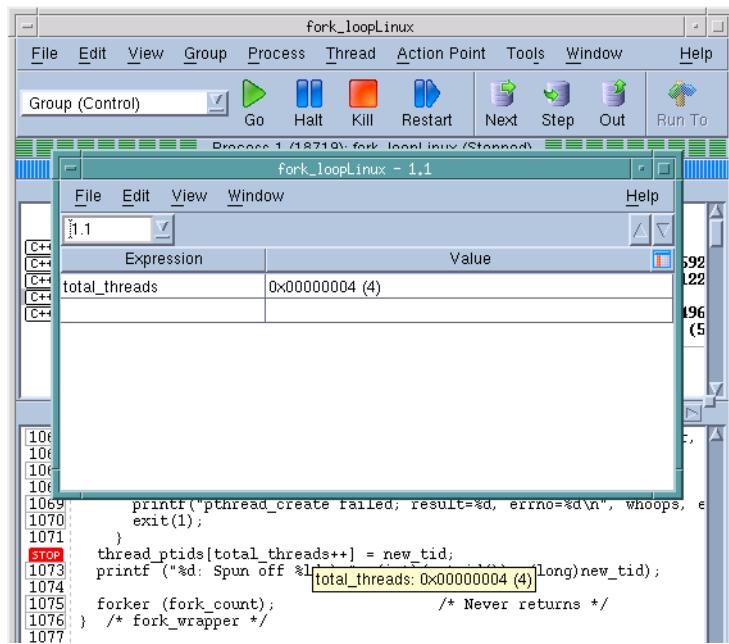
The Process Window Stack Frame Pane displays variables that are local to the current stack frame. This pane doesn't show the data for nonsimple variables, such as pointers, arrays, and structures. To see this information, you need to dive on the variable. This tells TotalView to display a Variable Window that contains the variable's data. For example, diving on an array variable tells TotalView to display the entire contents of the array.



Dive on a variable by clicking your middle mouse button on it. If your mouse doesn't have three buttons, you can single- or double-click on an item.

If you place your mouse cursor over a variable or an expression, TotalView displays its value in a tooltip window.

Figure 169: A Tooltip



If TotalView cannot evaluate what you place your mouse over, it will display some information. For example, if you place the mouse over a structure, the tooltip tells you the kind of structure. In all cases, what you see is similar to what you'd see if you placed the same information within the **Expression List** Window.

If you dive on simple variables or registers, TotalView still brings up a Variable Window; however, you will see some additional information about the variable or register.

Although a Variable Window is the best way to see all of an array's elements or all elements in a structure, using the **Expression List** Window is easier for variables with one value. Using it also cuts down on the number of windows that are open at any one time. For more information, see "Viewing a List of Variables" on page 303.

The following sections discuss how you can display variable information:

- "Displaying Program Variables" on page 284
- "Seeing Value Changes" on page 285
- "Displaying Variables in the Current Block" on page 287
- "Viewing Variables in Different Scopes as Program Executes" on page 288
- "Scoping Issues" on page 289
- "Browsing for Variables" on page 291
- "Displaying Local Variables and Registers" on page 292
- "Dereferencing Variables Automatically" on page 293
- "Displaying Areas of Memory" on page 296
- "Displaying Machine Instructions" on page 297
- "Rebinding the Variable Window" on page 298
- "Closing Variable Windows" on page 298

Displaying Program Variables

You can display local and global variables by:

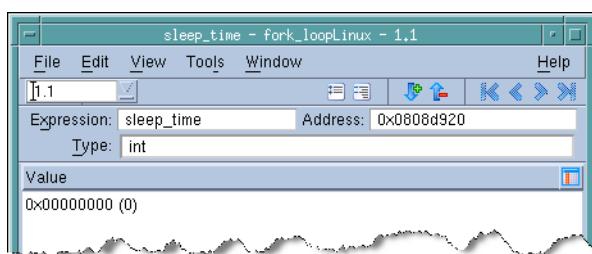
- Diving into the variable in the Source or Stack Panes.
- Selecting the **View > Lookup Variable** command. When prompted, enter the name of the variable.

CLI: `dprint variable`

- Using the **Tools > Program Browser** command.

After using one of these methods, TotalView displays a Variable Window that contains the information you want. The Variable Window can display simple variables, such as **ints**, sets of like elements such as arrays, or more complicated variables defined as structures and arrays of structures.

Figure 170: Variable Window for a Global Variable



If you keep a Variable Window open while a process or thread is running, the information being displayed might not be accurate. TotalView updates the window when the process or thread stops. If TotalView can't find a stack frame for a displayed local variable, the variable's status is **sparse**,

since the variable no longer exists. The **Status** area can contain other information that alerts you to issues and problems with a variable.

When you debug recursive code, TotalView doesn't automatically refocus a Variable Window onto different instances of a recursive function. If you have a breakpoint in a recursive function, you need to explicitly open a new Variable Window to see the local variable's value in that stack frame.

CLI: `dwhere`, `dup`, and `dprint`

Use `dwhere` to locate the stack frame, use `dup` to move to it, and then use `dprint` to display the value.

Select the **View > Compilation Scope > Floating** command to tell TotalView that it can refocus a Variable Window on different instances. For more information, see "Viewing Variables in Different Scopes as Program Executes" on page 288.

Controlling the Information Being Displayed

TotalView can display more information about your variable than its value. This information is sometimes called *meta-information*. You can control how much of this meta-information it displays by clicking on the **More** and **Less** buttons. (See Figure 171 on page 286.)

As the button names indicate, clicking **More** displays more meta-information and clicking **Less** displays less of it.

The two most useful fields are **Type**, which shows you what your variable's actual type is, and **Expression**, which allows you to control what is being displayed. This is sometimes needed because TotalView tries to show the type in the way that it thinks you declared it in your program.

The online help describes all the meta-information fields.

Seeing Value Changes

TotalView can tell you when a variable's value changes in several ways.

- When your program stops at a breakpoint, TotalView adds a yellow highlight to the variable's value if it has changed. This is shown in Figure 172 on page 286.

If the thread is stopped for another reason—for example, you've stepped the thread—and the value has changed, TotalView does not add yellow highlighting to the line.

- You can tell TotalView to display the **Last Value** column. Do this by selecting **Last Value** in the column menu, which is displayed after you click on the column menu () icon. (See Figure 173 on page 287.)

Notice that TotalView has highlighted all items that have changed within an array. In a similar fashion it can show the individual items that have changed within a structure.

In general, TotalView only retains the value for data items displayed within the Variable Window. At times, TotalView may track adjacent values within

Displaying Variables

Figure 171: Variable Window:
Using More and Less

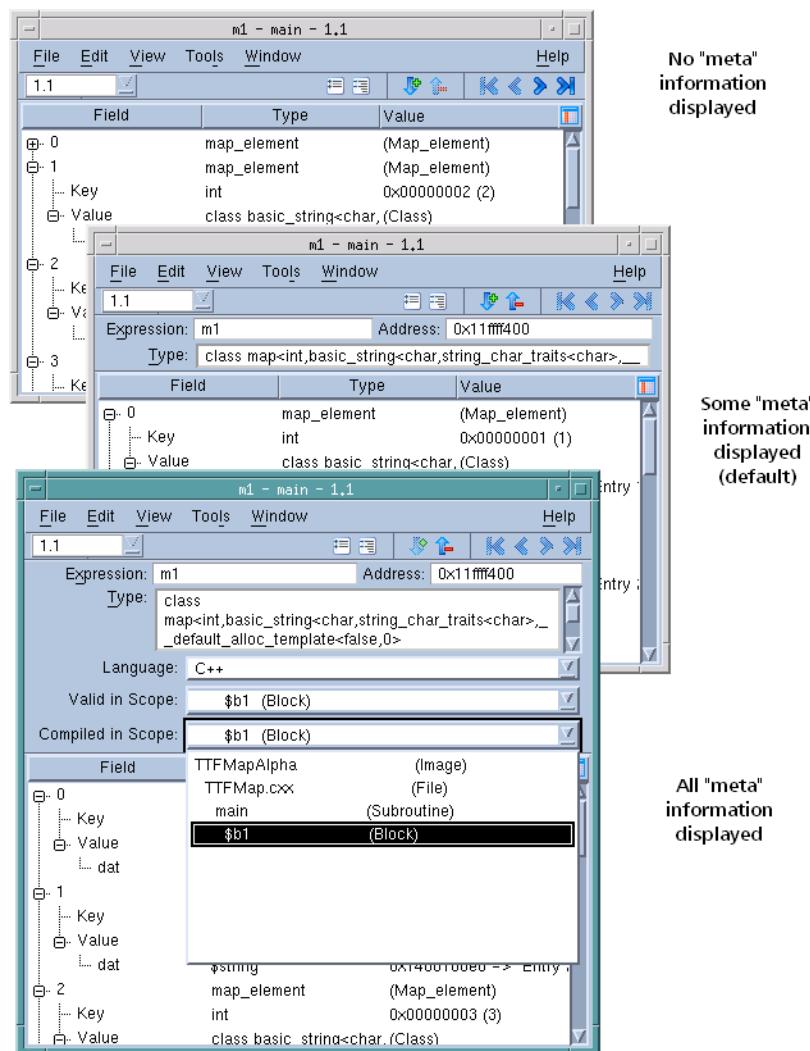
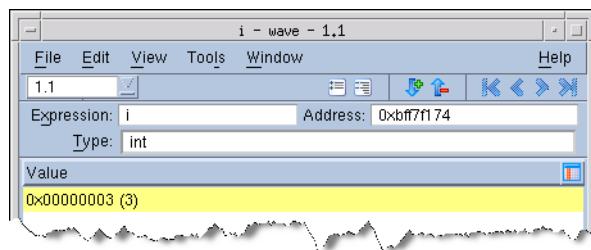


Figure 172: Variable Window
With "Change"
Highlighting

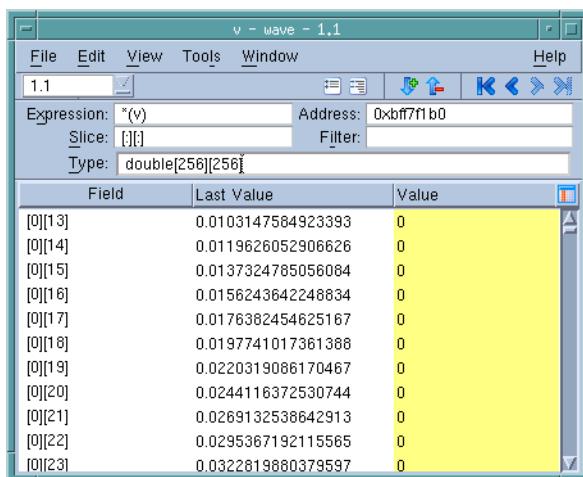


arrays and structures, but you should not rely on additional items being tracked.



When you scroll the Variable Window, TotalView discards the information it is tracking and fetches new information. So, while the values may have changed, TotalView does not have information about this change. That is, TotalView only tracks what it is visible. Similarly, when you scroll back to previously displayed values, TotalView needs to refetch this information. Because it is "new" information, no "last values" exist.

Figure 173: Variable Window
Showing Last Value Column



The screenshot shows the TotalView Expression List window titled "v - wave - 1.1". The window has a menu bar with File, Edit, View, Tools, Window, Help, and a toolbar with various icons. The main area displays a table with three columns: Field, Last Value, and Value. The table contains 11 rows, each representing an element from index [0][13] to [0][23]. The "Last Value" column shows non-zero values for indices [0][13] through [0][20], while the "Value" column shows zeros for all entries.

Field	Last Value	Value
[0][13]	0.0103147584923393	0
[0][14]	0.0119626052906626	0
[0][15]	0.0137324785056084	0
[0][16]	0.0156243642248834	0
[0][17]	0.0176382454625167	0
[0][18]	0.0197741017361368	0
[0][19]	0.0220319086170467	0
[0][20]	0.0244116372530744	0
[0][21]	0.0269132538642913	0
[0][22]	0.0295367192115565	0
[0][23]	0.0322819880379597	0

The Expression List window, described in “*Viewing a List of Variables*” on page 303, also highlights data and can display a **Last Value** column.

Seeing Structure Information

When TotalView displays a Variable Window, it displays structures in a compact form, concealing the elements within the structure. Click the + button to display these elements, or select the **View > Expand All** command to see all entries. If you want to return the display to a more compact form, you can click the – button to collapse one structure, or select the **View > Collapse All** command to return the window to what it was when you first opened it.

If a substructure contains more than about forty elements, TotalView does not let you expand it in line. That is, it does not place a + symbol in front of the substructure. To see the contents of this substructure, dive on it.

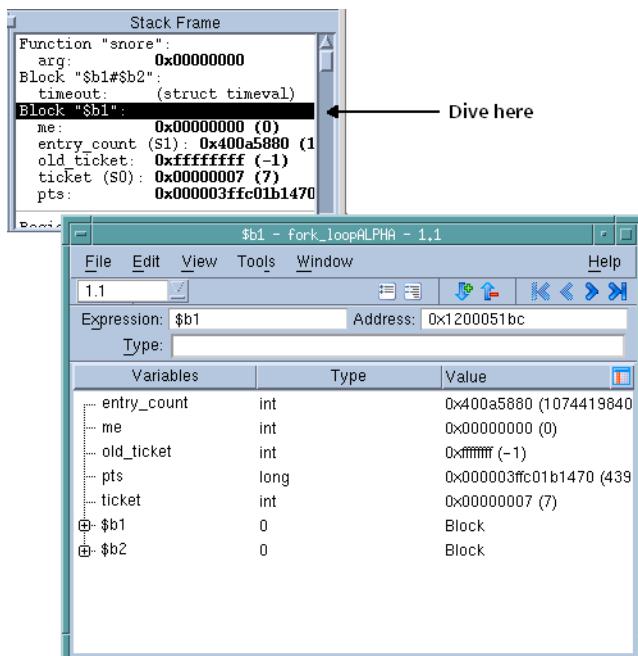
Similarly, if a structure contains an array as an element, TotalView only displays the array within the structure if it has less than about forty elements. To see the contents of an embedded array, dive on it.

Displaying Variables in the Current Block

In many cases, you may want to see all of the variables in the current block. If you dive on a block label in the Stack Frame Pane, TotalView opens a Variable Window that contains just those variables. (See Figure 174 on page 288.)

After you dive on a variable in this block window, TotalView displays a Variable Window for that scoped variable. In this figure, block \$b1 has two nested blocks.

Figure 174: Displaying Scoped Variables



Viewing Variables in Different Scopes as Program Executes

When TotalView displays a Variable Window, it understands the scope in which the variable exists. As your program executes, this scope doesn't change. In other words, if you're looking at variable **my_var** in one routine, and you then execute your program until it is within a second subroutine that also has a **my_var** variable, TotalView does not change the scope so that you are seeing the *in scope* variable.

If you would like TotalView to update a variable's scope as your program executes, select the **View > Compilation Scope > Floating** command. This tells TotalView that, when execution stops, it should look for the variable in the current scope. If it finds the variable, it displays the variable contained within the current scope.

Select the **View > Compilation Scope > Fixed** command to return TotalView to its default behavior, which is not to change the scope.

Selecting floating scope can be very handy when you are debugging recursive routines or have routines with identical names. For example, **i**, **j**, and **k** are popular names for counter variables.

Scoping Issues

When you dive into a variable from the Source Pane, the scope that TotalView uses is that associated with the current frame's PC; for example:

```
1: void f()
2: {
3:     int x;
4: }
5:
6: int main()
7: {
8:     int x;
9: }
```

If the PC is at line 3, which is in **f()**, and you dive on the **x** contained in **main()**, TotalView displays the value for the **x** in **f()**, not the **x** in **main()**. In this example, the difference is clear: TotalView chooses the PC's scope instead of the scope at the place where you dove. If you are working with templated and overloaded code, determining the scope can be impossible, since the compiler does not retain sufficient information. In all cases, you can click the **More** button within the Variable window to see more information about your variable. The **Valid in Scope** field can help you determine which instance of a variable you located.

You can, of course, use the **View > Lookup Variable** command to locate the correct instance.

Freezing Variable Window Data

Whenever execution stops, TotalView updates the contents of Variable Windows. More precisely, TotalView reevaluates the data found with the Expression area. If you do not want this reevaluation to occur, use the Variable Window's **View > Freeze** command. This tells TotalView that it should not change the information that is displaying.

After you select this command, TotalView writes information into the window, letting you know that the data is frozen. (See Figure 175 on page 290.)

Selecting the **View > Freeze** command a second time tells TotalView that it should evaluate this window's expression whenever execution stops.

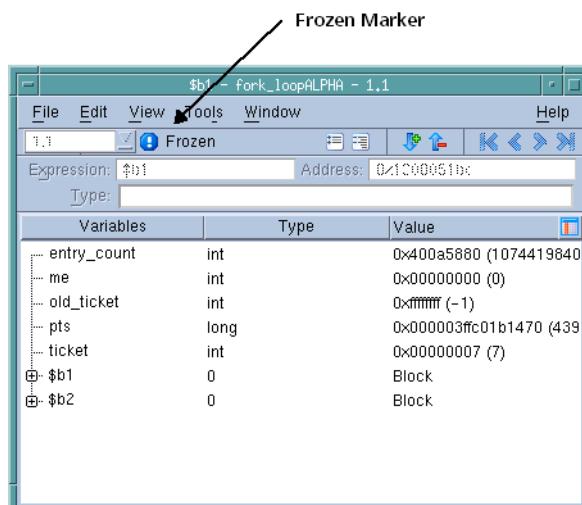
In most cases, you'll want to compare this information with an unfrozen copy. Do this by selecting the **Window > Duplicate** command before you freeze the display. As these two windows are identical, it doesn't matter which one you freeze. If you use the **Duplicate** command after you freeze the display, just select **View > Freeze** in one of the windows to get that window to update normally.

Locking the Address

The previous section discussed freezing the display so that TotalView does not update the Variable Window's contents. Sometimes you only want to freeze the address, not the data at that address. Do this by selecting the

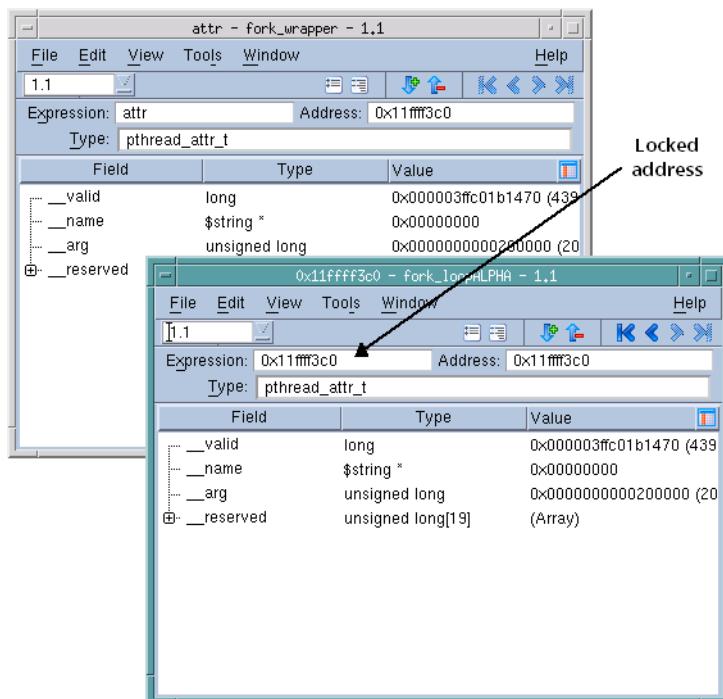
Displaying Variables

Figure 175: Variable Window Showing Frozen State



View > Lock Address command. Figure 176 shows two Variable Windows, one of which has had its address locked.

Figure 176: Locked and Unlocked Variable Windows



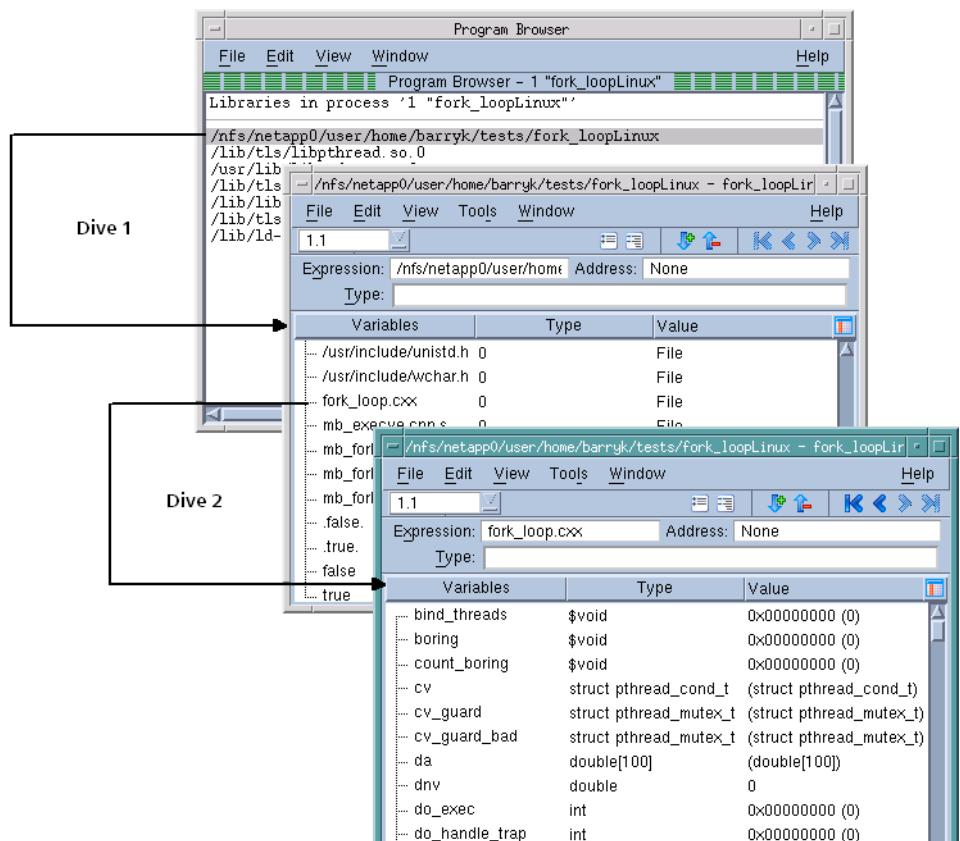
Using this command lets you continually reevaluate what is at that address as execution progresses. Here are two examples:

- If you need to look at a heap address access through a set of dive operations rooted in a stack frame that has become stale.
- If you dive on a ***this** pointer to see the actual value after ***this** goes stale.

Browsing for Variables

The Process Window Tools > Program Browser command displays a window that contains all your executable's components. By clicking on a library or program name, you can access all of the variables contained in it. (See Figure 177.)

Figure 177: Program Browser and Variable Windows (Part 1)

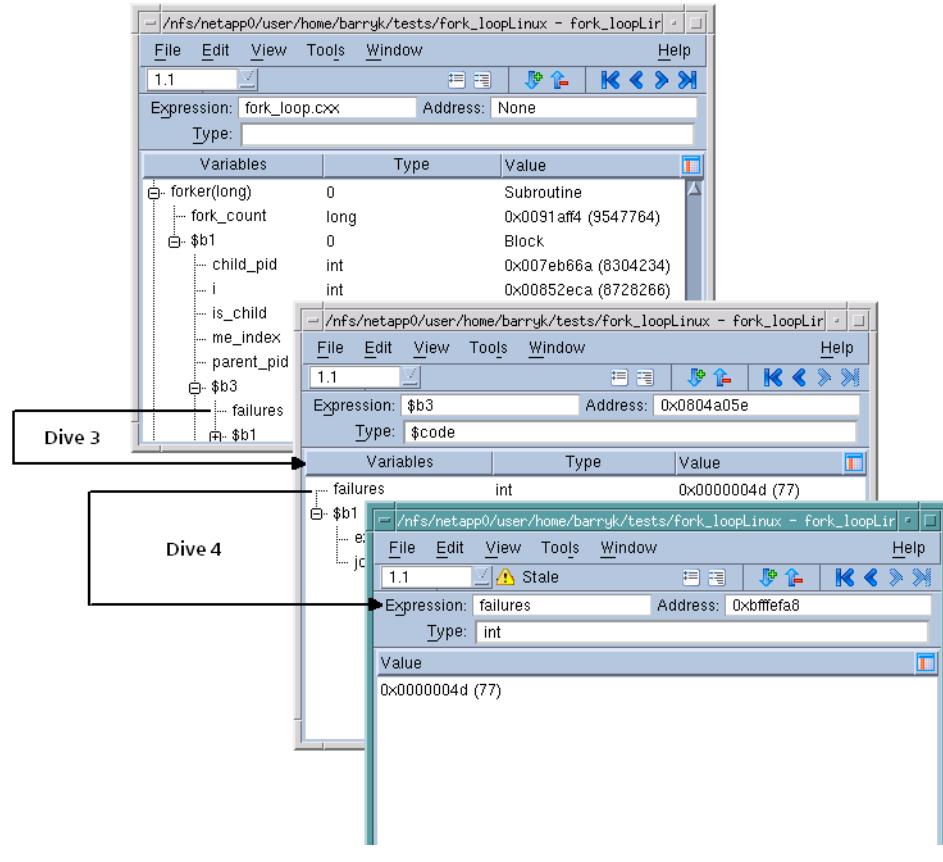


The window at the top of the figure shows programs and libraries that are loaded. If you have loaded more than one program with the **File > New Program** command, TotalView only displays information for the currently selected process. After diving on an entry in this window (labelled **Dive 1**), TotalView displays a Variable Window that contains a list of files that make up the program, as well as other related information.

Diving on an entry in this Variable Window (**Dive 2** in this figure) changes the display so that it contains variables and other information related to the file. A list of functions defined within the program is at the end of this list.

Diving on a function changes the Variable Window again. The window shown at the top of the next figure was created by diving on one of these functions. The window shown in the center is the result of diving on a block in that subroutine. The bottom window shows a variable. (See Figure 178 on page 292.)

Figure 178: Program Browser and Variable Window (Part 2)

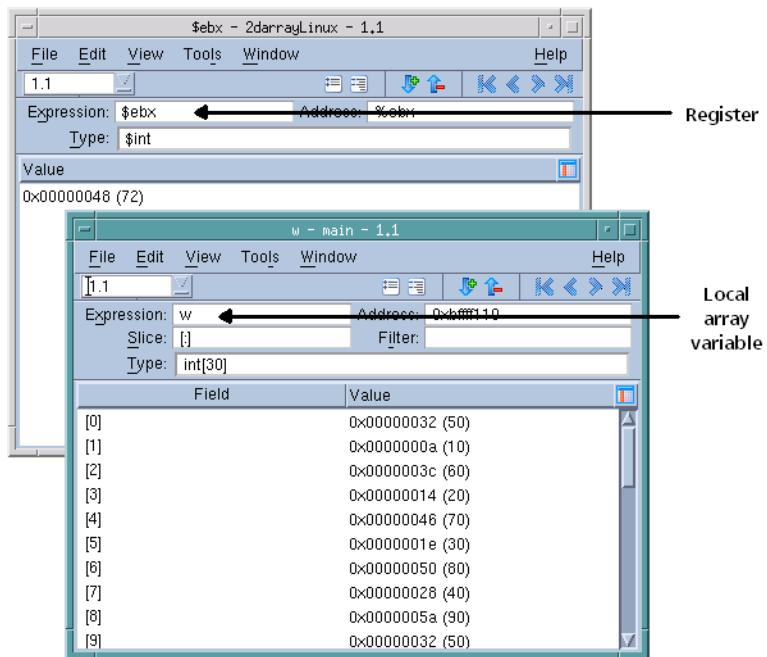


If you dive on a line in a Variable Window, the new contents replace the old contents, and you can use the undive/redive buttons to move back and forth.

Displaying Local Variables and Registers

In the Stack Frame Pane, diving on a function's parameter, local variable, or register tells TotalView to display information in a Variable Window. You can also dive on parameters and local variables in the Source Pane. The displayed Variable Window shows the name, address, data type, and value for the object. (See Figure 179 on page 293.)

Figure 179: Diving on Local Variables and Registers



The window at the top of the figure shows the result of diving on a register, while the bottom window shows the results of diving on an array variable.

CLI: `dprint variable`

This command lets you view variables and expressions without having to select or find them.

You can also display local variables by using the **View > Lookup Variable** command. After TotalView displays a dialog box, enter the name of the variable you want to see.

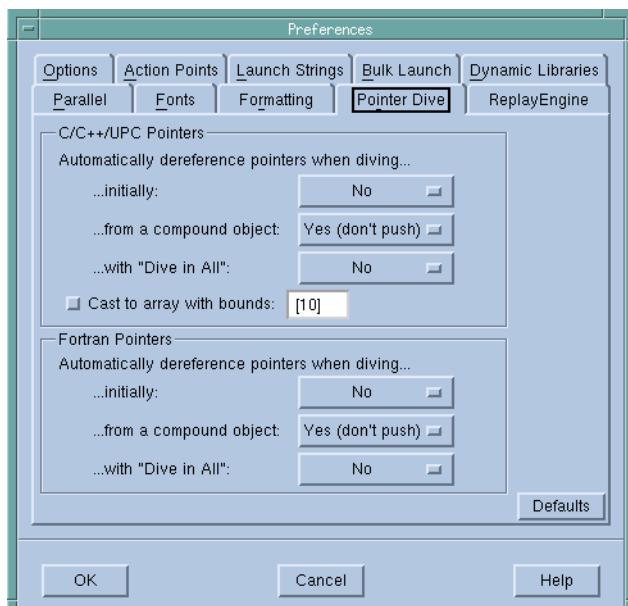
Dereferencing Variables Automatically

In most cases, you want to see what a pointer points to, rather than what the value of its variable is. Using the controls on the **File > Preferences** Pointer Dive Page (which is shown on the next page), you can tell TotalView to automatically dereference pointers. (See Figure 180 on page 294.)

Dereferencing pointers is especially useful when you want to visualize the data linked together with pointers, since it can present the data as a unified array. Because the data appears as a unified array, you can use TotalView's array manipulation commands and the Visualizer to view the data.

Each pulldown list on the Pointer Dive Page has three settings: **No**, **Yes**, and **Yes (don't push)**. The meaning for **No** is that automatic dereferencing does not occur. The remaining two values tell TotalView to automatically dereference pointers. The difference between the two is based on whether you can use the **Back** command to see the undereferenced pointer's value. If

Figure 180: File > Preferences
Pointer Dive Page



you choose **Yes**, you can see the value. If you choose **Yes (don't push)**, you cannot use the **Back** command to see this pointer's value.

```
CLI: TV::auto_array_cast_bounds
      TV::auto_deref_in_all_c
      TV::auto_deref_in_all_fortran
      TV::auto_deref_initial_c
      TV::auto_deref_initial_fortran
      TV::auto_deref_nested_c
      TV::auto_deref_nested_fortran
```

Automatic dereferencing can occur in the following situations:

- When TotalView *initially* displays a value.
- When you dive on a value in an aggregate or structure.
- When you use the **Dive in All** command.

Examining Memory

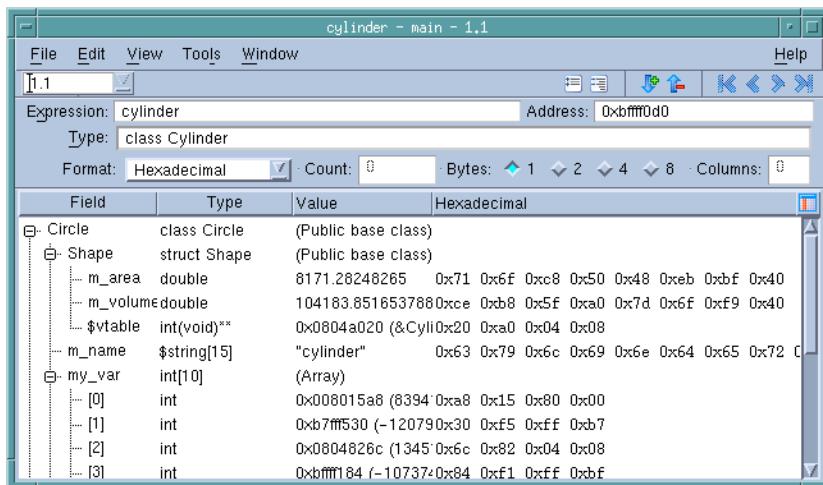
TotalView lets you display the memory used by a variable in different ways. If you select the **View > Examine Format > Structured** or **View > Examine Format > Raw** commands from within the Variable Window, TotalView displays raw memory contents. Figure 181 shows a structured view.



*The way this command displays data is similar to the way dump commands such as **od** that exist in your operating system display data.*

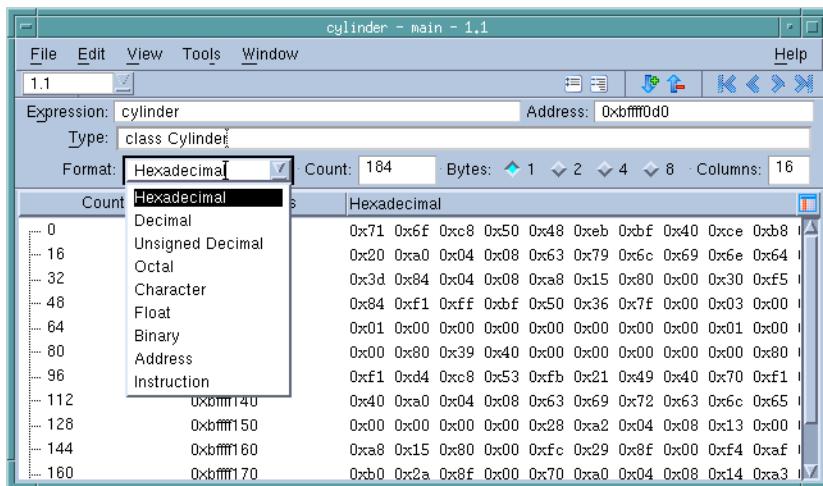
When displaying a structured view, the left portion of the Variable Window shows the elements of the data, whether it be a structure or an array. The right portion shows the value of the data in the way that it is normally displayed

Figure 181: View > Examine
Format > Structured Display



within TotalView. The right-most column displays the raw memory data. By default, this information is displayed in hexadecimal. However, you can change it to other formats by selecting a representation within the **Format** pulldown. The following figure shows a raw display with this pulldown extended:

Figure 182: View > Examine
Format > Raw Display



In either display, you can change the number of bytes grouped together and the amount of memory being displayed.

If you select the **View > Block Status** command, TotalView will also give you additional information about memory. For example, you are told if the memory is in a **text**, **data**, or **bss** section. (If you see **unknown**, you are probably seeing a stack variable.)

In addition, if you right-click on the header area of the table, a context menu lets you add a **Status** column. This column contains information such as "Allocated", "PostGuard", "Corrupted PreGuard", etc.

If you have enabled the Memory Debugger, this additional information includes letting you know if memory is allocated or deallocated or being used by a guard block or hoarded.

Displaying Areas of Memory

You can display areas of memory using hexadecimal, octal, or decimal values. Do this by selecting the **View > Lookup Variable** command, and then entering one of the following in the dialog box that appears:

■ An address

When you enter a single address, TotalView displays the word of data stored at that address.

CLI: `dprint address`

■ A pair of addresses

When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

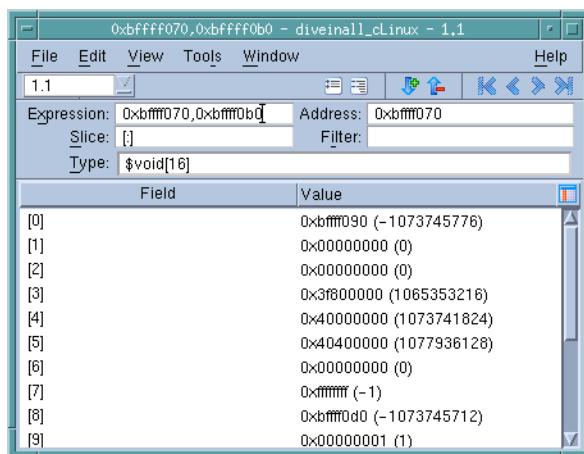
CLI: `dprint address,address`



All octal constants must begin with 0 (zero). Hexadecimal constants must begin with 0x.

The Variable Window for an area of memory displays the address and contents of each word. (See Figure 183.)

Figure 183: Variable Window for an Area of Memory



TotalView displays the memory area's starting location at the top of the window's data area. In the window, TotalView displays information in hexadecimal and decimal notation.

If a Variable Window is already being displayed, you can change the type to **\$void** and add an array specifier. If you do this, the results are similar to what is shown in this figure.

Changing Types to Display Machine Instructions

You can display machine instructions in a Variable Window by changing the text in the Variable Window **Type** field. All you need do is edit the type string to be an array of **\$code** data types. You also need to add an array specifier to tell TotalView how many instruction to display. For example, the following changes the Variable Window so that it displays three machine instructions:

\$code[3]

The Variable Window lists the following information about each machine instruction:

Offset+Label	The symbolic address of the location as a hexadecimal offset from a routine name.
Code	The hexadecimal value stored in the location.
Instruction	The instruction and operands stored in the location.

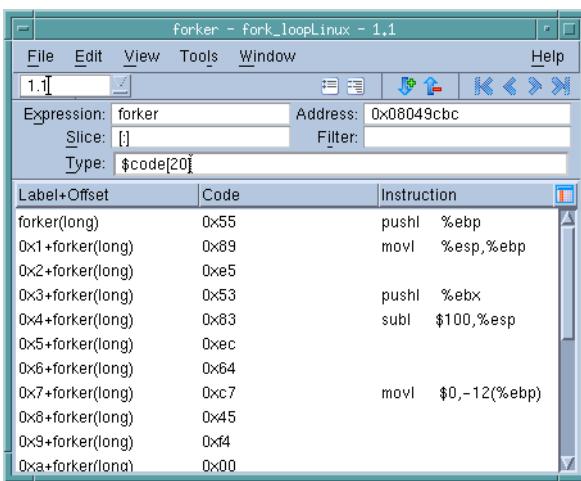
You can also edit the value listed in the **Value** field for each machine instruction.

Displaying Machine Instructions

You can display the machine instructions for entire routines as follows:

- Dive on the address of an assembler instruction in the Source Pane (such as **main+0x10** or **0x60**). A Variable Window displays the instructions for the entire function, and highlights the instruction you dove on.
- Dive on the PC in the Stack Frame Pane. A Variable Window displays the instructions for the entire function that contains the PC, and also highlights the instruction pointed to by the PC. (See Figure 184.)

Figure 184: Variable Window with Machine Instructions



- Cast a variable to type **\$code** or array of **\$code**.

Rebinding the Variable Window

When you restart your program, TotalView must identify the thread in which the variable existed. For example, suppose variable `my_var` was in thread 3.6. When you restart your program, TotalView tries to rebinding the thread to a newly created thread. Because the order in which the operating system starts and executes threads can differ, there's no guarantee that the thread 3.6 in the current context is the same thread as what it was previously. Problems can occur. To correct rebinding issues, use the **Threads** box in the Variable Window toolbar to specify the thread to which you want to bind the variable.

Another way to use the **Threads** box is to change to a different thread to see the variable or expression's value there. For example, suppose variable `my_var` is being displayed in thread 3.4. If you type 3.5 in the **Threads** box, TotalView updates the information in the **Expression List** Window so that it is what exists in thread 3.5.

Closing Variable Windows

When you finish analyzing the information in a Variable Window, use the **File > Close** command to close the window. You can also use the **File > Close Similar** command to close all Variable Windows.

Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive on the value. This new dive, which is called a *nested dive*, tells TotalView to replace the information in the Variable Window with information about the selected variable. If this information contains nonscalar data types, you can also dive on these data types. Although a typical data structure doesn't have too many levels, repeatedly diving on data lets you follow pointer chains. That is, diving lets you see the elements of a linked list.

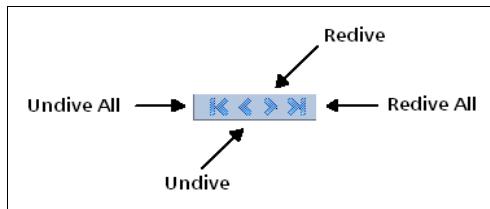
The following topics contain information related to this topic:

- "Displaying an Array of Structure's Elements" on page 300
- "Changing What the Variable Window Displays" on page 301

TotalView lets you see a member of an array of structures as a single array across all the structures. See "Displaying an Array of Structure's Elements" on page 300 for more information.

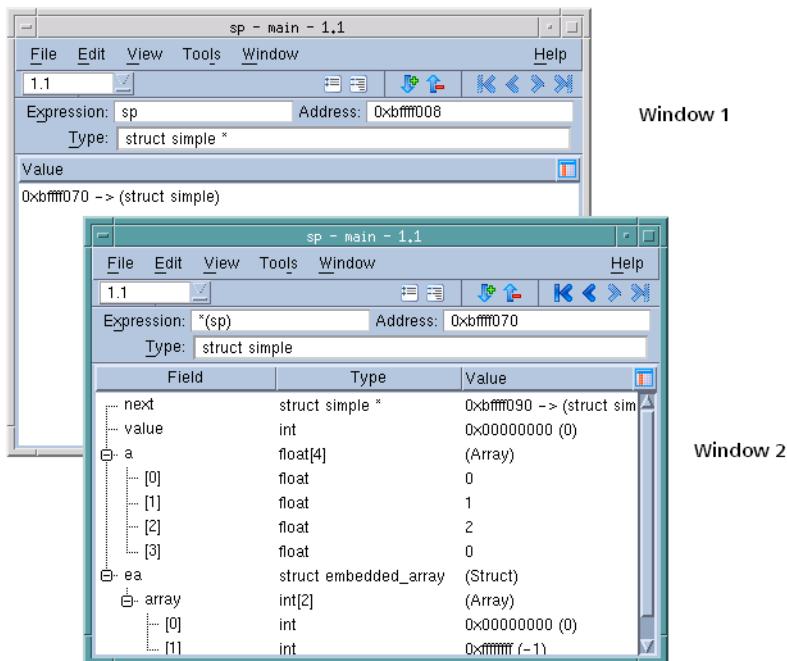
TotalView remembers your dives. This means that you can use the undive/redive buttons to view where you already dove. (See Figure 185 on page 299.)

Figure 185: Undive/Redive Buttons



The following figure shows a Variable Window after diving into a pointer variable named `sp` with a type of `simple*`. The first Variable Window, which is called the *base window*, displays the value of `sp`. (This is **Window 1** in Figure 186.)

Figure 186: Nested Dives



The nested dive window, (**Window 2** in this figure) shows the structure referenced by the `simple*` pointer.

You can manipulate Variable Windows and nested dive windows by using the undive/redive buttons, as follows:

- To undive from a nested dive, click the undive arrow button. The previous contents of the Variable Window appear.
- To undive from all your dive operations, click the undive all arrow button.
- To redive after you undive, click the redive arrow button. TotalView restores a previously executed dive operation.
- To redive from all your undive operations, click on the **Redive All** arrow button.
- If you dive on a variable that already has a Variable Window open, the Variable Window pops to the top of the window display.

- If you select the **Window > Duplicate** command, a new Variable Window appears, which is a duplicate of the current Variable Window.

Displaying an Array of Structure's Elements

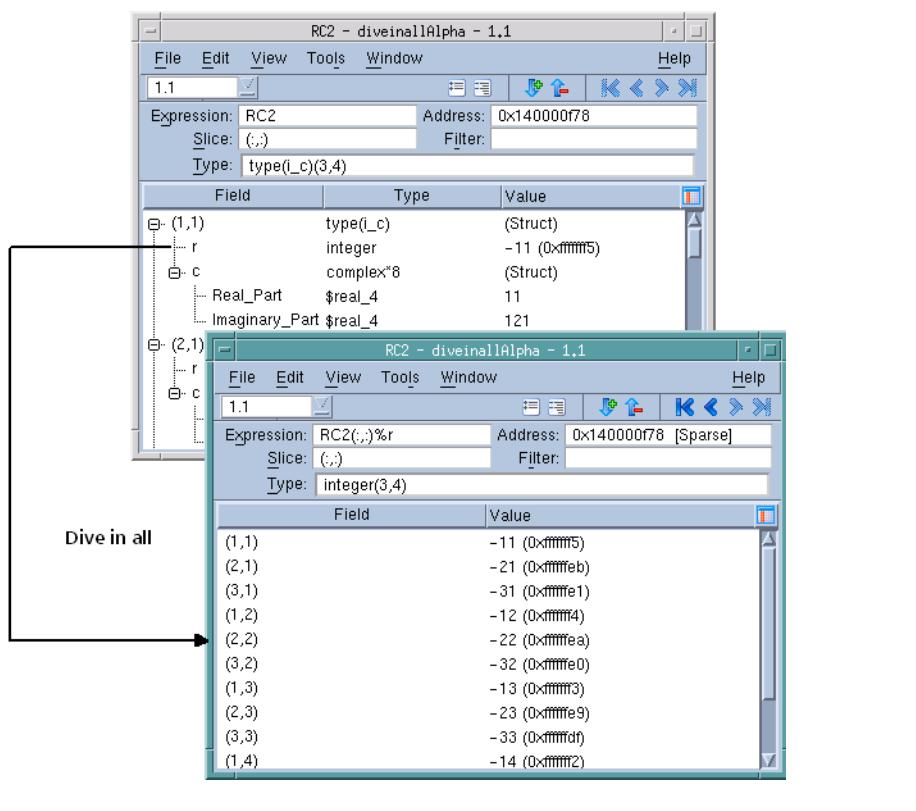
The **View > Dive In All** command, which is also available when you right-click on a field, lets you display an element in an array of structures as if it were a simple array. For example, suppose you have the following Fortran definition:

```
type i_c
    integer r
    complex c
end type i_c

type(i_c), target :: rc2(3,4)
```

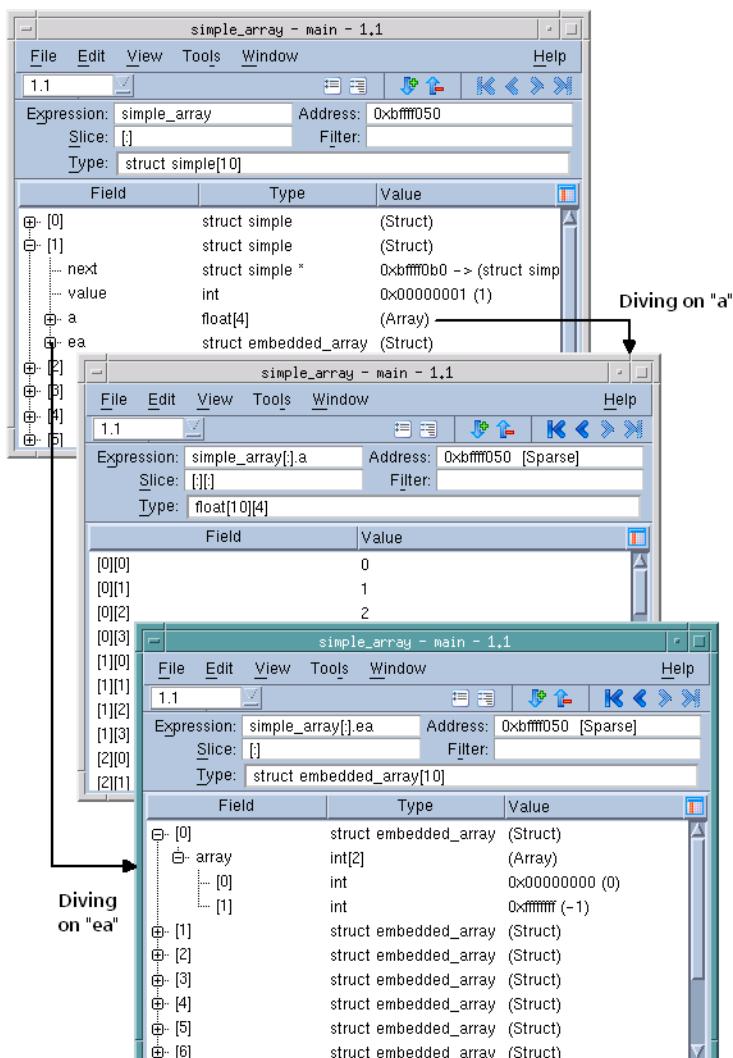
After selecting an **r** element, select the **View > Dive In All** command. TotalView displays all of the **r** elements of the **rc2** array as if they were a single array. (See Figure 187.)

Figure 187: Displaying a Fortran Structure



The **View > Dive in All** command can also display the elements of a C array of structures as arrays. Figure 188 on page 301 shows a unified array of structures and a multidimensional array in a structure.

Figure 188: Displaying C Structures and Arrays



As the array manipulation commands (described in Chapter 8) generally work on what's displayed and not what is stored in memory, TotalView commands that refine and display array information work on this virtual array. For example, you can visualize the array, obtain statistics about it, filter elements in it, and so on.

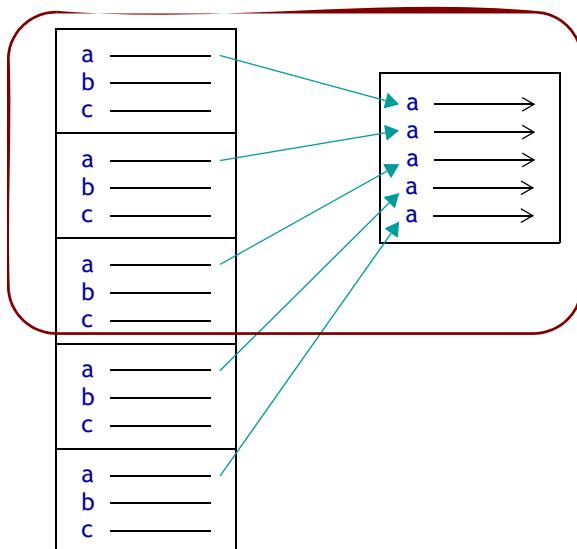
Figure 189 on page 302 is a high-level look at what a dive in all operation does.

In this figure, the rounded rectangle represents a Variable Window. On the left is an array of five structures. After you select the **Dive in All** command with element **a** selected, TotalView replaces the contents of your Variable Window with an array that contains all of these **a** elements.

Changing What the Variable Window Displays

When TotalView displays a Variable Window, the **Expression** field contains either a variable or an expression. Technically, a variable is also an expression. For example, `my_var.an_element` is actually an addressing expression.

Figure 189: Dive in All



Similarly, `my_var.an_element[10]` and `my_var[10].an_element` are also expressions, since both TotalView and your program must figure out where the data associated with the element resides.

The expression in the **Expression** field is dynamic. That is, you can tell TotalView to evaluate what you enter before trying to obtain a memory address. For example, if you enter `my_var.an_element[i]`, TotalView evaluates the value of `i` before it redisplays your information. A more complicated example is `my_var.an_element[i+1]`. In this example, TotalView must use its internal expression evaluation system to create a value before it retrieves data values.

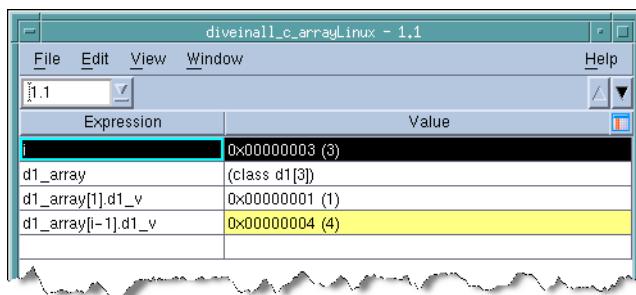
You can replace the variable expression with something completely different, such as `i+1`, and TotalView simply displays the value produced by evaluating the expression.

Chapter 17, "Evaluating Expressions," on page 381 has a discussion of the evaluation system and typing expressions in an eval point in the **Tools > Evaluate** Window. In contrast, the expressions you can type in the **Expression List** Window are restricted with the principal restriction being that what you type cannot have side effects. For example, you cannot use an expression that contains a function call or an operator that changes memory, such as `++` or `--`.

Viewing a List of Variables

As you debug your program, you may want to monitor a variable's value as your program executes. For many types of information, the **Expression List** Window offers a more compact display than the Variable Window for displaying scalar variables. (See Figure 190.)

Figure 190: The Tools > Expression List Window



The screenshot shows the TotalView Expression List Window titled "diveinall_c_arrayLinux - 1,1". The window has a menu bar with File, Edit, View, Window, Help, and a Threads dropdown set to "1,1". Below the menu is a toolbar with a search icon. The main area is a table with two columns: "Expression" and "Value". The table contains the following data:

Expression	Value
d1_array	0x00000003 (3)
d1_array[1].d1_v	0x00000001 (1)
d1_array[i-1].d1_v	0x00000004 (4)

The topics discussing the **Expression List** Window are:

- "Entering Variables and Expressions" on page 303
- "Entering Expressions into the Expression Column" on page 305
- "Using the Expression List with Multi-process/Multi-threaded Programs" on page 307
- "Reevaluating, Reopening, Rebinding, and Restarting" on page 307
- "Seeing More Information" on page 308
- "Sorting, Reordering, and Editing" on page 309

Entering Variables and Expressions

You can place information in the first column of the **Expression List** Window in the following ways:

- Type information into a blank cell in the **Expression** column. When you do this, the context for what you are typing is the current PC in the process and thread indicated in the **Threads** box. If you type **my_var** in the window shown in the previous section, you would type the value of **my_var** in process 1, thread 1.
- Right-click on a line in the Process Window Source or Stack Frame Panes. From the displayed context menu, select **Add to Expression List**. The following figure shows the context menu that TotalView displays in the Source Pane: (See Figure 191 on page 304.)
- Right-click on something in a Variable Window. Select **Add to Expression List** from the displayed context menu. You can also use the **View > Add to Expression List** command.

Viewing a List of Variables

Figure 191: A Context Menu



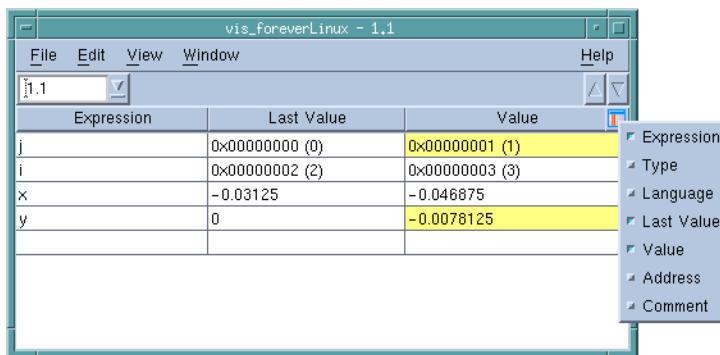
You can bring up this window directly by using the **Tools > Expression List** command.

When you enter information in the **Tools > Expression List Window**, where you place the cursor and what you select make a difference. If you click on a variable or select a row in the Variable Window, TotalView adds that variable to the Expression List Window. If you instead select text, TotalView adds that text. What's the difference? The Expression List figure in the previous section shows three variations of **d1_array**, and each was obtained in a different way, as follows:

- The first entry was added by just selecting part of what was displayed in the Source Pane.
- The second entry was added by selecting a row in the Variable Window.
- The third entry was added by clicking at a random point in the variable's text in the Source Pane.

You can tell TotalView to look for a variable in the scope that exists when your program stops executing, rather than keeping it locked to the scope from which it was added to the **Tools > Expression List Window**. Do this by right-clicking an item, then selecting **Compilation Scope > Floating** from the context menu.

Figure 192: Expression List Window Context Menu



For more information, see "Viewing Variables in Different Scopes as Program Executes" on page 288.

Seeing Variable Value Changes in the Expression List Window

TotalView can tell you when a variable's value changes in several ways.

- When your program stops at a breakpoint, TotalView adds a yellow highlight to the variable's value if it has changed. This is shown in Figure 193.

Figure 193: Expression List Window With "Change" Highlighting

The screenshot shows the TotalView Expression List window titled "vis_foreverLinux - 1,1". The window has a menu bar with File, Edit, View, Window, and Help. A toolbar with icons for file operations is visible above the main table. The table has three columns: Expression, Last Value, and Value. The rows show variables j, i, x, and y. The "Value" column contains the actual values, while the "Last Value" column is empty. The "Value" column for variable i is highlighted with a yellow background, indicating a change.

Expression	Last Value	Value
j		0x000000100 (256)
i		0x000000001 (1)
x		0
y		(Stale) -1.9921875

If the thread is stopped for another reason—for example, you've stepped the thread—and the value has changed, TotalView does not add yellow highlighting to the line.

- You can tell TotalView to display the **Last Value** column. Do this by selecting **Last Value** in the column menu, which is displayed after you click on the column menu () icon. (See Figure 194 on page 305.)

Figure 194: Variable Window Showing Last Value Column

The screenshot shows the TotalView Variable window titled "vis_foreverLinux - 1,1". The window has a menu bar with File, Edit, View, Window, and Help. A toolbar with icons for file operations is visible above the main table. The table has three columns: Expression, Last Value, and Value. The rows show variables j, i, x, and y. The "Value" column contains the actual values, while the "Last Value" column is empty. The "Value" column for variable i is highlighted with a yellow background, indicating a change.

Expression	Last Value	Value
j	0x000000000 (0)	0x000000001 (1)
i	0x000000002 (2)	0x000000003 (3)
x	-0.03125	-0.046875
y	0	-0.0078125

Notice that TotalView has highlighted all items that have changed within an array. In a similar fashion it can show the individual items that have changed within a structure.

Entering Expressions into the Expression Column

The simple answer is just about anything except function calls. (See "Entering Variables and Expressions" on page 303 for more information.) A variable is, after all, a type of expression. The following Expression List Window shows four different types of expressions. (See Figure 195 on page 306.)

Viewing a List of Variables

Figure 195: The Tools > Expression List Window

The screenshot shows the TotalView Expression List Window titled "diveinall_c_arrayLinux - 1.1". The window has a menu bar with File, Edit, View, Window, and Help. A toolbar with a search icon is at the top. The main area is a table with two columns: Expression and Value.

Expression	Value
i	0x00000003 (3)
d1_array	(class d1[3])
d1_array[1].d1_v	0x00000001 (1)
d1_array[i-1].d1_v	0x00000004 (4)

The expressions in this window are:

i A variable with one value. The **Value** column shows its value.

d1_array An aggregate variable; that is, an array, a structure, a class, and so on. Its value cannot be displayed in one line. Consequently, TotalView just gives you some information about the variable. To see more information, dive on the variable. After diving, TotalView displays the variable in a Variable Window.

When you place an aggregate variable in the **Expression** column, you need to dive on it to get more information.

d1_array[1].d1_v An element in an array of structures. If TotalView can resolve what you enter in the **Expression** column into a single value, it displays a value in the **Value** column. If TotalView can't, it displays information in the same way that it displays information in the **d1_array** example.

d1_array[i-1].d1_v An element in an array of structures. This expression differs from the previous example in that the array index is an expression. Whenever execution stops in the current thread, TotalView reevaluates the **i-1** expression. This means that TotalView might display the value of a different array item every time execution stops.

The expressions you enter cannot include function calls.

You can also enter methods and functions within an Expression. Figure 196 shows two get methods and a get method used in an expression.

Figure 196: Using Methods in the Tools > Expression List Window

The screenshot shows the TotalView Expression List Window titled "tx_cpp_stmt - 1.1". The window has a menu bar with File, Edit, View, Window, and Help. A toolbar with a search icon is at the top. The main area is a table with two columns: Expression and Value.

Expression	Value
x.get_memb1()	0x00003039 (12345)
x.get_memb2()	false (0)
x.get_memb1()*300	0x003882cc (3703500)

In a similar fashion, you can even directly enter functions. (See Figure 197.)

Figure 197: Using Functions in the Tools > Expression List Window

Expression	Value
nothing2(5)	0x0000000f (15)
update_j(10, 12)	0x0000000a (10)
get_random_time(400)	0x00000017a (378)

Using the Expression List with Multi-process/Multi-threaded Programs

You can change the thread in which TotalView evaluates your expressions by typing a new thread value in the **Threads** box at the top of the window. A second method is to select a value by using the drop-down list in the **Threads** box.

When you use an **Add to Expression List** command, TotalView checks whether an Expression List Window is already open for the current thread. If one is open, TotalView adds the variable to the bottom of the list. If an Expression List Window isn't associated with the thread, TotalView duplicates an existing window, changes the thread of the duplicated window, and then adds the variable to all open **Tools > Expression List** Windows. That is, you have two **Tools > Expression List** Windows. Each has the same list of expressions. However, the results of the expression evaluation differ because TotalView is evaluating them in different threads.

In all cases, the list of expressions in all **Tools > Expression List** Windows is the same. What differs is the thread in which TotalView evaluates the window's expressions.

Similarly, if TotalView is displaying two or more **Tools > Expression List** Windows, and you send a variable from yet another thread, TotalView adds the variable to all of them, duplicates one of them, and then changes the thread of the duplicated window.

Reevaluating, Reopening, Rebinding, and Restarting

This section explains what happens in the **Tools > Expression List** Window as TotalView performs various operations.

Reevaluating Contents: TotalView reevaluates the value of everything in the **Tools > Expression List** Window **Expression** column whenever your thread stops executing. More precisely, if a thread stops executing, TotalView reevaluates the contents of all **Tools > Expression List** Windows associated with the thread. In this way, you can see how the values of these expressions change as your program executes.

Viewing a List of Variables

You can use the **Window > Update All** command to update values in all other **Tools > Expression List** Windows.

Reopening Windows: If you close all open **Tools > Expression List** Windows and then reopen one, TotalView remembers the expressions you add. That is, if the window contains five variables when you close it, it has the same five variables when you open it. The thread TotalView uses to evaluate the window's contents is the Process Window from which you invoked the **Tools > Expressions List** command.

Rebinding Windows: The values displayed in an **Expression List** Window are the result of evaluating the expression in the thread indicated in the **Threads** box at the top of the window. To change the thread in which TotalView evaluates these expressions, you can either type a new thread value in the **Threads** box or select a thread from the pulldown list in the **Threads** box. (Changing the thread to evaluate expressions in that thread's context is called *rebinding*.)

Restarting a Program: When you restart your program, TotalView attempts to rebind the expressions in an **Tools > Expression List** Window to the *correct* thread. Unfortunately, it is not possible to select the right thread with 100% accuracy. For example, the order in which your operating system creates threads can differ each time you run your program. Or, program logic can cause threads to be opened in a different order.

You may need to manually change the thread by using the **Threads** box at the top of the window.

Seeing More Information

When you first open the **Tools > Expression List** Window, it contains two columns, but TotalView can display other columns. If you right-click on a column heading line, TotalView displays a context menu that indicates all possible columns. Clicking on a heading name listed in the context menu changes if from displayed to hidden or vice versa.

Figure 198: The **Tools > Expression List** Window
Showing Column Selector

The screenshot shows the 'vis_foreverLinux - 1.1' window with the title bar. The menu bar includes File, Edit, View, Window, and Help. A context menu is open over the column headers, listing options for Expression, Type, Language, Last Value, Value, Address, and Comment. The 'Value' option is currently selected. The main table has three columns: Expression, Last Value, and Value. The rows show variables j, i, x, and y with their corresponding values and last values.

Expression	Last Value	Value
j	0x00000000 (0)	0x00000001 (1)
i	0x00000002 (2)	0x00000003 (3)
x	-0.03125	-0.046875
y	0	-0.0078125

Even when you add additional columns, the **Expression List** Window might not show you what you need to know about a variable. If you dive on a row

(or select **Dive** from a context menu), TotalView opens a Variable Window for what you just dove on.

You can combine the **Expression List** Window and diving to bookmark your data. For example, you can enter the names of structures and arrays. When you want to see information about them, dive on the name. In this way, you don't have to clutter up your screen with the Variable Windows that you don't need to refer to often.

Sorting, Reordering, and Editing

This section describes operations you can perform on **Tools > Expression List** Window data.

Sorting Contents: You can sort the contents of the **Tools > Expression List** Window by clicking on the column header. After you click on the heading, TotalView adds an indicator that shows that the column was sorted and the way in which it was sorted. In the figure in the previous topic, the **Value** column is sorted in ascending order.

Reordering Row Display: The up and down arrows (on the right side of the **Tools > Expression List** Window toolbar let you change the order in which TotalView displays rows. For example, clicking the down arrow moves the currently selected row (indicated by the highlight) one row lower in the display.

Editing Expressions: You can change an expression by clicking in it, and then typing new characters and deleting others. Select **Edit > Reset Defaults** to remove all edits you make. When you edit an expression, TotalView uses the scope that existed when you created the variable.

Changing Data Type: You can edit an expression's data type by displaying the **Type** column and making your changes. Select **Edit > Reset Defaults** to remove all edits you make.

Changing an Expression's Value: You can change an expression's value if that value is stored in memory by editing the contents of the **Value** column.

About Other Commands: You can also use the following commands when working with expressions:

Edit > Delete Expression

Deletes the selected row. This command is also on a context (right-click) menu. If you have more than one **Expression List** Window open, deleting a row from one window deletes the row from all open windows.

Edit > Delete All Expressions

Deletes all of the **Expression List** Window rows. If you have more than one **Expression List** Window open, deleting all expressions from one window deletes all expressions in all windows.

View > Dive Displays the expression or variable in a Variable Window. Although this command is also on a context menu, you can just double-click or middle-click on the variable's name instead.

Edit > Duplicate Expression

Duplicates the selected column. You would duplicate a column to see a similar variable or expression. For example, if `myvar_looks_at[i]` is in the **Expression** column, duplicating it and then modifying the new row is an easy way to see `myvar_looks_at[i]` and `myvar_looks_at[i+j-k]` at the same time.

This command is also on a context menu.

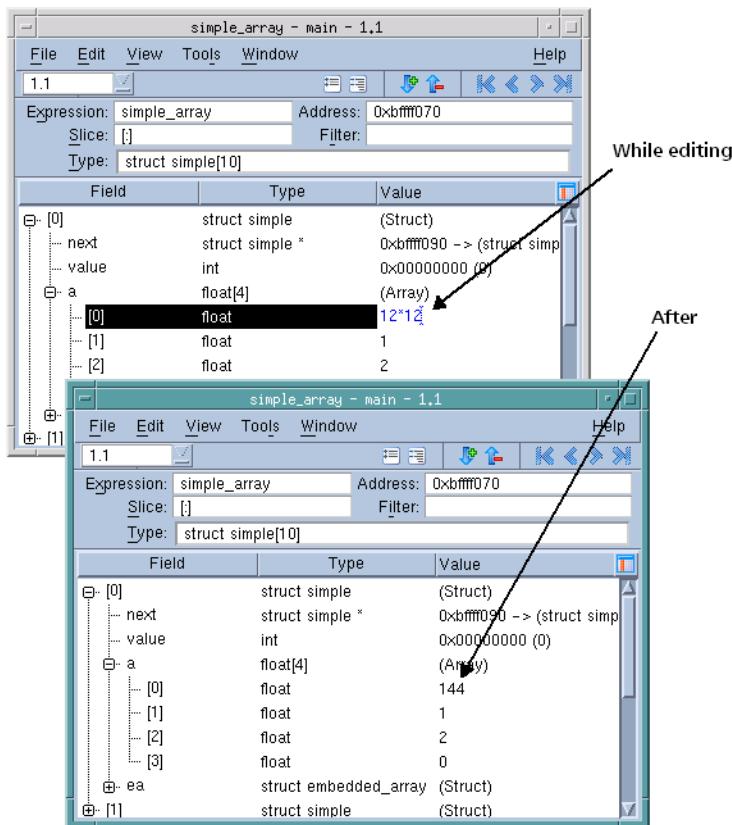
Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window, **Expression List** Window, or Stack Frame Pane by selecting the value and typing the new value. In addition to typing a value, you can also type an expression. For example, you can enter `12*12` as shown in the following figure. You can include logical operators in all TotalView expressions. (See Figure 199.)

```
CLI: set my_var [expr 1024*1024]
      dassign int8_array(3) $my_var
```

In most cases, you can edit a variable's value. If you right-click on a variable and the **Change Value** command isn't faded, you can edit the displayed value.

Figure 199: Using an Expression to Change a Value



TotalView does not let you directly change the value of bit fields; you can use the **Tools > Evaluate** Window to assign a value to a bit field. See Chapter 17, "Evaluating Expressions," on page 381.

CLI: Tcl lets you use operators such as & and | to manipulate bit fields on Tcl values.

Changing a Variable's Data Type

The data type declared for the variable determines its format and size (amount of memory). For example, if you declare an **int** variable, TotalView displays the variable as an integer.

The following sections discuss the different aspects of data types:

- "Displaying C and C++ Data Types" on page 312
- "Viewing Pointers to Arrays" on page 314
- "Viewing Arrays" on page 314
- "Viewing *typedef* Types" on page 315

- "Viewing Structures" on page 315
- "Viewing Unions" on page 315
- "Casting Using the Built-In Types" on page 315

You can change the way TotalView displays data in the Variable Window and the **Expression List** Window by editing the data type. This is known as *casting*. TotalView assigns types to all data types, and in most cases, they are identical to their programming language counterparts.

When a C or C++ variable is displayed in TotalView, the data types are identical to their C or C++ type representations, except for pointers to arrays. TotalView uses a simpler syntax for pointers to arrays. (See "Viewing Pointers to Arrays" on page 314.) Similarly, when Fortran is displayed in TotalView, the types are identical to their Fortran type representations for most data types including **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, and **CHARACTER**.

If the window contains a structure with a list of fields, you can edit the data types of the listed fields.



When you edit a data type, TotalView changes how it displays the variable in the current window. Other windows listing the variable do not change.

Displaying C and C++ Data Types

The syntax for displaying data is identical to C and C++ language cast syntax for all data types except pointers to arrays. That is, you use C and C++ cast syntax for data types. For example, you can cast using types such as **int**, **char**, **unsigned**, **float**, **double**, **union**, all named **struct** types, and so on. In addition, TotalView has a built-in type called **\$string**. Unless you tell it otherwise, TotalView maps **char** arrays to this type. (For information on wide characters, see "Viewing Wide Character Arrays (\$wchar Data Types)" on page 318.)

Read TotalView types from right to left. For example, **\$string*[20]*** is a pointer to an array of 20 pointers to **\$string**.

The following table shows some common TotalView data types:

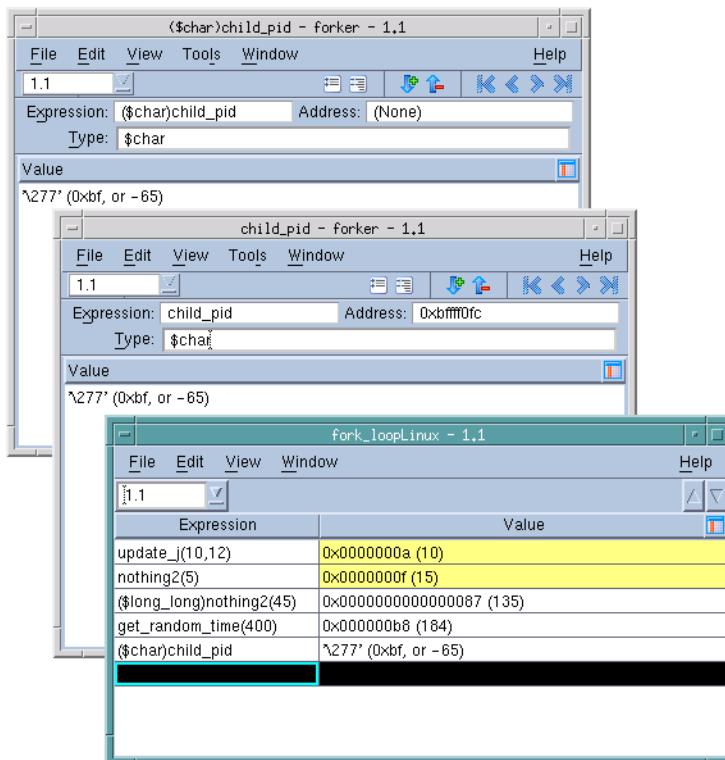
Data Type String	Description
int	Integer
int*	Pointer to an integer
int[10]	Array of 10 integers
\$string	Null-terminated character string
\$string**	Pointer to a pointer to a null-terminated character string
\$string*[20]*	Pointer to an array of 20 pointers to null-terminated strings

You can enter C and C++ Language cast syntax in the **Type** field.

Figure 200 on page 313 shows three different casts:

The two Variable Window cast the same data in the same way. In the top-left window, a cast was used in the **Expression** field. In the other Variable Window,

Figure 200: Three Casts



the data type was changed from **int** to **\$char**. In the first cast, TotalView changed the **Type** for you. In the second, it did not alter the **Expression** field.

The **Expression List** Window contains two casting examples. The first casts a function's returned value to **long long**. The second is the same cast as was made in the two Variable Windows.

TotalView also lets you cast a variable into an array. In the GUI, add an array specifier to the **Type** declaration. For example, adding [3] to a variable declared as an **int** changes it into an array of three **ints**.

When TotalView displays some complex arrays and structures, it displays the compound object or array types in the Variable Window.



Editing a compound object or array data type can produce undesirable results. TotalView tries to give you what you ask for, so if you get it wrong, the results are unpredictable. Fortunately, the remedy is quite simple: close the Variable Window and start over again.

The following sections discuss the following more complex data types.

- "Viewing Pointers to Arrays" on page 314
- "Viewing Arrays" on page 314
- "Viewing typedef Types" on page 315
- "Viewing Structures" on page 315
- "Viewing Unions" on page 315

Viewing Pointers to Arrays

Suppose you declared a variable **vbl** as a pointer to an array of 23 pointers to an array of 12 objects of type **mytype_t**. The C language declaration for this is:

```
mytype_t (*(*vbl)[23])[12];
```

Here is how you would cast the **vbl** variable to this type:

```
(mytype_t (*(*)[23])[12])vbl
```

The TotalView cast for **vbl** is:

```
mytype_t[12]*[23]*
```

Viewing Arrays

When you specify an array, you can include a lower and upper bound separated by a colon (:).

See Chapter 15, "Examining Arrays," on page 333 for more information on arrays.



By default, the lower bound for a C or C++ array is **0**, and the lower bound for a Fortran array is **1**. In the following example, an array of ten integers is declared in C and then in Fortran:

```
int a[10];
integer a(10)
```

The elements of the array range from **a[0]** to **a[9]** in C, while the elements of the equivalent Fortran array range from **a(1)** to **a(10)**.

TotalView also lets you cast a variable to an array. In the GUI, just add an array specifier to the **Type** declaration. For example, adding **(3)** to a variable declared as an **integer** changes it to an array of three **integers**.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements in the dimension). Consider the following Fortran array declaration:

```
integer a(1:7,1:8)
```

Since both dimensions of this Fortran array use the default lower bound, which is **1**, TotalView displays the data type of the array by using only the extent of each dimension, as follows:

```
integer (7,8)
```

If an array declaration doesn't use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, you declare an array of integers with the first dimension ranging from **-1** to **5** and the second dimension ranging from **2** to **10**, as follows:

```
integer a(-1:5,2:10)
```

TotalView displays this the same way.

When editing an array's dimension, you can enter just the extent (if using the default lower bound), or you can enter the lower and upper bounds separated by a colon.

TotalView also lets you display a subsection of an array, or filter a scalar array for values that match a filter expression. See "Displaying Array Slices" on page 334 and "Filtering Array Data Overview" on page 337 for more information.

Viewing `typedef` Types

TotalView recognizes the names defined with `typedef`, and displays these user-defined types; for example:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

TotalView displays the type for `p_vbl` as `dptr_t`.

Viewing Structures

TotalView lets you use the `struct` keyword as part of a type string. In most cases, this is optional.



This behavior depends upon which compiler you are using. In most cases, you'll see what is described here.

If you have a structure and another data type with the same name, however, you must include the `struct` keyword so that TotalView can distinguish between the two data types.

If you use a `typedef` statement to name a structure, TotalView uses the `typedef` name as the type string. Otherwise, TotalView uses the structure tag for the `struct`.

Viewing Unions

TotalView displays a union in the same way that it displays a structure. Even though the fields of a union are overlaid in storage, TotalView displays the fields on separate lines. (See Figure 201 on page 316.)

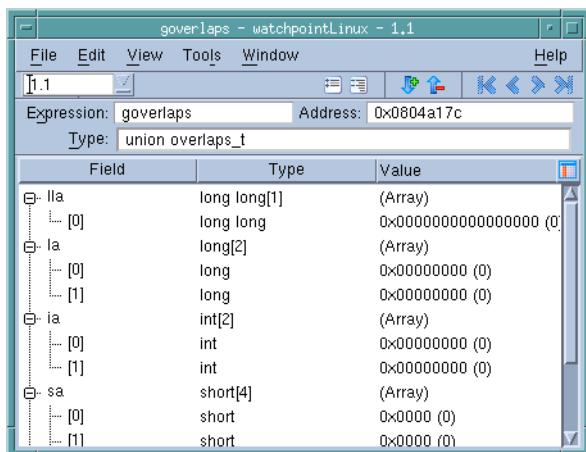
CLI: `dprint variable`

Casting Using the Built-In Types

TotalView provides a number of predefined types. These types are preceded by a \$. You can use these built-in types anywhere you can use the ones defined in your programming language. These types are also useful in

Changing a Variable's Data Type

Figure 201: Displaying a Union



debugging executables with no debugging symbol table information. The following table describes the built-in types:

Type String	Language	Size	Description
\$address	C	void*	Void pointer (address).
\$char	C	char	Character.
\$character	Fortran	character	Character.
\$code	C	architecture-dependent	Machine instructions. The size used is the number of bytes required to hold the shortest instruction for your computer.
\$complex	Fortran	complex	Single-precision floating-point complex number. The complex types contain a real part and an imaginary part, which are both of type real .
\$complex_8	Fortran	complex*8	A real*4 -precision floating-point complex number. The complex*8 types contain a real part and an imaginary part, which are both of type real*4 .
\$complex_16	Fortran	complex*16	A real*8 -precision floating-point complex number. The complex*16 types contain a real part and an imaginary part, which are both of type real*8 .
\$double	C	double	Double-precision floating-point number.
\$double_precision	Fortran	double precision	Double-precision floating-point number.

Type String	Language	Size	Description
\$extended	C	architecture-dependent; often <code>long double</code>	Extended-precision floating-point number. Extended-precision numbers must be supported by the target architecture. In addition, the format of extended floating point numbers varies depending on where it's stored. For example, the x86 register has a special 10-byte format, which is different than the in-memory format. Consult your vendor's architecture documentation for more information.
\$float	C	<code>float</code>	Single-precision floating-point number.
\$int	C	<code>int</code>	Integer.
\$integer	Fortran	<code>integer</code>	Integer.
\$integer_1	Fortran	<code>integer*1</code>	One-byte integer.
\$integer_2	Fortran	<code>integer*2</code>	Two-byte integer.
\$integer_4	Fortran	<code>integer*4</code>	Four-byte integer.
\$integer_8	Fortran	<code>integer*8</code>	Eight-byte integer.
\$logical	Fortran	<code>logical</code>	Logical.
\$logical_1	Fortran	<code>logical*1</code>	One-byte logical.
\$logical_2	Fortran	<code>logical*2</code>	Two-byte logical.
\$logical_4	Fortran	<code>logical*4</code>	Four-byte logical.
\$logical_8	Fortran	<code>logical*8</code>	Eight-byte logical.
\$long	C	<code>long</code>	Long integer.
\$long_long	C	<code>long long</code>	Long long integer.
\$real	Fortran	<code>real</code>	Single-precision floating-point number. When using a value such as <code>real</code> , be careful that the actual data type used by your computer is not <code>real*4</code> or <code>real*8</code> , since different results can occur.
\$real_4	Fortran	<code>real*4</code>	Four-byte floating-point number.
\$real_8	Fortran	<code>real*8</code>	Eight-byte floating-point number.
\$real_16	Fortran	<code>real*16</code>	Sixteen-byte floating-point number.
\$short	C	<code>short</code>	Short integer.
\$string	C	<code>char</code>	Array of characters.
\$void	C	<code>long</code>	Area of memory.
\$wchar	C	<code>platform-specific</code>	Platform-specific wide character used by <code>wchar_t</code> data types
\$wchar_s16	C	16 bits	wide character whose storage is signed 16 bits (not currently used by any platform)

Type String	Language	Size	Description
\$wchar_u16	C	16 bits	wide character whose storage is unsigned 16 bits
\$wchar_s32	C	32 bits	wide character whose storage is signed 32 bits
\$wchar_u32	C	32 bits	wide character whose storage is unsigned 32 bits
\$wstring	C	<i>platform-specific</i>	Platform-specific string composed of \$wchar characters
\$wstring_s16	C	16 bits	String composed of \$wchar_s16 characters (not currently used by any platform)
\$wstring_u16	C	16 bits	String composed of \$wchar_u16 characters
\$wstring_s32	C	32 bits	String composed of \$wchar_s32 characters
\$wstring_u32	C	32 bits	String composed of \$wchar_u32 characters

Viewing Character Arrays (\$string Data Type)

If you declare a character array as `char vbl[n]`, TotalView automatically changes the type to `$string[n]`; that is, a null-terminated, quoted string with a maximum length of *n*. This means that TotalView displays an array as a quoted string of *n* characters, terminated by a null character. Similarly, TotalView changes `char*` declarations to `$string*` (a pointer to a null-terminated string).

Since most character arrays represent strings, the TotalView `$string` type can be very convenient. But if this isn't what you want, you can edit the `$string` and change it back to a `char` (or `char[n]`), to display the variable as you declared it.

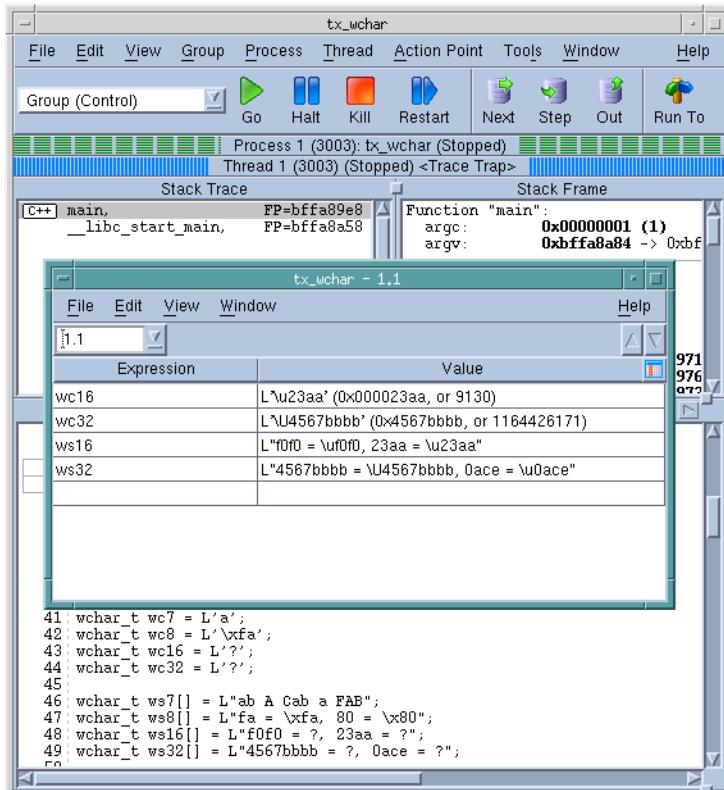
Viewing Wide Character Arrays (\$wchar Data Types)

If you create an array of `wchar_t` wide characters, TotalView automatically changes the type to `$wstring[n]`; that is, it is displayed as a null-terminated, quoted string with a maximum length of *n*. For an array of wide characters, the null terminator is `L'0'`. Similarly, TotalView changes `wchar_t*` declarations to `$wstring*` (a pointer to a null-terminated string). (See Figure 202 on page 319.)

This figure shows the declaration of two wide characters in the Process Window. The **Expression List** Window shows how TotalView displays their data. The `L` in the data indicates that TotalView is displaying a wide literal.

Since most wide character arrays represent strings, the `$wstring` type can be very convenient. But if this isn't what you want, you can edit the `$wstring` and change it back to a `wchar_t` (or `wchar[n]` or `$wchar` or `$wchar[n]`), to display the variable as you declared it.

Figure 202: Displaying wchar_t Data



If the wide character uses from 9 to 16 bits, TotalView displays the character using the following universal-character code representation:

\uXXXX

X represents a hexadecimal digit. If the character uses from 17 to 32 bits, TotalView uses the following representation:

\UXXXXXXXXX



Platforms and compilers differ in the way they represent wchar_t. In consequence, TotalView allows you to see this information in platform-specific ways. For example, you can cast a string to \$wstring_s16 or \$wstring_s32. In addition, many compilers have problems either using wide characters or handing off information about wide characters so that they can be interpreted by any debugger (not just TotalView). For information on supported compilers, see the TotalView Release Notes at http://www.totalview-tech.com/Support/release_notes.php.

Viewing Areas of Memory (\$void Data Type)

TotalView uses the **\$void** data type for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The **\$void** type is similar to the **int** type in the C Language.

If you dive on registers or display an area of memory, TotalView lists the contents as a **\$void** data type. If you display an array of **\$void** variables, the index for each object in the array is the address, not an integer. This address can be useful when you want to display large areas of memory.

If you want, you can change a **\$void** to another type. Similarly, you can change any type to a **\$void** to see the variable in decimal and hexadecimal formats.

Viewing Instructions (\$code Data Type)

TotalView uses the **\$code** data type to display the contents of a location as machine instructions. To look at disassembled code stored at a location, dive on the location and change the type to **\$code**. To specify a block of locations, use **\$code[n]**, where *n* is the number of locations being displayed.

Viewing Opaque Data

An opaque type is a data type that could be hidden, not fully specified, or be defined in another part of your program. For example, the following C declaration defines the data type for **p** to be a pointer to **struct foo**, and **foo** is not yet defined:

```
struct foo;  
struct foo *p;
```

When TotalView encounters a variable with an opaque type, it searches for a **struct**, **class**, **union**, or **enum** definition with the same name as the opaque type. If TotalView doesn't find a definition, it displays the value of the variable using an opaque type name; for example:

(Opaque foo)

Some compilers do not store sufficient information for TotalView to locate the type. This could be the reason why TotalView uses the opaque type.

You can tell TotalView to use the correct data type by having it read the source file. For example, if TotalView is showing you (Opaque foo) and you know that **struct foo** is defined in source file **foo.c**, use the **File > Open Source** Command. While this command's primary purpose is to tell TotalView to display the file within the Process Window, it also causes TotalView to read the file's debugging information. As a side-effect, **struct foo** should now be defined. Because TotalView now knows its definition, it can resolve the opaque type.

Type-Casting Examples

This section contains three type-casting examples:

- Displaying Declared Arrays
- Displaying Allocated Arrays
- Displaying the argv Array

Displaying Declared Arrays

TotalView displays arrays the same way it displays local and global variables. In the Stack Frame or Source Pane, dive on the declared array. A Variable Window displays the elements of the array.

```
CLI: dprint array-name
```

Displaying Allocated Arrays

The C Language uses pointers for dynamically allocated arrays; for example:

```
int *p = malloc(sizeof(int) * 20);
```

Since TotalView doesn't know that **p** actually points to an array of integers, you need to do several things to display the array:

- 1 Dive on the variable **p** of type **int***.
- 2 Change its type to **int[20]***.
- 3 Dive on the value of the pointer to display the array of 20 integers.

Displaying the argv Array

Typically, **argv** is the second argument passed to **main()**, and it is either a **char **argv** or **char *argv[]**. Suppose **argv** points to an array of three pointers to character strings. Here is how you can edit its type to display an array of three pointers, as follows:

- 1 Select the type string for **argv**.

```
CLI: dprint argv
```

- 2 Edit the type string by using the field editor commands. Change it to:

```
$string*[3]*
```

```
CLI: dprint {($string*[3]*)argv}
```

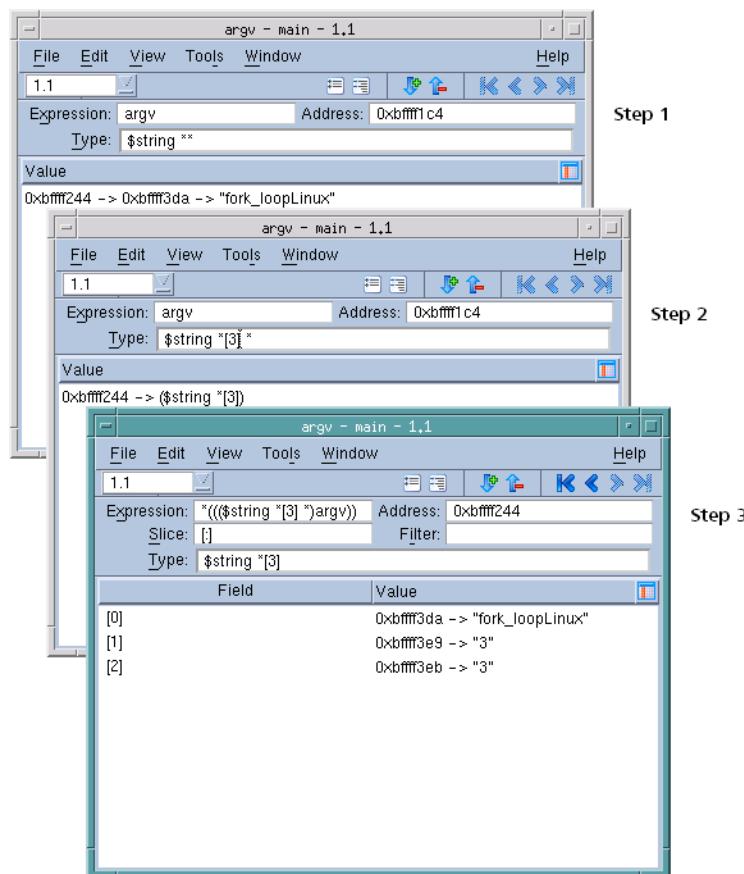
- 3 To display the array, dive on the value field for **argv**. (See Figure 203 on page 322.)

Changing the Address of Variables

You can edit the address of a variable in a Variable Window by editing the value shown in the **Address** field. When you edit this address, the Variable Window shows the contents of the new location.

You can also enter an address expression such as **0x10b8 – 0x80** in this area.

Figure 203: Editing the argv Argument



Displaying C++ Types

Viewing Classes

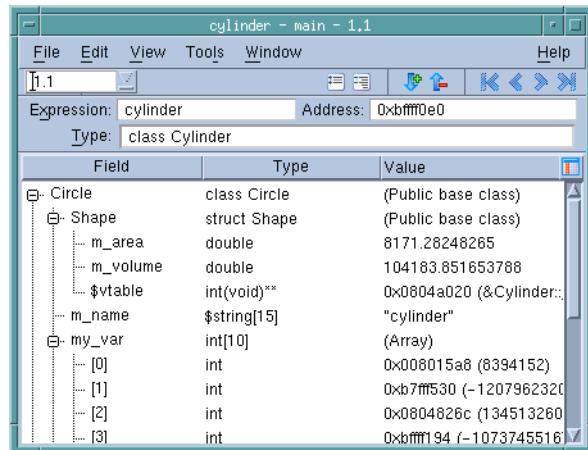
TotalView displays C++ classes and accepts **class** as a keyword. When you debug C++, TotalView also accepts the *unadorned* name of a **class**, **struct**, **union**, or **enum** in the type field. TotalView displays nested classes that use inheritance, showing derivation by indentation.

Some C++ compilers do not write accessibility information. In these cases, TotalView cannot display this information.



For example, Figure 204 displays an object of a class **c**.

Figure 204: Displaying C++ Classes That Use Inheritance



Its definition is as follows:

```

class b {
    char * b_val;
public:
    b() {b_val = "b value";}
};

class d : virtual public b {
    char * d_val;
public:
    d() {d_val = "d value";}
};

class e {
    char * e_val;
public:
    e() {e_val = "e value";}
};

class c : public d, public e {
    char * c_val;
public:
    c() {c_val = "c value";}
};

```

TotalView tries to display the correct data when you change the type of a Variable Window while moving up or down the derivation hierarchy. Unfortunately, many compilers do not contain the information that TotalView needs so you might need to cast your class.

Displaying Fortran Types

TotalView lets you display FORTRAN 77 and Fortran 90 data types.

The topics in this section describe the various types and how the debugger handles them:

- “[Displaying Fortran Common Blocks](#)” on page 324
- “[Displaying Fortran Module Data](#)” on page 324
- “[Debugging Fortran 90 Modules](#)” on page 326
- “[Viewing Fortran 90 User-Defined Types](#)” on page 327
- “[Viewing Fortran 90 Deferred Shape Array Types](#)” on page 327
- “[Viewing Fortran 90 Pointer Types](#)” on page 328
- “[Displaying Fortran Parameters](#)” on page 329

Displaying Fortran Common Blocks

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function’s common block list. The Stack Frame Pane displays the name of each common block for a function. The names of common block members have function scope, not global scope.

CLI: `dprint variable-name`

If you dive on a common block name in the Stack Frame Pane, the debugger displays the entire common block in a Variable Window, as shown in (See Figure 205 on page 325.)

Window 1 in this figure shows a common block list in a Stack Frame Pane. After several dives, **Window 2** contains the results of diving on the common block.

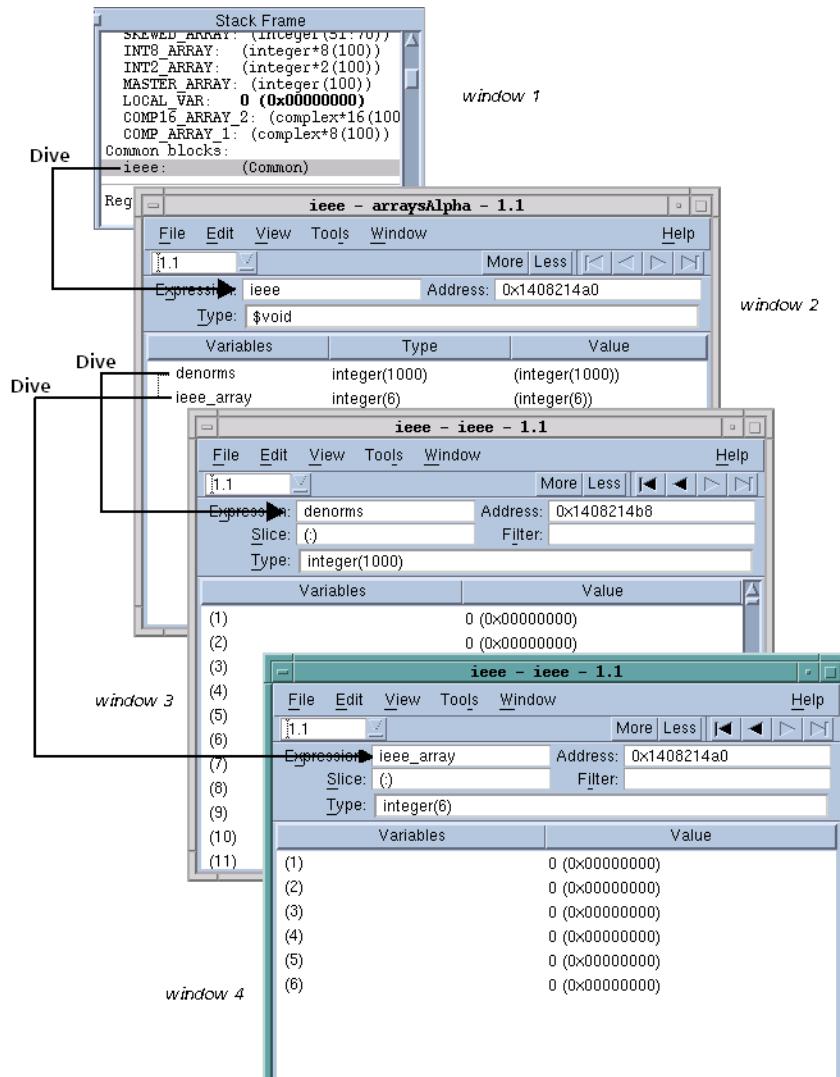
If you dive on a common block member name, TotalView searches all common blocks in the function’s scope for a matching member name, and displays the member in a Variable Window.

Displaying Fortran Module Data

TotalView tries to locate all data associated with a Fortran module and display it all at once. For functions and subroutines defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView can restrict what’s displayed.

Figure 205: Diving on a Common Block List in the Stack Frame Pane



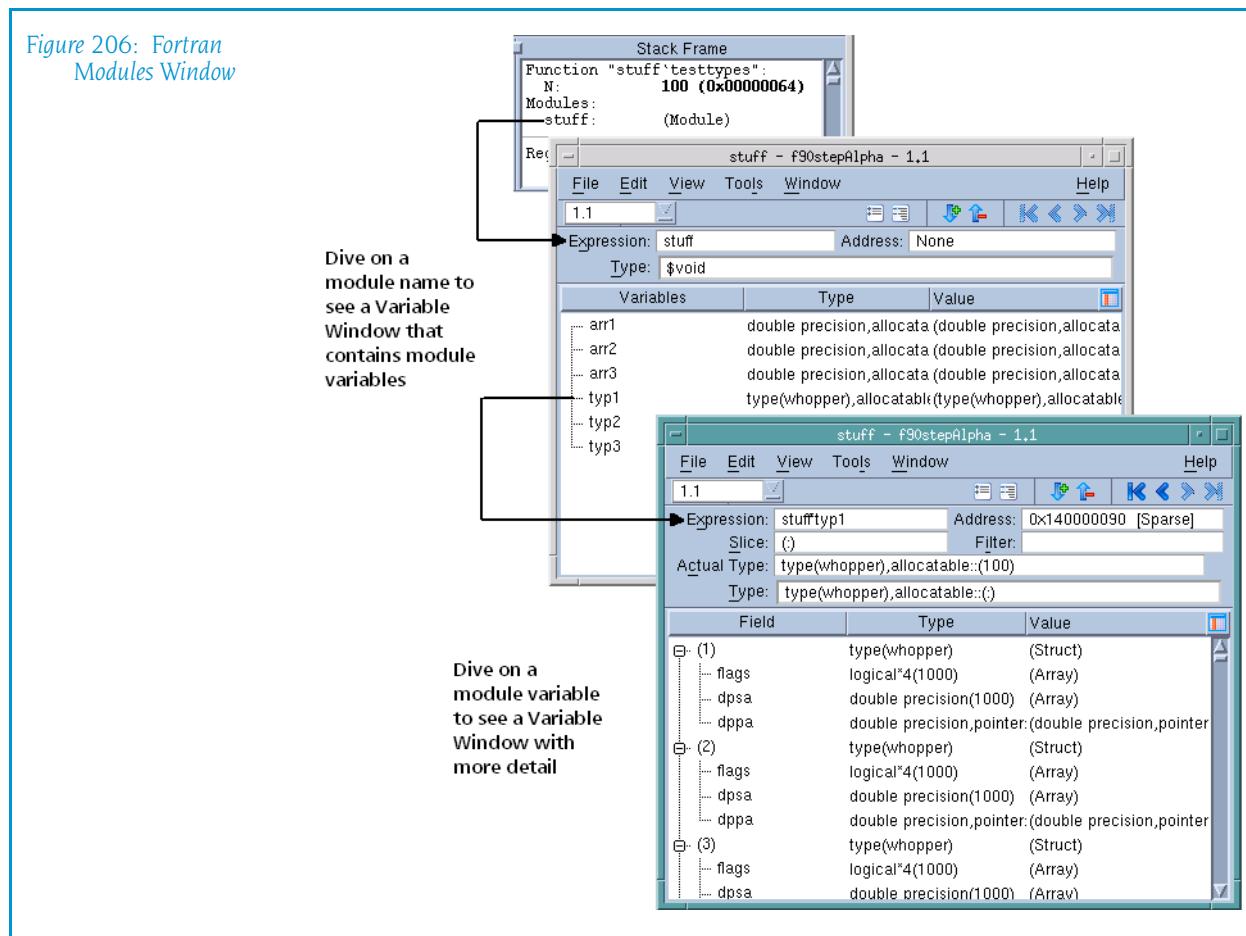
Although a function may use a module, TotalView doesn't always know if the module was used or what the true names of the variables in the module are. If this happens, either of the following occurs:

- Module variables appear as local variables of the subroutine.
- A module appears on the list of modules in the Stack Frame Pane that contains (with renaming) only the variables used by the subroutine.

CLI: `dprint variable-name`

Alternatively, you can view a list of all the known modules by using the **Tools > Fortran Modules** command. Because Fortran modules display in a Variable Window, you can dive on an entry to display the actual module data, as shown in Figure 206 on page 326.

Figure 206: Fortran Modules Window



If you are using the SUNPro compiler, TotalView can only display module data if you force it to read the debug information for a file that contains the module definition or a module function. For more information, see "Finding the Source Code for Functions" on page 225.

Debugging Fortran 90 Modules

Fortran 90 lets you place functions, subroutines, and variables inside modules. You can then include these modules elsewhere by using a **USE** command. When you do this, the names in the module become available in the *using* compilation unit, unless you either exclude them with a **USE ONLY** statement or rename them. This means that you don't need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you need to know the location of the function being called. Consequently, TotalView uses the following syntax when it displays a function contained in a module:

modulename`functionname

You can also use this syntax in the **File > New Program** and **View > Lookup Variable** commands.

Fortran 90 also lets you create a contained function that is only visible in the scope of its parent and siblings. There can be many contained func-

tions in a program, all using the same name. If the compiler gave TotalView the function name for a nested function, TotalView displays it using the following syntax:

parentfunction() `containedfunction

CLI: `dprint module_name`variable_name`

Viewing Fortran 90 User-Defined Types

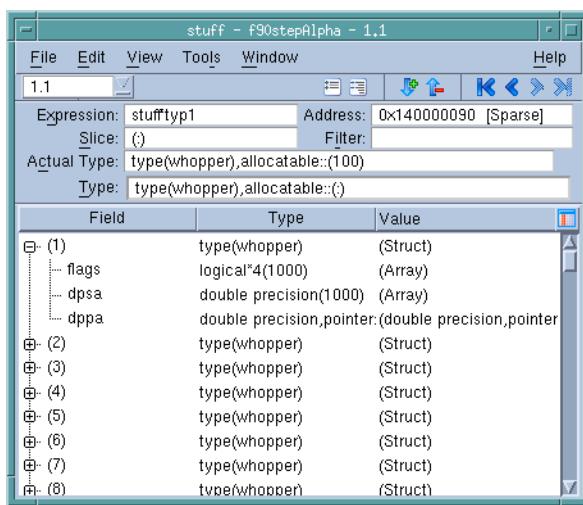
A Fortran 90 user-defined type is similar to a C structure. TotalView displays a user-defined type as `type(name)`, which is the same syntax used in Fortran 90 to create a user-defined type. For example, the following code fragment defines a variable `typ2` of `type(whopper)`:

```
TYPE WHOPPER
    LOGICAL, DIMENSION(ISIZE) :: FLAGS
    DOUBLE PRECISION, DIMENSION(ISIZE) :: DPSA
    DOUBLE PRECISION, DIMENSION(:), POINTER :: DPPA
END TYPE WHOPPER

TYPE(WHOPPER), DIMENSION(:), ALLOCATABLE :: TYP2
```

TotalView displays this type. (See Figure 207.)

Figure 207: Fortran 90 User-Defined Type



Viewing Fortran 90 Deferred Shape Array Types

Fortran 90 lets you define deferred shape arrays and pointers. The actual bounds of a deferred shape array are not determined until the array is allocated, the pointer is assigned, or, in the case of an assumed shape argument to a subroutine, the subroutine is called. TotalView displays the type of deferred shape arrays as `type(:)`.

When TotalView displays the data for a deferred shape array, it displays the type used in the definition of the variable and the actual type that this instance of the variable has. The actual type is not editable, since you can

achieve the same effect by editing the definition's type. The following example shows the type of a deferred shape rank 2 array of **real** data with runtime lower bounds of **-1** and **2**, and upper bounds of **5** and **10**:

```
Type: real(:,:)
Actual Type: real(-1:5,2:10)
Slice: (:,:)
```

Viewing Fortran 90 Pointer Types

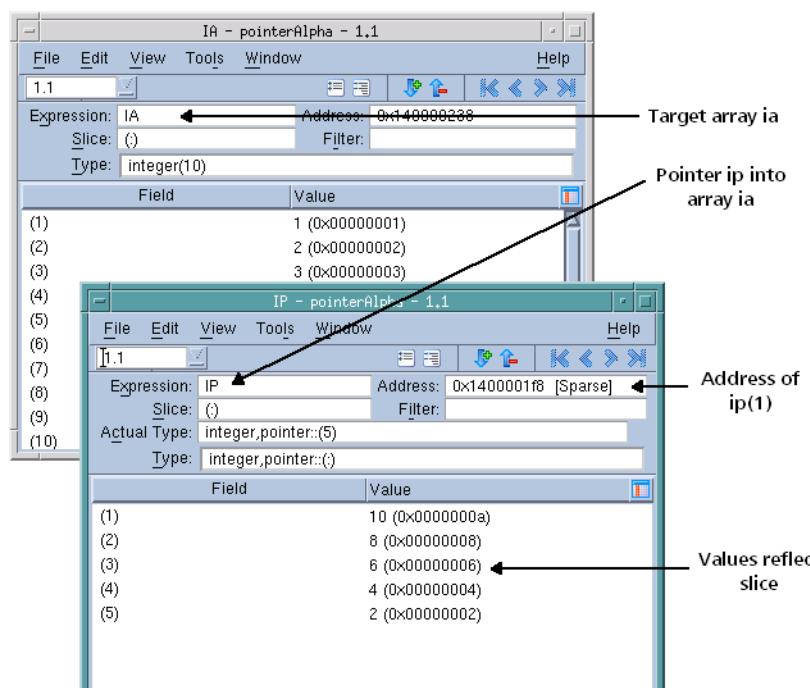
A Fortran 90 pointer type lets you point to scalar or array types.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that indices and values it displays in a Variable Window are the same as in the Fortran code; for example:

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = 1,10
    ia(i) = i
end do
ip => ia(10:1:-2)
```

After diving through the **ip** pointer, TotalView displays the windows shown in Figure 208:

Figure 208: Fortran 90 Pointer Value



The address displayed is not that of the array's base. Since the array's stride is negative, array elements that follow are at lower absolute addresses. Consequently, the address displayed is that of the array element

that has the lowest index. This might not be the first displayed element if you used a slice to display the array with reversed indices.

Displaying Fortran Parameters

A Fortran **PARAMETER** defines a named constant. If your compiler generates debug information for parameters, they are displayed in the same way as any other variable. However, some compilers do not generate information that TotalView can use to determine the value of a **PARAMETER**. This means that you must make a few changes to your program if you want to see this type of information.

If you're using Fortran 90, you can define variables in a module that you initialize to the value of these **PARAMETER** constants; for example:

```
INCLUDE 'PARAMS.INC'
MODULE CONSTS
SAVE
INTEGER PI_C = PI
...
END MODULE CONSTS
```

The **PARAMS.INC** file contains your parameter definitions. You then use these parameters to initialize variables in a module. After you compile and link this module into your program, the values of these parameter variables are visible.

If you're using FORTRAN 77, you can achieve the same results if you make the assignments in a common block and then include the block in **main()**. You can also use a block data subroutine to access this information.

Displaying Thread Objects

On HP Alpha Tru64 UNIX and IBM AIX systems, TotalView can display information about mutexes and conditional variables. In addition, TotalView can display information on read/write locks and data keys on IBM AIX. You can obtain this information by selecting the **Tools > Thread Objects** command. After selecting this command, TotalView displays a window that contains either two tabs (HP Alpha) or four tabs (IBM). Figure 209 on page 330 shows AIX examples.

Diving on any line in these windows displays a Variable Window that contains additional information about the item.

Here are some things you should know:

- If you're displaying data keys, many applications initially set keys to **0** (the NULL pointer value). TotalView doesn't display a key's information, however, until a thread sets a non-NUL value to the key.

Figure 209: Thread Objects Page on an IBM AIX Computer



- If you select a thread ID in a data key window, you can dive on it using the **View > Dive Thread** and **View > Dive Thread in New Window** commands to display a Process Window for that thread ID.

The online Help contains information on the contents of these windows.

Scoping and Symbol Names

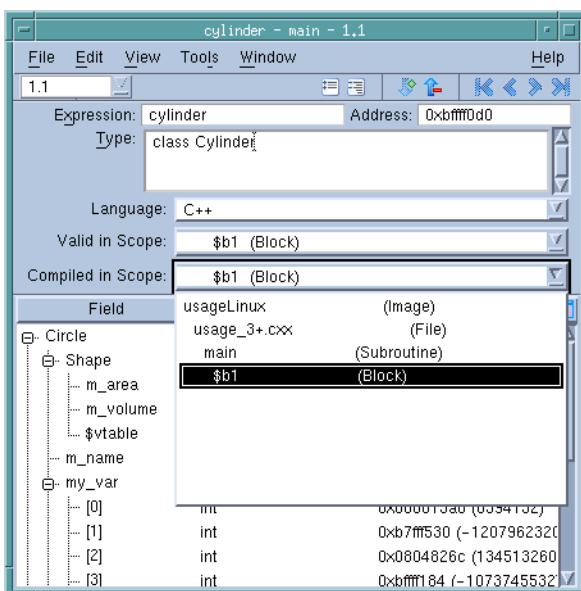
TotalView assigns a unique name to every element in your program based on the scope in which the element exists. A *scope* defines the part of a program that knows about a symbol. For example, the scope of a variable that is defined at the beginning of a subroutine is all the statements in the sub-

routine. The variable's scope does not extend outside of this subroutine. A program consists of multiple *scopes*. Of course, a block contained in the subroutine could have its own definition of the same variable. This would *hide* the definition in the enclosing scope.

All scopes are defined by your program's structure. Except for the simplest of programs, scopes are embedded in other scopes. The only exception is the outermost scope, which is the one that contains `main()`, which is not embedded. Every element in a program is associated with a scope.

To see the scope in which a variable is valid, click the **More** button in the Variable Window until the scope fields are visible. The Variable Window now includes additional information about your variable, as is shown in Figure 210 on page 331.

**Figure 210: Variable Window:
Showing Variable Properties**



The **Valid in Scope** list indicates the scope in which the variable resides. That is, when this scope is active, the variable is defined. The **Compiled in Scope** list can differ if you modify the variable with an expression. It indicates where variables in this expression have meaning.

When you tell the CLI or the GUI to execute a command, TotalView consults the program's symbol table to discover which object you are referring to—this process is known as *symbol lookup*. Symbol lookup is performed with respect to a particular context, and each context uniquely identifies the scope to which a symbol name refers.

For additional information, see "Scoping Issues" on page 289.

Qualifying Symbol Names

The way you describe a scope is similar to the way you specify a file. The scopes in a program form a tree, with the outermost scope (which is your

program) as the root. At the next level are executable files and dynamic libraries; further down are compilation units (source files), procedures, modules, and other scoping units (for example, blocks) supported by the programming language. Qualifying a symbol is equivalent to describing the path to a file in UNIX file systems.

A symbol is fully scoped when you name all levels of its tree. The following example shows how to scope a symbol and also indicates parts that are optional:

`[#executable-or-lib#][file#][procedure-or-line#]symbol`

The pound sign (#) separates elements of the fully qualified name.



Because of the number of different types of elements that can appear in your program, a complete description of what can appear and their possible order is complicated and unreadable. In contrast, after you see a name in the Stack Frame Pane, it is easy to read a variable's scoped name.

TotalView interprets most programs and components as follows:

- You do not need to qualify file names with a full path, and you do not need to use all levels in a symbol's scoping tree. TotalView conventions here are similar to the way UNIX displays file names.
- If a qualified symbol begins with #, the name that follows indicates the name of the executable or shared library (just as an absolute file path begins with a directory immediately in the root directory). If you omit the executable or library component, the qualified symbol doesn't begin with #.
- The source file's name can appear after the possibly omitted executable or shared library.
- Because programming languages typically do not let you name blocks, that portion of the qualifier is specified using the symbols **\$b** followed by a number that indicates which block. For example, the first unnamed block is named **\$b1**, the second is **\$b2**, and so on.

You can omit any part of the scope specification that TotalView doesn't need to uniquely identify the symbol. Thus, **foo#x** identifies the symbol **x** in the procedure **foo**. In contrast, **#foo#x** identifies either procedure **x** in executable **foo** or variable **x** in a scope from that executable.

Similarly, **#foo#bar#x** could identify variable **x** in procedure **bar** in executable **foo**. If **bar** were not unique in that executable, the name would be ambiguous unless you further qualified it by providing a file name. Ambiguities can also occur if a file-level variable (common in C programs) has the same name as variables declared in functions in that file. For instance, **bar.c#x** refers to a file-level variable, but the name can be ambiguous when there are different definitions of **x** embedded in functions that occur in the same file. In this case, you need to enter **bar.c#b1#x** to identify the scope that corresponds to the outer level of the file (that is, the scope that contains line 1 of the file).

Examining Arrays



This chapter explains how to examine and change array data as you debug your program. Since arrays also appear in the Variable Window, you need to be familiar with the information in Chapter 14, "Examining and Changing Data," on page 279.

The topics in this chapter are:

- "Examining and Analyzing Arrays" on page 333
- "Displaying a Variable in all Processes or Threads" on page 345
- "Visualizing Array Data" on page 347

15

Examining and Analyzing Arrays

TotalView can quickly display very large arrays in Variable Windows. An array can be the elements that you define in your program, or it can be an area of memory that you cast into an array.

If an array extends beyond the memory section in which it resides, the initial portion of the array is formatted correctly. If memory isn't allocated for an array element, TotalView displays **Bad Address** in the element's subscript.

Topics in this section are:

- "Displaying Array Slices" on page 334
- "Filtering Array Data Overview" on page 337
- "Sorting Array Data" on page 342
- "Obtaining Array Statistics" on page 343

Displaying Array Slices

TotalView lets you display array subsections by editing the **Slice** field in an array's Variable Window. (An array subsection is called a *slice*.) The **Slice** field contains placeholders for all array dimensions. For example, the following is a C declaration for a three-dimensional array:

```
integer an_array[10][20][5]
```

Because this is a three-dimensional array, the initial slice definition is `[::][::][::]`. This lets you know that the array has three dimensions and that TotalView is displaying all array elements.

The following is a deferred shape array definition for a two-dimensional array variable:

```
integer, dimension (:,:) :: another_array
```

The TotalView slice definition is `(:,:)`.

TotalView displays as many colons `(:)` as there are array dimensions. For example, the slice definition for a one-dimensional array (a vector) is `[:]` for C arrays and `(:)` for Fortran arrays.

```
CLI: dprint -slice “[n:m]” an_array
      dprint -slice “(n:m,p:q)” an_array
```

Using Slices and Strides

A slice has the following form:

$$\text{lower_bound:upper_bound[:stride]}$$

The *stride*, which is optional, tells TotalView to skip over elements and not display them. Adding a *stride* to a slice tells the debugger to display every *stride* element of the array, starting at the *lower_bound* and continuing through the *upper_bound*, inclusive.

For example, a slice of `[0:9:9]` used on a ten-element C array tells TotalView to display the first element and last element, which is the ninth element beyond the lower bound.

If the stride is negative and the lower bound is greater than the upper bound, TotalView displays a dimension with its indices reversed. That is, TotalView treats the slice as if it was defined as follows:

$$[\text{upperbound} : \text{lowerbound} : \text{stride}]$$

```
CLI: dprint an_array(n:m:p,q:r:s)
```

For example, the following definition tells TotalView to display an array beginning at its last value and moving to its first:

$$[:::-1]$$

This syntax differs from Fortran 90 syntax in that Fortran 90 requires that you explicitly enter the upper and lower bounds when you're reversing the order for displaying array elements.

Because the default value for the stride is 1, you can omit the stride (and the colon that precedes it) from your definition. For example, the following two definitions display array elements 0 through 9:

[0:9:1]
[0:9]

If the lower and upper bounds are the same, just use a single number. For example, the following two definitions tell TotalView to display array element 9:

[9:9:1]
[9]



The lower_bound, upper_bound, and stride must be constants. They cannot be expressions.

Example 1

A slice declaration of [::2] for a C or C++ array (with a default lower bound of 0) tells TotalView to display elements with even indices of the array; that is, 0, 2, 4, and so on. However, if this were defined for a Fortran array (where the default lower bound is 1), TotalView displays elements with odd indices of the array; that is, 1, 3, 5, and so on.

Example 2

The following figure displays a stride of (::9,:9). This definition displays the four corners of a ten-element by ten-element Fortran array.

Figure 211: Stride Displaying the Four Corners of an Array

Field	Value
(1,1)	0 (0x00000000)
(10,1)	9 (0x00000009)
(1,10)	90 (0x0000005a)
(10,10)	99 (0x00000063)

Example 3

You can use a stride to invert the order *and* skip elements. For example, the following slice begins with the upper bound of the array and displays every other element until it reaches the lower bound of the array:

(:::-2)

Using (:::-2) with a Fortran `integer(10)` array tells TotalView to display the elements 10, 8, 6, 4, and 2.

Example 4

You can simultaneously invert the array's order and limit its extent to display a small section of a large array. The following figure shows how to

specify a `(2:3,7::-1)` slice with an `integer*4(-1:5,2:10)` Fortran array. (See Figure 212.)

Figure 212: Fortran Array with Inverse Order and Limited Extent

The screenshot shows the TotalView Variable Window. The title bar says "SMALL_ARRAY - /nfs/netapp0/user/home/barryk/tests/ten_by_tenAlpha". The menu bar includes File, Edit, View, Tools, Window, and Help. A toolbar with various icons is at the top. The main area has a table with columns "Field" and "Value". The "Expression" field is set to "SMALL_ARRAY". The "Slice" field contains "(2:3,7::-1)". The "Type" field is "integer(10,10)". The table data is as follows:

Field	Value
(2,10)	91 (0x0000005b)
(3,10)	92 (0x0000005c)
(2,9)	81 (0x00000051)
(3,9)	82 (0x00000052)
(2,8)	71 (0x00000047)
(3,8)	72 (0x00000048)
(2,7)	61 (0x0000003d)
(3,7)	62 (0x0000003e)

After you enter this slice value, TotalView only shows elements in rows 2 and 3 of the array, beginning with column 10 and ending with column 7.

Using Slices in the Lookup Variable Command

When you use the **View > Lookup Variable** command to display a Variable Window, you can include a slice expression as part of the variable name. Specifically, if you type an array name followed by a set of slice descriptions in the **View > Lookup Variable** command dialog box, TotalView initializes the **Slice** field in the Variable Window to this slice description.

If you add subscripts to an array name in the **View > Lookup Variable** dialog box, TotalView will look up just that array element.

CLI: `dprint small_array(5,5)`

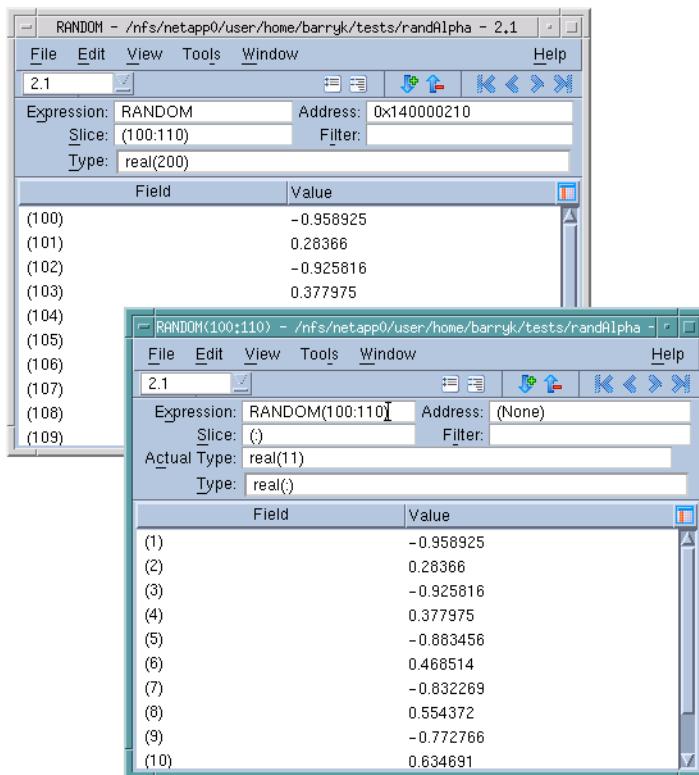
You can, of course, type an expression into the **View > Lookup Variable** dialog box; for example, you could type `small_array(i-1,j-1)`.

Array Slices and Array Sections

An array slice allows you to see a part of an array. The slice allows you to remove parts of the array you do not want to see. For example, if you have a 10,000 element array, you could tell TotalView that it should only display 100 of these elements. Fortran has introduced the concept of an array section. When you create an array section, you are creating a new array that is a subset of the old array. Because it is a new array, its first array index is 1.

In Figure 213 on page 337, the top left Variable Window displays an eleven-element array slice. The bottom right Variable Window displays an eleven-element array.

Figure 213: An Array Slice and an Array Section



While the data in both is identical, notice that the array numbering is different. In addition, the array slice shows an address for the array. The section, however, only exists within TotalView. Consequently, there is no address associated with it.

Filtering Array Data Overview

You can restrict what TotalView displays in a Variable Window by adding a filter to the window. You can filter arrays of type character, integer, or floating point. Your filtering options are:

- Arithmetic comparison to a constant value
- Equal or not equal comparison to IEEE NaNs, Infs, and Denorms
- Within a range of values, inclusive or exclusive
- General expressions

When an element of an array matches the filter expression, TotalView includes the element in the Variable Window display.

The following topics describe filtering options:

- “*Filtering Array Data*” on page 338
- “*Filtering by Comparison*” on page 338
- “*Filtering for IEEE Values*” on page 339
- “*Filtering a Range of Values*” on page 341
- “*Creating Array Filter Expressions*” on page 341
- “*Using Filter Comparisons*” on page 342

Filtering Array Data: The procedure for filtering an array is simple: select the **Filter** field, enter the array filter expression, and then press Enter.

TotalView updates the Variable Window to exclude elements that do not match the filter expression. TotalView only displays an element if its value matches the filter expression and the slice operation.

If necessary, TotalView converts the array element before evaluating the filter expression. The following conversion rules apply:

- If the filter operand or array element type is floating point, TotalView converts the operand to a double-precision floating-point value. TotalView truncates extended-precision values to double precision. Converting integer or unsigned integer values to double-precision values might result in a loss of precision. TotalView converts unsigned integer values to nonnegative double-precision values.
- If the filter operand or the array element is an unsigned integer, TotalView converts the operand to an unsigned 64-bit integer.
- If both the filter operand and array element are of type integer, TotalView converts the values to type 64-bit integer.

TotalView conversion operations modify a copy of the array's elements—conversions never alter the actual array elements.

To stop filtering an array, delete the contents of the **Filter** field in the Variable Window and press Enter. TotalView then updates the Variable Window so that it includes all elements.

Filtering by Comparison

The simplest filters are ones whose formats are as follows:

operator value

where *operator* is either a C/C++ or Fortran-style comparison operator, and *value* is a signed or unsigned integer constant or a floating-point number. For example, the filter for displaying all values greater than 100 is:

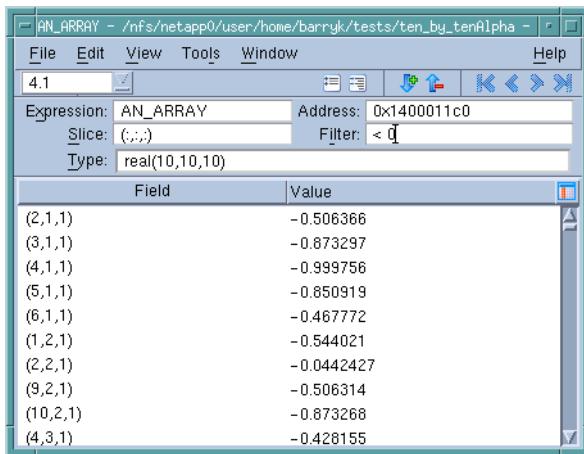
> 100

The following table lists the comparison operators:

Comparison	C/C++ Operator	Fortran Operator
Equal	<code>==</code>	.eq.
Not equal	<code>!=</code>	.ne.
Less than	<code><</code>	.lt.
Less than or equal	<code><=</code>	.le.
Greater than	<code>></code>	.gt.
Greater than or equal	<code>>=</code>	.ge.

The following figure shows an array whose filter is `< 0`. This tells TotalView to only display array elements whose value is less than 0 (zero). See Figure 214 on page 339.

Figure 214: Array Data Filtering by Comparison



If the *value* you’re using in the comparison is an integer constant, TotalView performs a signed comparison. If you add the letter **u** or **U** to the constant, TotalView performs an unsigned comparison.

Filtering for IEEE Values

You can filter IEEE NaN, Infinity, or denormalized floating-point values by specifying a filter in the following form:

operator ieee-tag

The only comparison operators you can use are *equal* and *not equal*.

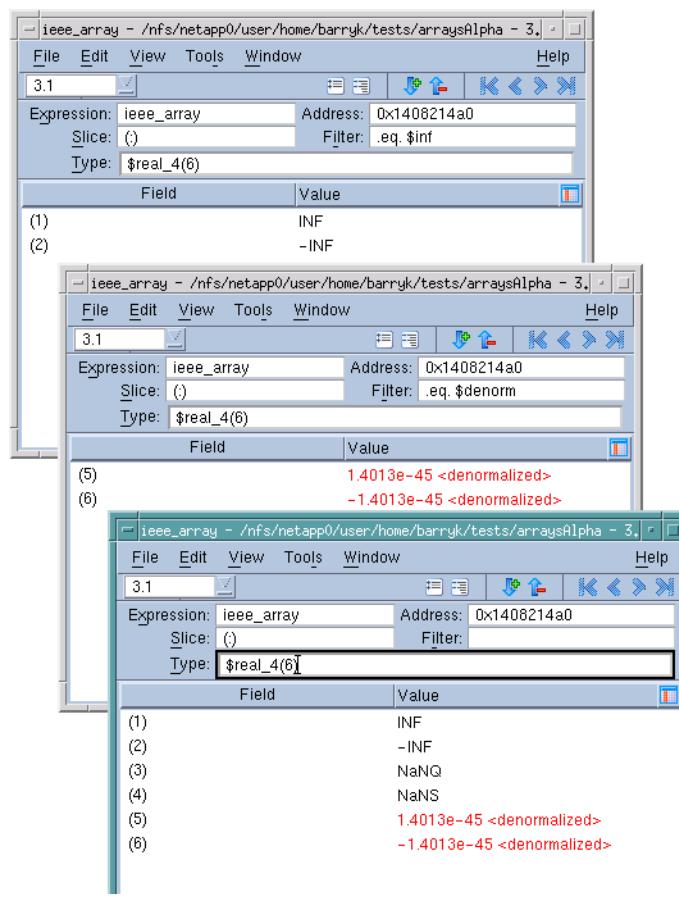
The *ieee-tag* represents an encoding of IEEE floating-point values, as the following table describes:

IEEE Tag Value	Meaning
\$nan	NaN (Not a number), either quiet or signaling
\$nanq	Quiet NaN
\$nans	Signaling NaN
\$inf	Infinity, either positive or negative
\$pinf	Positive Infinity
\$ninf	Negative Infinity
\$denorm	Denormalized number, either positive or negative
\$pdenorm	Positive denormalized number
\$ndenorm	Negative denormalized number

Figure 215 on page 340 shows an example of filtering an array for IEEE values. The bottom window in this figure shows how TotalView displays the unfiltered array. Notice the NaNQ, and NaNs, INF, and -INF values. The other two windows show filtered displays: the top window shows only infinite values; the remaining window only shows the values of denormalized numbers.

Examining and Analyzing Arrays

Figure 215: Array Data Filtering for IEEE Values



If you are writing an expression, you can use the following Boolean functions to check for a particular type of value:

IEEE Intrinsic	Meaning
\$is_denorm(value)	Is a denormalized number, either positive or negative
\$is_finite(value)	Is finite
\$is_inf(value)	Is Infinity, either positive or negative
\$is_nan(value)	Is a NaN (Not a number), either quiet or signaling
\$is_ndenorm(value)	Is a negative denormalized number
\$is_ninf(value)	Is negative Infinity
\$is_nnrm(value)	Is a negative normalized number
\$is_norm(value)	Is a normalized number, either positive or negative
\$is_nzero(value)	Is negative zero
\$is_pdenorm(value)	Is a positive denormalized number
\$is_pinf(value)	Is positive Infinity
\$is_pnorm(value)	Is a positive normalized number
\$is_pzero(value)	Is positive zero
\$is_qnan(value)	Is a quiet NaN
\$is_snan(value)	Is a signaling NaN
\$is_zero(value)	Is zero, either positive or negative

Filtering a Range of Values

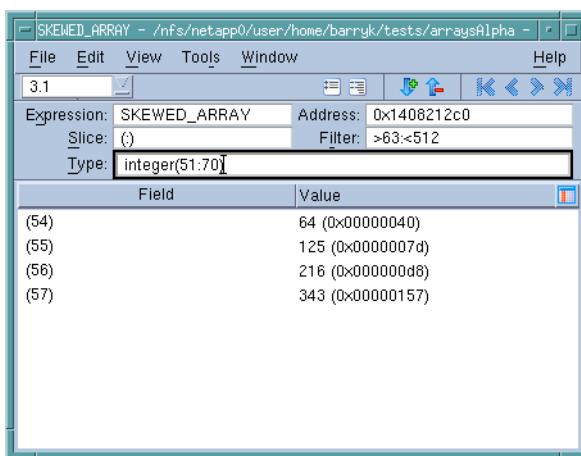
You can also filter array values by specifying a range, as follows:

`[>] low-value : [<] high-value`

where *low-value* specifies the lowest value to include, and *high-value* specifies the highest value to include, separated by a colon. The high and low values are inclusive unless you use less-than (`<`) and greater-than (`>`) symbols. If you specify a `>` before *low-value*, the low value is exclusive. Similarly, a `<` before *high-value* makes it exclusive.

The values of *low-value* and *high-value* must be constants of type integer, unsigned integer, or floating point. The data type of *low-value* must be the same as the type of *high-value*, and *low-value* must be less than *high-value*. If *low-value* and *high-value* are integer constants, you can append the letter `u` or `U` to the value to force an unsigned comparison. The following figure shows a filter that tells TotalView that to only display values greater than 63, but less than 512. (See Figure 216.)

Figure 216: Array Data Filtering by Range of Values



Creating Array Filter Expressions

The filtering capabilities described in the previous sections are those that you use most often. In some circumstances, you may need to create a more general expression. When you create a filter expression, you're creating a Fortran or C Boolean expression that TotalView evaluates for every element in the array or the array slice. For example, the following expression displays all array elements whose contents are greater than 0 and less than 50, or greater than 100 and less than 150:

```
($value > 0 && $value < 50) ||
($value > 100 && $value < 150)
```

Here's the Fortran equivalent:

```
($value .gt. 0 && $value .lt. 50) .or.
($value .gt. 100 .and. $value .lt. 150)
```

The `$value` variable is a special TotalView variable that represents the current array element. You can use this value when creating expressions.

Notice how the **and** and **or** operators are used in these expressions. The way in which TotalView computes the results of an expression is identical to the way it computes values at an eval point. For more information, see "Defining Eval Points and Conditional Breakpoints" on page 366.

Using Filter Comparisons

TotalView provides several different ways to filter array information. For example, the following two filters display the same array items:

```
> 100  
$value > 100
```

The following filters display the same array items:

```
>0:<100  
$value > 0 && $value < 100
```

The only difference is that the first method is easier to type than the second, so you're more likely to use the second method when you're creating more complicated expressions.

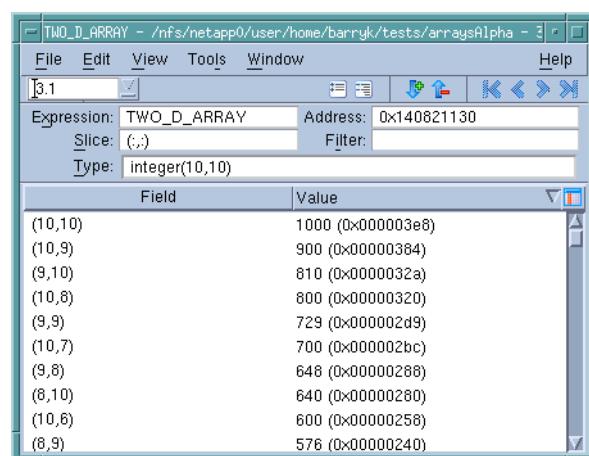
Sorting Array Data

TotalView lets you sort the displayed array data into ascending or descending order. (It does not sort the actual data.) To sort (or remove the sort), click the **Value** label.

- The first time you click, TotalView sorts the array's values into ascending order.
- The next time you click on the header, TotalView reverses the order, sorting the array's values into descending order.
- If you click again on the header, TotalView returns the array to its unsorted order.

Here is an example that sorts an array into descending order:

Figure 217: Sorted Variable Window



When you sort an array's values, you are just rearranging the information that's displayed in the Variable Window. Sorting does not change the order

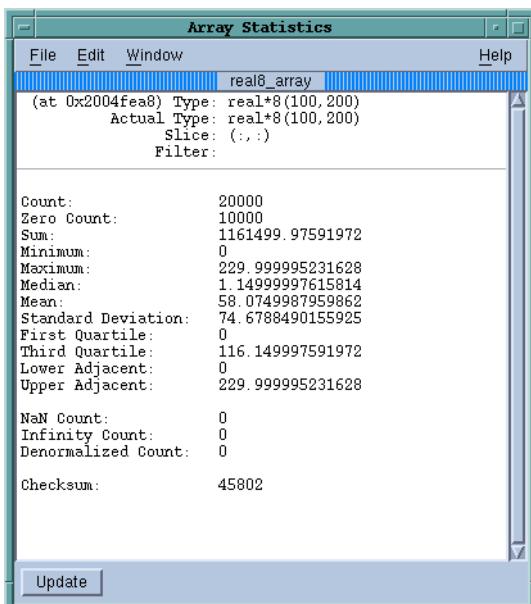
in which values are stored in memory. If you alter what TotalView is displaying by using a filter or a slice, TotalView just sorts the values that could be displayed; it doesn't sort all of the array.

If you are displaying the array created by a **Show across** command—see “*Displaying a Variable in all Processes or Threads*” on page 345 for more information—you can sort your information by process or thread.

Obtaining Array Statistics

The **Tools > Statistics** command displays a window that contains information about your array. Figure 218 shows an example.

Figure 218: Array Statistics Window



If you have added a filter or a slice, these statistics describe only the information currently being displayed; they do not describe the entire unfiltered array. For example, if 90% of an array's values are less than 0 and you filter the array to show only values greater than 0, the median value is positive even though the array's real median value is less than 0.

TotalView displays the following statistics:

■ Checksum

A checksum value for the array elements.

■ Count

The total number of displayed array values. If you're displaying a floating-point array, this number doesn't include NaN or Infinity values.

■ Denormalized Count

A count of the number of denormalized values found in a floating-point array. This includes both negative and positive denormalized values as defined in the IEEE floating-point standard. Unlike other floating-point statistics, these elements participate in the statistical calculations.

■ **Infinity Count**

A count of the number of infinity values found in a floating-point array. This includes both negative and positive infinity as defined in the IEEE floating-point standard. These elements don't participate in statistical calculations.

■ **Lower Adjacent**

This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first quartile value minus the value of 1.5 times the difference between the first and third quartiles.

■ **Maximum**

The largest array value.

■ **Mean**

The average value of array elements.

■ **Median**

The middle value. Half of the array's values are less than the median, and half are greater than the median.

■ **Minimum**

The smallest array value.

■ **NaN Count**

A count of the number of NaN (not a number) values found in a floating-point array. This includes both signaling and quiet NaNs as defined in the IEEE floating-point standard. These elements don't participate in statistical calculations.

■ **Quartiles, First and Third**

Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the third quartile value means that 75% of the array's values are less than this value and 25% are greater.

■ **Standard Deviation**

The standard deviation for the array's values.

■ **Sum**

The sum of all the displayed array's values.

■ **Upper Adjacent**

This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus the value of 1.5 times the difference between the first and third quartiles.

■ **Zero Count**

The number of elements whose value is 0.

Displaying a Variable in all Processes or Threads

When you're debugging a parallel program that is running many instances of the same executable, you usually need to view or update the value of a variable in all of the processes or threads at once.

Before displaying a variable's value in all threads or processes, you must display an instance of the variable in a Variable Window. After TotalView displays this window, use one of the following commands:

- **View > Show Across > Process**, displays the value of the variable in all processes.
- **View > Show Across > Thread**, displays the value of a variable in all threads within a single process.
- **View > Show Across > None**, returns the window to what it was before you used other Show Across commands.



You cannot simultaneously Show Across processes and threads in the same Variable Window.

After using one of these commands, the Variable Window switches to an array-like display of information, and displays the value of the variable in each process or thread. Figure 219 shows a simple, scalar variable in each of the processes in an OpenMP program.

Figure 219: Viewing Across Threads

The screenshot shows the TotalView Variable Window titled "source - main - 1.1". The window has a menu bar with File, Edit, View, Tools, Window, and Help. Below the menu is a toolbar with various icons. The main area has input fields for Expression (source), Address (Multiple), Slice, and Filter, and a dropdown for Type (int). A table below shows the variable "source" across 8 processes (mismatchLinux.0 to mismatchLinux.7). The table has columns for Process and Value.

Process	Value
mismatchLinux.0	0x00000001 (1)
mismatchLinux.1	0x00000000 (0)
mismatchLinux.2	0x00000000 (0)
mismatchLinux.3	0x00000000 (0)
mismatchLinux.4	0x00000000 (0)
mismatchLinux.5	0x00000000 (0)
mismatchLinux.6	0x00000000 (0)
mismatchLinux.7	0x00000000 (0)

When looking for a matching stack frame, TotalView matches frames starting from the top frame, and considers calls from different memory or stack locations to be different calls. For example, the following definition of

Displaying a Variable in all Processes or Threads

`recurse()` contains two additional calls to `recurse()`. Each of these calls generates a nonmatching call frame.

```
void recurse(int i) {
    if (i <= 0)
        return;
    if (i & 1)
        recurse(i - 1);
    else
        recurse(i - 1);
}
```

If the variables are at different addresses in the different processes or threads, the field to the left of the **Address** field displays **Multiple**, and the unique addresses appear with each data item.

TotalView also lets you Show Across arrays and structures. When you Show Across an array, TotalView displays each element in the array across all processes. You can use a slice to select elements to be displayed in an "across" display. The following figure shows the result of applying a **Show Across > Processes** command to an array of structures. (See Figure 220.)

Figure 220: Viewing across an Array of Structures

The screenshot shows the TotalView application window titled "status - main - 1.1". The menu bar includes File, Edit, View, Tools, Window, and Help. The toolbar has icons for file operations. The status bar shows "I.1" and "Address: Multiple". The main area has an expression bar with "Expression: status" and "Type: MPI_Status". A table displays the data:

Field	Type	Process	Value
count	int	mismatchLinux.0	0x00000000 (0)
MPI_SOURCEint		mismatchLinux.0	0x00000000 (0)
MPI_TAG	int	mismatchLinux.0	0x00000000 (0)
MPI_ERRORint		mismatchLinux.0	0x00000000 (0)
count	int	mismatchLinux.1	0x00000000 (0)
MPI_SOURCEint		mismatchLinux.1	0x00000000 (0)
MPI_TAG	int	mismatchLinux.1	0x00000000 (0)
MPI_ERRORint		mismatchLinux.1	0x00000000 (0)
count	int	mismatchLinux.2	0x00000000 (0)
MPI_SOURCEint		mismatchLinux.2	0x00000000 (0)
MPI TAG	int	mismatchLinux.2	0x00000000 (0)

Diving on a "Show Across" Pointer

You can dive through pointers in a Show Across display. This dive applies to the associated pointer in each process or thread.

Editing a "Show Across" Variable

If you edit a value in a "Show Across" display, TotalView asks if it should apply this change to all processes or threads or only the one in which you made a change. This is an easy way to update a variable in all processes.

Visualizing Array Data

The Visualizer lets you create graphical images of array data. This presentation lets you see your data in one glance and can help you quickly find problems with your data while you are debugging your programs.

You can execute the Visualizer from within TotalView, or you can run it from the command line to visualize data dumped to a file in a previous TotalView session.

For information about running the Visualizer, see Chapter 9, "Visualizing Programs and Data," on page 179.

Visualizing a "Show Across" Variable Window

You can export data created by using a *Show Across* command to the Visualizer by using the **Tools > Visualize** command. When visualizing this kind of data, the process (or thread) index is the first axis of the visualization. This means that you must use one less data dimension than you normally would. If you do not want the process/thread axis to be significant, you can use a normal Variable Window, since all of the data must be in one process.



This chapter explains how to use action points. TotalView has four kinds of action points:

- A *breakpoint* stops execution of processes and threads that reach it.
- A *barrier point* synchronizes a set of threads or processes at a location.
- An *eval point* causes a code fragment to execute when it is reached.
- A *watchpoint* lets you monitor a location in memory and stop execution when it changes.

This chapter contains the following sections:

- "About Action Points" on page 349
- "Setting Breakpoints and Barriers" on page 351
- "Defining Eval Points and Conditional Breakpoints" on page 366
- "Using Watchpoints" on page 373
- "Saving Action Points to a File" on page 380

About Action Points

Action points let you specify an action for TotalView to perform when a thread or process reaches a source line or machine instruction in your program. The different kinds of action points that you can use are shown in Figure 221 on page 350.

■ Breakpoints

When a thread encounters a breakpoint, it stops at the breakpoint. Other threads in the process also stop. You can indicate that you want other related processes to stop, as well. Breakpoints are the simplest kind of action point.

■ **Barrier points**

Barrier points are similar to simple breakpoints, differing in that you use them to synchronize a group of processes or threads. A barrier point holds each thread or process that reaches it until all threads or processes reach it. Barrier points work together with the TotalView hold-and-release feature. TotalView supports thread barrier and process barrier points.

■ **Eval points**

An eval point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an eval point, it executes this code. You can use eval points in a variety of ways, including conditional breakpoints, thread-specific breakpoints, countdown breakpoints, and patching code fragments into and out of your program.

■ **Watchpoints**

A watchpoint tells TotalView to either stop the thread so that you can interact with your program (unconditional watchpoint), or evaluate an expression (conditional watchpoint).

All action points share the following common properties.

- You can independently enable or disable action points. A disabled action isn't deleted; however, when your program reaches a disabled action point, TotalView ignores it.
- You can share action points across multiple processes or set them in individual processes.
- Action points apply to the process. In a multi-threaded process, the action point applies to all of the threads contained in the process.
- TotalView assigns unique ID numbers to each action point. These IDs appear in several places, including the Root Window, the Action Points Tab of the Process Window, and the **Action Point > Properties** Dialog Box.

The following figure shows the symbol that TotalView displays for an action point::.

CLI: **a**ctions shows information about action points.

Figure 221: Action Point Symbols

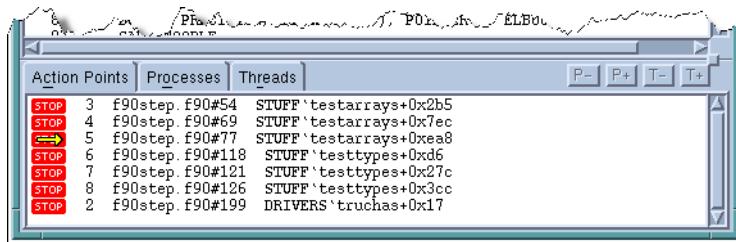
85	
ASM	Assembler-level action point
STOP	Breakpoint
STOP	Disabled breakpoint
BARR	Barrier breakpoint
BARR	Disabled barrier breakpoint
EVAL	Eval point
EVAL	Disabled eval point
93	
94	

The  icon is what TotalView displays when you create a breakpoint on an assembler statement.

CLI: All action points display as "@" when you use the dlist command to display your source code. Use the dactions command to see what type of action point is set.

When your program halts because it encountered an action point, TotalView lets you know what has happened in several ways. In the Root Window, the status is displayed with the letter "b" followed by a number. This is the same number that you will see in the Action Points tab within the Process Window. In the Process Window, the status lines above the Source Pane also let you know that the thread is at a breakpoint. Finally, TotalView places a yellow arrow over the action point's icon in the Action Point tab. For example:

Figure 222: Action Point Tab



If you are working with templated code, you will see ellipses (...) after the address. These ellipses indicate that there are additional addresses associated with the breakpoint.

Setting Breakpoints and Barriers

TotalView has several options for setting breakpoints. You can set:

- Source-level breakpoints
- Breakpoints that are shared among all processes in multi-process programs
- Assembler-level breakpoints

You can also control whether TotalView stops all processes in the control group when a single member reaches a breakpoint.

Topics in this section are:

- “*Setting Source-Level Breakpoints*” on page 352
- “*Setting Breakpoints at Locations*” on page 353
- “*Displaying and Controlling Action Points*” on page 355

- "Setting Machine-Level Breakpoints" on page 358
- "Setting Breakpoints for Multiple Processes" on page 359
- "Setting Breakpoints When Using the fork()/execve() Functions" on page 361
- "Setting Barrier Points" on page 362

Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a boxed line number in the Process Window. (A boxed line number indicates that the line is associated with executable code.) A **STOP** icon lets you know that a breakpoint is set immediately before the source statement.

CLI: @ next to the line number

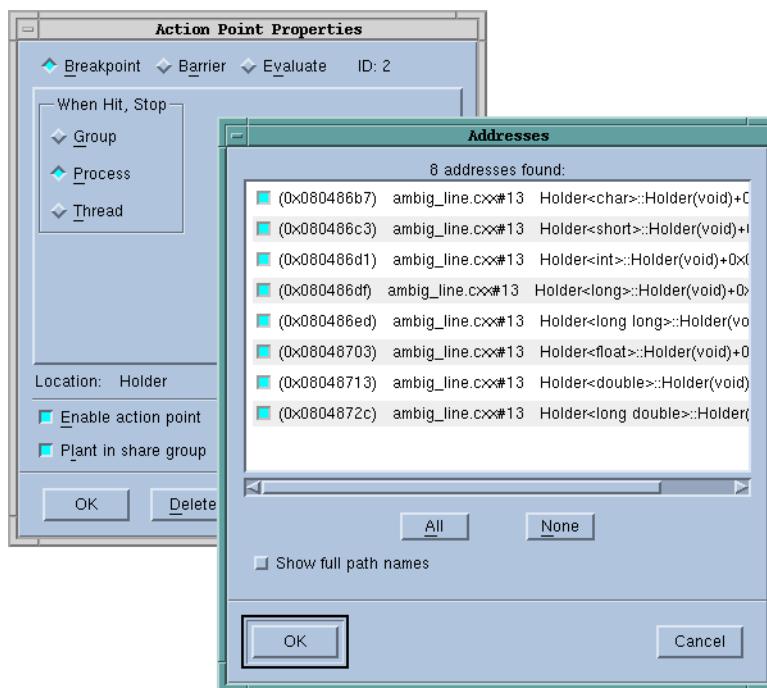
You can also set a breakpoint while a process is running by selecting a boxed line number in the Process Window.

CLI: Use dbreak whenever the CLI displays a prompt.

Choosing Source Lines

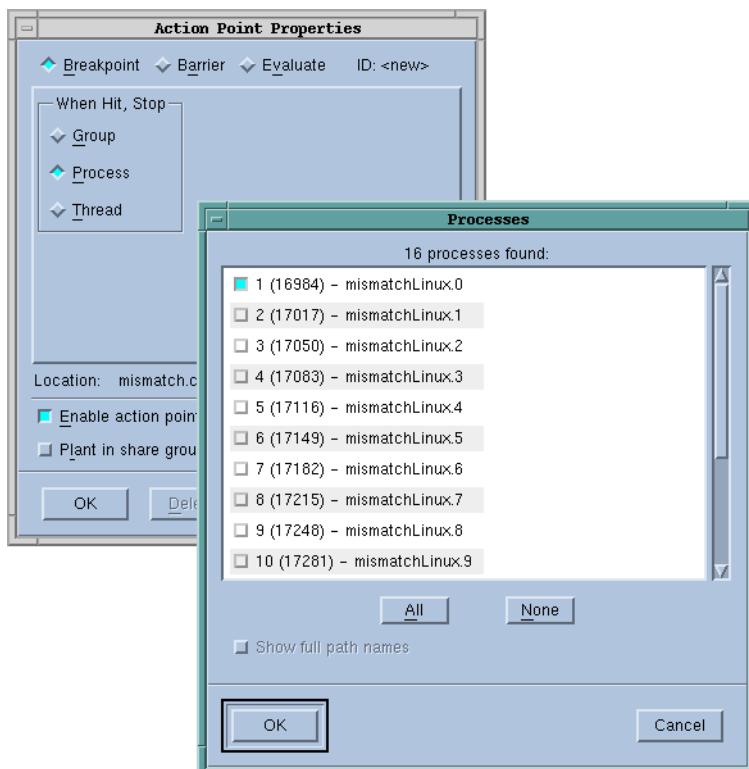
If you're using C++ templates, TotalView sets a breakpoint in all instantiations of that template. If this isn't what you want, clear the button and then select the **Addresses** button in the Action Point Properties Dialog Box. You can now clear locations where the action point shouldn't be set. (See Figure 223.)

Figure 223: Setting Breakpoints on Multiple Similar Addresses



Similarly, in a multi-process program, you might not want to set the breakpoint in all processes. If this is the case, select the **Process** button. (See Figure 224.)

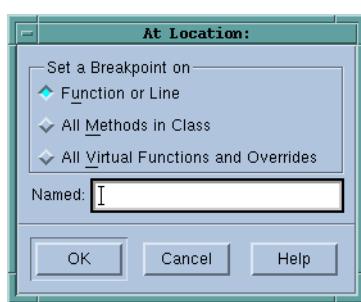
Figure 224: Setting Breakpoints on Multiple Similar Addresses and on Processes



Setting Breakpoints at Locations

You can set or delete a breakpoint at a specific function or source-line number without having to first find the function or source line in the Source Pane. Do this by entering a line number or function name in the **Action Point > At Location** Dialog Box. (Figure 225.)

Figure 225: Action Point > At Location Dialog Box



When you're done, TotalView sets a breakpoint at the location. If you enter a function name, TotalView sets the breakpoint at the function's first executable line. In either case, if a breakpoint already exists at a location, TotalView deletes it.

CLI: dbreak sets a breakpoint
ddelete deletes a breakpoint

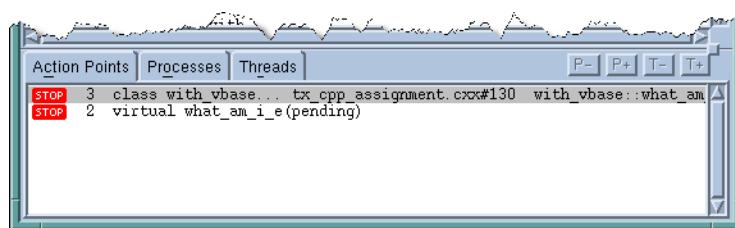
For detailed information about the kinds of information you can enter in this dialog box, see **dbreak** in the *TotalView Reference Guide*

Ambiguous Functions and Pending Breakpoints

If you type a function name that TotalView has no information about into the **Action Point > At Location** dialog box, it assumes that you have either mistyped the function name or that the library containing the function has not yet been loaded into memory.

If TotalView cannot find a location to set a breakpoint (or a barrier point), you can tell it to set it anyway because it could exist in a shared library or it could be loaded later. These kind of breakpoints are called *pending breakpoints*. When libraries are loaded, TotalView checks for the function's name. If the name is found, it sets the breakpoint. If it isn't in a newly library, TotalView just keeps on waiting for it to be loaded. You'll see information in the Action Points tab that tells you that the breakpoint is pending.

Figure 226: Pending Breakpoints



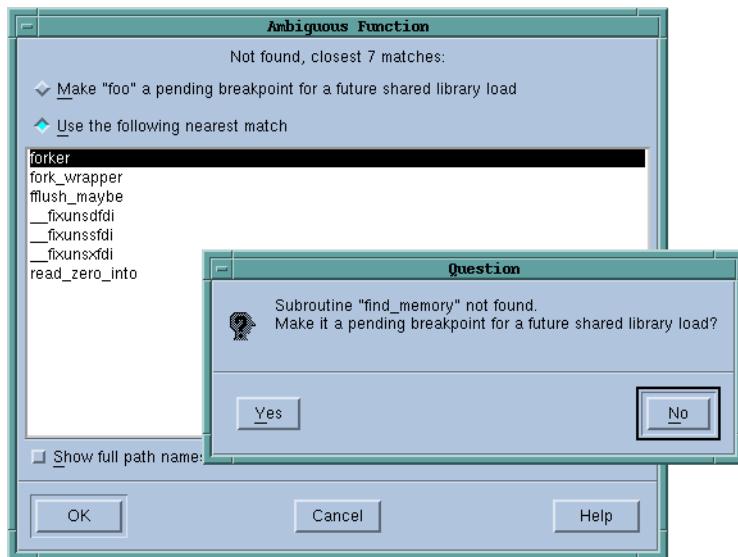
If the name you type is similar to the name of an existing function, TotalView displays its **Ambiguous Function** dialog box that lets you select which of these existing functions it should set a breakpoint on. If, however, the function will be loaded into memory later, you can set a pending breakpoint (See Figure 227 on page 355.).

If the name you entered was not ambiguous, TotalView just asks if it should set a pending breakpoint. This question box is also shown in Figure 227 on page 355.



TotalView can only place one action point on an address. Because the breakpoints you specify are actually expressions, the locations to which these expressions evaluate can overlap or even be the same. Sometimes, and this most often occurs with pending breakpoints in dynamically loaded libraries, TotalView cannot tell when action points over-

Figure 227: Ambiguous Function Dialog Box

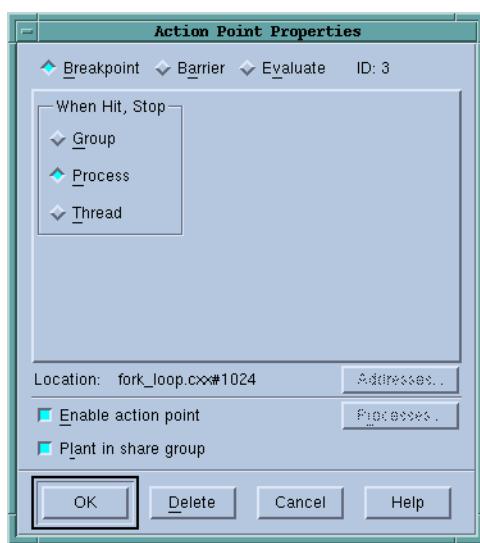


lap. If they do, TotalView only enables one of the action points and disables all others that evaluate to the same address. The actionpoint that TotalView enables is the one with the lowest actionpoint ID.

Displaying and Controlling Action Points

The Action Point > Properties Dialog Box lets you set and control an action point. Controls in this dialog box also lets you change an action point's type to breakpoint, barrier point, or eval point. You can also define what happens to other threads and processes when execution reaches this action point. (See Figure 228.)

Figure 228: Action Point > Properties Dialog Box



The following sections explain how you can control action points by using the Process Window and the **Action Point > Properties** Dialog Box.

```
CLI: dset SHARE_ACTION_POINT  
      dset STOP_ALL  
      ddisable action-point
```

Disabling Action Points

TotalView can retain an action point's definition and ignore it while your program is executing. That is, disabling an action point deactivates it without removing it.

```
CLI: ddisable action-point
```

You can disable an action point by:

- Clearing **Enable action point** in the **Action Point > Properties** Dialog Box.
- Selecting the **STOP** or **BARR** symbol in the Action Points Tab.
- Using the context (right-click) menu.
- Clicking on the **Action Points > Disable** command.

Deleting Action Points

You can permanently remove an action point by selecting the **STOP** or **BARR** symbol or selecting the **Delete** button in the **Action Point > Properties** Dialog Box.

To delete all breakpoints and barrier points, use the **Action Point > Delete All** command.

```
CLI: ddelete
```

If you make a significant change to the action point, TotalView disables it rather than delete it when you click the symbol.

Enabling Action Points

You can activate a previously disabled action point by selecting a dimmed **STOP**, **BARR**, or **EVAL** symbol in the Source or Action Points tab, or by selecting **Enable action point** in the **Action Point > Properties** Dialog Box.

```
CLI: denable
```

Suppressing Action Points

You can tell TotalView to ignore action points by using the **Action Point > Suppress All** command.

```
CLI: ddisable -a
```

When you suppress action points, you disable them. After you suppress an action point, TotalView changes the symbol it uses within the Source Panes line number area. In all cases, the icon's color will be lighter. If you have suppressed action points, you cannot update existing action points or create new ones.

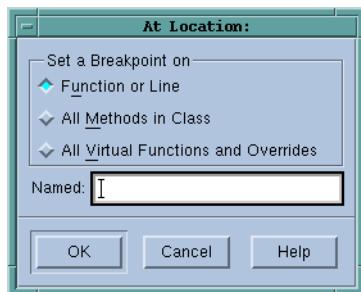
You can make previously suppressed action points active and allow the creation of new ones by reselecting the **Action Point > Suppress All** command.

CLI: denable -a

Setting Breakpoints on Classes and Virtual and Overloaded Functions

The **Action Point > At Location** dialog box lets you set breakpoints on all functions within a class or on a virtual function. The **All Methods in Class** and **All Virtual Functions and Overrides** check boxes tell TotalView that it should set multiple breakpoints. Each place that TotalView sets a breakpoint will have its own breakpoint icon. For example, if there are ten class functions, each will have its own unique breakpoint.

Figure 229: Action Point > At Location Dialog Box



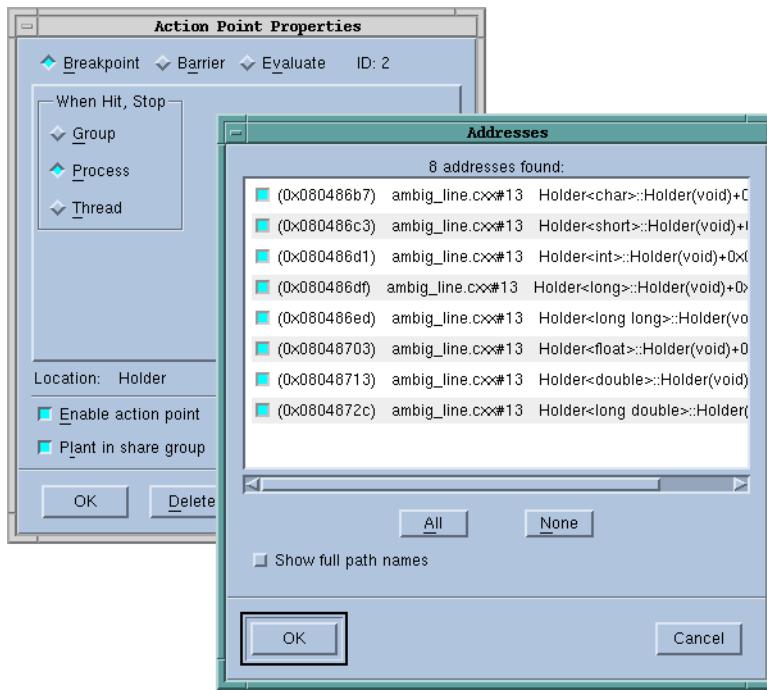
TotalView tells you that the action point is set on a virtual function or a class in the Action Points tab. If you dive on the action point in this tab, TotalView brings up its Ambiguous Function dialog box so that you can select which it should display. You may want to select the **Show full path names** check box if you can't tell which you want from the function's signature.

If a function name is overloaded, the debugger sets a breakpoint on each of these functions.

If you only want breakpoints on some functions, you will need to select the breakpoint and then get to the Properties Window. Do this either by right-clicking on the breakpoint and press **Properties** or by selecting the **Action Point > Properties** command, and then press **Addresses**. (See Figure 230 on page 358.)

You can now individually add or remove breakpoints.

Figure 230: Action Point > Properties: Selecting



Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code. (For information, see "Viewing the Assembler Version of Your Code" on page 171.) You can now select an instruction. TotalView replaces some line numbers with a dotted box (:::)—this indicates the line is the beginning of a machine instruction. If a line has a line number, this is the line number that appears in the Source Pane. Since instruction sets on some platforms support variable-length instructions, you might see a different number of lines associated with a single line contained in the dotted box. The **STOP** icon appears, indicating that the breakpoint occurs before the instruction executes.

If you set a breakpoint on the first instruction after a source statement, however, TotalView assumes that you are creating a source-level breakpoint, not an assembler-level breakpoint.

Figure 231: Breakpoint at Assembler Instruction

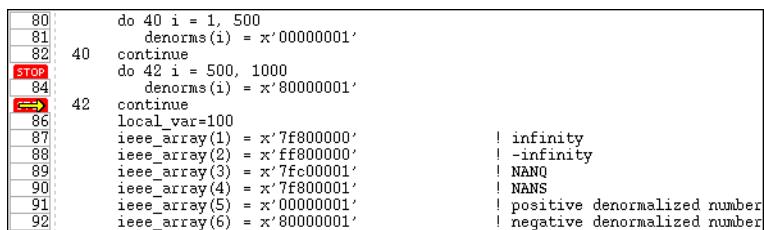
116	0x0804b213:	0xeb	jmp	0x0804b215
	0x0804b214:	0x00		
117	0x0804b215:	0x89	movl	%ebp,%esp
	0x0804b216:	0xec		
...	0x0804b217:	0x5d	popl	%ebp
...	0x0804b218:	0xc3	ret	
...	0x0804b219:	0xd8	leal	0(%esi),%esi
	0x0804b21a:	0x76		
	0x0804b21b:	0x00		
	MB_fork_notify_breakpoint_here:	0x55	pushl	%ebp
	0x0804b21d:	0x89	movl	%esp,%ebp
124	0x0804b21e:	0xe5		
	0x0804b21f:	0xeb	jmp	0x0804b221
	0x0804b220:	0x00		
125	0x0804b221:	0x89	movl	%ebp,%esp
	0x0804b222:	0xec		
...	0x0804b223:	0x5d	popl	%ebp
...	0x0804b224:	0xc3	ret	
...	0x0804b225:	0xd8	leal	0(%esi),%esi
	0x0804b226:	0x76		

If you set machine-level breakpoints on one or more instructions generated from a single source line, and then display source code in the Source Pane, TotalView displays an **ASM** icon (see Figure 221 on page 350) on the line number. To see the actual breakpoint, you must redisplay assembler instructions.

When a process reaches a breakpoint, TotalView does the following:

- Suspends the process.
- Displays the PC arrow icon (➡) over the stop sign to indicate that the PC is at the breakpoint.

Figure 232: PC Arrow Over a Stop Icon



```

80      do 40 i = 1, 500
81          denorms(i) = x'00000001'
82      continue
83  40      do 42 i = 500, 1000
84          denorms(i) = x'80000001'
85      continue
86      local_var=100
87      ieee_array(1) = x'7f800000' ! infinity
88      ieee_array(2) = x'ff800000' ! -infinity
89      ieee_array(3) = x'7fc00001' ! NANQ
90      ieee_array(4) = x'7f800001' ! NANS
91      ieee_array(5) = x'00000001' ! positive denormalized number
92      ieee_array(6) = x'80000001' ! negative denormalized number

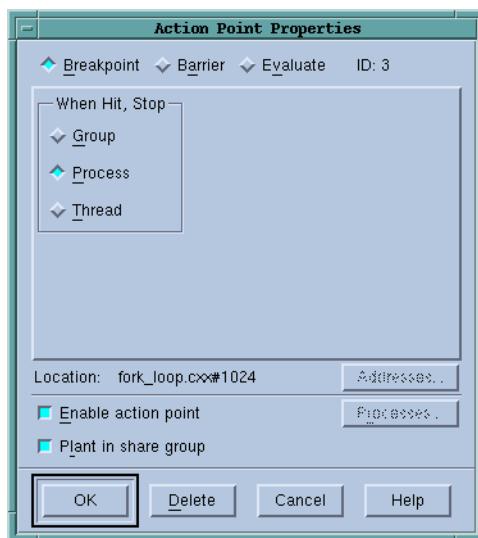
```

- Displays **At Breakpoint** in the Process Window title bar and other windows.
- Updates the Stack Trace and Stack Frame Panes and all Variable Windows.

Setting Breakpoints for Multiple Processes

In all programs, including multi-process programs, you can set breakpoints in parent and child processes before you start the program and while the program is executing. Do this using the **Action Point > Properties** Dialog Box. (See Figure 233.)

Figure 233: Action Point > Properties Dialog Box



Setting Breakpoints and Barriers

This dialog box provides the following controls for setting breakpoints:

■ When Hit, Stop

When your thread hits a breakpoint, TotalView can also stop the thread's control group or the process in which it is running.

```
CLI: dset STOP_ALL  
      dbreak -p | -g | -t
```

■ Plant in share group

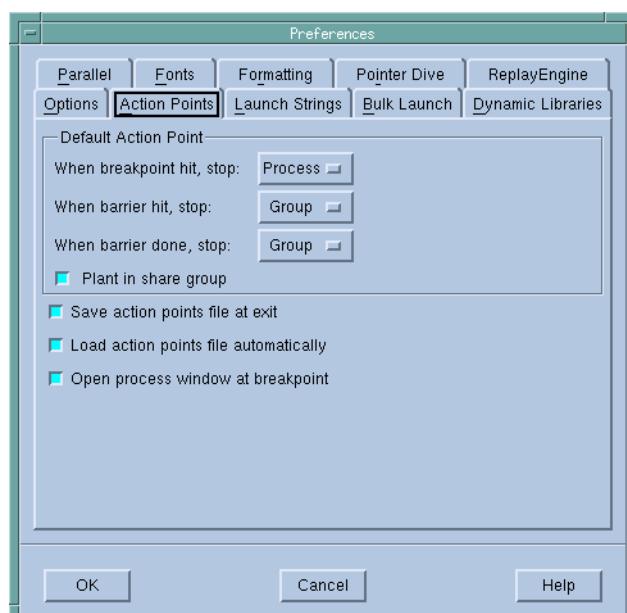
If you select this check box, TotalView enables the breakpoint in all members of this thread's share group at the same time. If not, you must individually enable and disable breakpoints in each member of the share group.

```
CLI: dset SHARE_ACTION_POINT
```

The **Processes** button lets you indicate which process in a multi-process program will have enabled breakpoints. If **Plant in share group** is selected, TotalView does not enable this button because you told TotalView to set the breakpoint in all of the processes.

You can preset many of the properties in this dialog box by selecting the **File > Preferences** command. Use the Action Points page to set action point preferences.

Figure 234: File > Preferences: Action Points Page



You can find additional information about this dialog box in the online Help.

If you select the **Evaluate** button in the **Action Point > Properties** Dialog Box, you can add an expression to the action point. This expression is

attached to control and share group members. See "Using Programming Language Elements" on page 385 for more information.

If you're trying to synchronize your program's threads, you need to set a barrier point. For more information, see "Setting Barrier Points" on page 362.

Setting Breakpoints When Using the `fork()`/`execve()` Functions

You must link with the dbfork library before debugging programs that call the `fork()` and `execve()` functions. See "Compiling Programs" on page 51.

Debugging Processes That Call the `fork()` Function

By default, TotalView places breakpoints in all processes in a share group. (For information on share groups, see "Organizing Chaos" on page 22.) When any process in the share group reaches a breakpoint, TotalView stops all processes in the control group. This means that TotalView stops the control group that contains the share group. This control can contain more than one share group.

To override these defaults:

- 1** Dive into the line number to display the **Action Point > Properties** Dialog Box.
- 2** Clear the **Plant in share group** check box and make sure that the **Group** radio button is selected.

```
CLI: dset SHARE_ACTION_POINT false
```

Debugging Processes that Call the `execve()` Function

Shared breakpoints are not set in children that have different executables.

To set the breakpoints for children that call the `execve()` function:

- 1** Set the breakpoints and breakpoint options in the parent and the children that do not call the `execve()` function.
- 2** Start the multi-process program by displaying the **Group > Go** command.

When the first child calls the `execve()` function, TotalView displays the following message:

*Process name has exec'd name.
Do you want to stop it now?*

```
CLI: G
```

- 3** Answer **Yes**.

TotalView opens a Process Window for the process. (If you answer **No**, you won't have an opportunity to set breakpoints.)

- 4** Set breakpoints for the process.

After you set breakpoints for the first child using this executable, TotalView won't prompt when other children call the `execve()` function.

This means that if you do not want to share breakpoints in children that use the same executable, dive into the breakpoints and set the breakpoint options.

- 5 Select the **Group > Go** command.

Example: Multi-process Breakpoint

The following program excerpt illustrates the places where you can set breakpoints in a multi-process program:

```
1 pid = fork();
2 if (pid == -1)
3     error ("fork failed");
4 else if (pid == 0)
5     children_play();
6 else
7     parents_work();
```

The following table describes what happens when you set a breakpoint at different places:

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes.
3	Stops the parent process if the fork() function failed.
5	Stops the child process.
7	Stops the parent process.

Setting Barrier Points

A barrier breakpoint is similar to a simple breakpoint, differing only in that it holds processes and threads that reach the barrier point. Other processes and threads continue to run. TotalView holds these processes or threads until all processes or threads defined in the barrier point reach this same place. When the last one reaches a barrier point, TotalView releases all the held processes or threads. In this way, barrier points let you synchronize your program's execution.

CLI: `dbarrier`

Topics in this section are:

- “About Barrier Breakpoint States” on page 363
- “Setting a Barrier Breakpoint” on page 363
- “Creating a Satisfaction Set” on page 364
- “Hitting a Barrier Point” on page 365
- “Releasing Processes from Barrier Points” on page 365
- “Deleting a Barrier Point” on page 365
- “Changing Settings and Disabling a Barrier Point” on page 365

About Barrier Breakpoint States

Processes and threads at a barrier point are held or stopped, as follows:

Held	A held process or thread cannot execute until all the processes or threads in its group are at the barrier, or until you manually release it. The various <i>go</i> and <i>step</i> commands from the Group , Process , and Thread menus cannot start held processes.
Stopped	When all processes in the group reach a barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you tell them to resume executing.

You can manually release held processes and threads with the **Hold** and **Release** commands found in the **Group**, **Process**, and **Thread** menus. When you manually release a process, the *go* and *step* commands become available again.

```
CLI: dfocus ... dhold
      dfocus ... dunhold
```

You can reuse the **Hold** command to again toggle the hold state of the process or thread. See "Holding and Releasing Processes and Threads" on page 231 for more information.

When a process or a thread is held, TotalView displays an **H** (for a held process) or an **h** (for a held thread) in the process's or thread's entry in the Root Window.

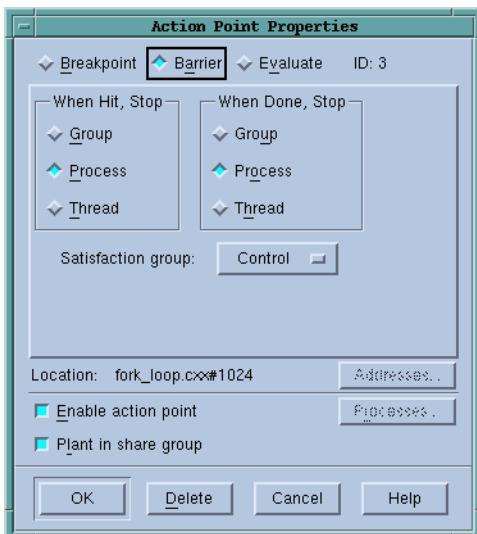
Setting a Barrier Breakpoint

You can set a barrier breakpoint by using the **Action Point > Set Barrier** command or from the **Action Point > Properties** Dialog Box. As an alternative, you can right-click on the line. From the displayed context menu, you can select the **Set Barrier** command. (See Figure 235 on page 364.)

You most often use barrier points to synchronize a set of threads. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread reaching the barrier from responding to resume commands (for example, *step*, *next*, or *go*) until all threads in the affected set arrive at the barrier. When all threads reach the barrier, TotalView considers the barrier to be *satisfied* and releases all of the threads being held there. *They are just released; they are not continued.* That is, they are left stopped at the barrier. If you continue the process, those threads also run.

If you stop a process and then continue it, the held threads, including the ones waiting at an unsatisfied barrier, do not run. Only unheld threads run.

Figure 235: Action Point > Properties Dialog Box



The **When Hit, Stop** radio buttons indicate what other threads TotalView stops when execution reaches the breakpoint, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: `dbarrier -stop_when_hit`

After all processes or threads reach the barrier, TotalView releases all held threads. *Released* means that these threads and processes can now run.

The **When Done, Stop** radio buttons tell TotalView what else it should stop, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: `dbarrier -stop_when_done`

Creating a Satisfaction Set

For even more control over what TotalView stops, you can select a *satisfaction set*. This set tells TotalView which threads must be held before it can release the group of threads. That is, the barrier is *satisfied* when TotalView has held all of the indicated threads. Use the **Satisfaction group** items to tell TotalView that the satisfaction set consists of all threads in the current thread's **Control**, **Workers**, or **Lockstep** group.

When you set a barrier point, TotalView places it in every process in the share group.

Hitting a Barrier Point

If you run one of the processes or threads in a group and it hits a barrier point, you see an **H** next to the process or thread name in the Root Window and the word [**Held**] in the title bar in the main Process Window. Barrier points are always shared.

CLI: `dstatus`

If you create a barrier and all the process's threads are already at that location, TotalView won't hold any of them. However, if you create a barrier and all of the processes and threads are not at that location, TotalView holds any thread that is already there.

Releasing Processes from Barrier Points

TotalView automatically releases processes and threads from a barrier point when they hit that barrier point and all other processes or threads in the group are already held at it.

Deleting a Barrier Point

You can delete a barrier point in the following ways:

- Use the **Action Point > Properties** Dialog Box.
- Click the **BARR** icon in the line number area.

CLI: `ddelete`

Changing Settings and Disabling a Barrier Point

Setting a barrier point at the current PC for a *stopped* process or thread holds the process there. If, however, all other processes or threads affected by the barrier point are at the same PC, TotalView doesn't hold them. Instead, TotalView treats the barrier point as if it was an ordinary breakpoint.

TotalView releases all processes and threads that are held and which have threads at the barrier point when you disable the barrier point. You can disable the barrier point in the **Action Point > Properties** Dialog Box by selecting **Enable action point** at the bottom of the dialog box.

CLI: `ddisable`

Defining Eval Points and Conditional Breakpoints

TotalView lets you define *eval points*. These are action points at which you have added a code fragment that TotalView executes. You can write the code fragment in C, Fortran, or assembler.



Assembler support is currently available on the HP Alpha Tru64 UNIX, IBM AIX, and SGI IRIX operating systems. You can enable or disable TotalView's ability to compile eval points.



When running on many AIX systems, you can speed up the performance of compiled expressions by using the `-use_aix_fast_trap` command when you start TotalView. For more information, see the TotalView Release Notes. Search for "fast trap".

Topics in this section are:

- "Setting Eval Points" on page 367
- "Creating Conditional Breakpoint Examples" on page 368
- "Patching Programs" on page 368
- "About Interpreted and Compiled Expressions" on page 370
- "Allocating Patch Space for Compiled Expressions" on page 371

You can do the following when you use eval points:

- Include instructions that stop a process and its relatives. If the code fragment can make a decision whether to stop execution, it is called a *conditional breakpoint*.
- Test potential fixes for your program.
- Set the values of your program's variables.
- Automatically send data to the Visualizer. This can produce animated displays of the changes in your program's data.

You can set an eval point at any source line that generates executable code (marked with a box surrounding a line number) or a line that contains assembler-level instructions. This means that if you can set a breakpoint, you can set an eval point.

At each eval point, TotalView or your program executes the code contained in the eval point before your program executes the code on that line. Although your program can then go on to execute this source line or instruction, it can do the following instead:

- Include a **goto** in C or Fortran that transfers control to a line number in your program. This lets you test program patches.
- Execute a TotalView function. These functions let you stop execution, create barriers, and countdown breakpoints. For more information on these statements, see "Using Built-in Variables and Statements" on page 394.

TotalView evaluates code fragments in the context of the target program. This means that you can refer to program variables and branch to places in your program.



If you call a function from in an eval point and there's a breakpoint within that function, TotalView will stop execution at that point. Similarly, if there's an eval point in the function, TotalView also evaluates that eval point.

For information on what you can include in code fragments, refer to "Using Programming Language Elements" on page 385.

Eval points only modify the processes being debugged—they do not modify your source program or create a permanent patch in the executable. If you save a program's action points, however, TotalView reapplies the eval point whenever you start a debugging session for that program. For information about how to save your eval points, see "Saving Action Points to a File" on page 380.



You should stop a process before setting an eval point in it. This ensures that the eval point is set in a stable context.

Setting Eval Points

This section contains the steps you must follow to create an eval point. These steps are as follows:

- 1 Display the **Action Point > Properties** Dialog Box. You can do this, for example, by right-clicking a **STOP** icon and selecting **Properties** or by selecting a line and then invoking the command from the menu bar.
- 2 Select the **Evaluate** button at the top of the dialog box.
- 3 Select the button (if it isn't already selected) for the language in which you plan to write the fragment.
- 4 Type the code fragment. For information on supported C, Fortran, and assembler language constructs, see "Using Programming Language Elements" on page 385.
- 5 For multi-process programs, decide whether to share the eval point among all processes in the program's share group. By default, TotalView selects the **Plant in share group** check box for multi-process programs, but you can override this by clearing this setting.
- 6 Select the **OK** button to confirm your changes.

If the code fragment has an error, TotalView displays an error message. Otherwise, it processes the code, closes the dialog box, and places an **EVAL** icon on the line number in the Source Pane.

```
CLI: dbreak -e  
      dbarrier -e
```

The variables that you refer to in your eval point can either have a global scope or be local to the block of the line that contains the eval point. If you declare a variable in the eval point, its scope is the block that contains the

eval point unless, for example, you declare it in some other scope or declare it to be a static variable.

Creating Conditional Breakpoint Examples

The following are examples that show how you can create conditional breakpoints:

- The following example defines a breakpoint that is reached whenever the **counter** variable is greater than 20 but less than 25:

```
if (counter > 20 && counter < 25) $stop;
```

- The following example defines a breakpoint that stops execution every tenth time that TotalView executes the **\$count** function

```
$count 10
```

- The following example defines a breakpoint with a more complex expression:

```
$count my_var * 2
```

When the **my_var** variable equals **4**, the process stops the eighth time it executes the **\$count** function. After the process stops, TotalView reevaluates the expression. If **my_var** equals **5**, the process stops again after the process executes the **\$count** function ten more times.

The TotalView internal counter is a “static” variable, which means that TotalView remembers its value every time it executes the eval point. Suppose you create an eval point within a loop that executes 120 times and the eval point contains **\$count 100**. Also assume that the loop is within a subroutine. As expected, TotalView stops execution the 100th time the eval point executes. When you resume execution, the remaining 20 iterations occur.

The next time the subroutine executes, TotalView stops execution after 80 iterations because it will have counted the 20 iterations from the last time the subroutine executed.

This isn't a bug that we're documenting as a feature. Suppose you have a function that is called from lots of different places from within your program. Because TotalView remembers every time a statement executes, you could, for example, stop execution every 100 times the function is called. In other words, while **\$count** is most often used within loops, you can use it outside of them as well.

For descriptions of the **\$stop**, **\$count**, and variations on **\$count**, see “*Using Built-in Variables and Statements*” on page 394.

Patching Programs

You can use expressions in eval points to patch your code if you use the **goto** (C) and **GOTO** (Fortran) statements to jump to a different program location. This lets you do the following:

- Branch around code that you don't want your program to execute.
- Add new statements.

In many cases, correcting an error means that you will do both operations: you use a goto to branch around incorrect lines and add corrections.

Branching Around Code

The following example contains a logic error where the program dereferences a null pointer:

```
1 int check_for_error (int *error_ptr)
2 {
3     *error_ptr = global_error;
4     global_error = 0;
5     return (global_error != 0);
6 }
```

The error occurs because the routine that calls this function assumes that the value of `error_ptr` can be 0. The `check_for_error()` function, however, assumes that `error_ptr` isn't null, which means that line 3 can dereference a null pointer.

You can correct this error by setting an eval point on line 3 and entering:

```
if (error_ptr == 0) goto 4;
```

If the value of `error_ptr` is null, line 3 isn't executed. Notice that you are not naming a label used in your program. Instead, you are naming one of the line numbers generated by TotalView.

Adding a Function Call

The example in the previous section routed around the problem. If all you wanted to do was monitor the value of the `global_error` variable, you can add a `printf()` function call that displays its value. For example, the following might be the eval point to add to line 4:

```
printf ("global_error is %d\n", global_error);
```

TotalView executes this code fragment before the code on line 4; that is, this line executes before `global_error` is set to 0.

Correcting Code

The following example contains a coding error: the function returns the maximum value instead of the minimum value:

```
1 int minimum (int a, int b)
2 {
3     int result; /* Return the minimum */
4     if (a < b)
5         result = b;
6     else
7         result = a;
8     return (result);
9 }
```

Correct this error by adding the following code to an eval point at line 4:

```
if (a < b) goto 7; else goto 5;
```

This effectively replaces the `if` statement on line 4 with the code in the eval point.

About Interpreted and Compiled Expressions

On all platforms, TotalView can interpret your eval points. It can compile them on HP Alpha Tru64 UNIX, IBM AIX, and SGI IRIX platforms. On HP Alpha Tru64 UNIX and IBM AIX platforms, compiling the expressions in eval points is the default.

If your platform supports compiled eval points, your performance will be significantly better, particularly if your program is using multi-processors. This is because interpreted eval points are single-threaded through the TotalView process. In contrast, compiled eval points execute on each processor.

The `TV::compile_expressions` CLI variable enables or disables compiled expressions. See "Operating Systems" in the *TotalView Reference Guide* for information about how TotalView handles expressions on specific platforms.



Using any of the following functions forces TotalView to interpret the eval point rather than compile it: \$clid, \$duid, \$nid, \$processuid, \$systid, \$tid, and \$visualize. In addition, \$pid forces interpretation on IBM AIX.

About Interpreted Expressions

Interpreted expressions are interpreted by TotalView. Interpreted expressions run slower, possibly much slower, than compiled expressions. With multi-process programs, interpreted expressions run even more slowly because processes may need to wait for TotalView to execute the expression.

When you debug remote programs, interpreted expressions always run slower because the TotalView process on the host, not the TotalView server (`tvdsvr`) on the client, interprets the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView process on the host machine to evaluate one interpreted expression. In contrast, if TotalView compiles the expression, it evaluates them on each remote process.



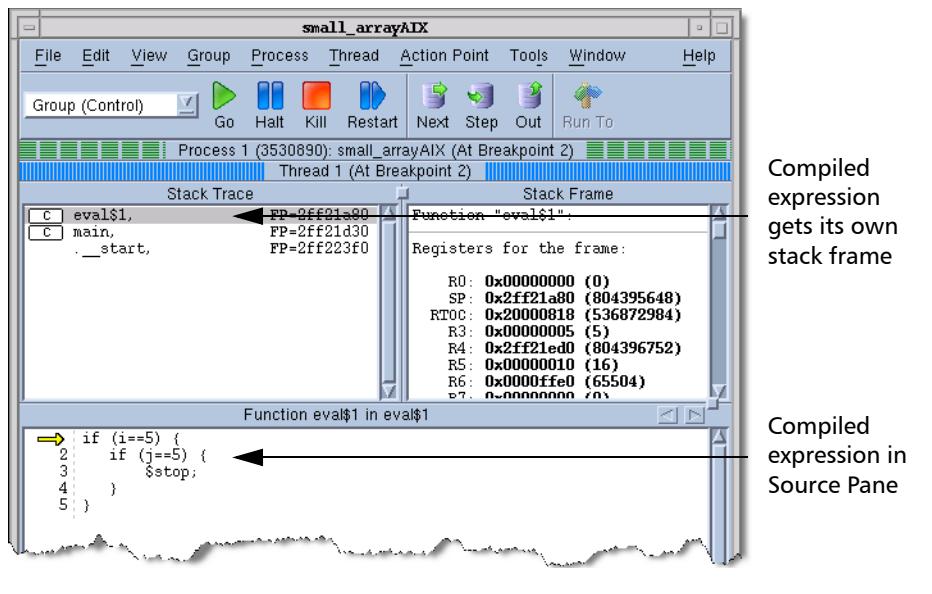
Whenever a thread hits an interpreted eval point, TotalView stops execution. This means that TotalView creates a new set of lockstep groups. Consequently, if goal threads contain interpreted patches, the results are unpredictable.

About Compiled Expressions

TotalView compiles, links, and patches expressions into the target process. Because the target thread executes this code, eval points and conditional breakpoints execute very quickly. (Conditional watchpoints are always interpreted.) Also, this code doesn't need to communicate with the TotalView host process until it needs to.

If the expression executes a **\$stop** function, TotalView stops executing the compiled expression. At this time, you can single-step through it and continue executing the expression as you would the rest of your code.

Figure 236: Stopped Execution of Compiled Expressions



If you plan to use many compiled expressions or your expressions are long, you may need to think about allocating patch space.

Allocating Patch Space for Compiled Expressions

TotalView must either allocate or find space in your program to hold the code that it generates for compiled expressions. Since this patch space is part of your program's address space, the location, size, and allocation scheme that TotalView uses might conflict with your program. As a result, you may need to change how TotalView allocates this space.

You can choose one of the following patch space allocation schemes:

- **Dynamic patch space allocation:** Tells TotalView to dynamically find the space for your expression's code.
- **Static patch space allocation:** Tells TotalView to use a statically allocated area of memory.

Allocating Dynamic Patch Space

Dynamic patch space allocation means that TotalView dynamically allocates patch space for code fragments. If you do not specify the size and location for this space, TotalView allocates 1 MB. TotalView creates this space using system calls.

TotalView allocates memory for read, write, and execute access in the addresses shown in the following table:

Platform	Address Range
HP Alpha Tru64 UNIX	0xFFFFF00000 – 0xFFFFFFFFFFFF
IBM AIX (-q32)	0xEFF00000 – 0xEFFFFFFF
IBM AIX (-q64)	0x07f0000000000000 – 0x07fffffffffffff
SGI IRIX (-n32)	0x4FF00000 – 0x4FFFFFFF
SGI IRIX (-64)	0x8FF00000 – 0x8FFFFFFF



You can only allocate dynamic patch space for the computers listed in this table.

If the default address range conflicts with your program, or you would like to change the size of the dynamically allocated patch space, you can change the following:

- *Patch space base address* by using the **-patch_area_base** command-line option.
- *Patch space length* by using the **-patch_area_length** command-line option.

Allocating Static Patch Space

TotalView can statically allocate patch space if you add a specially named array to your program. When TotalView needs to use patch space, it uses the space created for this array.

You can include, for example, a 1 MB statically allocated patch space in your program by adding the **TVDB_patch_base_address** data object in a C module. Because this object must be 8-byte aligned, declare it as an array of doubles; for example:

```
/* 1 megabyte == size TV expects */
#define PATCH_LEN 0x100000
double TVDB_patch_base_address [PATCH_LEN /
 sizeof(double)]
```

If you need to use a static patch space size that differs from the 1 MB default, you must use assembler language. The following table shows sample assembler code for three platforms that support compiled patch points:

Platform	Assembler Code
HP Alpha Tru64 UNIX	<pre>.data .align 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .byte 0x00 : PATCH_SIZE TVDB_patch_end_address:</pre>

Platform	Assembler Code
IBM AIX	.csect .data{RW}, 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .space PATCH_SIZE TVDB_patch_end_address:
SGI IRIX	.data .align 3 .globl TVDB_patch_base_address .globl TVDB_patch_end_address TVDB_patch_base_address: .space PATCH_SIZE TVDB_patch_end_address:

To use the static patch space assembler code:

- 1 Use an ASCII editor and place the assembler code into a file named **tvdb_patch_space.s**.
- 2 Replace the **PATCH_SIZE** tag with the decimal number of bytes you want. This value must be a multiple of 8.
- 3 Assemble the file into an object file by using a command such as:
cc -c tvdb_patch_space.s
On SGI IRIX, use **-n32** or **-n64** to create the correct object file type.
- 4 Link the resulting **tvdb_patch_space.o** into your program.

Using Watchpoints

TotalView lets you monitor the changes that occur to memory locations by creating a special type of action point called a *watchpoint*. You most often use watchpoints to find a statement in your program that is writing to places to which it shouldn't be writing. This can occur, for example, when processes share memory and more than one process writes to the same location. It can also occur when your program writes off the end of an array or when your program has a dangling pointer.

Topics in this section are:

- “Using Watchpoints on Different Architectures” on page 374
- “Creating Watchpoints” on page 375
- “Watching Memory” on page 377
- “Triggering Watchpoints” on page 377
- “Using Conditional Watchpoints” on page 378

TotalView watchpoints are called *modify watchpoints* because TotalView only *triggers* a watchpoint when your program modifies a memory location. If a program writes a value into a location that is the same as what is already

stored, TotalView doesn't trigger the watchpoint because the location's value did not change.

For example, if location 0x10000 has a value of 0 and your program writes a value of 0 to this location, TotalView doesn't trigger the watchpoint, even though your program wrote data to the memory location. See "Triggering Watchpoints" on page 377 for more details on when watchpoints trigger.

You can also create *conditional watchpoints*. A conditional watchpoint is similar to a conditional breakpoint in that TotalView evaluates the expression when the value in the watched memory location changes. You can use conditional watchpoints for a number of purposes. For example, you can use one to test whether a value changes its sign—that is, it becomes positive or negative—or whether a value moves above or below some threshold value.

Using Watchpoints on Different Architectures

The number of watchpoints, and their size and alignment restrictions, differ from platform to platform. This is because TotalView relies on the operating system and its hardware to implement watchpoints.



Watchpoints are not available on Macintosh computers running OS X, IBM PowerPC computers running Linux Power, and Hewlett Packard (HP) computers running HP-UX (PA-RISC).

The following list describes constraints that exist on each platform:

Computer	Constraints
HP Alpha Tru64	Tru64 places no limitations on the number of watchpoints that you can create, and no alignment or size constraints. However, watchpoints can't overlap, and you can't create a watchpoint on an already write-protected page. Watchpoints use a page-protection scheme. Because the page size is 8,192 bytes, watchpoints can degrade performance if your program frequently writes to pages that contain watchpoints.
IRIX6 MIPS	Watchpoints are implemented on IRIX 6.2 and later operating systems. These systems let you create approximately 100 watchpoints. There are no alignment or size constraints. However, watchpoints can't overlap.
IBM AIX	You can create one watchpoint on AIX 4.3.3.0-2 (AIX 4.3R) or later systems running 64-bit chips. These are Power3 and Power4 systems. (AIX 4.3R is available as APAR IY06844.) A watchpoint cannot be longer than 8 bytes, and you must align it within an 8-byte boundary. If your watchpoint is less than 8 bytes and it doesn't span an 8-byte boundary, TotalView figures out what to do. You can create compiled conditional watchpoints when you use this system. When watchpoints are compiled, they are evaluated by the process rather than having to be evaluated in TotalView where all evaluations are single-threaded and must be sent from separately executing processes. Only systems having fast traps can have compiled watchpoints.

Computer	Constraints
Linux x86,	You can create up to four watchpoints and each must be 1, 2, or 4 bytes in length, and a memory address must be aligned for the byte length. That is, you must align a 4-byte watchpoint on a 4-byte address boundary, and you must align 2-byte watchpoint on a 2-byte boundary, and so on.
Linux x86-64 (AMD and Intel)	You can create up to four watchpoints and each must be 1, 2, 4, or 8 bytes in length, and a memory address must be aligned for the byte length. For example, you must align a 4-byte watchpoint on a 4-byte address boundary.
HP-UX IA-64 and Linux IA-64	You can create up to four watchpoints. The length of the memory being watched must be a power of 2 and the address must be aligned to that power of 2; that is, (address % length) == 0 .
Solaris SPARC	TotalView supports watchpoints on Solaris 7 or later operating systems. These operating system let you create hundreds of watchpoints, and there are no alignment or size constraints. However, watchpoints can't overlap.

Typically, a debugging session doesn't use many watchpoints. In most cases, you are only monitoring one memory location at a time. Consequently, restrictions on the number of values you can watch seldom cause problems.

Creating Watchpoints

Watchpoints are created by using either the **Action Points> Create Watchpoint** command in the Process Window or the **Tools > Create Watchpoint** Dialog Box. (If your platform doesn't support watchpoints, TotalView dims this menu item.) Here are some things you should know:

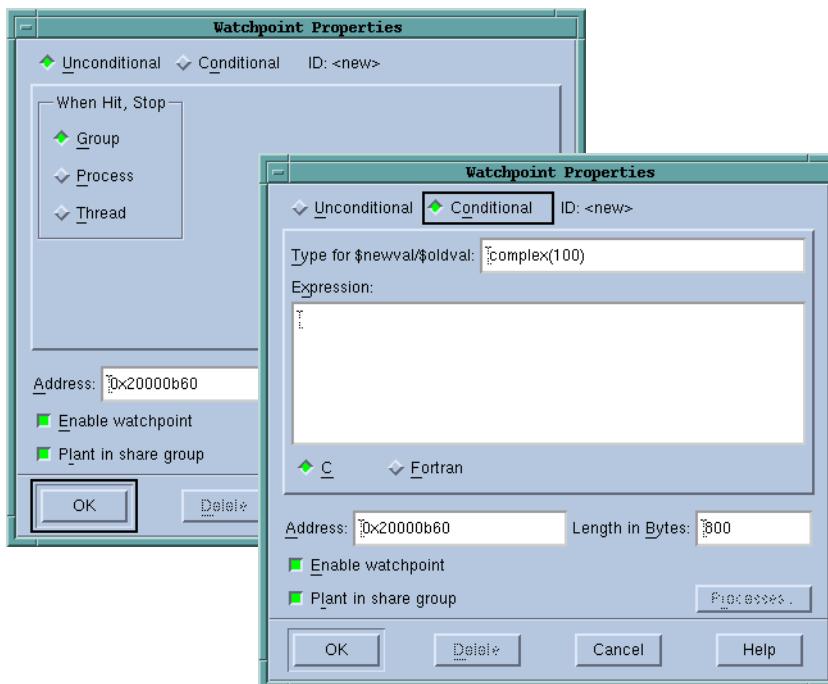
- You can also set watchpoints by right-clicking within the Process and Variable Windows and then select **Create Watchpoint** from the context menu.
- You can select an expression within the Source and Stack Frame panes and then use a context menu or select the **Action Points > Create Watchpoint** command. If you invoke either of these commands and TotalView cannot determine where to set the expression, it displays a dialog box into which you type the variable's name.
- If you select the **Tools > Create Watchpoint** command and a compound variable such an array or structure is being displayed, TotalView sets the watchpoint on the first element. However, if you select an element before invoking this command, TotalView sets the watchpoint on that element.

If you set a watchpoint on a stack variable, TotalView lets you know that you're trying to set a watchpoint on "non-global" memory. For example, the variable is on the stack or in a block and the variable will no longer exist when the stack is popped or control leaves the block. In either of these cases, it is likely that your program will overwrite the memory and the watchpoint will no longer be meaningful. See "*Watching Memory*" on page 377 for more information.

Using Watchpoints

After you select a **Create Watchpoint** command, TotalView displays its Watchpoint Properties dialog box. See Figure 237.

Figure 237: Tools > Watchpoint Dialog Boxes



Controls in this dialog box let you create unconditional and conditional watchpoints. When you set a watchpoint, you are setting it on the complete contents of the information being displayed in the Variable Window. For example, if the Variable Window displays an array, you can only set a watchpoint on the entire array (or as many bytes as TotalView can watch.) If you only want to watch one array element, dive on the element and then set the watchpoint. Similarly, if the Variable Window displays a structure and you only want to watch one element, dive on the element before you set the watchpoint.

See the online Help for information on the fields in this dialog box.

Displaying Watchpoints

The watchpoint entry, indicated by UDWP (Unconditional Data Watchpoint) and CDWP (Conditional Data Watchpoint), displays the action point ID, the amount of memory being watched, and the location being watched.

If you select a watchpoint, TotalView toggles the enabled/disabled state of the watchpoint.

Watching Memory

A watchpoint tracks a memory location—it does not track a variable. This means that a watchpoint might not perform as you would expect it to when watching stack or automatic variables. For example, suppose that you want to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is deallocated. At this time, TotalView is watching unallocated stack memory. When the stack memory is reallocated to a new stack frame, TotalView is still watching this same position. This means that TotalView triggers the watchpoint when something changes this newly allocated memory.

Also, if your program reinvokes a subroutine, it usually executes in a different stack location. TotalView cannot monitor changes to the variable because it is at a different memory location.

All of this means that in most circumstances, you shouldn't place a watchpoint on a stack variable. If you need to watch a stack variable, you will need to create and delete the watchpoint each time your program invokes the subroutine.

This doesn't mean you can't place a watchpoint on a stack or heap variable. It just means that what happens is undefined after this memory is released. For example, after you enter a routine, you can be assured that memory locations are always tracked accurately until the memory is released.



In some circumstances, a subroutine may be called from the same location. This means that its local variables might be in the same location. So, you might want to try.

If you place a watchpoint on a global or static variable that is always accessed by reference (that is, the value of a variable is always accessed using a pointer to the variable), you can set a watchpoint on it because the memory locations used by the variable are not changing.

Triggering Watchpoints

When a watchpoint triggers, the thread's program counter (PC) points to the instruction *following* the instruction that caused the watchpoint to trigger. If the memory store instruction is the last instruction in a source statement, the PC points to the source line *following* the statement that triggered the watchpoint. (Breakpoints and watchpoints work differently. A breakpoint stops *before* an instruction executes. In contrast, a watchpoint stops *after* an instruction executes.)

Using Multiple Watchpoints

If a program modifies more than one byte with one program instruction or statement, which is normally the case when storing a word, TotalView triggers the watchpoint with the lowest memory location in the modified region. Although the program might be modifying locations monitored by other watchpoints, TotalView only triggers the watchpoint for the lowest memory

location. This can occur when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these locations.

For example, suppose that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also suppose that your program uses a single instruction to store a 2-byte value at locations 0x10000 and 0x10001. If the 2-byte storage operation modifies both bytes, the watchpoint for location 0x10000 triggers. The watchpoint for location 0x10001 does not trigger.

Here's a second example. Suppose that you have a 4-byte integer that uses storage locations 0x10000 through 0x10003, and you set a watchpoint on this integer. If a process modifies location 0x10002, TotalView triggers the watchpoint. Now suppose that you're watching two adjacent 4-byte integers that are stored in locations 0x10000 through 0x10007. If a process writes to locations 0x10003 and 0x10004 (that is, one byte in each), TotalView triggers the watchpoint associated with location 0x10003. The watchpoint associated with location 0x10004 does not trigger.

Copying Previous Data Values

TotalView keeps an internal copy of data in the watched memory locations for each process that shares the watchpoint. If you create watchpoints that cover a large area of memory or if your program has a large number of processes, you increase TotalView's virtual memory requirements. Furthermore, TotalView refetches data for each memory location whenever it continues the process or thread. This can affect performance.

Using Conditional Watchpoints

If you associate an expression with a watchpoint (by selecting the **Conditional** button in the Watchpoint Properties dialog box entering an expression), TotalView evaluates the expression after the watchpoint triggers. The programming statements that you can use are identical to those used when you create an eval point, except that you can't call functions from a watchpoint expression.

The variables used in watchpoint expressions must be global. This is because the watchpoint can be triggered from any procedure or scope in your program.



Fortran does not have global variables. Consequently, you can't directly refer to your program's variables.

TotalView has two variables that are used exclusively with conditional watchpoint expressions:

\$oldval	The value of the memory locations before a change is made.
\$newval	The value of the memory locations after a change is made.

The following is an expression that uses these values:

```
if (iValue != 42 && iValue != 44) {
    inewValue = $newval; ioldValue = $oldval; $stop;}
```

When the value of the **iValue** global variable is neither 42 nor 44, TotalView stores the new and old memory values in the **inewValue** and **ioldValue** variables. These variables are defined in the program. (Storing the old and new values is a convenient way of letting you monitor the changes made by your program.)

The following condition triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And, here is a condition that triggers a watchpoint when the sign of the value in the memory location changes:

```
if ($newval * $oldval <= 0) $stop
```

Both of these examples require that you set the **Type for \$oldval/\$newval** field in the **Watchpoint Properties** Dialog Box.

For more information on writing expressions, see "Using Programming Language Elements" on page 385.

If a watchpoint has the same length as the **\$oldval** or **\$newval** data type, the value of these variables is apparent. However, if the data type is shorter than the length of the watch region, TotalView searches for the first changed location in the watched region and uses that location for the **\$oldval** and **\$newval** variables. (It aligns data in the watched region based on the size of the data's type. For example, if the data type is a 4-byte integer and byte 7 in the watched region changes, TotalView uses bytes 4 through 7 of the watchpoint when it assigns values to these variables.)

For example, suppose you're watching an array of 1000 integers called **must_be_positive**, and you want to trigger a watchpoint as soon as one element becomes negative. You declare the type for **\$oldval** and **\$newval** to be **int** and use the following condition:

```
if ($newval < 0) $stop;
```

When your program writes a new value to the array, TotalView triggers the watchpoint, sets the values of **\$oldval** and **\$newval**, and evaluates the expression. When **\$newval** is negative, the **\$stop** statement halts the process.

This can be a very powerful technique for range-checking all the values your program writes into an array. (Because of byte length restrictions, you can only use this technique on IRIX and Solaris.)



On all platforms except for IBM AIX, TotalView always interprets conditional watchpoints; it never compiles them. Because interpreted watchpoints are single-threaded in TotalView, every process or thread that writes to the watched location must wait for other instances of the watchpoint to finish executing. This can adversely affect performance.

Saving Action Points to a File

You can save a program's action points to a file. TotalView then uses this information to reset these points when you restart the program. When you save action points, TotalView creates a file named *program.TVD.breakpoints*, where *program* is the name of your program.



TotalView does not save watchpoints because memory addresses can change radically every time you restart TotalView and your program.

Use the **Action Point > Save All** command to save your action points to a file. TotalView places the action points file in the same directory as your program. In contrast, the **Action Point > Save As** command lets you name the file to which TotalView saves this information.

CLI: `dactions -save filename`

If you're using a preference to automatically save breakpoints, TotalView automatically saves action points to a file. Alternatively, starting TotalView with the **-sb** option (see "TotalView Command Syntax" in the *TotalView Reference Guide*) also tells TotalView to save your breakpoints.

At any time, you can restore saved action points if you use the **Action Points > Load All** command. After invoking this command, TotalView displays a File Explorer Window that you can use to navigate to or name the saved file.

CLI: `dactions -load filename`

You control automatic saving and loading by setting preferences. (See **File > Preferences** in the online Help for more information.)

CLI: `dset TV::auto_save_breakpoints`

Evaluating Expressions



CHAPTER
17

Whether you realize it or not, you've been telling TotalView to evaluate expressions and you've even been entering them. In every programming language, variables are actually expressions—actually they are lvalues—whose evaluation ends with the interpretation of memory locations into a displayable value. Structure, pointer and array variables, particularly arrays where the index is also a variable, are slightly more complicated.

While debugging, you also need to evaluate expressions that contain function calls and programming language elements such as **for** and **while** loops.

This chapter discusses what you can do evaluating expressions within TotalView. The topics discussed are:

- “*Why is There an Expression System?*” on page 381
- “*Using Programming Language Elements*” on page 385
- “*Using the Evaluate Window*” on page 389
- “*Using Built-in Variables and Statements*” on page 394

Why is There an Expression System?

Either directly or indirectly, accessing and manipulating data requires an evaluation system. When your program (and TotalView, of course) accesses data, it must determine where this data resides. The simplest data lookups involves two operations: looking up an address in your program's symbol table and interpreting the information located at this address based on a variable's datatype. For simple variables such as an integer or a floating point number, this is all pretty straightforward.

Why is There an Expression System?

Looking up array data is slightly more complicated. For example, if the program wants `my_var[9]`—this chapter will most often use C and C++ notation rather than Fortran—it looks up the array’s starting address, then applies an offset to locate the array’s 10 element. In this case, if each array element uses 32 bits, `my_var[9]` is located 9 times 32 bits away.

In a similar fashion, your program obtains information about variables stored in structures and arrays of structures.

Structures complicate matters slightly. For example `ptr->my_var` requires three operations: extract the data contained within address of the `my_var` variable, use this information to access the data at the address being pointed to, then display the data according to the variable’s datatype.

Accessing an array element such as `my_var[9]` where the array index is an integer constant is rare in most programs. In most cases, your program uses variables or expressions as array indices; for example, `my_var[cntr]` or `my_var[cntr+3]`. In the later case, TotalView must determine the value of `cntr+3` before it can access an array element.

Using variables and expressions as array indices are common. However, the array index can be (and often is) an integer returned by a function. For example:

```
my_var[access_func(first_var, second_var)+2]
```

In this example, a function with two arguments returns a value. That returned value is incremented by two, and the resulting value becomes the array index. Here is an illustration showing TotalView accessing the `my_var` array in the four ways discussed in this section:

Figure 238: Expression List Window: Accessing Array Elements

Expression	Value
my_var[9]	0x0000010e (270)
my_var[cntr]	0x00000168 (360)
my_var[cntr+3]	0x000001c2 (450)
my_var[access_func(first_var, second_var)+2]	0x00000582 (1410)

In Fortran and C, access to data is usually through variables with some sort of simple evaluation or a function. Access to variable information can be the same in C++ as it is in these languages. However, accessing private variables within a class almost always uses a method. For example:

```
myDataStructureList.get_current_place()
```

TotalView built-in expression evaluation system is able to understand your class inheritance structure in addition to following C++ rules for method

invocation and polymorphism. (This is discussed in "Using C++" on page 384.)

Calling Functions: Problems and Issues

Unfortunately, calling functions in the expression system can cause problems. Some of these problems are:

- What happens if the function has a side-effect. For example, suppose you have enter `my_var[cntr]` in one row in an Expression List Window, followed by `my_var[++cntr]` in another? If `cntr` equals 3, you'll be seeing the values of `my_var[3]` and `my_var[4]`. However, since `cntr` now equals 4, the first entry is no longer correct.
- What happens when the function crashes (after all you are trying to debug problems), doesn't return, returns the wrong value, or hits a breakpoint?
- What does calling functions do to your debugging interaction if evaluation takes an excessive amount of time?
- What happens if a function creates processes and threads? Or worse, kills them?

In general, there are some protections in the code. For example, if you're displaying items in an **Expression List** Window, TotalView avoids being in an infinite loop by only evaluating items once. This does mean that the information is only accurate at the time at which TotalView made the evaluation.

In most other cases, you're basically on your own. If there's a problem, you'll get an error message. If something takes too long, you can press the Halt button. But if a function alters memory values or starts or stops processes or threads and you can't live with it, you'll need to restart your program. However, if an error occurs while using the **Evaluate** Window, pressing the **Stop** button pops the stack, leaving your program in the state it was in before you used the **Evaluate** command. However, changes made to heap variables will, of course, not be undone.

Expressions in Eval Points and the Evaluate Window

Expression evaluation is not limited to a Variable Window or an Expression List Window. You can use expressions within eval points and in the **Tools > Evaluate** Window. The expressions you type here also let you use programming language constructs. For example, here's a trivial example of code that can execute within the **Evaluate** Window:

```
int i, j, k;
j = k = 10;
for (i=0; i< 20; i++)
{
    j = j + access_func(i, k);
}
j;
```

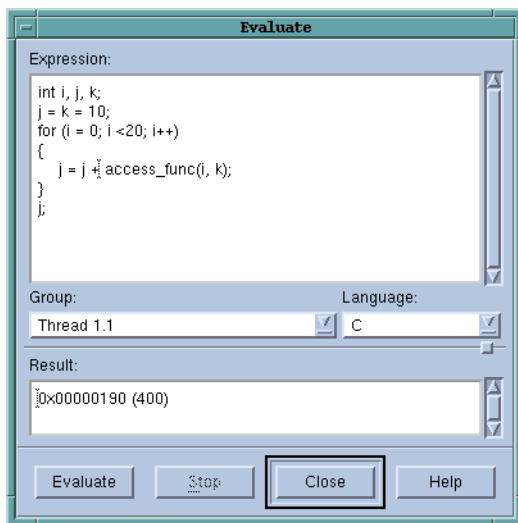
This code fragment declares a couple of variables, runs them through a **for** loop, then displays the value of `j`. In all cases, the programming language constructs being interpreted or compiled within TotalView are based on

Why is There an Expression System?

code within TotalView. TotalView is not using the compiler you used to create your program or any other compiler or interpreter on your system.

Notice the last statement in Figure 239. TotalView displays the value returned by the last statement. This value is displayed. (See “*Displaying the Value of the Last Statement*” on page 384.)

Figure 239: Displaying the Value of the Last Statement



TotalView assumes that there is always a return value, even if it's evaluating a loop or the results of a subroutine returning a void. The results are, of course, not well-defined. If the value returned is not well-defined, TotalView displays a zero in the **Result** area.

The code within eval points and the **Evaluate** Window does not run in the same address space as that in which your program runs. Because TotalView is a debugger, it knows how to reach into your program's address space. The reverse isn't true: your program can't reach into the TotalView address space. This forces some limitations upon what you can do. In particular, you can not enter anything that directly or indirectly needs to pass an address of variable defined within the TotalView expression into your program. Similarly, invoking a function that expects a pointer to a value and whose value is created within TotalView can't work. However, you can invoke a function whose parameter is an address and you name something within that program's address space. For example, you could say something like `adder(an_array)` if `an_array` is contained within your program.

Using C++

The TotalView expression system is able to interpret the way you define your classes and their inheritance hierarchy. For example, if you declare a method in a base class and you invoke upon an object instantiated from a derived class, TotalView knows how to access the function. It also under-

stands when a function is virtual. For example, assume that you have the following declarations:

```
class Circle : public Shape {
public:
...
    virtual double area();
    virtual double area(int);
    double area(int, int);
```

Figure 240 shows an expression list calling an overloaded function. It also shows a setter (mutator) that changes the size of the circle object. A final call to area shows the new value.

Figure 240: Expression List Window: Showing Overloads

The screenshot shows the TotalView Expression List window titled "usageLinux - 2,1". The menu bar includes File, Edit, View, Window, and Help. The window displays a table with two columns: Expression and Value. The expressions listed are: circle, (class Circle); circle.area(), 804.2477184; circle.area(start), 806.2477184; circle.area(start, step+3), 812.2477184; circle.resize(16.0), 0x00000000 (0); and circle.area(), 804.2477184. The last row is highlighted with a blue selection bar.

If your object is instantiated from a class that is part of an inheritance hierarchy, TotalView shows you the hierarchy when you dive on the object. See Figure 241 on page 386.

Using Programming Language Elements

Using C and C++

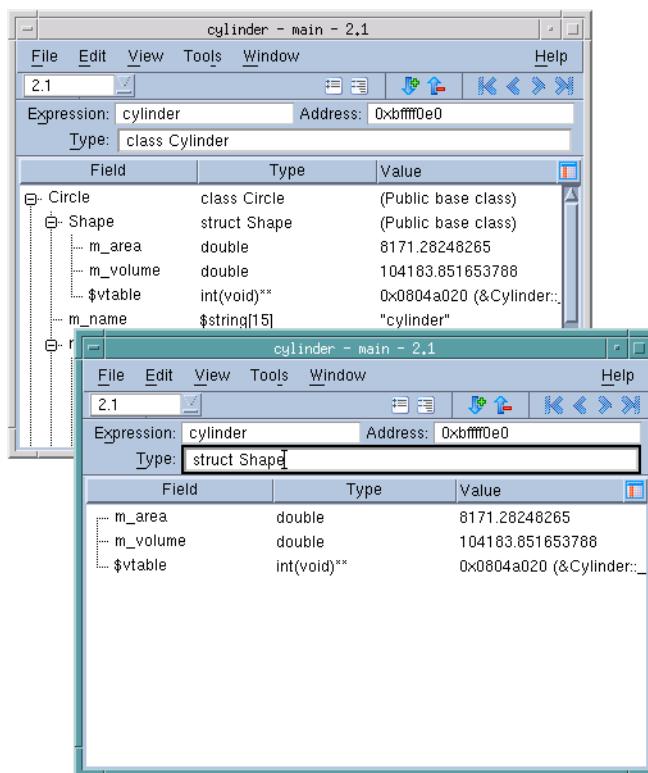
This section contains guidelines for using C and C++ in expressions.

- You can use C-style (`/* comment */`) and C++-style (`// comment`) comments; for example:

```
// This code fragment creates a temporary patch
i = i + 2; /* Add two to i */
```

- You can omit semicolons if the result isn't ambiguous.
- You can use dollar signs (\$) in identifiers. However, we recommend that you do not use dollar signs in names created within the expression system.

Figure 241: Class Casting



If your program does not use a templated function within a library, your compiler may not include a reference to the function in the symbol table. That is, TotalView does not create template instances. In some cases, you might be able to overcome this limitation by preloading the library. However, this only works with some compilers. Most compilers only generate STL operators if your program uses them.

You can use the following C and C++ data types and declarations:

- You can use all standard data types such as **char**, **short**, **int**, **float**, and **double**, modifiers to these data types such as **long int** and **unsigned int**, and pointers to any primitive type or any named type in the target program.
- You can only use simple declarations. Do not define **struct**, **class**, **enum** or **union** types or variables.

You can define a pointer to any of these data types. If an **enum** is already defined in your program, you can use that type when defining a variable.

- The **extern** and **static** declarations are not supported.

You can use the following the C and C++ language statements.

- You can use the **goto** statement to define and branch to symbolic labels. These labels are local to the window. You can also refer to a line number in the program. This line number is the number displayed in the Source Pane. For example, the following **goto** statement branches to source line number 432 of the target program:

```
goto 432;
```

- Although you can use function calls, you can't pass structures.
- You can use type casting.

- You can use assignment, **break**, **continue**, **if/else** structures, **for**, **goto**, and **while** statements. Creating a **goto** that branches to another TotalView evaluation is undefined.

Using Fortran

When writing code fragments in Fortran, you need to follow these guidelines:

- In general, you can use free-form syntax. You can enter more than one statement on a line if you separate the statements with semi-colons (;). However, you cannot continue a statement onto more than one line.
- You can use **GOTO**, **GO TO**, **ENDIF**, and **END IF** statements; Although **ELSEIF** statements aren't allowed, you can use **ELSE IF** statements.
- Syntax is free-form. No column rules apply.
- The space character is significant and is sometimes required. (Some Fortran 77 compilers ignore all space characters.) For example:

Valid	Invalid
DO 100 I=1, 10	D0100I=1, 10
CALL RINGBELL	CALL RING BELL
X .EQ. 1	X.EQ.1

You can use the following data types and declarations in a Fortran expression:

- You can use the **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX** data types.
- You can't define or declare variables that have implied or derived data types.
- You can only use simple declarations. You can't use a **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, or array declaration.
- You can refer to variables of any type in the target program.
- TotalView assumes that **integer (kind=n)** is an n-byte integer.

Fortran Statements

You can use the Fortran language statements:

- You can use assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not alternate return) statements.
- If you enter a comment in an expression, precede the comment with an exclamation point (!).
- You can use array sections within expressions. For more information, see "Array Slices and Array Sections" on page 336.
- A **GOTO** statement can refer to a line number in your program. This line number is the number that appears in the Source Pane. For example, the following **GOTO** statement branches to source line number 432:

GOTO \$432;

You must use a dollar sign (\$) before the line number so that TotalView knows that you're referring to a source line number rather than a statement label.

You cannot branch to a label within your program. You can instead branch to a TotalView line number.

- The following expression operators are not supported: CHARACTER operators and the .EQV., .NEQV., and .XOR. logical operators.
- You can't use subroutine function and entry definitions.
- You can't use Fortran 90 pointer assignment (the => operator).
- You can't call Fortran 90 functions that require assumed shape array arguments.

Fortran Intrinsic Functions

TotalView supports some Fortran intrinsics. You can use these supported intrinsics as elements in expressions. The classification of these intrinsics into groups is that contained within Chapter 13 of the *Fortran 95 Handbook*, by Jeanne C. Adams, *et al.*, published by the MIT Press.

TotalView does not support the evaluation of expressions involving complex variables (other than as the arguments for **real** or **aimag**). In addition, we do not support function versions. For example, you cannot use **dcos** (the double-precision version of **cos**).

The supported intrinsics are:

- Bit Computation functions: **btest**, **iand**, **ibclr**, **ibset**, **ieor**, **ior**, and **not**.
- Conversion, Null and Transfer functions: **achar**, **aimag**, **char**, **dble**, **iachar**, **ichar**, **int**, and **real**.
- Inquiry and Numeric Manipulation Functions: **bit_size**.
- Numeric Computation functions: **acos**, **asin**, **atan**, **atan2**, **ceiling**, **cos**, **cosh**, **exp**, **floor**, **log**, **log10**, **pow**, **sin**, **sinh**, **sqrt**, **tan**, and **tanh**.

Complex arguments to these functions are not supported. In addition, on MacIntosh and AIX, the **log10**, **ceiling**, and **floor** intrinsics are not supported.

The following are not supported:

- Array functions
- Character computation functions.
- Intrinsic subroutines



If you statically link your program, you can only use intrinsics that are linked into your code. In addition, if your operating system is Mac OS X, AIX, or Linux/Power, you can only use math intrinsics in expressions if you directly linked them into your program. The ****** operator uses the **pow** function. Consequently, it too must either be used within your program or directly linked. In addition, **ceiling** and **log10** are not supported on these three platforms.

Using the Evaluate Window

TotalView lets you open a window to evaluate expressions in the context of a particular process and evaluate them in C, Fortran, or assembler.



Not all platforms let you use assembler constructs. See "Architectures" in the TotalView Reference Guide for details.

You can use the **Tools > Evaluate** Dialog Box in many different ways. The following are two examples:

- Expressions can contain loops, so you can use a **for** loop to search an array of structures for an element set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression field.
- Because you can call subroutines, you can test and debug a single routine in your program without building a test program to call it.



Although the CLI does not have an evaluate command, the information in the following sections does apply to the expression argument of the dbreak, dbarrier, dprint, and dwatch commands.

To evaluate an expression: Display the **Evaluate** Dialog Box by selecting the **Tools > Evaluate** command.

An **Evaluate** Dialog Box appears. If your program hasn't yet been created, you won't be able to use any of the program's variables or call any of its functions.

- 1 Select a button for the programming language you're writing the expression in (if it isn't already selected).
- 2 Move to the **Expression** field and enter a code fragment. For a description of the supported language constructs, see "Using Built-in Variables and Statements" on page 394.

The following figure shows a sample expression. The last statement in this example assigns the value of **my_var1-3** back to **my_var1**. Because this is the last statement in the code fragment, the value placed in the **Result** field is the same as if you had just typed **my_var1-3**. (See Figure 242 on page 390.)

- 3 Click the **Evaluate** button.

If TotalView finds an error, it places the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the **Result** field.

While the code is being executed, you can't modify anything in the dialog box. TotalView might also display a message box that tells you that it is waiting for the command to complete. (See Figure 243 on page 390.)

Using the Evaluate Window

Figure 242: Tools > Evaluate Dialog Box

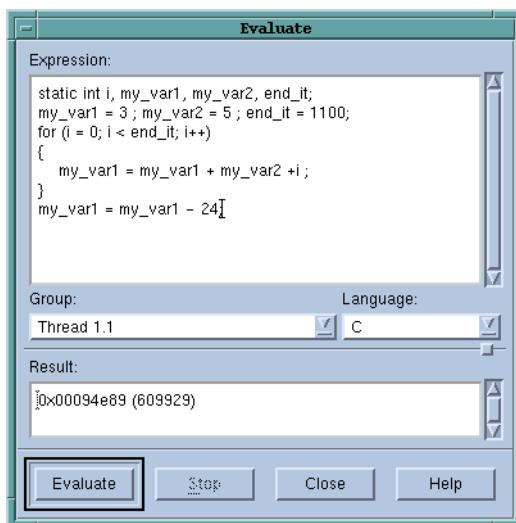
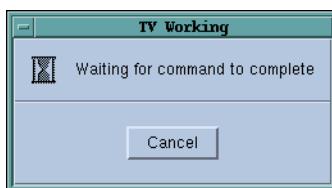


Figure 243: Waiting to Complete Message Box



If you click **Cancel**, TotalView stops execution.

Since TotalView evaluates code fragments in the context of the target process, it evaluates stack variables according to the current program counter. If you declare a variable, its scope is the block that contains the program counter unless, for example, you declare it in some other scope or declare it to be a static variable.

If the fragment reaches a breakpoint (or stops for any other reason), TotalView stops evaluating your expression. Assignment statements in an expression can affect the target process because they can change a variable's value.

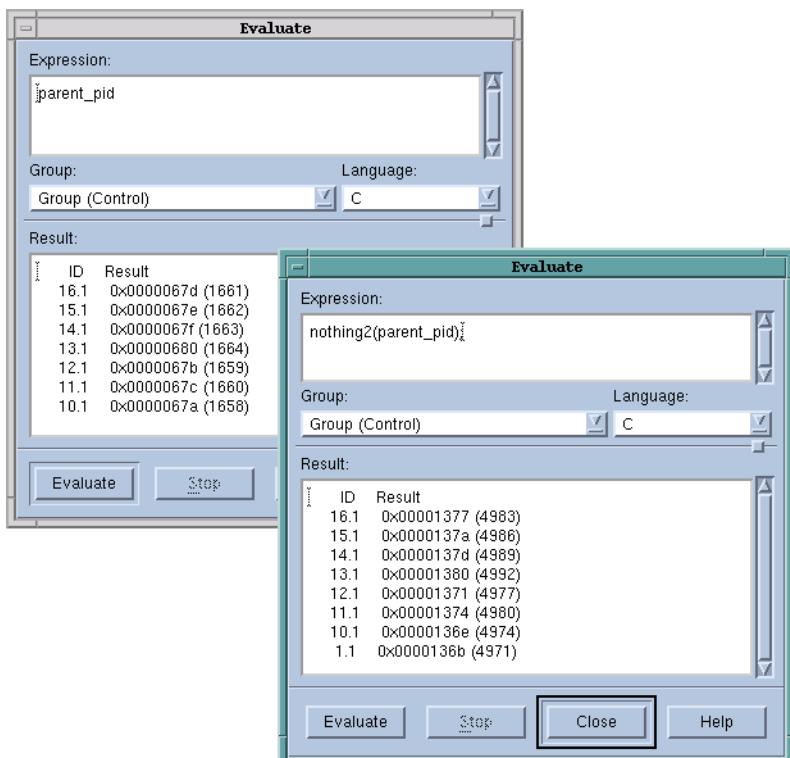
The controls at the top of the dialog box let you refine the scope at which TotalView evaluates the information you enter. For example, you can evaluate a function in more than one process. The following figure shows TotalView displaying the value of a variable in multiple processes, and then sending the value as it exists in each process to a function that runs on each of these processes. (See Figure 244 on page 391.)

See Chapter 13, "Using Groups, Processes, and Threads," on page 251 for information on using the P/T set controls at the top of this window.

Writing Assembler Code

On HP Alpha Tru64 UNIX, RS/6000 IBM AIX, and SGI IRIX operating systems, TotalView lets you use assembler code in eval points, conditional

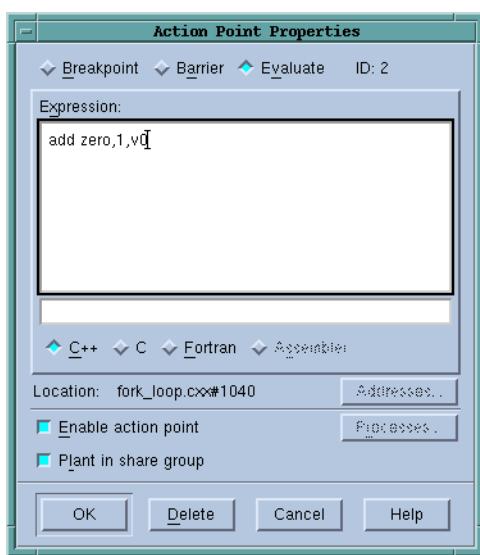
Figure 244: Evaluating Information in Multiple Processes



breakpoints, and in the **Tools > Evaluate** Dialog Box. However, if you want to use assembler constructs, you must enable compiled expressions. See “*About Interpreted and Compiled Expressions*” on page 370 for instructions.

To indicate that an expression in the breakpoint or **Evaluate** Dialog Box is an assembler expression, click the **Assembler** button in the **Action Point > Properties** Dialog Box. (See Figure 245.)

Figure 245: Using Assembler Expressions



You write assembler expressions in the target machine's native assembler language and in a TotalView assembler language. However, the operators available to construct expressions in instruction operands, and the set of available pseudo-operators, are the same on all machines, and are described below.

The TotalView assembler accepts instructions using the same mnemonics recognized by the native assembler, and it recognizes the same names for registers that native assemblers recognize.

Some architectures provide extended mnemonics that do not correspond exactly with machine instructions and which represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView assembler recognizes mnemonics if:

- They assemble to exactly one instruction.
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence.

Assembler language labels are indicated as *name*: and appear at the beginning of a line. You can place a label alone on a line. The symbols you can use include labels defined in the assembler expression and all program symbols.

The TotalView assembler operators are described in the following table:

Operators	Description
+	Plus
-	Minus (also unary)
*	Multiplication
#	Remainder
/	Division
&	Bitwise AND
^	Bitwise XOR
!	Bitwise OR NOT (also unary minus, bitwise NOT)
	Bitwise OR
(expr)	Grouping
<<	Left shift
>>	Right shift
"text"	Text string, 1-4 characters long, is right-justified in a 32-bit word
hi16 (expr)	Low 16 bits of operand <i>expr</i>
hi32 (expr)	High 32 bits of operand <i>expr</i>
lo16 (expr)	High 16 bits of operand <i>expr</i>
lo32 (expr)	Low 32 bits of operand <i>expr</i>

The TotalView assembler pseudo-operations are as follows:

Pseudo Ops	Description
<code>\$debug [0 1]</code>	<i>Internal debugging option.</i> With no operand, toggle debugging; 0 => turn debugging off 1 => turn debugging on
<code>\$hold</code> <code>\$holdprocess</code> <code>\$holdstopall</code> <code>\$holdprocessstopall</code> <code>\$holdthread</code> <code>\$holdthreadstop</code> <code>\$holdthreadstopprocess</code> <code>\$holdthreadstopall</code> <code>\$long_branch expr</code>	Hold the process Hold the process and stop the control group Hold the thread Hold the thread and stop the process Hold the thread and stop the control group Branch to location <i>expr</i> using a single instruction in an architecture-independent way; using registers is not required
<code>\$stop</code> <code>\$stopprocess</code> <code>\$stopall</code> <code>\$stopthread</code> <code>name=expr</code> <code>align expr [, expr]</code>	Stop the process Stop the control group Stop the thread Same as <code>def name,expr</code> Align location counter to an operand 1 alignment; use operand 2 (or 0) as the fill value for skipped bytes
<code>ascii string</code> <code>asciz string</code> <code>bss name, size-expr[,expr]</code>	Same as <i>string</i> Zero-terminated string Define <i>name</i> to represent <i>size-expr</i> bytes of storage in the bss section with alignment optional <i>expr</i> ; the default alignment depends on the size: if <i>size-expr</i> >= 8 then 8 else if <i>size-expr</i> >= 4 then 4 else if <i>size-expr</i> >= 2 then 2 else 1
<code>byte expr [, expr] ...</code> <code>comm name,expr</code>	Place <i>expr</i> values into a series of bytes Define <i>name</i> to represent <i>expr</i> bytes of storage in the bss section; <i>name</i> is declared global; alignment is as in bss without an alignment argument
<code>data</code> <code>def name,expr</code> <code>double expr [, expr] ...</code> <code>equiv name,name</code> <code>fill expr, expr, expr</code>	Assemble code into data section (data) Define a symbol with <i>expr</i> as its value Place <i>expr</i> values into a series of doubles Make operand 1 an abbreviation for operand 2 Fill storage with operand 1 objects of size operand 2, filled with value operand 3
<code>float expr [, expr] ...</code>	Place <i>expr</i> values into a series of floating point numbers
<code>global name</code> <code>half expr [, expr] ...</code> <code>lcomm name,expr[,expr]</code> <code>lsym name,expr</code>	Declare <i>name</i> as global Place <i>expr</i> values into a series of 16-bit words Identical to <code>bss</code> Same as <code>def name,expr</code> but allows redefinition of a previously defined name
<code>org expr [, expr]</code>	Set location counter to operand 1 and set operand 2 (or 0) to fill skipped bytes

Pseudo Ops	Description
quad <i>expr [, expr] ...</i>	Place <i>expr</i> values into a series of 64-bit words
string <i>string</i>	Place <i>string</i> into storage
text	Assemble code into text section (code)
word <i>expr [, expr] ...</i>	Place <i>expr</i> values into a series of 32-bit words
zero <i>expr</i>	Fill <i>expr</i> bytes with zeros

Using Built-in Variables and Statements

TotalView contains a number of built-in variables and statements that can simplify your debugging activities. You can use these variables and statements in eval points and in the **Tools > Evaluate** Dialog Box.

Topics in this section are:

- “Using TotalView Variables” on page 394
- “Using Built-In Statements” on page 395

Using TotalView Variables

TotalView variables that let you access special thread and process values. All variables are 32-bit integers, which is an **int** or a **long** on most platforms. The following table describes built-in variables:

Name	Returns
\$clid	The cluster ID. (Interpreted expressions only.)
\$duid	The TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.)
\$newval	The value just assigned to a watched memory location. (Watchpoints only.)
\$nid	The node ID. (Interpreted expressions only.)
\$oldval	The value that existed in a watched memory location before a new value modified it. (Watchpoints only.)
\$pid	The process ID.
\$processduid	The DUID (debugger ID) of the process. (Interpreted expressions only.)
\$systid	The thread ID assigned by the operating system. When this is referenced from a process, TotalView throws an error.
\$tid	The thread ID assigned by TotalView. When this is referenced from a process, TotalView throws an error.

The built-in variables let you create thread-specific breakpoints from the expression system. For example, the **\$tid** variable and the **\$stop** built-in function let you create a thread-specific breakpoint, as the following code shows:

```
if ($tid == 3)
    $stop;
```

This tells TotalView to stop the process only when the third thread evaluates the expression.

You can also create complex expressions using these variables; for example:

```
if ($pid != 34 && $tid > 7)
    printf ("Hello from %d.%d\n", $pid, $tid);
```

Using any of the following variables means that the eval point is interpreted instead of compiled: **\$clid**, **\$duid**, **\$nid**, **\$processduid**, **\$systid**, **\$tid**, and **\$visualize**. In addition, **\$pid** forces interpretation on AIX.

You can't assign a value to a built-in variable or obtain its address.

Using Built-In Statements

TotalView statements help you control your interactions in certain circumstances. These statements are available in all languages, and are described in the following table. The most commonly used statements are **\$count**, **\$stop**, and **\$visualize**.

Statement	Use
\$count <i>expression</i>	Sets a process-level countdown breakpoint.
\$countprocess <i>expression</i>	When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the process stops. The other processes in the control group continue to execute.
\$countall <i>expression</i>	Sets a program-group-level countdown breakpoint. All processes in the control group stop when any process in the group executes this statement for the number of times specified by <i>expression</i> .
\$countthread <i>expression</i>	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by <i>expression</i> , the thread stops. Other threads in the process continue to execute.
	If the target system cannot stop an individual thread, this statement performs the same as \$countprocess . A thread evaluates <i>expression</i> when it executes \$count for the first time. This expression must evaluate to a positive integer. When TotalView first encounters this variable, it determines a value for <i>expression</i> . TotalView does not reevaluate until the expression actually stops the thread. This means that TotalView ignores changes in the value of <i>expression</i> until it hits the breakpoint. After the breakpoint occurs, TotalView reevaluates the expression and sets a new value for this statement.
	The internal counter is stored in the process and shared by all threads in that process.

Statement	Use
<code>\$hold</code> <code>\$holdprocess</code>	Holds the current process. If all other processes in the group are already held at this eval point, TotalView releases all of them. If other processes in the group are running, they continue to run.
<code>\$holdstopall</code> <code>\$holdprocessstopall</code>	Like <code>\$hold</code> , except that any processes in the group which are running are <i>stopped</i> . The other processes in the group are not automatically held by this call—they are just stopped.
<code>\$holdthread</code>	Freezes the current thread, leaving other threads running.
<code>\$holdthreadstop</code> <code>\$holdthreadstopprocess</code> <code>\$holdthreadstopall</code>	Like <code>\$holdthread</code> , except that it <i>stops</i> the process. The other processes in the group are left running. Like <code>\$holdthreadstop</code> , except that it stops the entire group.
<code>\$stop</code> <code>\$stopprocess</code>	Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.
<code>\$stopall</code>	Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.
<code>\$stopthread</code>	Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as to <code>\$stopprocess</code> .
<code>\$visualize(expression[, slice])</code>	Visualizes the data specified by <i>expression</i> and modified by the optional <i>slice</i> value. <i>Expression</i> and <i>slice</i> must be expressed using the code fragment's language. The <i>expression</i> must return a dataset (after modification by <i>slice</i>) that can be visualized. <i>slice</i> is a quoted string that contains a slice expression. For more information on using <code>\$visualize</code> in an expression, see "Using the Visualizer" on page 184.

Glossary



ACTION POINT: A debugger feature that lets a user request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, eval points, and barriers.

ACTION POINT IDENTIFIER: A unique integer ID associated with an action point.

ACTIVATION RECORD: See stack frame.

ADDRESS SPACE: A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

ADDRESSING EXPRESSION: A set of instructions that tell TotalView where to find information. These expressions are only used within the *type transformation facility* on page 410.

AFFECTED P/T SET: The set of process and threads that are affected by the command. For most commands, this is identical to the target P/T set, but in some cases it might include additional threads. (See “*p/t (process/thread) set*” on page 405 for more information.)

AGGREGATE DATA: A collection of data elements. For example, a structure or an array is an aggregate.

AGGREGATED OUTPUT: The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

API: Application Program Interface. The formal interface by which programs communicate with libraries.

ARENA: A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

ARRAY SECTION: In Fortran, a portion of an array that is also an array. The elements of this array is a new unnamed array object with its own indices. Compare this with a TotalView *array slice* on page 398.

ARRAY SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 403, *upper bound* on page 410, and *stride* on page 408. Displaying a slice of an array can be useful when you are working with very large arrays. Compare this with a TotalView *array section* on page 397.

ASYNCHRONOUS: When processes communicate with one another, they send messages. If a process decides that it doesn't want to wait for an answer, it is said to run "asynchronously." For example, in most client/server programs, one program sends an RPC request to a second program and then waits to receive a response from the second program. This is the normal *synchronous* mode of operation. If, however, the first program sends a message and then continues executing, not waiting for a reply, the first mode of operation is said to be *asynchronous*.

ATTACH: The ability for TotalView to gain control of an already running process on the same machine or a remote machine.

AUTOLAUNCHING: When a process begins executing on a remote computer, TotalView can also launch a **tvdsrv** (TotalView Debugger Server) process on the computer that will send debugging information back to the TotalView process that you are interacting with.

AUTOMATIC PROCESS ACQUISITION: TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you don't have to attach to them manually. If the process is on a remote computer, automatic process acquisition automatically starts the TotalView Debugger Server (**tvdsrv**).

BARRIER POINT: An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

BASE WINDOW: The original Process Window or Variable Window before you dive into routines or variables. After diving, you can use a **Reset** or **Undive** command to restore this original window.

BLOCKED: A thread state in which the thread is no longer executing because it is waiting for an event to occur. In most cases, the thread is blocked because it is waiting for a mutex or condition state.

BREAKPOINT: A point in a program where execution can be suspended to permit examination and manipulation of data.

BUG: A programming error. Finding them is why you're using TotalView.

BULK LAUNCH: A TotalView procedure that launches multiple tvdsrv processes simultaneously.

CALL FRAME: The memory area that contains the variables belonging to a function, subroutine, or other scope division, such as a block.

CALL STACK: A higher-level view of stack memory, interpreted in terms of source program variables and locations. This is where your program places stack frames.

CALLBACK: A function reference stored as a pointer. By using the function reference, this function can be invoked. For example, a program can hand

off the function reference to an event processor. When the event occurs, the function can be called.

CHILD PROCESS: A process created by another process (see "parent process" on page 405) when that other process calls the **fork()** function.

CLOSED LOOP: See *closed loop* on page 399.

CLUSTER DEBUGGING: The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are of the same type and have the same operating system version.

COMMAND HISTORY LIST: A debugger-maintained list that stores copies of the most recent commands issued by the user.

CONDITION SYNCHRONIZATION: A process that delays thread execution until a condition is satisfied.

CONDITIONAL BREAKPOINT: A breakpoint containing an expression. If the expression evaluates to true, program stops. TotalView does not have conditional breakpoints. Instead, you must explicitly tell TotalView to end execution by using the \$stop directive.

CONTEXT SWITCHING: In a multitasking operating system, the ability of the CPU to move from one task to another. As a switch is made, the operating system must save and restore task states.

CONTEXTUALLY QUALIFIED (SYMBOL): A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

CONTROL GROUP: All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by the **fork()** function are in the same control group.

CORE FILE: A file that contains the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

CORE-FILE DEBUGGING: A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

CPU: Central Processing Unit. The component within the computer that most people think of as "the computer". This is where computation and activities related to computing occur.

CROSS-DEBUGGING: A special case of remote debugging where the host platform and the target platform are different types of machines.

CURRENT FRAME: The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

CURRENT LANGUAGE: The source code language used by the file that contains the current source location.

CURRENT LIST LOCATION: The location governing what source code appears in response to a list command.

DATASET: A set of array elements generated by TotalView and sent to the Visualizer. (See *visualizer process* on page 410.)

DBELOG LIBRARY: A library of routines for creating event points and generating event logs from TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

DBFORK LIBRARY: A library of special versions of the **fork()** and **execve()** calls used by TotalView to debug multi-process programs. If you link your program with the TotalView **dbfork** library, TotalView can automatically attach to newly spawned processes.

DEADLOCK: A condition where two or more processes are simultaneously waiting for a resource such that none of the waiting processes can execute.

DEBUGGING INFORMATION: Information relating an executable to the source code from which it was generated.

DEBUGGER PROMPT: A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER: See *tvdsrv process* on page 410.

DEBUGGER STATE: Information that TotalView or the CLI maintains to interpret and respond to user commands. This includes debugger modes, user-defined commands, and debugger variables.

DEPRECATED: A feature that is still available but might be eliminated in a future release.

DISASSEMBLED CODE: A symbolic translation of binary code into assembler language.

DISTRIBUTED DEBUGGING: The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PAR-MACS), run on more than one host.

DIVING: The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

DLL: Dynamic Link Library. A shared library whose functions can be dynamically added to a process when a function with the library is needed. In contrast, a statically linked library is brought into the program when it is created.

DOPE VECTOR: This is a run time descriptor that contains all information about an object that requires more information than is available as a single pointer or value. For example, you might declare a Fortran 90 pointer variable that is a pointer to some other object, but which has its own upper bound, as follows:

```
integer, pointer, dimension () :: iptr
```

Suppose that you initialize it as follows:

```
iptr => iarray (20:1:-2)
```

`iptr` is a synonym for every other element in the first twenty elements of `iarray`, and this pointer array is in reverse order. For example, `iptr(1)` maps to `iarray(20)`, `iptr(2)` maps to `iarray(18)`, and so on.

A compiler represents an `iptr` object using a run time descriptor that contains (at least) elements such as a pointer to the first element of the actual data, a stride value, and a count of the number of elements (or equivalently, an upper bound).

DPID: Debugger ID. This is the ID TotalView uses for processes.

DYNAMIC LIBRARY: A library that uses dynamic loading to load information in an external file at runtime. Dynamic loading implies dynamic linking, which is a process that does not copy a program and its data into the executable at compile time. For more information, see http://en.wikipedia.org/wiki/Dynamic_linking.

EDITING CURSOR: A black line that appears when you select a TotalView GUI field for editing. You use field editor commands to move the editing cursor.

EVAL POINT: A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG: A file that contains a record of events for each process in a program.

EVENT POINT: A point in the program where TotalView writes an event to the event log for later analysis with TimeScan.

EXCEPTION: A condition generated at runtime that indicates that a non-standard event has occurred. The program usually creates a method to handle the event. If the event is not handled, either the program's result will be inaccurate or the program will stop executing.

EXECUTABLE: A compiled and linked version of source files

EXPRESSION SYSTEM: A part of TotalView that evaluates C, C++, and Fortran expressions. An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of a source language. Not all Fortran 90, C, and C++ operators are supported.

EXTENT: The number of elements in the dimension of an array. For example, a Fortran array of `integer(7,8)` has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

FIELD EDITOR: A basic text editor that is part of TotalView. The field editor supports a subset of GNU Emacs commands.

FOCUS: The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you're using the default prompt).

FRAME: An area in stack memory that contains the information corresponding to a single invocation of a subprocedure. See *stack frame* on page 408.

FRAME POINTER: See *stack pointer* on page 408.

FULLY QUALIFIED (SYMBOL): A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

GARBAGE COLLECTION: Examining memory to determine if it is still be referenced. If it is not, it sent back to the program's memory manager so that it can be reused.

GID: The TotalView group ID.

GLOBAL ARRAYS: (from a definition on the Global Arrays web site) The Global Arrays (GA) toolkit provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without need for explicit cooperation by other processes. For more information, see <http://www.emsl.pnl.gov/docs/global/>.

GRID: A collection of distributed computing resources available over a local or wide area network that appears as if it were one large virtual computing system.

GOI: The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

GROUP: When TotalView starts processes, it places related processes in families. These families are called "groups."

GROUP OF INTEREST: The primary group that is affected by a command. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

HEAP: An area of memory that your program uses when it dynamically allocates blocks of memory. It is also how people describe my car.

HOST COMPUTER: The computer on which TotalView is running.

IMAGE: All of the programs, libraries, and other components that make up your executable.

INFINITE LOOP: See *loop, infinite* on page 403.

INSTRUCTION POINTER: See program counter.

INITIAL PROCESS: The process created as part of a load operation, or that already existed in the runtime environment and was attached by TotalView or the CLI.

INITIALIZATION FILE: An optional file that establishes initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called `.tvdr`.

INTERPRETER: A program that reads programming language statements and translates the statements into machine code, then executes this code.

LAMINATE: A process that combines variables contained in separate processes or threads into a unified array for display purposes.

LHS EXPRESSION: This is a synonym for **lvalue**.

LINKER. A program that takes all the object files creates by the compiler and combines them and libraries required by the program into the executable program.

LOCKSTEP GROUP: All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group. By definition, all members of a lockstep group are in the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

LOOP, INFINITE: see *infinite loop* on page 402.

LOWER BOUND: The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

LVALUE: A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

MACHINE STATE: Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

MANAGER THREAD: A thread created by the operating system. In most cases, you do not want to manage or examine manager threads.

MESSAGE QUEUE: A list of messages sent and received by message-passing programs.

MIMD: An acronym for Multiple Instruction, Multiple Data, which describes a type of parallel computing.

MISD: An acronym for Multiple Instruction, Single Data, which describes a type of parallel computing.

MPI: An acronym for "Message Passing Interface."

MPICH: MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see <http://www.mcs.anl.gov/mpi>.

MPMD PROGRAMS: An acronym for Multiple Program, Multiple Data, which describes programs that involve multiple executables, executed by multiple threads and processes.

MULTITASK: In the context of high performance computing, this is the ability to divide a program into smaller pieces or tasks that execute separately.

MULTI-PROCESS: The ability of a program to spawn off separate programs, each having its own context and memory. multi-process programs can (and most often do) run processes on more than one computer. They can also run multiple processes on one computer. In this case, memory can be shared

MULTI-THREADED: The ability of a program to spawn off separate tasks that use the same memory. Switching from task to task is controlled by the operating system.

MUTEX (MUTUAL EXCLUSION): Techniques for sharing resources so that different users do not conflict and cause unwanted interactions.

NATIVE DEBUGGING: The action of debugging a program that is running on the same machine as TotalView.

NESTED DIVE: TotalView lets you dive into pointers, structures, or arrays in a variable. When you dive into one of these elements, TotalView updates the display so that the new element appears. A nested dive is a *dive* within a *dive*. You can return to the previous display by selecting the left arrow in the top-right corner of the window.

NODE: A machine on a network. Each machine has a unique network name and address.

OFF-BY-ONE: An error usually caused by forgetting that arrays begin with element 0 in C and C++.

OPENMP: (from a definition on the OpenMP web site) OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. The MP in OpenMP stands for Multi Processing. We provide Open specifications for Multi Processing via collaborative work with interested parties from the hardware and software industry, government and academia. For more information, see <http://www.openmp.org/>.

OUT-OF-SCOPE: When symbol lookup is performed for a particular symbol name and it isn't found in the current scope or any that contains scopes, the symbol is said to be out-of-scope.

PAGE PROTECTION: The ability to segregate memory pages so that one process cannot access pages owned by another process. It can also be used to generate an exception when a process tries to access the page.

PARALLEL PROGRAM: A program whose execution involves multiple threads and processes.

PARALLEL TASKS: Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results.

PARALLELIZABLE PROBLEM: A problem that can be divided into parallel tasks. This type of program might require changes in the code and/or the underlying algorithm.

PARCEL: The number of bytes required to hold the shortest instruction for the target architecture.

PARENT PROCESS: A process that calls the `fork()` function to spawn other processes (usually called child processes).

PARMACS LIBRARY: A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

PARTIALLY QUALIFIED (SYMBOL): A symbol name that includes only some of the levels of source code organization (for example, file name and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PATCHING: Inserting code in a breakpoint that is executed immediately preceding the breakpoint's line. The patch can contain a GOTO command to branch around incorrect code.

PC: An abbreviation for *Program Counter*.

PID: Depending on the context, this is either the process ID or the program ID. In most cases, this is the process ID.

POI: The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and so on.

/PROC: An interface that allows debuggers and other programs to control or obtain information from running processes. ptrace also does this, but /proc is more general.

PROCESS: An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP: A group of processes associated with a multi-process program. A process group includes program control groups and share groups.

PROCESS/THREAD IDENTIFIER: A unique integer ID associated with a particular process and thread.

PROCESS OF INTEREST: The primary process that TotalView uses when it is trying to determine what to step, stop, and so on.

PROGRAM CONTROL GROUP: A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent), and children that were forked with a subsequent call to the `execve()` function (processes that don't share the same source code as the parent). Contrast this with *share group* on page 407.

PROGRAM EVENT: A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

PROGRAM STATE: A higher-level view of the machine state, where addresses, instructions, registers, and such are interpreted in terms of source program variables and statements.

P/T (PROCESS/THREAD) SET: The set of threads drawn from all threads in all processes of the target program.

PTHREAD ID: This is the ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

PVM LIBRARY: Parallel Virtual Machine library. A message-passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

QUEUE: A data structure whose data is accessed in the order in which it was entered. This is like a line at a tollbooth where the first in is the first out.

RACE CONDITION: A problem that occurs when threads try to simultaneously access a resource. The result can be a deadlock, data corruption, or a program fault.

REMOTE DEBUGGING: The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

RESUME COMMANDS: Commands that cause execution to restart from a stopped state: `dstep`, `dgo`, `dcont`, `dwait`.

RHS EXPRESSION: This is a synonym for `rvalue`.

RVALUE: An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

SATISFACTION SET: The set of processes and threads that must be held before a barrier can be satisfied.

SATISFIED: A condition that indicates that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier, or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is *satisfied*, the held processes or threads are released, which means they can be run. Prior to this event, they could not run.

SCOPE: The region in your program in which a variable or a function exists or is defined. This region begins with its declaration and extends to the end of the current block.

SEARCH PATH: A list that contains places that software looks to locate files contained within the file system. In TotalView, the search path contains locations containing your program's source code.

SERIAL EXECUTION: Execution of a program sequentially, one statement at a time.

SERIAL LINE DEBUGGING: A form of remote debugging where TotalView and the `tvdsrv` communicate over a serial line.

SERVICE THREAD: A thread whose purpose is to *service* or manage other threads. For example, queue managers and print spoolers are service

threads. There are two kinds of service threads: those created by the operating system or runtime system and those created by your program.

SHARE GROUP: All the processes in a control group that share the same code. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

SHARED LIBRARY: A compiled and linked set of source files that are dynamically loaded by other executables.

SIGNALS: Messages informing processes of asynchronous events, such as serious errors. The action that the process takes in response to the signal depends on the type of signal and whether the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

SIMD: An acronym for Single Instruction, Multiple Data, which describes a type of parallel computing.

SINGLE PROCESS SERVER LAUNCH: A TotalView procedure that individually launches tvdsvr processes.

SINGLE STEP: The action of executing a single statement and stopping (as if at a breakpoint).

SISD: An acronym for Single Instruction, Single Data, which describes a type of parallel computing.

SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 403, *upper bound* on page 410, and *stride* on page 408. Displaying a slice of an array can be useful when you are working with very large arrays. Compare this with a TotalView *array section* on page 397.

SOID: An acronym for symbol object ID. A SOID uniquely identifies all TotalView information. It also represents a handle by which you can access this information.

SOURCE FILE: Program file that contains source language statements. TotalView lets you debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler files.

SOURCE LOCATION: For each thread, the source code line it executes next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

SPAWNED PROCESS: The process created by a user process executing under debugger control.

SPMD PROGRAMS: An acronym for Single Program, Multiple Data, which describe a type of parallel computing that involves just one executable, executed by multiple threads and processes.

STACK: A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

STACK FRAME: Whenever your program calls a function, it creates a set of information that includes the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the program counter (PC) at the time the routine was called. The information for one function is called a "stack frame" as it is placed on your program's stack.

When your program begins executing, it has only one frame: the one allocated for function `main()`. As your program calls functions, new frames are allocated. When a function returns to the function from which it is called, the frame is deallocated.

STACK POINTER: A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored. This is also called a frame pointer.

STACK TRACE: A sequential list of each currently active routine called by a program, and the frame pointer that points to its stack frame.

STATIC (SYMBOL) SCOPE: A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

STEPPING: Advancing program execution by fixed increments, such as by source code statements.

STL: An acronym for Standard Template Library.

STOP SET: A set of threads that TotalView stops after an action point triggers.

STOPPED/HELD STATE: The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) is required before it is capable of continuing execution.

STOPPED/RUNNABLE STATE: The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

STOPPED STATE: The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

STRIDE: The interval between array elements in a slice and the order in which TotalView displays these elements. If the stride is 1, TotalView displays every element between the lower bound and upper bound of the slice. If the stride is 2, TotalView displays every other element. If the stride is -1, TotalView displays every element between the upper bound and lower bound (reverse order).

SYMBOL: Entities within program state, machine state, or debugger state.

SYMBOL LOOKUP: Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively so that contains scopes are searched in an outward progression.

SYMBOL NAME: The name associated with a symbol known to TotalView (for example, function, variable, data type, and so on).

SYMBOL TABLE: A table of symbolic names used in a program (such as variables or functions) and their memory locations. The symbol table is part of the executable object generated by the compiler (with the `-g` option) and is used by debuggers to analyze the program.

SYNCHRONIZATION: A mechanism that prevents problems caused by concurrent threads manipulating shared resources. The two most common mechanisms for synchronizing threads are *mutual exclusion* and *condition synchronization*.

TARGET COMPUTER: The computer on which the process to be debugged is running.

TARGET PROCESS SET: The target set for those occasions when operations can only be applied to entire processes, not to individual threads in a process.

TARGET PROGRAM: The executing program that is the target of debugger operations.

TARGET P/T SET: The set of processes and threads that a CLI command acts on.

TASK: A logically discrete section of computational work. (This is an informal definition.)

THREAD: An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

THREAD EXECUTION STATE: The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of predefined states.

THREAD OF INTEREST: The primary thread affected by a command. This is abbreviated as TOI.

TID: The thread ID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

TLA: An acronym for Three-Letter Acronym. So many things from computer hardware and software vendors are referred to by a three-letter acronym that yet another acronym was created to describe these terms.

TOI: The thread of interest. This is the primary thread affected by a command.

TRIGGER SET: The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

TRIGGERS: The effect during execution when program operations cause an event to occur (such as arriving at a breakpoint).

TTF: See *type transformation facility* on page 410.

TRAP: An instruction that stops program execution and which allows a debugger to gain control over your program.

TVDSVR PROCESS: The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

TYPE TRANSFORMATION FACILITY: This is abbreviated as TTF. A TotalView subsystem that allows you to change the way information appears. For example, an STL vector can appear as an array.

UNDISCOVERED SYMBOL: A symbol that is referred to by another symbol. For example, a `typedef` is a reference to the aliased type.

UNDIVING: The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you click the `undive` icon in the upper-right corner of the window.

UPC: (from a definition on the UPC web site) The Unified Parallel C language, which is an extension to the C programming language that is designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. See <http://upc.nerc.gov/> for more information.

UPPER BOUND: The last element in the dimension of an array or the slice of an array.

USER THREAD: A thread created by your program.

USER INTERRUPT KEY: A keystroke used to interrupt commands, most commonly defined as Ctrl+C.

VARIABLE WINDOW: A TotalView window that displays the name, address, data type, and value of a particular variable.

VISUALIZATION: In TotalView, visualization means graphically displaying an array's values.

VISUALIZER PROCESS: A process that works with TotalView in a separate window, allowing you to see a graphic representation of program array data.

WATCHPOINT: An action point that tells TotalView to stop execution when the value of a memory location changes.

WORKER THREAD: A thread in a workers group. These are threads created by your program that performs the task for which you've written the program.

WORKERS GROUP: All the worker threads in a control group. Worker threads can reside in more than one share group.

Index



Symbols

scope separator character 332
#string data type 312
\$address data type 316
\$char data type 316
\$character data type 316
\$clid built-in variable 394
\$code data type 297, 316, 320
\$complex data type 316
\$complex_16 data type 316
\$complex_8 data type 316
\$count built-in function 6, 368, 371, 395
\$countall built-in function 395
\$countthread built-in function 395
\$debug assembler pseudo op 393
\$denorm filter 339
\$double data type 316
\$double_precision data type 316
\$duid built-in variable 394
\$extended data type 317
\$float data type 317
\$hold assembler pseudo op 393
\$hold built-in function 396
\$holdprocess assembler pseudo op 393
\$holdprocess built-in function 396
\$holdprocessall built-in function 396
\$holdprocessstopall assembler pseudo op 393
\$holdstopall assembler pseudo op 393
\$holdstopall built-in function 396
\$holdthread assembler pseudo op 393
\$holdthread built-in function 396
\$holdthreadstop assembler pseudo op 393
\$holdthreadstop built-in function 396
\$holdthreadstopall assembler pseudo op 393

\$holdthreadstopall built-in function 396
\$holdthreadstopprocess assembler pseudo op 393
\$holdthreadstopprocess built-in function 396
\$inf filter 339
\$int data type 317
\$integer data type 317
\$integer_1 data type 317
\$integer_2 data type 317
\$integer_4 data type 317
\$integer_8 data type 317
\$is_denorm intrinsic function 340
\$is_finite intrinsic function 340
\$is_inf intrinsic function 340
\$is_nan intrinsic function 340
\$is_ndenorm intrinsic function 340
\$is_ninf intrinsic function 340
\$is_nnrm intrinsic function 340
\$is_norm intrinsic function 340
\$is_pdenorm intrinsic function 340
\$is_pinf intrinsic function 340
\$is_pnorm intrinsic function 340
\$is_pzero intrinsic function 340
\$is_qnan intrinsic function 340
\$is_snan intrinsic function 340
\$is_zero intrinsic function 340
\$logical data type 317
\$logical_1 data type 317
\$logical_2 data type 317
\$logical_4 data type 317
\$logical_8 data type 317
\$long data type 317
\$long_branch assembler pseudo op 393
\$long_long data type 317
\$nan filter 339
\$nanq filter 339
\$nans filter 339
\$ndenorm filter 339
\$newval built-in function 378
\$newval built-in variable 394
\$nid built-in variable 394
\$ninf filter 339
\$oldval built-in function 378
\$oldval built-in variable 394
\$oldval watchpoint variable 378
\$pdenorm filter 339
\$pid built-in variable 394
\$pinf filter 339
\$processuid built-in variable 394
\$real data type 317
\$real_16 data type 317
\$real_4 data type 317
\$real_8 data type 317
\$short data type 317
\$stop assembler pseudo op 393
\$stop built-in function 6, 371, 379, 396
\$stopall assembler pseudo op 393
\$stopall built-in function 396
\$stopprocess assembler pseudo op 393
\$stopprocess built-in function 396
\$stopthread assembler pseudo op 393
\$stopthread built-in function 396
\$string data type 312, 317, 318
\$systid built-in variable 394
\$tid built-in variable 394
\$visualize built-in function 192, 396
 in animations 192
 using casts 192
\$void data type 317, 319
\$wchar data type 317, 318
\$wchar_s16 data type 317
\$wchar_s32 data type 318
\$wchar_u16 data type 318
\$wchar_u32 data type 318
\$wstring data type 318
\$wstring_s16 data type 318
\$wstring_s32 data type 318

\$wstring_u16 data type 318
 \$wstring_u32 data type 318
 %B bulk server launch command 90
 %C server launch replacement characters 87
 %D bulk server launch command 88
 %D single process server launch command 87
 %F bulk server launch command 90
 %H bulk server launch command 89
 %H hostname replacement character 90
 %I bulk server launch command 90
 %K bulk server launch command 90
 %L bulk server launch command 89
 %L single process server launch command 88
 %N bulk server launch command 90
 %P bulk server launch command 89
 %P single process server launch command 88
 %R single process server launch command 87
 %t1 bulk server launch command 90
 %t2 bulk server launch command 90
 %V bulk server launch command 89
 & intersection operator 275
 . (dot) current set indicator 259, 276
 . (period), in suffix of process names 235
 .dmg installer 53
 .rhosts file 91, 116
 .totalview subdirectory 56
 .tvrc initialization files 56
 .Xdefaults file 57, 78
 autoLoadBreakpoints 78
 deprecated resources 78
 / slash in group specifier 264
 /usr/lib/array/arrayd.conf file 89
 : (colon), in array type strings 314
 : as array separator 334
 < first thread indicator (CLI) 259
 > (right angle bracket), indicating nested dives 299
 @ action point marker, in CLI 352
 - difference operator 275
 | union operator 275
 ' module separator 326

A

–a command-line option 54, 206
 passing arguments to program 54
 a width specifier 264
 general discussion 267
 absolute addresses, display assembler as 172
 acquiring processes 117
 action points tab 356, 357

Action Point > At Location command 5, 353, 354, 357
 Action Point > Delete All command 356
 Action Point > Properties command 5, 129, 238, 350, 355, 356, 359, 361, 363, 365, 367
 deleting barrier points 365
 Action Point > Properties dialog box 360
 Action Point > Properties dialog box figure 355, 359, 364
 Action Point > Save All command 117, 380
 Action Point > Save As command 380
 Action Point > Set Barrier command 363
 Action Point > Suppress All command 356, 357
 action point files 57
 action point identifiers 211
 never reused in a session 211
 Action Point Symbol figure 350
 action points 211, 360
 see also barrier points
 see also eval points
 common properties 350
 definition 5, 349
 deleting 356
 disabling 356
 enabling 356
 evaluation points 5
 ignoring 356
 list of 171
 multiple addresses 352
 saving 380
 suppressing 356
 unsuppressing 357
 watchpoint 10
 action points list, see action points tab
 Action Points Page 127, 171
 actor mode, Visualizer 181
 adapter_use option 115
 Add new host option 59
 Add to Expression List command 9, 303, 307
 Add to Expression List context menu command 303
 adding command-line arguments 64
 adding environment variables 65
 adding members to a group 262
 adding program arguments 55
 Additional starter arguments area 98
 \$address 316
 address range conflicts 372
 addresses
 changing 321
 editing 321
 specifying in variable window 296
 tracking in variable window 284
 advancing and holding processes 210
 advancing program execution 210
 aggregates, in Expression List window 305
 aliases
 built-in 208
 group 208
 group, limitations 208
 align assembler pseudo op 393
 all width specifier 260
 allocated arrays, displaying 321
 Ambiguous Function Dialog Box 354
 Ambiguous Function dialog box 357
 Ambiguous Function dialog box figure 225, 355
 ambiguous function names 224
 Ambiguous Line dialog box figure 352, 353
 ambiguous names 226
 ambiguous scoping 332
 ambiguous source lines 237
 angle brackets, in windows 299
 animation using \$visualize 192
 areas of memory, data type 319
 arena specifiers 259
 defined 259
 incomplete 272
 inconsistent widths 273
 arenas
 and scope 252
 defined 252, 259
 iterating over 259
 ARGS variable 206
 modifying 206
 ARGS_DEFAULT variable 55, 206
 clearing 206
 arguments
 in server launch command 87, 92
 passing to program 54
 replacing 206
 argv, displaying 321
 Array Data Filter by Range of Values figure 341
 array data filtering
 by comparison 337
 by range of values 341
 for IEEE values 339
 see also arrays, filtering
 Array Data Filtering by Comparison figure 339
 Array Data Filtering for IEEE Values figure 340
 array of structures 298
 displaying 300
 in Expression List window 305
 array pointers 293
 array rank 194
 array services handle (ash) 122
 Array Statistics Window figure 343

array structure
 viewing limitations 287

arrays
 array data filtering 337
 bounds 314
 casting 314
 character 318
 checksum statistic 343
 colon separators 334
 count statistic 343
 deferred shape 327, 334
 denormalized count statistic 343
 display subsection 315
 displaying 193, 333
 displaying allocated 321
 displaying argv 321
 displaying contents 174
 displaying declared 321
 displaying multiple 193
 displaying slices 334
 diving into 298
 editing dimension of 315
 evaluating expressions **382**
 extent 315
 filter conversion rules 338
 filtering 315, 337, 338
 filtering expressions 341
 filtering options 337
 in C 314
 in Fortran 314
 infinity count statistic 344
 limiting display 335
 lower adjacent statistic 344
 lower bound of slices 334
 lower bounds 314
 maximum statistic 344
 mean statistic 344
 median statistic 344
 minimum statistic 344
 NaN statistic 344
 non-default lower bounds 314
 overlapping nonexistent memory
 333
 pointers to 314
 quartiles statistic 344
 skipping elements 335
 slice example 334, 335
 slice, initializing 215
 slice, printing 215
 slice, refining 192
 slices with the variable command
 336
 slicing 8
 sorting 342
 standard deviation statistic 344
 statistics 343
 stride 334
 stride elements 334
 subsections 334
 sum statistic 344

type strings for 314
 upper adjacent statistic 344
 upper bound 314
 upper bound of slices 334
 viewing across elements 346
 visualizing 182, 192
 writing to file 217
 zero count statistic 344

arrow buttons 7
 arrow over line number 171
 ascii assembler pseudo op 393
 asciz assembler pseudo op 393
 ash (array services handle) 122
 ash (array services handle) 122
 ASM icon 351, 359
 assembler
 absolute addresses 172
 and -g compiler option 174
 constructs 390
 displaying 172
 examining 171
 expressions 392
 in code fragment 366
 symbolic addresses 172

Assembler > By Address command
 172
 Assembler > Symbolically command
 172
 Assembler command 171
 assigning output to variable 205
 assigning p/t set to variable 261
 asynchronous processing 16
 at breakpoint state 67
 At Location command 5, 353, 354, 357
 atering standard I/O 65
 Attach Page 118, 168
 Attach Subset command 124, 125
 Attach Subset command, when not
 usable 98
 Attach to an existing process option
 66
 Attach to process page 66
 attached process states 67
 attaching
 commands 61
 restricting 124
 restricting by communicator 125
 selective 124
 to a task 156
 to all 126
 to existing process 59
 to HP MPI job 114
 to job 117
 to MPI tasks 126
 to MPICH application 102
 to MPICH job 102
 to none 126
 to PE 117
 to poe 118
 to processes 61, 117, 124, 155

to PVM task 155
 to RMS processes 121
 to SGI MPI job 122

attaching to processes preference 126
 Auto Visualize command 183
 Auto Visualize, in Dataset Window 185
 auto_array_cast_bounds variable 294
 auto_deref_in_all_c variable 294
 auto_deref_in_all_fortran variable 294
 auto_deref_initial_c variable 294
 auto_deref_initial_fortran variable 294
 auto_deref_nested_c variable 294
 auto_deref_nested_fortran variable
 294

auto_save_breakpoints variable 380
 autolaunch 82, 83
 changing 91
 defined 55
 disabling 55, 83, 84, 91
 launch problems 86
 sequence 92

autolaunching 92
 autoLoadBreakpoints .Xdefault 78
 automatic dereferencing 293
 automatic process acquisition 101,
 115, 154
 averaging data points 189
 averaging surface display 189
 axis, transposing 187

B

B state 67
 backtick separator 326
 backward icon 175
 barrier points 12, 31, 231, 362, 363
 see also process barrier break-
 point
 clearing 356
 defined 211
 defined (again) 362
 deleting 365
 httng 365
 satisfying 364
 states 363
 stopped process 365
 baud rate, for serial line 94
 bit fields 311
 block scoping 331
 Block Status command 295
 blocking send operations 112
 blocks
 displaying 287
 naming 332

BlueGene, see IBM BlueGene **118**
 bluegene_io_interface variable 119
 bluegene_server_launch variable 119
 bold data 7
 Both command 172, 249
 bounds for arrays 314
 boxed line number 171, 252, 352

branching around code 369
 Breakpoint at Assembler Instruction figure 358
 breakpoint files 57
 breakpoint operator 275
 breakpoints
 and MPI_Init() 117
 apply to all threads 350
 automatically copied from master process 101
 behavior when reached 359
 changing for parallelization 127
 clearing 166, 252, 356
 conditional 366, 368, 395
 copy, master to slave 102
 countdown 368, 395
 default stopping action 127
 defined 211, 349
 deleting 356
 disabling 356
 enabling 356
 entering 122
 example setting in multiprocess program 362
 fork() 361
 hitting within eval point 390
 ignoring 356
 in child process 359
 in parent process 359
 in spawned process 154
 listing 171
 machine-level 358
 multiple processes 359
 not shared in separated children 361
 placing 171
 reloading 117
 removed when detaching 63
 removing 166
 saving 380
 set while a process is running 352
 set while running parallel tasks 116
 setting 116, 166, 218, 252, 351, 352, 359
 shared by default in processes 361
 sharing 360, 361
 stop all related processes 360
 suppressing 356
 thread-specific 395
 toggling 353
 while stepping over 240
 bss assembler pseudo op 393
 built-in aliases 208
 built-in functions
 \$count 6, 368, 371, 395
 \$countall 395
 \$countthread 395
 \$hold 396
 \$holdprocess 396
 \$holdprocessall 396

\$holdstopall 396
 \$holdthread 396
 \$holdthreadstop 396
 \$holdthreadstopall 396
 \$holdthreadstopprocess 396
 \$stop 6, 371, 379, 396
 \$stopall 396
 \$stopprocess 396
 \$stopthread 396
 \$visualize 192, 396
 forcing interpretation 370
 built-in variables 394
 \$clid 394
 \$duid 394
 \$newval 394
 \$nid 394
 \$oldval 394
 \$pid 394
 \$processduid 394
 \$string 315
 \$sytid 394
 \$tid 394
 forcing interpretation 395
 Bulk Launch page 86
 bulk server launch 81, 84
 command 85
 connection timeout 85
 on HP Alpha 89
 on IBM RS/6000 90
 on Cray 90
 on SGI MIPS 88
 bulk server launch command
 %B 90
 %D 88
 %F 90
 %H 89
 %I 90
 %K 90
 %L 89
 %N 90
 %P 89
 %t1 90
 %t2 90
 %V 89
 –callback_host 89
 –callback_ports 89
 –set_pws 89
 –verbosity 89
 –working_directory 88
 bulk_incr_timeout variable 85
 bulk_launch_base_timeout variable 85
 bulk_launch_enabled variable 84, 86
 bulk_launch_incr_timeout variable 85
 bulk_launch_string variable 85
 bulk_launch_tmpfile1_header_line variable 85
 bulk_launch_tmpfile1_header_line variable 85
 bulk_launch_tmpfile1_host_line variable 85

bulk_launch_tmpfile1_host_lines variable 85
 bulk_launch_tmpfile1_trailer_line variable 85
 bulk_launch_tmpfile1_trailer_line variable 85
 bulk_launch_tmpfile2_header_line variable 85
 bulk_launch_tmpfile2_header_line variable 85
 bulk_launch_tmpfile2_host_lines variable 85
 bulk_launch_tmpfile2_host_line variable 85
 bulk_launch_tmpfile2_trailer_line variable 85
 bulk_launch_tmpfile2_trailer_line variable 85
 By Address command 172
 byte assembler pseudo op 393

C casting for Global Arrays 149, 150
 C control group specifier 264, 265
 C++
 changing class types 323
 display classes 322
 C++/C++
 in expression system 385
 C/C++
 array bounds 314
 arrays 314
 filter expression 341
 how data types are displayed 312
 in code fragment 366
 type strings supported 312
 C/C++ statements
 expression system 386
 Call Graph command 179
 call graph, updating display 179
 call stack 170
 call_graph group 180
 –callback command-line option 91
 –callback_host bulk server launch command 89
 –callback_option single process server launch command 88
 –callback_ports bulk server launch command 89
 camera mode, Visualizer 181
 capture command 205
 casting 301, 311, 312, 313
 examples 320
 to type \$code 297
 types of variable 311
 casting arrays 314
 casting Global Arrays 149, 150
 Cell broadband engine 138
 CLI focus within SPU 143
 CLI variables named 142

context 138
 description 140
 empty context 138
 loading SPU images 138
 PPE defined 138
 PPU defined 138
 PPU organization 140
 share groups 140
 SPE defined 138
 SPU breakpoints 141
 SPU defined 138
 SPU images, loading 138
 SPU naming in TotalView 141
 SPU registers 143
 SPU threads 140
 SPU threads share group 140
 synergistic processor unit 138
 thread IDs 141
 union describing SPU register contents 143

CGROUP variable 262, 268
`ch_lfshmem` device 100
`ch_mpl` device 100
`ch_p4` device 100, 102, 129
`ch_shmem` device 100, 102
 Change Value command 310
 changing autolaunch options 83
 changing command-line arguments 64
 changing expressions 302
 changing precision 282
 changing process thread set 258
 changing processes 230
 changing program state 201
 changing remote shell 91
 changing size 282
 changing threads 230
 changing threads in Variable Window 298
 changing values 176
 changing variables 310
`$char` data type 316
`$character` data type 316
 character arrays 318
 chasing pointers 293, 298
 checksum array statistic 343
 child process names 235
 classes, displaying 322
 Clear All STOP and EVAL command 356
 clearing
 breakpoints 166, 252, 356, 359
 continuation signal 243
 evaluation points 166

CLI
 components 199
 in startup file 202
 initialization 202
 interface 201
 introduced 14

invoking program from shell example 202
 not a library 199
 output 205
 prompt 203
 relationship to TotalView 200
 starting 53, 201
 starting from command prompt 201
 starting from TotalView GUI 201
 starting program using 203

CLI and Tcl relationship 201

CLI commands
 assigning output to variable 205
 capture 205
 dactions 350
 dactions –load 117, 380
 dactions –save 117, 380
 dassign 310
 dattach 54, 61, 63, 102, 117, 118, 124, 210
 dattach mprun 123
 dbarrier 362, 364
 dbarrier –e 367
 dbarrier –stop_when_hit 129
 dbreak 219, 352, 354, 360
 dbreak –e 367
 dcheckpoint 244
 ddelete 106, 354, 356, 365
 ddetach 63
 ddisable 356, 365
 ddlopen 245
 ddown 241
 default focus 258
 denable 356, 357
 dfocus 239, 257, 258
 dga 150
 dgo 113, 116, 117, 122, 127, 236, 273
 dgroups –add 262, 268
 dhalt 127, 229, 239
 dhold 232, 363
 dhold –thread 232
 dkill 128, 204, 210, 243
 dload 61, 86, 203, 204, 210
 dnext 128, 237, 240
 dnexti 237, 240
 dout 241, 253
 dprint 135, 136, 216, 225, 250, 284, 285, 293, 296, 315, 321, 324, 325, 327, 334, 336
 dptsets 66, 230
 drerun 204, 243
 drestart 244
 drun 203, 206
 dset 206, 208
 dstatus 66, 365
 dstep 128, 237, 240, 253, 259, 261, 273
 dstepi 237, 240

dunhold 232, 363
`dunhold –thread` 232
 sunset 206
`duntil` 241, 253, 255
`dup` 241, 285
`dwhere` 260, 273, 285
`exit` 58
`read_symbols` 248
 run when starting TotalView 56

CLI variables
`ARG$` 206
`ARG$, modifying` 206
`ARG$_DEFAULT` 55, 206
 clearing 206
`auto_array_cast_bounds` 294
`auto_deref_in_all_c` 294
`auto_deref_in_all_fortran` 294
`auto_deref_initial_c` 294
`auto_deref_initial_fortran` 294
`auto_deref_nested_c` 294
`auto_deref_nested_fortran` 294
`auto_save_breakpoints` 380
`bulk_incr_timeout` 85
`bulk_launch_base_timeout` 85
`bulk_launch_enabled` 84, 86
`bulk_launch_incr_timeout` 85
`bulk_launch_string` 85
`bulk_launch_tmpefile1_trailer_line` 85
`bulk_launch_tmpefile2_trailer_line` 85
`bulk_launch_tmppfile1_header_line` 85
`bulk_launch_tmppfile1_header_line` 85
`bulk_launch_tmppfile1_host_lines` 85
`bulk_launch_tmppfile1_host_line` 85
`bulk_launch_tmppfile1_trailer_line` 85
`bulk_launch_tmppfile2_header_line` 85
`bulk_launch_tmppfile2_header_line` 85
`bulk_launch_tmppfile2_host_line` 85
`bulk_launch_tmppfile2_host_lines` 85
`bulk_launch_tmppfile2_trailer_line` 85
 data format 282
`dll_read_all_symbols` 248
`dll_read_loader_symbols_only` 248
`dll_read_no_symbols` 248
`dpvm` 153
`EXECUTABLE_PATH` 58, 70, 72, 152, 214
`LINES_PER_SCREEN` 206

parallel_attach 127
 parallel_stop 126
 pop_at_breakpoint 69
 pop_on_error 69
 process_load_callbacks 57
 PROMPT 208
 pvm 153
 server_launch_enabled 83, 86, 91
 server_launch_string 83
 server_launch_timeout 84
 SHARE_ACTION_POINT 356, 360, 361
 signal_handling_mode 68
 STOP_ALL 356, 360
 suffixes 52
 ttf 281
 warn_step_throw 69
 \$clid built-in variable 394
 Close command 175, 298
 Close command (Visualizer) 186
 Close Relatives command 175
 Close Similar command 175, 298
 Close, in dataset window 186
 closed loop, see closed loop
 closing similar windows 175
 closing variable windows 298
 closing windows 175
 cluster ID 394
 \$code data type 316
 code constructs supported
 assembler 390
 C/C++ 385
 Fortran 387
 \$code data type 297, 320
 code fragments 366, 390, 394
 modifying instruction path 366
 when executed 366
 which programming languages 366
 code, branching around 369
 collapsing structures 287
 colons as array separators 334
 colors used 229
 columns, displaying 308
 comm assembler pseudo op 393
 command arguments 206
 clearing example 206
 passing defaults 206
 setting 206
 Command Line command 53, 201
 Command Line Interpreter 14
 command prompts 207
 default 207
 format 207
 setting 208
 starting the CLI from 201
 command scope 331
 command-line options 204
 -a 54, 206
 launch Visualizer 193
 -nodes_allowed 144
 passing to TotalView 54
 -remote 55, 83
 -s startup 202
 commands 53
 Action Point > At Location 5, 353
 Action Point > Delete All 356
 Action Point > Properties 129, 356, 359, 361, 363, 365
 Action Point > Save All 117, 380
 Action Point > Save As 380
 Action Point > Set Barrier 363
 Action Point > Suppress All 356
 Add to Expression List 307
 Auto Visualize (Visualizer) 185
 change Visualizer launch 195
 Clear All STOP and EVAL 356
 CLI, see CLI commands
 Custom Groups 276
 dmpirun 112, 113
 dpvm 153
 Edit > Copy 177
 Edit > Cut 177
 Edit > Delete 177
 Edit > Delete All Expressions 309
 Edit > Delete Expression 309
 Edit > Duplicate Expression 310
 Edit > Find 4, 224
 Edit > Find Again 224
 Edit > Paste 177
 Edit > Reset Defaults 309
 Edit > Undo 177
 File > Close 175, 298
 File > Close (Visualizer) 186
 File > Close Similar 175, 298
 File > Delete (Visualizer) 185, 186
 File > Edit Source 227
 File > Exit (Visualizer) 185
 File > New Program 53, 58, 59, 61, 63, 66, 73, 83, 86, 91, 95, 98
 File > Options (Visualizer) 186
 File > Preferences 73
 Formatting page 282
 Launch Strings page 194
 Options page 281
 Pointer Dive page 293
 File > Save Pane 177
 File > Search Path 58, 70, 72, 118, 152
 File > Signals 68
 Group > Attach 121, 122
 Group > Attach Subset 124
 Group > Control > Go 231
 Group > Detach 62
 Group > Edit 262
 Group > Go 117, 127, 236, 361
 Group > Halt 127, 229, 239
 Group > Hold 232
 Group > Kill 106, 243
 Group > Next 128
 Group > Release 232
 Group > Restart 243
 Group > Run To 127
 Group > Step 128
 group or process 127
 interrupting 201
 Load All Symbols in Stack 248
 mpirun 114, 118, 122
 Options > Auto Visualize 183
 poe 101, 115
 Process > Create 237
 Process > Detach 63
 Process > Go 113, 114, 116, 120, 122, 127, 236, 243
 Process > Halt 127, 229, 239
 Process > Hold 232
 Process > Next 237
 Process > Next Instruction 237
 Process > Out 253
 Process > Run To 253
 Process > Startup 55
 Process > Step 237
 Process > Step Instruction 237
 Process Startup Parameters 73
 prun 120
 pvm 152, 153
 remsh 91
 rsh 91, 116
 server launch, arguments 87
 Set Signal Handling Mode 153
 single-stepping 239
 Startup 55
 step 4
 Thread > Continuation Signal 62, 242
 Thread > Go 237
 Thread > Hold 232
 Thread > Set PC 249
 Tools > Attach Subset 125
 Tools > Call Graph 179
 Tools > Command Line 201
 Tools > Create Checkpoint 244
 Tools > Evaluate 192, 193, 245, 302, 389
 Tools > Evaluate, see Expression List window
 Tools > Global Arrays 150
 Tools > Manage Shared Libraries 245
 Tools > Message Queue 109
 Tools > Message Queue Graph 11, 107
 Tools > Program Browser 284
 Tools > PVM Tasks 155
 Tools > Restart 244
 Tools > Statistics 343
 Tools > Thread Objects 329
 Tools > Variable Browser 291
 Tools > View Across 160
 Tools > Visualize 8, 183
 Tools > Visualize Distribution 159

Tools > Watchpoint 10, 378
 totalview
 core files 53, 63
 totalview command 53, 112, 116,
 118, 122
 totalviewcli command 53, 118, 122
 tvdsvr 81
 launching 87
 View > Add to Expression List 303
 View > Assembler > By Address
 172
 View > Assembler > Symbolically
 172
 View > Block Status 295
 View > Collapse All 287
 View > Compilation Scope 288
 View > Display Managers 168
 View > Dive 310
 View > Dive In All 300
 View > Dive in New Window 7
 View > Dive Thread 330
 View > Dive Thread New 330
 View > Examine Format > Raw
 294
 View > Examine Format > Structured 294
 View > Expand All 287
 View > Graph (Visualizer) 185
 View > Lookup 154
 View > Lookup Function 224, 227
 View > Lookup Variable 284, 293,
 296, 326, 336
 View > Reset 226, 227
 View > Reset (Visualizer) 190
 View > Source As > Assembler
 171
 View > Source As > Both 172, 249
 View > Source As > Source 171
 View > Surface (Visualizer) 185
 View > View Across > None 345
 View > View Across > Process 345
 View > View Across > Thread 345
 Visualize 8
 visualize 193, 195
 Window > Duplicate 175, 300
 Window > Duplicate Base Window
 (Visualizer) 186
 Window > Memorize 176
 Window > Memorize All 176
 Window > Update 231
 Window > Update (PVM) 155
 common block
 displaying 324
 diving on 324
 members have function scope 324
 comparing variable values 289
 comparisons in filters 342
 Compilation Scope > Floating com-
 mand 304
 Compilation Scope command 288
 compiled expressions 370
 allocating patch space for 371
 performance 370
 compiled in scope list 331
 compiling
 -g compiler option 51, 52
 multiprocess programs 51
 -O option 52
 optimization 52
 programs 3, 51
 completion rules for arena specifiers
 272
 \$complex data type 316
 \$complex_8 data type 316
 \$complex_16 data type 316
 compound objects 313
 conditional breakpoints 366, 368, 395
 conditional watchpoints, see watch-
 points
 conf file 89
 configure command 100
 configuring the Visualizer 193
 connection for serial line 94
 connection timeout 83, 85
 altering 83
 connection timeout, bulk server
 launch 85
 contained functions 326
 displaying 327
 context menus 166
 Add to Expression 9
 continuation signal 242
 clearing 243
 Continuation Signal command 62, 242
 continuing with a signal 242
 continuous execution 201
 Control Group and Share Groups Ex-
 amples figure 236
 control groups 24, 235
 defined 22
 discussion 235
 overview 262
 specifier for 264
 control in parallel environments 210
 control in serial environments 210
 control registers 250
 interpreting 250
 controlling program execution 210
 conversion rules for filters 338
 Copy command 177
 copying 177
 copying between windows 177
 core dump, naming the signal that
 caused 64
 core files
 debugging 54
 examining 63
 in totalview command 53, 63
 multi-threaded 63
 opening 60, 61
 correcting programs 369
 count array statistic 343
 \$count built-in function 395
 \$countall built-in function 395
 countdown breakpoints 368, 395
 counter, loop 368
 \$countthread built-in function 395
 CPU registers 250
 cpu_use option 115
 Cray
 configuring TotalView for 145
 loading TotalView 146
 qsub 146
 starting the CLI 146
 starting TotalView 146
 Cray XT CNLL
 using TotalView 146
 Cray XT3 debugging 144
 tvdsvr 144
 Create Checkpoint command 244
 creating custom groups 276
 creating groups 26, 236
 creating new processes 204
 creating processes 236
 and starting them 236
 using Step 237
 without starting it 237
 without starting them 237
 creating threads 18
 creating type transformations 281
 crt0.o module 154
 Ctrl+C 201
 current location of program counter
 171
 current set indicator 259, 276
 current stack frame 227
 current working directory 70, 72
 Custom Groups command 276
 Cut command 177
 Cycle Detection tab 107

D

D control group specifier 264
 dactions command 350
 -load 117, 380
 -save 117, 380
 daemons 16, 18
 dassign command 310
 data
 editing 7
 examining 6
 filtering 8
 see also Variable Window
 slicing 8
 viewing, from Visualizer 183
 data assembler pseudo op 393
 data dumping 294
 data filtering, see arrays, filtering
 data precision, changing display 77
 data types 316

see also TotalView data types
 C++ 322
 changing 311
 changing class types in C++ 323
 for visualization 182
 int 312
 int* 312
 int[] 312
 opaque data 320
 pointers to arrays 314
 predefined 315
 to visualize 182
 data watchpoints, see watchpoints
 data_format variables 282
 dataset
 defined for Visualizer 182
 visualizing 192
 window (Visualizer) 185
 window (Visualizer), display
 commands 186
 window, menu commands 185
 deleting 185
 dattach command 54, 61, 63, 102,
 117, 118, 124, 210
 mprun command 123
 dbarrier command 362, 364
 -e 367
 -stop_when_hit 129
 dbfork library 52, 361
 linking with 52
 dbreak command 219, 352, 354, 360
 -e 367
 dcheckpoint command 244
 ddelete command 106, 354, 356, 365
 dd detach command 63
 ddisable command 356, 365
 ddlopen command 245
 ddown command 241
 deadlocks 255
 message passing 109
 \$debug assembler pseudo op 393
 -debug, using with MPICH 106
 debugger initialization 202
 debugger PID 210
 debugger server 81
 starting manually 86
 Debugger Unique ID (DUID) 394
 debugging
 core file 54
 executable file 53
 multiprocess programs 52
 not compiled with -g 52
 OpenMP applications 132
 over a serial line 94
 PE applications 115
 programs that call execve 52
 programs that call fork 52
 PVM applications 151, 152
 QSW RMS 120

SHMEM library code 156
 UPC programs 158
 debugging Fortran modules 326
 debugging MPI programs 55
 debugging session 211
 debugging symbols, reading 246
 debugging techniques 30, 106, 124
 declared arrays, displaying 321
 def assembler pseudo op 393
 default address range conflicts 372
 default control group specifier 264
 default focus 269
 default process/thread set 258
 default programming language 52
 default text editor 228
 default width specifier 260
 deferred shape array
 definition 334
 types 327
 deferred symbols
 force loading 248
 reading 246
 deferring order for shared libraries 247
 Delete All command 356
 Delete command 177
 Delete command (Visualizer) 185, 186
 Delete, in dataset window 186
 deleting
 action points 356
 datasets 185
 programs 243
 denable command 356, 357
 denorm filter 339
 denormalized count array statistic 343
 DENORMs 337
 deprecated X defaults 78
 deprecated, defined 78
 dereferencing 7
 automatic 293
 controlling 78
 pointers 293
 Detach command 62, 63
 detaching 125
 detaching from processes 62
 detaching removes all breakpoints 63
 detecting cycles 107
 determining scope 251, 289
 dfocus command 239, 257, 258
 example 258
 dga command 150
 dgo command 113, 116, 117, 122,
 127, 236, 273
 dgroups command
 -add 262, 268
 -remove 31
 dhalt command 127, 229, 239
 dhold command 232, 363
 -process 232
 -thread 232
 difference operator 275
 directories, setting order of search 70
 directory search path 152
 disabling
 action points 356
 autolaunch 83, 91
 autolaunch feature 84
 visualization 193
 disassembled machine code 225
 in variable window 297
 discard dive stack 226
 discard mode for signals 70
 discarding signal problem 70
 disconnected processing 16
 displaying 174
 areas of memory 296
 argv array 321
 array data 174
 arrays 333, 334
 blocks 287
 columns 308
 common blocks 324
 declared and allocated arrays 321
 exited threads 169
 Fortran data types 324
 Fortran module data 324
 global variables 284, 291
 long variable names 285
 machine instructions 297
 memory 296
 pointer 174
 pointer data 174
 Process window 174
 registers 292
 remote hostnames 167
 stack trace pane 174
 STL variables 280
 structs 315
 subroutines 174
 thread objects 329
 typedefs 315
 unions 315
 variable 174
 Variable Windows 283
 distributed debugging
 see also PVM applications
 remote server 81
 dive icon 175, 298
 Dive In All command 300, 301
 Dive In New Window command 7
 Dive Thread command 330
 Dive Thread New command 330
 dividing work up 17
 diving 108, 118, 166, 174, 283
 creating call_graph group 180
 defined 7
 in a "view acrosss" pane 346
 in a variable window 298
 in source code 226
 into a pointer 174, 298
 into a process 174

into a stack frame 174
 into a structure 298
 into a thread 174
 into a variable 7, 174
 into an array 298
 into formal parameters 292
 into Fortran common blocks 324
 into function name 226
 into global variables 284, 291
 into local variables 292
 into MPI buffer 111
 into MPI processes 110
 into parameters 292
 into pointer 174
 into processes 174
 into PVM tasks 155
 into registers 292
 into routines 174
 into the PC 297
 into threads 170, 174
 into variables 174
 nested 174
 nested dive defined 298
 program browser 291
 registers 283
 scoping issue 289
 using middle mouse button 177
 variables 283
 dkill command 128, 204, 210, 243
 dll_read_all_symbols variable 248
 dll_read_loader_symbols variable 248
 dll_read_loader_symbols_only variable 248
 dll_read_no_symbols variable 248
 dload command 61, 86, 203, 204, 210
 returning process ID 205
 dlopen(), using 245
 dmg installer 53
 dmpirun command 112, 113
 dnext command 128, 237, 240
 dnxti command 237, 240
 double assembler pseudo op 393
 \$double_precision data type 316
 dout command 241, 253
 dpid 210
 dprint command 135, 136, 216, 225,
 250, 284, 285, 293, 296, 315,
 321, 324, 325, 327, 334, 336
 dptsets command 66, 230
 DPVM
 see also PVM
 enabling support for 153
 must be running before TotalView
 153
 starting session 153
 -dpvm command-line option 153
 dpvm shell command 153
 dpvm variable 153
 drawing options 187
 drerun command 204, 243

drestart command 244
 drun command 203, 206
 dset command 206, 208
 dstatus command 66, 365
 dstep command 237, 240, 253, 259,
 261, 273
 dstep commands 128
 dstepi command 237, 240
 DUID 394
 of process 394
 \$duid built-in variable 394
 dunhold command 232, 363
 -thread 232
 dunset command 206
 duntil command 241, 253, 255
 dup command 241
 dup commands 285
 Duplicate Base Window
 in Visualizer dataset window 186
 Duplicate command 175, 300
 dwhere command 260, 273, 285
 dynamic call graph 179
 Dynamic Libraries page 246
 dynamic patch space allocation 371
 dynamically linked, stopping after
 start() 154

E

E state 67
 Edit > Copy command 177
 Edit > Cut command 177
 Edit > Delete All Expressions command 309
 Edit > Delete command 177
 Edit > Delete Expression command 309
 Edit > Duplicate Expression command 310
 Edit > Find Again command 224
 Edit > Find command 4, 224
 Edit > Paste command 177
 Edit > Reset Defaults command 309
 Edit > Undo command 177
 edit mode 166
 Edit Source command 227
 editing
 addresses 321
 compound objects or arrays 313
 source text 227
 text 176
 type strings 311
 view across data 346
 editing groups 277
 EDITOR environment variable 228
 editor launch string 227
 effects of parallelism on debugger behavior 209
 Enable action point 356
 Enable Single Debug Server Launch check box 91
 Enable Visualizer Launch check box 194
 enabling
 action points 356
 environment variables
 adding 65
 before starting poe 115
 EDITOR 228
 how to enter 65
 LC_LIBRARY_PATH 57
 LM_LICENSE_FILE 57
 MP_ADAPTER_USE 115
 MP_CPU_USE 115
 MP_EUIDEVELOP 111
 PATH 70, 71
 SHLIB_PATH 57
 TOTALVIEW 55, 101, 129
 TVDSVRLAUNCHCMD 87
 equiv assembler pseudo op 393
 error state 67
 errors, in multiprocess program 69
 EEXECUTABLE_PATH variable 72
 EVAL icon 166
 for evaluation points 166
 eval points
 and expression system 383
 see evaluation points
 Evaluate command 192, 193, 389, 394
 Evaluate Window
 expression system 384
 Evaluate window 383
 evaluating an expression in a watchpoint 374
 evaluating expressions 389
 evaluating state 211
 evaluation points 5, 366
 assembler constructs 390
 C constructs 385
 clearing 166
 defined 211, 350
 defining 366
 examples 368
 Fortran constructs 387
 hitting breakpoint while evaluating 390
 listing 171
 lists of 171
 machine level 366
 patching programs 6
 printing from 5
 saving 367
 setting 166, 218, 367
 using \$stop 6
 where generated 366
 evaluation system limitations 384
 evaluation, see also expression system
 event points listing 171
 Examine Format > Raw Format command 294

Examine Format > Structured command 294
 examining
 core files 63
 data 6
 memory 294
 processes 235
 source and assembler code 171
 stack trace and stack frame 292
 status and control registers 250
 exception enable modes 250
 excluded information, reading 248
 exclusion list, shared library 247
 EXECUTABLE_PATH tab 71
 EXECUTABLE_PATH variable 58, 70, 152, 214
 setting 214
 executables
 debugging 53
 specifying name in scope 332
 execution
 controlling 210
 halting 229
 out of function 241
 resuming 231
 startup file 56
 to completion of function 241
 execution models 11
 execve() 52, 235, 361
 debugging programs that call 52
 setting breakpoints with 361
 existent operator 275
 exit CLI command 58
 Exit command 57
 Exit command (Visualizer) 185
 exited threads, displaying 169
 expanding structures 287
 expression evaluation window
 compiled and interpreted expressions 370
 discussion 389
 Expression List window 8, 284, 298, 303, 313
 Add to Expression List command 303
 aggregates 305
 and expression system 383
 array of structures 305
 diving 305
 editing contents 309
 editing the value 309
 editing type field 309
 entering variables 303
 expressions 305
 highlighting changes 305
 multiple windows 307
 multiprocess/multithreaded behavior 307
 rebinding 308
 reevaluating 307

reopening 308
 reordering rows 309
 restarting your program 308
 selecting before sending 304
 sorting columns 309
 Expression List window, 383
 expression system
 accessing array elements 382
 and arrays 382
 C/C++ declarations 386
 C/C++ statements 386
 defined 381
 eval points 383
 Expression List Window 383
 Fortran 387
 Fortran intrinsics 388
 functions and their issues 383
 methods 382
 structures 382
 templates and limitations 386
 Tools > Evaluate Window 384
 using C++ 385
 Variable Window 383
 expressions 275, 360
 can contain loops 389
 changing in Variable Window 302
 compiled 370
 evaluating 389
 in Expression List window 305
 performance of 370
 side effects 302
 expressions and variables 302
 \$extended data type 317
 extent of arrays 315

F

figures
 Action Point > Properties Dialog Box 355, 359, 364
 Action Point Symbol 350
 Ambiguous Function Dialog Box 225, 355
 Ambiguous Line Dialog Box 352, 353
 Array Data Filter by Range of Values 341
 Array Data Filtering by Comparison 339
 Array Data Filtering for IEEE Values 340
 Array Statistics Window 343
 Breakpoint at Assembler Instruction Dialog Box 358
 Control and Share Groups Example 236
 File > Preferences: Action Points Page 360
 Five Processes and Their Groups on Two Computers 25
 Fortran Array with Inverse Order and Limited Extent 336
 PC Arrow Over a Stop Icon 359
 Sorted Variable Window 342
 Stopped Execution of Compiled Expressions 371
 Stride Displaying the Four Corners of an Array 335
 Tools > Evaluate Dialog Box 390, 391
 Tools > Watchpoint Dialog Box 376
 Two Computers Working on One Problem 17
 Undive/Redive Buttons 299
 Using Assembler 391
 View > Display Exited Threads 169
 Viewing Across an Array of Structures 346
 Viewing Across Threads 345
 Waiting to Complete Message Box 390
 File > Close command 175, 298
 File > Close command (Visualizer) 186
 File > Close Relatives command 175
 File > Close Similar command 175, 298
 File > Delete command (Visualizer) 185, 186
 File > Edit Source command 227
 File > Exit command 57
 File > Exit command (Visualizer) 185
 File > New Program command 53, 58, 59, 61, 63, 66, 73, 83, 86, 91, 95, 98
 File > Options command (Visualizer) 186
 File > Preferences
 Bulk Launch page 86
 Options page 176
 File > Preferences command
 Action Points page 69, 74, 127
 Bulk Launch page 75, 84, 86
 different values between platforms 73
 Dynamic Libraries page 76, 246
 Fonts page 77
 Formatting page 77, 282
 Launch Strings page 75, 83, 194
 Options page 69, 74, 281
 overview 73
 Parallel page 76, 126
 Pointer Dive page 78, 293
 File > Preferences: Action Points Page figure 360
 File > Save Pane command 177
 File > Search Path command 58, 70, 71, 72, 118, 152
 search order 70, 71
 File > Signals command 68

-file command-line option to Visualizer 193, 195
 file extensions 52
 file, start up 56
 files
 .rhosts 116
 hosts.equiv 116
 fill assembler pseudo op 393
 filter expression, matching 337
 filtering 8
 array data 337, 338
 array expressions 341
 by comparison 338
 comparison operators 338
 conversion rules 338
 example 338
 IEEE values 339
 options 337
 ranges of values 341
 unsigned comparisons 339
 filters 342
 \$denorm 339
 \$inf 339
 \$nan 339
 \$nanq 339
 \$nans 339
 \$ninf 339
 \$pdenorm 339
 \$pinf 339
 comparisons 342
 Find Again command 224
 Find command 4, 224
 finding
 functions 225
 source code 225, 227
 source code for functions 225
 first thread indicator of < 259
 Five Processes and Their Groups on Two Computers figure 25
 \$float data type 317
 float assembler pseudo op 393
 floating scope 304
 focus
 as list 272
 changing 258
 pushing 258
 restoring 258
 setting 257
 for loop 389
 Force window positions (disables window manager placement modes)
 check box 176
 fork() 52, 235, 361
 debugging programs that call 52
 setting breakpoints with 361
 fork_loop.tvd example program 202
 Formatting page 282
 Fortran
 array bounds 314
 arrays 314

common blocks 324
 contained functions 326
 data types, displaying 324
 debugging modules 326
 deferred shape array types 327
 expression system 387
 filter expression 341
 in code fragment 366
 in evaluation points 387
 intrinsics in expression system 388
 module data, displaying 324
 modules 324, 326
 pointer types 328
 type strings supported by
 TotalView 312
 user defined types 327
 Fortran Array with Inverse Order and Limited Extent figure 336
 Fortran casting for Global Arrays 149, 150
 Fortran modules 329
 command 325
 Fortran parameters 329
 forward icon 175
 four linked processors 19
 4142 default port 86
 frame pointer 241
 freezing window display 289
 function calls, in eval points 369
 function visualization 179
 functions
 finding 225
 IEEE 340
 in expression system 383
 locating 224
 returning from 242

G

-g compiler option 51, 52, 174
 g width specifier 265, 269
 \$GA cast 149, 150, 149
 \$ga cast 149, 150
 gcc UPC compiler 158
 generating a symbol table 52
 Global Arrays 149
 casting 149, 150
 debugging 148
 diving on type information 150
 Intel IA-64 149
 global assembler pseudo op 393
 global variables
 changing 237
 displaying 237
 diving into 284, 291
 gnu_debuglink file 56
 Go command 4, 113, 116, 117, 120, 122, 127, 236
 GOI defined 251
 going parallel 126
 goto statements 366
 Graph command (Visualizer) 185
 Graph Data Window 186
 graph points 186
 Graph visualization menu 185
 graph window, creating 185
 Graph, in Dataset Window 185
 graphs, two dimensional 186
 group
 process 256
 thread 256
 Group > Attach Subset command 121, 122, 124
 Group > Control > Go command 231
 Group > Custom Group command 31
 Group > Detach command 62
 Group > Edit command 262
 Group > Go command 117, 127, 233, 236, 361
 Group > Halt command 127, 229, 239
 Group > Hold command 232
 Group > Kill command 106, 128, 243
 Group > Next command 128
 Group > Release command 232
 Group > Restart command 243
 Group > Run To command 127
 Group > Step command 128
 group aliases 208
 limitations 208
 group commands 127
 group indicator
 defined 263
 group name 264
 group number 264
 group stepping 254
 group syntax 263
 group number 264
 naming names 264
 predefined groups 263
 GROUP variable 269
 group width specifier 260
 groups 152
 see also processes
 and barriers 12
 behavior 254
 creating 26, 236, 276
 defined 22, 23
 editing 277
 examining 235
 holding processes 232
 overview 22
 process 255
 relationships 260
 releasing processes 232
 running 126
 selecting processes for 276
 setting 268
 starting 236
 stopping 126
 thread 255

Groups > Custom Groups command
180, 276
GUI namespace 207

H

h held indicator 231
–h localhost option for HP MPI 114
half assembler pseudo op 393
Halt command 127, 229, 239
halt commands 229
halting 229
 groups 229
 processes 229
 threads 229
handler routine 68
handling signals 68, 153
held indicator 231
held operator 275
held processes, defined 363
hexadecimal address, specifying in variable window 296
hi16 assembler operator 392
hi32 assembler operator 392
hierarchy toggle button, Root Window 168
highlighted variables 285, 286
highlighting changes in Expression List window 305
hitting a barrier point 365
hold and release 231
\$Hold assembler pseudo op 393
\$Hold built-in function 396
Hold command 232
hold state 232
 toggling 363
Hold Threads command 232
holding and advancing processes 210
holding problems 234
holding threads 256
\$Holdprocess assembler pseudo op 393
\$Holdprocess built-in function 396
\$Holdprocessall built-in function 396
\$Holdprocessstopall assembler pseudo op 393
\$Holdstopall assembler pseudo op 393
\$Holdstopall built-in function 396
\$Holdthread assembler pseudo op 393
\$Holdthread built-in function 396
\$Holdthreadstop assembler pseudo op 393
\$Holdthreadstop built-in function 396
\$Holdthreadstopall assembler pseudo op 393
\$Holdthreadstopall built-in function 396
\$Holdthreadstopprocess assembler pseudo op 393

\$Holdthreadstopprocess built-in function 396

hostname
 expansion 90
 for tvdsvr 55
 in square brackets 167
hosts.equiv file 116
how TotalView determines share group 236
hung processes 61

I

I state 67
IBM BlueGene
 bluegene_io_interface 119
 bluegene_server_launch 119
 starting TotalView 118
 starting tvdsvrs 118
IBM cell broadband enginesee Cell broadband engine
IBM MPI 115
IBM SP machine 100, 101
idle state 67
IEEE functions 340
Ignore mode warning 70
ignoring action points 356
implicitly defined process/thread set 258
incomplete arena specifier 272
inconsistent widths 273
inf filter 339
infinite loop, see loop, infinite
infinity count array statistic 344
INFs 337
inheritance hierarchy 385
initial process 209
initialization search paths 56
initialization subdirectory 56
initializing an array slice 215
initializing debugging state 56
initializing the CLI 202
initializing TotalView 56
instructions
 data type for 320
 displaying 297
\$int data type 317
int data type 312
int* data type 312
int[] data type 312
\$integer_2 data type 317
\$integer_4 data type 317
\$integer_8 data type 317
interactive CLI 199
interface to CLI 201
internal counter 368
interpreted expressions 370
 performance 370
interrupting commands 201
intersection operator 275
intrinsic functions

\$is_Inf 340
\$is_inf 340
\$is_nan 340
\$is_ndenorm 340
\$is_ninf 340
\$is_nnrm 340
\$is_norm 340
\$is_pdenorm 340
\$is_pinf 340
\$is_pnom 340
\$is_pzero 340
\$is_qnan 340
\$is_snan 340
\$is_zero 340

intrinsics, see built-in functions
inverting array order 335
inverting axis 187
invoking CLI program from shell example 202
invoking TotalView on UPC 158
IP over the switch 115
iterating
 over a list 273
 over arenas 259

J

joystick mode, Visualizer 181

K

K state, unviewable 67
–KeepSendQueue command-line option 112
kernel 67
Kill command 128, 243
killing processes when exiting 62
killing programs 243
–ksq command-line option 112

L

L lockstep group specifier 264, 265
labels, for machine instructions 297
LAM/MPI 119
 starting 120
Last Value column 285, 305
launch
 configuring Visualizer 193
 options for Visualizer 194
 TotalView Visualizer from command line 193
 tvdsvr 81
Launch Strings page 83, 91, 194
lcomm assembler pseudo op 393
LD_LIBRARY_PATH environment variable 57, 158
left margin area 171
left mouse button 165
libraries
 dbfork 52
 debugging SHMEM library code 156
 naming 246

see also shared libraries
 limitations in evaluation system 384
 limiting array display 335
 line number area 166
 line numbers 171
 for specifying blocks 332
 linear view 168
 LINES_PER_SCREEN variable 206
 linked lists, following pointers 298
 list transformation, STL 281
 lists of processes 166
 lists of variables, seeing 8
 lists with inconsistent widths 273
 lists, iterating over 273
 LM_LICENSE_FILE environment variable 57
 lo16 assembler operator 392
 lo32 assembler operator 392
 Load All Symbols in Stack command 248
 loader symbols, reading 246
 loading
 file into TotalView 54
 new executables 58
 remote executables 55
 shared library symbols 247
 loading loader symbols 247
 loading no symbols 247
 local hosts 55
 locations, toggling breakpoints at 353
 lockstep group 25, 252, 259
 defined 23
 L specifier 264
 number of 263
 overview 263
 \$logical data type 317
 \$logical_1 data type 317
 \$logical_2 data type 317
 \$logical_4 data type 317
 \$logical_8 data type 317
 \$long data type 317
 long variable names, displaying 285
 \$long_branch assembler pseudo op 393
 \$long_long data type 317
 Lookup Function command 154, 224, 227
 Lookup Variable command 136, 224, 284, 293, 296, 326
 specifying slices 336
 loop counter 368
 loop infinite, see infinite loop
 lower adjacent array statistic 344
 lower bounds 314
 non default 314
 of array slices 334
 lysm TotalView pseudo op 393

M

M state 67

Mac OS X
 procmod permission 54
 starting execution 53
 starting from an xterm 53
 machine instructions
 data type 320
 data type for 320
 displaying 297
 main() 154
 stopping before entering 154
 make_actions.tcl sample macro 202, 218
 manager processes, displaying 168
 manager threads 21, 25
 displaying 168
 manual hold and release 231
 manually starting tvdsvr 91
 map templates 280
 map transformation, STL 280
 master process, recreating slave processes 128
 master thread 133
 OpenMP 133, 137
 stack 135
 matching processes 255
 matching stack frames 345
 maximum array statistic 344
 mean array statistic 344
 median array statistic 344
 Memorize All command 176
 Memorize command 176
 memory contents, raw 295
 memory information 295
 memory locations, changing values of 310
 memory, displaying areas of 296
 memory, examining 294
 menus, context 166
 message passing deadlocks 109
 Message Passing Interface/Chameleon Standard, see MPICH
 Message Queue command 109
 message queue display 106, 122
 Message Queue Graph 108
 diving 108
 rearranging shape 109
 updating 108
 Message Queue Graph command 107
 message queue graph window 11
 message tags, reserved 156
 message-passing programs 127
 messages
 envelope information 111
 operations 110
 reserved tags 156
 unexpected 111
 messages from TotalView, saving 205
 methods, in expression system 382
 middle mouse button 165
 middle mouse dive 177
 minimum array statistic 344
 missing TID 260
 mixed state 67
 mixing arena specifiers 273
 modify watchpoints, see watchpoints
 modifying code behavior 366
 module data definition 324
 modules 324, 326
 debugging
 Fortran 326
 displaying Fortran data 324
 modules in Fortran 329
 more processing 206
 more prompt 206
 mouse button
 diving 165
 left 165
 middle 165
 right 166
 selecting 165
 mouse buttons, using 165
 MP_ADAPTER_USE environment variable 115
 MP_CPU_USE environment variable 115
 MP_EUIDEVELOP environment variable 111
 MP_TIMEOUT 116
 MPI
 attaching to 122
 attaching to HP job 114
 attaching to running job 113
 buffer diving 111
 communicators 109
 debugging 55
 LAM 119
 library state 109
 on IBM 115
 on SGI 122
 on SiCortex 121
 on Sun 123
 process diving 110
 rank display 106
 starting 98
 starting on Cray 112
 starting on HP Alpha 112
 starting on HP machines 113
 starting on SGI 122
 starting processes 113, 121
 starting processes, SGI 122
 toolbar settings for 13
 troubleshooting 106
 mpi tasks, attaching to 126
 MPI_Init() 109, 117
 breakpoints and timeouts 129
 MPI_Iprobe() 111
 MPI_Recv() 111
 MPICH 100, 101
 and SIGINT 106

and the TOTALVIEW environment variable 101
 attach from TotalView 102
 attaching to 102
 ch_lfshmem device 100, 102
 ch_mpl device 100
 ch_p4 device 100, 102
 ch_shmem device 102
 ch_smem device 100
 configuring 100
 debugging tips 129
 diving into process 102
 MPICH/ch_p4 129
 mpirun command 100, 101
 naming processes 103
 obtaining 100
 P4 103
 -p4pg files 103
 starting TotalView using 100
 -tv command-line option 100
 using –debug 106
 mpirun command 100, 101, 114, 118, 122, 129
 examples 114
 for HP MPI 114
 options to TotalView through 129
 passing options to 129
 mpirun process 122
 MPL_Init() 117
 and breakpoints 117
 mprun command 123
 MOD, see message queue display
 multiple classes, resolving 226
 Multiple indicator 346
 multiple sessions 151
 multi-process debugging 10
 multi-process programming library 52
 multi-process programs
 and signals 69
 compiling 51
 process groups 235
 setting and clearing breakpoints 359
 multiprocessing 19
 multi-threaded core files 63
 multi-ithreaded debugging 10
 multi-threaded signals 242

N

–n option, of rsh command 92
 –n single process server launch command 87
 names of processes in process groups 235
 namespaces 207
 TV:: 207
 TV::GUI:: 207
 naming libraries 246
 naming MPICH processes 103
 naming rules

for control groups 235
 for share groups 235
 nan filter 339
 nanq filter 339
 NaNs 337, 339
 array statistic 344
 nans filter 339
 navigating, source code 227
 ndenorm filter 339
 nested dive 174
 defined 298
 window 299
 nested stack frame, running to 256
 New Program command 53, 58, 59, 61, 63, 66, 73, 86, 91, 95, 98
 Next command 128, 237, 239
 "next" commands 240
 Next Instruction command 237
 \$nid built-in variable 394
 ninf filter 339
 –no_stop_all command-line option 129
 node ID 394
 nodes, attaching from to poe 117
 nodes, detaching 125
 –nodes_allowed command-line option 144
 Cray 145
 –nodes_allowed tvdsvr command-line option 144
 nodes_allowed,tvdsvr command-line option 144
 None (lView Across) command 345
 nonexistent operators 275
 non-sequential program execution 201

O

–O option 52
 offsets, for machine instructions 297
 \$oldval built-in variable 394
 omitting array stride 335
 omitting components in creating scope 332
 omitting period in specifier 272
 omitting width specifier 272
 opaque data 320
 opaque type definitions 320
 Open process window at breakpoint check box 69
 Open process window on signal check box 69
 opening a core file 60, 61
 opening shared libraries 245
 OpenMP 132, 133
 debugging 132
 debugging applications 132
 master thread 133, 135, 137
 master thread stack context 135
 on HP Alpha 133

private variables 134
 runtime library 132
 shared variables 134, 137
 stack parent token 137
 THREADPRIVATE common blocks 136
 THREADPRIVATE variables 136
 threads 133
 TotalView-supported features 132
 viewing shared variables 135
 worker threads 133

operators
 – difference 275
 & intersection 275
 | union 275
 breakpoint 275
 existent 275
 held 275
 nonexistent 275
 running 275
 stopped 275
 unheld 275
 watchpoint 275
 optimizations, compiling for 52
 options
 for visualize 193
 in dataset window 186
 –patch_area 372
 –patch_area_length 372
 –sb 380
 setting 78
 Options > Auto Visualize command (Visualizer) 183, 185
 Options command (Visualizer) 186
 Options page 176, 281
 org assembler pseudo op 393
 ORNL PVM, see PVM
 Out command 239
 "out" commands 241
 out command, goal 242
 outliers 344
 outlined routine 132, 136, 137
 outlining, defined 132
 output
 assigning output to variable 205
 from CLI 205
 only last command executed returned 205
 printing 205
 returning 205
 when not displayed 205

P

p width specifier 265
 P+/P- buttons 230
 p.t notation 259
 P/T set controls 256
 p/t sets
 arguments to Tcl 258
 defined 257

expressions 275
 set of arenas 259
 syntax 260
 p/t syntax, group syntax 263
 p4 listener process 102
 -p4pg files 103
 -p4pg option 103
 panes
 source code, see source code pane
 stack frame, see stack frame pane
 stack trace, see stack trace pane
 panes, saving 177
 parallel debugging tips 124
 PARALLEL DO outlined routine 133
 Parallel Environment for AIX, see PE
 parallel environments, execution control of 210
 Parallel page 126
 parallel program, defined 209
 parallel program, restarting 128
 parallel region 133
 Parallel tab, File > New Program 98
 parallel tasks, starting 117
 Parallel Virtual Machine, see PVM
 parallel_attach variable 127
 parallel_stop variables 126
 parameters, displaying in Fortran 329
 parsing comments example 218
 passing arguments 54
 passing default arguments 206
 Paste command 177
 pasting 177
 between windows 177
 with middle mouse 165
 patch space size, different than 1MB 372
 patch space, allocating 371
 -patch_area_base option 372
 -patch_area_length option 372
 patching
 function calls 369
 programs 368
 PATH environment variable 58, 70, 71
 pathnames, setting in procgroup file 103
 PC Arrow Over a Stop Icon figure 359
 PC icon 249
 pdenorm filter 339
 PE 117
 adapter_use option 115
 and slow processes 129
 applications 115
 cpu_use option 115
 debugging tips 129
 from command line 116
 from poe 116
 options to use 115
 switch-based communication 115
 PE applications 115
 pending messages 109
 pending receive operations 110, 111
 pending send operations 110, 112
 configuring for 112
 pending unexpected messages 110
 performance of interpreted, and compiled expressions 370
 performance of remote debugging 81
 -persist command-line option to Visualizer 193, 195
 phase, UPC 160
 pick, Visualizer 181
 picking a dataset point value 187
 \$pid built-in variable 394
 pid specifier, omitting 272
 pid.tid to identify thread 170
 pinf filter 339
 piping data 177
 piping information 177
 plant in share group 360
 Plant in share group check box 361, 367
 poe
 and mpirun 101
 and TotalView 116
 arguments 115
 attaching to 117, 118
 interacting with 129
 on IBM SP 102
 placing on process list 118
 required options to 115
 running PE 116
 TotalView acquires poe processes 117
 poe, and bulk server launch 90
 POI defined 251
 point of execution for multiprocess or multithreaded program 171
 pointer data 174
 Pointer Dive page 293
 pointers 174
 as arrays 293
 chasing 293, 298
 dereferencing 293
 diving on 174
 in Fortran 328
 to arrays 314
 pointer-to-shared UPC data 160
 points, in graphs 186
 pop_at_breakpoint variable 69
 pop_on_error variable 69
 popping a window 174
 port 4142 86
 -port command-line option 86
 port number for tvdsvr 55
 PPE
 defined 138
 PPU
 organization 140
 PPUDescription 140
 precision 282
 changing 282
 changing display 77
 predefined data types 315
 preference file 56
 preferences
 Action Points page 74
 Bulk Launch page 75, 84, 86
 Dynamic Libraries page 76
 Fonts page 77
 Formatting page 77
 Launch Strings page 75, 83
 Options page 69, 74
 Parallel page 76
 Pointer Dive page 78
 setting 78
 preloading shared libraries 245
 primary thread, stepping failure 255
 print statements, using 5
 printing an array slice 215
 printing in an eval point 5
 private variables 133
 in OpenMP 134
 procedures
 debugging over a serial line 94
 displaying 321
 displaying declared and allocated arrays 321
 process
 detaching 62
 holding 256
 ID 394
 numbers are unique 209
 selecting in processes/rank tab 230
 state 65
 states 67, 171
 states, attached 67
 stepping 254
 synchronization 127, 256
 width specifier 260
 width specifier, omitting 272
 Process > Create command 237
 Process > Detach command 63
 Process > Go command 113, 114, 116, 120, 122, 127, 233, 236, 243
 Process > Halt command 127, 229, 239
 Process > Hold command 232
 Process > Hold Threads command 232
 Process > Next command 237
 Process > Next Instruction command 237
 Process > Out command 253
 Process > Release Threads command 232
 Process > Run To command 253

Process > Startup Parameters command 55, 73
 Process > Step command 237
 Process > Step Instruction command 237
 process as dimension in Visualizer 184
 process barrier breakpoint
 changes when clearing 365
 changes when setting 365
 defined 350
 deleting 365
 setting 363
 process DUID 394
 process focus 257
 process groups 23, 255, 256, 262
 behavior 268
 behavior at goal 255
 stepping 254
 synchronizing 255
 Process Window 4, 169
 displaying 174
 host name in title 167
 raising 69
 process, attaching to existing 59
 process, starting a new 59
 process/set threads
 saving 261
 process/thread identifier 209
 process/thread notation 209
 process/thread sets 209
 as arguments 258
 changing focus 258
 default 258
 implicitly defined 258
 inconsistent widths 273
 structure of 260
 target 257
 widths inconsistent 273
 process_id.thread_id 259
 process_load_callbacks variable 57
 \$processduid built-in variable 394
 processes
 see also automatic process acquisition
 see also groups
 acquiring 101, 103, 154
 acquiring in PVM applications 152
 acquisition in poe 117
 apparently hung 128
 attaching to 61, 117, 155
 barrier point behavior 365
 behavior 254
 breakpoints shared 360
 call graph 179
 changing 230
 cleanup 156
 copy breakpoints from master
 process 101
 creating 236, 237
 creating by single-stepping 237
 creating new 204
 creating using Go 236
 creating without starting 237
 deleting 243
 deleting related 243
 detaching from 62
 displaying data 174
 displaying manager 168
 diving into 118
 diving on 174
 groups 235
 held defined 363
 holding 231, 362, 396
 hung 61
 initial 209
 killing while exiting 62
 list of 166
 loading new executables 58
 master restart 128
 MPI 110
 names 235
 refreshing process info 231
 released 363
 releasing 231, 362, 365
 restarting 243
 single-stepping 253
 slave, breakpoints in 102
 spawned 209
 starting 236
 state 66
 status of 65
 stepping 12, 128, 254
 stop all related 360
 stopped 363
 stopped at barrier point 365
 stopping 229, 366
 stopping all related 69
 stopping intrinsic 396
 stopping spawned 101
 switching between 11
 synchronizing 211, 255
 tab 229
 terminating 204
 types of process groups 235
 when stopped 254
 Processes button 360
 process-level stepping 128
 processors and threads 19
 procgroup file 103
 using same absolute path names 103
 procmod permission, Mac OS X 54
 Program Browser 291
 explaining symbols 291
 program control groups
 defined 262
 naming 235
 program counter (PC) 171
 arrow icon for PC 171
 indicator 171
 setting 249
 setting program counter 248
 setting to a stopped thread 249
 program execution
 advancing 210
 controlling 210
 program state, changing 201
 Program tab 59
 program visualization 179
 programming languages, determining which used 52
 programming TotalView 14
 programs
 compiling 3, 51
 compiling using -g 51
 correcting 369
 deleting 243
 killing 243
 not compiled with -g 52
 patching 6, 368
 restarting 243
 prompt and width specifier 266
 PROMPT variable 208
 Properties command 129, 350, 355, 359, 363, 367
 Properties window 357
 properties, of action points 5
 prototypes for temp files 85
 prun command 120
 prun, and bulk server launch 89
 pthread ID 210
 pthreads, see threads
 pushing focus 258
 PVM
 acquiring processes 152
 attaching procedure 155
 attaching to tasks 155
 automatic process acquisition 154
 cleanup of tvdsrv 156
 creating symbolic link to tvdsrv 152
 daemons 156
 debugging 151
 message tags 156
 multiple instances not allowed by single user 151
 multiple sessions 151
 running with DPVM 151
 same architecture 155
 search path 152
 starting actions 154
 tasker 154
 tasker event 154
 tasks 151, 152
 TotalView as tasker 151
 TotalView limitations 151
 tvdsrv 154
 Update Command 155
 pvm command 152, 153

PVM groups, unrelated to process groups 152
 PVM Tasks command 155
 pvm variable 153
 pvm_joingroup() 156
 pvm_spawn() 152, 154, 155
 pvmgs process 152, 156
 terminated 156

Q

QSW RMS applications 120
 attaching to 121
 debugging 120
 starting 120
 quad assembler pseudo op 394
 Quadrics RMS 120
 quartiles array statistic 344

R

R state 67
 raising process window 69
 rank display 106
 rank for Visualizer 194
 ranks 107
 ranks tab 106, 229
 Raw Format command 294
 raw memory contents 294
 raw memory data 295
 read_symbols command 248
 reading loader and debugger symbols 246
 \$real data type 317
 \$real_16 data type 317
 \$real_4 data type 317
 \$real_8 data type 317
 rebinding the Variable Window 298
 recursive functions 242
 single-stepping 241
 redive 299
 redive all 299
 redive buttons 298
 redive icon 175, 298
 redive/undive buttons 7
 registers
 editing 250
 interpreting 250
 Release command 232
 release state 232
 Release Threads command 232
 reloading breakpoints 117
 remembering window positions 176
 –remote command-line option 55, 83
 Remote Debug Server Launch preferences 83
 remote debugging 81
 see also PVM applications
 launching tvdsvr 81
 performance 81
 remote executables, loading 55
 remote hosts 55
 remote login 116

–remote option 55
 remote shell command, changing 91
 removing breakpoints 166
 remsh command 91
 used in server launches 87
 replacing default arguments 206
 researching directories 72
 reserved message tags 156
 Reset command 226, 227
 Reset command (Visualizer) 190
 resetting command-line arguments 64
 resetting the program counter 249
 resolving ambiguous names 226
 resolving multiple classes 226
 resolving multiple static functions 226
 Restart Checkpoint command 244
 Restart command 243
 restarting
 parallel programs 128
 program execution 204, 243
 restoring focus 258
 restricting output data 177
 results, assigning output to variables 205
 resuming
 executing thread 248
 execution 231, 236
 processes with a signal 242
 returning to original source location 226
 reusing windows 174
 .rhosts file 91
 right angle bracket (>) 174
 right mouse button 166
 RMS applications 120
 attaching to 121
 starting 120
 Root Window 10, 166
 Attached Page 118, 168
 collapsing entries 168
 expanding entries 168
 selecting a process 174
 sorting columns 168
 starting CLI from 201
 state indicator 66
 Unattached page 102
 rounding modes 250
 routine visualization 179
 routines, diving on 174
 routines, selecting 170
 RS_DBG_CLIENTS_PER_SERVER environment variable 144
 rsh command 91, 116
 rules for scoping 332
 Run To command 4, 127, 239
 “run to” commands 241, 255
 running CLI commands 56
 running groups 126
 running operator 275
 running state 67

S

–s command-line option 56, 202
 S share group specifier 264
 S state 67
 S width specifier 265
 sample programs
 make_actions.tcl 202
 sane command argument 202
 Satisfaction group items pulldown 364
 satisfaction set 364
 satisfied barrier 364
 Save All (action points) command 380
 Save All command 380
 Save Pane command 177
 saved action points 57
 saving
 action points 380
 TotalView messages 205
 window contents 177
 saving data, restricting output 177
 –sb option 380
 scope
 determining 289
 scopes
 compiled in 331
 scoping 288, 330
 ambiguous 332
 as a tree 331
 floating 304
 issues 289
 omitting components 332
 rules 332
 Variable Window 285
 variables 287
 scrolling 165
 output 206
 undoing 227
 sctotalview command on SiCortex 148
 sctotalviewcli command on SiCortex 148
 sctv8 command on SiCortex 148
 sctv8cli command on SiCortex 148
 Search Path command 58, 70, 71, 72, 118
 search order 70, 71
 search paths
 default lookup order 70
 for initialization 56
 not passed to other processes 72
 order 70
 setting 70, 152
 –search_port command-line option 86
 searching 224
 case-sensitive 224
 for source code 227
 functions 225
 locating closest match 224
 see also Edit > Find, View > Lookup Function, View > Lookup Variable

source code 225
 wrapping to front or back 224
 searching, variable not found 224
 seeing structures 287
 seeing value changes 285
 limitations 286
 select button 165
 selected line, running to 256
 selecting
 different stack frame 170
 routines 170
 source code, by line 249
 source line 237
 text 176
 selecting a target 228
 selecting process for a group 276
 selection and Expression List window
 304
 sending signals to program 70
 –serial command-line option 94
 serial line
 baud rate 94
 debugging over a 94
 radio button 95
 server launch 83
 command 83
 enabling 83
 replacement character %C 87
 server on each processor 17
 –server option 86
 server_launch_enabled variable 83, 86, 91
 server_launch_string variable 83
 server_launch_timeout variable 84
 service threads 21, 25
 Set Barrier command 363
 set expressions 275
 set indicator, uses dot 259, 276
 Set PC command 249
 Set Signal Handling Mode command 153
 –set_pw command-line option 91
 –set_pw single process server launch command 88
 –set_pws bulk server launch command 89
 setting
 barrier breakpoint 363
 breakpoints 116, 166, 218, 252, 352, 359
 breakpoints while running 352
 evaluation points 166, 367
 groups 268
 options 78
 preferences 78
 search paths 70, 152
 thread specific breakpoints 395
 timeouts 116
 setting focus 257
 setting up, debug session 51

setting up, parallel debug session 97, 131
 setting up, remote debug session 81
 setting X resources 78
 SGI, and bulk server launch 88
 SGROUP variable 269
 shape arrays, deferred types 327
 Share > Halt command 229
 share groups 24, 235, 262
 cell broadband engine 140
 defined 23
 determining 236
 determining members of 236
 discussion 235
 naming 235
 overview 262
 S specifier 264
 SHARE_ACTION_POINT variable 356, 360, 361
 shared libraries 245
 controlling which symbols are read 246
 loading all symbols 247
 loading loader symbols 247
 loading no symbols 247
 preloading 245
 reading excluded information 248
 shared library, exclusion list order 247
 shared library, specifying name in scope 332
 shared memory library code, see SHMEM library code debugging
 shared variables 133
 in OpenMP 134
 OpenMP 134, 137
 procedure for displaying 135
 sharing action points 361
 shell, example of invoking CLI program 202
 SHLIB_PATH environment variable 57
 SHMEM library code debugging 156
 \$short data type 317
 Show full path names check box 227, 357
 showing areas of memory 296
 SiCortex
 installation notes 147
 sctotalview command 148
 sctotalviewcli command 148
 sctv8 command 148
 sctv8cli command 148
 using TotalView 147
 side 383
 side-effects of functions in expression system 383
 SIGALRM 129
 SIGFPE errors (on SGI) 68
 SIGINT signal 106
 signal handling mode 68
 signal/resignal loop 70
 signal_handling_mode variable 68
 signals
 affected by hardware registers 68
 clearing 243
 continuing execution with 242
 discarding 70
 error option 70
 handler routine 68
 handling 68
 handling in PVM applications 153
 handling in TotalView 68
 handling mode 68
 ignore option 70
 resend option 70
 sending continuation signal 242
 SIGALRM 129
 SIGTERM 153
 stop option 70
 stops all related processes 69
 that caused core dump 64
 Signals command 68
 SIGSTOP
 used by TotalView 68
 when detaching 62
 SIGTERM signal 153
 stops process 153
 terminates threads on SGI 133
 SIGTRAP, used by TotalView 68
 single process server launch 81, 83, 87
 single process server launch command
 %D 87
 %L 88
 %P 88
 %R 87
 %verbosity 88, 90
 –callback_option 88
 –n 87
 –set_pw 88
 –working_directory 87
 single-stepping 239, 253
 commands 239
 in a nested stack frame 256
 into function calls 240
 not allowed for a parallel region 133
 on primary thread only 253
 operating system dependencies 241, 242
 over function calls 240
 recursive functions 241
 skipping elements 335
 slash in group specifier 264
 sleeping state 67
 slices 8
 defining 334
 descriptions 336
 examples 334, 335
 lower bound 334
 of arrays 334
 operations using 328

stride elements 334
 UPC 158
 upper bound 334
 with the variable command 336
 SLURM 137
 smart stepping, defined 253
 SMP machines 100
 sockets 94
 Sorted Variable Window figure 342
 sorting
 array data 342
 Root Window columns 168
 Source As > Assembler 171
 Source As > Both 172, 249
 Source As > Both command 249
 Source As > Source 171
 source code
 examining 171
 finding 225, 227
 navigating 227
 Source command 171
 source file, specifying name in scope 332
 source lines
 ambiguous 237
 editing 227
 searching 237
 selecting 237
 Source Pane 169, 171
 source-level breakpoints 352
 space allocation
 dynamic 371
 static 371, 372
 spawned processes 209
 stopping 101
 SPE
 defined 138
 specifier combinations 264
 specifiers
 and dfocus 266
 and prompt changes 266
 example 269
 examples 265, 266, 267
 specifying groups 263
 specifying search directories 72
 splitting up work 17
 SPU
 breakpoints 141
 CLI focus 143
 naming in TotalView 141
 registers 143
 thread share groups 140
 threads 140
 union describing register contents 143
 SPU, defined 138
 stack
 master thread 135
 trace, examining 292
 unwinding 249
 stack context of the OpenMP master thread 135
 stack frame 285
 current 227
 examining 292
 matching 345
 pane 170
 selecting different 170
 Stack Frame Pane 6, 170, 297
 stack parent token 137
 diving 137
 Stack Trace Pane 170, 171, 248
 displaying source 174
 standard deviation array statistic 344
 Standard I/O page 65
 standard I/O, altering 65
 standard input, and launching tvdsvr 92
 Standard Template Library 280
 standard template library, see STL
 start(), stopping within 154
 start_pes() SHMEM command 157
 starting 149
 CLI 53, 201
 groups 236
 parallel tasks 117
 TotalView 4, 53, 63, 116
 tvdsvr 55, 81, 86, 154
 tvdsvr manually 91
 starting a new process 59
 starting LAM/MPI programs 120
 starting MPI programs 98
 starting program under CLI control 203
 Startup command 55
 startup file 56
 Startup Parameters command 73
 state characters 67
 states
 and status 66
 initializing 56
 of processes and threads 66
 process and thread 67
 unattached process 67
 static constructor code 237
 static functions, resolving multiple 226
 static internal counter 368
 static patch space allocation 371, 372
 statically linked, stopping in start() 154
 statistics for arrays 343
 status
 and state 66
 of processes 65
 of threads 65
 status registers
 examining 250
 interpreting 250
 Step command 4, 128, 237, 239
 "step" commands 240
 Step Instruction command 237
 stepping
 see also single-stepping
 apparently hung 128
 at process width 254
 at thread width 255
 goals 254
 into 240
 multiple statements on a line 240
 over 240
 primary thread can fail 255
 process group 254
 processes 128
 Run (to selection) Group command 127
 smart 253
 target program 210
 thread group 254
 threads 273
 using a numeric argument in CLI 240
 workers 273
 stepping a group 254
 stepping a process 254
 stepping commands 237
 stepping processes and threads 12
 STL 280
 list transformation 281
 map transformation 280
 platforms supported 280
 STL preference 281
 STLView 280
 \$stop assembler pseudo op 393
 \$stop built-in function 396
 Stop control group on error check box 70
 Stop control group on error signal option 69
 stop execution 4
 STOP icon 166, 252, 352, 358
 for breakpoints 166, 352
 stop, defined in a multiprocess environment 211
 STOP_ALL variable 356, 360
 \$stopall built-in function 396
 Stopped Execution of Compiled Expressions figure 371
 stopped operator 275
 stopped process 365
 stopped state 67
 unattached process 67
 stopping
 all related processes 69
 groups 126
 processes 229
 spawned processes 101
 threads 229
 \$stopprocess assembler pseudo op 393
 \$stopprocess built-in function 396
 \$stopthread built-in function 396

stride 334
 default value of 335
 elements 334
 in array slices 334
 omitting 335
Stride Displaying the Four Corners of an Array figure 335
\$String data type 317
string assembler pseudo op 394
\$String data type 318
structs
 see also structures
 defined using **typedefs** 315
 how displayed 315
structure information 287
Structured command 294
structures 298, 315
 see also **structs**
 collapsing 287
 editing types 312
 expanding 287
 expression evaluation 382
 viewing across 346
stty sane command 202
subroutines, displaying 174
subset attach command 125
substructure viewing, limitations 287
suffixes of processes in process groups 235
suffixes variables 52
sum array statistic 344
Sun MPI 123
Suppress All command 356, 357
suppressing action points 356
surface
 in dataset window 185
Surface command (Visualizer) 185
surface view 187, 189
 Visualizer 181
surface visualization window 185
surface window, creating 185
suspended windows 389
switch-based communication 115
 for PE 115
symbol lookup 331
 and context 331
symbol name representation 330
symbol reading, deferring 246
symbol scoping, defined 331
symbol specification, omitting components 332
symbol table debugging information 51
symbolic addresses, displaying assembler as 172
Symbolically command 172
symbols
 loading all 247
 loading loader 247
 not loading 247

synchronizing execution 231
 synchronizing processes 211, 255, 256
 synergistic processor unit 138
 syntax 263
 system PID 209
 system TID 209
 system variables, see **CLI variables**
 systid 169, 209
\$systid built-in variable 394
T
T state 67
t width specifier 265
T+/T- buttons 230
tag field 358
 area 171
Talking to Rank control 125
target process/thread set 210, 257
target program
 stepping 210
target, changing 258
tasker event 154
tasks
 attaching to 155
 diving into 155
 PVM 151
 starting 117
Tcl
 and the CLI 14
 CLI and thread lists 200
 version based upon 199
Tcl and CLI relationship 201
TCP/IP address, used when starting 55
TCP/IP sockets 94
temp file prototypes 85
templates
 expression system 386
 lists 280
 maps 280
 STL 280
 vectors 280
terminating processes 204
testing for IEEE values 340
testing when a value changes 374
text
 editing 176
 locating closest match 224
 saving window contents 177
 selecting 176
text assembler pseudo op 394
text editor, default 228
third party visualizer 182
thread
 width specifier, omitting 272
Thread > Continuation Signal command 62, 242
Thread > Go command 237
Thread > Hold command 232
Thread > Set PC command 249
thread as dimension in Visualizer 184

thread focus 257
thread group 256
 stepping 254
thread groups 23, 255, 262
 behavior 268
 behavior at goal 255
thread ID 169, 210
 system 394
TotalView 394
thread IDs, cell broadband engine 141
thread local storage 136
 variables stored in different locations 136
thread numbers are unique 209
Thread Objects command 329
thread objects, displaying 329
Thread of Interest 236
thread of interest 259, 260
 defined 229, 259
thread state 67
thread stepping 273
 platforms where allowed 255
Thread Tab 170
THREADPRIVATE common block, procedure for viewing variables in 136
THREADPRIVATE variables 136
threads
 call graph 179
 changing 230
 changing in Expression List window 308
 changing in Variable window 298
 creating 18
 displaying manager 168
 displaying source 174
 diving on 170, 174
 finding window for 170
 holding 231, 256, 363
 ID format 170
 listing 169, 170
 manager 21
 opening window for 170
 releasing 231, 362, 363
 resuming executing 248
 service 21
 setting breakpoints in 395
 single-stepping 253
 stack trace 170
 state 65
 status of 65
 stepping 12
 stopping 229
 switching between 11
 systid 169
 tid 169
 user 21
 width 255
 width specifier 260
 workers 21, 22

threads model 18
 threads tab 230
 thread-specific breakpoints 395
 tid 169, 210
 \$tid built-in variable 394
 TID missing in arena 260
 timeouts
 avoid unwanted 129
 during initialization 117
 for connection 83
 TotalView setting 116
 timeouts, setting 116
 TOI defined 229
 again 251
 toolbar, using 228
 Tools > Attach Subset command 125
 Tools > Call Graph command 179
 Tools > Command Line command 53, 201
 Tools > Create Checkpoint command 244
 Tools > Evaluate command 192, 193, 245, 302, 389, 394
 Tools > Evaluate command, see Expression List window
 Tools > Evaluate Dialog Box figure 390, 391
 Tools > Evaluate Window
 expression system 384
 Tools > Expression List Window 304
 Tools > Fortran Modules command 325
 Tools > Global Arrays command 150
 Tools > Manage Shared Libraries command 245
 Tools > Message Queue command 109
 Tools > Message Queue Graph command 11, 107
 Tools > Program Browser command 284
 Tools > PVM Tasks command 155
 Tools > Restart Checkpoint command 244
 Tools > Statistics command 343
 Tools > Thread Objects command 329
 Tools > Variable Browser command 291
 Tools > View Across command 160
 Tools > Visualize command 8, 183, 347
 Tools > Visualize Distribution command 159
 Tools > Watchpoint command 10, 375, 378
 Tools > Watchpoint Dialog Box figure 376
 tooltips 283
 evaluation within 283

TotalView
 and MPICH 100
 as PVM tasker 151
 core files 53
 initializing 56
 invoking on UPC 158
 programming 14
 relationship to CLI 200
 starting 4, 53, 63, 116
 starting on remote hosts 55
 starting the CLI within 201
 Visualizer configuration 193

TotalView assembler operators
 hi16 392
 hi32 392
 lo16 392
 lo32 392

TotalView assembler pseudo ops
 \$debug 393
 \$hold 393
 \$holdprocess 393
 \$holdprocessstopall 393
 \$holdstopall 393
 \$holdthread 393
 \$holdthreadstop 393
 \$holdthreadstopall 393
 \$holdthreadstopprocess 393
 \$long_branch 393
 \$stop 393
 \$stopall 393
 \$stopprocess 393
 \$stopthread 393
 align 393
 ascii 393
 asciz 393
 bss 393
 byte 393
 comm 393
 data 393
 def 393
 double 393
 equiv 393
 fill 393
 float 393
 global 393
 half 393
 lcomm 393
 lysm 393
 org 393
 quad 394
 string 394
 text 394
 word 394
 zero 394

totalview command 53, 56, 63, 112, 116, 118, 122
 for HP MPI 114

TotalView data types
 \$address 316
 \$char 316

\$character 316
 \$code 316, 320
 \$complex 316
 \$complex_16 316
 \$complex_8 316
 \$double 316
 \$double_precision 316
 \$extended 317
 \$float 317
 \$int 317
 \$integer 317
 \$integer_1 317
 \$integer_2 317
 \$integer_4 317
 \$integer_8 317
 \$logical 317
 \$logical_1 317
 \$logical_2 317
 \$logical_4 317
 \$logical_8 317
 \$long 317
 \$long_long 317
 \$real 317
 \$real_16 317
 \$real_4 317
 \$real_8 317
 \$short 317
 \$string 317, 318
 \$void 317, 319
 \$wchar 317
 \$wchar_s16 317
 \$wchar_s32 318
 \$wchar_u16 318
 \$wchar_u32 318
 \$wstring 318
 \$wstring_s16 318
 \$wstring_s32 318
 \$wstring_u16 318
 \$wstring_u32 318

TotalView Debugger Server, see tvdsvr
 TOTALVIEW environment variable 55, 101, 129
 totalview subdirectory 57
 totalview subdirectory, see .totalview subdirectory
 TotalView Visualizer
 see Visualizer
 TotalView windows
 action point List tab 171
 editing cursor 176
 totalviewcli command 53, 55, 56, 63, 118, 122, 201, 203
 -remote 55
 trackball mode, Visualizer 181
 tracking changed values 285
 limitations 286
 transformations, creating 281
 transposing axis 187
 TRAP_FPE environment variable on SGI 68

troubleshooting xxix
 MPI 106
 ttf variable 281
`-tv` command-line option 100
 TV:: namespace 207
 TV::GUI:: namespace 207
 TVD.breakpoints file 380
 TVDB_patch_base_address object 372
 tvdb_patch_space.s 373
 tvdrc file, see `.tvdrc` initialization file
 tvdsvr 55, 81, 83, 84, 92, 94, 370
 attaching to 155
 `-callback` command-line option 91
 cleanup by PVM 156
 Cray XT3 144
 editing command line for poe 117
 fails in MPI environment 106
 launch problems 84, 85
 launching 87
 launching, arguments 92
 manually starting 91
 `-port` command-line option 86
 `-search_port` command-line option 86
 `-server` command-line option 86
 `-set_pw` command-line option 91
 starting 86
 starting for serial line 94
 starting manually 86, 91
 symbolic link from PVM directory 152
 with PVM 154
 tvdsvr command 86
 starting 81
 timeout while launching 84, 85
 use with PVM applications 152
 tvdsvr_rs 144
 TVDSVRLAUNCHCMD environment variable 87
 Two Computers Working on One Problem figure 17
 two-dimensional graphs 186
 type casting 311
 examples 320
 type strings
 built-in 315
 editing 311
 for opaque types 320
 supported for Fortran 312
 type transformation variable 281
 type transformations, creating 281
 typedefs
 defining structs 315
 how displayed 315
 types supported for C language 312
 types, user defined type 327

UDT 327
 UDWP, see watchpoints

UID, UNIX 86
 Unattached page 102
 unattached process states 67
 undive 299
 undive all 299
 undive buttons 298
 undive icon 175, 226, 298
 undive/redive buttons 7
 Undive/Redive Buttons figure 299
 undiving, from windows 299
 unexpected messages 109, 111
 unheld operator 275
 union operator 275
 unions 315
 how displayed 315
 unique process numbers 209
 unique thread numbers 209
 unsupressing action points 357
 unwinding the stack 249
 UPC
 assistant library 158
 compilers supported 158
 phase 160
 pointer-to-shared data 160
 shared scalar variables 158
 slicing 158
 starting 158
 viewing shared objects 158
 UPC debugging 158
 Update command 231
 updating visualization displays 183
 upper adjacent array statistic 344
 upper bounds 314
 of array slices 334
 USEd information 326
 user defined data type 327
 user mode 21
 user threads 21
 Using Assembler figure 391
 Using the Attached Page 168

Valid in Scope list 331
 value changes, seeing 285
 limitations 286
 value field 389
 values
 changing 176
 editing 7
 Variable Browser command 291
 variable scope 288
 variable scoping 331
 Variable Window 302
 and expression system 383
 changing threads 298
 closing 298
 displaying 283
 duplicating 300
 expression field 285
 in recursion, manually refocus 285

!View Across display 345
 rebinding 298
 scope 288
 scoping display 285
 stale in pane header 284
 tracking addresses 284
 type field 285
 updates to 284
 view across 346
 variables
 assigning p/t set to 261
 at different addresses 346
 bluegene_io_interface 119
 bluegene_server_launch 119
 CGROUP 262, 268
 changing the value 310
 changing values of 310
 comparing values 289
 display width 282
 displaying all globals 291
 displaying contents 174
 displaying long names 285
 displaying STL 280
 diving 174
 freezing 289
 GROUP 269
 in modules 325
 in Stack Frame Pane 7
 intrinsic, see built-in functions
 locating 224
 not updating display 289
 precision 282
 previewing size and precision 282
 setting command output to 205
 SGROUP 269
 stored in different locations 136
 ttf 281
 View Across display 345
 watching for value changes 10
 WGROUP 268
 variables and expressions 302
 variables, viewing as list 303
`-verbosity` bulk server launch command 89
 verbosity level 122
`-verbosity` single process server launch command 88, 90
 View > Add to Expression List command 303
 View > Assembler > By Address command 172
 View > Assembler > Symbolically command 172
 View > Block Status command 295
 View > Collapse All command 287
 View > Compilation Scope > Fixed command 288
 View > Compilation Scope > Floating command 285, 288

View > Compilation Scope commands
 288
 View > Display Exited Threads figure
 169
 View > Display Managers command
 168
 View > Dive command 310
 View > Dive In All command 300
 View > Dive in New Window command
 7
 View > Dive Thread command 330
 View > Dive Thread New command
 330
 View > Examaine Format > Structured command 294
 View > Examine Format > Raw command 294
 View > Expand All command 287
 View > Freeze command 289
 View > Graph command 185
 View > Graph command (Visualizer)
 185
 View > Lookup Function command
 154, 224, 227
 View > Lookup Variable command
 136, 224, 284, 293, 296, 326
 specifying slices 336
 View > Reset command 226, 227
 View > Reset command (Visualizer)
 190
 View > Source As > Assembler command 171
 View > Source As > Both command
 172, 249
 View > Source As > Source command
 171
 View > Surface command (Visualizer)
 185
 View > View Across > None command 345
 View > View Across > Threads command 136
 View Across
 arrays and structures 346
 view across
 editing data 346
 View Across command. 136
 View Across None command 345
 View simplified STL containers preference 281
 viewing across
 variables 345
 Viewing Across an Array of Structures figure 346
 viewing across processes and threads
 11
 Viewing Across Threads figure 345
 Viewing Across Variable Window 346
 viewing across variables and processes 345

viewing acrosscross
 diving in pane 346
 viewing assembler 172
 viewing existed threads 169
 Viewing manager processes 168
 Viewing manager threads 168
 viewing opaque data 320
 viewing shared UPC objects 158
 viewing templates 280
 viewing variables in lists 303
 viewing wide characters 318
 virtual functions 385
 visualization
 deleting a dataset 185
 \$visualize 396
 visualize 192
 \$visualize built-in function 192
 Visualize command 8, 183, 195, 347
 visualize command 193
 Visualizer 184, 347
 actor mode 181, 191
 auto reduce option 189
 autolaunch options, changing 194
 camera mode 181, 191
 choosing method for displaying data 183
 command summary 181
 configuring 193
 configuring launch 193
 connecting points using lines 187
 creating graph window 185
 creating surface window 185
 data sets to visualize 182
 data types 182
 dataset defined 182
 dataset window 184, 185
 deleting datasets 185
 dimensions 184
 display not automatically updated 183
 displaying points 187
 exiting from 185
 -file command-line option 193, 195
 graphs, display 186
 joy stick mode 181
 joystick mode 191
 launch command, changing shell 195
 launch from command line 193
 launch options 194
 method 183
 number of arrays 182
 obtaining a dataset value 187
 pan 191
 -persist command-line option 193, 195
 pick 181
 picking 191
 rank 194

relationship to TotalView 182
 restricting data 183
 rotate 191
 rotate, Visualizer 181
 scale 191
 shell launch command 194
 slices 182
 surface display option 189
 surface view 181, 187, 189, 191
 third party 182
 trackball mode 181, 191
 using casts 192
 view across data 184
 view window 184
 windows, types of 184
 wireframe mode 181
 wireframe view 191
 XY option 189
 zoom 191

visualizer
 closing connection to 194
 customized command for 194
 visualizing
 data 181, 185
 data sets from a file 193
 from variable window 183
 in expressions using \$visualize 192
 visualizing a dataset 192
 \$void data type 317, 319

W

W state 67
 W width specifier 265
 W workers group specifiers 264
 Waiting for Command to Complete window 128
 Waiting to Complete Message Box figure 390
 warn_step_throw variable 69
 watching memory 377
 Watchpoint command 10, 375, 378
 watchpoint operator 275
 watchpoint state 67
 watchpoints 10, 373
 \$newval watchpoint variable 378
 \$oldval 378
 alignment 379
 conditional 374, 378
 copying data 378
 creating 375
 defined 211, 350
 disabling 376
 enabling 376
 evaluated, not compiled 379
 evaluating an expression 374
 example of triggering when value goes negative 379
 length compared to \$oldval or \$newval 379
 lists of 171

lowest address triggered 377
 modifying a memory location 373
 monitoring adjacent locations 378
 multiple 377
 not saved 380
 on stack variables 375
 PC position 377
 platform differences 374
 problem with stack variables 377
 supported platforms 374
 testing a threshold 374
 testing when a value changes 374
 triggering 373, 377
 watching memory 377
 \$whchar data type 318
 wchar_t wide characters 318
 WGROUP variable 268
 When a job goes parallel or calls exec()
 radio buttons 126
 When a job goes parallel radio buttons
 126
 When Done, Stop radio buttons 364
 When Hit, Stop radio buttons 364
 wide characters 318
 width relationships 260
 width specifier 259
 omitting 272
 wildcards, when naming shared libraries 247
 Window > Duplicate Base Window (Visualizer) 186
 Window > Duplicate command 175, 289, 300
 Window > Memorize All command 176
 Window > Memorize command 176
 Window > Update command 231
 window contents, saving 177
 windows 298
 closing 175, 298
 copying between 177
 dataset 185
 dataset window 185
 dataset window (Visualizer) 186
 graph data 186
 pasting between 177
 popping 174
 resizing 175
 surface view 187
 suspended 389
 Windows > Update command (PVM) 155
 wireframe view, Visualizer 181
 word assembler pseudo op 394
 worker threads 21, 133
 workers group 25, 256
 defined 23
 overview 262
 workers group specifier 264
 working directory 72

working independently 17
 -working_directory bulk server launch
 command 88
 -working_directory single process
 server launch command 87
 writing array data to files 217
 \$wstring data type 318

X

X resources setting 78
 Xdefaults file, see .Xdefaults file
 xterm, launching tvdsvr from 92

Y

yellow highlighted variables 285, 286

Z

Z state 67
 zero assembler pseudo op 394
 zero count array statistic 344
 zombie state 67