Getting Started with TotalView

version 8.6

Copyright © 2007–2008 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 http://www.totalviewtech.com/Products/TotalView/developers.

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

Contents

Discovering TotalView

Getting Started 1
Starting TotalView 2
What About Print Statements? 3
Examining Data 4
Examining Arrays 6
Seeing Groups of Variables 6
Setting Watchpoints 8
Debugging Multi-process and Multi-threaded Programs 8
Program Using Almost Any Execution Model 9
Supporting Multi-process and Multi-threaded Programs 9
Using Groups and Barriers 10
Memory Debugging 11
Introducing the CLI 12
What's Next 12

INDEX 13

Discovering TotalView

chapter 1

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 1
- "*Debugging Multi-process and Multi-threaded Programs*" on page 8
- "*Using Groups and Barriers*" on page 10
- "*Memory Debugging*" on page 11
- "*Introducing the CLI*" on page 12
- "*What's Next*" on page 12

Getting Started

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

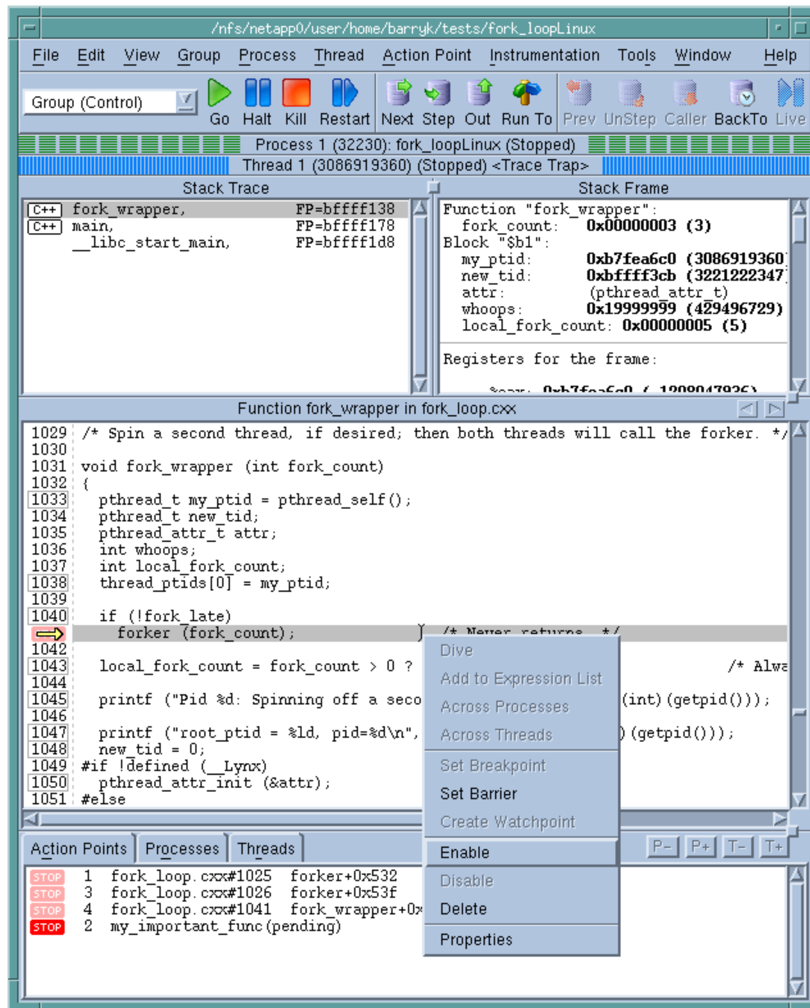
- You use the **–g** option when you compile your program.
- You start your program under TotalView control.
- You set a breakpoint.
- You examine data.

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**.

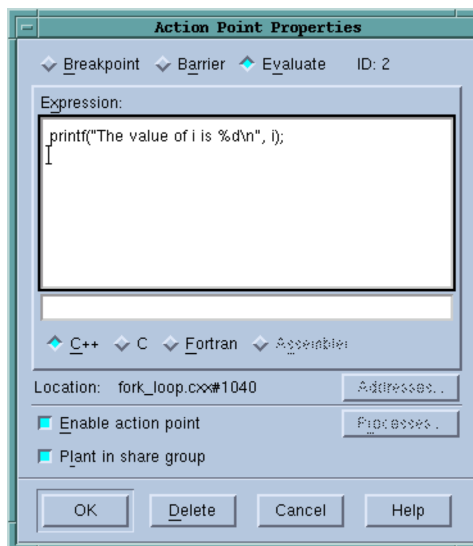
What About Print Statements?

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

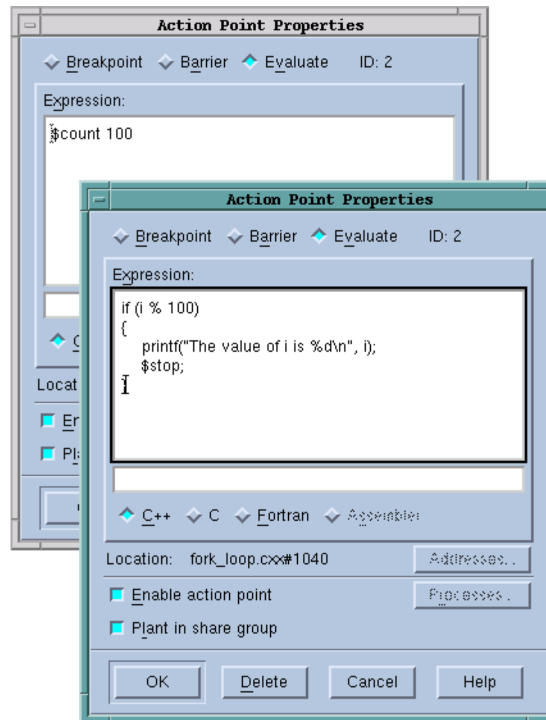


You can add any code you want to a breakpoint. Because there's code associated with this breakpoint, it is called an *eval point*. Here's where TotalView does things a little differently. When your program reaches this eval point, TotalView executes the code you've entered. In this case, TotalView prints 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 on page 4 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.

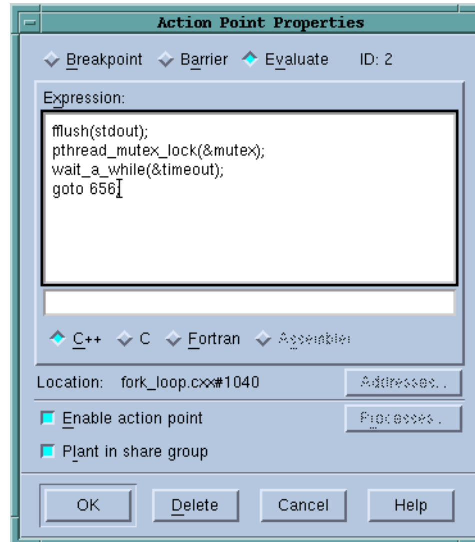
Examining Data

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.

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 2) has a list 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.

Figure 4: Patching Using an Eval Point



***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.*

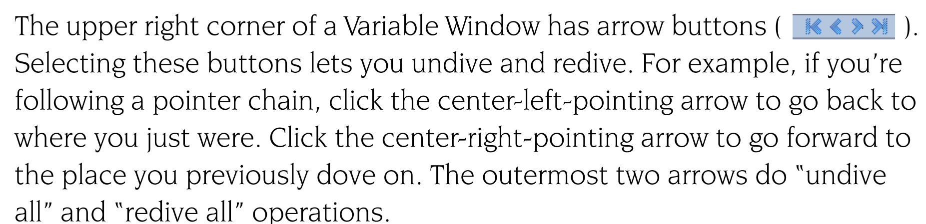
Diving on a variable tells TotalView to display a window that contains information about the variable. (As you read this manual, you'll come across many other types of diving.)

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 6 shows two Variable Windows. One window was created by diving on a structure and the second by diving on an array.

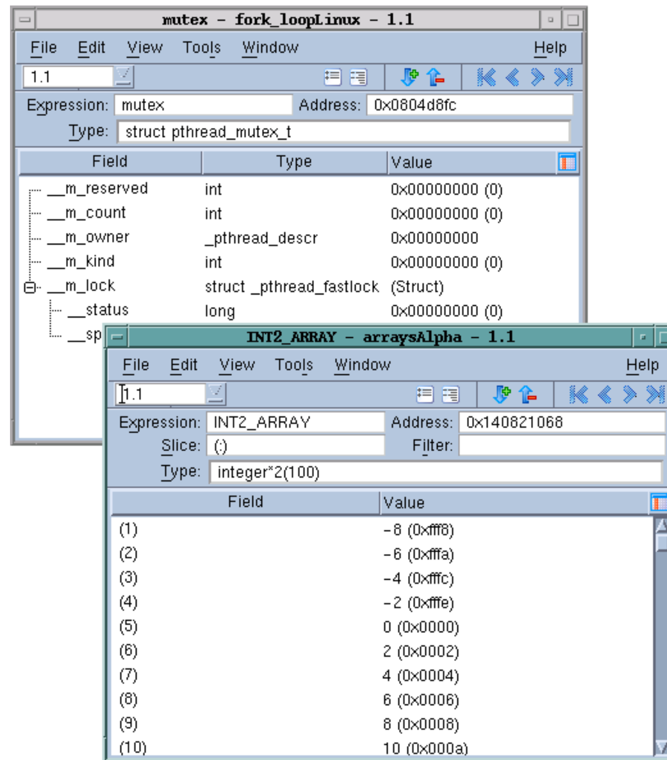
Because the data displayed in a Variable Window might not be simple, you can redive on this data. When you dive in a Variable Window, TotalView replaces the window's contents with the new information. If you don't want to replace the contents, you can use the View > Dive Thread in New Window command 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 the place you previously dove on. The outermost two arrows do "undive all" and "redive all" operations.

Figure 5: Diving on a Structure and an Array



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."

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 7.)

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 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 8.)

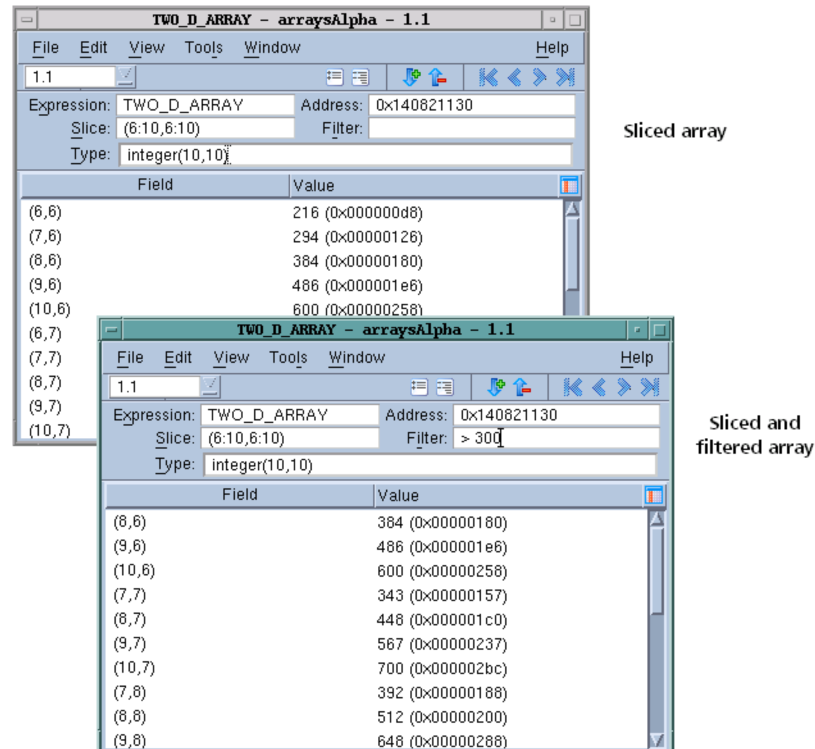


Figure 6: Slicing and Filtering Arrays

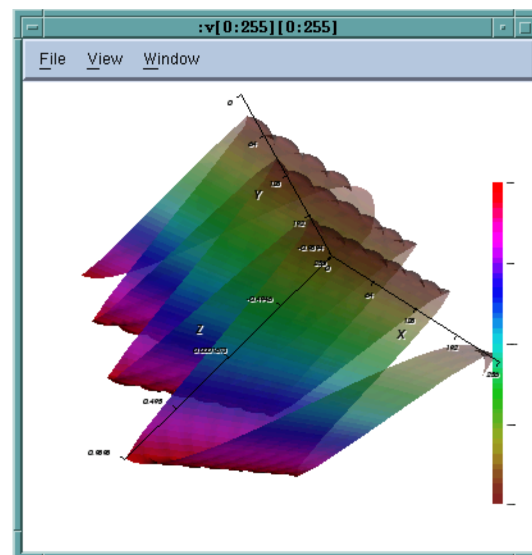
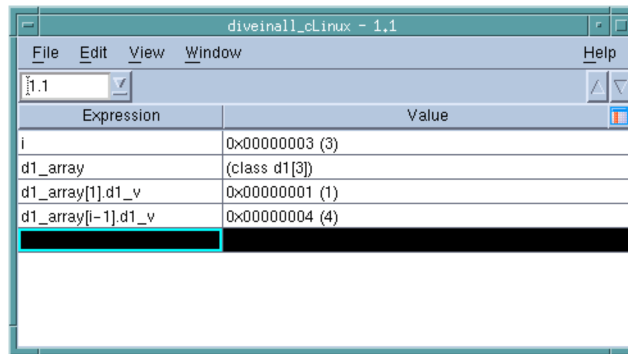


Figure 7: Visualizing an Array

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



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

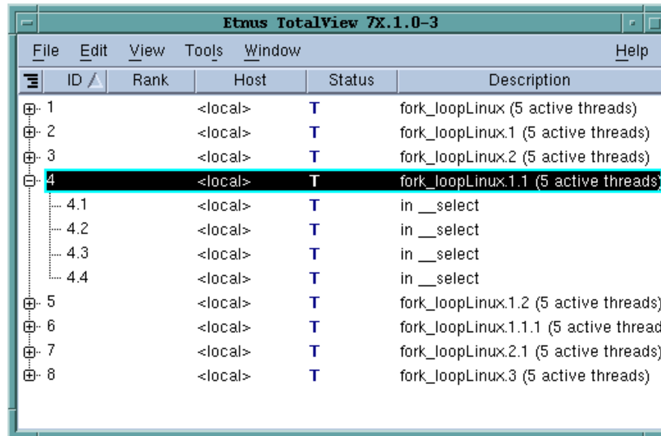
When your program creates processes and threads, TotalView can automatically bring them under its control. If the processes are already running, TotalView can acquire them aw well. You don't need to have multiple debuggers running: one TotalView is all you need.

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 9.)

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

Figure 9: The Root Window



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 10 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 10.)

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.

Figure 10: Viewing Across Processes

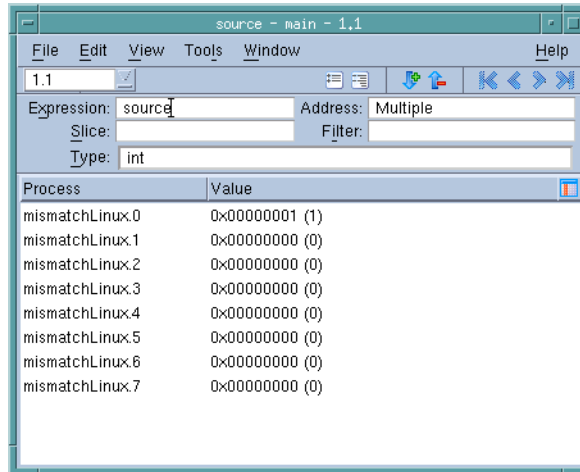
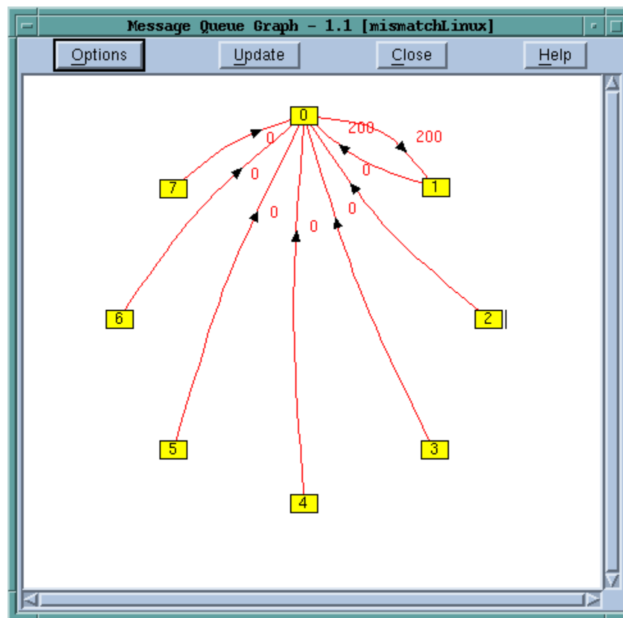


Figure 11: A Message Queue Graph

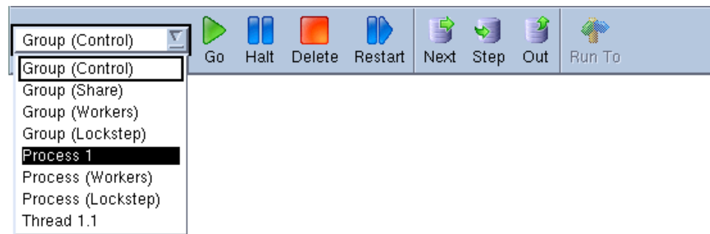


Using Groups and Barriers

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. 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 is a lot easier because it is part of TotalView. It is not a separate program that you run at a different time.

The Memory Debugger is described in its own book. For information, see "*Debugging Memory Problems Using TotalView.*"

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
}
```

Information about the CLI is scattered throughout this book. Chapters 2 and 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.

Index

Symbols

$count built-in function 4
$stop built-in function 4

A

Action Point > At Location command 3
Action Point > Properties command 3
action points
- see also barrier points
- see also eval points
- definition 3
- evaluation points 3
- watchpoint 8

Add to Expression List command 7
arrays
- slicing 6

arrow buttons 5
At Location command 3

B

barrier points 10
bold data 5
built-in functions
- $count 4
- $stop 4

C

CLI
- introduced 12

Command Line Interpreter 12
commands
- Action Point > At Location 3
- Edit > Find 2
- step 2
- Tools > Message Queue Graph 9
- Tools > Visualize 6
- Tools > Watchpoint 8
- View > Dive in New Window 5
- Visualize 6

compiling
- programs 1

context menus
- Add to Expression 7

D

data
- editing 5
- examining 4
- filtering 6
- see also Variable Window
- slicing 6

dereferencing 5
Dive In New Window command 5
diving
- defined 5
- into a variable 5

E

Edit > Find command 2
eval points
- see evaluation points

evaluation points 3
- patching programs 4
- printing from 3
- using $stop 4

examining
- data 4

execution models 9
Expression List window 6

F

filtering 6
Find command 2

G

Go command 2
groups
- and barriers 10

L

lists of variables, seeing 6

M

message queue graph window 9
MPI
- toolbar settings for 11

multi-process debugging 8
mult-ithreaded debugging 8

P

print statements, using 3
printing in an eval point 3
Process Window 2
processes
- stepping 10
- switching between 9

programming TotalView 12
programs
- compiling 1
- patching 4

properties, of action points 3

R

redive/undive buttons 5
Root Window 8
Run To command 2

S

slices 6
Stack Frame Pane 4
starting
- TotalView 2

Step command 2
stepping processes and threads 10
stop execution 2

T

Tcl
- and the CLI 12

threads
- stepping 10
- switching between 9

Tools > Message Queue Graph command 9
Tools > Visualize command 6
Tools > Watchpoint command 8
TotalView
  programming 12
  starting 2

U

undive/redive buttons 5

V

values
  editing 5
variables
  in Stack Frame Pane 4
  watching for value changes 8
View > Dive in New Window command 5
viewing across processes and threads 9
Visualize command 6

W

Watchpoint command 8
watchpoints 8