TotalView
User Guide

September 2003

Version 6.3

Book Overview

Contents

part II - Setting Up

3 Setting Up a Debugging Session

4 Setting Up Remote Debugging Sessions

5 Setting Up Parallel Debugging Sessions

part IV - Using the CLI

8 Seeing the CLI at Work

9 Using the CLI

part V - Debugging

10 Debugging Programs

12 Examining and Changing Data

13 Examining Arrays

14 Setting Action Points

About This Book

This book describes how to use TotalView®, a source- and machine-level debugger for multiprocess, multithreaded programs. It assumes that you are familiar with programming languages, the UNIX operating systems, the X Window System, and the processor architecture of the platform on which you are running TotalView and your program.

You will be reading a user guide that combines information for two TotalView debuggers. One uses Motif to present windows and dialog boxes. The other runs in an xterm-like window and requires that you type commands. This book emphasizes the Motif interface, as it is easier to use. As you will see, once you "see" what you can do using Motif, you will know what can be done using the command interface.

This book covers using TotalView on all supported platform.

How to Use This Book

The information in this book is presented in five parts.

- **I: Introduction**

  Here you'll find an overview of some of TotalView's features and an introduction to TotalView's process/thread model. Please read this information. It's easy reading and you'll get a feel for what TotalView can do.

- **II: Setting Up**

  Most people don't spend a lot of time in this section. Chapter 3 tells you what you need to know about configuring TotalView. Chapters 4 and 5 tell you how to get your programs running under TotalView's control. Look at Chapter 4 if you're having problems getting the TotalView Debugger Server (tvdsvr) running and if you're reconfiguring how the tvdsvr gets launched.

  You will never need to read all of Chapter 5. Instead, go to the table of contents and find the section that has the information you need.

- **III: Using the GUI**

  The chapters in this section look at some of TotalView's windows and how you use them. You are also shown tools such as the Visualizer and the Call Tree that help you analyze what your program is doing.
- **IV: Using the CLI**

  The chapters in this section explain the conventions of using a command-line debugger and how to create Tcl macros.
- **V: Debugging**

  In many ways, most of what has preceded this part of the book is introductory material. This part of the book is where you'll find out how to examine your program and its data. Here is where you'll find information on setting the action points that allow you to stop and monitor your program's execution.

  Equally important, Chapter 11 is a detailed examination of TotalView's group, process, and thread model. The more you understand this model, the easier time you'll have debugging multiprocess and multithreaded programs.

Using the CLI

To use the CLI (Command Line Interface), you need to be familiar with and have experience debugging programs with the TotalView GUI. As CLI commands are embedded within a Tcl interpreter, you will get better results if you are familiar with Tcl. However, if you don't know Tcl, you will still be able to use the CLI, but you will lose the programmability features that Tcl gives. For example, CLI commands operate upon a set of processes and threads. You can save this set and apply it to commands based upon what you have saved.

You can obtain information on using Tcl at many bookstores, and you can also order these books from online bookstores. Two excellent books are

- 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, 1997.

There is also a rich set of resources available on the Web. The best starting point is www.tcltk.com.

The fastest way to gain an appreciation of the actions performed by CLI commands is to review 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, client/server interactions, and the like. So, this book won't try to tell you about what you're doing. Instead, it tells you about TotalView.

As you will see, 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. We also know you don't want to spend a lot of time reading about using TotalView. Consequently, you're not going to see a lot of quasi-procedural discussions that tell you what to put in dialog boxes. You already know what to do.

This book also doesn't spend a lot of time explaining what you do with a dialog box or the kinds of data you can type. If you want that information, you'll find it in the online Help. If you prefer, an HTML version 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.
arguments	In a command description, text in italic represent information you type. Elsewhere, italic is used for emphasis. You won't have any problems distinguishing between the uses.
Dark text	In a command description, **dark text** represent 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. Bolding this kind of text is done only when it's important. You'll usually be able to differentiate what you type from what the system prints.
	This graphic symbol indicates that the information that follows—*it is printed in italics*—is a note. This information is an important qualifier to what was just said.

Convention	Meaning
	This graphic symbol indicates that a feature is only available in the GUI. If you see it on the first line of a section, all the information in the section is just for GUI users. When it is next to a paragraph, it tells you that just the sentence or two being discussed applies to the GUI.
CLI:	The primary emphasis of this book is on the GUI. It shows the windows and dialog boxes that you use. This symbol tells you how to do the same thing using the CLI.

TotalView Documentation

The following table describes other TotalView documentation:

Title	Contents	Online Help	HTML	PDF	Print
TotalView User Guide	Describes how to use the TotalView GUI and the CLI; this is the most used of all the TotalView books	✓	✓	✓	✓
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				✓
TotalView Commands	Defines all TotalView GUI commands	✓	✓	✓	
Creating Type Transformations	Tells how to create Tcl CLI macros that change the way structures and STL containers appear		✓	✓	
TotalView Installation Guide	Contains the procedures to install TotalView and the FLEXlm license manager		✓	✓	
TotalView New Features	Tells you about new features added to TotalView	✓	✓	✓	
TotalView Release Notes	Lists known bugs and other information related to the current release		✓	✓	
Platforms and System Requirements	Lists the platforms upon which TotalView runs and the compilers it supports		✓	✓	

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 E-Mail address for support issues is:

support@etnus.com

For documentation issues, the address is:

documentation@etnus.com

Here are our phone numbers:

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.

Part I: Introduction

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

Chapter 1: Discovering 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: Understanding Threads, Processes, and Groups
: Defines TotalView's model for organizing processes and threads. While most programmers have an intuitive understanding of what their programs are doing, debugging multiprocess and multithreaded programs requires an exact knowledge of what's being done. This chapter begins a two-part look at TotalView's process/thread model. This chapter contains introductory information. Chapter 11: "*Using Groups, Processes, and Threads*" on page 197 contains information on using these concepts with TotalView commands.

Discovering TotalView 1

The Etnus TotalView® debugger is a powerful, sophisticated, and programmable tool that allows you to debug, analyze, and tune the performance of complex serial, multiprocessor, and multithreaded programs.

If you want to jump in and get started quickly, you should go to our Website at http://www.etnus.com and select TotalView's "Getting Started" area.

Topics in this chapter are:

First Steps

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

- You use the –g option when compiling your program.
- You start your program under the debugger's control.
- You set breakpoints.
- You examine data.

And, the way you go about doing these things is just about the same. 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 multiprocess, multithreaded programs.

Starting TotalView

After execution begins—you'll probably have typed something like totalview *programname*—you'll see a five-paned window. (See Figure 1.)

Figure 1: The Process Window

You can start program execution within TotalView in several ways. Perhaps the easiest is to click on the Step icon in the toolbar. This gets your program started, which means all the initialization stuff performed by the program gets done but no statements are executed. Alternatively, you could scroll your program to find where you want it to run to, select the line, then click on Run To in the toolbar. Or you could click on the line number, which tells TotalView to create a breakpoint on that line, and then click Go in the toolbar.

If your program is large, and usually it will be, you can use the Edit > Find command to locate the line for you. Or, if you want to stop execution when your program reaches a subroutine, use the Action Point > At Location command to set a breakpoint before clicking on Go.

As you can see, you've got lots of choices. Unlike other debuggers, TotalView gives you choices that allow you to debug your program in whatever way you want to debug it.

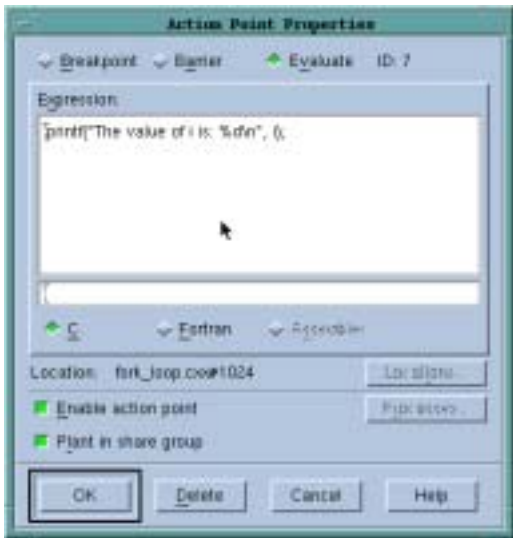
What About Print Statements?

Most programmers learned to debug by using print statements. That is, you inserted lots of printf() or PRINT statements in your code and then inspected what gets written. There's a problem with this. Every time you want to add a new statement, you've got to recompile your program. What's worse is that in a multiprocess, multithreaded program, what gets printed is probably not in the right order. While TotalView is much more sophisticated than this about showing your data (as you'll soon see), you can still use printf() statements if that's your style.

In TotalView, breakpoints are called "action points". This is because they can be much more powerful than the breakpoints you've used in other debuggers.

So, if you don't want to change the way you've been debugging, you can add a breakpoint that prints information for you. Figure 2 shows the Action Point Properties Dialog Box. The easiest way to display this dialog box is to right-click on a line and then select Properties in the context menu. This menu was shown within Figure 1 on page 4.

Figure 2: Action Point Properties Dialog Box

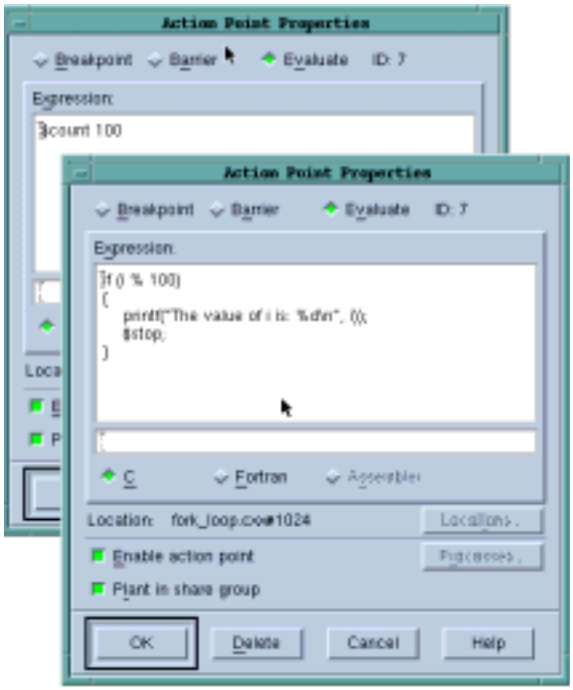


You can add any code you want to a breakpoint. Because there's code associated with this breakpoint, it is now called an "eval point." Here's where TotalView does things a little differently. When your program reaches this eval point, TotalView executes the code you've entered. In this case, TotalView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

Eval points do exactly what you tell them to do. In this case, because you didn't tell TotalView to stop executing, it keeps on going. In other words, you don't have to stop program execution just to see data. You could, of course, have told TotalView to stop. Figure 3 on page 6 shows two evaluation points that stop execution. (One of them does something else as well.)

The one in the foreground uses a programming language statements and a built-in TotalView function to stop a loop every 100 iterations. It also prints

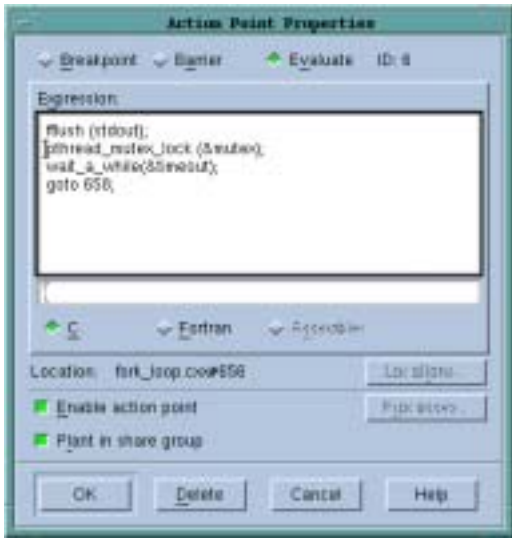
Figure 3: More Conditions



what the value of i is. In contrast, the one in the background just stops the program every 100 times a statement gets executed.

Eval points even allow you to 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 evaluation point shown in Figure 4 tells TotalView to execute three statements and then skip to line 658.

Figure 4: Patching Using an Evaluation Point



Examining Data

Programmers use print statements as an easy way to examine data. They usually do this because the debugger doesn't have sophisticated ways of showing data. In contrast, Chapter 12, "*Examining and Changing Data*," on page 227 and Chapter 13, "*Examining Arrays*," on page 259 explain how you can display data values with TotalView. In addition, Chapter 7, "*Visualizing Programs and Data*," on page 129 describes how to visualize your data in a graphical way.

Because data is difficult to see, the Stack Frame Pane (the pane in the upper right corner of the Process Window, which was shown in Figure 1 on page 4) has a list of all variables that exist in your current routine. If the value is simple, you can see its value in this pane.

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

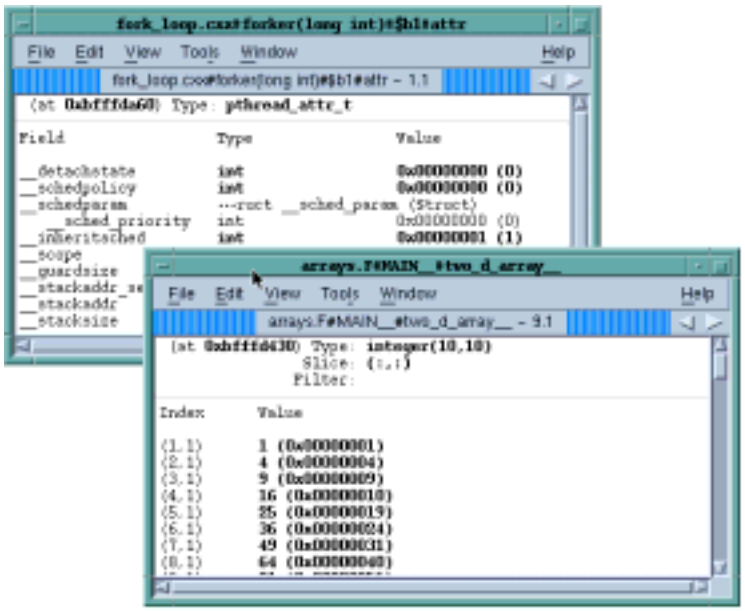
"Diving" is something you can do almost everywhere in TotalView. What happens depends on where you are. To dive on something, position the cursor over the item and click your middle mouse button. If you have a two-button mouse, double-click your left mouse button.

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

Notice that 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 shows two Variable Windows. One was created by diving on a structure and the other by diving on an array.

Figure 5: Two Variable Windows



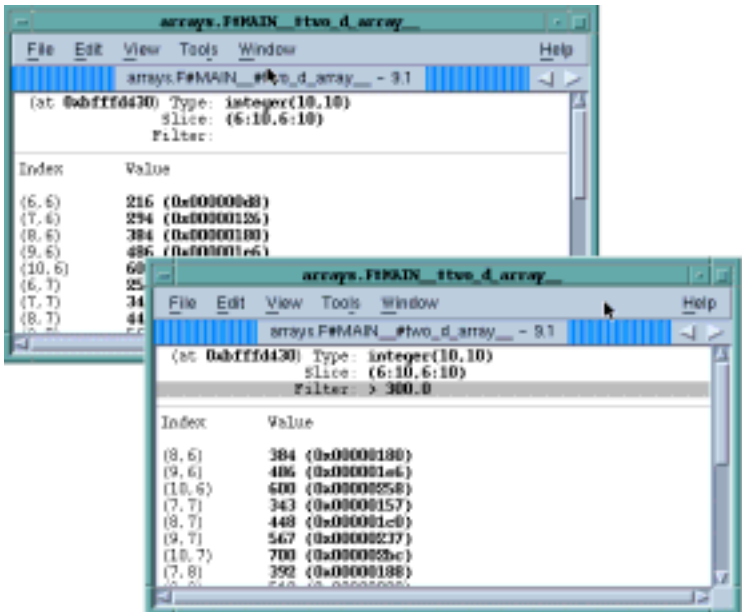
Because the data displayed in a Variable Window may not be simple, you can *also* dive on data in the Variable Window. When you dive in a Variable

Window, TotalView replaces the window's contents with the new information. If this isn't what you want, you can use the View > Dive Anew 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. Notice the forward- and backward-facing arrows in the upper right corner of the Variable Windows. Selecting them lets you "undive" and "redive." For example, if you're following a pointer chain, clicking the left-pointing arrow takes you back to where you just were. Clicking the right-pointing arrow takes you "forward" to the place you previously dove on.

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

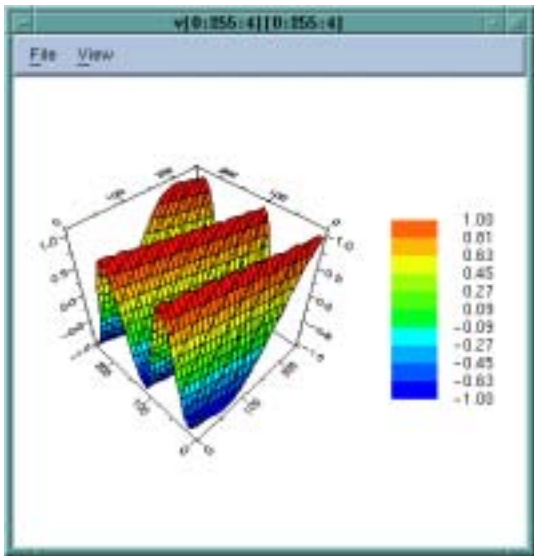
Figure 6: Two More Variable Windows



The Variable Window in the upper left corner of Figure 6 shows a basic *slice* operation. This operation tells TotalView that it should only 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 that it should only 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 will display, there are many times when you 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. Figure 7 on page 9 has an example.

Figure 7: Array Visualization



There's yet another way to look at data. TotalView's watchpoints let you see when a variable's data changes. This works in a different way than other action points. A watchpoint stops execution whenever a data value 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. A better example is that something is trashing a memory location. So, you put a watchpoint on that location and then wait until TotalView stops execution because the watchpoint was executed.

To create a watchpoint, select the Tools > Watchpoint command from any Variable Window.

Debugging Multiprocess and Multithreaded Programs

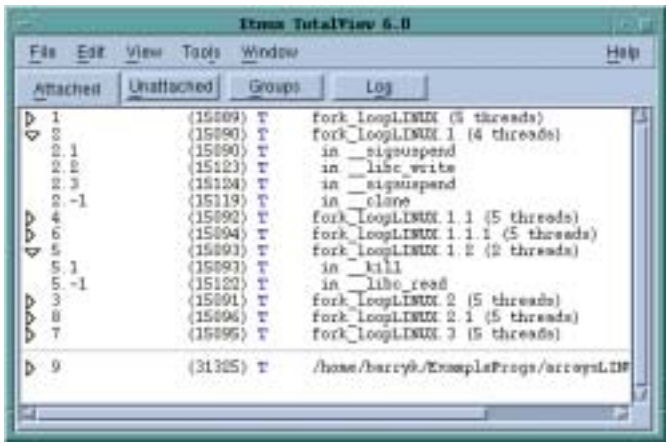
When your program creates processes and threads, TotalView can automatically bring them under its control. If the processes are already running, they too can be acquired. You don't need to have multiple debuggers running. TotalView is enough.

The processes that your program creates can be local or remote. Both are presented to you in the same way. The only difference between debugging a single-process program and a multiprocess, multithreaded program is that you gain the ability to display these additional threads and processes in Process Windows. You can display them in the current Process Window or display them in another window. As always, there are several ways to do it.

TotalView's Root Window (see Figure 8 on page 10), which is automatically displayed after you start TotalView, contains an overview of all processes and threads being debugged. Diving on a process or a thread listed in the

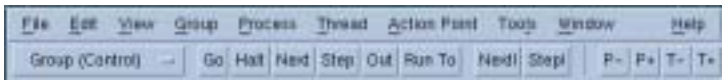
Root Window takes you quickly to the information you want to see. If you need to debug processes that are already running, the Unattached Page lets you dive on other processes you own. After diving on them, you can debug them in the same way as any other process or thread.

Figure 8: The Root Window



In the Process Window, you can switch between processes and threads by clicking the process and thread switching buttons in the toolbar. These are the four buttons on the right side of the toolbar shown in Figure 9.

Figure 9: Process and Thread Switching Icons



Every time you click on one of these buttons, TotalView switches contexts. The switching order is the order in which you see things in the Root Window.

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 your programs can be compiled using products provided by your hardware vendor or third-party compilers such as those from Intel and the Free Software Foundation (the GNU compilers).

Supporting Multiprocess and Multithreaded Programs

When debugging multiprocess, multithreaded programs, you'll often want to see the value of a variable in each process or thread simultaneously. TotalView's laminated data view does this for you. Figure 10 on page 11 shows an example of what you'll see after you laminate a multithreaded program.

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

Figure 10: A Laminated Variable Window

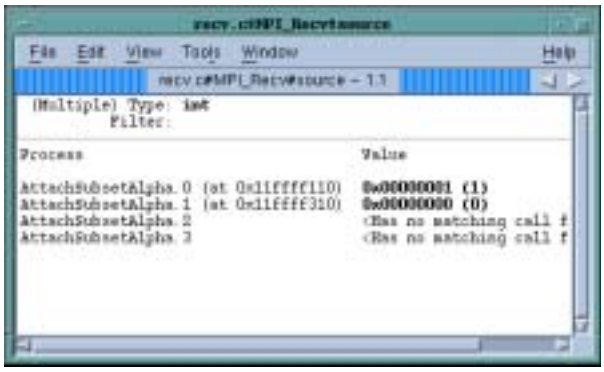
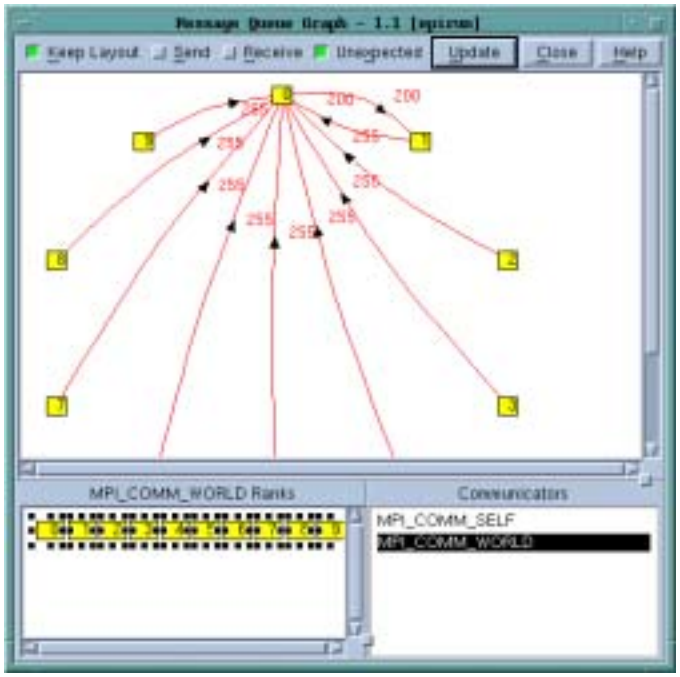


Figure 11: A Message Queue Graph



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.

As you go through this book, you'll find many more examples.

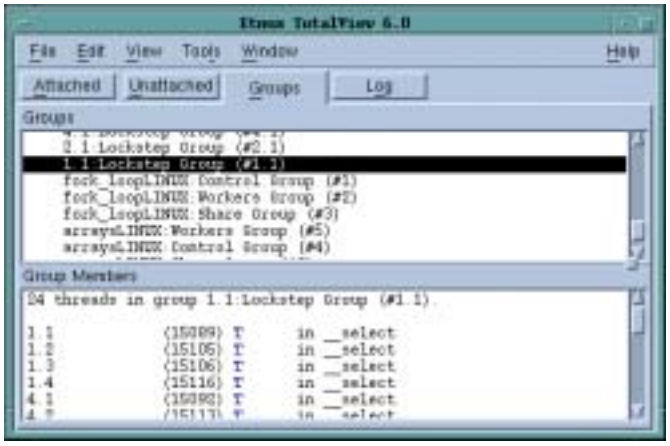
Using Groups and Barriers

When running a multiprocess and multithreaded 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'll be doing the same kinds of operations on the same kinds of things. The two pulldown menus on the toolbar let you indicate what the target of your action will be. See Figure 12 on page 12.

Figure 12: Toolbar with Pulldown

For example, if you are debugging an MPI program, you'd probably set the pulldown to Process (Workers). The reasons for setting them like this are in Chapter 11. Chapter 2 contains definitions for the groups manipulated by TotalView. The Groups Page of the Root Window (Figure 13) shows you the processes and threads that are in a group.

Figure 13: The Root Window's Group Page



Introducing the CLI

The CLI, the TotalView Command Line Interface, 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 Etnus has added to Tcl that allow you to debug your program. Because these debugging commands are native to TotalView's 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, here's how you'd 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, here's how to set a group of breakpoints.

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

While this examination doesn't really save you anything, Chapter 8 presents some examples that are more realistic.

You'll find information about the CLI scattered throughout this book. CLI Commands are described in Chapter 2 of the *TotalView Reference Guide*.

What's Next

This chapter has presented just a few of TotalView's highlights. The rest of this book tells you more about all of TotalView features, both the ones mentioned here and those not yet discussed.

All TotalView documentation is available on our Web site at http://www.etnus.com/Support/docs in PDF and HTML formats. In addition, this information is also contained within TotalView's online Help.

Understanding Threads, Processes, and Groups

2

While the specifics of how multiprocess, multithreaded 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 processes and threads.

This chapter is presents the concepts of thread, process, and group. Chapter 11, "*Using Groups, Processes, and Threads*," on page 197 is a more exacting and comprehensive look at these topics.

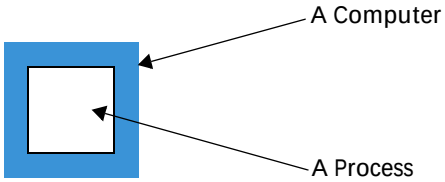
Topics in this chapter are:

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 kind of programs are rather simple, looking something like what's shown in Figure 14 on page 16.

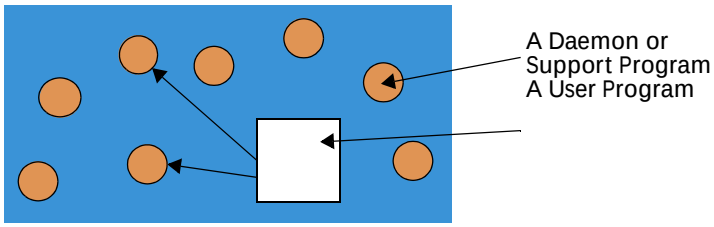
If you use any debugger on these kinds of programs, you can almost always figure out what's going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then inspect variables to see their values. If you suspect there's a logic problem, you can step the program through its statements, seeing what happens and where things are going wrong.

Figure 14: A Uniprocessor



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. (See Figure 15.)

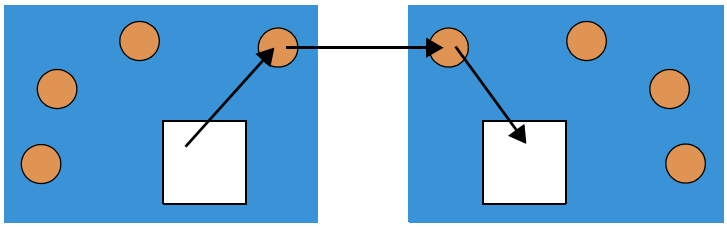
Figure 15: A Program and Daemons



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

Figure 15 assumes that the application program just sends requests to a daemon. This architecture is very simple. More typical is the kind of architecture shown in Figure 16. Here, an E-mail program is communicating with a daemon on one computer. After receiving a request, this daemon sends data to an E-mail daemon on another computer, which then delivers the data to another mail program.

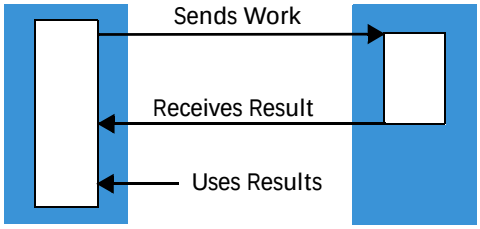
Figure 16: Mail Using Daemons



This architecture assumes that the jobs are disconnected and that they do not need to cooperate. This model has one program handing off work to another. After the handoff, the programs do not interact. While this is a useful model for many kinds of computation, a more general model allows a program to divide its work into smaller jobs, and parcel them out to other computers. This model relies on programs on other machines to 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. (See Figure 17.)

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 multiprocessing 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.

Here is one of the problems with this model: how does a programmer debug 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 is better. It places a server on all remote processor as they are launched. These servers then communicate with the "main" TotalView. 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've got it working, you can split it up so 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 machine means that it will take weeks to execute.

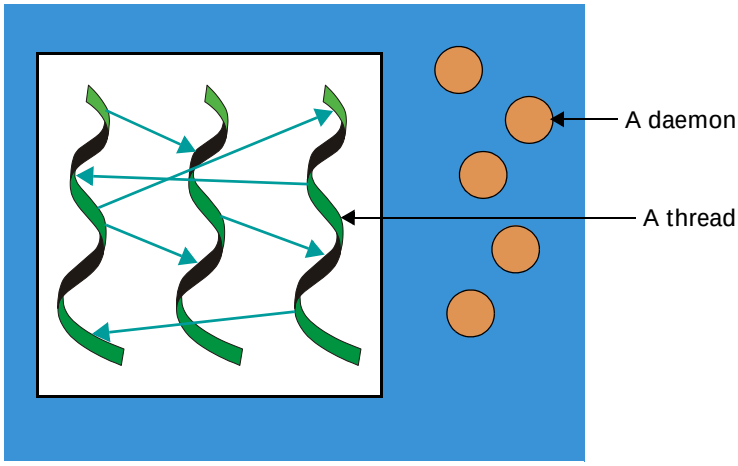
Threads

The daemon programs discussed in the previous section are owned by the operating system. They 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, it can and there are various ways to do this. One programming model splits the work off into somewhat indepen-

dent tasks within the same process. This is the *threads* model. (See Figure 18.) This figure also shows, for the last time, the daemon processes that are executing. From now on, just assume that they are there.

Figure 18: Threads



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 problem here is similar to the problem of processes running on different machines. In both, a debugger must intervene with more than one executing entity.

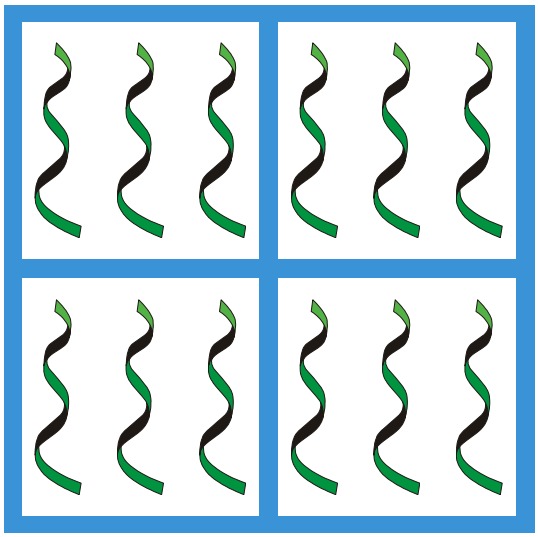
There's not a lot of difference between a multithreaded or a multiprocess programs when you are using TotalView. Except for operating system support, the way in which TotalView displays process information is very similar to how it displays thread information.

Complicated Programming Models

While most computers being sold today have one processor, high-performance computing uses computers that have more than one processor. 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 Figure 18 changes to look something like what's shown in Figure 19 on page 19.

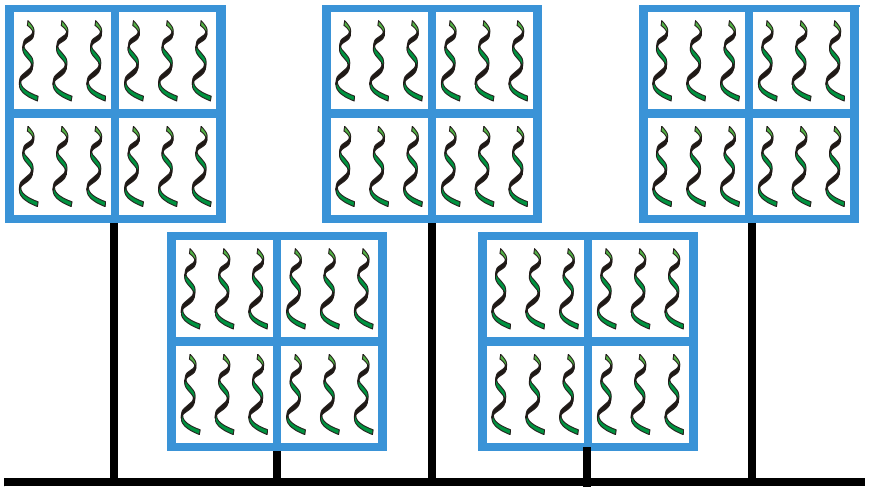
This figure shows four linked processors in one computer, each of which has three threads. This architecture is an extension to the model that links more than one computer together. Its advantage is that the processor doesn't need to communicate with other processors over a network as it is completely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

Figure 19: Four Processor Computer



The next step, of course, is to join many multiprocessor computers together. Figure 20 shows five computers, each having four processors with each processsor running three threads. If this figure is showing the execution of one program, then the program is using 60 threads.

Figure 20: Four-Processor Computer Networks



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

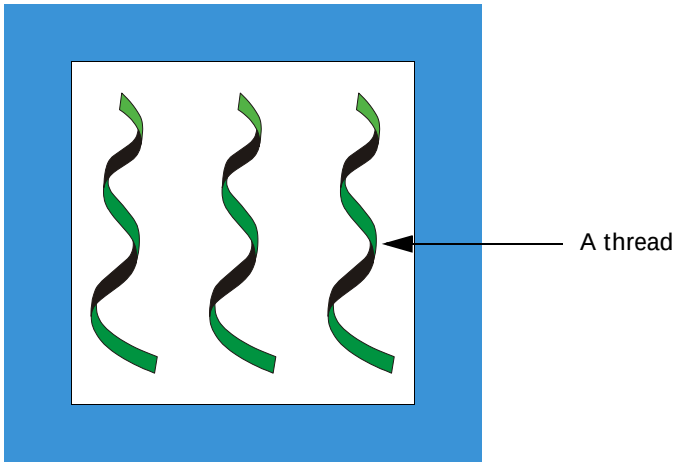
At any time, it is next to impossible to guess which threads are executing and what a thread is actually doing. To make matters worse, many multiprocessor programs begin by invoking a process such as mpirun or IBM's poe whose function is to distribute and control the work being performed. In this kind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

When there are problems in this scenario—and there are always problems—traditional debuggers and solutions are helpless. As you will see, 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.

Kinds of Threads

All threads aren't the same. Figure 21 shows a program with three threads.

Figure 21: Threads



For the moment, 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. Without trying to be rigorous, the TotalView definition of a "user thread" is simply a unit of execution created by a program.

Because they are created by your program to do the work of your program, they are called *worker threads*.

Other threads can also be executing within the process. For example, the threads that are part of the operating environment are manager threads. A *manager thread* is a thread that your environment or operating system adds to your program to help it get work done. In Figure 22 on page 21, the horizontal threads at the bottom are user-created manager threads.

Things would be nice and easy if this was all there was to it. Unfortunately, all threads are not created equal and all threads do not execute equally. In most cases, a program also creates manager-like threads. As these user-created manager threads are designed to perform services for other threads, they can also be called *service threads*. (See Figure 23 on page 21.)

Figure 22: User Threads and Service Threads

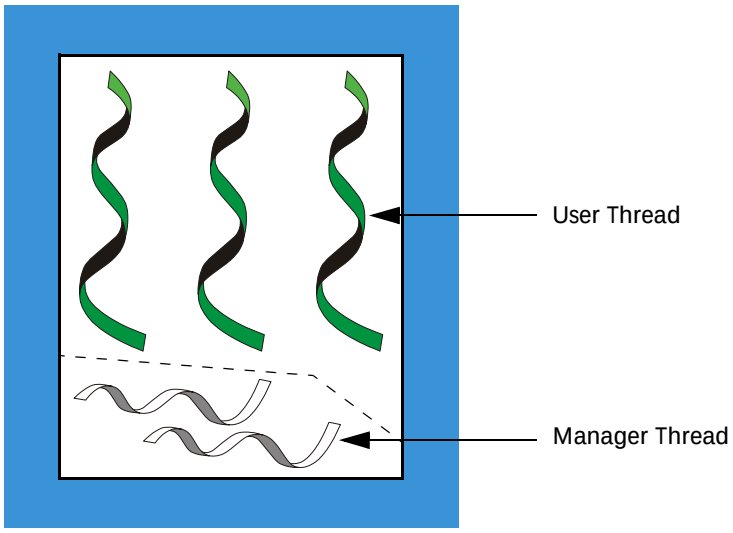
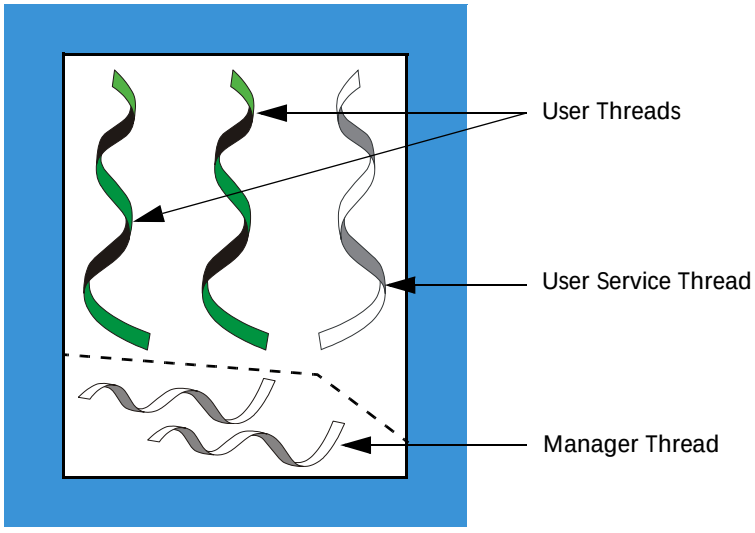


Figure 23: User, Service, and Manager Threads



These service threads are, of course, also worker threads. They are called different things just to keep the different kinds of things that they do separate. As an example, this could be a thread whose sole function is 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 kinds of activities from 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. The code that sends messages between processes is far different than the code that performs fast Fourier transforms. For example, a service thread that queues and dispatches messages sent from other threads may have bugs, but the bugs are different than the rest of your code and you can deal with them separately from the bugs that occur in non-service user threads.

In contrast, your user threads are the agents performing the program's work, and their interactions are where the action is. Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that are actively participating in an activity, rather than on threads in the background performing subordinate activities.

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

Organizing Chaos

While it is possible to debug programs that are running thousands of processes and threads across hundreds of computers by individually looking at each, this is clearly 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 multiprocess, multithreaded 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.

While TotalView cannot know your program's architecture, 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 has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by fork() are in the same control group.
- **Share Group**: All the processes within a control group that share the same code. In most cases, your program will have 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.

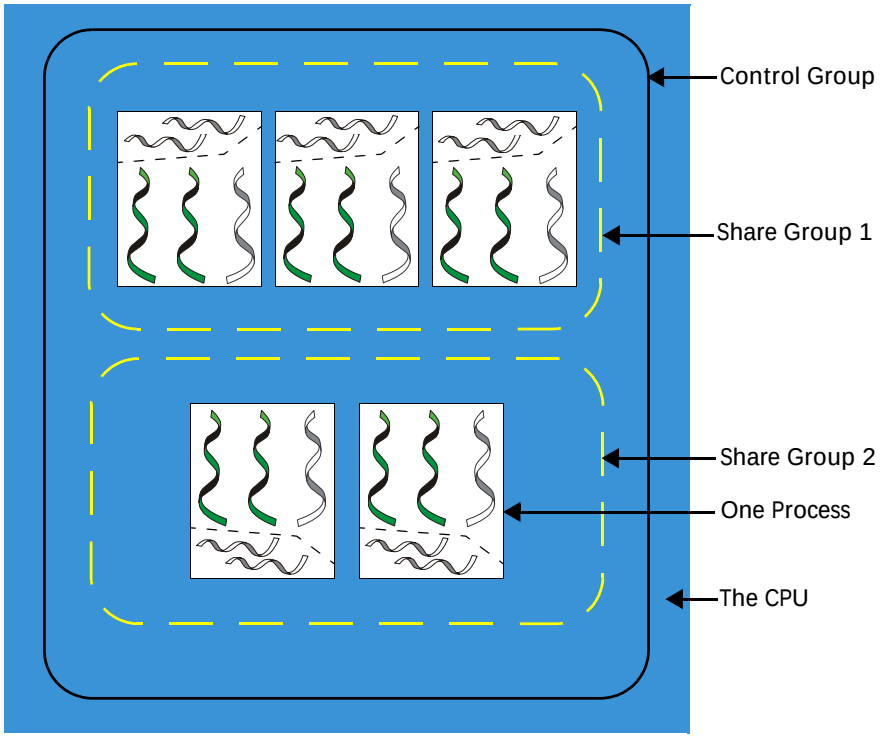
The first two groups in the above list only contain processes, and the last two only contain threads. Notice that "same code" means that the processes have the same executable file name and path.

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).

Not all operating systems let you individually manipulate threads. If TotalView cannot manipulate your program's threads, it will dim the commands within its menu and toolbar.

Figure 24 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 this control group.

Figure 24: Five Processes and Their Groups (Part 1)

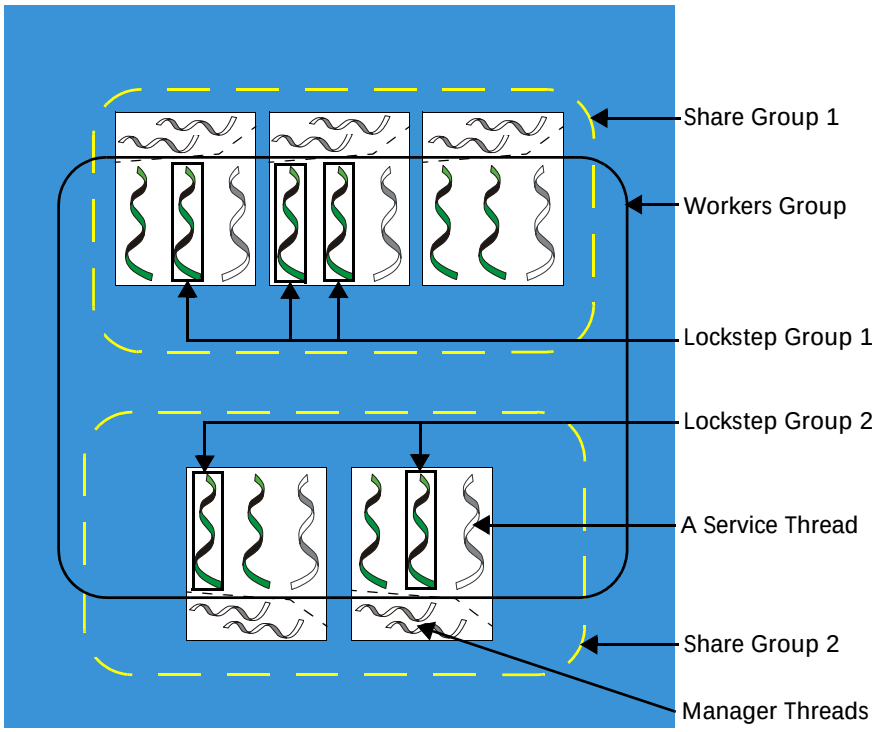


The elements in this figure are as follows:

CPU	The one outer square. 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. This drawing 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. In contrast, the workers group and the lockstep group only contain threads. Figure 25 show how TotalView organizes the threads in Figure 24. As you can see, this figure adds the workers group and two lockstep groups.

Figure 25: Five Processes and Their Groups (Part 2)



The control group is not shown as it encompasses everything in Figure 25. That is, this example's control group contains all of the program's lockstep, share, and worker group's processes and threads.

The elements in this figure are as follows:

Workers Group	All nonmanager threads within the control group make up the workers group. Notice that this group includes service threads.
Lockstep Group	Each share group has its own lockstep groups. Figure 25 shows two lockstep groups, one in each share group. If other threads are stopped, this picture indicates that they are not participating in either of these two lock-step groups. Recall that a stopped thread is always in a lockstep group. (It's OK if a lockstep group has only one member.)
Service Threads	Each process has one service thread. While a process can have any number of service threads, this figure only shows one.

Manager Threads
: The only threads that are not participating in the workers group are the ten manager threads.

Figure 26 extends Figure 25 to show the same kinds of information executing on two processors.

Figure 26: Five Processes and Their Groups on Two Computers

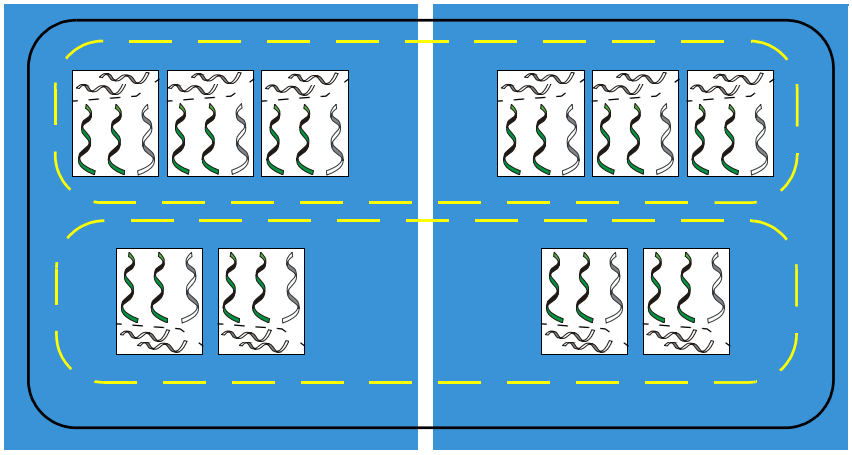


Figure 26 differs from Figure 25 in that it has ten processes executing within two processors rather than five processes within one processor. Although the number of processors has changed, the number of control and share groups is unchanged. This is not to say that the number of groups could not be different. It's just that they do not differ in this example.

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. While there are many ways in which this kind of organization can be built, the following steps indicate the beginning of how this might occur.

Step 1 TotalView and your program are launched and your program begins executing. (See Figure 27 on page 26.)

- **Control group**: A group is created as the program is loaded.
- **Share group**: A group is created as the program begins executing.
- **Workers group**: The thread in the main() routine is the workers group.
- **Lockstep group**: There is no lockstep group because the thread is running.

Step 2 The program forks a process. (See Figure 28 on page 26.)

- **Control group**: A second process is added to the existing group.
- **Share group**: A second process is added to the existing group.

Figure 27: Step 1: A Program Starts

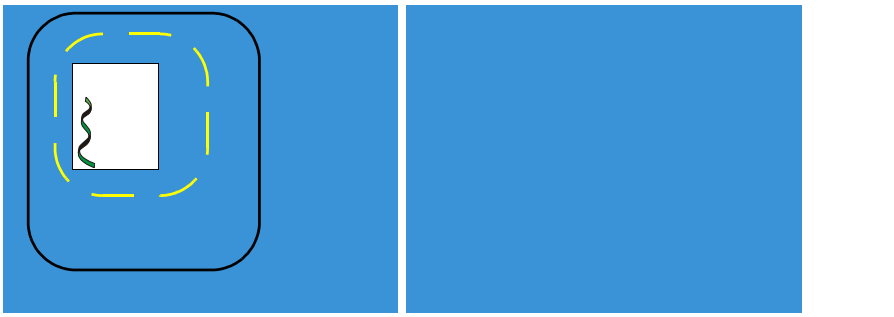
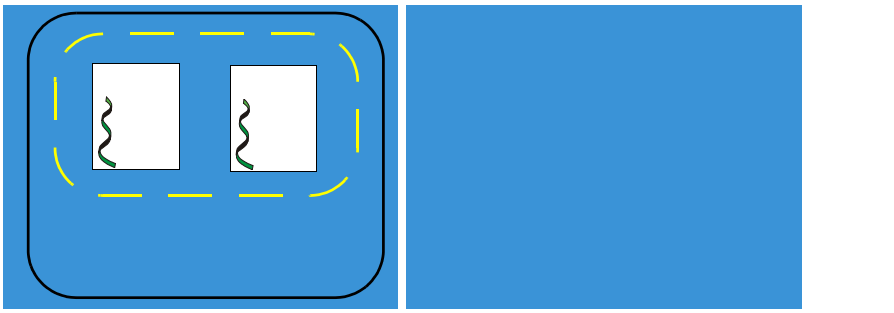


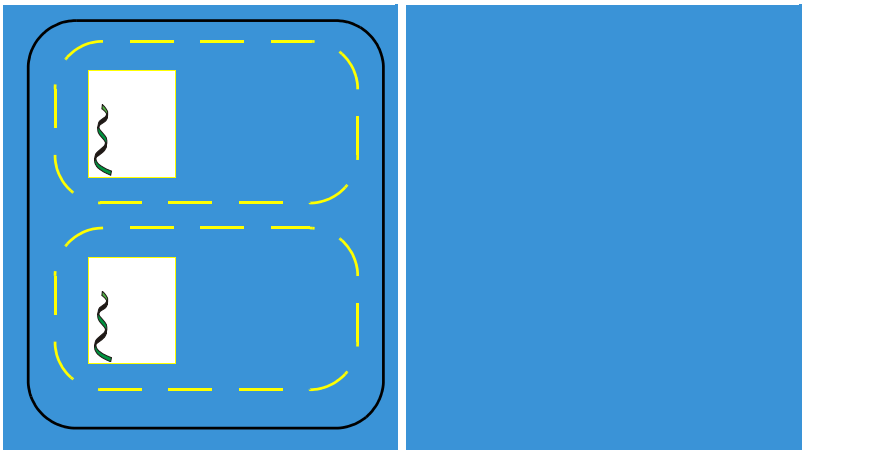
Figure 28: Step 2: Forking a Process



- **Workers group**: TotalView adds the thread in the second process to the existing group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 3 The second process is exec'd. (See Figure 29.)

Figure 29: Step 3: Exec'ing a Process



- **Control group**: The group is unchanged.
- **Share group**: TotalView creates a second share group having this exec'd process as a member. TotalView removes this process from the first share group.
- **Workers group**: Both threads are in the workers group.

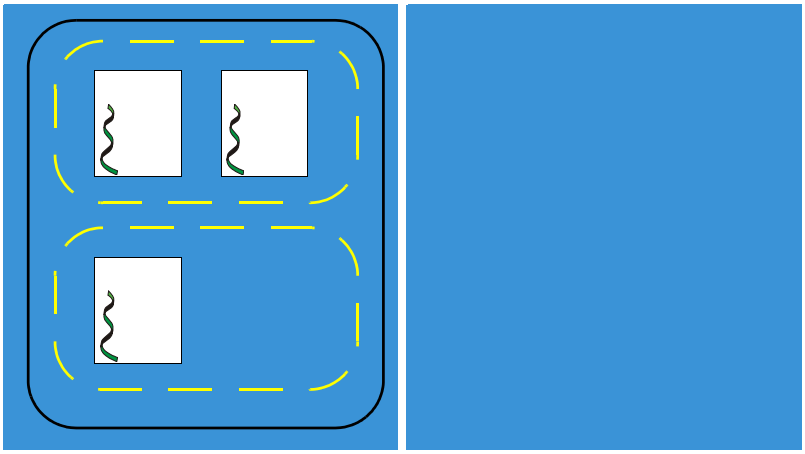
- **Lockstep group**: There are no lockstep groups because the threads are running.

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. (See Figure 30.)

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 this 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. (See Figure 31 on page 28.)

- **Control group**: TotalView extends the control group so that it contains the fourth process 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 control group 1 creates a thread. This adds a second thread to one of the processes. (See Figure 32 on page 28.)

Figure 31: Step 6: Creating a Remote Process

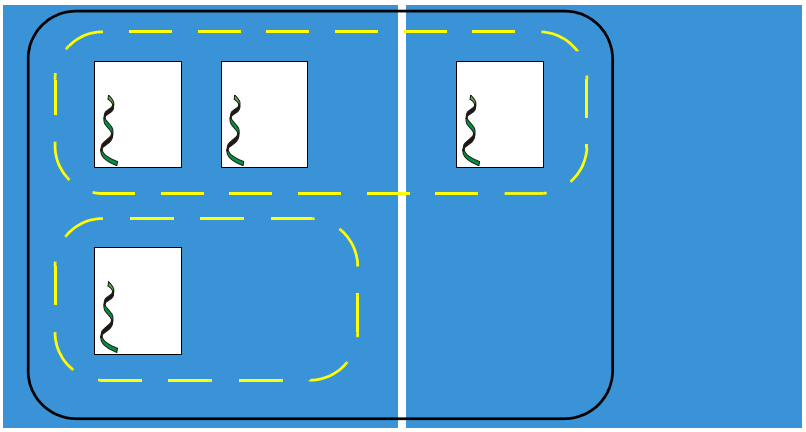
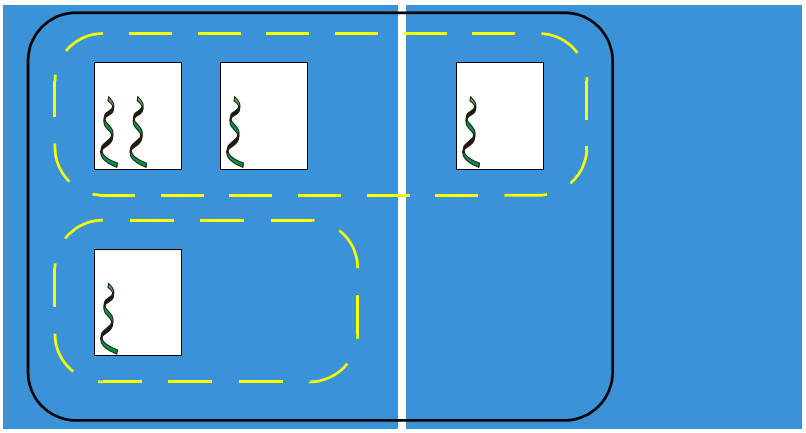


Figure 32: Step 7: A Thread Is Created



- **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, and the breakpoint is shared. The process executes until all three processes are at the breakpoint. (See Figure 33 on page 29.)

- **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. (See Figure 34 on page 29.)

- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.
- **Workers group**: The group is unchanged.

Figure 33: Step 8: Hitting a Breakpoint

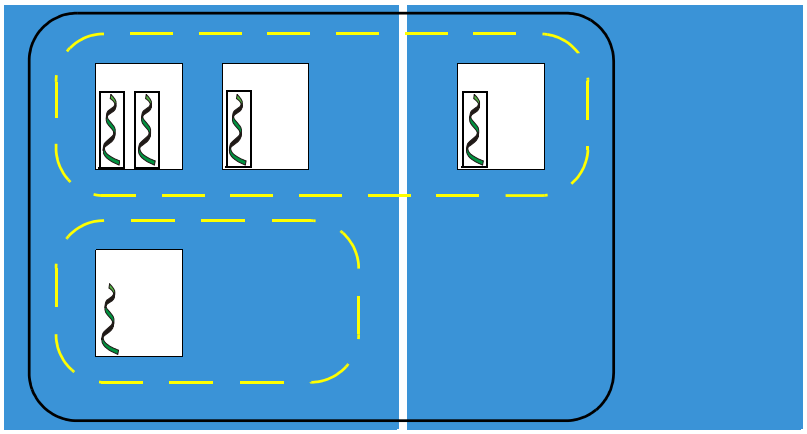
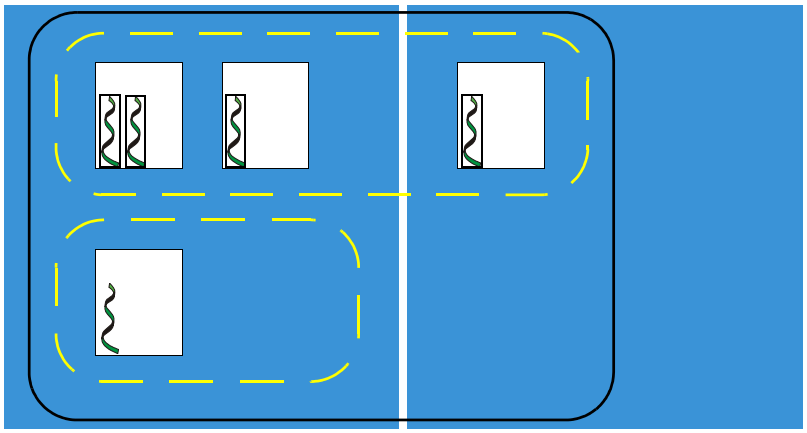


Figure 34: Step 9: Stepping the Lockstep Group



- **Lockstep group**: The lockstep groups are unchanged. (Note that there are other lockstep groups. This will be explained in Chapter 11.)

What Comes Next

Clearly, this example could keep on going until a much more complicated system of processes and threads was created. However, adding more processes and threads won't do anything much different than what's been discussed.

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 multiprocess, multithreaded 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's control, you will want to run the process being debugged to an action point so you can inspect the program's state at that place. 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, or threads as you need to control. While each can be controlled individually, you will probably have problems remembering what you're doing if you're controlling large numbers of these things independently. The reason that TotalView creates and manages groups is so 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 is manipulating. Things get complicated when the process being investigated is using data created by other processes, and these processes might be dependent on other processes.

This means that there is a rather typical pattern to the way you use TotalView to locate problems:

1. At some point, you should make sure that the groups you are manipulating do not contain service or manager threads. (You can remove processes and threads from a group with the dgroups –remove command or the Groups > Edit Group command.)
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 multiprocess, multithreaded program, you will want to set a *barrier point*—this is a special kind of breakpoint—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. Using barrier points is much more efficient. Barrier points are discussed in Chapter 14. Online, you'll find information additional information within the Action Point area of TotalView's Tip of the Week archive. This is located at http://www.etnus.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.

   Here's where things begin to get complicated. You've been focusing on one process or thread. If another process or thread is modifying the data and you become convinced that this is the problem, you'll want to go off to it and see what's going on.

At this point, 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 multiprocess, multithreaded program can be doing a great number of things. Understanding where to look when problems occur is the "art."

For example, you'll most often execute commands at the default focus. Only when you think that the problem is occurring in another process will you change to that process. You'll still be executing in a default focus, but this time the default focus is concentrated on other process.

While it will often seem like you need to do a lot of shifting to another focus, what you will probably do is:

- 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 would use the dfocus command to limit the scope of a future command. For example, here's 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.

What you've been reading is just an overview of the threads, processes, and groups. You'll find a lot more information in Chapter 11, "*Using Groups, Processes, and Threads*," on page 197.

Part II: Setting Up

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

Chapter 3: Setting Up a Debugging Session
: The way you configure your TotalView environment is the same on all operating systems and in all environments. This chapter tells you what you need to know to start TotalView and tailor how it works.

  You should, at a minimum, glance at this chapter to see what's here so you can come back at a later time, if you need to.

Chapter 4: 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. Consequently, the primary focus of this chapter is what to do when there are problems.

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

Chapter 5: Setting Up Parallel Debugging Sessions
: TotalView lets you debug programs created using many different parallel environments such as OpenMP, MPI, MPICH, UPC, and the like. This chapter discusses these environments.

  Because each environment's discussion is self-contained, you don't need to read the entire chapter. Instead, just locate what you need and skip the rest.

3 Setting Up a Debugging Session

This chapter explains how to set up a TotalView session. It also describes some of the most used commands and procedures. Depending upon your needs, you may need information on setting up remote debugging sessions, which is found in Chapter 4, "*Setting Up Remote Debugging Sessions*," on page 61. For information on setting up parallel debugging sessions, see Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 75.

In this chapter, you will learn about:

Compiling Programs

Before you start debugging a program, you must compile it. All you need do to make your program ready for debugging with TotalView is to add the –g option to your compile command. 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, refer to "*Viewing the Assembler Version of Your Code*" on page 175.

Table 1 presents some general considerations, but you should check "*Compilers and Platforms* in the *TotalView Reference Guide* to see if there are any special considerations.

Table 1: Compiler 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 *any* program with TotalView.
Optimization option (usually –O)	Rearranges code to optimize your program's execution. Some compilers won't let you use the –O and the –g option at the same time. Even if your compiler lets you use the –O option, don't do it when debugging your program as strange results often occur.	After you finish debugging your program with TotalView.
Multiprocess programming library (usually dbfork)	Uses special versions of the fork() and execve() system calls. In some cases, you will need to use –lpthread. Using dbfork is discussed in "*Linking with the dbfork Library*" contained in the "*Compilers and Platforms*" Chapter of the *TotalView Reference Guide*.	Before debugging a multiprocess program that explicitly calls fork() or execve(). Refer to "*Processes That Call fork()*" on page 281 and "*Processes That Call execve()*" on page 282.

File Extensions

When TotalView reads in a file, it uses the file's extension to determine which programming language you used. If TotalView does the wrong thing, you can have it do the right thing by setting the TV::suffixes variable in a startup file. For more information, see the "*TotalView Variables*" chapter in the *TotalView Reference Guide*.

Starting TotalView

TotalView can debug programs that run in many different computing environments and which use a variety of parallel processing modes. This section looks at few of the ways you can start TotalView. The "*TotalView Command Syntax*" chapter in the *TotalView Reference Guide* contains more detailed information.

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

totalview [*executable* [*corefile*]] [*options*]

where *executable* is the name of the executable file you will be debugging and *corefile* is the name of the core file being examined.

CLI: totalviewcli [*executable* [*corefile*]] [*options*]

In some cases, you may need to do something different. For example, if you are debugging an MPI program, you must invoke TotalView on mpirun. You'll find details in Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 75.

If you are using the GUI, you can also use the CLI at the same time by selecting the Tools > Command Line command.

Here are some examples that show how to start TotalView:

Starting TotalView

totalview

Starts TotalView without loading a program or core file. After TotalView starts, you can load a program by using the File > New Program command.

CLI: totalviewcli *then* dload *executable*

Debugging a program

totalview *executable*

Starts TotalView and loads the *executable* program.

CLI: totalviewcli *executable*

Debugging a core file

totalview *executable corefile*

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

CLI: dattach –c *corefile* –e *executable*

Passing arguments to the program being debugged

totalview *executable* –a *args*

Starts TotalView and passes all the arguments following –a 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 –a and want to add arguments after TotalView loads your program, use the Process > Startup command.

CLI: dset ARGS_DEFAULT {*value*}

Debugging a program that runs on another computer

totalview *executable* –remote *hostname_or_address*[*:port*]

Starts TotalView on your local host and the TotalView Debugger Server (tvdsvr) 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*]

For more information on:

- Debugging parallel programs such as MPI, PVM, or UPC, refer to Chapter 5, "*Setting Up Parallel Debugging Sessions*," on page 75.
- The totalview command, refer to "*TotalView Command Syntax*" in the *TotalView Reference Guide*.
- Remote debugging, refer to "*Setting Up and Starting the TotalView Debugger Server*" on page 61 and "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

Initializing TotalView

When TotalView begins executing, it can grab initialization and startup information from different kinds of files. The two most commonly used 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 will execute this file whenever it starts executing. This file, which you must name tvdrc, resides in a .totalview subdirectory contained in your home directory. TotalView will create 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, or the directory in which you invoke TotalView. If the file is present in one or all of these places, TotalView reads and executes its contents. Only the initialization file within your .totalview directory has the name tvdrc. The other initialization files have the name .tvdrc. Notice the dot preceding the file name.

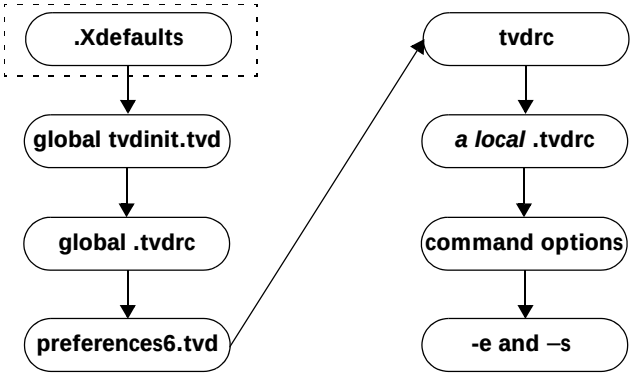
Before Version 6.0, you would place your personal .tvdrc *file in your home directory. If you do not move this file into the .totalview directory, TotalView will still find it. However, if you also have a* tvdrc *file in the* .totalview *directory, TotalView will ignore the* .tvdrc *in your home directory.*

TotalView automatically writes your preferences file into your .totalview subdirectory. It's name is preferences6.tvd. Do not modify this file as TotalView will overwrite it whenever 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 will execute after a tvdrc 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 lets you create a shell command that starts the CLI.

Figure 35 shows the order in which TotalView executes initialization and startup files.

Figure 35: 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. Prior to TotalView release 6.0, the .Xdefaults file was extensively used. Beginning at TotalView 6.0, its use is negligible.

If you have an X resources file, TotalView will read it the first time Release 6.0 starts executing. It will then write any TotalView resources it finds to your preferences6.tvd *file. If you change a value after this file is written, TotalView will ignore your change. The only exceptions are Visualizer X resources. For information on these resources, go to www.etnus.com/Support/docs/xresources/XResources.html. You can force TotalView to reread your X resources by deleting your preferences file.*

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 a action point file into the same subdirectory as your program, TotalView automatically reads the information in this file when it loads your program.

The format of a Release 6.0 action point file differs from that used in earlier releases. While TotalView 6.0 can read action point files created by earlier versions, earlier versions cannot read a Release 6.0 action point file.

You can also invoke scripts by naming them in the TV::process_load_callbacks list. Information on using this variable is contained within the "*Variables*" chapter of the *TotalView Reference Guide*.

If you are debugging multiprocess programs that run on more than one computer, TotalView caches library information in the .totalview subdirectory. If you wish to move this cache to another location, set the TV::library_cache_directory to this location. The files within this cache directory can be shared among users.

Exiting from TotalView

You can exit from TotalView by selecting the File > Exit command. You can select this command in the Root, Process, and Variable Windows. (See Figure 36.)

Figure 36: File > Exit Dialog Box

Select Yes to exit. Otherwise, select No. As TotalView exits, it kills all programs and processes that it started. However, programs and processes that TotalView did not start continue to execute.

CLI: exit

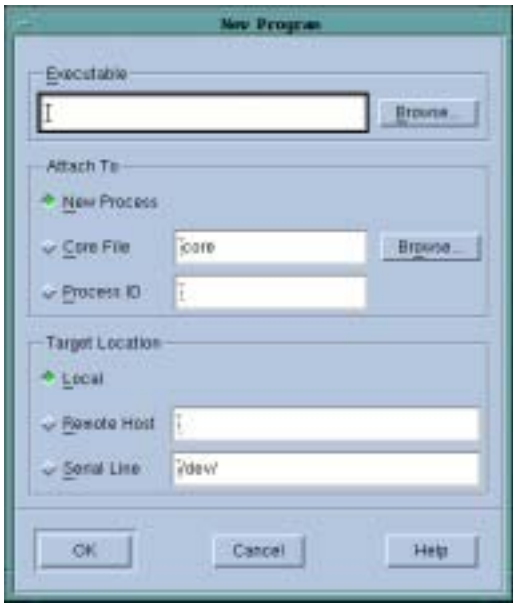
If you have a CLI window open, TotalView also closes this window. Similarly, if you type exit *in the CLI, the CLI will close GUI windows. If you type* exit *within the CLI and you have a GUI window open, TotalView will still display this dialog box. If you have iconified your TotalView GUI windows, it is possible that will not see this dialog box under some window managers and TotalView will appear to be hung.*

Loading Executables

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. (See Figure 37 on page 41.)

The controls within this dialog box lets you:

Figure 37: File > New Program Dialog Box



- **Load a new executable**

  Type the path name into the Executable field.

  CLI: dload –e *executable*

- **Load a core file**

  Type the name into the Core File field. You must also type the path name of the executable associated with this core file in the Executable field.

  CLI: dattach –c *corefile* –e *executable*

- **Load a program using process ID**

  Type a process ID into the Process ID field *and* type the associated executable's path name into the Executable field.

  CLI: dattach *executable pid*

If you need to debug a program on a remote machine, type the machine's host name or IP address in the Remote Host field. If the program is local, make sure that you have selected the Local button.

CLI: dload *executable* –r *hostname*

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 direc-

tories named using the File > Search Path command or listed in your PATH environment variable.

CLI: dset EXECUTABLE_PATH

If you select New Process, TotalView always loads a new copy of the program named in the Executable field. Even if the program is already loaded, TotalView still loads another copy.

Debugging over a serial line is discussed in "*Debugging Over a Serial Line*" on page 71.

Loading Remote Executables

If TotalView fails to automatically load a remote executable, you may need to disable *autolaunching* for this connection and manually start the TotalView Debugger Server (tvdsvr). (*Autolaunching* is the process of automatically launching tvdsvr processes.) You can disable autolaunching by adding the *hostname*:*portnumber* suffix to the name you type in the Remote Host field. As always, the *portnumber* is the TCP/IP port number on which the debugger server is communicating with TotalView. Refer to "*Setting Up and Starting the TotalView Debugger Server*" on page 61 for more information.

You cannot examine core files on remote systems. However, if you rlogin or rsh to that system, you'll be executing TotalView locally. While there's another step involved, the result will be the same as if the core file were local.

You can connect to a remote machine in three ways:

- Using the –remote command-line option when you start TotalView. For details on the syntax for the –remote command-line option, see "*Starting TotalView*" on page 36.
- Using the File > New Program command after you start TotalView.

  CLI: dload *executable* –r *hostname*

- Connecting to a remote host using the File > New Program command and then displaying the Unattached Page of the Root Window. You can now attach to these programs by diving into them.

  CLI: If you're using the CLI, you will need to know the file's name so that you can attach to the program using the dattach command.

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 5, "Setting Up Parallel Debugging Sessions," on page 75.

Attaching to Processes

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. You can attach to single

processes, multiprocess programs, and these programs can be running remotely.

To attach to a process, either use the Unattached Page in the Root Window or use the File > New Program command located on the Root and Process Windows. (Using the Unattached Page is easier if the process is listed. However, if it's not there, you must use the File > New Program command.)

CLI: dattach *executable pid*

If the process or any of its children calls the execve() routine, you may need to attach to it by creating a new Process Window. This is because TotalView relies on the ps command to obtain the process name, and it can make mistakes.

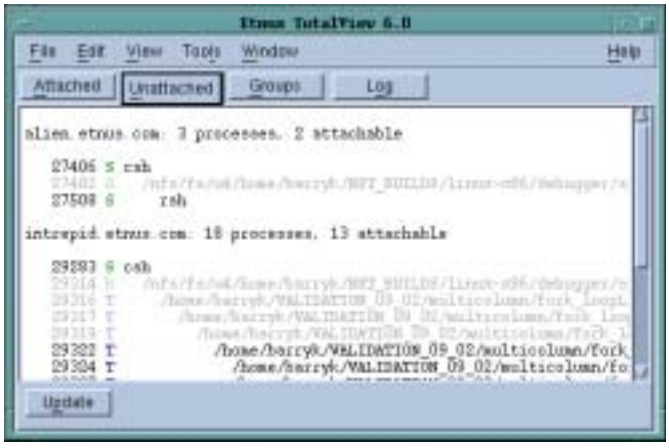
When you exit from TotalView, TotalView kills all programs and processes that it started. However, programs and processes that were executing before you brought them under TotalView's control continue to execute.

Attaching Using the Unattached Page

Here's the procedure for using the Unattached Page to attach a process:

1. Go to the Root Window and select the Unattached tab.
   This page lists the process ID, status, and name of each process associated with your username. The processes that appear dimmed are those that are being debugged or those that TotalView won't allow you to debug. For example, you can't debug the TotalView process itself. (See Figure 38.) The processes at the top of this figure are local. The remaining processes are remote.

Figure 38: Unattached Page



If you're debugging a remote process, the Unattached Page also shows processes running under your username on each remote host name. You can attach to any of these remote processes. For more information on remote debugging, refer to "*Setting Up and Starting the TotalView Debugger Server*" on page 61 and "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

2 Dive into the process you wish to debug by double-clicking on it.

A Process Window appears. The right arrow points to the current program counter (PC), indicating where the program was executing when TotalView attached to it.

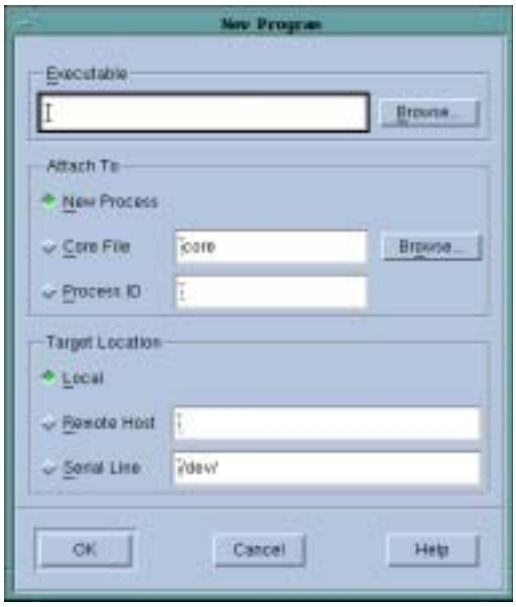
Attaching Using File > New Program and dattach

Here's the procedure for using the Root Window's File > New Program command to attach to a process:

1 Use the ps shell command to obtain the process ID (PID) of the process.

2 Select the File > New Program command. TotalView displays the dialog box shown in Figure 39.

Figure 39: File > New Program Dialog Box



Enter a file name in the Executable field. This name can be a full or relative path name. If you supply a simple file name, TotalView searches for it in the directories named using the File > Search Path command or listed in your PATH environment variable.

Enter the process ID (PID) of the *unattached* process into the Process ID field.

CLI: dattach *pid*
dset EXECUTABLE_PATH

3 Select OK.

If the executable is a multiprocess program, TotalView asks if you want to attach to all relatives of the process. To examine all processes, select Yes.

If the process has children that call execve(), TotalView tries to determine each child's executable. If TotalView can't figure it out, you must delete (*kill*) the parent process and start it again using TotalView.

A Process Window will appear. In this window, the right arrow points to the current program counter (PC), which is where the program was executing when TotalView attached to it.

Detaching from Processes

You can use the following procedure 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. Choose the signal that TotalView should send 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. (See Figure 40.)

CLI: No equivalent to Thread > Continuation exists.

Figure 40: Thread > Continuation Signal Dialog Box

2 Select the Process > Detach command.

CLI: ddetach

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

Examining Core Files

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. In the middle section of the dialog box, type the name of the core file in the Core File field, and then select OK. In the top portion, enter the *executable*'s name.

CLI: dattach –c *corefile* –e *executable*

You can only debug local core files. However, if you do an rsh or rlogin to that system and then start TotalView, you'll be debugging these files locally. That is, while you can't debug remote core file, they will always be local to some system.

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 12, "*Examining and Changing Data*," on page 227 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.

Viewing Process and Thread State

Process and thread state is displayed in:

- The Attached Page of the Root Window, for processes and threads.
- The Unattached Page of the Root Window, for processes.
- The process and thread status bars of the Process Window.
- The Threads Pane of the Process Window.
- The P/T Set Browser.

Figure 41 on page 47 shows TotalView displaying process state information in the Attached Page.

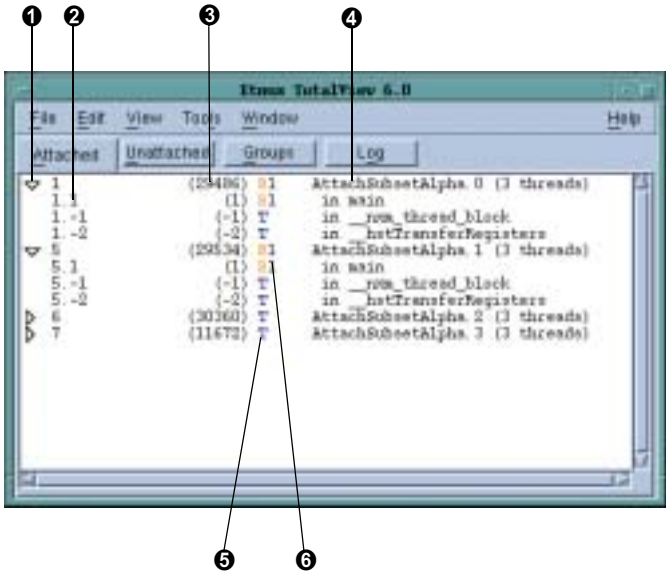
CLI: dstatus and dptsets

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

The status of a process includes the process location, the process ID, and the state of the process. (These characters are explained in "*Attached Process States*" on page 47.)

The Unatttached Page lists all processes associated with your username. The information in this page is similar to the information in the Attached Page, differing only in that TotalView dims out the processes being

Figure 41: Attached Page Showing Process and Thread Status



❶ Collapse/expand toggle
❷ TotalView thread ID (TID)
❸ System thread ID (SYSTID)
❹ Program name
❺ Process status
❻ Action point ID number

debugged. The status bars in the Process Window display similar information. (See Figure 42.)

Figure 42: Process and Thread Labels in the Process Window

If the TotalView-assigned thread ID and the system-assigned thread ID are the same, TotalView displays only one ID value.

Attached Process States

TotalView uses the following letters to indicate process and thread state. (The position of these letters in the Attached Page is indicated by ❺ in Figure 41.)

Table 2: Attached Process and Thread States

State Code	State Name
blank	Exited or never created
B	At breakpoint
E	Error reason
K	In kernel
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 "*Han-*

dling Signals" on page 48 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."

Unattached Process States

TotalView derives the state information for a process displayed in the Unattached Page 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. Here are the state indicators that TotalView displays:

Table 3: Summary of Unattached Process States

State Code	State Description
I	Idle
R	Running
S	Sleeping
T	Stopped
Z	Zombie

Handling Signals

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. You can do this using:

- A dialog box (described in this section)
- The –signal_handling_mode command-line option to the totalview and totalviewcli commands (refer to "*TotalView Command Syntax*" in the *TotalView Reference Guide*)

Unless you tell TotalView otherwise, here is how it handles UNIX signals:

Table 4: Default Signal Handling Behavior

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	

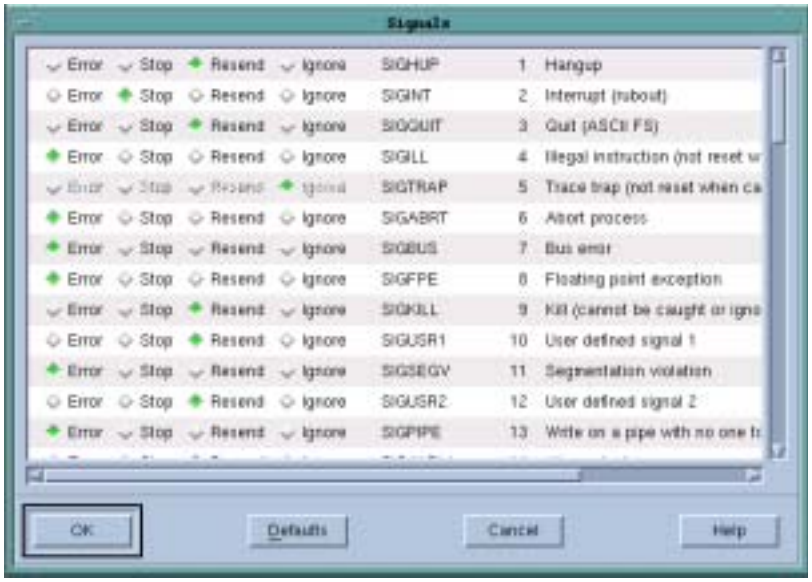
TotalView uses the SIGTRAP *and* SIGSTOP *signals internally. If a process receives either of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. You cannot alter the way TotalView uses these signals.*

On some systems, hardware registers affect how TotalView and your program handle signals such as SIGFPE. For more information, refer to "*Interpreting the Status and Control Registers*" on page 195 and "*Architectures*" in the *TotalView Reference Guide*.

If you are using an SGI computer, setting the TRAP_FPE *environment variable to any value indicates that your program will trap underflow errors. If you set this variable, however, you will 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 will set SIGFPE to Resend.) As an alternative, you can use the* –signal_handling_mode *"action_list" option.*

You can change the signal handling mode using the File > Signals command. (See Figure 43.)

Figure 43: File > Signals Dialog Box



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

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 button (which is found in the Options Page of the File > Preferences Dialog Box. (See Figure 48 on page 54.)

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 that it should not open or raise windows.

CLI: dset TV::GUI::pop_on_error

If processes in a multiprocess 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, which is found in the File > Preferences' Action Points Page, TotalView opens or raises the Process Window when your program reaches a breakpoint.

CLI: TV::GUI::pop_at_breakpoint

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

Table 5: Signal Handling Buttons

Button	Meaning
Error	Stops the process, places it in the *error* 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 will also stop all related processes. Select this button for severe error conditions such as SIGSEGV and SIGBUS.
Stop	Stops the process and places it in the *stopped* 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 as most programs have handlers to handle program termination.
Ignore	Discards the signal and continues the process. The process will not know that something raised a signal.

Do not use Ignore mode 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 will immediately recur because the failing instruction repeatedly reexecutes.*

Setting Search Paths

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.

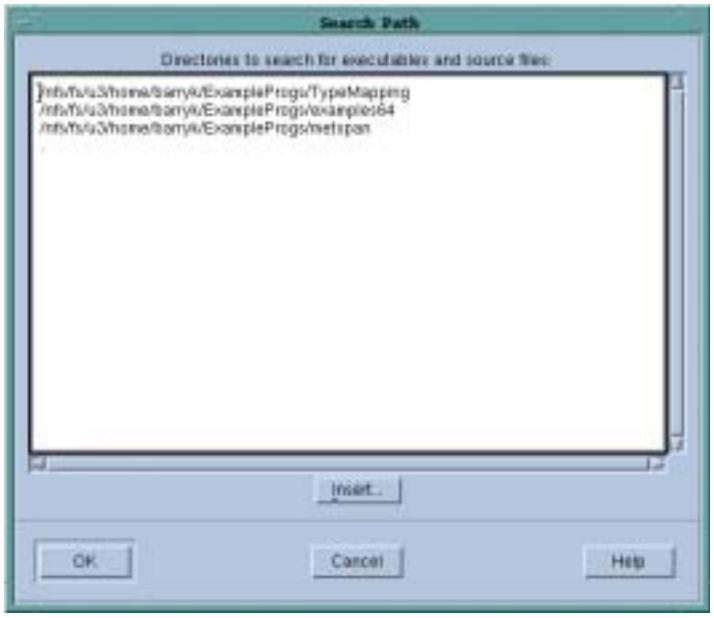
CLI: dset EXECUTABLE_PATH

These search paths apply to *all* processes that you're debugging. (See Figure 44 on page 51.)

TotalView searches the following directories (in order):

1. The current working directory (.).
2. The directories you specify by using the File > Search Path command in the exact order you enter them.

Figure 44: File > Search Path Dialog Box



3. If you entered a full path name for the executable when you started TotalView, TotalView searches this directory.
4. If your executable is a symbolic link, TotalView will look in the directory in which your executable actually resides for the new file.

   As you can have multiple levels of symbolic links, TotalView keeps on following links until it finds the actual file. After it has found the current executable, it will look in its directory for your file. If it isn't there, it'll back 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.

When entering 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 type path names directly.
- You can cut and paste directory information.
- You can use the Insert button to tell TotalView to display the Select Directory dialog box that lets you browse through the file system, interactively selecting directories. (See Figure 45 on page 52.)

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, 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 TV::search_path variable.

Figure 45: Select Directory Dialog Box

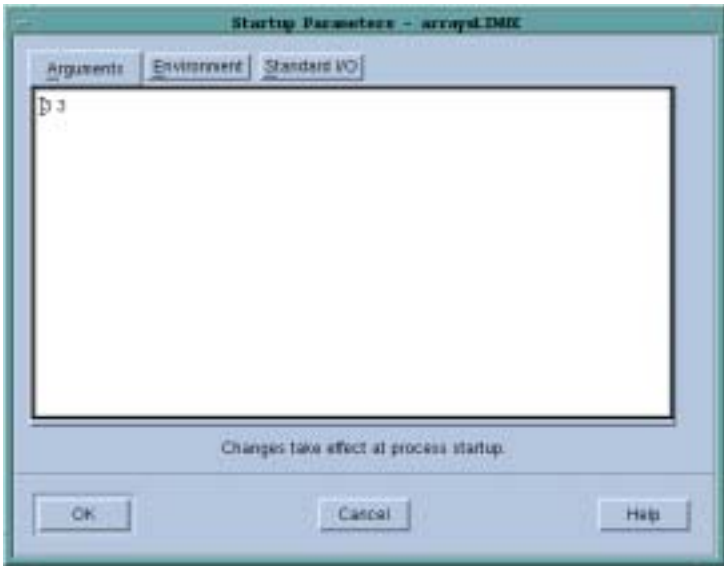
Setting Command Arguments

When TotalView creates a process, it uses the name of the file containing the executable code for the process's program name. If your program requires command-line arguments and you hadn't entered them using TotalView's –a command-line option, here's how you can set these arguments *before* you start the process:

1 Select the Arguments Tab within the Process > Startup Parameters Dialog Box. (See Figure 46.)

CLI: dset ARGS_DEFAULT {*value*}

Figure 46: Process > Startup Parameters Dialog Box: Arguments Page



2 Type the arguments you want TotalView to pass to your program. 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 quotes. When you're done, select OK.

Setting Input and Output Files

Before your program begins executing, TotalView defines how it will manage standard input (stdin) and standard output (stdout). Unless you tell it otherwise, stdin and stdout use the shell window from which you invoked TotalView.

The Process > Startup command lets you redirect stdin or stdout. You can only do this before your program begins executing. Here's how:

1 Select the Standard I/O Tab from the dialog box displayed when you invoke the Process > Startup Parameters command. (See Figure 47.)

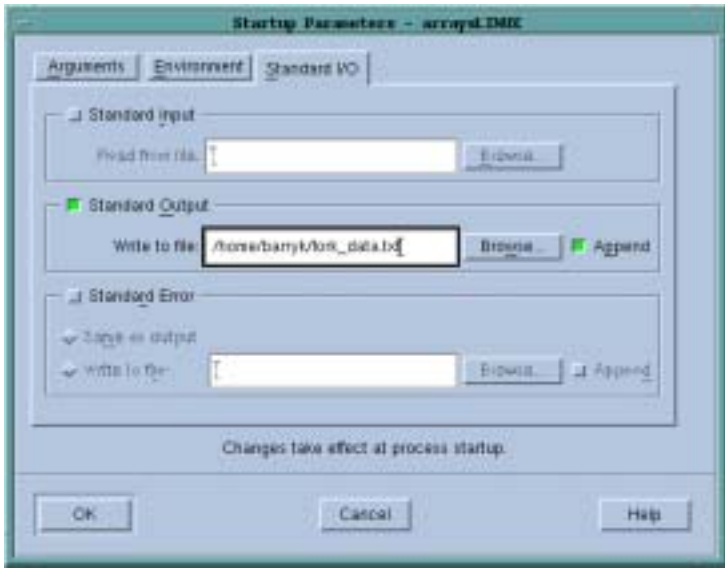


Figure 47: Process > Startup Parameters Dialog Box: Standard I/O Page

2 Type the name of the file, relative to your current working directory. Entering names in these text boxes is equivalent to using <, >, or >& symbols in most shells.

3 Select OK.

If you select the Append check box, TotalView appends new information to the end of the file if the file already exists. If it isn't checked, TotalView overwrites the file's contents.

If you select the Same as output check box, TotalView writes stderr information to the same place indicated in the Standard Output field.

CLI: drun and drerun have arguments that let you reset stdin, stdout, and stderr.

Setting Preferences

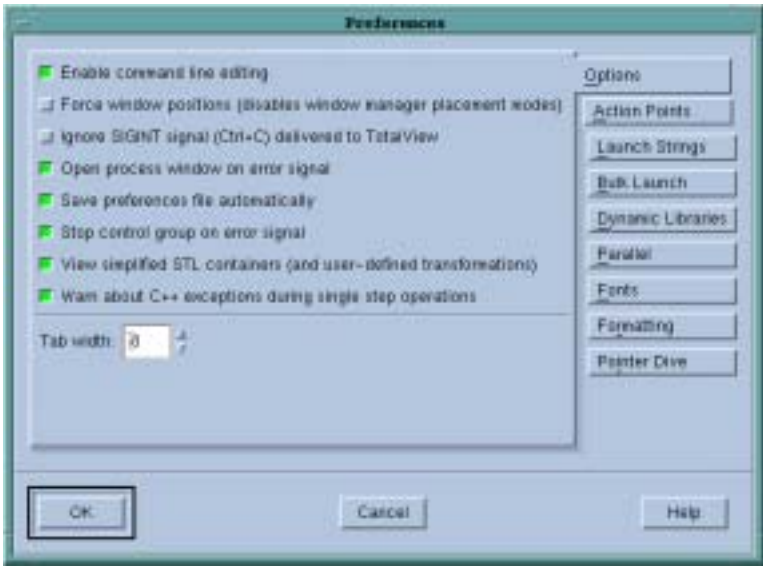
The File > Preferences command lets you tailor many of TotalView's behaviors. This section contains an overview of these preferences. Detailed explanations are in the online Help.

Some settings such as the prefixes and suffixes examined before loading dynamic libraries can be different from operating system to operating system. If these settings can differ, TotalView will let you set values for each operating system. This is done transparently, which means that you only see an operating system's values when you are running TotalView on that operating system. In general, this applies to the server launch strings and dynamic library paths.

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

- Options. This page contains check boxes that are either general in nature or that influence different parts of the system. (See Figure 48.)

Figure 48: File > Preferences Dialog Box: Options Page



- Action Points. The commands on this page indicate if 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 49 on page 55.)
- 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. Notice that default values exist for these launch strings. (See Figure 50 on page 55.)
- Bulk Launch. The fields and commands on this page configure TotalView's bulk launch system. See Chapter 4 for more information. (See Figure 51 on page 56.)

Figure 49: File > Preferences Dialog Box: Action Points Page

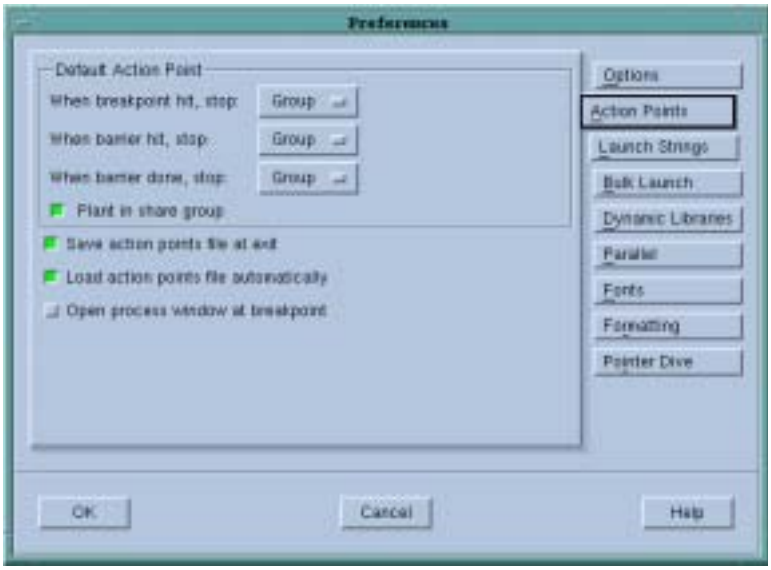
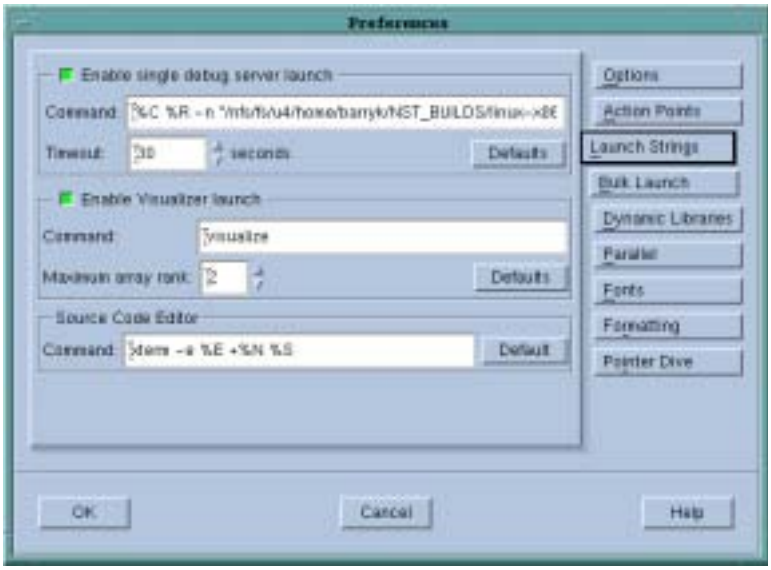


Figure 50: File > Preferences Dialog Box: Launch Strings Page



- Dynamic Libraries. This page lets you control which symbols are added to TotalView when it loads a dynamic library and how much of a libraries symbols are read in. (See Figure 52 on page 56.)
- Parallel. This page lets you define what will occur when your program goes parallel. (See Figure 53 on page 57.)
- Fonts. Use this page to specify the fonts used in the user interface and when TotalView displays your code. (See Figure 54 on page 57.)
- Formatting. Use this page to control how your program's variables are displayed. (See Figure 55 on page 58.)
- Pointer Dive. Use this page to control how pointers are dereferenced and how pointers to arrays are cast. (See Figure 56 on page 58.)

Figure 51: File > Preferences Dialog Box: Bulk Launch Page

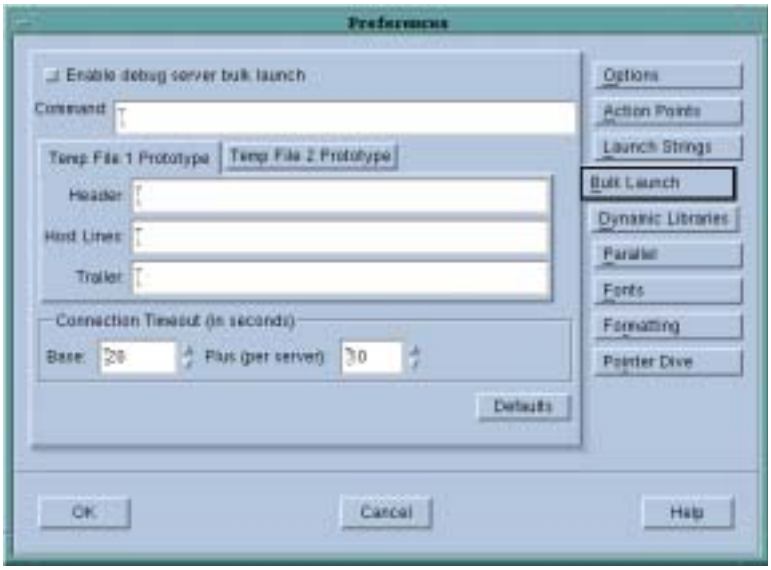
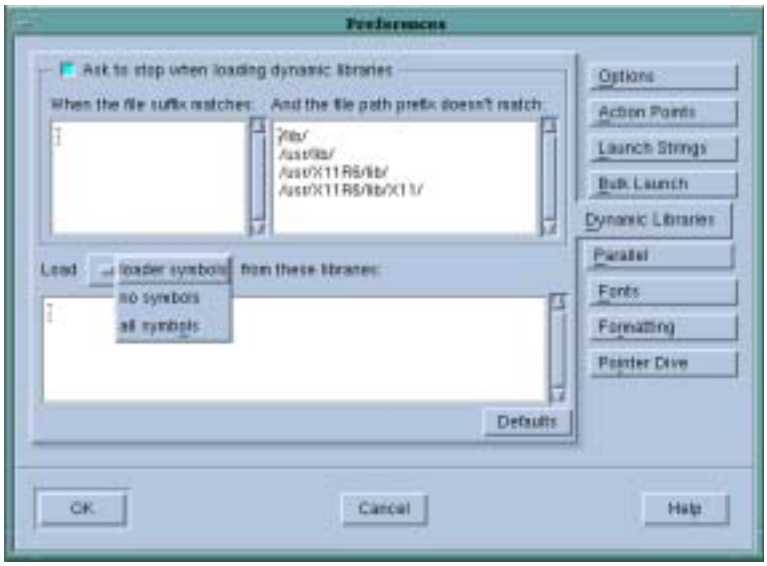


Figure 52: File > Preferences Dialog Box: Dynamic Libraries Page



Setting Preferences, Options, and X Resources

While preferences are the best way to set many of TotalView's features and characteristics, TotalView also lets you use variables and command-line options to set features and characteristics.

Older versions of TotalView did not have a preference system. Instead, you needed to set values in your .Xdefaults file or in a command-line option. For example, setting totalview*autoLoadBreakpoints to true tells TotalView that is should 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's dset command, this X resource has been *deprecated*.

Figure 53: File > Preferences Dialog Box: Parallel Page

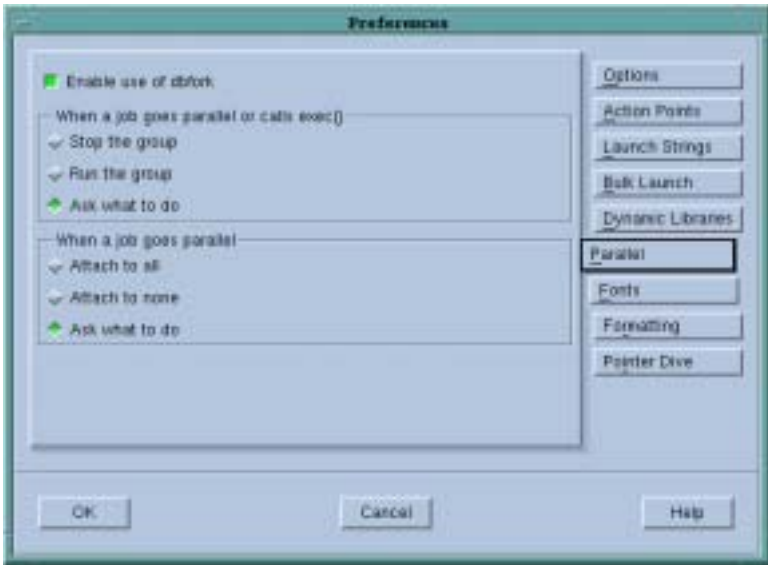
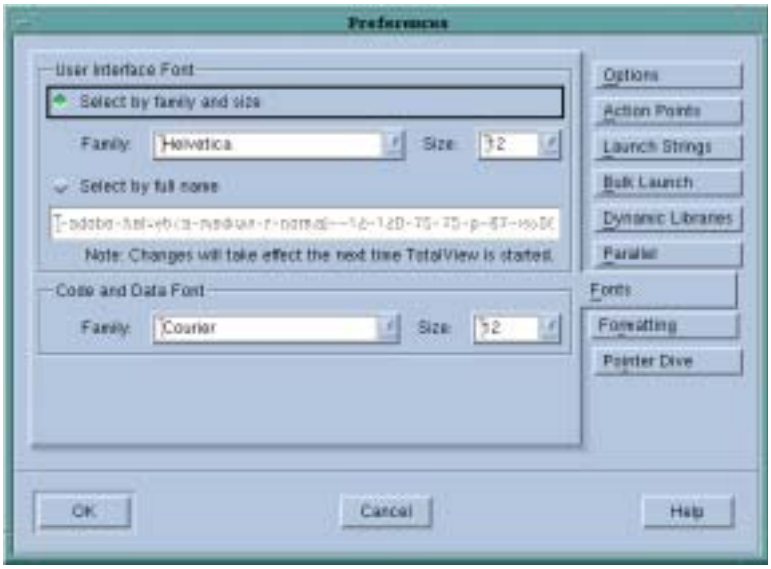


Figure 54: File > Preferences Dialog Box: Fonts Page



"Deprecated" means that the feature is still available. While the feature may exist for a while, there's no guarantee that it will continue to work. All "totalview" options have been deprecated. Those used for setting the Visualizer are still supported. Documentation for these resources can be found at www.etnus.com/Support/docs/xresources/XResources.html.

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 may want to set a state for one session or you may want to override one of your preferences. (A preference indicates a behav-

Figure 55: File > Preferences Dialog Box: Formatting Page

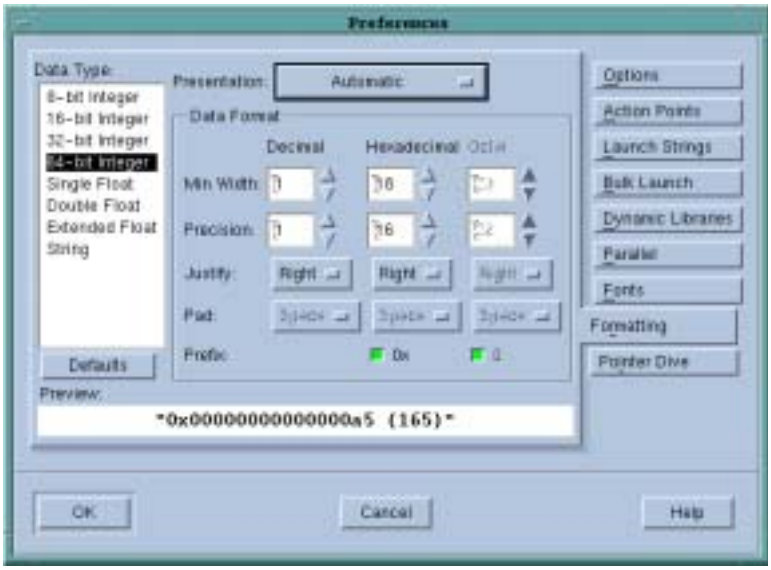
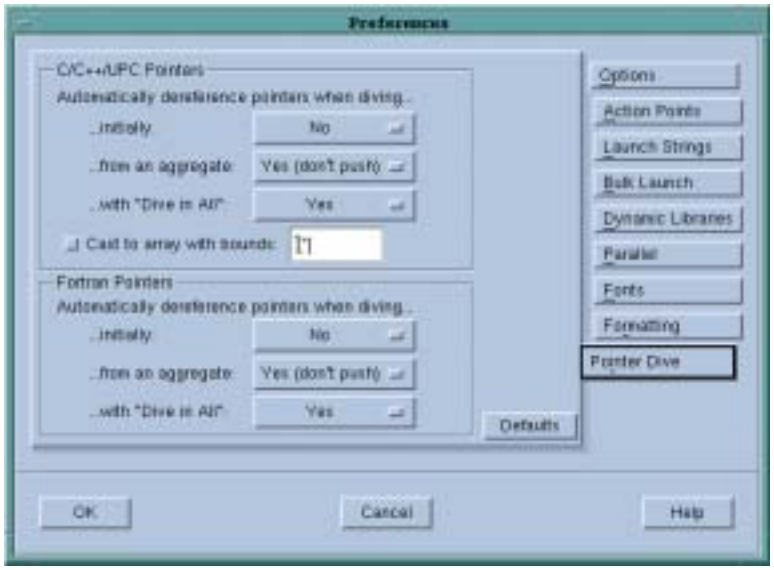


Figure 56: File > Preferences Dialog Box: Pointer Dive Page



ior 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 –bg to set the background color for TotalView windows in the TotalView 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 Environment Variables

You can set and edit the environment variables that TotalView passes to processes. When TotalView creates a new process, it passes a list of environment variables to the process. You can add to this list by using the Environment Page in the Process > Startup Parameters Dialog Box.

TotalView does not display the variables that were passed to it when you started your debugging session. Instead, this dialog box just displays the variables you have 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
```

To add, delete, or modify environment variables that you enter, select the Environment Tab from the dialog box displayed when you invoke the Process > Startup Parameters command. See Figure 57.

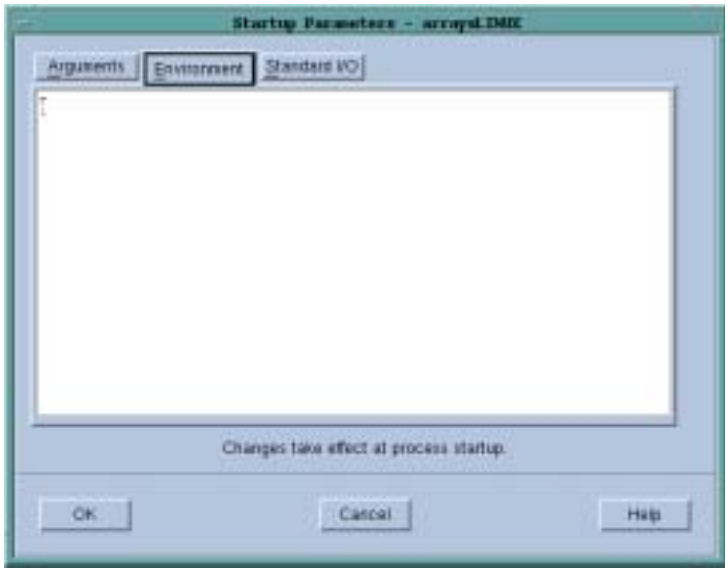


Figure 57: Process > Startup Parameters Dialog Box: Environment Page

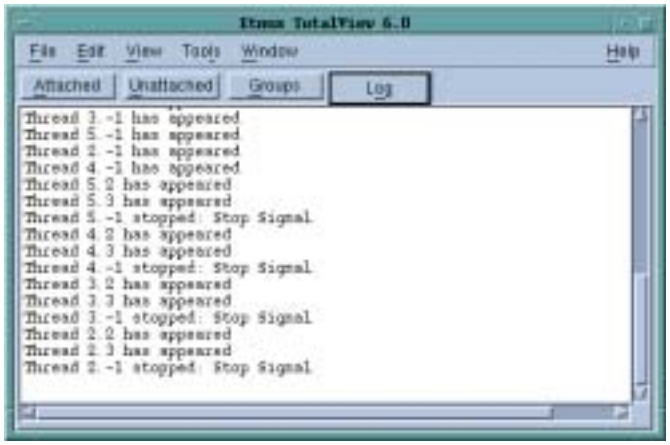
When entering environment variables, place each on a separate line. The actions you can now perform are:

- Changing the name or value of an environment variable by editing a line.
- Adding a new environment variable by inserting a new line and specifying a name and value.
- Deleting an environment variable that you added by deleting a line.

Monitoring TotalView Sessions

TotalView logs all significant events occurring for all processes being debugged. To view the event log, select the Root Window's Log Tab. This page displays a list of these events. See Figure 58 for an example.

Figure 58: Root Window Log Page



You can set the amount of information TotalView writes to this window by using the CLI's dset command to set the VERBOSE variable. If you always want it set to a value, you can set it in your .tvdrc file. For example:

```
dset VERBOSE WARNING
```

Setting Up Remote Debugging Sessions

4

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

Setting Up and Starting the TotalView Debugger Server

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

- TotalView needs to work with a TotalView processes that will be running on remote machines. This remote TotalView process, which TotalView usually automatically launches, is called the TotalView Debugger Server (tvdsvr).
- TotalView's performance depends on your network's performance. If the network is overloaded, debugging can be slow.

Unless you tell it otherwise, TotalView automatically launches tvdsvr in one of the following ways:

- It can independently launch a tvdsvr 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*.

Because TotalView can automatically launch tvdsvr, there's nothing you need to do when you're debugging remote processes. It shouldn't matter to you if a process is local or remote.

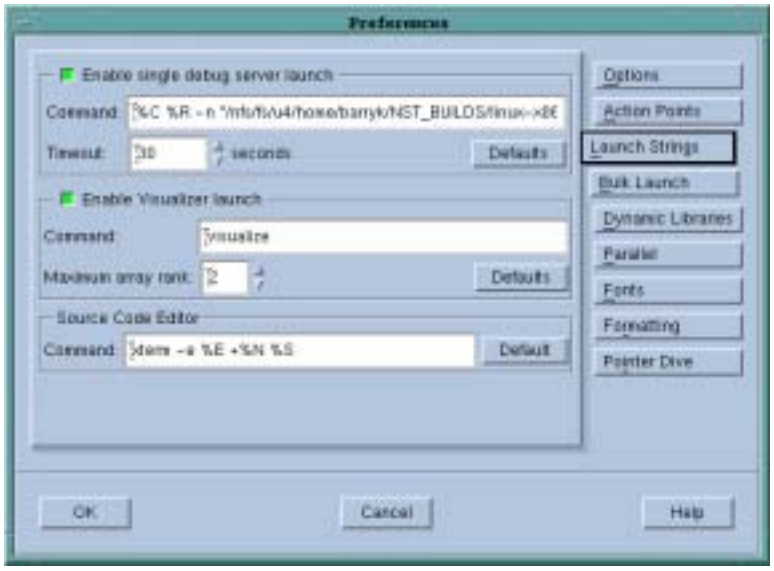
If the default single-process server launch procedure meets your needs and you're not experiencing an problems accessing remote processes from within TotalView, you can safely ignore the information in this chapter. If you do experience a problem launching the server, you should check that the tvdsvr process is in your path.

Topics in this section are:

Setting Single-Process Server Launch Options

The Enable single debug server launch preferences in the Launch Strings Page of the File > Preferences Dialog Box lets you disable autolaunch, change the command that TotalView uses when it launches remote servers, and alters the amount of time TotalView waits when establishing connections to a tvdsvr process. (See Figure 59.)

Figure 59: File > Preferences: Server Launch Strings Page



Enable single debug server launch

If you select this check box, TotalView will independently launch the TotalView Debugger Server (tvdsvr) on each remote system.

CLI: dset TV::server_launch_enabled

Even if you have enabled bulk server launch, you probably also want this option to be enabled. TotalView uses this launch string when you start TotalView upon a file when you have named a host

within 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 tvdsvr. For information on this command and its options, see "*Using the Single-Process Server Launch Command*" on page 66.

CLI: dset TV::server_launch_string

Timeout

After TotalView automatically launches tvdsvr, it waits 30 seconds for tvdsvr to respond. If the connection isn't made in this time, TotalView times out. You can change the amount 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 tvdsvr (as shown in the xterm window from which you started the debugger) before the timeout expires, select Yes in the Question Dialog Box that will appear.

Defaults

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

Selecting 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 select the OK button.

Setting Bulk Launch Window Options

The fields in the File > Preferences' Bulk Launch Page lets you change the bulk launch command, disable bulk launch, and alter connection timeouts that TotalView uses when it launches tvdsvr programs. Figure 60 on page 64 shows this page.

CLI: dset TV::bulk_launch_enabled

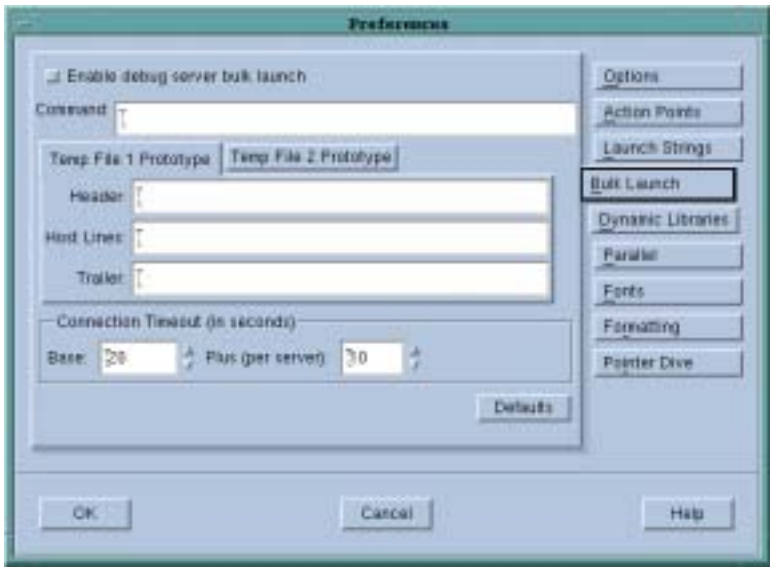
Enable debug server bulk launch

If you select this check box, TotalView uses its bulk launch procedure when launching the TotalView Debugger Server (tvdsvr). By default, bulk launch is disabled; that is, TotalView uses its single-server launch procedure.

Command

If you have enabled bulk launch, TotalView will use this command to launch tvdsvr. For information on this command and its options, see "*Bulk Server Launch on an*

Figure 60: File > Preferences: Bulk Launch Page



SGI MIPs Machine" on page 67 and "*Bulk Server Launch on an IBM RS/6000 AIX Machine*" on page 68.

CLI: dset TV::bulk_launch_string

Temp File 1 Prototype
Temp File 2 Prototype

Both tab pages have three fields. These fields let you specify the contents of temporary files that the bulk launch operation will use. For information on these fields, see "*TotalView Debugger Server (tvdsvr) 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_tmpefile2_trailer_line
```

Connection Timeout (in seconds)

After TotalView launches tvdsvr processes, it waits 20 seconds (the Base time) plus 10 seconds for each server that it will launch for responses from successfully connected processes. If connections are not made in this time, TotalView times out.

The Base timeout value can be 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 presetting these values.

CLI: dset TV::bulk_launch_base_timeout
dset TV::bulk_incr_timeout

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

Defaults

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

Selecting 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 select the OK button.

Starting the Debugger Server Manually

If TotalView can't automatically launch tvdsvr, you can start it manually. Unfortunately, this method isn't completely secure: other users can connect to your instance of tvdsvr and begin using your UNIX UID.

If you specify hostname:portnumber when opening a remote process, TotalView will not launch a debugger server.

Here is how you manually start tvdsvr:

1. Begin by insuring that both the bulk launch and single server launch or disabled. To disable the bulk launch, select the Bulk Launch Tab within the File > Preferences Dialog Box. (You can select this command from the Root Window or the Process Window.) The dialog box shown in Figure 60 on page 64 appears. Next, clear the Enable debug server bulk launch check box within the Bulk Launch Tab to disable the autolaunch feature and then select OK.

   CLI: dset TV::bulk_launch_enabled

   Similarly, select the Server Launch Tab and clear the Enable single debug server launch button.

   CLI: dset TV::server_launch_enabled

2. Log in to the remote machine and start tvdsvr:

   tvdsvr –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, refer to "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

   After printing out the port number and the assigned password, the server begins listening for connections. Be sure to remember the password; you'll need to enter it in step 3.

Because the –server option is not secure, it must be explicitly enabled. (This is usually done by your system administrator.) For details, see –server *in the "TotalView Command Syntax" chapter of the TotalView Reference Guide.*

3 From the Root Window, select the File > New Program command. Type the program's name in the Executable field and the *hostname:portnumber* in the Remote Host field and then select OK.

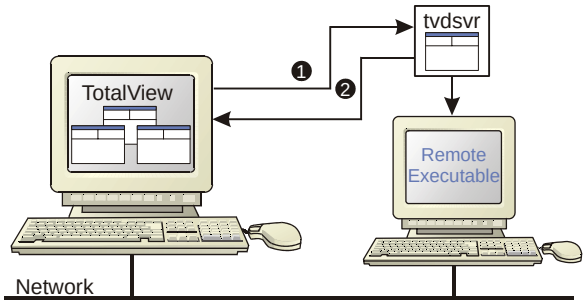
CLI: dload *executable* –r *hostname*

4 TotalView now tries to connect to tvdsvr.

When TotalView prompts you for the password, enter the password that tvdsvr displayed in step 2.

Figure 61 summarizes the steps used when you start tvdsvr manually.

Figure 61: Manual Launching of Debugger Server



❶ Makes connection
❷ Listens

Using the Single-Process Server Launch Command

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

```
%C %R –n "tvdsvr –working_directory %D –callback %L \
        –set_pw %P –verbosity %V"
```

where:

%C	Expands to the name of the server launch command being used. On most platforms, this is rsh. On HP machines, this command is remsh. If the TVDSVRLAUNCHCMD environment variable exists, TotalView uses its value instead of its platform-specific default value.
%R	Expands to the host name of the remote machine that you specified in the File > New Program or dload commands.
–n	Tells the remote shell to read standard input from /dev/null; that is, the process will immediately receive an EOF (End-Of-File) signal.

–working_directory %D
: Makes %D the directory to which TotalView will be connected. %D expands to the absolute path name of the directory.

  Using this option assumes that the host machine and the target machine are mounting identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on host and target machines.

  After changing to this directory, the shell will invoke the tvdsvr command.

  You must make sure the tvdsvr directory is in your path on the remote machine.

–callback %L
: Establishes a connection from tvdsvr to TotalView. %L expands to the host name and TCP/IP port number (*hostname*:*port*) upon which TotalView is listening for connections from tvdsvr.

–set_pw %P
: Sets a 64-bit password. TotalView must supply this password when tvdsvr establishes a connection with it. %P expands to the password that TotalView automatically generates. For more information on this password, see "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

–verbosity %V
: Sets the verbosity level of the TotalView Debugger Server. %V expands to the current TotalView verbosity setting.

You can also use the %H option with this command. This option is discussed in "*Bulk Server Launch on an SGI MIPs Machine*" on page 67.

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

Bulk Server Launch on an SGI MIPs Machine

On an SGI machine, the bulk server launch string is similar to the single-process server launch and is:

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

where:

–working_directory %D
: Makes %D the directory to which TotalView will be connected. %D expands to this directory's absolute path name.

  TotalView assumes that the host machine and the target machine mount identical file systems. That is, the path name of the directory to which TotalView is connected must be identical on both host and target machines.

  After performing this operation, tvdsvr starts executing.

–callback_host %H
: Names the host upon which TotalView makes this callback. %H expands to the host name of the machine TotalView is running on.

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

–set_pws %P
: Sets 64-bit passwords. TotalView must supply these passwords when tvdsvr 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 (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

–verbosity %V
: Sets tvdsvr's verbosity level. %V expands to the current TotalView verbosity setting.

You must enable tvdsvr's use of the array command 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
```

This assumes that the location of tvdsvr is /opt/totalview/bin. For information on the syntax of the tvdsvr command, refer to "*TotalView Debugger Server (tvdsvr) Command Syntax*" in the *TotalView Reference Guide*.

Bulk Server Launch on an IBM RS/6000 AIX Machine

On an IBM RS/6000 AIX machine, the bulk server launch string is:

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

where the options unique to this command are:

%N
: The number of servers that TotalView will launch.

%t1
: A temporary file created by TotalView that contains a list of the hosts upon which tvdsvr will run. This is the information you enter in the Temp File 1 Prototype field in 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 in the Bulk Launch Page.

%t2
: A file that contains the commands to start the tvdsvr processes on each machine. TotalView creates these lines by expanding the following template:

```
tvdsvr –working_directory %D \
   –callback %L –set_pw %P \
   –verbosity %V
```

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

Bulk Server Launch on an HP Alpha Machine

On an HP Alpha machine, the bulk server launch string is:

```
prun –T –1 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*.

Disabling Autolaunch

If after changing the autolaunch options, TotalView still can't automatically start tvdsvr, you must disable autolaunching and start tvdsvr manually. Here are two ways to do this:

- Clear the Enable single debug server launch check box in the Launch Strings Page of the File > Preferences Dialog Box.

  CLI: dset TV::server_launch_enabled

- When you debug the remote process, as described in "*Setting Up and Starting the TotalView Debugger Server*" on page 61, 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 tvdsvr *before you load a remote executable or attach to a remote process.*

Changing the Remote Shell Command

Some environments require that you create your own autolaunch 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 autolaunch command, you must use the tvdsvr command's –callback and –set_pw arguments.

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

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

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

Once telnet establishes a connection to the remote host, you could use the cd and tvdsvr commands directly, using the values of %D, %L, %P, and %V that were displayed by the echo command. For example:

```
cd directory
tvdsvr -callback hostname:portnumber –set_pw password
```

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

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

Changing the 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 machine doesn't mount the same file systems as your target machine, the debugger server may need to use a different path to access the executable being debugged. If this is the case, you could change %D to the directory used on the target machine.

If the remote executable reads from standard input, you cannot use the –n option with your remote shell command because the remote executable will receive an EOF immediately on standard input. If you omit –n, the remote executable reads standard input from the xterm in which you started TotalView. This means that you should invoke tvdsvr from another xterm window if your remote program reads from standard input. Here's an example:

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

Now, each time TotalView launches tvdsvr, a new xterm appears on your screen to handle standard input and output for the remote program.

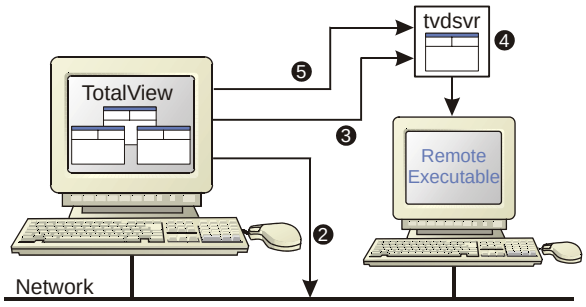
Autolaunch Sequence

If you want to know more about autolaunch, here is the sequence of actions carried out by you, TotalView, and tvdsvr:

1. With the File > New Program or dload commands, you specify the host name of the machine on which you want to debug a remote process, as described in "*Setting Up and Starting the TotalView Debugger Server*" on page 61.
2. TotalView begins listening for incoming connections.
3. TotalView launches the tvdsvr process with the server launch command. ("*Using the Single-Process Server Launch Command*" on page 66 describes this command.)
4. The tvdsvr process starts on the remote machine.
5. The tvdsvr process establishes a connection with TotalView.

Figure 62 on page 71 summarizes these actions.

Figure 62: Root Window Showing Process and Thread Status



❷ Listens
❸ Invokes commands
❹ tvdsvr starts
❺ Makes connection

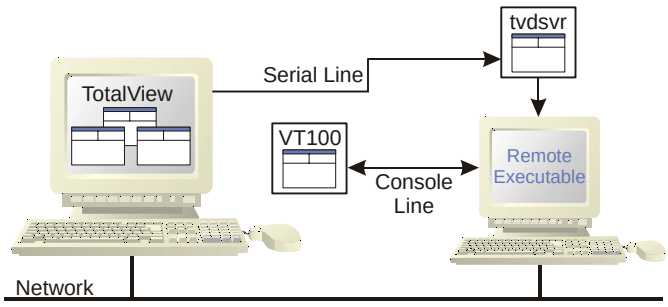
Debugging Over a Serial Line

TotalView allows you to debug programs over a serial line as well as TCP/IP sockets. However, if a network connection exists, you will probably want to use it because performance will be much better.

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

Figure 63 illustrates a TotalView debugging session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a TotalView Debugger Server running on the target host. A VT100 terminal is connected to the target host's console line, allowing you to type commands on the target host.

Figure 63: TotalView Debugging Session Over a Serial Line



Topics in this section are:

Starting the TotalView Debugger Server

To start a TotalView debugging session over a serial line from the command line, you must first start the TotalView Debugger Server (tvdsvr).

Using the console connected to the target machine, start tvdsvr and enter the name of the serial port device on the target machine. Here is the syntax of the command you would use:

tvdsvr –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:

```
tvdsvr –serial /dev/com1:baud=38400
```

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

Starting TotalView on a Serial Line

Start TotalView on the host machine and include the name of the serial line device. The syntax of this command is:

totalview –serial *device*[:baud=*num*] *filename*

or

totalviewcli –serial *device*[:baud=*num*] *filename*

where:

device	The name of the serial line device on the host machine.
num	The serial line's baud rate. If you omit the baud rate, TotalView uses a default value of 38400.
filename	The name of the executable file.

For example:

```
totalview –serial /dev/term/a test_pthreads
```

Using the New Program Window

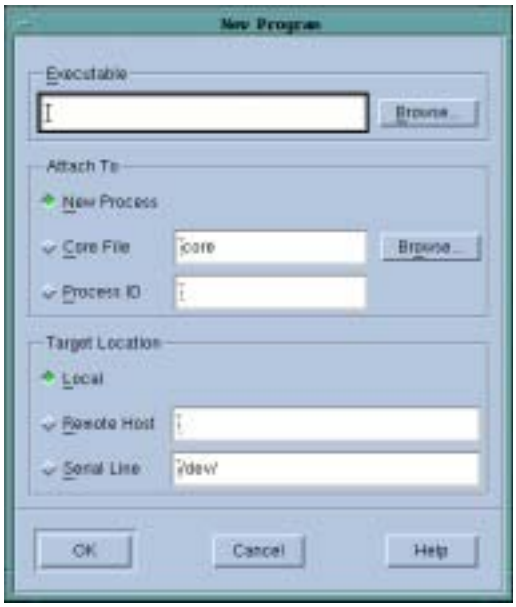
Here is the procedure for starting a TotalView debugging session over a serial line when you're already in TotalView:

1. Start the TotalView Debugger Server. (This is discussed in "*Starting the TotalView Debugger Server*" on page 72).
2. Select the File > New Program command. TotalView responds by displaying the dialog box shown in Figure 64 on page 73.

   Type the name of the executable file in the Executable field.

   Type the name of the serial line device in the Serial Line field.
3. Select OK.

Figure 64: File > New Program Dialog Box



5 Setting Up Parallel Debugging Sessions

This chapter explains how to set up TotalView parallel debugging sessions for applications that use the following parallel execution models.

The information in this chapter describes running many different environments on many different architectures. While there is a lot of information in this chapter you do need, you probably don't need the information on many of the environments and architectures. This means that you shouldn't just read this chapter. Instead, go to this book's table of contents and decide what's important to you.

This chapter discusses:

There are a few things that are of general interest:

- TotalView lets you decide which process you want it to attach. You will find information in *"Attaching to Processes"* on page 110.
- If you're using a messaging system, TotalView displays this information visually as a message queue graph and textually in a message queue window.
- The end of this chapter has some hints on how you can approach debugging parallel programs.

Debugging MPICH Applications

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.3 or later on a homogenous collection of machines. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at www.mcs.anl.gov/mpi. (You are strongly urged to use a later version of MPICH. Information on versions that work with TotalView can be found in the *TotalView Platforms* document.)

The MPICH library should use the ch_p4, ch_p4mpd, ch_shmem, ch_lfshmem, or ch_mpl devices. For networks of workstations, ch_p4 is the default. For shared-memory SMP machines, use ch_shmem. On an IBM SP machine, use the ch_mpl device. The MPICH source distribution includes all of these devices and you can choose one 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 1.2.3 distribution.*

Topics in this section are:

Starting TotalView on an MPICH Job

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

At Version 1.1.2, here's the command line that starts a job under TotalView's 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.

At Version 1.2.4, this changes to:

mpirun –dbg=totalview [*other_mpich-arguments*] *program* \
[*program-arguments*]

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 different versions of TotalView or pass command-line options to TotalView.

For example, here 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 machine 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 Job*" on page 82.

Starting TotalView using ch_p4mpd is similar to starting TotalView using poe on IBM or other methods you might use on Sun and HP platforms. In general, you start TotalView using the totalview command. Here's the syntax;

totalview mpirun [*totalview_arguments*] \
–a [*mpich-arguments*] *program* [*program-arguments*]

CLI: totalviewcli mpirun [*totalview_arguments*]\
–a [*mpich-arguments*] *program* \
[*program-arguments*]

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 your answer is 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's Attached Page to show these newly acquired processes. For more information, see *"Attaching to Processes"* on page 110.

Attaching to an MPICH Job

TotalView allows you to attach to an MPICH application even if it was not started under TotalView's control. Here is the procedure:

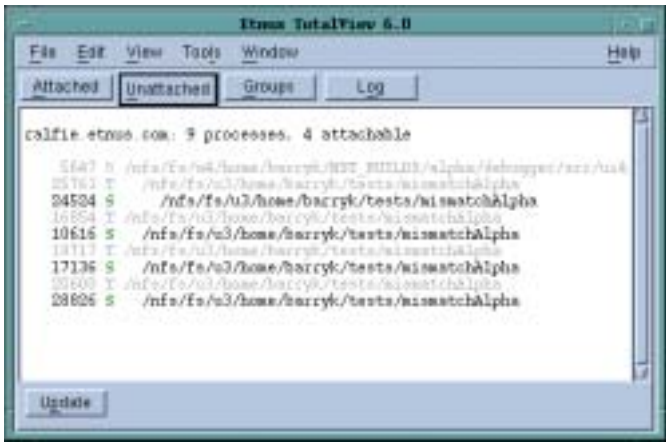
1. Start TotalView.
2. The Root Window's Unattached Page displays the processes that are not yet owned.

   CLI: dattach *executable pid*

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

   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 Job*" on page 82. Figure 65 shows the Unattached window after some attaching has occurred.

Figure 65: Root Window: Unattached Page



Normally, the first MPICH process is the highest process with the correct image 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_lfshmem.
- Additional slave processes if MPICH was configured with ch_p4 and have a machine file that places multiple processes on the same machine.

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

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

As an alternative, you can use the Group > Attach Subsets command to predefine what TotalView should do. For more information, see *"Attaching to Processes"* on page 110.

In some situations, the processes you expect to see may not exist (for example, they may have crashed or exited). TotalView acquires all the processes it can and then warns you if it could 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.

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 treats it as different executable, which causes debugging problems.

The following example uses the *same* absolute path name on TotalView's 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:

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

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

Debugging HP Tru64 Alpha MPI Applications

You can debug HP Alpha MPI applications on the HP Alpha platform. To use TotalView with HP Alpha MPI, you must use HP Alpha MPI version 1.7 or later.

Starting TotalView on a HP Alpha MPI Job

HP Alpha MPI programs are most often started with the dmpirun command. You would use very similar command when starting an MPI program under TotalView's control:

{ totalview | totalviewcli } dmpirun –a *dmpirun-command-line*

This 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, you can use the Process > Go command to let the program run.

CLI: dfocus p dgo

The dmpirun command runs and starts all MPI processes. TotalView will also acquire them and ask if you want to stop them.

Problems can occur if you rerun HP Alpha MPI programs that are under TotalView control due to resource allocation issues within HP Alpha MPI. Consult the HP Alpha MPI manuals and release notes for information on using mpiclean to clean up the MPI system state.

Attaching to a 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 42. You can also use the Group > Attach Subset command with is discussed in *"Attaching to Processes"* on page 110.

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

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

Debugging HP MPI Applications

You can debug HP MPI applications on a PA-RISC 1.1 or 2.0 processor. To use TotalView with HP MPI, 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 three ways:

{ totalview | totalviewcli } *program* –a *mpi-arguments*

This command tells TotalView to start the MPI process. TotalView will then show you the machine code for the HP MPI mpirun executable.

CLI: dfocus p dgo

mpirun *mpi-arguments* –tv –f *startup_file*

This command tells MPI that it should start TotalView and then start the MPI processes as they are defined within the *startup_file* script. This file names the processes that MPICH will start. Typically, this file has contents that are similar to:

```
-h localhost –np 1 sendrecv
-h localhost –np 1 sendrecva
```

In this example, sendrecv and sendrecva are two different executable programs.

Your HP MPI documentation describes the contents of this startup file.

mpirun *mpi-arguments* –tv *program*
This command tells MPI that it should start TotalView.

Just before mpirun starts the MPI processes, TotalView acquires them and asks if you want to stop the processes before they start executing. If your answer is 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 42.

After TotalView attaches to the HP MPI mpirun process, it displays the same dialog as it does with MPICH. (See step 4 on page 78 of *"Attaching to an MPICH Job"* on page 78.)

Debugging IBM MPI Parallel Environment (PE) Applications

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 running release 3,1 or later of the Parallel Environment for AIX.

See "*Displaying the Message Queue Graph Window*" on page 88 for message queue display information.

Topics in this section are:

Preparing to Debug a PE Application

The following sections describe what you must do before TotalView can display a PE application.

Using Switch-Based Communication

If you're using switch-based communications (either "IP over the switch" or "user space") on an SP machine, you must configure your PE debugging session so that TotalView can use "IP over the switch" for communicating with the TotalView Debugger Server (tvdsvr). Do this by setting adapter_use to shared and cpu_use 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 in 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; to be safe, set them explicitly by using one of these techniques.

When you're using switch-based communications, you must run TotalView on one of the SP or SP2 nodes. Since TotalView will be using "IP over the switch" in this case, you cannot run TotalView on an RS/6000 workstation.

Performing Remote Logins

You must be able to perform a remote login using the rsh command. You will 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 Debugger 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 Job

Here is the syntax for running Parallel Environment (PE) programs from the command line:

program [*arguments*] [*pe_arguments*]

You can use the poe command to run programs:

poe *program* [*arguments*] [*pe_arguments*]

If, however, you start TotalView on a PE application, you must start poe as TotalView's target. The syntax for this is:

{ totalview | totalviewcli } poe –a *program*[*arguments*] [*PE_arguments*]

For example:

```
totalview poe –a sendrecv 500 –rmpool 1
```

Setting Breakpoints

After TotalView is running, you can start the poe process; this process will then start the program's parallel processes with 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 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 Points > Save All command and 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's Group > Go command.

CLI: dfocus G dgo
Abbreviation: G

No parallel tasks will reach the first line of code in your main routine until all parallel tasks start.

You should 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. Once you allow the parallel processes to proceed into the MPI_Init() or MPL_Init() call, you should 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 115.

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 its various nodes. This set of acquired processes will include the processes you want to debug.

Attaching from a Node Running poe

Here's the procedure for attaching 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 TotalView Debugger Server (tvdsvr) command line

before attaching to poe. See "*Using the Single-Process Server Launch Command*" on page 66.

2 In the Root Window's Unattached Page, find the poe process list, and attach to it by diving into it. When necessary, TotalView launches TotalView Debugger Servers. TotalView will also update the Root Window's Attached Page and open 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 may 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 not running poe is essentially the same as the procedure for attaching from a node 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 your Root Window's Attached Page and you won't be able to start it by diving into it.

The procedure for placing poe within this list is:

1 Connect TotalView to the startup node. For details, see "*Setting Up and Starting the TotalView Debugger Server*" on page 61 and "*Attaching to Processes*" on page 42.

2 Select the Root Window's Unattached Page, and then invoke the Window > Update command.

3 Look for the process named poe and continue as if attaching from a node running poe.

CLI: dattach -r *hostname* poe *poe-pid*

Debugging LAM/MPI Applications

The following is a description of the LAM/MPI implementation of the MPI standard. This is the first two paragraphs of Chapter 2 of he "LAM/MPI User's Guide, version 7.0. The URL for this document is: http://www.lam-mpi.org/download/files/7.0-user.pdf.

LAM/MPI us 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. (http://www.lam-mpi.org).

> 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. ...

The way in which you debug a LAM/MPI program is similar to the way you debug most MPI programs. Here's the syntax if TotalView is in your path:

```
mpirun -tv mpirun args prog prog_args
```

As an alternative, you could invoke TotalView upon mpirun:

```
totalview mpirun -a prog prog_args
```

Pages 79-83 of the LAN/MPI User's Guide discusses how you would use TotalView to debug LAN/MPI programs.

Debugging QSW RMS Applications

TotalView supports automatic process acquisition on AlphaServer SC systems and 32-bit Red Hat Linux systems that use Quadrics's 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's control, use TotalView as though you were debugging prun:

```
{ totalview | totalviewcli } prun –a prun-command-line
```

TotalView starts up 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 processes the same way you attach to other processes. For details on attaching to processes, see *"Attaching to Processes" on page 42*.

After you attach to the RMS prun process, TotalView asks if you also wish to attach to slave MPICH processes. If you do, press Return or choose Yes. If you do not, select No.

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

As an alternative, you can use the Group > Attach Subsets command to predefine what TotalView should do. For more information, see *"Attaching to Processes"* on page 110.

Debugging SGI MPI Applications

TotalView can acquire processes started by SGI MPI, which is part of the Message Passing Toolkit (MPT) 1.3 and 1.4 packages.

Message queue display is supported by release 1.3 and 1.4 of the Message Passing Toolkit. See "*Displaying the Message Queue Graph Window*" on page 88 for message queue display.

Starting TotalView on a SGI MPI Job

SGI MPI programs are normally started by using the mpirun command. You would use a similar command to start an MPI program under TotalView's control:

{ 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 showing 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 job, attach to the SGI 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 42.

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

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

As an alternative, you can use the Group > Attach Subsets command to predefine what TotalView should do. For more information, see *"Attaching to Processes"* on page 110.

Debugging Sun MPI Applications

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*.

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 that says:

```
Process mprun is a parallel job. Do you want to stop
the job now?
```

2 If you had compiled using the –g option, clicking Yes tells TotalView to open a Process Window showing your source. All processes will be halted.

Attaching to a Sun MPI Job

This section describes how 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, refer to the mpps(1M) manuel page.

Here 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

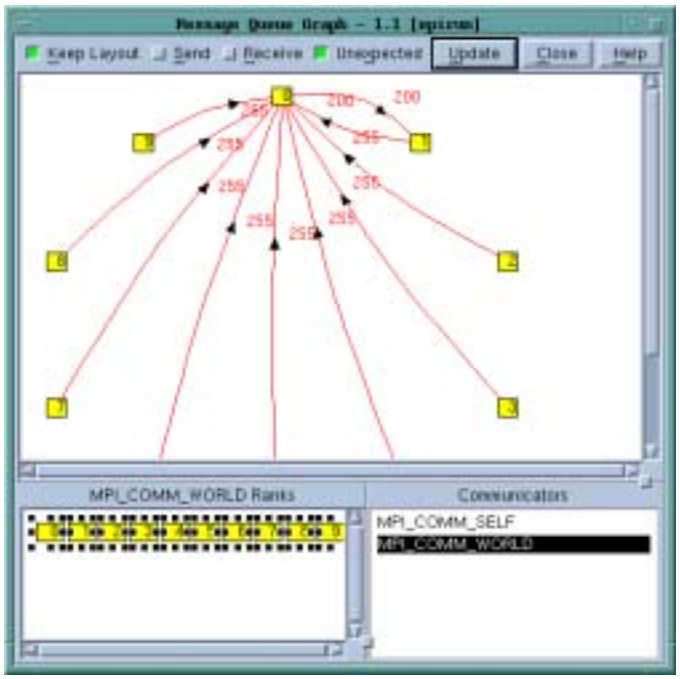
3 If TotalView is running on a different node than the mprun job, enter the host name in the Remote Host field.

CLI: dattach –r *host-name* mprun *mprun-pid*

Displaying the Message Queue Graph Window

TotalView can graphically display your MPI program's message queue state. If you select the Process Window's Tools > Message Queue Graph command, TotalView displays a window with a large empty area. After you select the ranks to be monitored, the kind of messages, and message states, TotalView updates this window to show the current queue state. Figure 66 shows a sample window.

Figure 66: Tools > Message Queue Graph Window



The numbers in the boxes indicate the MPI message tag number. Diving on a box tells TotalView to open a Process Window for that process.

The numbers next to the arrows indicate the number of messages that existed when TotalView created the graph. Diving on an arrow tells TotalView that it should display its Tools > Message Queue Window, which will have detailed information about the messages. A grey box indicates a process to which TotalView is not attached.

The colors used to draw the lines and arrows have the following meaning:

- Green: sent messages
- Blue: receive messages
- Red: unexpected messages

This graph shows you the state of your program at a particular instant. Selecting the Update button tells TotalView that it should update the display.

While you can use this window in many ways, here are some to consider:

- 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 indicated 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 it looks wrong.
- You can change the shape of the graph by dragging either nodes or the arrows. This is often useful when you're comparing sets of nodes and their messages with one another. TotalView doesn't remember the places to which you have dragged the nodes and arrows. This means that if you select the Display button after you arrange the graph, your changes are lost.

Topics related to this one are:

- "*Message Queue Display Overview*" on page 89
- "*Using Message Operations*" on page 90
- "*OpenMP Stack Parent Token Line*" on page 97

Displaying the Message Queue

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

To use the message queue display feature, you must be using one of the following versions of MPI:

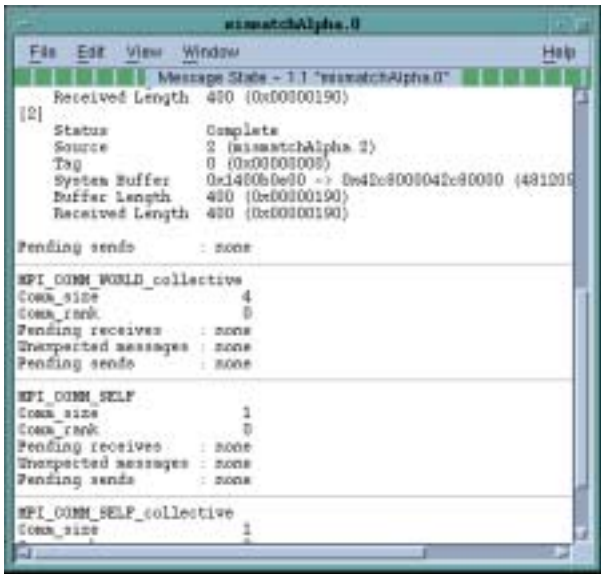
- MPICH version 12.3 or later.
- HP Alpha MPI (DMPI) version 1.8, 1.9, and 1.96.
- HP HP-UX version 1.6 and 1.7.
- IBM MPI Parallel Environment (PE) version 3.1 or 3.2, but only programs using the threaded IBM MPI libraries. MQD is not available with earlier releases, or with the non-thread-safe version of the IBM MPI library. Therefore, to use TotalView MQD with IBM MPI applications, you must use the mpcc_r, mpxlf_r, or mpxlf90_r compilers to compile and link your code.
- For the SGI MPI TotalView message queue display, you must obtain the Message Passing Toolkit (MPT) release 1.3 and 1.4. Check with SGI for availability.

Message Queue Display Overview

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. The information is shown in Figure 67 on page 90.

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

Figure 67: Message Queue Window



tor structures, TotalView displays both of them. One will be used for point-to-point operations and the other for collective operations.

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 can be displayed.

Topics in this section are:

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.
- If a Process Window doesn't exist, TotalView creates a new one for the process.

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's length and the data's alignment. If TotalView guesses incorrectly, you can edit the type field in the Variable Window.

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

Pending Receive Operations

TotalView displays each pending receive operation in the Pending receives list. Figure 68 shows an example of an MPICH pending receive operation.

Figure 68: Message Queue Window Showing Pending Receive Operation

❶ Operation index
❷ One receive operation
❸ Diving here displays a Process Window
❹ Diving here displays a Variable Window

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

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 already received messages 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.

Pending Send Operations

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

MPICH does not normally keep information about pending send operations. However, when you configure MPICH, you can tell it to maintain a list

of them. Start your program under TotalView's control and use mpirun –ksq, or –KeepSendQueue to see these messages.

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
```

MPI Debugging Troubleshooting

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 machines in your machines 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 Debugger Server (tvdsvr) fail to start?

  tvdsvr 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 > Delete command to restart 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.

Debugging OpenMP Applications

TotalView supports many OpenMP C and Fortran compilers. Supported compilers and architectures are listed in the *TotalView Platforms* document and on our Web site.

Here 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

Debugging OpenMP code is very similar to debugging multithreaded code, differing only in that the OpenMP compiler makes the following special code transformations:

- The most visible transformation is *outlining*. The compiler pulls the body of a parallel region out of the original routine and places it into an *outlined routine*. In some cases, the compiler will generate 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.
- The compiler inserts calls to the OpenMP runtime library.
- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables are maintained in the master thread's original routine, and private variables are maintained 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.

TotalView OpenMP Features

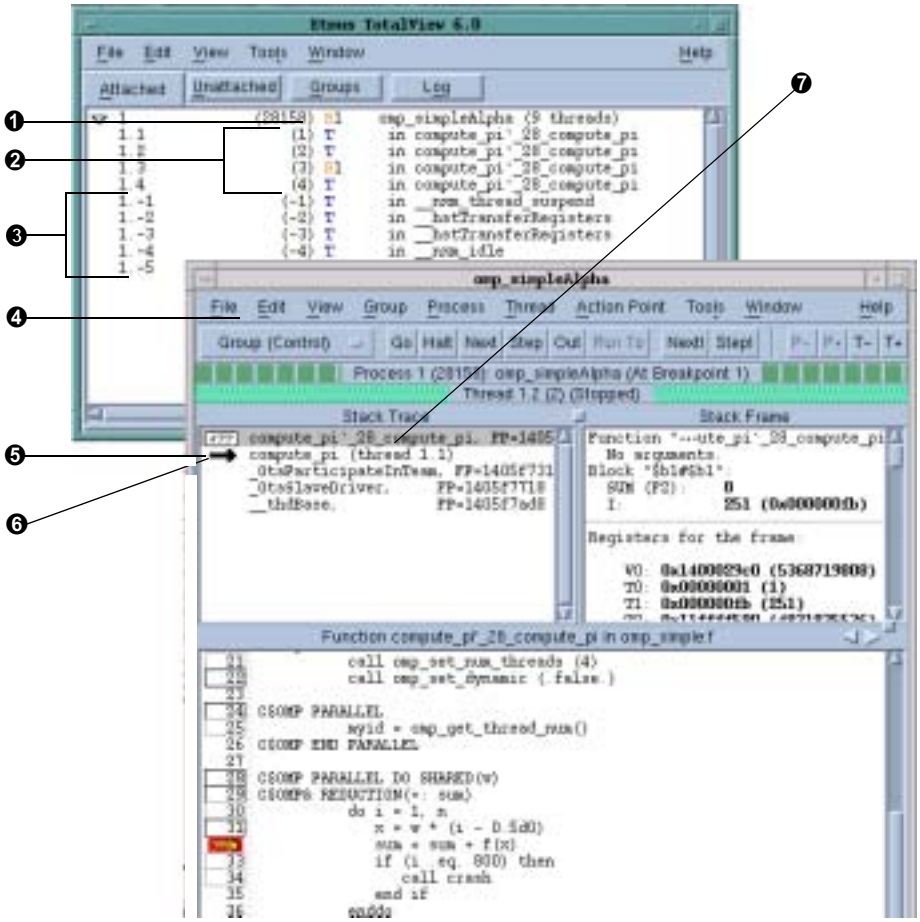
TotalView makes these transformations visible in the debugging session. Here are some things you should know:

- The compiler may 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 allow the process to run to it. Once inside a parallel region, you can single step within it.
- OpenMP programs are multithreaded programs, so the rules for debugging multithreaded programs apply.

Figure 69 shows a sample OpenMP debugging session.

Figure 69: Sample OpenMP Debugging Session



❶ OpenMP master thread
❷ OpenMP worker threads
❸ Manager threads (don't touch)
❹ Slave Thread Window
❺ "Original" routine name
❻ Stack parent token (select or dive to view master)
❼ "Outlined" routine name

OpenMP Platform Differences

The following list contains information on platform differences:

- On HP Alpha Tru64 UNIX and on the Guide compilers, the OpenMP threads are implemented by the compiler as pthreads, and on SGI IRIX as sprocs. TotalView shows the threads' logical and/or system thread ID, not the OpenMP thread number.
- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.

- In HP Alpha Tru64 UNIX, the system manager threads have a negative thread ID; as they do not take part in your OpenMP program, you should never manipulate them.
- 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*.
- When you stop the OpenMP master thread in a PARALLEL DO outlined routine, the stack backtrace shows the following call sequence:
  - The outlined routine called from.
  - The OpenMP runtime library called from.
  - The original routine (containing the parallel region).
- 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.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.

OpenMP Private and Shared Variables

TotalView allows you to 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 232. 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, as follows:

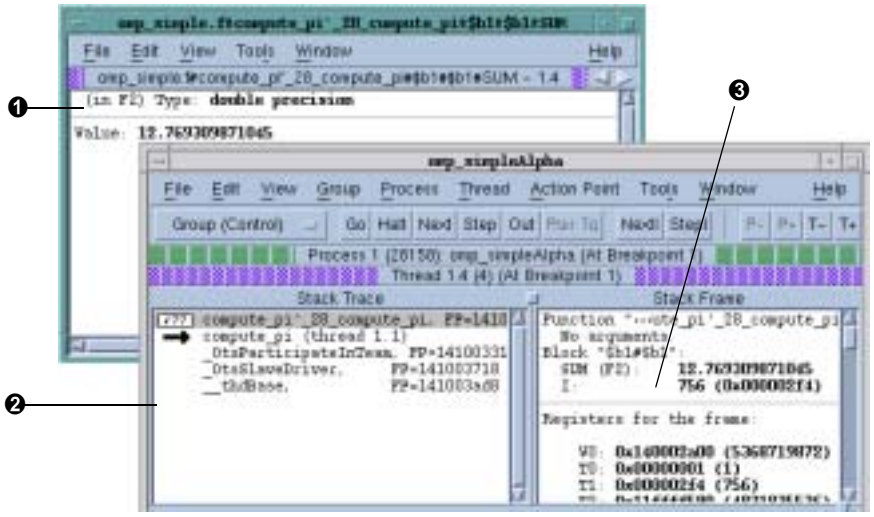
1. Select the outlined routine in the Stack Trace Pane; or select the original routine stack frame in the OpenMP master thread.
2. Dive on the variable name, or select the View > Lookup Variable command. When prompted, enter the variable name.

CLI: dprint
You will need to set your focus to the OpenMP master thread first.

TotalView will open a Variable Window displaying the value of the OpenMP shared variable, as shown in Figure 70 on page 96.

Shared variables are stored on 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.

Figure 70: OpenMP Shared Variable



❶ OpenMP shared variables have master thread's context
❷ Original routine's stack frame selected
❸ Stack Frame Pane includes shared variables

You can also view OpenMP shared variables in the Stack Frame Pane by selecting the original routine stack frame in the OpenMP master thread, or by selecting the stack parent token line in the Stack Trace Pane of OpenMP worker threads, as shown in Figure 70.

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.

On SGI, the THREADPRIVATE variables are mapped to the same virtual address. However, they have different physical addresses.

Here's how you can view a variable in an OpenMP THREADPRIVATE common block, or the OpenMP THREADPRIVATE common block itself:

1. In the Threads Pane of the Process Window, select the thread containing the private copy of the variable or common block you would like to view.
2. In the Stack Trace Pane of the Process Window, select the stack frame that will allow you to 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 (_) after the common block name.

CLI: dprint

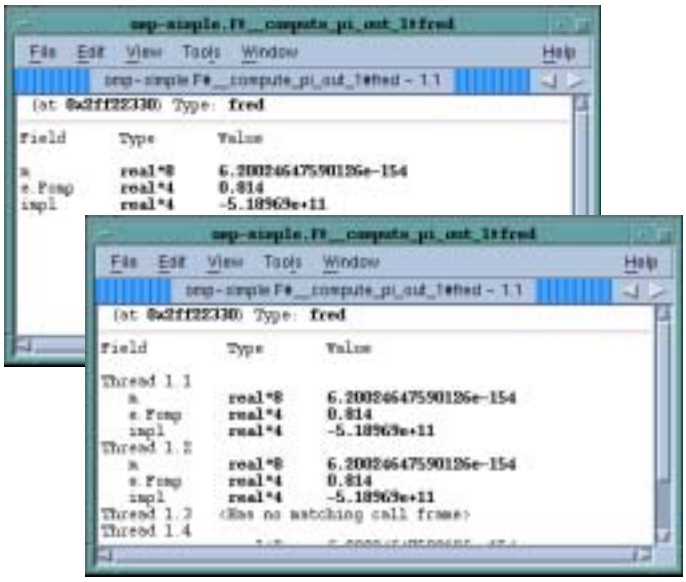
TotalView opens a Variable Window displaying the value of the variable or common block for the selected thread.

See "*Displaying Variables*" on page 230 for more information on displaying variables.

4 To view OpenMP THREADPRIVATE common blocks or variables across all threads, you can use the Variable Window's View > Laminate Threads command. See "*Displaying a Variable in All Processes or Threads*" on page 270.

Figure 71 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. In the laminated views, the values of the common block across all threads are displayed.

Figure 71: OpenMP THREADPRIVATE Common Block Variables



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 72 on page 98.)

This context includes the OpenMP shared variables.

Figure 72: OpenMP Stack Parent Token Line

Debugging Global Arrays Applications

Here's the description contained on the Global Arrays home page (http://www.emsl.pnl.gov:2080/docs/global/ga.html):

> 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. The way in which you debug a Global Arrays program is basically identical to the way you would debug any other multiprocess program. The one difference is that you will use the Tools > Global Arrays command to display information about your global data.

Here are some of the unique activities you can perform:

- 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 either C or Fortran format.

- Cast the data so that TotalView will interpret data as a global array handle. This means that TotalView will display 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 within the current context.

Within a Variable Window, the commands that operate upon a local array such as slicing, filtering, obtaining statistics, and visualization also operate upon global arrays.

The command you would use to start TotalView depends upon your operating system. For example, here's an example of starting TotalView upon a program that would normally be invoked using prun and which would use three processes:

```
totalview prun -a -N 3 boltz.x
```

Before your program starts parallel execution, TotalView asks if you want to stop the job.

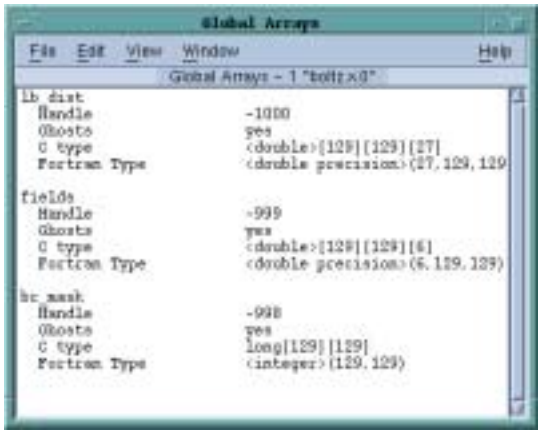
Figure 73: Question Window for Global Arrays Program

Select Yes if you want to set breakpoints or inspect the program before it begins execution.

After your program hits a breakpoint, use the Tools > Global Arrays command to begin inspecting your program's global arrays. Figure 74 shows the window that TotalView displays.

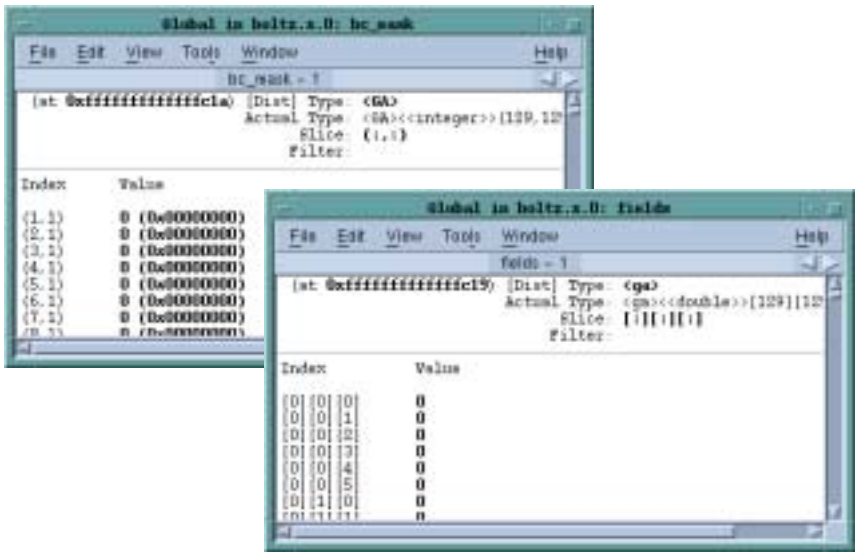
CLI: dga

Figure 74: Tools > Global Arrays Window



The arrays named in this window are displayed using their C and Fortran type names. Diving on the line containing the type definition tells TotalView to display Variable Windows containing information about that array. (See Figure 75.)

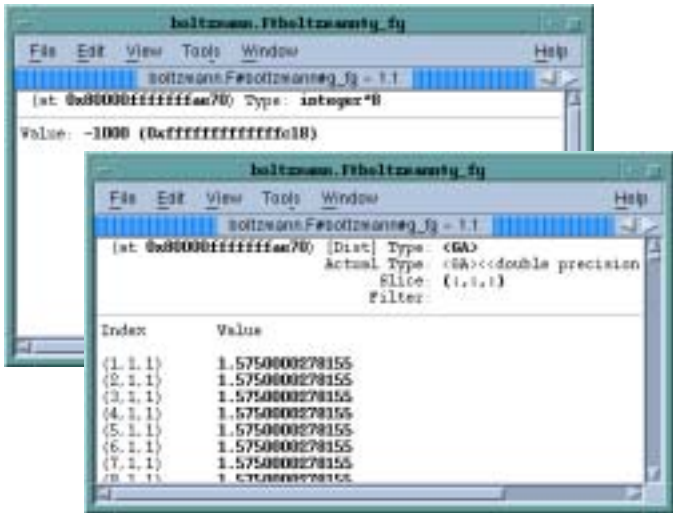
Figure 75: Fortran and C Variable Windows



After TotalView displays this information, you can use other standard command and operations upon 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 will not know that it is a component of a global array. If you do this, you can cast the variable into a global array using either <ga> for a C language cast or <GA> for a Fortran cast. Figure 76 shows a Variable Window before and after the data was cast.

Figure 76: 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 "*Parallel Debugging Tips*" on page 110.

Topics in this section are:

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 run:

- One 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 machine or same machine architecture.

  One user can start TotalView to debug the same PVM or DPVM application on different machine architectures. However, a single user can't have multiple instances of TotalView debugging the same PVM or DPVM session on a single machine architecture.

  For example, suppose you start a PVM session on Sun 5 and HP Alpha machines. You must start two TotalView sessions: one on the Sun 5 machine to debug the Sun 5 portion of the PVM session, and one on the HP Alpha machine to debug the HP Alpha portion of the PVM session. These two TotalView sessions are separate and don't interfere with one another.
- Similarly, in one TotalView session, you can run either a PVM application or a DPVM application, but not both. However, if you run TotalView on a 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 $HOME/pvm3/bin/$PVM_ARCH/tvdsvr) to the TotalView Debugger Server (tvdsvr). With this link in place, TotalView invokes pvm_spawn() to spawn the tvdsvr tasks.

For example, if tvdsvr is installed in the /opt/totalview/bin directory, enter the following command:

```
ln -s /opt/totalview/bin/tvdsvr \
            $HOME/pvm3/bin/$PVM_ARCH/tvdsvr
```

If the symbolic link doesn't exist, TotalView can't spawn tvdsvr. When TotalView can't spawn tvdsvr, 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 following steps outline this procedure.

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, refer to "*Examining Groups*" on page 180.
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 a CLI variable. For more information, refer to "*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 will vary, but should always contain the current directory and your home directory.

   You can set the directory search path by setting the TV::search_path variable or you can use the File > Search Directory command. Refer to "*Setting Search Paths*" on page 50 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's File > Signals command. Refer to "*Handling Signals*" on page 48 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.

Continue with *"Automatically Acquiring PVM/DPVM Processes"* on page 103.

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.

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. You can enable DPVM support in two ways. The first uses the TV::dvpm CLI variable. As an alternative, you can add the –dpvm command-line option to the totalview command. This option enables DPVM support.

   The command-line options override the TV:dpvm command variable. For more information on the totalview command, refer to "*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's File > Signals command in TotalView. Refer to "*Handling Signals*" on page 48 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 can 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

This section describes how TotalView automatically acquires PVM and DPVM processes in a PVM or DPVM debugging session. Specifically, TotalView uses the PVM tasker to intercept pvm_spawn() calls.

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 Debugger Server (tvdsvr) 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:

  ```
  Spawning TotalView Debugger Server onto PVM
  host 'host'
  ```

If you add a host with a compatible machine architecture to your PVM or DPVM debugging session after you start TotalView, TotalView automatically starts a debugger server on that host.

After all debugger servers are running, TotalView will intercept 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 machine architecture, TotalView ignores that task.

When TotalView receives a PVM or DPVM tasker event, it takes the following actions:

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 machines, 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 will need to use the View > Lookup Function command to display the source code for the main() routine.

   CLI: dlist *function-name*

   For more information on this command, refer to "*Finding the Source Code for Functions*" on page 173.

Attaching to PVM/ DPVM Tasks

You can attach to a PVM or DPVM task if the task meets the following criteria:

- The machine architecture on which the task is running is the same as the machine architecture on 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.

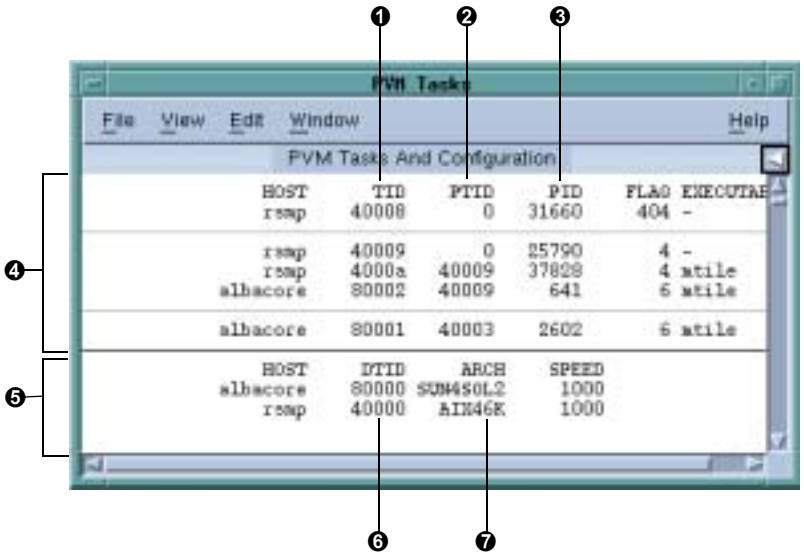
- The executable name must be known. If the executable name is listed as a dash (–), TotalView cannot determine the name of the executable. (This can occur if a task was not created with the pvm_spawn() call.)

To attach to a PVM or DPVM task, complete the following steps:

1. Select Tools > PVM Tasks command from TotalView's Root Window.

   The PVM Tasks is displayed, as shown in Figure 77. This window displays current information about PVM tasks and hosts—TotalView automatically updates this information as it receives events from PVM.

Figure 77: PVM Tasks and Configuration Window



❶ Task ID (TID)
❷ Parent TID
❸ UNIX Process ID (PID)
❹ Tasks
❺ Hosts
❻ Daemon TID
❼ Machine Architecture

Since PVM doesn't always generate an event that allows TotalView to update this window, you should use the Windows > 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 Figure 77 because flag 4 is set. In contrast, you can't attach to the tvdsvr and – (dash) executables 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.

Reserved Message Tags

TotalView uses PVM message tags in the range 0xDEB0 through 0xDEBF to communicate with PVM daemons and the TotalView Debugger Server. Avoid sending messages that use these reserved tags.

Cleanup of Processes

The pvmgs process registers its task ID in the PVM database. If the pvmgs process is terminated, the pvm_joingroup() routine hangs because PVM won't clean up the database. If this happens, you must terminate and then restart the PVM daemon.

TotalView attempts to clean up the TotalView Debugger Server daemons (tvdsvr), 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 the SGI IRIX logically shared, distributed memory access (SHMEM) library.

To debug a SHMEM program, follow these steps:

1. Link it 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 3, "*Setting Up a Debugging Session*," on page 35.
3. Set at least one breakpoint after the call to the start_pes() SHMEM routine. (This is illustrated in Figure 78 on page 107.)

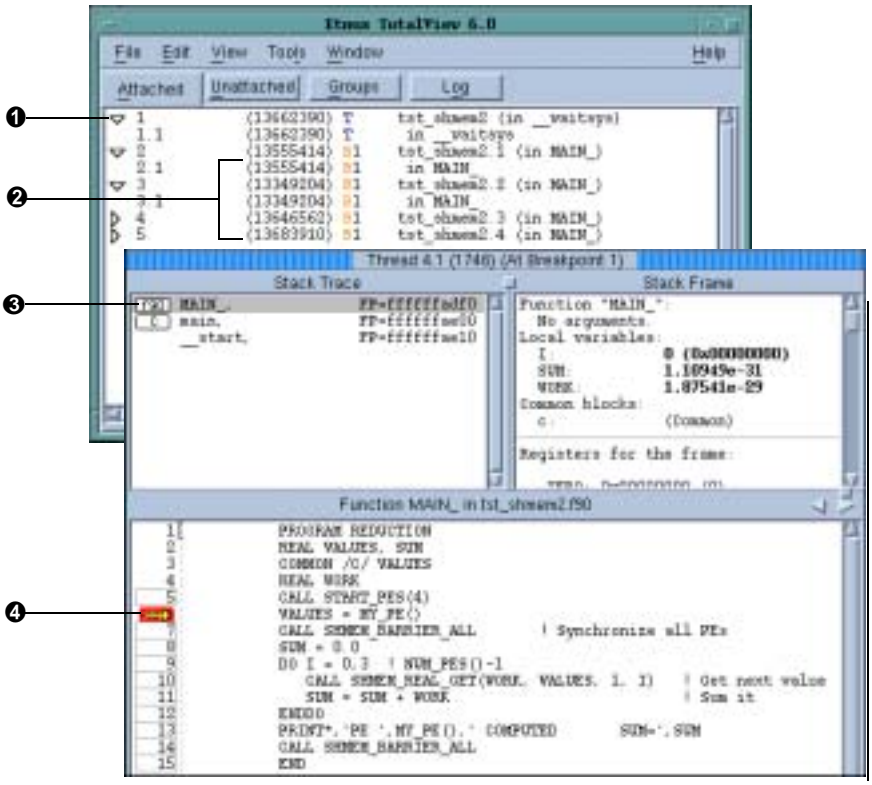
You cannot single-step over the call to start_pes().

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 UPC-specific features of TotalView. It not an introduction to the UPC language. If you're looking for an introduction, you'll find information at http://www.gwu.edu/~upc.

Figure 78: SHMEM Sample Session



When debugging UPC code, TotalView requires help from a UPC assistant library that your compiler vendor provides. You may need to include the location of this library in your LD_LIBRARY_PATH *variable. Etnus also provides assistants that you can use. You can find these assistants at http://www.etnus.com/Products/TotalView/developers/index.html*

Topics in this section are:

Invoking TotalView

Here's how to invoke TotalView upon UPC programs:

- When running on an SGI system using the gcc UPC compiler, invoke TotalView upon your UPC program in the same way as you would invoke it on most other programs. For example:

```
totalview prog_upc -a args_to_foo_upc
```

- When running on HP Compaq SC machines, debug your UPC code in the same way that you would debug other kinds of parallel code. For example:

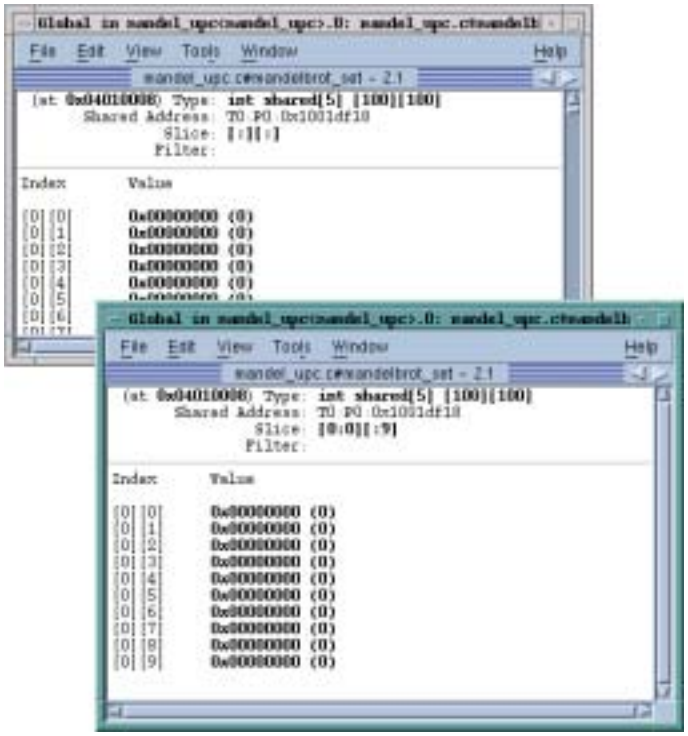
```
totalview prun -a -n node_count prog_upc \
   args_to_prog_upc
```

Viewing Shared Objects

Totalview displays UPC shared objects, and will fetch 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 figure in Figure 79 displays elements of a large shared array. You can manipulate an examine shared arrays the same as any other array. For example, you can slice, filter, obtain statistical information on, and so forth. (For more information on displaying array data, see Chapter 13, "*Examining Arrays*," on page 259) The bottom-right illustration shows a 10-element slice of this array.

Figure 79: A Sliced UPC Array



In this illustration, TotalView displays the value of a pointer-to-shared variable whose target is the array in the Shared Address area. As usual, the address within the process is shown in the top left of the display.

As the array is shared, it has an additional property: the element's affinity. You can display this information if you select the Variable Window's View > Node Display command. This command tells TotalView to add a column that indicates the node associated with the value. This is shown in the Figure 80 on page 109.

You can also use the Tools > Visualize Distribution command to visualize this array. For more information on visualization, see "*Visualizing Array Data*" on page 130.

Figure 80: UPC Variable Window Showing Nodes

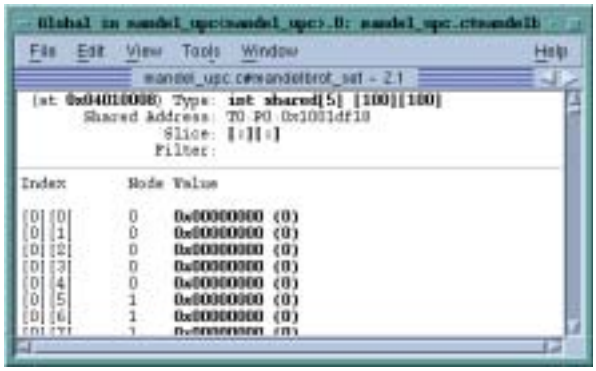
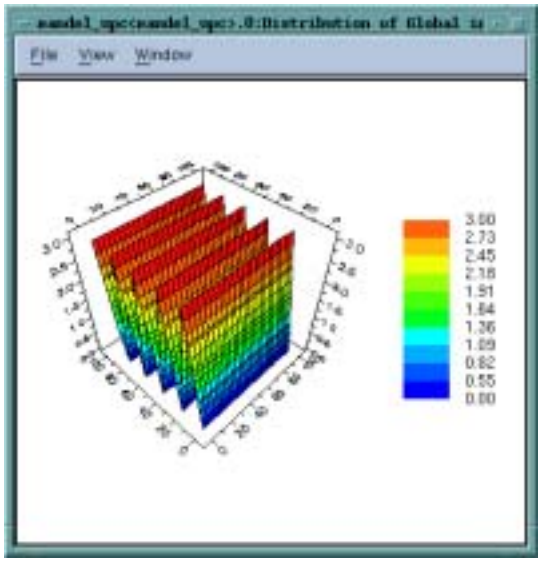


Figure 81: Laminated UPC Variable Window



Pointer to Shared

TotalView understands pointer-to-shared data and displays the components of the data, as well as the target of the pointer to shared. For example, Figure 82 shows what is displayed:

Figure 82: Pointer to a Shared Variable

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 5.

In this figure, TotalView also displays the upc_threadof ("T0"), the upc_phaseof ("P1"), and the upc_addrfield (0x0x1001df30) 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. TotalView will fetch the target of the pointer from the UPC thread with which the pointer has affinity.

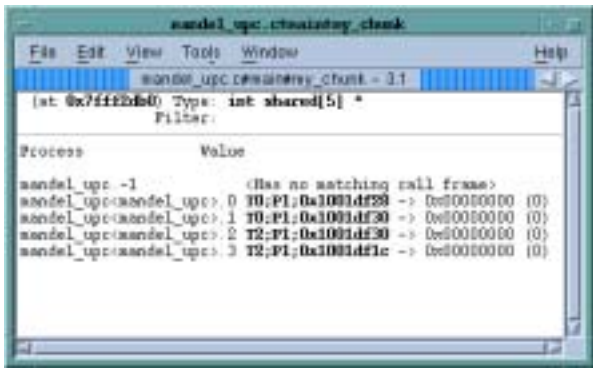
You can update the pointer by selecting the pointer value and editing the thread, phase, or address values. If the phase is corrupt, you'll see something like the following in the Value field:

```
Value: 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 Tools > Laminate command to display the value from each of the UPC threads.

Figure 83: UPC Laminated Variable



Parallel Debugging Tips

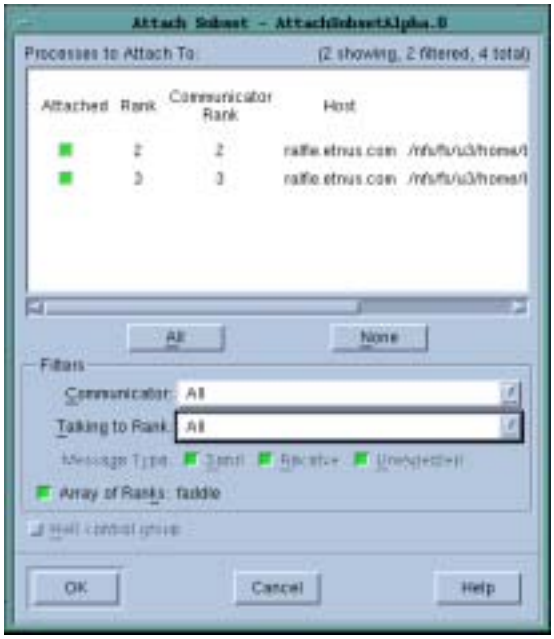
This section contains some information that you may find useful when debugging parallel programs. The topics in this section are:

Attaching to Processes

In a typical multiprocess 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 the processes that you program starts. If there are a lot of processes, there may be considerable overhead involved in opening and communicating with the jobs.

You can minimize this overhead by using the Group > Attach Subsets command, which displays the dialog box shown in Figure 84.

Figure 84: Group > Attach Subset Dialog Box



By selecting the boxes at the left side of the list, you tell TotalView which processes it should attach to. So, while your program will launch all of these processes, TotalView will only attach to the processes that you have selected here.

The four controls underneath the All and the None buttons let you limit what TotalView automatically attaches to.

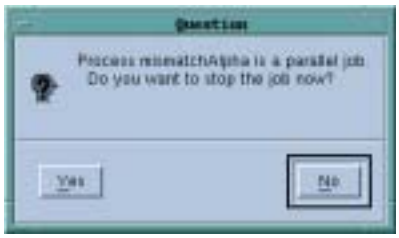
- The Communicator control specifies that the processes must be involved with the communicators that you select. For example, if something goes wrong that involves a communicator, selecting it from the list tells TotalView that it should only attach to the processes that use that communicator.
- The 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. Two other entries are useful: All and MPI_ANY_SOURCE.
- The three checkboxes in the Message Type area add yet another qualifier. Checking a box tells TotalView that it should only display communicators that are involved with a Send, Receive, or Unexpected messages.

After you've found the problem, you can detach from these nodes by selecting None. In most cases, you would use the All button to set all the check boxes, then clear the ones that you're not interested in.

Many applications place the ranks numbers in a variable so they can be referred to easily. 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

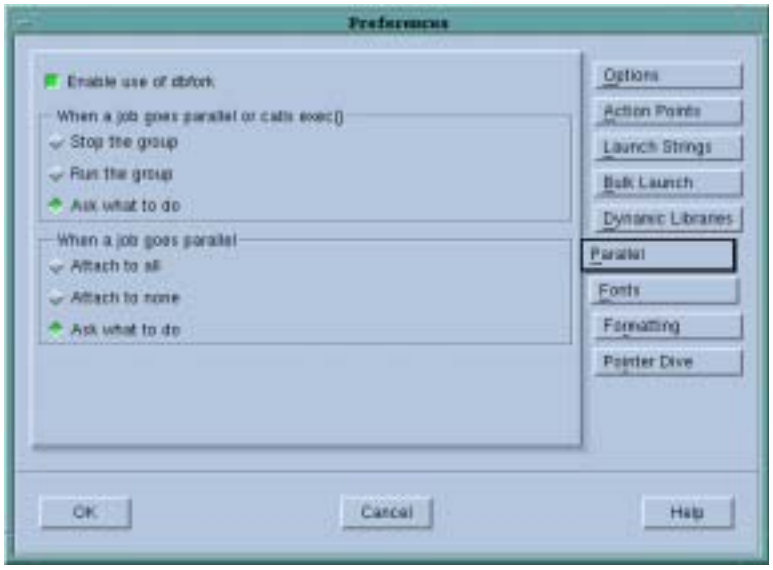
While you can use the Group > Attach command at any time, you would probably use it immediately before TotalView launches processes. Unless you have set preferences otherwise, TotalView will stop and ask if you want it stop your processes. When selected, the Halt control group check box also tells TotalView that it stop a process just before it begins executing. (See Figure 85.)

Figure 85: Stop Before Going Parallel Question Dialog Box



The commands on the Parallel Page with the File > Preferences Dialog Box let you control what TotalView will do when your program goes parallel. (See Figure 86.)

Figure 86: File > Preferences: Parallel Page



The When a job goes parallel or calls exec() radio buttons have the following meanings:

- Stop the group: Stops the control group immediately after the processes are created.
- Run the group: Allows all newly created processes in the control group to run freely.
- Ask what to do: Asks what should occur. If you select this option, TotalView will ask if it should start the created processes.

CLI: dset TV::parallel_stop

The When a job goes parallel radio buttons have the following meaning:

- Attach to all: TotalView automatically attaches to all processes when they begin executing.
- Attach to none: TotalView will 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 Subsets. TotalView will then attach to the processes that you have selected. Note that this dialog box isn't displayed when you set the preference. Instead, it controls what will happen when your program creates parallel processes.

CLI: dset TV::parallel_attach

General Parallel Debugging Tips

Here are some tips that are useful for debugging most parallel programs:

- **Breakpoint behavior**

  When you're debugging message-passing and other multiprocess 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 multiprocess program hits a breakpoint, TotalView will stop all the other processes.

  To change the default stopping action of breakpoints and barrier breakpoints, you can set TotalView preferences. Information on these preferences can be found in the online Help. These preferences tell TotalView if it should allow other processes and threads to continue to run when a process or thread hits the breakpoint.

  These options only affect the default behavior. As usual, you can choose a behavior for a breakpoint by setting the breakpoint properties in the File > Preferences's Action Points Page. See "*Setting Breakpoints for Multiple Processes*" on page 280.

- **Process synchronization**

  TotalView has two features that make it easier to get all of the processes in a multiprocess 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 "*Barrier Points*" on page 283.

  The Process Window's Group > Run To command is a special kind of stepping command. It allows you to run a group of processes to a selected source line or instruction. See "*Stepping (Part I)*" on page 199.

- **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: dfocus g dgo
Abbreviation: G

You would then use the Group > Halt command instead of Process > Halt.

CLI: dfocus g dhalt
Abbreviation: H

The group-level single-stepping commands such as Group > Step and Group > Next allow you to single-step a group of processes in a parallel. See "*Stepping (Part I)*" on page 199.

CLI: dfocus g dstep
Abbreviation: S

dfocus g dnext
Abbreviation: N

- **Process-level stepping**

  If you use a process-level single-stepping command in a multiprocess program, TotalView may appear to be hung (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 will display. As an alternative, consider using a group-level single-step command.

CLI: Type Ctrl+C

Etnus 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 TotalView 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's Attached Page is updated to show which processes and threads are executing that line. See "*Displaying Your Program's Thread and Process Locations*" on page 187.

- **Viewing variable values**

  You can view (laminate) 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 270.

- **Restarting from within TotalView**

  You can restart a parallel program at any time. If your program runs too far, you can kill the program by selecting the Group > Delete 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. Startup is faster when you do this because

TotalView doesn't need to reread the symbol tables or restart its server processes since they are already running.

CLI: dfocus g dkill

MPICH Debugging Tips

Here are some debugging tips that apply only to MPICH:

- **Passing options to mpirun**

  You can pass options to TotalView through 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. If you configure MPICH/ch_p4 to use a different start-up mechanism from another process, you will 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 62. For more information about rsh, see "*Using the Single-Process Server Launch Command*" on page 66.

IBM PE Debugging Tips

Here are some debugging tips that 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 may cause a timeout.

  To turn the option off, select the Process Window's Action Point > Properties command while the line with the stop symbol is selected. After the Properties Dialog Box appears, you should select the Process button in the When Hit, Stop area and also select Plant in share group.

  CLI: dbarrier *location* –stop_when_hit process

- **Control the poe process**

  Even though the poe process continues under TotalView control, you should not attempt to start, stop, or otherwise interact with it. Your parallel tasks require that poe continue 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 may run noticeably slower than they would run if you were not debugging them.

In general, the number of processes your are 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 may make hardly any 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 SIGALRMs being sent, thus slowing down the tasks.

Part III: Using the GUI

The two chapters in this part of the Users Guide only contain information that you'll need if you're using TotalView's GUI.

Chapter 6: Using TotalView's Windows
: Describes using the mouse and the more important Windows.

Chapter 7: Visualizing Programs and Data
: Some of TotalView's commands and tools are only useful if you're using the GUI. For example, the Visualizer graphically displays an array's data.

Using TotalView's Windows

6

This chapter introduces you to the TotalView interface and describes:

Using the Mouse Buttons

TotalView uses the buttons on your three-button mouse as follows:

Table 6: Mouse Button Functions

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.
	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 where you can alter the selected item's value.

In some places such as the Stack Trace Pane, selecting a line tells TotalView that it should perform an action. In this pane, TotalView dives on the selected routine. (In this case, *diving* means that TotalView finds the selected routine and show 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 STOP icon instead of a line number.

Selecting the STOP icon a second time deletes the breakpoint. If you change any of the breakpoint's properties are if you had created an evaluation point—this is indicated by an EVAL icon—selecting the icon disables it. For more information on breakpoints and evaluation points, refer to Chapter 14, "*Setting Action Points*," on page 273.

Using the Root Window

The Root Window appears when you start TotalView. If you do not enter a program name when starting TotalView, it is the only window that appears. If you indicate a program name, TotalView also open a Process Window containing the program's source code.

The *Root Window* contains the following four tabbed pages:

- Attached: 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 processes and threads are created, 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 87 on page 121 shows the Attached Page for an executing multithreaded multiprocess program.

  Notice the triangles on the left. If a triangle is pointing right, you can click on it to display the process's threads. If it is pointing down, clicking on it conceals this thread information.

  When debugging a remote process, TotalView displays an abbreviated version of the host name on which the process is running in brackets ([ ]). The full host name appears in brackets in the title bar of the Process Window. In Figure 88 on page 121, the process is running on the machine dewey.etnus.com. This name is abbreviated in the Root Window. This figure also describes the contents of the columns in this window.

  When you dive on a line in this window, TotalView displays the source for the process or thread in a Process Window.

Figure 87: Root Window Attached Page

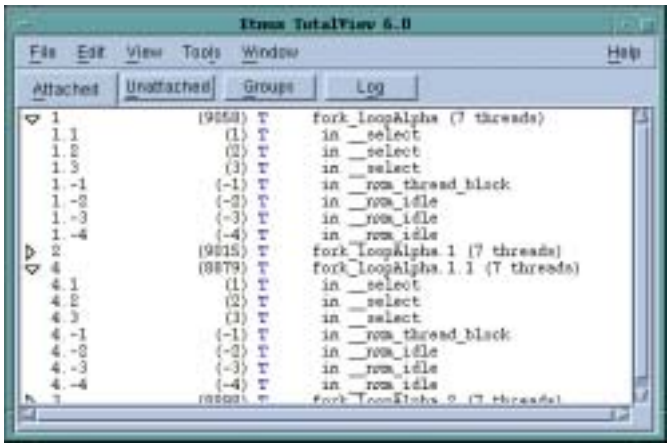
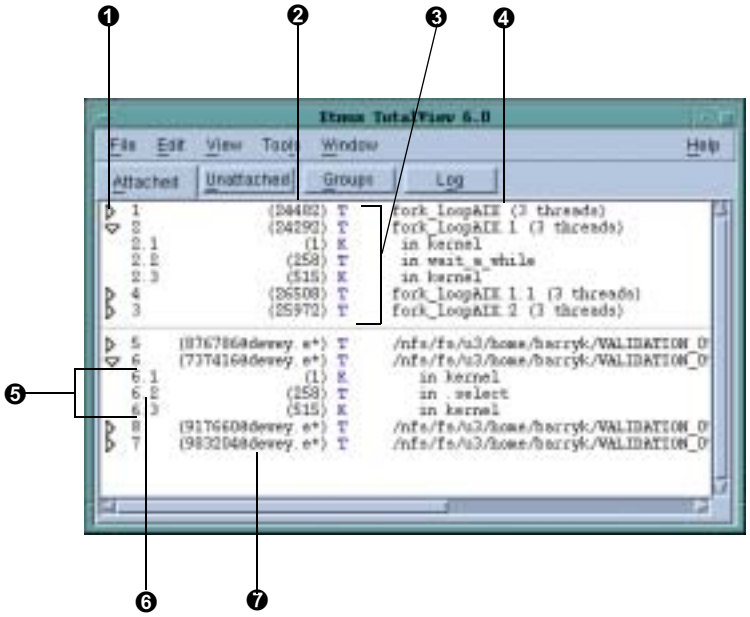


Figure 88: Root Window Showing Remote



❶ Collapse/expand toggle
❷ Process ID (PID)
❸ Thread status
❹ Program name
❺ Thread list
❻ Thread ID (TID/SYSTID)
❼ Remote process location

- Unattached: Displays processes over which you have control. If you can't attach to one of these processes, TotalView displays it in gray. Figure 89 on page 122 shows the Unattached Page.

  Diving on processes in this pane tells TotalView to attach to them.
- Groups: Lists the groups used by your program. The top pane in Figure 90 on page 122 lists all of your program's groups. This list includes all the groups that TotalView creates and all that you create using the CLI. When you select a group in the top pane, the group's members are displayed in the bottom pane.

Figure 89: Root Window Unattached Page

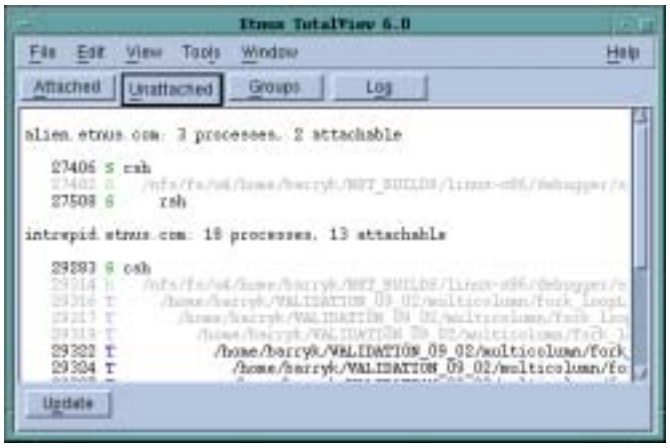
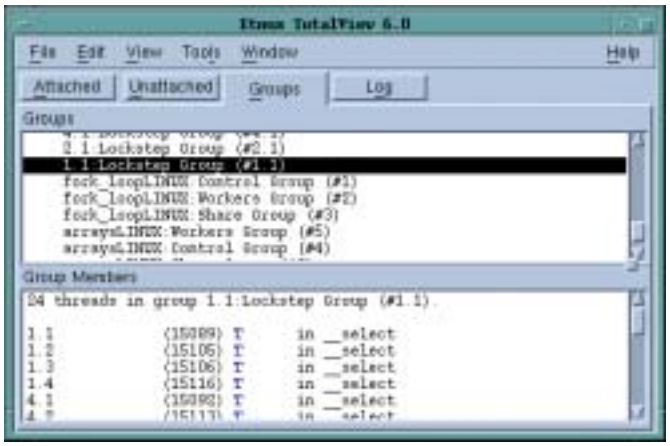
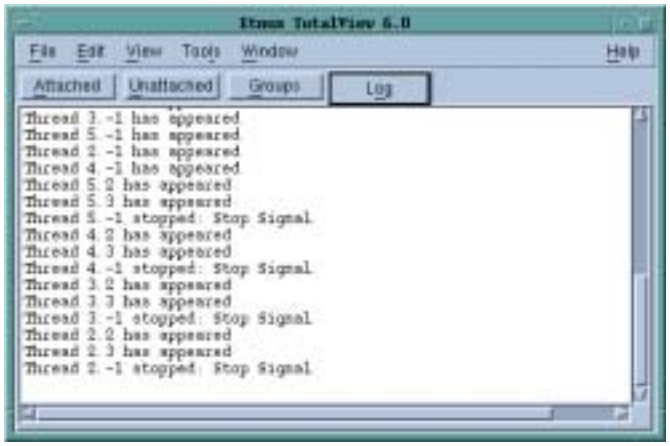


Figure 90: Root Window Groups Page



- Log: Contains a log of the debugging actions. (See Figure 91.) This information is sometimes useful when analyzing the behavior of misbehaving multiprocess/multithreaded programs.

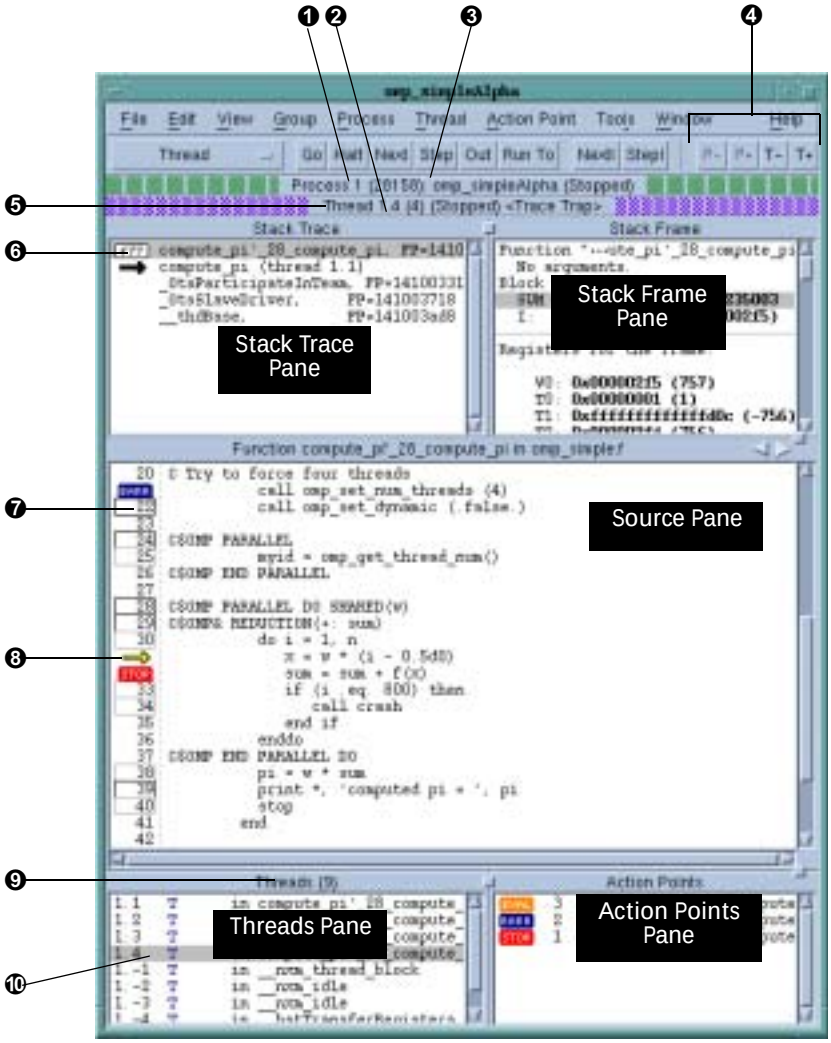
Figure 91: Root Window Log Page



Using the Process Window

The *Process Window*, which is shown in Figure 92, contains the code for the process or thread you're debugging, as well as other related information. This window contains five *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.)

Figure 92: Process Window



As you examine the Process Window, notice the following:

- The thread ID shown in the Root Window and in the process's Threads Pane is the TotalView-assigned logical thread ID (TID) and 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.

  In other windows, TotalView uses the value *pid.tid* to identify a process's threads.

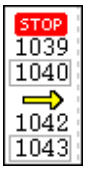
The *Threads Pane* shows the list of threads that currently exist in the process. When you select a different thread in this list, TotalView updates the Stack Trace Pane, Stack Frame Pane, and Source Pane to show the information for that thread. When you dive on a different thread in the thread list, TotalView finds or opens a new window displaying information for that thread.

The number in the Threads Pane title (❾ in Figure 92 on page 123) is the number of threads that currently exist in the process.

- 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 the process when it was last stopped. This means that the information they are displaying is not up-to-date while the 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. (See Figure 93.) 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) within the selected stack frame. See Figure 93.

Figure 93: Line Numbers, with Stop Icon and PC Arrow



Each thread has its own unique program counter (PC). When you stop a multiprocess or multithreaded program, the routine displayed in the Stack Trace Pane for a thread depends on the thread's PC. Because threads execute asynchronously, you'll usually find that threads are stopped at different places. (When your thread hits a breakpoint, the TotalView default is to stop all the other threads in the process as well.)

- The *Action Points Pane* shows the list of breakpoints, evaluation points, and watchpoints for the process.

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 preforms 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. While this may sound confusing, it's pretty intuitive and you'll be diving without thinking almost instantaneously.

For example, 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. Here's another example: diving on variables in the Process Window tells TotalView to display information about the variable in a Variable Window.

Table 7 describes typical diving operations.

Table 7: Diving

Dive on:	Information Displayed by Diving:
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.
Subroutine	The source code for the routine replaces the current contents of the Source Pane—this is called a nested dive. When this occurs TotalView places a right angle bracket (>) in the process's title. Every time it dives, it adds another angle bracket. See Figure 94, which follows this table. A routine must be compiled with source-line information (usually, with the –g 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.
Variable	The contents of the variable appear in a separate Variable Window.
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 something that is more to your liking.
Array element, structure element, or referenced memory area	The contents of the element or memory area replaces the contents that were in the Variable Window—this is known as a *nested* dive.
Routine in the Stack Trace Pane	The stack frame and source code for the routine appear in a Process Window.

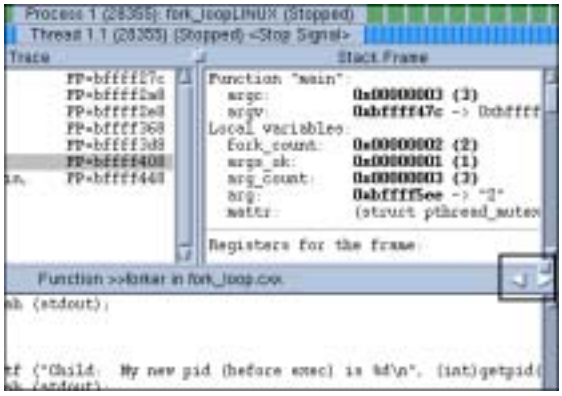
Figure 94: Nested Dive

TotalView tries to reuse windows whenever possible. 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 Anew command.

Using on a process or a thread may 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's 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* icons. The location of the two controls is shown in the boxed area in Figure 95.

Figure 95: Backward and Forward Buttons



For additional information about displaying variable contents, refer to "*Diving in Variable Windows*" on page 237.

Other windowing commands that you can use are:

- Window > Duplicate: (Variable Window) Creates a duplicate copy of the current Variable Window.
- Window > Duplicate Base: (Variable Window) Creates a copy of the current Variable Window. Unlike what happens when you use the Window > Duplicate command, this command retains the dive stack.
- 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 currently open window and all windows similar to it. When you have lots of similar windows, this is a great timesaver.

Resizing and Positioning Windows and Dialog Boxes

You can resize most of TotalView's 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 96 on page 127 shows a before and after look where a dialog box was made too small.

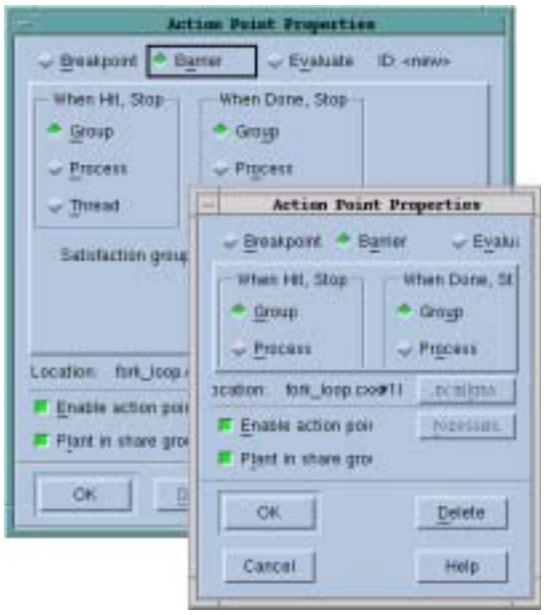


Figure 96: Resizing (and Sometimes Its Consequences)

Many programmers like to have their windows always appear in the same position in each session. TotalView has two commands that can help:

- Window > Memorize: Tells TotalView it should 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 it should remember the positions of just about all of its windows. The next time you bring up any of the windows displayed when you had used this command, it will be in the same position.

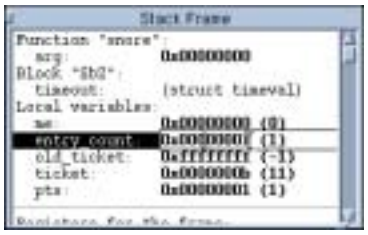
Most modern window managers such as KDE or Gnome do an excellent job managing window position. If you are using an older window manager such as twm or mwm, you may want to select the Force window positions (disables window manager placement modes) check box option located on the Options Page of the File > 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 wish to change. If you can edit the selected text, it appears within a highlighted rectangle, and you will see an editing cursor. (See Figure 97 on page 128.)
2. Edit the text and press Return.

Figure 97: Editing Cursor



Like other Motif-based applications, you can use your mouse to copy and paste text within 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.

In most cases, clicking your middle mouse button tells TotalView to dive. However, if TotalView is displaying an editing cursor, clicking your middle mouse button tells TotalView to paste information.

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 98.)

Figure 98: File > Save Pane Dialog Box

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, here 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
```

Visualizing Programs and Data

7

TotalView provides a set of tools that allow you to visualize how your program is performing and the values of variables. This chapter describes:

Other visualization tools are described in the following sections:

Displaying Your Program's Call Tree

Debugging is an art, not a science. Debugging often means having the "intuition" to know what a problem means and where to look for it. Locating a problem is often 90% or more of the effort. TotalView's call tree is one tool that helps you get an understanding of what your program is doing so that you can begin to understand how your program is executing.

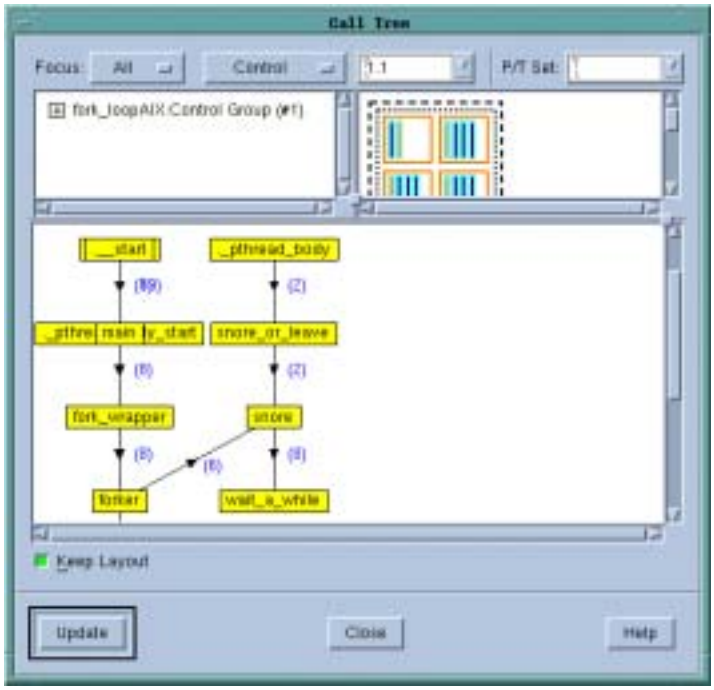
Use the Tools > Call Tree command in the Process Window to tell TotalView to display a Call Tree Window. (See Figure 99 on page 130.)

The call tree is a diagram showing all the currently active routines. These routines are linked by arrows indicating that one routine is called by another. TotalView's call tree is a *dynamic* call tree in that it displays the call tree at the time when TotalView creates it. The Update button tells TotalView to recreate this display.

You'll find information on using the P/T Set Controls in the top portion of this window in Chapter 11, "Using Groups, Processes, and Threads," on page 197.

You can tell TotalView to display a call tree for the processes and threads specified with the controls at the top of this window. If you don't touch

Figure 99: Tools > Call Tree Dialog Box



these controls, TotalView displays a call tree for the group defined in the icon bar of your Process Window. If TotalView is displaying the call tree for a multiprocess or multithreaded program, numbers next to the arrows indicate how many times a routine is on the call stack.

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.

Looking at the call tree 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 may be negatively affecting performance. For example, in Figure 99, the snore routine might be a bottleneck. Giving good names to routines helps. For example, if you see lots of routines in a routine named snore, you've probably designed things to that routines will be waiting there, so this wouldn't represent a problem.

Visualizing Array Data

The TotalView Visualizer creates graphic images of your program's array data.

The Visualizer isn't available on Linux Alpha and 32-bit SGI Irix. It's available on all other platforms.

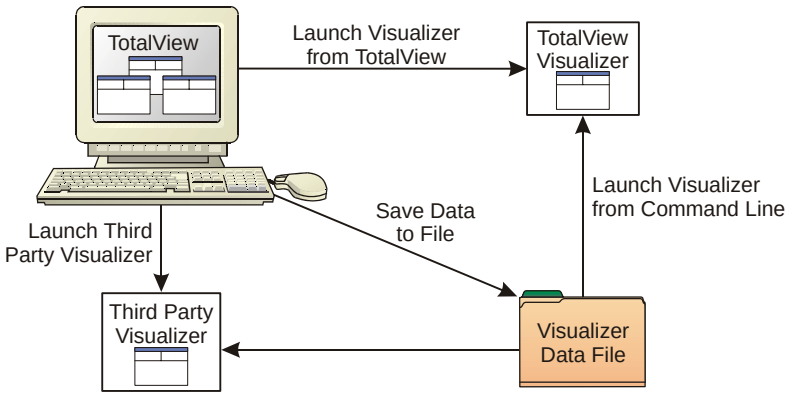
Topics in this section are:

How the Visualizer Works

The Visualizer is a stand-alone program that is integrated with TotalView. This relationship gives you a lot of flexibility:

- If you launch the Visualizer from within TotalView, you can visualize your program's data as you are debugging your program.
- You can save the data that would be sent to the Visualizer, and then invoke the Visualizer from the command line and have it read this previously written data.(See Figure 100.)

Figure 100: 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.

Visualizing your program's data is a two step process:

1. You select the data that you want visualized.
2. You tell the Visualizer how it should display this data.

TotalView marshals the program's data and pipes it to the Visualizer. The Visualizer reads this date and displays it for analysis. (See Figure 101 on page 132.)

Figure 101: TotalView Visualizer Connection

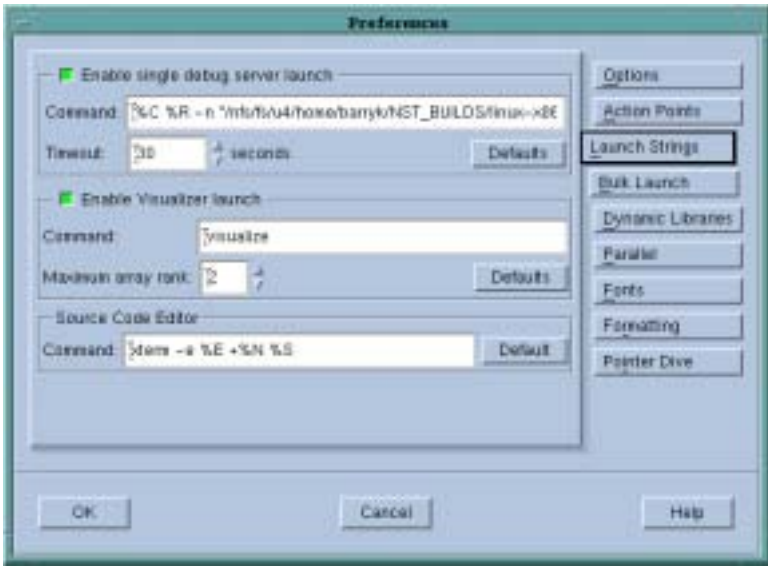
Configuring TotalView to Launch the Visualizer

TotalView launches the Visualizer when you select the Tools > Visualize command from the Variable Window. It will also launch it if or when you use a $visualize function within an evaluation 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 containing evaluation points, without having to individually disable all the evaluation points.

To change the Visualizer launch options interactively, select File > Preferences, and then select the Launch Strings Tab. (See Figure 102.)

Figure 102: File > Preferences Launch Strings Page



Using the commands in this page, you can:

- Customize the command TotalView uses to start a visualizer by entering the visualizer's startup command in the Command edit box. Entering information in this field is discussed a little later in this section.

- Change the autolaunch 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 edit field to save the data exported from the debugger 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 TotalView 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 on the Defaults button returns all values to their defaults. This reverts options to their defaults 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.

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 reasons you'd do this is if you're having path problems or you're running a different visualizer.) You could also change what's entered here so that you can view this information at another time. Here's an example:

```
cat > your_file
```

Later, you can visualize this information using either of the following commands:

```
visualize -persist < your_file
visualize -file your_file
```

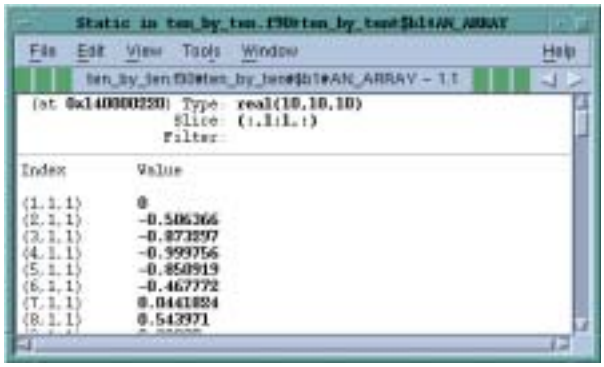
You can preset the Visualizer launch options by setting X resources. These resources are described on our Web site. For more information, go to www.etnus.com/Support/docs/.

Data Types That TotalView Can Visualize

The data selected for visualization is called a *dataset*. Each dataset is tagged with a TotalView-generated numeric identifier that lets the Visualizer know whether it is seeing a new dataset or an update to an existing dataset. TotalView treats stack variables at different recursion levels or call paths as different datasets.

TotalView can visualize one- and two-dimensional arrays of character, 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 103 on page 134 shows a three-dimensional variable sliced into two dimensions by selecting a single index in the middle dimension.

Figure 103: A Three-Dimensional Array Sliced into Two Dimensions



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 will occur:

- If a Data Window is currently displaying the dataset, the Visualizer raises it to the top of the desktop. If the window was minimized, the Visualizer restores it.
- If you haven't visualized the dataset in this session, the Visualizer chooses a method based on how well your dataset matches what is best shown for each kind of visualization method. You can enable and disable this feature from the Options menu in the Visualizer's Directory Window.
- If you've previously visualized a dataset but you've killed its window, the Visualizer creates a new Data Window by using the most recent visualization method.

Visualizing Data Manually

Before you can visualize an array, you must:

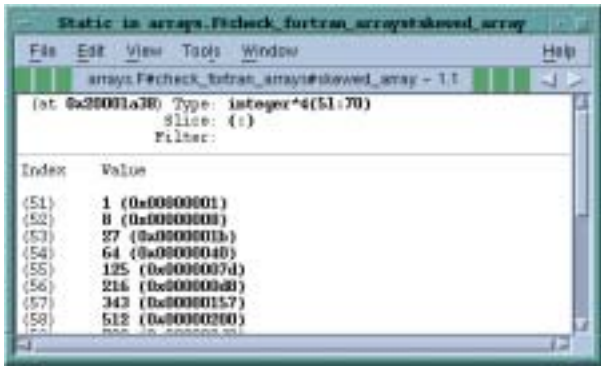
- Open a Variable Window for the array's data.
- Stop program execution where the array's values are set to what you want them to be when they are visualized.

Figure 104 on page 135 shows an Variable Window containing an array.

You can restrict the data being visualized by editing the Type and Slice fields. For example, editing the Slice fields limits the amount of data being visualized. (See "*Displaying Array Slices*" on page 259.) Limiting the amount increases the Visualizer's speed.

After selecting the Variable Window's Tools > Visualize command, the Visualizer begins executing and then 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 reentering the Tools > Visualize command.

Figure 104: Variable Window



TotalView can visualize laminated variables. (See "*Visualizing a Laminated Variable Window*" on page 272.) The process or thread index will be one of the visualized data's dimensions. This means that you can only visualize scalar or vector information. If you don't want the process or thread index to be a dimension, use a nonlaminated display.

Visualizing Data Programmatically

TotalView's $visualize function allows you add visualization to expressions in evaluation action points or with expressions entered in the Tools > Evaluate Window. If you enter this function within an expression, TotalView will interpret rather than compile the expression, which can greatly decrease performance. See "*Defining Evaluation Points and Conditional Breakpoints*" on page 286 for information about compiled and interpreted expressions. Adding this function also lets you visualize several different variables from a single expression or evaluation point.

Using $visualize in an evaluation point lets you animate the changes that occur in your data because the Visualizer will update the array's display every time TotalView reaches the evaluation point. Here's 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 defining a constant slice expression that modifies the *array* parameter's dataset.

Here are six examples showing how you can use this function. Notice that the array's dimension ordering differs in C and in Fortran.

C
```
$visualize(my_array);
$visualize (my_array,"[::2][10:15]");
$visualize (my_array,"[12][:]");
```

Fortran
```
$visualize (my_array)
$visualize (my_array,'(11:16,::2)')
$visualize (my_array,'(:,13)')
```

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 will know what the array's dimensions are. Here's a C function declaration that passes a two-dimensional array parameter. Notice that it does not specify the major dimension's extent.

```
void my_procedure (double my_array[][32])
{ /* procedure body */ }
```

Here's how you can cast the array so that 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 $visualize into the Tools > Evaluate Dialog Box. When you select the Evaluate button, you'll quickly see the result. You can even use this technique to display several arrays simultaneously.

Using the Visualizer

The Visualizer uses two types of windows:

- **Data Windows**

  These are the windows that display your data. The commands in a Data Window let you set viewing options and change the way the Visualizer displays your data.
- **A Directory Window**

  This window lists 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 opening more than one Data Window.

The top window in Figure 105 on page 137 is a Directory Window. The two remaining windows show a surface and a graph view.

Directory Window

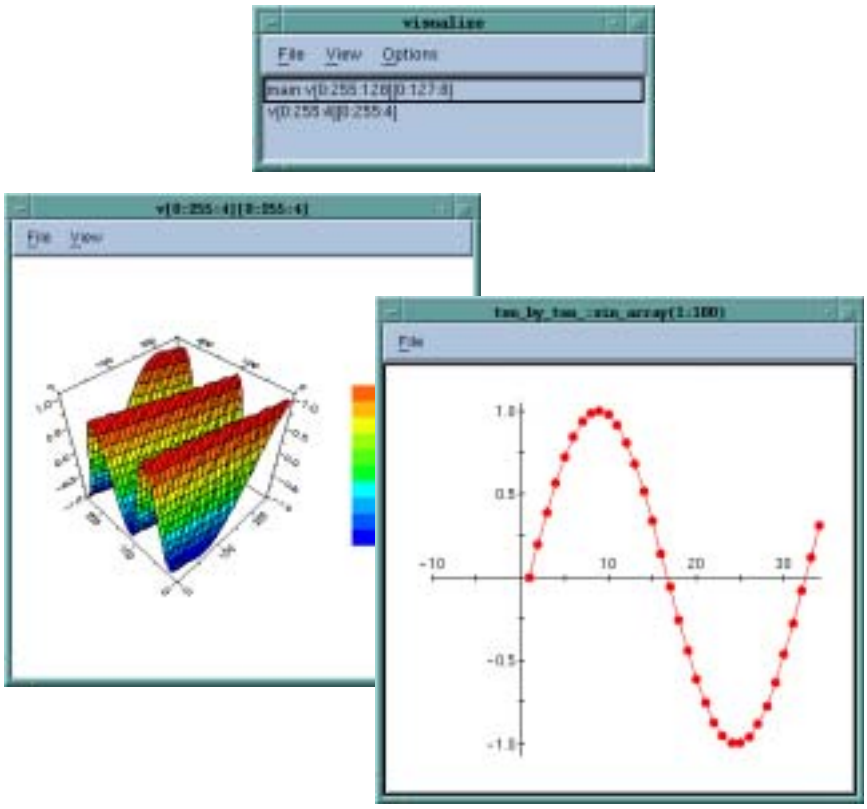
The Directory Window contains a list of the datasets you can display. You can select a dataset by clicking on it. Double-clicking on the dataset tells the Visualizer to display it. While you can display multiple datasets, you can only select one dataset at a time.

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.)

Here are the commands contained in the Directory Window's menu bar:

File > Delete	Deletes the currently selected dataset. It removes the dataset from the dataset list and *destroys* the Data Windows displaying it.

Figure 105: Sample Visualizer Windows



File > Exit	Closes all windows and exits the Visualizer.
View > Graph	Creates a new Graph Window; see "*Using the Graph Window*" on page 138 for more detail.
View > Surface	Creates a new Surface Window; see "*Using the Surface Window*" on page 140 for more detail.
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 need to configure how the Visualizer displays it, you may want to disable this option.

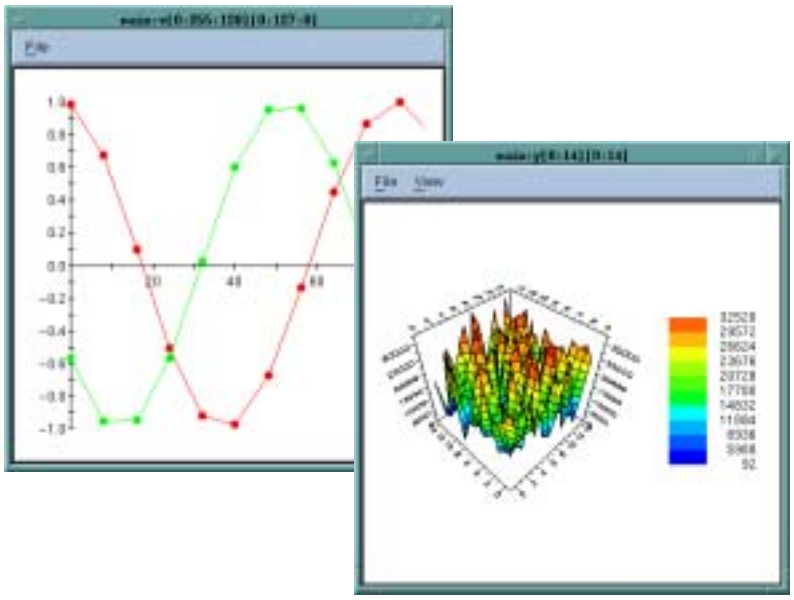
Data Windows

Data Windows display graphic images of your data. Figure 106 on page 138 shows a surface view and a graph view. Every Data Window contains a menu bar and a drawing area. The Data Window title is its dataset identification.

The Data Window menu commands are as follows:

File > Close	Closes the Data Window.
File > Delete	Deletes the Data Window's dataset from the dataset list. This also destroys other Data Windows viewing the dataset.

Figure 106: Sample Visualizer Data Windows



File > Directory — Raises the Directory Window to the front of the desktop. If you have minimized the Directory Window, the Visualizer restores it.

File > New Base Window — Creates a new Data Window having the same visualization method and dataset as the current Data Window.

File > Options — Pops up a window of viewing options.

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 graph 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. A pop-up window displays the information. (See Figure 107 on page 139.)

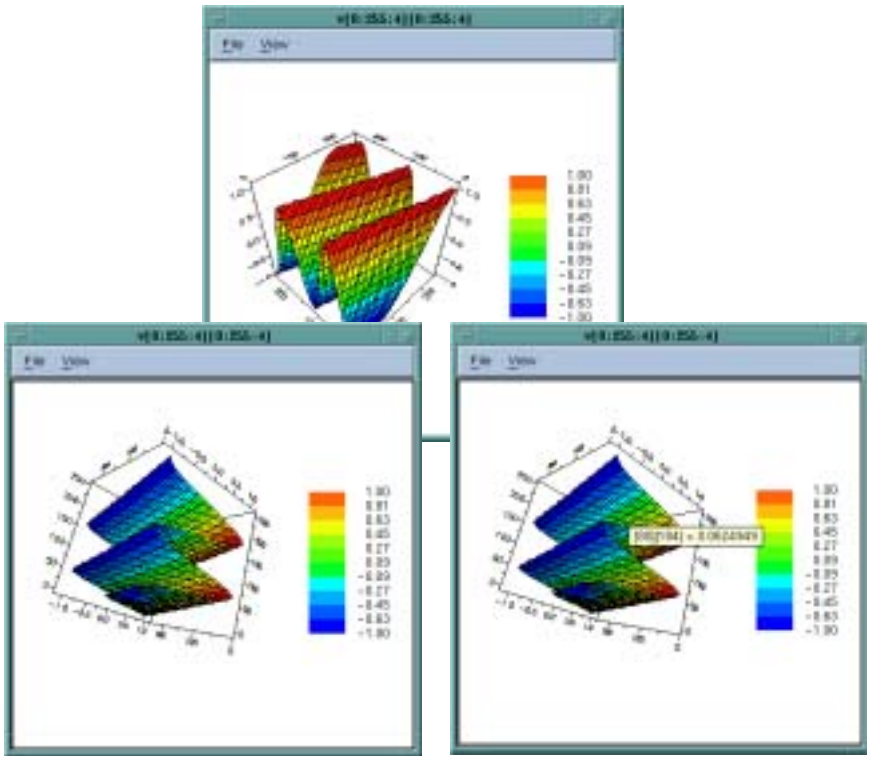
Using the Graph Window

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 having the smaller number of elements. If you don't like this choice, you can transpose the data.

You probably don't want to use a graph to visualize two-dimensional datasets with large extents in both dimensions, as the display will be very cluttered.

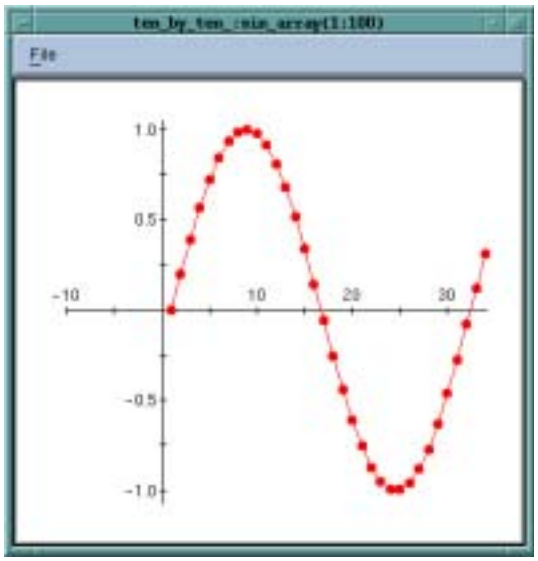
You can display graphs with markers for each element of the dataset, with lines connecting dataset elements, or with both lines and markers as shown in Figure 108 on page 139. See "*Displaying Graphs*" on page 140 for more details. Multiple graphs are displayed in different colors. The X axis of

Figure 107: Rotating and Querying



the graph is annotated with the indices of the long dimension. The Y axis shows you the data value.

Figure 108: Visualizer Graph Data Window



You can scale and translate the graph, or pop up a window displaying the indices and values for individual dataset elements. See "*Manipulating Graphs*" on page 140 for details.

Displaying Graphs

The File > Options Dialog Box lets you control how the Visualizer displays the graph. (See Figure 109.)

Figure 109: Graph Options Dialog Box

Here's what the check boxes in this dialog box mean.

Lines	If this is set, the Visualizer displays lines connecting dataset elements.
Points	If this is set, the Visualizer displays markers for dataset elements.
Transpose	If this is set, the Visualizer inverts the X and Y axis of the displayed graph.

Manipulating Graphs

You can manipulate the way the Visualizer displays a graph by using the following actions:

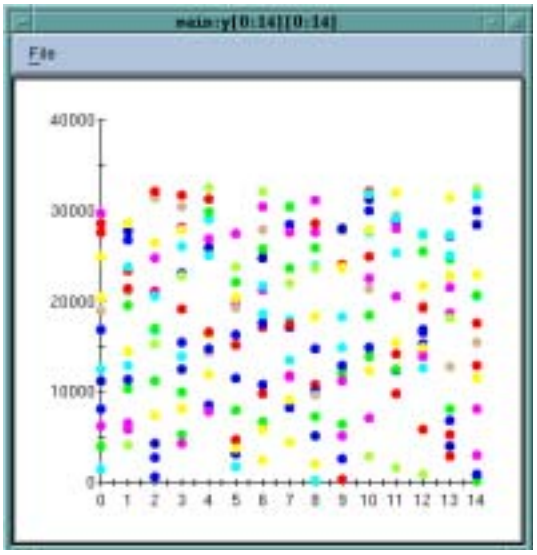
Scale	Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
Translate	Press the Shift key and hold down the middle mouse button. Moving the mouse drags the graph.
Zoom	Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses an area. The Visualizer scales the graph to fit the drawing area.
Reset View	Select View > Reset to reset the display to its initial state.
Query	Hold down the left mouse button near a graph marker. A window pops up displaying the dataset element's indices and value.

Figure 110 on page 141 shows a graph view of two-dimensional random data created by selecting Points and clearing Lines in the Data Window's Graph Options Dialog Box.

Using the Surface Window

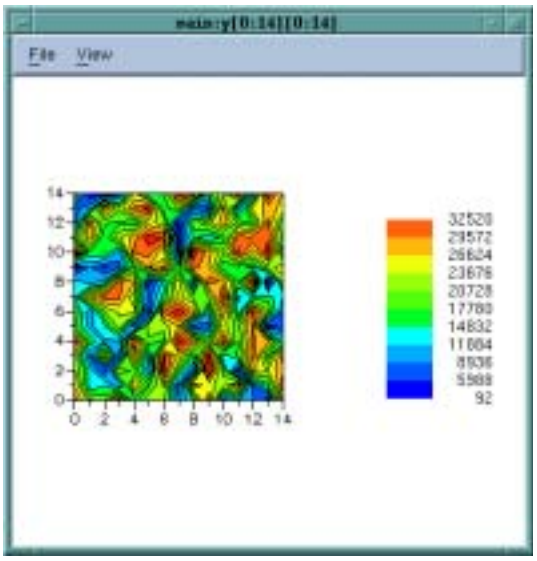
The Surface Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (X and Y axes) of the display. Figure 111 on page 141 shows a two-dimensional map, where the dataset values are shown using only the Zone option. (This demarcates ranges of element values.) For a zone map

Figure 110: Display of Random Data



with contour lines, turn the Zone and Contour settings on and Mesh and Shade off.

Figure 111: Two-Dimensional Surface Visualizer Data Display



You can display random data by selecting only the Zone setting and turning Mesh, Shade, and Contour off. The display shows where the data is located, and you can click on the display to get the values of the data points.

Figure 112 on page 142 shows a three-dimensional surface that maps element values to the height (Z axis).

Displaying Surface Data

The Surface Window's File > Options command lets you control how the Visualizer displays the graph. (See Figure 113 on page 142.)

Figure 112: Three-Dimensional Surface Visualizer Data Display

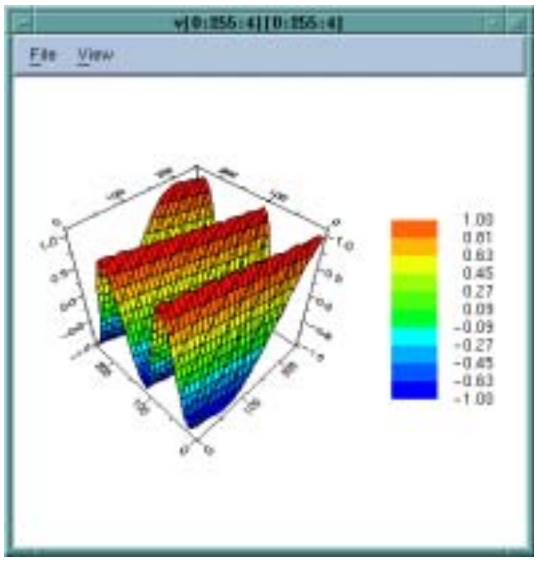
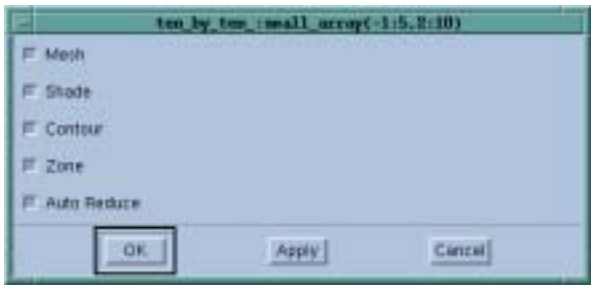


Figure 113: Surface Options Dialog Box



This dialog box has the following choices:

Mesh	If this option is set, the Visualizer displays the surface as a three dimensional mesh, with the X-Y grid projected onto the surface. If you don't set this or the Shade option, the Visualizer displays the surface in two dimensions. (See Figure 111 on page 141.)
Shade	If this option is set, the Visualizer displays the surface in three dimensions and shaded either in a "flat" color to differentiate the top and bottom sides of the surface, or in colors corresponding to the value if the Zone option is also set. When neither this nor the Mesh option are set, the Visualizer displays the surface in two dimensions. (See Figure 111 on page 141.)
Contour	If this option is set, the Visualizer displays contour lines indicating ranges of element values.
Zone	If this option is set, the Visualizer displays the surface in colors showing ranges of element values.
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 reduc-

ing the resolution of the surface. Clear this option if you want to accurately visualize all dataset elements.

The Auto Reduce option allows you to 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.

You can reset the viewing parameters to those used when the Visualizer first came up by selecting the View > Reset command, which restores all translation, rotation, and scaling to its initial state and enlarges the display slightly.

Manipulating Surface Data

The following commands change the display or give you information about it:

Query	Hold down the left mouse button near the surface. A window pops up displaying the nearest dataset element's indices and value.
Rotate	Hold down the middle mouse button and drag the mouse to freely rotate the surface. You can also press the X, Y, or Z keys to select a single axis of rotation. The Visualizer lets you rotate the surface in two dimensions simultaneously. While you're rotating the surface, the Visualizer displays a wire-frame bounding box of the surface and moves it as your mouse moves.
Scale	Press the Control key and hold down the middle mouse button. Move the mouse down to zoom in on the center of the drawing area, or up to zoom out.
Translate	Press the Shift key and hold down the middle mouse button. Moving the mouse drags the surface.
Zoom	Press the Control key and hold down the left mouse button. Drag the mouse button to create a rectangle that encloses the area of interest. The Visualizer then translates and scales the area to fit the drawing area. See Figure 114 on page 144.

Launching the Visualizer from the Command Line

To start the Visualizer from the shell, use the following syntax:

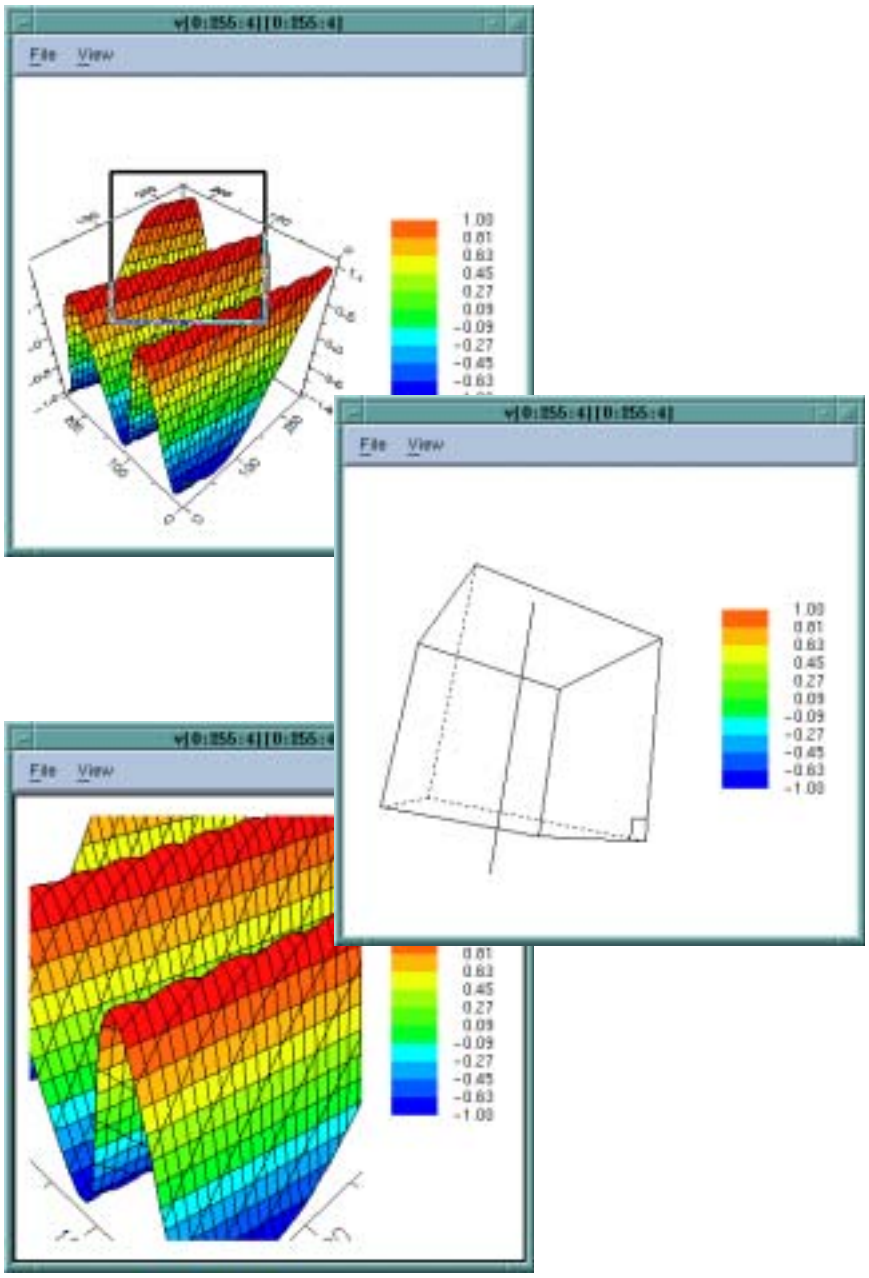
visualize [–file *filename* | –persist]

where:

–file *filename*	Reads data from *filename* instead of reading from standard input.
–persist	Continues to run after encountering an EOF 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

Figure 114: Zooming, Rotating, About an Axis



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. The available resources are described in "*TotalView Command Syntax*" in the *TotalView Reference Guide*. Use of X resources to modify the default behavior of TotalView or the TotalView Visualizer is described in greater detail on our Web site at www.etnus.com/Support/docs/xresources/XResources.html.

Part IV: Using the CLI

While other parts of this book deal with both the GUI and the CLI or with just the GUI, the chapters in this part deal exclusively with the CLI. Most CLI commands must have a process/thread focus for what they will be doing. See Chapter 11: "*Using Groups, Processes, and Threads*" on page 197 for more information.

Chapter 8: Seeing the CLI at Work

While you can use the CLI as a stand-alone debugger, using the GUI is usually easier. Where the CLI shines is in creating debugging functions that are unique to your program or in automating repetitive tasks. This chapter presents a few Tcl macros in which TotalView CLI commands are embedded.

While most of these examples are simple, you are urged to, at a minimum, skim over this information so you get a feel for what can be done.

Chapter 9: Using the CLI

You can use TotalView's CLI commands without knowing much about Tcl, which is the approach taken in this chapter. Here you will read about how to enter CLI commands and how the CLI and TotalView interact with one another when used in a nongraphical way.

Seeing the CLI at Work

8

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 within 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 a few macros that show how the CLI programmatically interacts with your program and with TotalView. Reading a few examples without bothering too much with details will give you an appreciation for what the CLI can do and how you can use it. As you will see, you really need to have a basic knowledge of Tcl before you can make full use of all CLI features.

The chapter presents a few macros. In each macro, all Tcl commands that are unique to the CLI are displayed in bold. The macros in this chapter are for:

Setting the EXECUTABLE_PATH State 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 will ignore directories named in the *filter* argument. The result is then 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 TotalView 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. Here is how you use this function:

```
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x0000001)
   (2) = 2 (0x0000001)
   (3) = 3 (0x0000001)
 }
d1.<> array_set list 2 3 99
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x0000001)
   (2) = 99 (0x0000063)
   (3) = 99 (0x0000063)
 }
```

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like other ones 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 could specify: `dprint a(1:4,1:4)`.

After invoking this macro, the CLI prints a two-dimensional slice (i1:i2:i3, j1:j2:j3) 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 "*CLI Output*" on page 162.

Here are several examples:

```
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

There are many times when you would like 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 shows how you might use this macro. Notice that using the exec command lets cat 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 will occur. The following CLI macro parses comments that you can include within a source file and, depending on the comment's text, sets a breakpoint or an evaluation point.

Immediately following this listing is an excerpt from a program that uses this macro.

```
# 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 kind of
      # action point; they are buried in the comments.
   while {[gets $fsource line] != -1} {
      incr lineno
      set bpline $lineno

            # Look for a one-line evaluation 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 evaluation 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 example 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 sets evaluation points and breakpoints.

The following excerpt from a larger program shows how you would embed comments within a source file that would be read by the next 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, an evaluation point at an *eval* line, or executes a command at a *cmd* line.

Using evaluation points can be confusing because evaluation point syntax differs from that of Tcl. In this example, the $stop command is a command contained in TotalView (and the CLI). It is not a Tcl variable. In other cases, the evaluation statements will be in the C or Fortran programming languages.

Using the CLI 9

The two components of the Command Line Interface (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that allow you to debug your program. This chapter looks at how these components interact and describes how you specify processes, groups, and threads.

This chapter tends to emphasize interactive use of the CLI rather than using the CLI as a programming language because many of the concepts that will be discussed are easier to understand in an interactive framework. However, everything in this chapter can be used in both environments.

Topics discussed in this chapter are:

Tcl and the CLI

The TotalView CLI is built within version 8.0 of Tcl, so TotalView's 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 TotalView 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

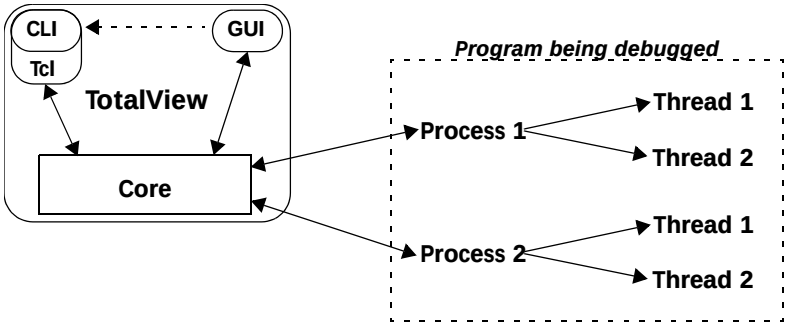
as you enter and execute built-in Tcl commands. As CLI commands are also Tcl commands, you can embed Tcl primitives and functions within CLI commands and embed CLI commands within sequences of Tcl commands.

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. Or you create a Tcl function that dynamically builds the arguments that a process will use when it begins executing.

The CLI and TotalView

The following figure illustrates the relationship between the CLI, the TotalView GUI, the TotalView core, and your program:

Figure 115: The CLI and TotalView



The CLI and the 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 and receives information back from these processes, and passes them 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.

The CLI Interface

The way in which you interact with the CLI is by entering a CLI or Tcl command. (As entering a Tcl command does exactly the same thing in the CLI as it does when interacting with a Tcl interpreter, entering commands and the command environment won't be discussed here.) 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 may 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've asked it to do.

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 may want to limit the amount of information TotalView displays by setting the VERBOSE variable to WARNING or ERROR. (For more information, see the "*Variable*s" chapter in the *TotalView Reference Guide*.)

Pressing Ctrl + C while a CLI command is executing interrupts that CLI command or executing Tcl macro. If the CLI is displaying its prompt, typing Ctrl + C stops executing processes.

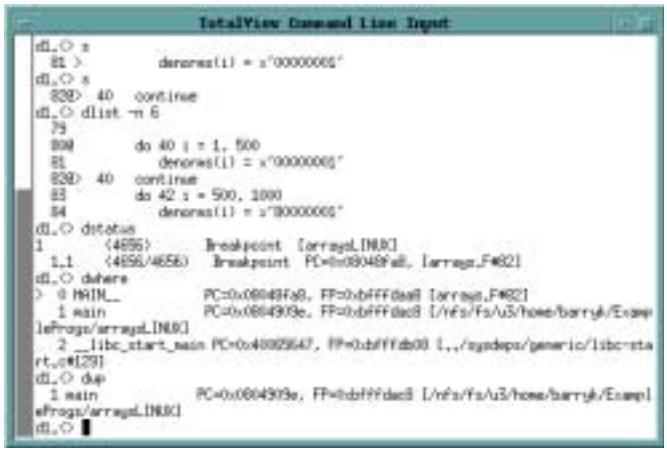
Starting the CLI

You can start the CLI in two ways:

- You can start the CLI from within the TotalView window by selecting the Tools > Command Line command in the Root and 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 116 is a snapshot of a CLI window that shows part of a program being debugged.

Figure 116: CLI xterm Window



If you have problems entering and editing commands, it could be because you invoked the CLI from a shell or process that manipulates your stty settings. You can eliminate these problems if you use the stty sane CLI command. (If the sane option isn't available, you will 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've done

Startup Example

Here 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 8, "*Seeing the CLI at Work*," on page 149. It then loads the fork_loop executable, sets its default startup arguments, and then steps one source-level statement.

If you stored this in a file named fork_loop.tvd, here is how you would tell TotalView to start the CLI and execute this file:

```
totalviewcli –s fork_loop.tvd
```

Information on TotalView's 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 would 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
```

Notice that the only difference is the first few lines in the file. In the second line, the shell ignores the backslash continuation character while Tcl processes it. This means that the shell will execute 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. The following example uses the totalviewcli command to start the CLI. This is followed by dload and drun commands. As 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 ">" symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in "Command and Prompt Formats" on page 164.

```
% totalviewcli
Copyright 1999-2002 by Etnus, LLC. ALL RIGHTS RESERVED.
Copyright 1999 by Etnus, Inc.
Copyright 1989-1996 by BBN Inc.
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 then running allows you to 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'll get a new process each time and the old one will still be around.

The dkill command terminates one or more processes of a program started by using dload, drun, or drerun. The following example continues where the previous example left off:

```
d1.<> dkill            # kill process
d1.<> drun              # runs arraysLINUX from start
d1.<> dlist -e -n 3     # show 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's 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, then you know that the CLI is still executing.

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.)

Here are two cases where it matters whether TotalView directly prints output or returns and then prints it:

- When the Tcl interpreter executes a list of commands, TotalView 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 dload 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 help 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, here's how to place help command 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 Root Window's Log Pane. If it is being written there, you can use the File > Save Pane *command to write this information to a file.*

"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 discard 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*.)

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 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 dset to modify the ARGS() variables directly, or you can start the process with the drun command. For example, here is how you can clear 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 drun) that can create a process—including dgo, drerun, dcont, dstep, and dnext—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 within 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. The namespaces that TotalView uses are:

TV::	Contains commands and variables that you will use when creating functions. While they can be used interactively, this is not their primary role.
TV::GUI::	Contains state variables that define and describe properties of the user interface such as window placement, color, and the like.

If you discover other namespaces beginning with TV, you have found a place containing internal functions and variables. These objects can (and will) change and disappear, so don't use them. Also, don't create namespaces that begin with TV, as you could cause problems by interfering with built-in functions and variables.

The CLI's 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::
```

Command and Prompt Formats

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.>	All worker threads in the control group containing process 3.
p3/3	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! > "
```

Built-In Aliases and Group Aliases

Many CLI commands have an alias that allows you to 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.*

After a few minutes of entering CLI commands, you will quickly come to the conclusion that it is much more convenient to use the command abbreviation. For example, you could type:

```
dfocus g dhalt
```

(This command tells the CLI to halt the current group.) It is much easier to type:

```
f g h
```

While less-used commands are often typed in full, a few commands are almost always abbreviated. These command 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.)

There are two ways in which group aliases differ from the kind of group-level command that you would type:

- They do not work if the current focus is a list. The g focus specifier modifies the current focus, and it 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.

Effects of Parallelism on TotalView and CLI 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 the CLI's 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 would 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

Kinds of IDs

Multithreaded, multiprocess, and distributed program contain a variety of IDs. Here is some background on the kinds used in the CLI and TotalView:

System PID	This is the process ID and is generally called the *PID*.
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. Note that if the target process is killed and restarted (that is, you use the dkill and drun commands), the debugger PID doesn't change. The system PID, however, changes since the operating system has created a new target process.
System TID	This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no ob-

vious 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.

Controlling Program Execution

Knowing what's going on and where you 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 the debugger to stop the program. Sometime later, you will tell the serial program to continue executing. Multiprocess and multithreaded 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 the other threads and processes. Some may stop; some may 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 command to start the program executing. These three commands work at process level and you can't use them to start an 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 Chapters 2 and 11.) 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 allows you to alter a program's state in a controlled way. And debugging is the process of stopping the process to examine its state. However, the term "stop" has a slightly different meaning in a multiprocess, multithreaded 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.

Action Points

Action points tell the CLI that it should stop a program's execution. You can specify four different kinds 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. (Note that you can only set a barrier on processes; you can't set then on individual threads.)
- An *evaluation 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. Evaluation points typically do not stop the process; instead, they perform an action. In most cases, an evaluation point stops the process when some condition that you specify is met.

Extensive information on action points can be found in "Setting Action Points" on page 273.

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 TotalView 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. (Setting multiple action points is required by debuggers that limit what you can do.)

Part V: Debugging

The chapters in this part of the TotalView Users Guide describe how you actually go about debugging your programs. The preceding sections describe, for the most part, what you need to do before you get started with TotalView. In contrast, the chapters in this section are what TotalView is really about.

Chapter 10: Debugging Programs
: Reading this chapter well help you find your way around your program. It also tells you how to start it under TotalView's control, and the ways to step your program's execution. Of course, it also tells you how to halt, terminate, and restart your program.

Chapter 11: Using Groups, Processes, and Threads
: The stepping information in Chapter 10 describes the commands and the different kinds of stepping. In a multiprocess, multithreaded program, you may need to finely control what is executing. This chapter tells you how to do this.

Chapter 12: 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 13: Examining Arrays
: Displaying the information in arrays presents special problems. This chapter tells how TotalView solves these problems.

Chapter 14: Setting Action Points
: TotalView's 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.

Debugging Programs 10

This chapter explains how to perform basic debugging tasks with TotalView. The topics discussed are:

Searching and Looking Up 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

You can search for text strings in most windows with the Edit > Find command. After invoking this command, TotalView displays the dialog box shown in Figure 117.

Figure 117: 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, and keep the dialog box up between searches. You can also tell TotalView if it should 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.

Looking for Functions and Variables

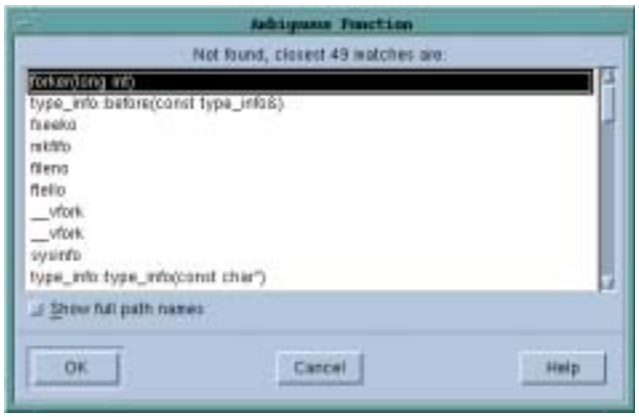
Having TotalView locate a variable or a function is usually easier than scrolling through your sources to look for it. Do this with the View > Lookup Function and View > Lookup Variable commands. Figure 118 shows the dialog box displayed by these commands.

Figure 118: View > Lookup Variable Dialog Box

If TotalView doesn't find the name and it can find something similar, it displays a dialog box containing the names of functions that could match. (See Figure 119 on page 173.)

If the one you want is listed, click on its name and then, select OK to have it displayed in the Source Pane.

Figure 119: Ambiguous Function Dialog Box



Finding the Source Code for Functions

The View > Lookup Function command lets you search for a function's declaration. Here's the dialog box displayed after you select this command.

CLI: dlist *function-name*

Figure 120: View > Lookup Function Dialog Box

After locating your function, TotalView displays it in the Source Pane. If you didn't compile the function using –g, TotalView displays disassembled machine code.

When you want to return to the previous contents of the Source Pane, use the undive icon located in the upper right corner of the Source Pane and just below the Stack Frame Pane. In the following figure, a square surrounds the undive icon.

Figure 121: 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.

The File > Edit Source command (see ""*Editing Source Text*" on page 177 for details) lets you display a file in a text editor. The default editor is vi. However, TotalView will use the editor named in an EDITOR environment variable or the editor you name in the Source Code Editor field of the File > Preferences's Launch Strings Page.

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'll see the source.

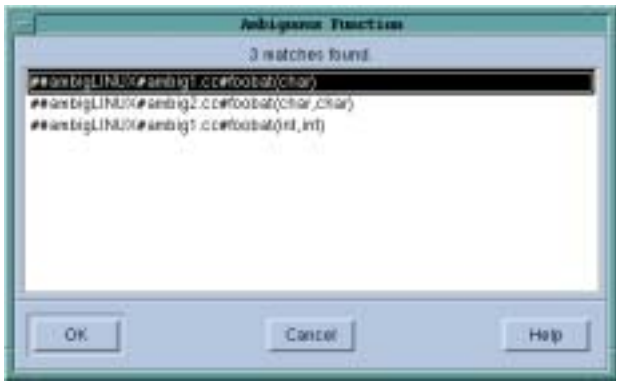
Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you may 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.

Figure 122 shows the dialog box that TotalView displays when it encounters an ambiguous function name. You can resolve the ambiguity by clicking on the function you desire.

Figure 122: Ambiguous Function Dialog Box



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 Figure 123 on page 175.

If a header file contains source lines that produce executable code, you can enter its 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 onto the screen.

Figure 123: View > Lookup Function Dialog Box

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 your program will execute.

Viewing the Assembler Version of Your Code

You can display your program in source form or as assembler. Here are the commands that you can use:

Source code (Default)
: Use the View > Source As > Source command.

Assembler code
: Use the View > Source As > Assembler command.

Both Source and assembler
: Use the View > Source As > Both command.

  The Source Pane divides into two parts. The left contains the program's source code and the right contains the assembler version of this code. You can set breakpoints in either of these panes. Note that setting an action point at the first instruction after a source statement, is equivalent to 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:

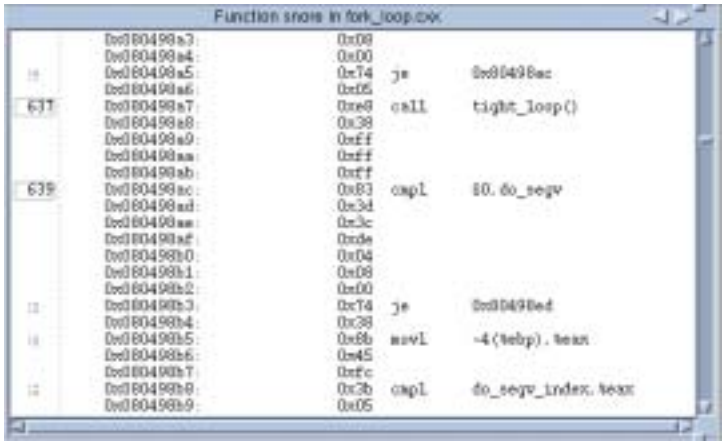
Table 8: Assembler Code Display Styles

Command	TotalView Shows
View > Assembler > By Address	Absolute addresses for locations and references; this is the 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 236.

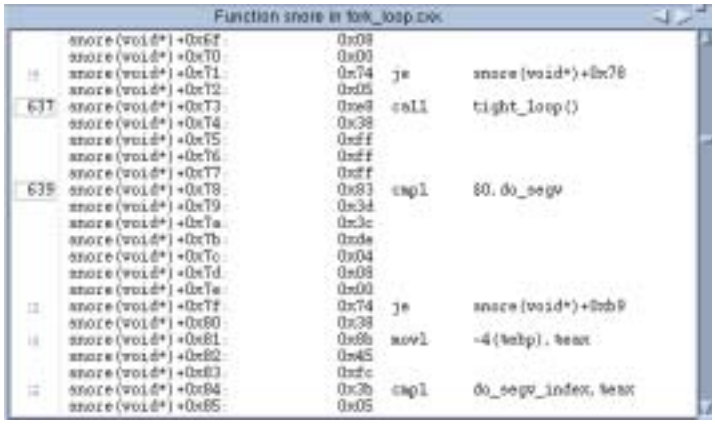
The following three figures illustrate the different ways TotalView can display assembler code. In Figure 124, the second column (the one to the right of the line numbers) shows the absolute address location. The fourth column shows references using absolute addresses.

Figure 124: Address Only (Absolute Addresses)



This next figure shows information symbolically. The second column shows locations using functions and offsets.

Figure 125: Assembler Only (Symbolic Addresses)



The final "assembler" figure (see Figure 126 on page 177) 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 upon whether you've selected View > Assembler > By Address or View > Assembler > Symbolically.

Figure 126: Both Source and Assembler (Symbolic Addresses)

Editing Source Text

The File > Edit Source command lets you examine the current routine in a text editor. TotalView uses an *editor launch string* to determine how to start your editor. TotalView expands this string into a command that TotalView sends to the sh shell.

The fields within the Launch Strings Page of the File > Preferences Window let you name the editor and the way TotalView launches the editor. 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

The Process Window's toolbar has three groups of buttons. The first group, which is a single pulldown list, defines the *focus* of the command selected in the second group of the toolbar. The third group changes the process and thread being displayed. Figure 127 shows this toolbar.

Figure 127: 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 upon. In the CLI, you specify this target using the dfocus command. When using the GUI, you specify the focus using this pulldown. For example, if you select 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 (this thread is called the *thread of interest*).

Chapter 11, "Using Groups, Processes, and Threads," on page 197 fully 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, suppose you select Process > Halt. This tells TotalView to determine the process in which the TOI is running. It will then halt this process. Similarly, if you select Group > Share > Halt, TotalView determines what processes are in the share group 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 197.

When you select the Halt button in the toolbar instead of the commands within 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 had stopped the process.

Updating Process Information

Normally, TotalView only updates information when the thread being executed stops executing. You can force TotalView to update a window if you use the Window > Update command. You'll 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 it needs. It then restarts the thread.

Holding and Releasing Processes and Threads

When you are running a multiprocess or multithreaded program, there will be many times when you will want to synchronize execution to the same statement. You can do this manually using a *hold* command, or you can let TotalView do this 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 > Control > Go, two of the three processes will resume executing. The held process ignores the Go command.

At a later time, you will want to run whatever 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 these 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'd 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 "*Barrier Points*" on page 283 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 letter H. When TotalView is holding a thread, it displays the letter h.

Here are four ways to hold or release a thread, process, or group of processes:

- You can hold a group of processes with the Group > Hold command.
- You can release a group of processes with 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.

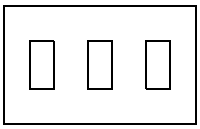
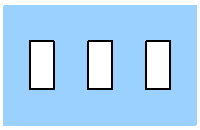
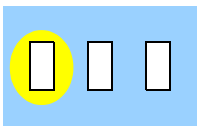
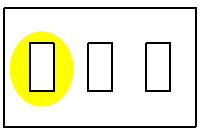
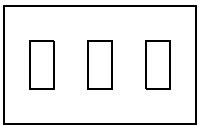
CLI: dhold and dunhold
Setting the focus changes the scope.

If a process or a thread is running when you enter a hold or release command, TotalView stops the process or thread, and then holds it.TotalView allows you to hold and release processes independently from threads.

Notice that the Process pulldown menu contains a Hold Threads and a Release Threads command. While they appear to do the same thing, they are used in a slightly different way. If you use Hold Threads on a multi-threaded process, you'll be placing a hold on all threads. This is seldom what you want. If you then uncheck the Threads > Hold command, TotalView allows you to release the one you want. This is 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's Attached pane.

CLI: dhold –thread
dhold –process
dunhold –thread

The following table 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 within the process can be run.
	Select the Process > Hold toggle. The button will be depressed. The blue indicates that you've hold the process. Nothing will run when you select Process > Go.
	Go to the Threads menu. Notice that the button next to the Hold command isn't selected. This is because the *thread hold* state is independent from the *process hold* state. Select it. The circle indicate that thread 1 is held. At this time, there are two different holds on thread 1. One is at process level, the other is at thread level. Nothing will run when you select Process > Go.
	Go back to the Process menu and reselect the Hold command. After you select Process > Go, the 2nd and 3rd threads run.
	Select Process > Release Threads. This releases the hold placed on the first thread by the Thread > Hold command. After you select Process > Go, all threads run.

Examining Groups

When you debug a multiprocess 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, "*Understanding 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, it decides where to place a process based on the type of

system call (fork() or execve()) that created or changed the process. The two types of process groups are:

Control Group	Includes 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 execve(). 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	Is the set of processes in a control group that share the same source code. Members of the same share group share action points.

See Chapter 11, "Using Groups, Processes, and Threads," on page 197 for a complete discussion of groups.

TotalView automatically creates share groups when your processes fork children that call execve() 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. Here are the naming rules it uses:

- It 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 numerical suffix is the process's rank in COMM_WORLD.
- If a child process calls execve() after it is forked, TotalView places a new executable name within angle brackets (< >).

In Figure 128, assume that the generate process doesn't fork any children, and that the filter process forks two child process. Later, the first child forks another child and then calls execve() to execute the expr program. In this figure, the middle column shows the names that TotalView will use.

Figure 128: Example of Control Groups and Share Groups

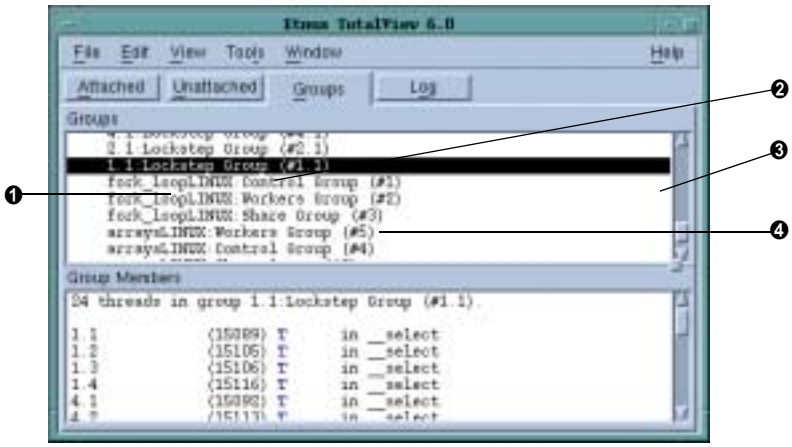
Process Groups		Process Names	Relationship
Control Group 1	Share Group 1	filter	parent process #1
		filter.1	child process #1
		filter.2	child process #2
	Share Group 2	filter<expr>.1.1	grandchild process #1
Control Group 2	Share Group 3	generate	parent process #2

Displaying Groups

To display a list of process and thread groups, select the Root Window's Groups Tab. (See Figure 129.)

CLI: dptsets

Figure 129: Root Window: Groups Page



❶ Name of executable
❷ Type of process or thread group
❸ Select a group in the top pane to display members in the bottom pane
❹ Group number

When you select a group in the top list pane, TotalView updates the bottom pane to show the group's members. After TotalView updates the bottom pane, you can dive into anything shown there.

Placing Processes into Groups

TotalView uses your executable's name to determine the share group the program belongs to. If the path names are identical, it assumes they are the same program. If the path names differ, TotalView assumes they are different even if the filename within the path name is the same and will place them in different share groups.

Starting Processes and Threads

To start a process, go to the Process Window and select a Go command from the Group, Process, or Thread pulldown menus.

After you select a Go command, TotalView decides what it will 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 enter Group > Workers > Go, TotalView continues all threads in the workers group 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 multiprocess program. There are some limitations, however. TotalView only resumes a process if it:

- Is not being held.
- Already exists and is stopped.
- 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, a 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 will try to perform an equivalent action. For example, it will continue a process instead of a thread.*

For a single-process program, Process > Go and Group > Go are equivalent. For a single-threaded process, Process > Go and Thread > Go 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 had linked 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:

- 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 equivalent to the Process > Create command, executing dstepi produces the same effect.

Creating a Process by Single-Stepping

The TotalView single-stepping commands allow you to create a process and run it to a location in your programs. The single-stepping commands available from the Process menu are as shown in the following table:

Table 9: Creating a Process by Stepping

GUI Command	CLI Command	Creates the process and ...
Process > Step	dfocus p dstep	Runs it to the first line of the main() routine.
Process > Next	dfocus p dnext	Runs it to the first line of the main() routine; this is the same as Process > Step.
Process > Step Instruction	dfocus p dstepi	Stops it before any of your program executes.

GUI Command	CLI Command	Creates the process and ...
Process > Next Instruction	dfocus p dnexti	Runs it to the first line of the main() routine. this is the same as Process > Step.
Process > Run To	dfocus p duntil	Runs it to the line or instruction selected in the Process Window.

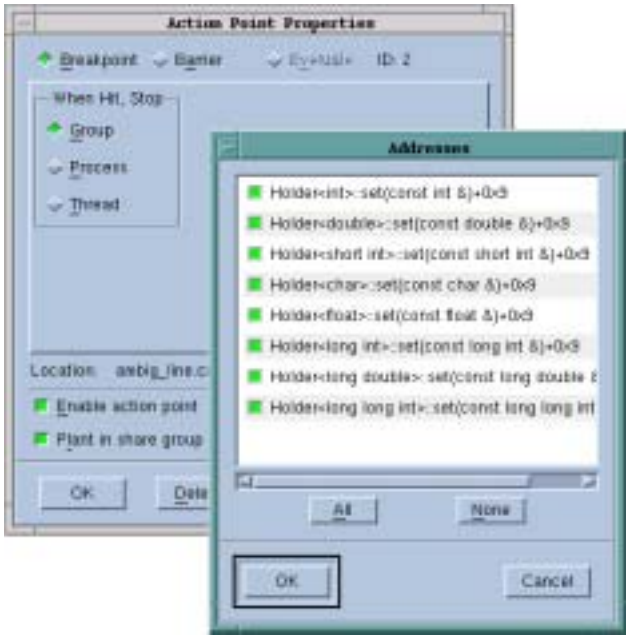
If a group-level or thread-level stepping command creates a process, the behavior is the same as if it were a process-level command.

Chapter 11, "Using Groups, Processes, and Threads," on page 197 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 Chapter 10, "*Debugging Programs*," on page 171.)

Figure 130: Action Point Properties and Address Dialog Boxes



If TotalView cannot figure out which instantiation to set a breakpoint at, it will display its Address Dialog Box. (See Figure 131 on page 185.)

Figure 131: Ambiguous Addresses 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`

Here are some things to remember about single-stepping commands:

- To cancel a single-step command, put the cursor in the Process Window and 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 a 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.

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 should step. 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.

It is possible that you might not realize your program is calling a function. For example, if you've overloaded an operator, you'll step into the code that defines the overloaded operator.

If the function being stepped into wasn't compiled with –g, TotalView will always step over the function.

The TotalView GUI has eight Step commands and eight Step Instruction commands. These commands are located on the Group, Process, and Thread pulldowns. The difference is the focus.

CLI: dfocus ... dstep
dfocus ... dstepi

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 TotalView 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 ... dnext
dfocus ... dnexti

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 TotalView 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 11, "*Using Groups, Processes, and Threads*," on page 197.

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 frame pointer (FP) of the selected stack frame.
- The selected source line or instruction to determine.

CLI: dup and ddown

Executing to the Completion of a Function

You can step your program out of a function by using the Out commands. The eight commands within the TotalView GUI are located on the Group, Process, and Thread menus.

CLI: dfocus ... dout

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 this is your source line:

```
routine1; routine2
```

Suppose you step into routine1, then use an Out command. While the PC arrow hasn't moved, the actual PC is just after routine1. This means that if you use a step command, you will step into routine2.

TotalView's PC arrow will not move, when the source line only has one statement on it. The internal PC will, of course, have changed.

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'll be running out of.

Displaying Your Program's Thread and Process Locations

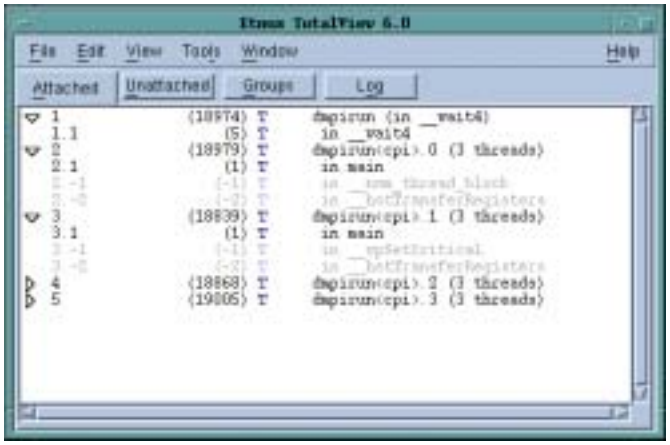
You can see which processes and threads in the share group are at a location by selecting a source line or machine instruction in the Source Pane of the Process Window. TotalView dims thread and process information in the Root Window's Attached Page for share group members if the thread or process is not at the selected line. TotalView considers a process to be at the selected line if any of the threads in the process are at that line. Selecting a line in the Process Window that is already selected removes the dimming in the Attached Page.

CLI: dstatus

The Attached Page reflects the line that you last selected. If you have several Process Windows open, the information in the Attached Page will change depending on the line you selected last in each Process Window. The display can also change after an operation that changes the process state or when you issue a Window > Update command.

Figure 132 shows an Attached Page with dimmed process information. In this example, the parallel program was run to a barrier breakpoint, and one process (dmpirun<cpi>.1) was stepped to the next source line.

Figure 132: Dimmed Process Information in the Root Window



Since the MPI starter process (dmpirun) isn't in the same share group as the processes running the cpi program, TotalView doesn't dim its process information.

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. (See Figure 133 on page 189.)
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 it the specified signals.

You can clear the continuation signal by selecting signal 0.

Figure 133: Thread > Continuation Signal Dialog Box

Deleting Programs

To delete all the processes in a control group, use the Group > Delete 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 multiprocess 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 > Delete command followed by the Process > Go command.

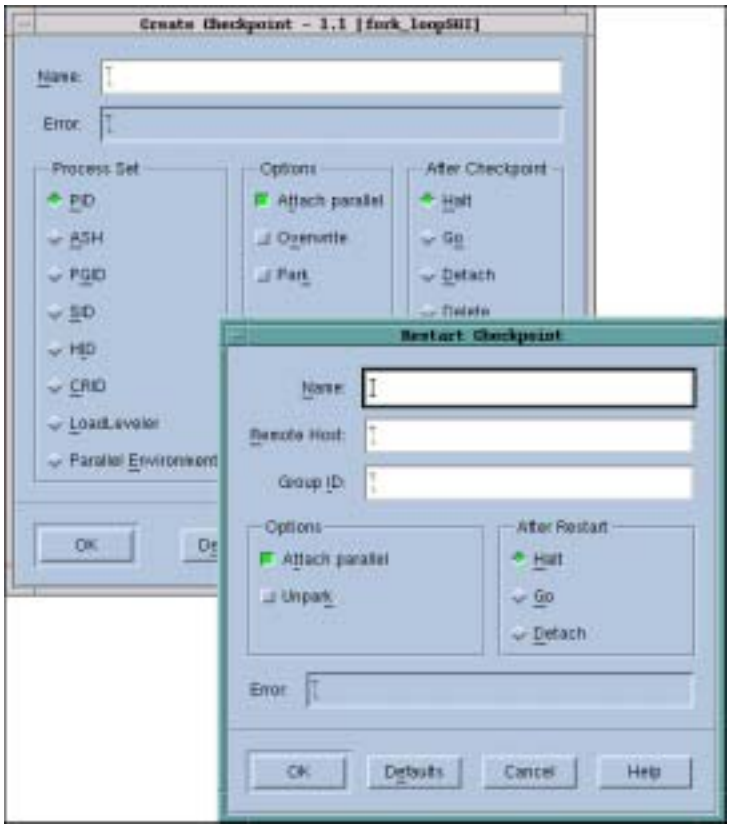
Checkpointing

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's Tools > Create Checkpoint and Tools > Restart Checkpoint commands in TotalView's Help information. (See Figure 134.)

CLI: dcheckpoint
drestart

Figure 134: Checkpoint and Restart Dialog Boxes



Fine Tuning Shared Library Use

When TotalView encounters a reference to a shared library, it normally reads in all of that library's symbols. In some cases, you may need to explicitly read in a library before TotalView would automatically read it in.

On the other hand, you may not want TotalView to read in 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 that it shouldn't read these symbols.

For more information, see:

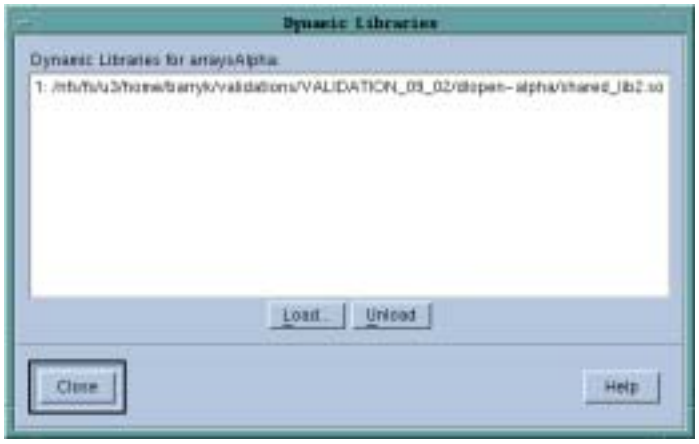
- "*Preloading Shared Libraries*" on page 191
- "*Controlling Which Symbols TotalView Reads*" on page 192

Preloading Shared Libraries

As your program executes, it can call dlopen() to access code contained in shared libraries. In some cases, you may need to do something from within TotalView that requires you to preload the information within the library. For example, you may need to refer to one of a library's functions in an evaluation point or in a Tools > Evaluate command. If you use the function's name before TotalView reads the dynamic library, you'll get an error message.

Use the Tools > Manage Shared Libraries command to tell TotalView to open a library. After selecting this command, TotalView displays the following dialog box:

Figure 135: Tools > Manage Shared Libraries Dialog Box

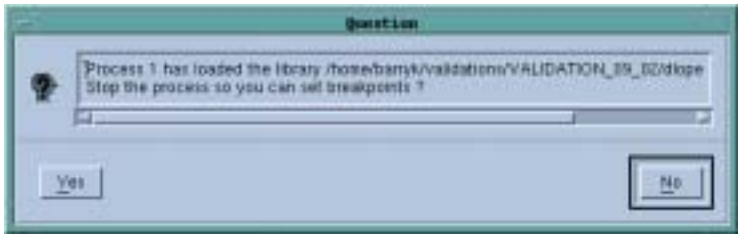


CLI: ddlopen

This CLI command gives you more ways to control how a library's symbols are used than exist in the GUI.

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 136: Stopping to Set a Breakpoint Question Box



TotalView may not read in information symbol and debugging information when you use this command. See "Controlling Which Symbols TotalView Reads" on page 192 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 upon your debugging session.

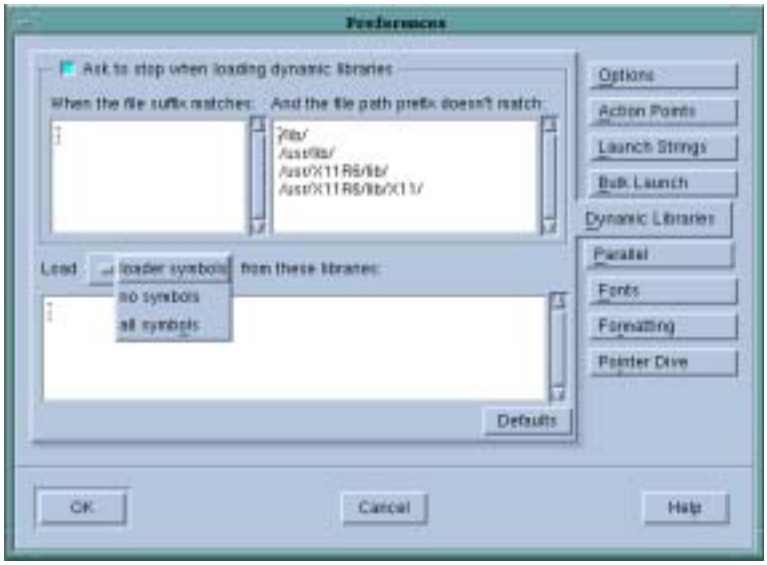
Using the preference settings and variables described in this section will always increase performance. However, for most programs, even large ones, the difference is inconsequential. If, however, you are debugging a very large program with large libraries, significant performance improvements can occur.

A shared library contains, among other things, loader and debugging symbols. Typically, loader symbols are read quite quickly. Debugging symbols, on the other hand, may require considerable processing. TotalView's default behavior is, of course, to read all symbols. You can change this behavior by telling TotalView that it should only read in loader symbols or even tell it that it should not read in any symbols.

Specifying Which Libraries are Read

After invoking the File > Preferences command, select the Dynamic Libraries Page.

Figure 137: File > Preferences: Dynamic Libraries Page



The lower portion of this page lets you enter the names of libraries for which you need to manage the information that TotalView reads.

When you enter a library name, you can use the * (asterisk) and ? (question mark) wildcard characters. These characters have their standard meaning. Here's what placing entries into these areas does:

all symbols	This is TotalView's 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 just reads in 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 *. You would 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 as TotalView may 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

So, 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 in a library's symbols, it cannot now 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.

TotalView always reads the loader symbols for shared system libraries.

If your program stops in a library that has not already had its symbols read, TotalView will read the library's symbols. For example, if your program SEGVs in a library, TotalView will read the symbols from that library before it reports the error.

Reading Excluded Information

While debugging your program, you may 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 select the Load All Symbols in Stack command form the context menu.

Figure 138: Load All Symbols in Stack Context Menu

After selecting this command, TotalView examines all active stack frames and if finds unread libraries in any frame, reads them in.

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 icon 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, a selected instruction, or an absolute value (in hexadecimal). 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, display the View > Source As > Both command. TotalView responds by displaying the 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 icon 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. To learn about the meaning of these registers, you need to consult the user's guide for your CPU and "*Architectures*" in the *TotalView Reference Guide*.

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 continue execution of your program. 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), refer to "*Displaying Areas of Memory*" on page 235.

Using Groups, Processes, and Threads

11

While the specifics of how multiprocess, multithreaded programs execute differ greatly from platform to platform and environment to environment, all share some general characteristics. This chapter discusses TotalView's process/thread model. It also describes the way in which you tell the GUI and the CLI what processes and threads it should direct a command to.

The topics discussed in this chapter are:

Defining the GOI, POI, and TOI

This chapter consistently uses three related acronyms:

- GOI, which means Group of Interest
- POI, which means Process of Interest
- TOI, which means Thread of Interest

These terms are important in TotalView's process/thread model because TotalView must determine the scope of what it will do when executing a command. For example, Chapter 2 introduced the kinds of groups contained within TotalView. For reasons that will become obvious in this chapter, that chapter ignored what happens when you execute a TotalView command upon a group. For example, what does "stepping a group" actually

mean? Which processes and threads will TotalView are actually stepped? What happens to processes and threads that aren't in this group?

Associated with these three terms is a fourth: *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 two pulldown menus in the toolbar. If you examine the menubar, you'll see that there are 8 *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 will run. Depending on the command, TotalView determines the TOI, POI, or GOI, and then executes the command's action upon that thread, process, or group. For example, assume that you tell TotalView to step the current control group.

- TotalView needs to know what the TOI is so it can determine what threads are in the lockstep group—TotalView only allows you to step a lockstep group.
- The lockstep group is part of a share group.
- This share group is also contained in a control group.

So, by knowing what the TOI is, the TotalView GUI also knows what the GOI is. This is important because, as you will see, while TotalView now 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.

Using the GOI, POI, and TOI will become clearer as you read the rest of this chapter.

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 single-step commands described in "*Using Stepping Commands*" on page 185.
- 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 194.

- Set breakpoints at lines you choose and allow your program to execute to that breakpoint. "*Setting Breakpoints and Barriers*" on page 275.
- 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 8. See "*Setting Evaluation Points*" on page 287.

TotalView provides additional features for working with breakpoints, process barrier breakpoints, and evaluation points. For more information, refer to Chapter 14, "*Setting Action Points*," on page 273.

Stepping (Part I)

TotalView's stepping commands allow you 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 containing 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 machine steps every instruction, you will probably never get past this statement. *Smart stepping* means that TotalView will single-step through the assignment statement at a speed that is very close to your machine's native speed.

Other topics in this section are:

Group Width

TotalView's 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 are doing a *step* command, it is the next line. If it is a *run to* command, it is the selected line.

If the GOI is a:

- *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 just 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 will stop 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.

Process Width

TotalView's 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. If the GOI is a:

- *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 will TotalView stop the other threads in the process.
- *Thread group*, TotalView allows all threads in the GOI and all manager threads to 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.

Thread Width

When TotalView performs a stepping command, it decides what it will step 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 that it should just run that thread. It does not step other threads. In contrast, process width tells TotalView that it should run all threads in the process that are allowed to run while the TOI is stepped. While TotalView is stepping the thread, manager threads are running 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 should instead use process-width stepping.

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 tells TotalView to execute program statements *until* a selected statement is reached.) When applied to a process group, TotalView identifies all processes in the group already having a thread stopped at the goal. These are the *matching* processes. TotalView then runs only the 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 *sync up* all the processes in a group in preparation for group-stepping them.

Here is what you should know 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 is run. This allows you to use the Run To command repeatedly within 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 allows you to synchronize the threads in the POI at a source line.

If you're running at group width:

Process group — TotalView examines each process in the process and share group to determine if 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 if 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 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. Issue 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 TotalView windows have P/T set control elements. For example, Figure 139 shows the top portion of the Process Window.

Figure 139: The P/T Set Control in the Process Window

This pulldown menu differs from the P/T set controls on other elements. On other windows, there are two pulldowns. However, in the context of the Process Window, elements from the two pulldowns have been combined both to eliminate actions that don't have meaning. When you select a group and a modifier, you are telling TotalView that when you press one of the remaining buttons on the toolbar, this element names the focus upon which TotalView will act. For example, if Thread is selected and you 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 multiprocess program, other processes would continue to execute.

Other windows have similar controls. For example:

The first pulldown menu, which is called the *Width Pulldown*, has three elements on it: Group, Process, and Thread. Your choices here indicate the width of the command. For example, if Group is selected, a Go command continues the group. Which group TotalView will continue is set by the

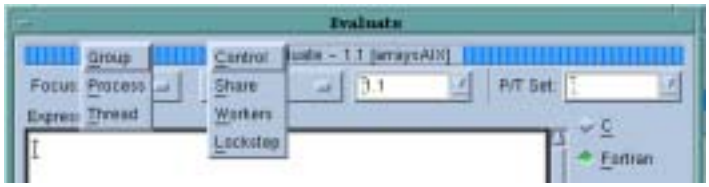


Figure 140: The P/T Set Control in the Tools > Evaluate Window

choices on the second pulldown menu. The *Width Pulldown* tells TotalView where it should look when it tries to determine what it will manipulate. The second pulldown, which is called the *Scope Pulldown*, tells TotalView which processes and threads within the scope defined by the *Width Pulldown* it should manipulate. For example, you could tell TotalView to step the threads defined in the current workers group that are contained in the current process.

Finally, the *P/T Selector* (the third pulldown menu from the left) lets you change the focus of the action from the currently defined process and threads to any other process and thread that TotalView controls. That is, this changes the POI and TOI

The P/T Set expression box on the right allows you to directly enter a P/T set expression. The focus of what you enter is modified by the other P/T set controls.

What is selected can be quite complicated when you use the GUI to set these controls, or when you specify a focus using the CLI.

Setting Process and Thread Focus

While the previous sections have emphasized the GUI, this section and the ones that follow emphasize the CLI. In all cases, the selection of what TotalView runs is based directly or indirectly on P/T set syntax. While the focus is obvious in the CLI, it is often buried within the internals of the GUI. Reading the rest of this chapter is important when you want to have full asynchronous debugging control over your program. Having this level of control, however, is seldom necessary.

When it executes a command, TotalView must decide which processes and threads it should 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's prompt.

There are times, however, when you'll need to change this default. This section begins a rather intensive look at how you tell TotalView what processes and threads it should use as the target of a command.

Topics in this section are:

Process/Thread Sets

All TotalView commands operate on a set of processes and threads. This set is called a *P/T (Process/Thread) set.* The right-hand text box in windows containing 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 where 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 Tcl list containing one or more P/T identifiers. (The next section, "*Arenas*" on page 205, explains what a P/T identifier is.) Tcl lets you create lists in two 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 Tcl's 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 "*Command and Prompt Formats*" on page 164.)

You can change the focus upon which a command acts by using the dfocus command. If the CLI executes dfocus 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 will focus on process 2.

In the GUI, you set the focus by displaying a Process Window containing this process. Do this by using the P+ *and* P- *icons on the toolbar or by making a selection in the Root Window.*

If you begin a command with dfocus, TotalView changes the target for just the command that follows. After the command executes, TotalView

restores the *old* 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.

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, here is a command with two arenas:

```
dfocus {p1 p2}
```

The two arenas are process 1 and process 2.

So, what is the GOI, POI, and TOI when there is an arena list? In this case, each arena within the list will have its own GOI, POI, and TOI.

Specifying Processes and Threads

A previous section described a P/T set as being a list, but 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 will act; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some CLI output commands, however, will 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.

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 (Pro-

cess 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 number worker thread* in a process and is used instead of the TID value. If, however, the arena explicitly names a thread group, < means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which may not be thread 1; for example, the first and only user thread may be thread number 3 on HP Alpha systems.
- A dot (.) indicates the current set. While this is seldom used interactively, it can be useful in scripts.

Process and Thread Widths

You can enter a P/T set in two ways. If you're not manipulating groups, the format is:

[*width_letter*][*pid*][*.tid*]

The next section extends this format to include groups. When using P/T sets, you can create sets that just width indicators or group indicators or both.

For example, p2.3 indicates process 2, thread 3.

While the syntax seems to indicate that you do not need to enter any element, TotalView requires that you enter at least one. Because TotalView will try to determine what it can do based on what you type, it will try 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 "*Incomplete Arena Specifiers*" on page 218.

The *width_letter* indicates which processes and threads are part of the arena. The letters you can use are:

t	*Thread width* A command's target is the indicated thread.
p	*Process width* A command's target is the process containing the TOI.
g	*Group width* A command's target is the group containing the POI. This indicates control and share groups.

a — *All processes*

A command's target is all threads in the GOI that are in the POI.

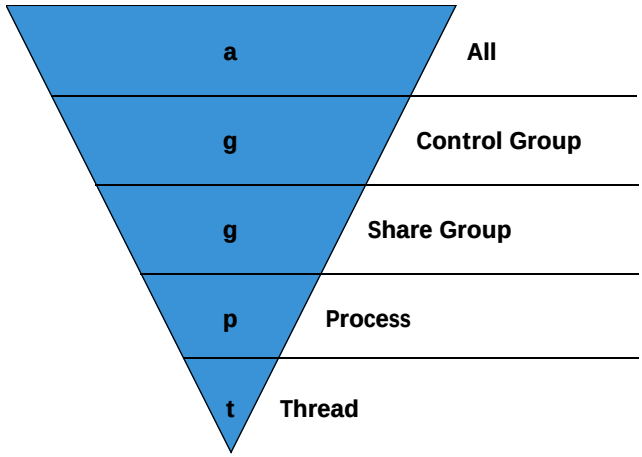
d — *Default width*

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. Default widths are listed in "*Default Arena Widths*" in the *TotalView Reference Guide*.

You must use lowercase letters to enter these widths.

Figure 141 illustrates how these specifiers relate to one another.

Figure 141: Width Specifiers



Notice that the "g" specifier indicates control and share groups. This inverted triangle is indicating 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 example, the dstep command always requires a TOI, but entering this command can:

- Step just the TOI during the step operation (single-thread single-step).
- Step the TOI and step all threads in the process containing the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC (program counter) 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): Some Examples*" on page 219.

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

Here are some sample specifiers:

g2.3	Select process 2, thread 3 and set the width to “group”.
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 “<” sets the TID to the first user thread.

Setting Group Focus

TotalView has two kinds 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:

TotalView has four predefined groups. Two of these only contain processes while the other two only contain threads. As you will see, TotalView also allows you to create your own groups, and these groups can have elements that are processes and threads. The predefined process groups are:

- **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 which subsequently called execve().

  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 > Edit command.

- **Share Group**

  Contains all members of a control group that share the same executable image. TotalView automatically places processes in share groups based on their control group and their executable image.

You can't change a share group's members. In addition, dynamically loaded libraries may vary between share group members.

The predefined thread groups are:

- **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 will create 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 to you, you can ignore this distinction in most cases. 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's 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 multiprocess program, the control group and share group differ only when the program has children that it forked with by calling execve().

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.

Target group specifiers are most often used with the stepping commands, as they give these commands more control over what's being stepped.

Here is how you add groups to an arena specifier:

[*width_letter*][*group_indicator*][*pid*][.*tid*]

This format adds the *group_indicator* to the what was discussed in "*Specifying Processes and Threads*" on page 205.

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 will determine 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. These sets are identified by letters. For example, the following command sets the focus to the workers group:

```
dfocus W
```

The group letter, which is always uppercase, can be:

C	*Control group* All processes in the control group.
D	*Default control group* All processes in the control group. The only difference between this specifier and the C specifier is that D tells the CLI that it should not display a group letter within the CLI prompt.
S	*Share group* The set of processes in the control group that have the same executable as the arena's TOI.
W	*Workers group* The set of all worker threads in the control group.
L	*Lockstep group* A set containing 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 will proceed in lockstep.

A Group Number

You can identify a group by the number TotalView assigns to it. For example, here is how you set the focus to group 3:

```
dfocus 3/
```

Notice the trailing slash. This slash lets TotalView know that you're 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. For example, here is how you would set the focus to the set of threads contained in process 3 and that are also contained in a group called my_group:

```
dfocus p/my_group/3
```

Arena Specifier Combinations

The following table lists what's selected when you use arena and group specifiers to step your program.

Table 10: Specifier Combinations

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.”
gC	All threads in the Thread of Interest’s (TOI) control group.
gS	All threads in the TOI’s share group.
gW	All worker threads in the control group containing the TOI.
gL	All threads in the same share group within the process containing the TOI that have the same PC and 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 tS tW tL	Very little. These four combinations, while syntactically correct, are meaningless. The t specifier overrides the group specifier. So, for example, tW and t both name the current thread.

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 while* 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.

Here are some additional examples. These add PIDs and TIDs to the specifier combinations just discussed:

pW3	All worker threads in process 3.
pW3.<	All worker threads in process 3. Notice that the focus of this specifier is the same as the previous example’s.
gW3	All worker threads in the control group containing process 3. Notice the difference between this and pW3, which restricts the focus to one of the processes in the control group.
gL3.2	All threads in the same *share* 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. As the arena width, POI, and TOI are inherited from the existing P/T set, the exact meaning of this specifier depends on the previous context.

While the "/" 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 process 3's 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.

  Setting the process with an explicit group should be done with care, as what you get may not be what you expect given that commands, depending on their scope, must look at the TOI, POI, and GOI.

Specifying thread width with an explicit group ID probably doesn't mean much as the 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) will operate 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 "<". 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. Here is what runs when you use this arena:

Table 11: a (all) Specifier Combinations

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."

This is the same information as was presented in *"Arena Specifier Combinations"* on page 210. Here are some combinations and the meaning of these 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 *thing* 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's share group have at least one thread at the goal. Processes in other control groups run freely until this happens.

  Notice that 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 until command because, as was just mentioned, the goals for f aC until and f aS until are the same, and the processes that are in this scope are identical.

  While more than one share group can exist in a control group, these other share groups do not participate in the goal.

f aL duntil
: While everything will run, it is again 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 will run to a goal as just those thread in the TOI's lockstep group. While

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 — While everything will run, TotalView will wait until all members of the TOI's workers group arrive at the goal.

There are two broad distinctions between process and thread group behavior:

- 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, and TotalView doesn't care where they end up.

  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. Of course, 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. Many of the examples use CLI commands that have not yet been introduced. You will probably need to refer to the command's definition before you can appreciate what's occurring. These commands are described in the *TotalView Reference Guide*.

If you will only be using the GUI, there's nothing you need to know in this section as groups are created graphically there.

Here are some methods for indicating 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.

Here 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 containing all the threads in process 2's share group except for thread 2.3 and then continues that set of threads. The first command creates a new group containing 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'll most often use g, is that it forces the group when the default focus indicates something different from what you want it to be.

Here's an example that shows this. The first step is to set a breakpoint in a multithreaded 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 "tx_omp_guide_llnl1"
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  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
d1.<> dfocus p dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
```

Here's what the CLI displays when you ask for the status of the lockstep group. (The rest of this example will use the `f` abbreviation for `dfocus` and `st` for `dstatus`.)

```
d1.<> f L st
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
```

This command tells the CLI to display the status of the threads in thread 1.1's lockstep group as 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 next command runs thread 1.3 to the same line as thread 1.1. The next 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  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
```

As expected, the CLI has added a thread to the lockstep group:

```
d1.<> f L dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.3: 37258.3  Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.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  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
```

While 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.

```
t1.<> f pL dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
t1.<> f gL dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./tx_omp_llnl1.f#35]
t1.<>
```

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 dstatus. If, however, there were additional processes in the group, you would only see them when you use the gL specifier.

Focus Merging

When you specify more than one focus for a command, the CLI will merge them together. In the following example, the focus indicated by the prompt—this focus is called the *outer* focus—controls the display. Notice what happens when dfocus commands are strung together:

```
t1.<> f d
d1.<
```

```
d1.<> f tL dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
d1.<> f tL f p dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
d1.<> f tL f p f D dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
d1.<> f tL f p f D f L dstatus
1:       37258   Breakpoint  [tx_omp_guide_llnl1]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                             [./tx_omp_llnl1.f#35]
d1.<>
```

Stringing multiple focuses together may not produce the most readable result. In this case, it shows how one `dfocus` command can modify what another sees and will act on. The ultimate result is an arena that a command will act on. In these examples, the `dfocus` command is telling the `dstatus` command what it should be displaying.

Incomplete Arena Specifiers

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. Here is 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 `<`.)

  You can type a PID without typing a TID. If you omit the TID, TotalView uses its default of "`<`", 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 lets TotalView know if you're specifying 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.

Of course, focus merging can also influence how TotalView fills in missing specifiers. For more information, see "*Focus Merging*" on page 217.

Lists with Inconsistent Widths

TotalView lets you create lists containing more than one width specifier. While this can be very useful, 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: a command 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 will act. 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. It may wait for threads in multiple processes for group-level arenas. The command description in the *TotalView Reference Guide* will point out anything that you need to watch out for.

Stepping (Part II): Some Examples

Here are some examples of things that you'll probably do using the CLI's stepping commands:

- **Step a single thread**

  While the thread runs, no other thread runs (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**

  Runs worker threads 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 from the entire share group. This differs from the previous two bullets in that TotalView uses the set of threads that 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 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 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 kind 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 the sets.
-	Creates a difference; that is, all members of the first set that are not also members of a second set.

&	Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, here is how you would create a union of two P/T sets:

```
p3 | L2
```

A set operator only operates on two sets. You can, however, apply these operations repeatedly. For example:

```
p2 | p3 & L2
```

This statement creates a union between p2 and p3, and then creates an intersection between the union and L2. As this example suggests, TotalView associates sets from left to right. You can change this order by using parentheses. For example:

```
p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint()	Returns a list of all threads that are stopped at a breakpoint.
error()	Returns a list of all threads stopped due to an error.
existent()	Returns a list of all threads.
held()	Returns a list of all threads that are held.
nonexistent()	Returns a list of all processes that have exited or which, while loaded, have not yet been created.
running()	Returns a list of all running threads.
stopped()	Returns a list of all stopped threads.
unheld()	Returns a list of all threads that are not held.
watchpoint()	Returns a list of all threads that are stopped at a watchpoint.

The argument that all of these operators use is a P/T set. You specify this set in the same way that a P/T set is specified for the dfocus command. For example, watchpoint(L) returns all threads in the current lockstep group.

The dot (.) operator, which indicates the current set, can be helpful when you are editing an existing set.

The following examples should clarify how you use these operators and functions. The P/T set that is the argument to these operators is a (all).

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.

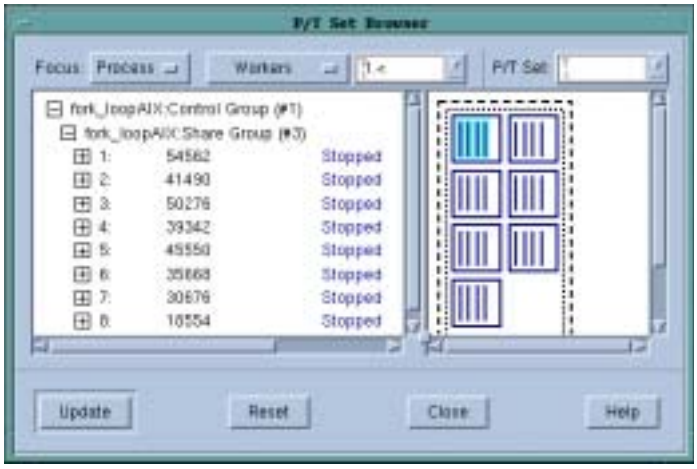
f {g.3 - p6} duntil 577
: Runs thread 3 along with all other processes in the group to line 577. However, do not run anything in process 6.

f {($PTSET) & p123}
: Uses just process 123 within the current P/T set.

Using the P/T Set Browser

There's no question that specifying P/T sets can be confusing. As has been mentioned, there are few programs that need all the power that TotalView's P/T set syntax provides. In all cases, however, the ability to previsualize what the contents of a P/T set will be before you execute the command is essential. This is what the P/T Set Browser is designed to do. The browser, which is accessed from the Root Window's Tool menu, shows the current state of processes and threads as well as show what is or will be selected when you specify a P/T set. Figure 142 shows a P/T browser displaying information about a a multiprocess, multithreaded program.

FIGURE 142: *A P/T Set Browser Window*

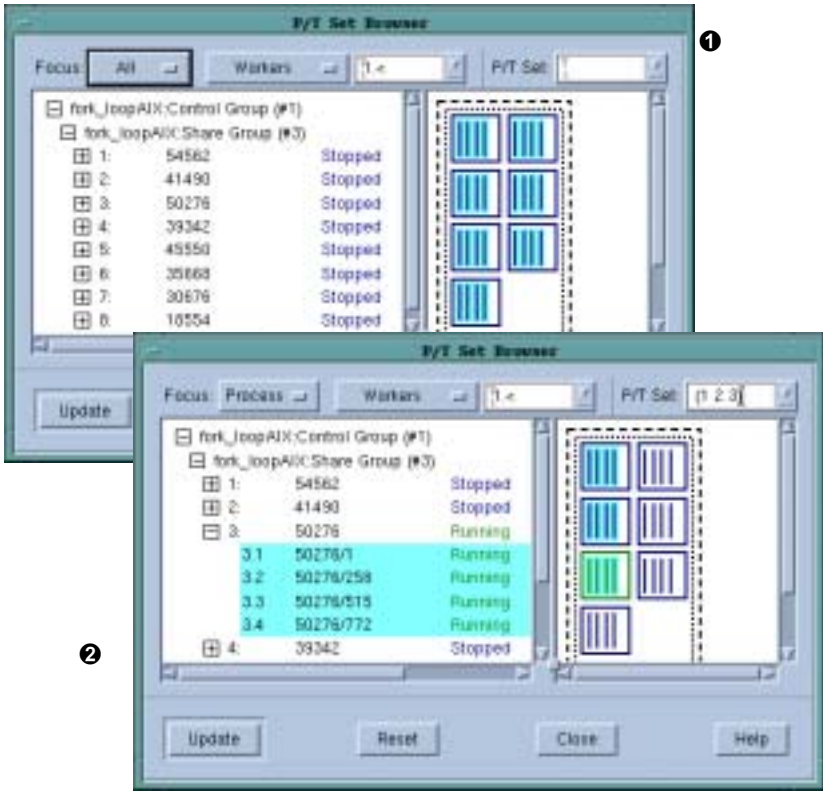


The top part of this window contains the standard P/T set controls. (See "*Using P/T Set Controls*" on page 202 for more information.) The large area on the left is a "tree" control where clicking on the "+" shows more information, and clicking on a "–" (not shown in this figure) condenses the information. Here you will find a list of all your program's processes and threads. The information is organized in a hierarchy, with the outermost level being your program's control groups. In a control group, information is further organized by share group, where you are shown the processes contained in a share group. Finally, if the innermost "+" symbols were clicked, the browser would show information on the threads within a process.

The control and share group numbers displayed in this window are the same as those that are displayed in the Groups Page in the Root Window.

The right-hand side contains a graphical depiction of your program's threads. In the preceding figure, notice that TotalView has highlighted some of the threads. These are the threads in the current focus, which in this case is "1.<". As you make changes to the P/T set, the threads highlighted in the right-hand side change, showing you what the scope of a P/T set definition is. The next figures contains a variety of P/T set examples.

Figure 143: P/T Set Browser Windows (Part 1)



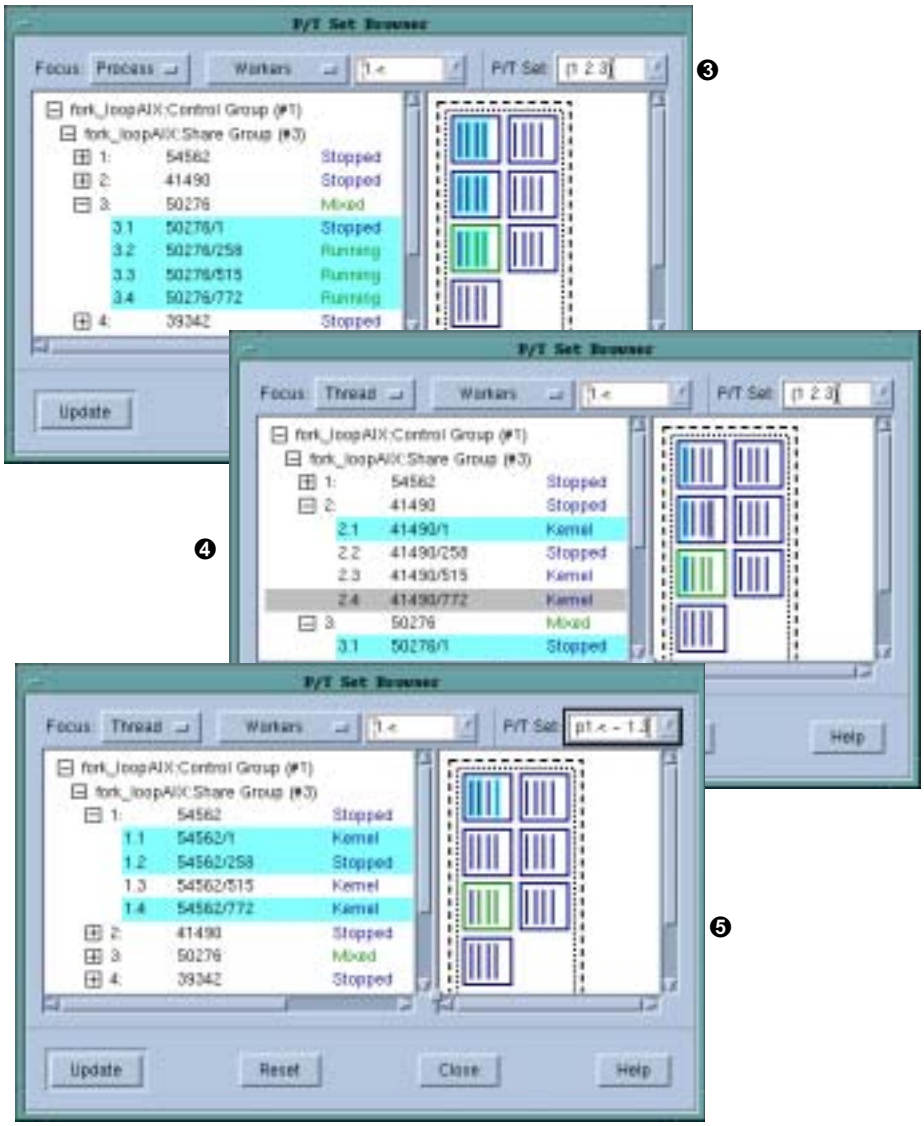
❶ This P/T set displayed differs from the one in Figure 142 on page 222 in that the Focus pulldown menu is now set to All. TotalView responds by highlighting all threads on the right-hand side.

❷ The Focus pulldown menu was changed back to Process but the number of processes was limited to 1, 2, and 3. Before these changes were made, process 3 was told to go. As you can see, the browser shows those processes as running.

❸ Thread 3.1 was halted.

❹ Thread 2.4 was selected with the mouse. It doesn't matter if it was selected in the left or right-hand sides, as selecting it causes it to be highlighted in both. After selecting a thread, you can extend the selection by clicking your mouse's left button on another thread

Figure 144: P/T Set Browser Windows (Part 2)



while holding down the Shift key. You can select non-contiguous threads by holding down the Control key while clicking your mouse's left button.

If you are seeing this document online, you'll notice that your selection is in gray while the selection indicating the P/T set is in blue.

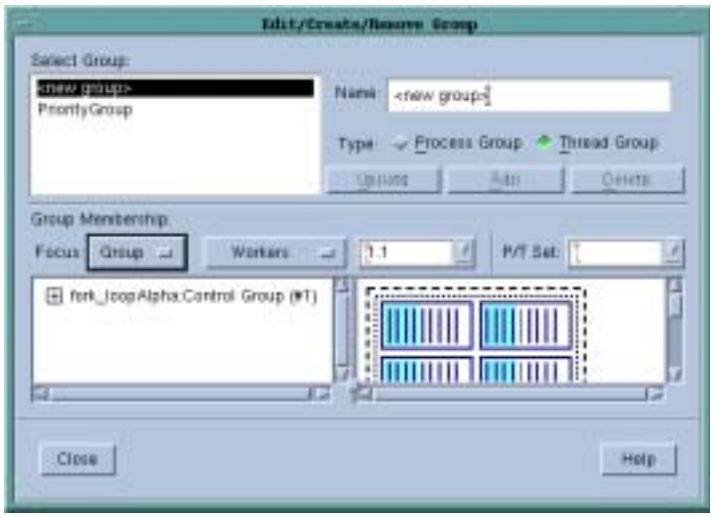
❺ The P/T set information was modified to show a difference expression; in this case, thread 1.3 was eliminated from the set of threads named by "p1.<".

The elements on the right side are drawn within two boxes. These boxes represent the control and share groups. Clicking on them tells the browser to select that group.

Using the Group Editor

The visual group editor, which is displayed after you select the Group > Edit Group command, can simplify the way in which you create named groups.

Figure 145: Group > Edit Group



This dialog box can be divided into two halves. The top half allows you to add, update, and delete named sets. The bottom half contains controls that allow you to specify which processes and threads will become part of the group. These controls are discussed in the previous section.

The controls in the upper portion work generally as you'd expect them to. The only thing to be careful about is that you must define the group, and be sure to give the group a name, before you click on the Add button. Details on using the controls in this dialog box are contained in the online Help.

12 Examining and Changing Data

This chapter explains how to examine and change data as you debug your program. The topics discussed in this chapter are:

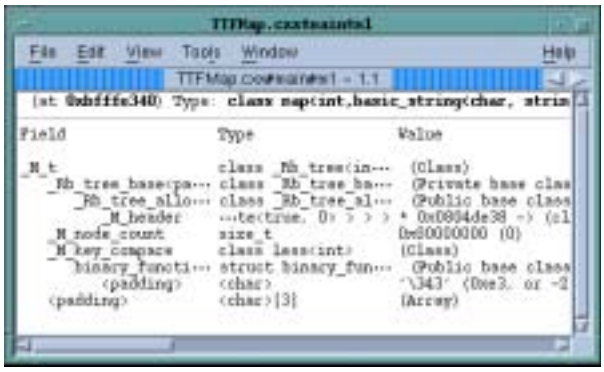
Changing How Data Is Displayed

One of the problems you'll face is that TotalView, like all debuggers, displays data in the way that your compiler stored it. The following two sections let you change the the way TotalView displays this information. These sections are:

Displaying STL Variables

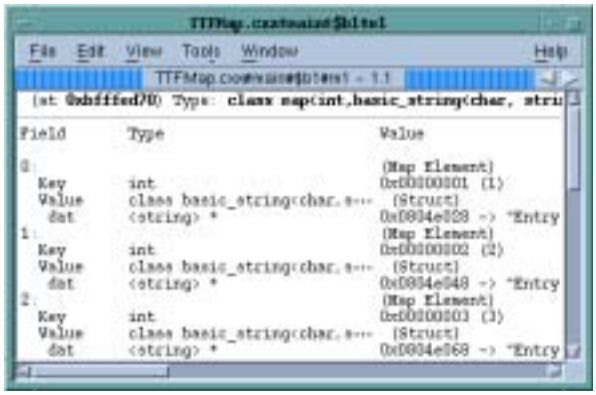
The C++ STL (Standard Template Library) greatly simplifies the way in which you can access data. By offering standard and prepackaged ways to organize data, you do not have to be concerned with the mechanics of the access method. The only real downside to using the STL is while debugging. This is because the information you are shown is the compiler's view of your data rather than the STL's logical view. For example, Figure 146 on page 228 shows how your compiler sees a map compiled using GNU C++:

Figure 146: An Untransformed Map



TotalView comes with a set of transforms that changes how it displays some STL data. For example, Figure 147 shows the transformed map.

Figure 147: A Transformed Map



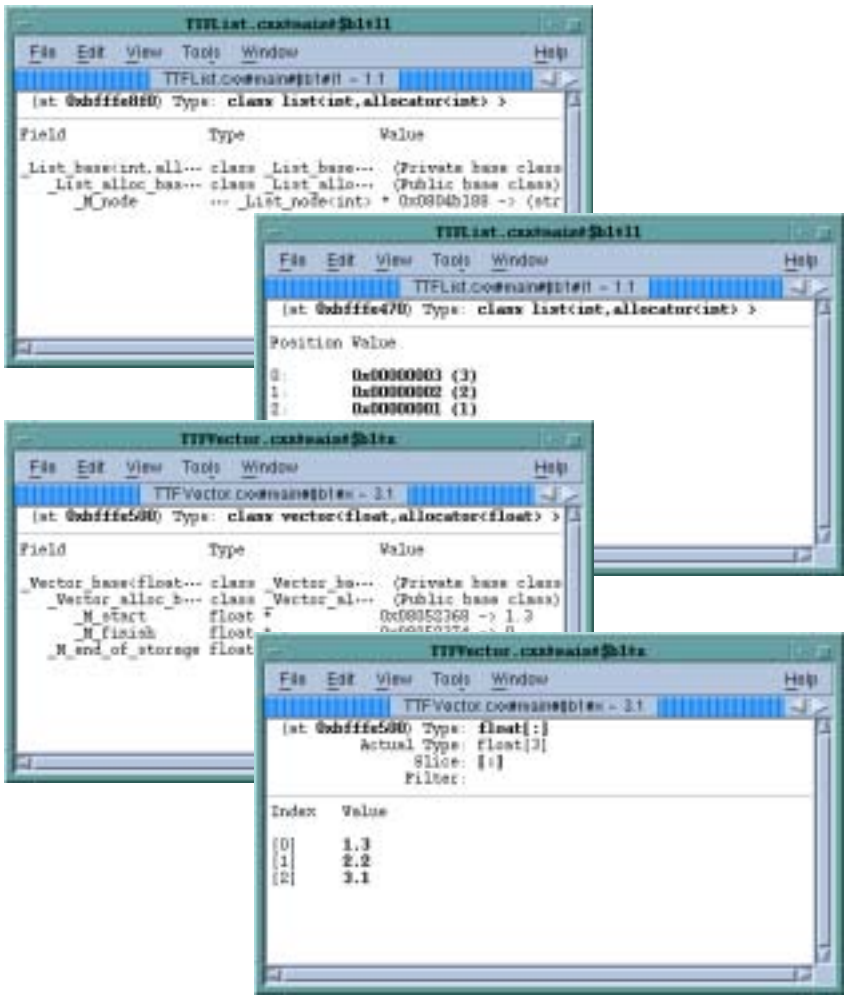
TotalView can transform STL vectors, lists, and maps using native and GCC compilers on IBM AIX, IRIX/MIPS, and HP Tru64 Alpha. It also supports GCC and Intel's Version 7 C++ 32-bit compiler running on Red Hat x86 platform.

Figure 148 on page 229 shows an untransformed and transformed list and vector.

You can create your own transformations. The process for doing this is described in the "Creating Type Transformations Guide".

By default, TotalView transforms these data structures. If you do not want them transformed, uncheck the View simplified STL containers (and user-

Figure 148: List and Vector Transformation



defined transformations) checkbox within the File > Preference's Options Page.

CLI: dset TV::ttf { true | false }

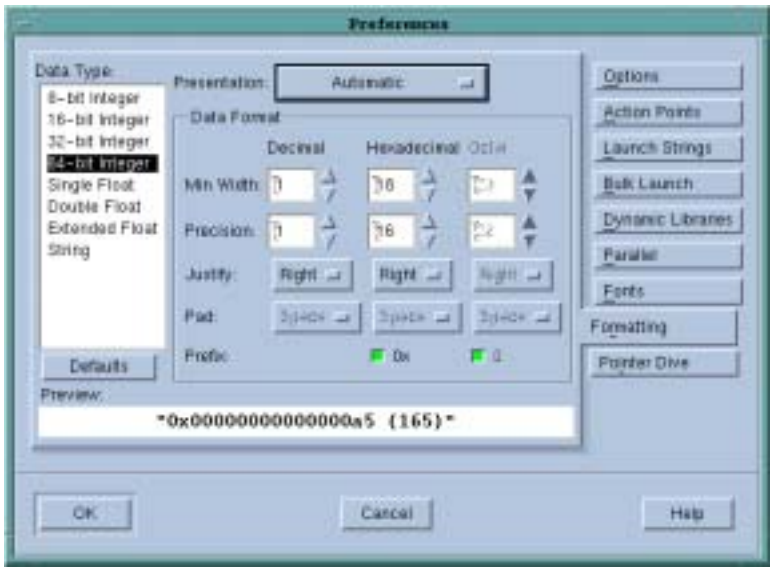
Changing Size and Precision

In most cases, TotalView does a reasonable job of displaying a variable's value. If TotalView's defaults don't meet your needs, you can indicate the precision at which to display simple data types by using the Formatting Page of the File > Preferences Dialog Box. (See Figure 149 on page 230.)

After selecting one of the data types listed on the left, you can set how many character positions a value will use when TotalView displays it (Min Width) and how may numbers it should display to the right of the decimal place (the Precision). You can also tell TotalView how it should align the value in the Min Width area and if it should pad numbers with zeros or spaces.

While the way in which these controls relate and interrelate may appear to be complex, the Preview area shows you exactly the result of a change.

Figure 149: File > Preferences: Formatting Page



After you play with the controls for a minute or so, what each control does will be clear. You will probably 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 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

TotalView displays variables that are local to the current stack frame in the Process Window's Stack Frame Pane. For non-simple variables—for example, pointers, arrays, and structs—this pane doesn't show the data; instead, you need to dive on the variable to bring up a Variable Window that contains the variable's information. 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 a three buttons, you can either single- or double-click on an item, depending upon context.

If you dive on simple variables or registers, TotalView still brings up a Variable Window. In this case, you'll see some additional information about the variable.

Topics in this section are:

Displaying Program Variables

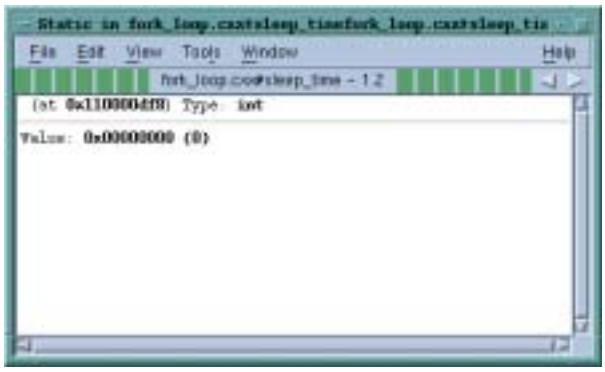
You can display local and global variables by:

- Diving into the variable in either the Source or Stack Panes.
- Selecting the View > Lookup Variable command. When prompted, enter the name of the variable.

CLI: dprint *variable*

A Variable Window appears for the global variable. (See Figure 150.)

Figure 150: Variable Window for a Global Variable



Displaying Variables in the Current Block

In many cases, you're not really interested in just seeing one variable. Instead, you want to see all of the variables in the current block. If you dive on a block label within the Stack Frame Pane, TotalView opens a Variable Window containing just those variables. See Figure 151 on page 232.

You can dive on any variable in this window to see more information.

Browsing for Variables

The Process Window's Tools > Program Browser command displays a window containing all your executable's components. By clicking on a library or program name, you can access all of the variables contained within it. (See Figure 152 on page 232.)

The window in the upper left corner shows programs and libraries that are loaded. If you have loaded more than one program with the File > New Program command, only the currently selected process list will appear. The center window contains a list of files that make up the program as well as other related information. Diving again on a line displays a Variable Window that contains variables and other information related to the file. Figure 153 on page 233 shows three more diving operations.

Figure 151: Displaying Scoped Variables

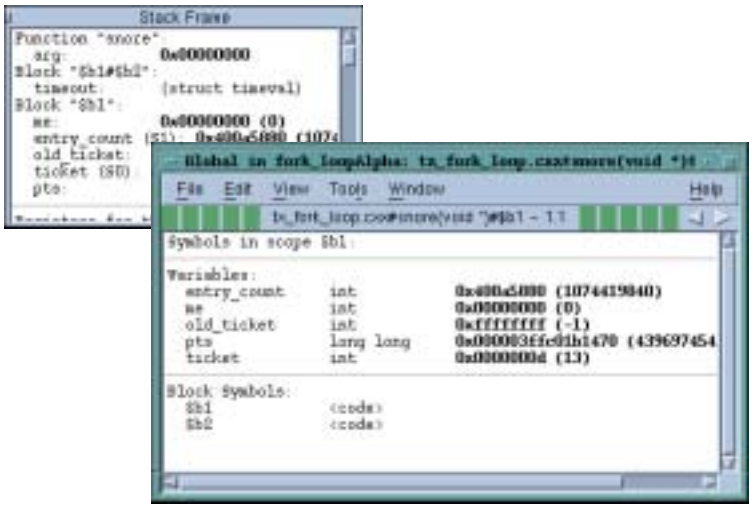
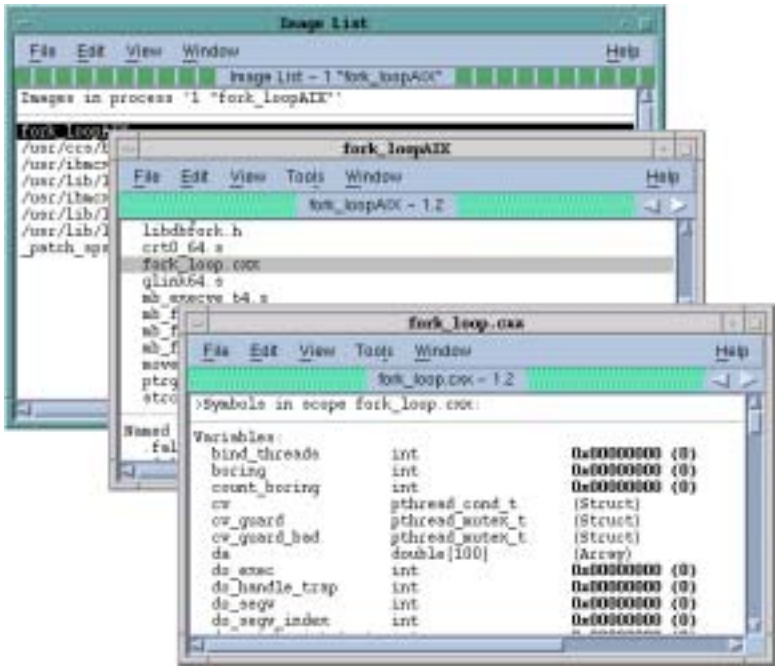


Figure 152: Program Browser and Variables Window (Part 1)

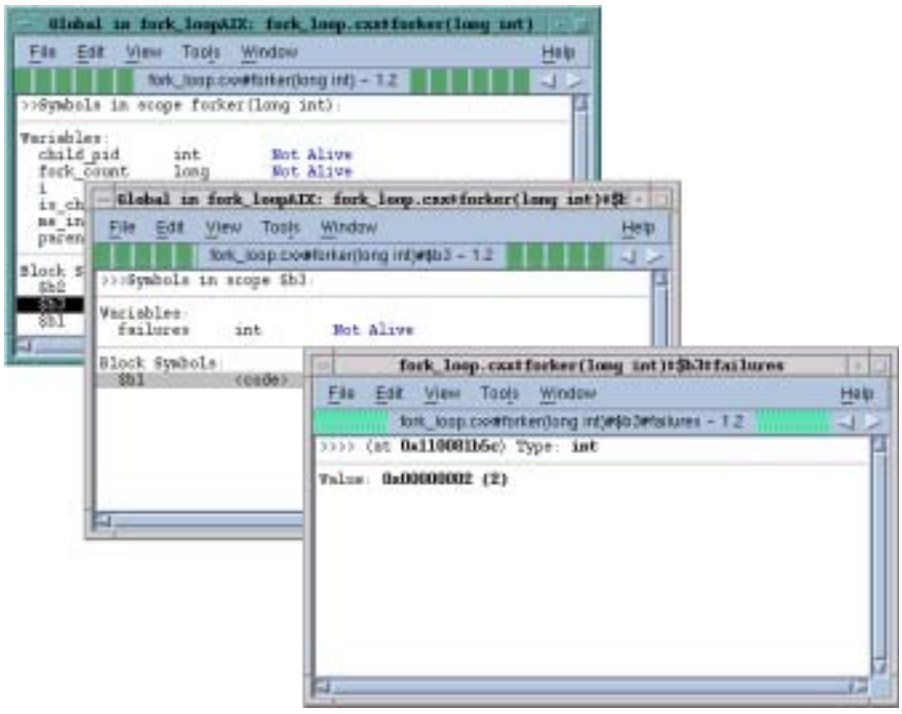


The screen in the upper-left corner shows a Variable Window created by diving on one of the files in Figure 153 on page 233. The center screen dives on a block in that subroutine. Finally, the screen in the lower-right corner shows a variable. (These screens were created using the View > Dive Anew command.) If you just dived on a line in a Variable Window, the new contents replace the old contents, and you can use the dive/undive icons to move back and forth.

Displaying Local Variables and Registers

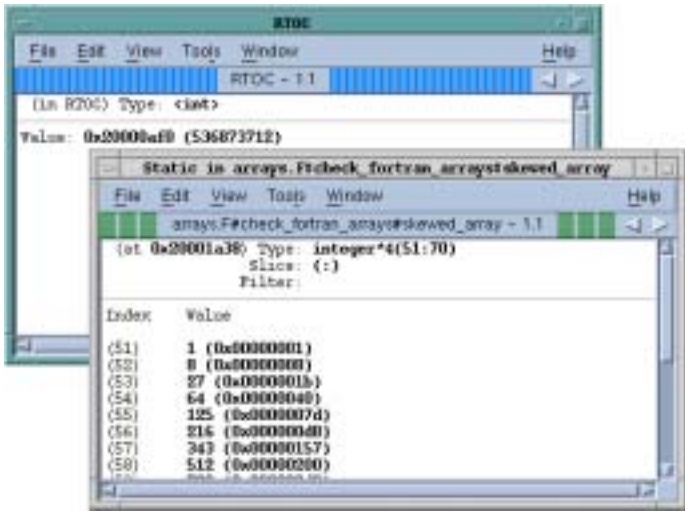
In the Stack Frame Pane, diving into a formal parameter, local variable, or register tells TotalView to display a Variable Window. You can also dive into parameters and local variables in the Source Pane. The displayed Variable

Figure 153: Program Browser and Variables Window (Part 2)



Window shows the name, address, data type, and value for the object. (See Figure 154.)

Figure 154: Diving into Local Variables and Registers



The top-left window shows the result of diving on a register, while the bottom-right 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 a local variable by using the View > Lookup Variable command. When prompted, enter the name of the variable in the dialog box that appears.

If Variable Windows remain open while a process or thread runs, TotalView updates the information in these windows when the process or thread stops. If TotalView can't find a stack frame for a displayed local variable, it displays Stale in the pane's header to warn you that you can't trust the data, since the variable no longer exists.

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'll need to explicitly open a new Variable Window to see the value of a local variable in that stack frame.

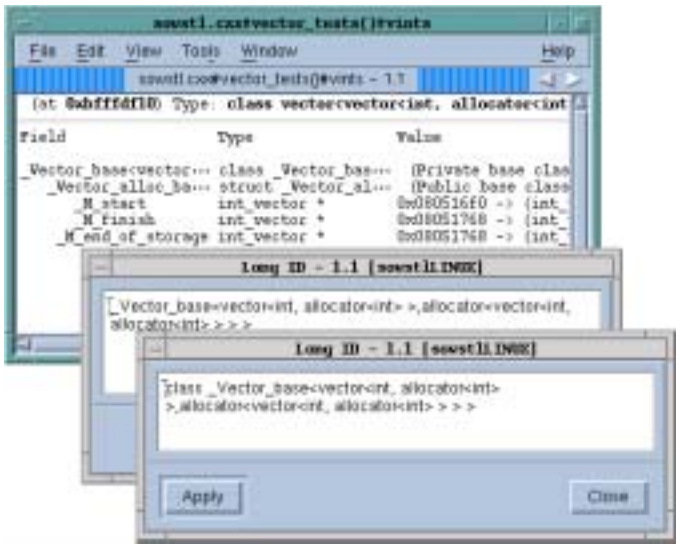
CLI: dwhere, dup, and dprint

You'll locate the stack frame using dwhere, move to it using dup, and then display the value using dprint.

Displaying Long Variable Names

If TotalView doesn't have enough space to display all the characters in a variable name, it inserts ellipses (...) to indicate that it has truncated the name. Typically, this occurs when it is displaying demangled C++ names or STL variables. Figure 155 shows three windows. The top-left Variable Window contains a series of STL names. The other two windows show what TotalView displays when you click on the ellipses. Notice that one of the windows has an Apply button. This indicates that the field is editable.

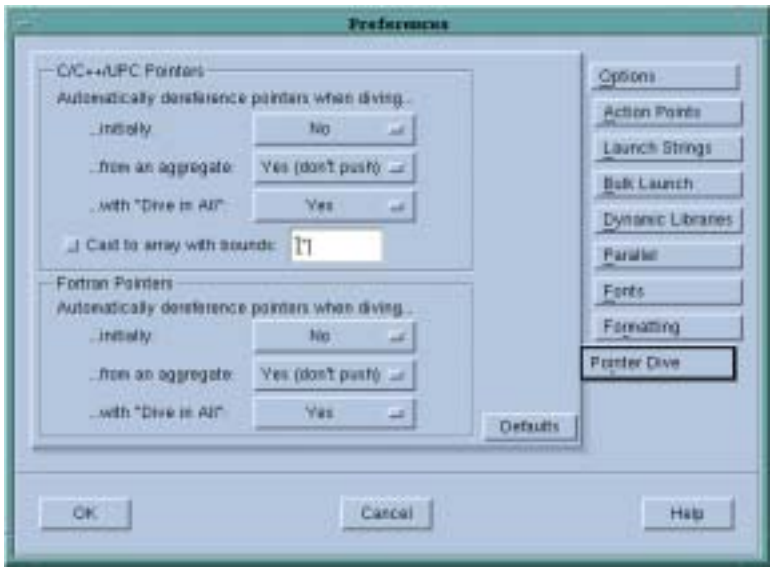
Figure 155: Displaying Long STL Names



Automatic Dereferencing

In most cases, you aren't interested in the value contained in a pointer variable. Instead, you want to see what the pointer is pointing to. Using the controls contained in the File > Preferences's Pointer Dive Page, you can tell TotalView if it should automatically dereference pointers. (See Figure 156 on page 235.)

Figure 156: File > Preferences: Pointer Dive Page



This preference is especially useful when you want to visualize data that is linked together with pointers, as it can present the data as a unified array. Because the data appears to be a unified array, you can use TotalView's array manipulation commands and the Visualizer to view this data.

Each pulldown list has three settings: No, Yes, and Yes (don't push). The meaning for No is obvious: automatic dereferencing will not occur. Both of the remaining values tell TotalView that it should 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 you set this to Yes, you can see the value. Setting it to Yes (don't push) means you can't use the Back command to see the 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

The three situations in which automatic dereferencing can occur are:

- When TotalView *initially* displays a value.
- When you dive on a value within an *aggregate* or structure.
- When you use the Dive in All command.

Displaying Areas of Memory

You can display areas of memory in hexadecimal and decimal values. Do this by selecting the View > Lookup Variable command and then entering one of the following in the dialog box that appears:

- **A hexadecimal address**

  When you enter a single address, TotalView displays the word of data stored at that address.

  CLI: dprint *address*

- **A pair of hexadecimal addresses**

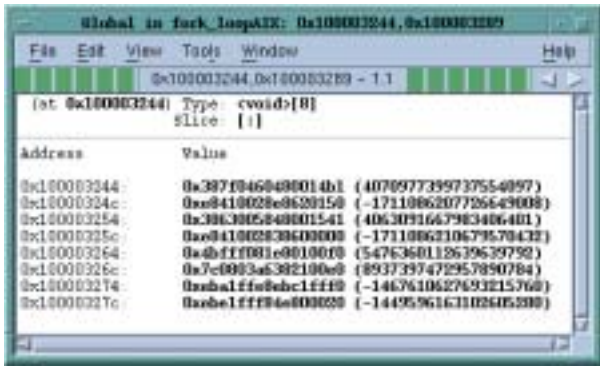
  When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

  CLI: dprint *address*,*address*

All hexadecimal constants must have a "0x" prefix. You can use an expression to enter these addresses.

The Variable Window for an area of memory displays the address and contents of each word. (See Figure 157.)

Figure 157: Variable Window for 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.

Displaying Machine Instructions

You can display the machine instructions for entire routines as follows:

- Dive into 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 dived into.
- Dive into the PC in the Stack Frame Pane. A Variable Window lists the instructions for the entire function containing the PC, and highlights the instruction the PC points to. (See Figure 158 on page 237.)
- Cast a variable to type <code> or array of <code>, as described in "*Changing Types to Display Machine Instructions*" on page 249. (See Figure 159 on page 237.)

Figure 158: Variable Window with Machine Instructions

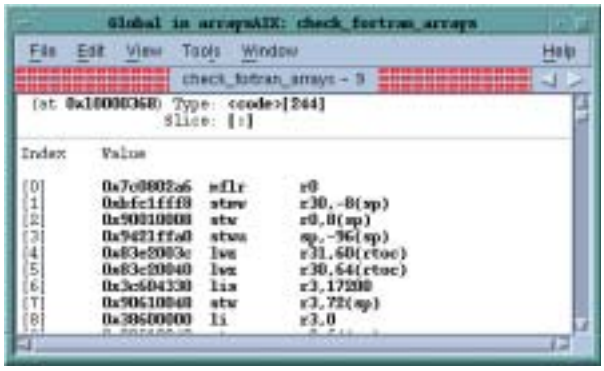
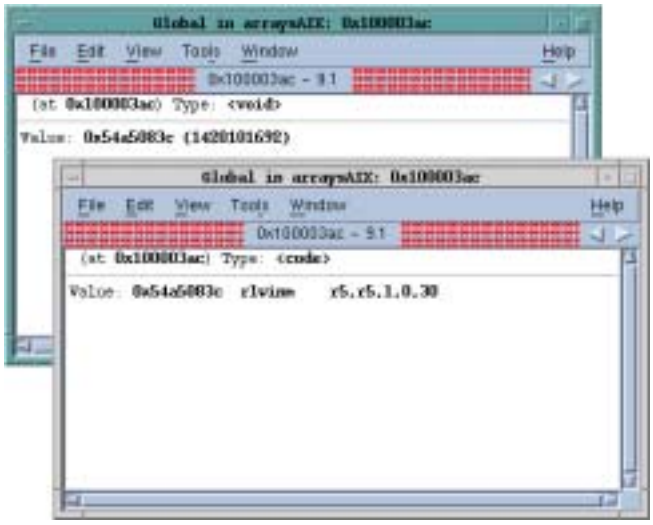


Figure 159: Casting Code



Closing Variable Windows

When you're finished 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.

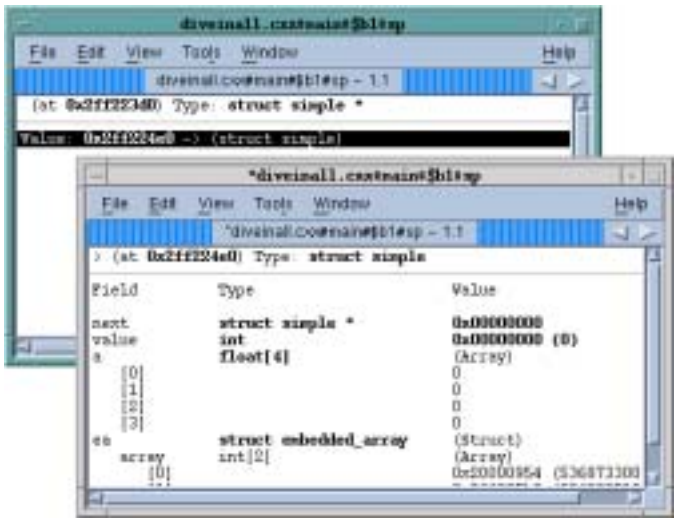
Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive into a value shown in the Variable Window. 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 non-simple data types, you can dive on these data types. While a typical data structure doesn't have too many levels, repeatedly diving on data lets you follow pointer chains. That is, diving allows you to see the elements of a linked list.

TotalView remembers your dives. This means you can use the "undive" and "redive" buttons  as a convenient way to see other dive results.

Figure 160 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 the window in the upper left corner.)

Figure 160: Nested Dives



The nested dive window—displayed in the bottom right corner of the figure—shows the structure referenced by the simple* pointer.

You can manipulate Variable Windows and nested dive windows in the following ways:

- To “undive” from a nested dive, select the left-facing arrow in the upper right-hand corner of the Variable Window. After clicking on the arrow, the previous contents of the Variable Window appears.
- To “redive” after you “undive,” select the right-facing arrow in the upper right-hand corner of the Variable Window. After clicking on the arrow, TotalView performs a previously executed dive operation.
- If you dive into a variable that already has a Variable Window open, the Variable Window pops to the top of the window display.
- If you have performed several nested dives and want to create a new copy of the base window, select the Window > Duplicate Base command.
- If you select the Window > Duplicate command, a new Variable Window appears that is a duplicate of the current Variable Window. It differs internally as it has an empty dive stack.

Displaying Array of Structure Elements

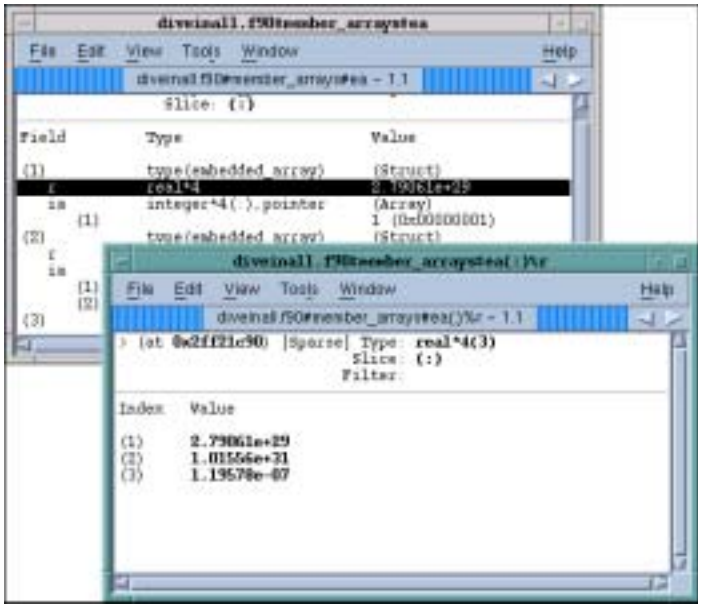
The View > Dive In All command (which is also available when you right-click on a field) allows you to display an element in an array of structures as if it were a simple array. For example, suppose you have the following Fortran definition:

```
type embedded_array
   real r
   integer, pointer :: ia(:)
end type embedded_array
```

```
type(embedded_array) ea (3)
```

After selecting an r element, select the View > Dive In All command, TotalView displays all three r elements of the ea array as if it were a single array. (See Figure 161.)

Figure 161: Displaying a Fortran Structure



The View > Dive in All command can also display the elements of C array of structures as arrays. Figure 162 on page 240 shows TotalView displaying a unified array of structures and a multidimensional array in a structure.

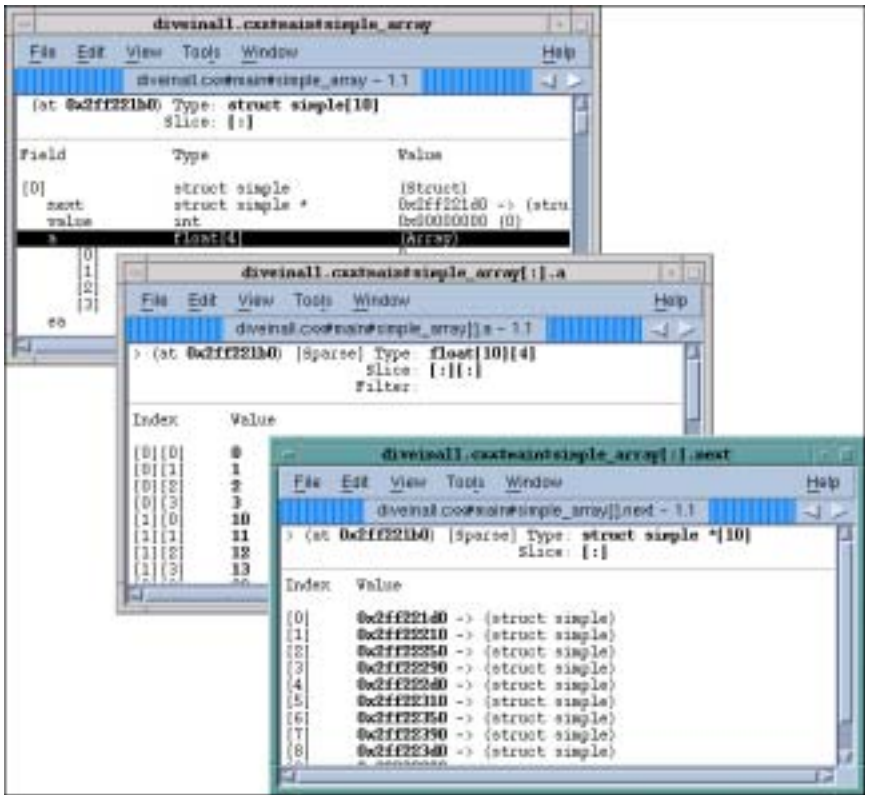
As TotalView's array manipulation commands (which are described in Chapter 8) work on what's displayed and not what is stored in memory, you can operate on an array created by this command in the same manner as any other array. For example, you can visualize the array, obtain statistics about it, filter elements in it, and so on.

Scoping and Symbol Names

Many CLI and some GUI commands have arguments whose elements are variables and other things found in your program. TotalView assigns a unique name to all of your program's element based on the scope in which the element exists. A *scope* defines what part of a program knows about a symbol. For example, the scope of a variable that is defined at the beginning of a subroutine is all statements in the subroutine. The variable's scope does not extend outside of this subroutine. A program consists of *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 most trivial program, scopes are embedded in other scopes. The exception is, of

Figure 162: Displaying C Structures and Arrays



course, the top-most scope. Every element in a program is associated with a scope.

Whenever you tell the CLI or the GUI to execute a command, TotalView consults the program's symbol table to discover what object you are referring to—this process is known as *symbol lookup*. A symbol lookup is performed with respect to a particular context, and each context uniquely identifies the scope to which a symbol name refers.

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 this is done. It 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 kinds of things that can appear in your program, a formal specification of what can appear and the order in which things can appear complicated, and, unreadable. After you see the name, in the Stack Frame Pane, you'll know a variable's scoped name.

TotalView interprets the components as follows:

- Just as file names need not be qualified with a full path, you do not need to use all levels in a symbol's scoping tree.
- 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 within the root directory). If you omit the executable or library component, the qualified symbol doesn't begin with `#`.
- The source file's name may 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 letter `b` followed by a number indicating which block. For example, the first unnamed block is named `#b1`, the second as `#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 within 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 within 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 occurring in the same file. In this case, you would need to say `bar.c#b1#x` to identify the scope that corresponds to the "outer level" of the file (that is, the scope containing line 1 of the file).

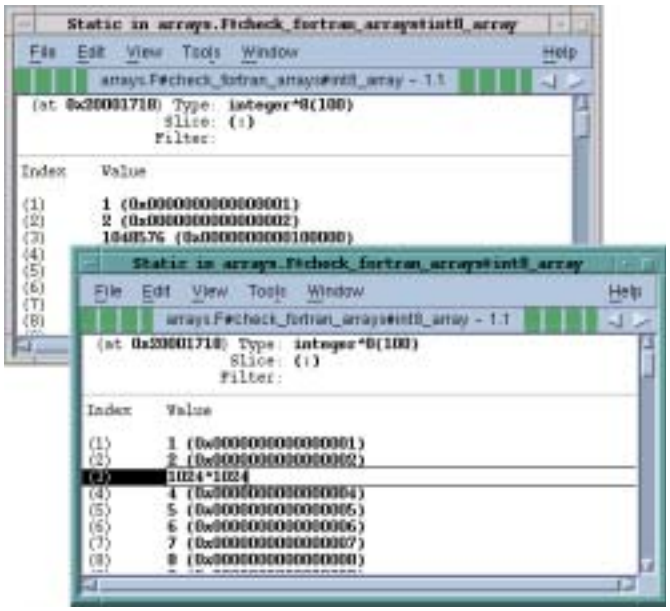
Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window 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 1024*1024 as shown in Figure 163 on page 242. You can include logical operators in all TotalView expressions.

CLI:
```
set my_var [expr 1024*1024]
dassign int8_array(3) $my_var
```

If a value is displayed in bold in the Stack Frame Pane, you can edit the value.

Figure 163: Using an Expression to Change a Value



While TotalView does not let you directly change the value of bit fields, the Tools > Evaluate Window lets you assign a value to a bit field. See "*Evaluating Expressions*" on page 298. Similarly, you cannot directly change the value of fields in nested structures; you must first dive into the value. When TotalView displays a value in bold, it is ready to be edited.

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.

Topics in this section are:

You can change the way TotalView displays data in the Variable Window by editing its 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 variable is displayed in TotalView, the data types are identical to C type representations, except for pointers to arrays. TotalView uses a simpler syntax for pointers to arrays. (See "*Pointers to Arrays*" on page 243.)
- When Fortran is displayed in TotalView, the types are identical to 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 fields listed in the window.

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 Data Types

TotalView's syntax for displaying data is identical to C Language cast syntax for all data types except pointers to arrays. That is, you should use C Language cast syntax for int, short, unsigned, float, double, union, and all named struct types. In addition, TotalView has a built-in type called <string>. Unless you tell it otherwise, it maps char arrays into this type.

TotalView types are read from right to left. For example, <string>*[20]* is a pointer to an array of 20 pointers to <string>.

Table 12 shows some common data types.

Table 12: Common Types

Data Type String	Meaning
int	Integer
int*	Pointer to 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 also enter C Language cast syntax verbatim in the type field for any type.

The following sections discuss the more complex types.

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]*
```

Arrays

Array type names can include a lower and upper bound separated by a colon (:).

See Chapter 12, "Examining and Changing Data," on page 227 for a detailed discussion of arrays.

By default, the lower bound for a C or C++ array is 0, and the lower bound for Fortran 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).

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 the array use the default lower bound for Fortran, 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 would 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 in exactly 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 matching a filter expression. Refer to "*Displaying Array Slices*" on page 259 and *"Array Data Filtering*" on page 262 for further information.

Typedefs

TotalView recognizes the names defined with typedef, and displays these user-defined types. For example:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

TotalView will display the type for p_vbl as dptr_t.

Structures

TotalView lets you use the struct keyword as part of a type string. In all cases, it is usually optional. 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 name a structure using typedef, the debugger uses the typedef name as the type string. Otherwise, the debugger uses the structure tag for the struct.

For example, consider the structure definition:

```
typedef struct mystruc_struct {
   int field_1;
   int field_2;
} mystruc_type;
```

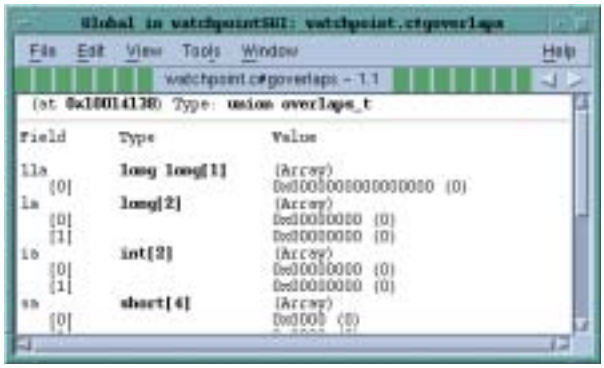
TotalView displays mystruc_type as the type for struct mystruc_struct.

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 164.)

CLI: dprint *variable*

Figure 164: Displaying a Union



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 types could yield undesirable results. TotalView will try very hard to give you what you say, so if you get it wrong, the results are quite unpredictable. Fortunately, the remedy is quite simple: delete the Variable Window and start over again.

Built-In Types

TotalView provides a number of predefined types. These types are enclosed in angle brackets (< >) to avoid conflict with types contained in a programming language. You can use these built-in types anywhere you can use ones defined in your programming language. These types are also useful when debugging executables with no debugging symbol table information. The following table lists the built-in types

Table 13: Built-in Types

Type String	Language	Size	Meaning
<address>	C	void*	Void pointer (address)
<char>	C	char	Character
<character>	Fortran	character	Character

Type String	Language	Size	Meaning
<code>	C	architecture-dependent	Machine instructions The size used here is the number of bytes required to hold the shortest instruction for your computer.
<complex>	Fortran	complex	Single-precision floating-point complex number. complex types contain a real part and an imaginary part, which are both of type real.
<*complex*8*>	Fortran	complex*8	real*4-precision floating-point complex number complex*8 types contain a real part and an imaginary part, which are both of type real*4.
<complex*16>	Fortran	complex*16	real*8-precision floating-point complex number 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
<extended>	C	long double	Extended-precision floating-point number Extended-precision numbers must be supported by the target architecture.
<float>	C	float	Single-precision floating-point number
<int>	C	int	Integer
<integer>	Fortran	integer	Integer
<integer*1>	Fortran	integer*1	One-byte integer
<integer*2>	Fortran	integer*2	Two-byte integer
<integer*4>	Fortran	integer*4	Four-byte integer
<integer*8>	Fortran	integer*8	Eight-byte integer
<logical>	Fortran	logical	Logical
<logical*1>	Fortran	logical*1	One-byte logical
<logical*2>	Fortran	logical*2	Two-byte logical
<logical*4>	Fortran	logical*4	Four-byte logical
<logical*8>	Fortran	logical*8	Eight-byte logical
<long>	C	long	Long integer
<long long>	C	long long	Long long integer

Type String	Language	Size	Meaning
<real>	Fortran	real	Single-precision floating-point number NOTE When using a value such as real, be careful that the actual data type used by your computer is not real*4 or real*8 as different results could occur.
<real*4>	Fortran	real*4	Four-byte floating-point number
<real*8>	Fortran	real*8	Eight-byte floating-point number
<real*16>	Fortran	real*16	Sixteen-byte floating-point number
<short>	C	short	Short integer
<string>	C	char	Array of characters
<void>	C	long	Area of memory

The next sections contain more information about the following built-in types:

- Character Arrays (<string> Data Type)
- Areas of Memory (<void> Data Type)
- Instructions (<code> Data Type)

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 will display 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 C character arrays represent strings, the <string> type can be very convenient. If this isn't what you want, you can edit the <string> back to a char (or char[*n*]) to display the variable as you declared it.

Areas of Memory (<void> Data Type)

TotalView uses the <void> 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 in the C language.

If you dive into registers or display an area of memory, TotalView lists the contents as a <void> data type. Furthermore, 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 in displaying large areas of memory.

If desired, you can change a <void> into another type. Similarly, you can change any type into a <void> to see the variable in decimal and hexadecimal formats.

Instructions (<code> Data Type)

TotalView uses the <code> data type to display the contents of a location as machine instructions. Thus, 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.

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 in the same way as it displays local and global variables. In the Stack Frame or Source Pane, dive into 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);
```

Because TotalView doesn't know that p actually points to an array of integers, here is how you would 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:

1. Select the type string for argv.

   CLI: dprint argv

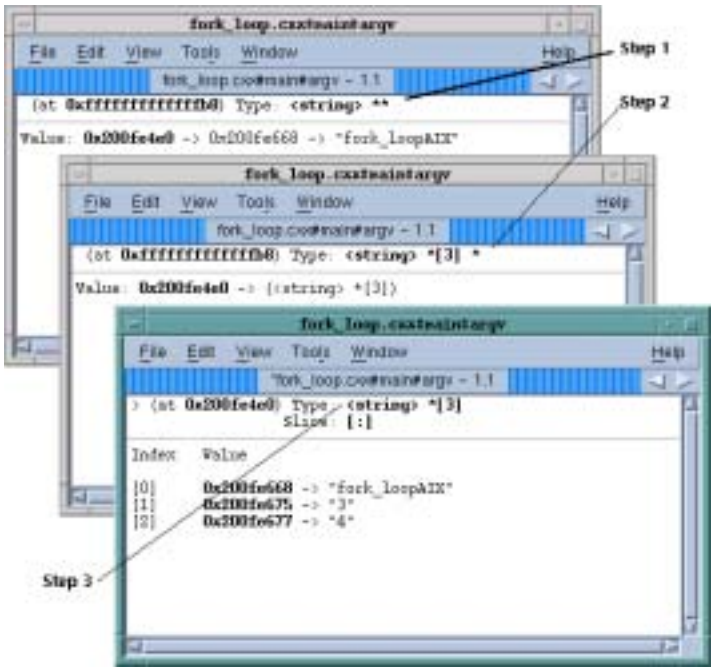
2. Edit the type string using the field editor commands. Change it to:

   <string>*[3]*

   CLI: dprint (<string>*\[3\]*)argv

3. To display the array, dive into the value field for argv. (See Figure 165 on page 249.)

Figure 165: Editing argv



Working with Opaque Data

An opaque type is a data type that isn't fully specified, is hidden, or whose declaration is deferred. For example, the following C declaration defines the data type for p as a pointer to struct foo, which is not yet defined:

```
struct foo;
struct foo *p;
```

When TotalView encounters this kind of information, it may indicate that foo's data type is <opaque>. For example:

```
struct foo <opaque>
```

Changing the Address of Variables

You can edit the address of a variable in a Variable Window. When you edit the address, the Variable Window shows the contents of the new location.

You can also enter an address expression, such as 0x10b8 – 0x80.

Changing Types to Display Machine Instructions

Here is how you can display machine instructions in a Variable Window:

1. Select the type string at the top of the Variable Window.
2. Change the type string to be an array of <code> data types, where *n* indicates the number of instructions to be displayed. For example:

   <code>[*n*]

   TotalView displays the contents of the current variable, register, or area of memory as machine-level instructions.

The Variable Window (shown in Figure 158 on page 237) lists the following information about each machine instruction:

Address	The machine address of the instruction.
Value	The hexadecimal value stored in the location.
Disassembly	The instruction and operands stored in the location.
Offset+Label	The symbolic address of the location as a hexadecimal offset from a routine name.

You can also edit the value listed in the Value field for each machine instruction.

Displaying C++ Types

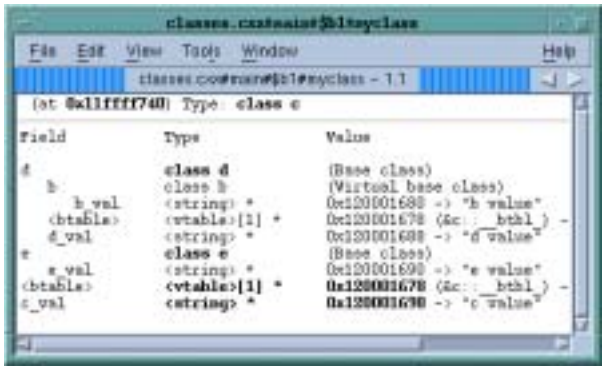
Classes

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 output accessibility information. In these cases, TotalView can not display this information.

For example, Figure 166 displays an object of a class c.

Figure 166: Displaying C++ Classes That Use Inheritance



The definition is as follows:

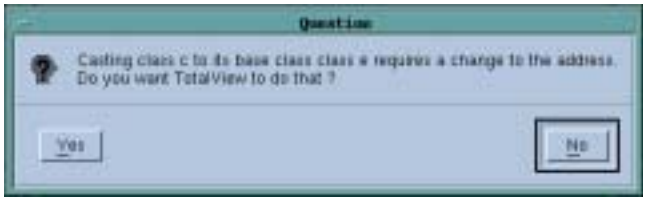
```
class b {
   char * b_val;
public:
   b() {b_val = “b value“;}
};

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“;}
};
```

Changing Class Types in C++

TotalView tries to display the correct data when you change the type of a Variable Window as it and you move up or down the derivation hierarchy.

If a change in the data type also requires a change in the address of the data being displayed, TotalView asks you about changing the address. For example, if you edit a Variable Window’s Type field from class c to class e, TotalView displays the following dialog box.

Figure 167: C++ Type Cast to Base Class Question Window



Selecting Yes tells TotalView to change the address to ensure that it displays the correct base class member. Selecting No tells TotalView to display the memory area as if it were an instance of the type to which it is being cast, leaving the address unchanged.

Similarly, if you change a data type in the Variable Window because you want to cast a base class to a derived class, and that change requires an address change, TotalView asks that you confirm the address change. For example, Figure 168 on page 252 shows the dialog posted if you cast from class e to class c.

Figure 168: C++ Type Cast to Derived Class Question Window

Displaying Fortran Types

TotalView allows you to display FORTRAN 77 and Fortran 90 data types.

Topics in this section are:

Displaying Fortran Common Blocks

For each common block defined within 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, TotalView displays the entire common block in a Variable Window, as shown in Figure 169 on page 253.

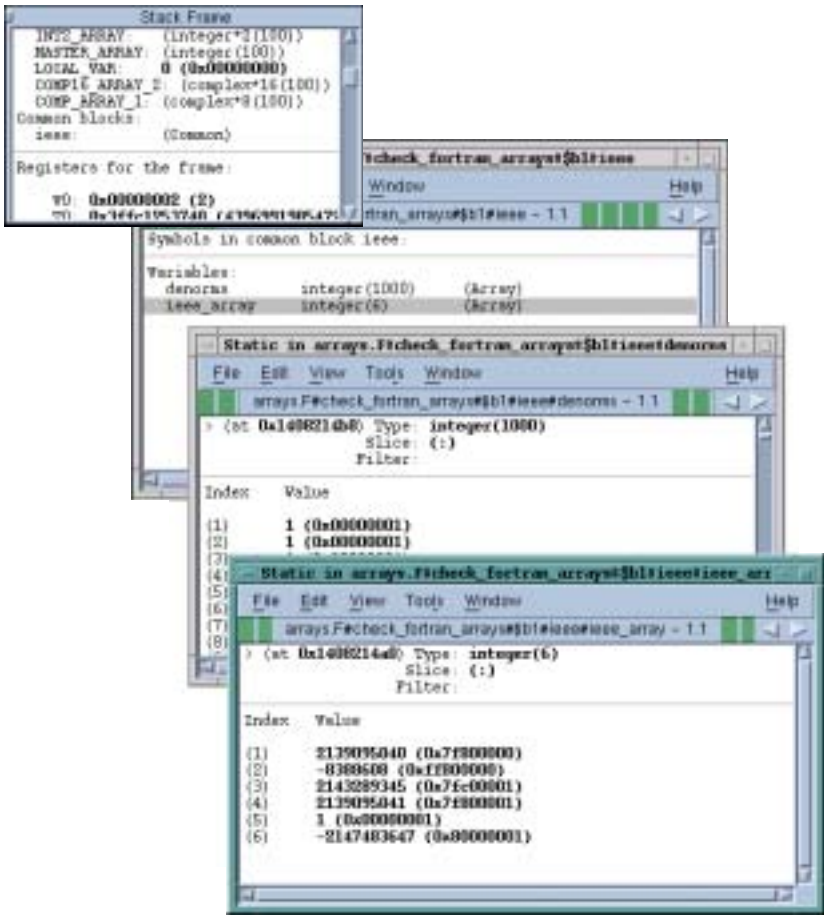
The top-left pane shows a common block list in a Stack Frame Pane. The bottom right window shows the results of diving on the common block to see its elements.

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 provide a single display that contains all of it. For functions and subroutines

Figure 169: Diving into a Common Block List in the Stack Frame Pane



defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

CLI: dprint *variable-name*

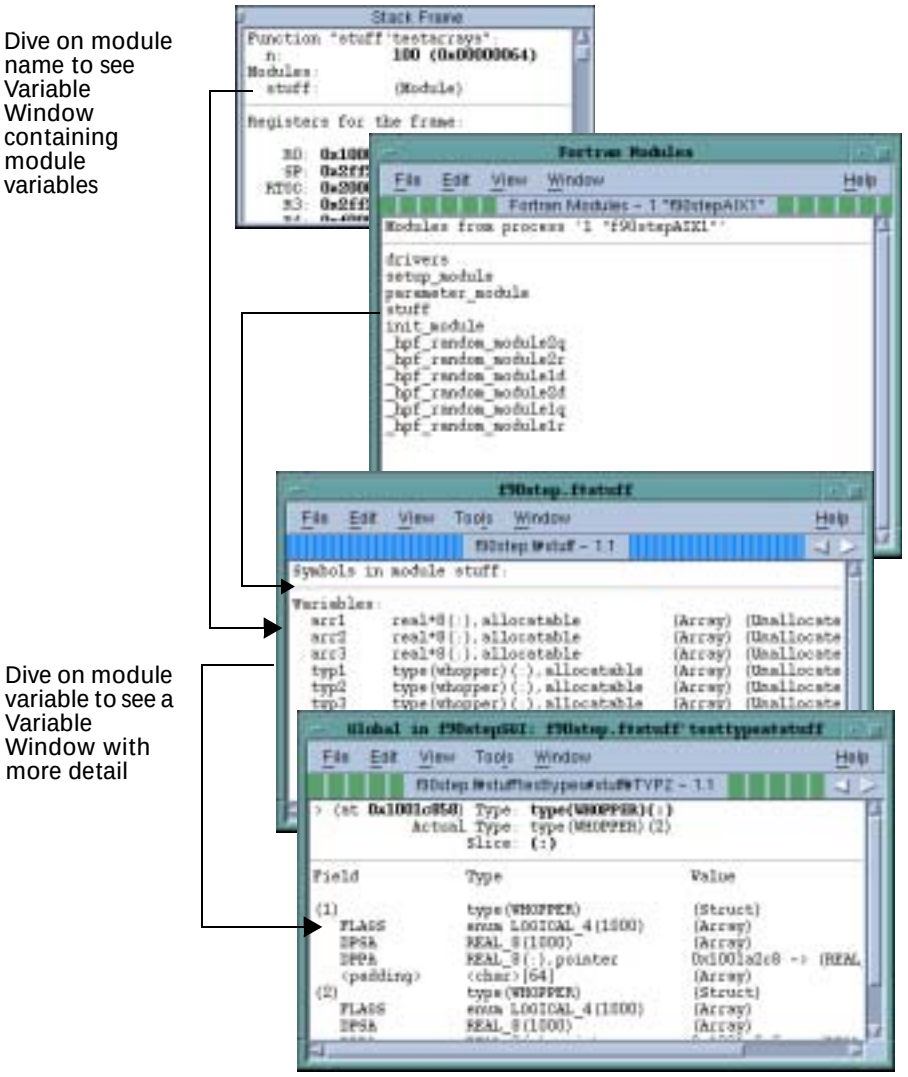
TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView may restrict what's displayed.

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:

- 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.

Alternatively, you can view a list of all the known modules by using the Tools > Fortran Modules command. Like in any Variable Window, you can dive through an entry to display the actual module data, as shown in Figure 170 on page 254.

Figure 170: 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 173.

Debugging Fortran 90 Modules

Fortran 90 and Fortran 95 let you place functions, subroutines, and variables inside modules. These modules can then be included elsewhere using a USE command. When doing this, the names in the module become available in the *using* compilation unit, unless you 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 will 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 introduced the idea of a contained function that is only visible in the scope of its parent and siblings. There can be many contained functions 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*

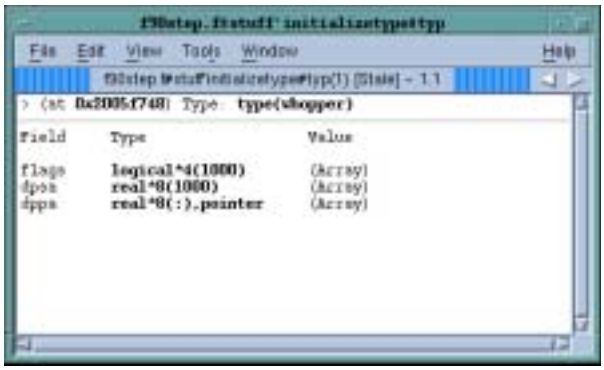
Fortran 90 User-Defined Types

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, here is a code fragment that 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 code as shown in Figure 171.

Figure 171: Fortran 90 User-Defined Type



Fortran 90 Deferred Shape Array Types

Fortran 90 allows you to define deferred shape arrays and pointers. The actual bounds of the 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: (:,:)
```

Fortran 90 Pointer Types

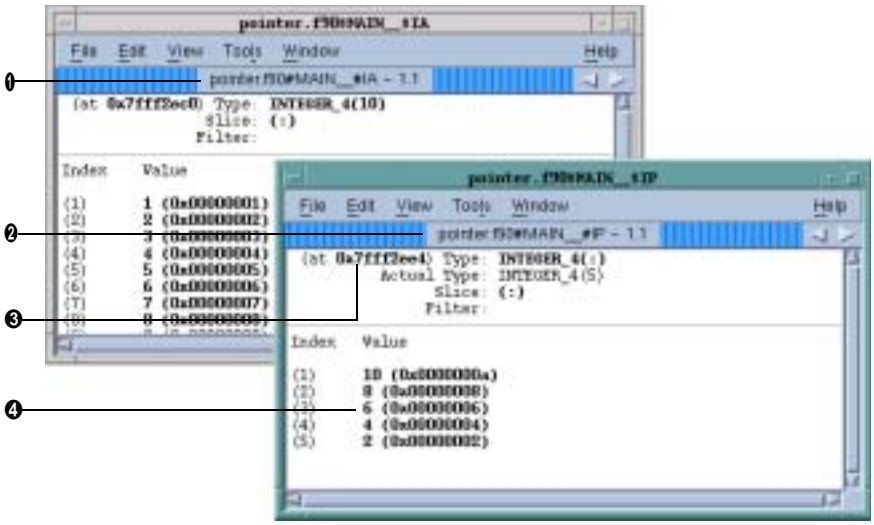
A Fortran 90 pointer type allows you to point to scalar or array types.

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 you would see 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 window shown in Figure 172.

Figure 172: Fortran 90 Pointer Value



❶ Target array ia
❷ Pointer ip into array ia
❸ Address of ip(1)
❹ Values reflect slice

Notice that 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 having the lowest index. This may 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. Most compilers do not generate information that TotalView can use to determine what a PARAMETER's value is. This means that you must make a few changes to your program if you want to see this kind 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 will contain your parameter definitions. You would then use these parameters to initialize variables in a module. After you compile and link this module into your program, the value of these *parameter variables* are visible.

If you're using Fortran 77, you could achieve the same results if you make the assignments in a common block and then include the block in main(). You could 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 will either contain two tabs (HP Alpha) or four tabs (IBM). Figure 173 on page 258 shows some AIX examples.

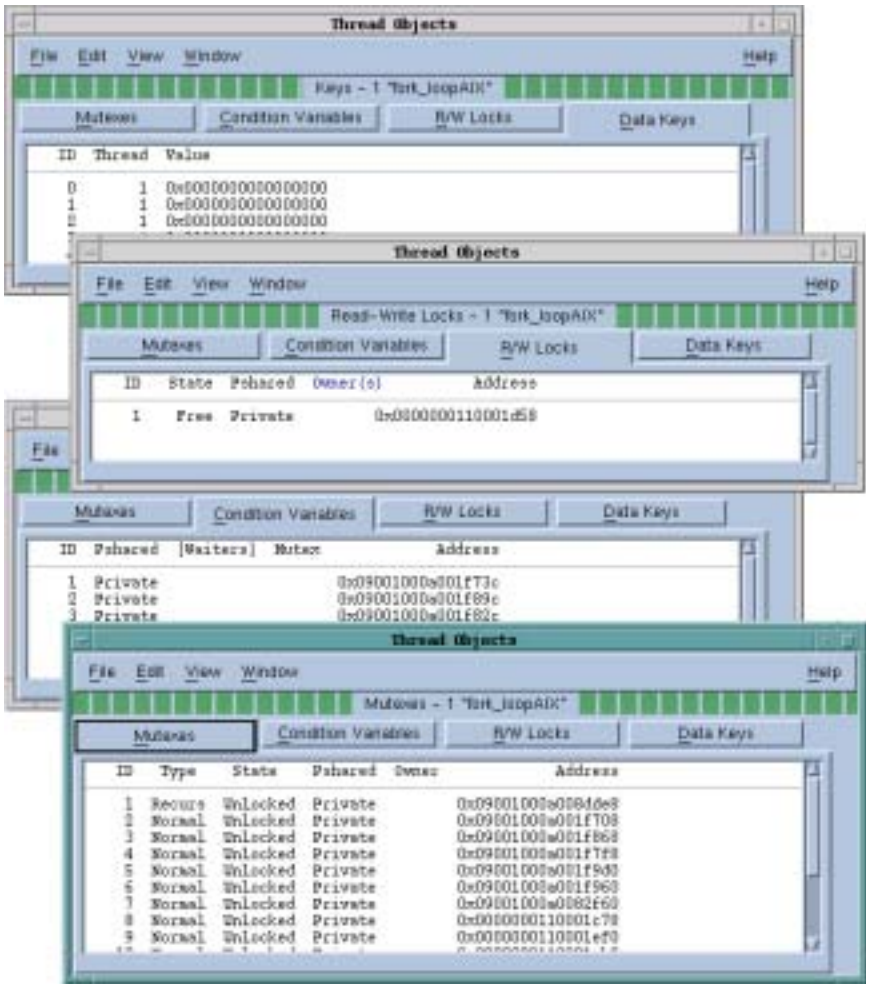
Diving on a any line in these windows displays a Variable Window containing additional information about the item.

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-NULL value to the key.
- 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 New commands to display a Process Window for that thread ID.

The online Help contains considerable information on the contents of these windows.

Figure 173: Thread Objects Page on an IBM AIX machine



Examining Arrays 13

This chapter explains how to examine and change data as you debug your program. You will learn about the following:

Examining and Analyzing Arrays

TotalView can quickly display very large arrays in Variable Windows. An array can be the elements that you've defined in your program or it can be an area of memory that you've cast into an array.

If an array overlaps nonexistent memory, the initial portion of the array is correctly formatted. 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

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, here 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.

Here is a deferred shape array definition for a two-dimensional array variable:

```
integer, dimension (:,:) :: another_array
```

Its TotalView slice definition is (:,:).

As you can see, 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 an_array\[*n*:*m*,*p*:*q*\]
dprint an_array(*n*:*m*,*p*:*q*)

Using Slices and Strides

A slice definition has the following form:

lower_bound:upper_bound:stride

(The *stride* tells TotalView that it should skip over elements and not display them. Adding a *stride* to a slice tells TotalView 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 10-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 upper bound is greater than the lower bound, TotalView lets you view a dimension with reversed indexing. That is, TotalView treats the slice as if it were:

[*ub* : *lb* : *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]
```

In contrast, Fortran 90 requires that you explicitly enter the upper and lower bounds when you're reversing the order in which it displays array elements.

Because the default value for the stride is 1, you can omit the stride (and the colon that precedes it) if your stride value is 1. 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:

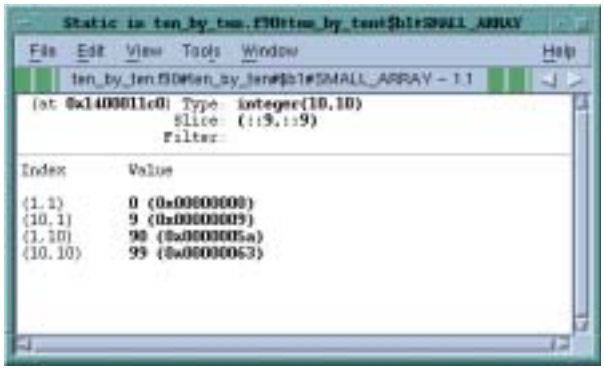
```
[9:9:1]
[9]
```

The lower_bound, upper_bound, and stride can only be constants. They cannot beexpressions.

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: 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: 1, 3, 5, and so on.

Example 2 Figure 174 displays a slice of (::9,::9). This definition displays the four corners of a 10-element by 10-element Fortran array.

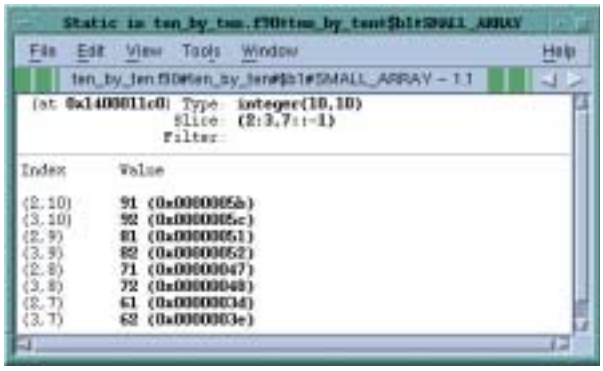
Figure 174: Slice Displaying the Four Corners of an Array



Example 3 You can use a stride to invert the order *and* skip elements. For example, here is a slice that begins with the upper bound of the array and display every other element until it reaches the lower bound of the array: (::–2). Thus, 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. Figure 175 shows how to specify a (2:3,7::–1) slice with an integer*4(–1:5,2:10) Fortran array.

Figure 175: Fortran Array with Inverse Order and Limited Extent



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's dialog box, TotalView initializes the slice field in the Variable Window to this slice descriptions.

If you add subscripts to an array name in the View > Lookup Variable command's dialog box, TotalView interprets these subscripts as a slice description rather than as a request to display an individual value of the array. As a result, you can display different values of the array by changing the slice expression.

For example, suppose that you have a 10-element by 10-element Fortran array named small_array, and you want to display element (5,5). Using the View > Lookup Variable command, type small_array(5,5). This sets the initial slice to (5:5,5:5). This is the top-left screen in Figure 176 on page 263.

CLI: dprint small_array(5,5)

You can tell TotalView to display one of the array's values by enclosing the array name and subscripts (that is, the information normally included in a slice expression) within parentheses, such as (small_array(5,5)).

CLI: dprint (small_array(5,5))

In this case, the Variable Window just displays the type and value of the element and doesn't show its array index. This is shown in the center screen in Figure 176.

Perhaps the most interesting of the screens in Figure 176 on page 263 is the one in the bottom-right corner. This was created by doing a View > Lookup Variable with a value of small_array(i,j). Here, TotalView evaluated the values of i and j before it displayed the window. If you do this, you should know that the values of i and j are just computed once. This means that if the values of i and j change, the displayed value will not change.

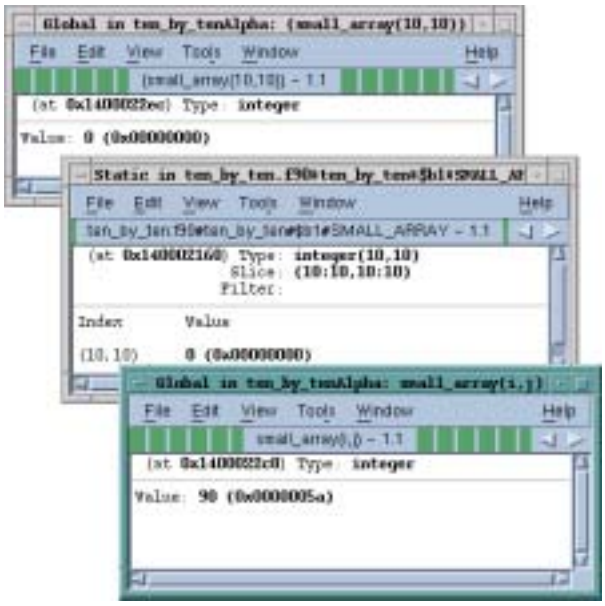
Array Data Filtering

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.

Figure 176: Variable Window for small_array



Topics in this section are:

Filtering Array Data

The procedure for filtering an array is quite simple: select the Filter field, enter the array filter expression, and then press Return.

TotalView updates the Variable Window to exclude only the 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 it 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 may result in a loss of precision. TotalView converts unsigned integer values to non-negative double-precision values.

- If the filter operand or the array element is an unsigned integer, TotalView converts the values 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.

These conversions modify a copy of the array's elements—they never alter the actual array elements.

To stop filtering an array, delete the contents of the Filter field in the Variable Window and press Return. TotalView will then update the Variable Window so that it includes all elements.

Filtering by Comparison

The simplest filters are ones whose formats are:

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, here's the filter for displaying all values greater than 100:

```
> 100
```

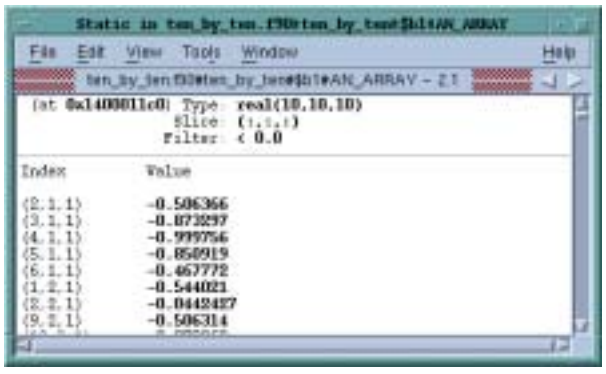
Table 14 lists the comparison operators.

Table 14: Array Data Filtering Comparison Operators

Comparison	C/C++ Operator	Fortran Operator
Equal	==	.eq.
Not equal	!=	.ne.
Less than	<	.lt.
Less than or equal	<=	.le.
Greater than	>	.gt.
Greater than or equal	>=	.ge.

Figure 177 shows an array whose filter is "< 0". This indicates that TotalView should only display array elements whose value is less than 0 (zero).

Figure 177: 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 a 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 explained in the following table:

Table 15: Array Data Filtering IEEE Tag Values

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 178 on page 266 shows an example of filtering an array for IEEE values. The bottom left Variable Window shows how TotalView displays the unfiltered array. Notice the NANQ, and NANS, INF, and –INF values. Then other two windows show filtered displays. The top left window only shows infinite values. The center window only shows the values of denormalized numbers.

Filtering By a Range of Values

Specify ranges as follows:

[>] *low-value* : [<] *high-value*

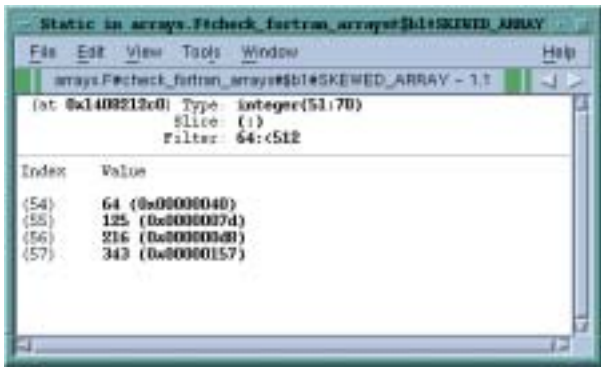
where *low-value* specifies the lowest value to include, and *high-value* specifies the highest value to include, separated by a colon. The high and low values are inclusive unless you use < and > symbols. If you specify a > before *low-value*, the low value is exclusive. Similarly, a < before *high-value* makes it exclusive.

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 a u or U to the value to force an unsigned comparison. Figure 179 on page 266 shows a filter that

Figure 178: Array Data Filtering for IEEE Values

tells TotalView that it should only display values equal to or greater than 64 but less than 512.

Figure 179: Array Data Filtering by Range of Values



Creating Array Filter Expressions

The filtering capabilities described in the previous sections are those that you will most often use. 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, here is an expression that 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)
```

$value is a special TotalView variable that represents the current array element. You can now use this value when creating expressions.

Notice also the use of the and and or operators within the expression. The way in which TotalView computes the results of an expression is identical to the way it computes values at an evaluation point. For more information, see "*Defining Evaluation Points and Conditional Breakpoints*" on page 286.

You cannot use any of the IEEE tag values described in "Filtering for IEEE Values" on page 265 in these kinds of expressions.

Using Filter Comparisons

TotalView lets you filter array information in a variety of ways. This means that you can do the same thing in more than one way. For example, the following two filters display the same array items:

```
> 100
$value > 100
```

Similarly, the following expression displays 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. In general, you'd only 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, of course, sort the actual data.)

If you select the Variable Window's View > Sort > Ascending command, TotalView places all of the array's elements in ascending order. (See Figure 180 on page 268 for an example.)

As you would expect, View > Sort > Descending places array elements into descending order. The View > Sort > None command returns the array to its original order.

The sort commands only manipulate the displayed elements. This means that if you limit the number of elements by defining a slice or a filter, TotalView only sorts the result of the filtering and slicing operations.

Obtaining Array Statistics

The Tools > Statistics command displays a window containing information about your array. Figure 181 on page 268 shows an example.

Figure 180: Sorted Variable Window

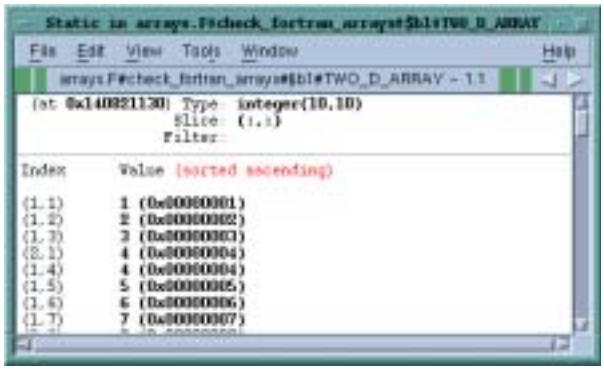
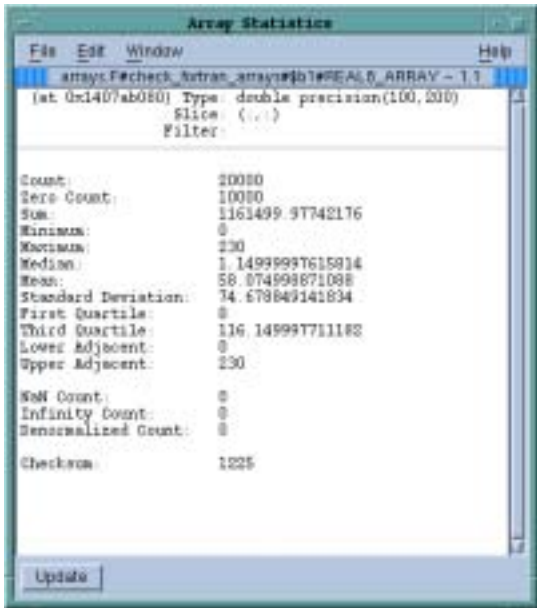


Figure 181: Array Statistics Window



If you have added a filter or a slice, these statistics only describe 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 zero, the median value will be positive even though the array's real median value is less than zero.

The statistics TotalView displays are as follows:

- **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 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 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 fourth 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 of 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 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.

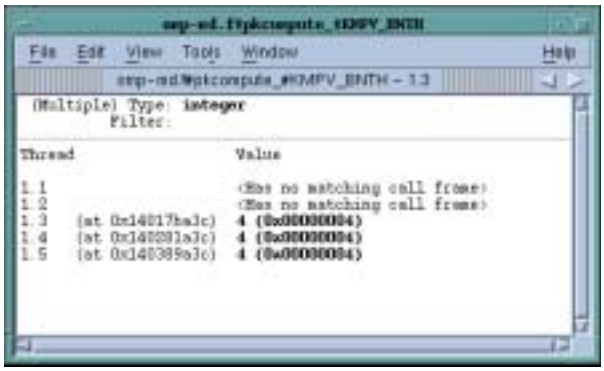
Before displaying a variable's value in all threads or processes, you must display an instance of the variable in a Variable Window. After TotalView displays this window, use one of the following commands:

- View > Laminate > Process, which displays the value of the variable in all of the processes.
- View > Laminate > Thread, which displays the value of a variable in all threads within a single process.

You cannot simultaneously laminate across processes and threads in the same Variable Window.

After using one of these commands, the Variable Window switches to "laminated" mode, and displays the value of the variable in each process or thread. Figure 182 shows a simple, scalar variable in each of the processes in an OpenMP program. Notice that the first six have a variable in a matching call frame. The corresponding variable can't be found for the seventh thread.

Figure 182: Laminated Scalar Variable



If you decide that you no longer want the pane to be laminated, select the View > Laminate > None command to delaminate it.

When looking for a matching call 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 recurse contains two additional calls to recurse. Each of these generate nonmatching call frames.

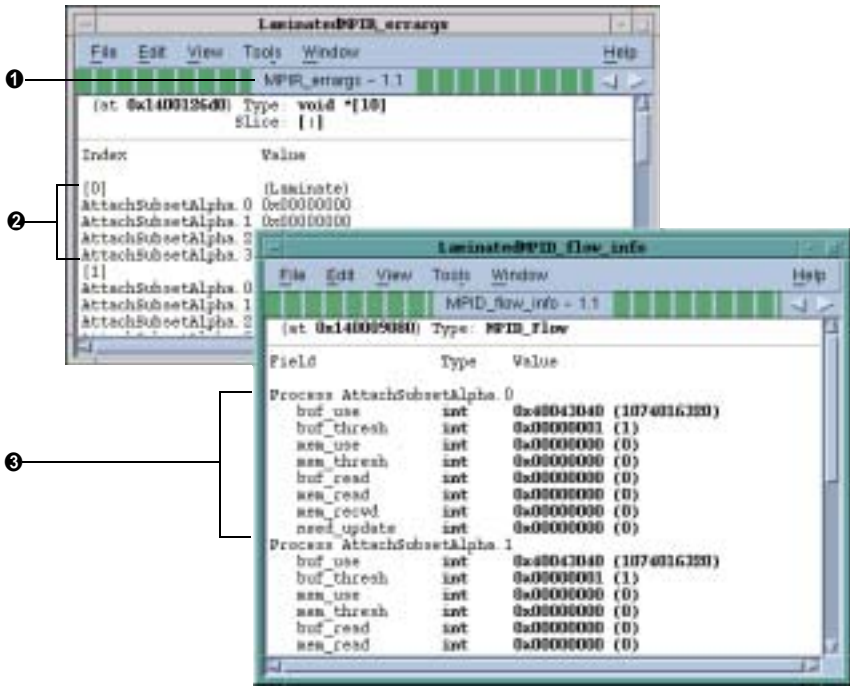
```
void recurse(int i) {
   if (i <= 0)
      return;
```

```
    if (i & 1)
       recurse(i – 1);
    else
       recurse(i – 1);
}
```

If the variables are at different addresses in the different processes or threads, the address field at the top of the pane displays (Multiple) and the unique addresses are displayed with each data item, as was shown in Figure 182 on page 270.

TotalView also allows you to laminate arrays and structures. When you laminate an array, TotalView displays each element in the array across all processors. You can use a slice to select elements to be displayed in laminated windows. Figure 183 shows an example of a laminated array and a laminated structure. You can also laminate an array of structures.

Figure 183: Laminated Array and Structure



1. Laminated array
2. Element [0] for each of the processes
3. Structure elements for one process

Diving in a Laminated Pane

You can dive through pointers in a laminated Variable Window, and the dive will apply to the associated pointer in each process or thread.

Editing a Laminated Variable

If you edit a value in a laminated Variable Window, TotalView asks if it should apply this change to all of the 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 TotalView Visualizer lets you create graphic images of array data. This visualization 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 TotalView Visualizer, see Chapter 7, "*Visualizing Programs and Data*," on page 129.

Visualizing a Laminated Variable Window

You can export data from a laminated Variable Window to the Visualizer by using the Tools > Visualize command. When visualizing laminated 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.

Setting Action Points 14

This chapter explains how to use action points. TotalView supports four kinds of action points: breakpoints, barrier points, evaluation points, and watchpoints. 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 *evaluation 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.

Topics in this chapter are:

Action Points Overview

Actions points allow you to specify an action that TotalView will perform when a thread or process reaches a source line or machine instruction in your program. Here are the different kinds of action points that you can use:

- **Breakpoints**

  When a thread encounters a breakpoint, it stops at the breakpoint. Other threads in the process will also stop. You can also indicate that you want other related processes to stop.

  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. They hold 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.

- **Evaluation points**

  An evaluation point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an evaluation point, it executes this code. You can use evaluation 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 that it should either stop the thread so that you can interact with your program (unconditional watchpoint) or evaluate an expression (conditional watchpoint).

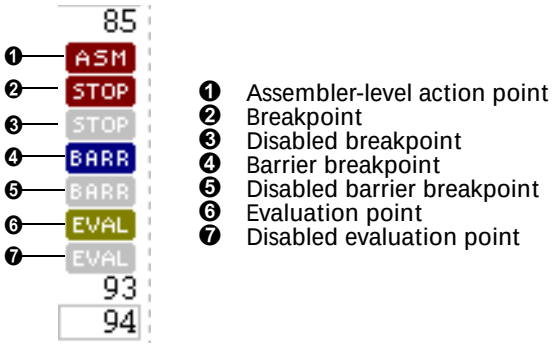
All action points share some common properties.

- You can independently enable or disable them. 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, so in a multithreaded 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 Pane of the Process Window, and the Action Point > Properties Dialog Box.

Each type of action point has a unique symbol. Figure 184 shows some of them.

CLI: dactions shows information about action points.

Figure 184: Action Point Symbols



The ASM icon lets you know that there are one or more assembler-level action points associated with the source line.

CLI: All action points display as "@" when you use the dlist command to display your source code. Use the dactions command to see what kind of action point is set.

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 multiprocess programs
- Assembler-level breakpoints

You can also control whether or not TotalView stops all processes in the control group when a single member reaches a breakpoint.

Topics in this section are:

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 is displaying a prompt.

Choosing Source Lines

If you're using C++ templates, TotalView will set a breakpoint in all instantiations of that template if Plant in share group is selected. 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 the top portion Figure 185 on page 276.)

Similarly, in a multiprocess program, you may not want to set the breakpoint in all processes. If this is the case, select the Process button. (See the bottom portion on Figure 185 on page 276.)

Setting and Deleting 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. All you need do is enter a line number or function name in the Action Point > At Location Dialog Box. (See Figure 186 on page 276.)

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 exe-

Figure 185: Setting Breakpoints on Multiple Similar Addresses and on Processes

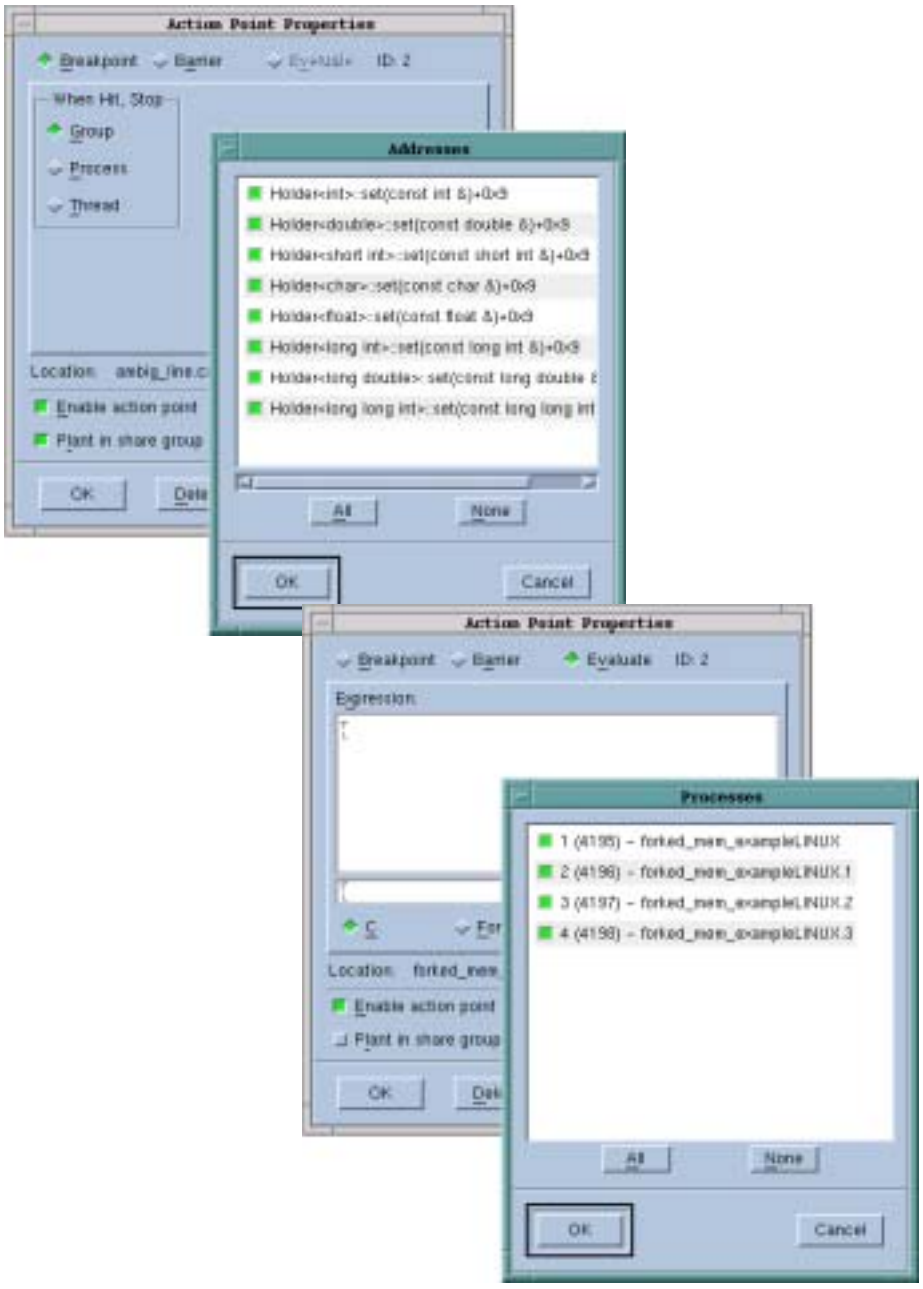
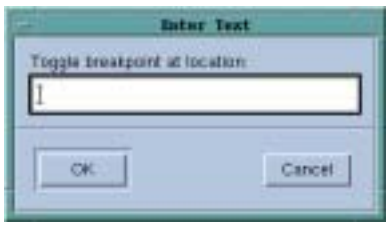


Figure 186: Action Point > At Location Dialog Box

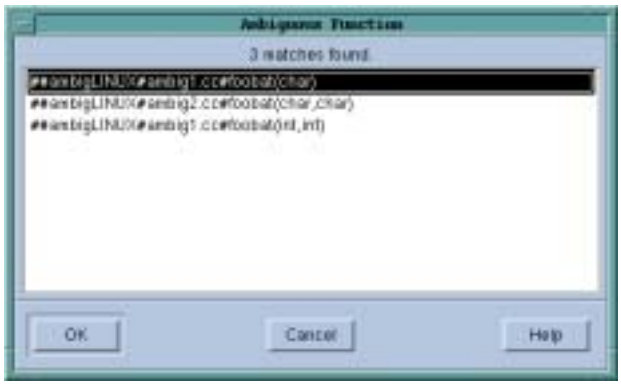


cutable line. In either case, if a breakpoint already exists at a location, TotalView deletes it.

CLI: dbreak sets a breakpoint.
ddelete deletes a breakpoint.

If you enter an ambiguous function name using the Action Point > At Location command, TotalView displays its Ambiguous Function Dialog Box. See Figure 187.

Figure 187: Ambiguous Function Dialog Box



The procedure for resolving ambiguous function names is similar to the procedure described in "*Choosing Source Lines*" on page 275.

Displaying and Controlling Action Points

The Action Point > Properties Dialog Box lets you set and control an action point. (See Figure 188.) Controls in this dialog box also allows you to set an action point's type to breakpoint, barrier point, or evaluation point. You can also define what will happen to other threads and processes when execution reaches this action point.

Figure 188: 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

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 Properties Dialog Box.
- Selecting the STOP or BARR symbol in the Action Points Pane.
- Using the context (right-click) menu.
- Clicking on a disable command in the menubar.

Deleting

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

Enabling

You can activate an action point that was previously disabled by selecting a dimmed STOP, BARR, or EVAL symbol in the Source or Action Points Pane, or by selecting Enable action point in the Properties Dialog Box.

CLI: denable

Suppressing

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. 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 reusing the Action Point > Suppress All command.

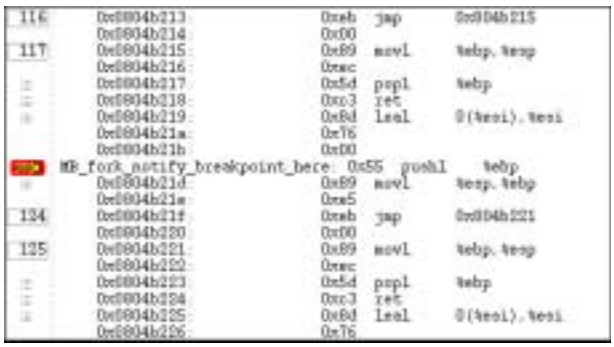
CLI: denable –a

Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code. (Refer to "*Viewing the Assembler Version of Your Code*" on page 175 for information.) You can now select an instruction's line number. The line number must be replaced with a dotted box (⋮⋮)—this indicates the line is the beginning of a machine instruction. Since instruction sets on some platforms support variable-length instructions, you might see more than one line 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 one.

Figure 189: Breakpoint at Assembler Instruction

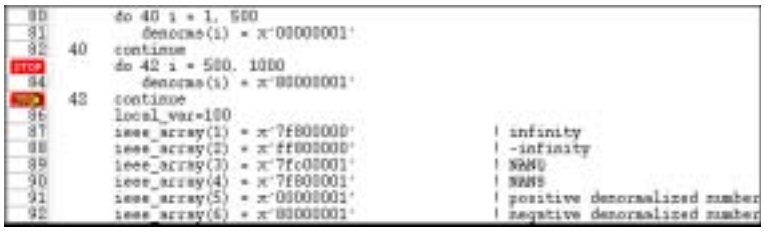


If you set machine-level breakpoints on one or more instructions generated from a single source line and then display source code in the Source Pane, TotalView displays an ASM icon (see Figure 184 on page 274) on the line number. To see the actual breakpoint, you must redisplay assembler instructions.

When a process reaches a breakpoint, TotalView:

- Suspends the process.
- Displays the PC arrow icon ( ) over the stop sign to indicate that the PC is at the breakpoint. (See Figure 190 on page 280.)
- Displays At Breakpoint in the Process Window title bar and other windows.
- Updates the Stack Trace and Stack Frame Panes and all Variable Windows.

Figure 190: PC Arrow Over a Stop Icon



Setting Breakpoints for Multiple Processes

In all programs including multiprocess 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 191.)

Figure 191: Action Point > Properties Dialog Box

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

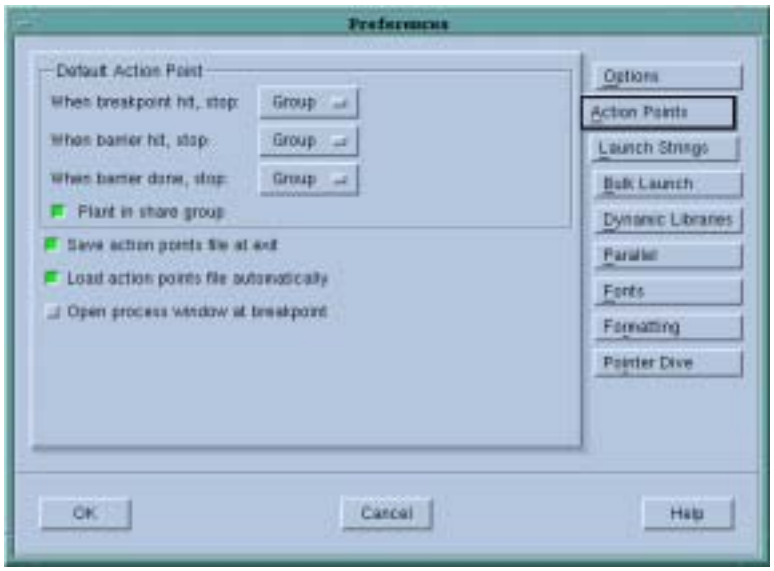
  If this is selected, TotalView enables the breakpoint in all members of this thread's share group at the same time. If this isn't selected, you must individually enable and disable breakpoints in each member of the share group.

  CLI: dset SHARE_ACTION_POINT

The Process button lets you indicate which process in a multiprocess program will have enabled breakpoints. Note that if Plant in share group is selected, this button won't be enabled because you've told TotalView to set the breakpoint in all of the processes.

You can preset many of the properties in this dialog box by using TotalView preferences, as shown in Figure 192.

Figure 192: File > Preferences: Action Points Page



You can find additional information about this dialog box within 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 will be attached to control and share group members. Refer to "*Writing Code Fragments*" on page 301 for more information.

If you're trying to synchronize your program's threads, you will want to set a barrier point. For more information, see "*Barrier Points*" on page 283.

Setting Breakpoints When Using fork()/ execve()

You must link with the dbfork library before debugging programs that call fork() and execve(). See "*Compiling Programs*" on page 35.

Processes That Call fork()

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 containing the share group. This control can, of course, 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

Processes That Call execve()

Shared breakpoints are not set in children having different executables. To set the breakpoints for children that call execve():

1 Set the breakpoints and breakpoint options in the parent and the children that do not call execve().

2 Start the multiprocess program by displaying the Group > Go command. When the first child calls execve(), 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 execve(). This means that if you do not want to share breakpoints in children using the same executable, dive into the breakpoints and set the breakpoint options.

5 Select the Group > Go command.

Example: Multiprocess Breakpoint

The following program excerpt illustrates the places where you can set breakpoints in a multiprocess 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();
```

Here's 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 fork() failed.
5	Stops the child process.
7	Stops the parent process.

Barrier Points

A barrier breakpoint is similar to a simple breakpoint, differing 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.

CLI: dbarrier

Topics in this section are:

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 *go* and *step* commands from the Group, Process, and Thread menus will not 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 contained 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 179 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. (See Figure 193.) As an alternative, you can right-click on the line. From the displayed context menu, you can select the Set Barrier command.

Figure 193: Action Point > Properties Dialog Box

Barrier points are most often used 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 now continue the process, those threads will also run.

If a process is stopped and then continued, the held threads, including the ones waiting at an unsatisfied barrier, do not run. Only unheld threads run.

The When Hit, Stop radio buttons indicate what other threads TotalView will stop when execution reaches the breakpoint, as follows:

Scope	TotalView will:
Group	Stop all threads in the current thread's control group.
Process	Stop all threads in the current thread's process.
Thread	Only stop 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	TotalView will:
Group	Stop all threads in the current thread's control group.
Process	Stop all threads in the current thread's process.
Thread	Only stop this thread.

CLI: dbarrier –stop_when_done

Creating a Satisfaction Set

For even more control over what TotalView will stop, 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 will 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 will hold 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 two ways:

- Using the Action Point > Properties Dialog Box.
- Clicking on the BARR icon in the line number area.

CLI: ddelete

Changes When Setting 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 clicking on Enable action point at the bottom of the dialog box.

CLI: ddisable

Defining Evaluation Points and Conditional Breakpoints

TotalView lets you define *evaluation points*. These are action points at which you have added a code fragment that TotalView will execute. 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. While any user can enable or disable TotalView's ability to compile evaluation points, they must be enabled if you are entering assembler code.

If you are running on a properly configured AIX system, 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.*

Topics in this section are:

By using evaluation points, you can:

- Include instructions that stop a process and its relatives. If the code fragment can make a decision whether it should 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 evaluation point at any source line that generates executable code (marked with a boxed line number surrounding a line number) or

a line containing assembler-level instructions. This means that if you can set a breakpoint, you can set an evaluation point.

At each evaluation point, TotalView or your program executes the code contained in the evaluation point before your program executes the code on that line. While your program can then go on to execute this source line or instruction, it can:

- Include a branching instruction (such as goto in C or Fortran). The instruction can transfer control to a different point in the target program, enabling you to test program patches.
- Execute a TotalView function. TotalView's functions let you stop execution, create barriers, and count down breakpoints. For more information on these statements, refer to "*Built-In Statements*" on page 302.

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.

For complete information on what you can include in code fragments, refer to "*Writing Code Fragments*" on page 301.

Evaluation 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 evaluation points, however, TotalView reapplies them whenever you start a debugging session for that program. To save your evaluation points, refer to "*Saving Action Points to a File*" on page 298.

You should stop a process before setting an evaluation point in it. This ensures that the evaluation point is set in a stable context.

Setting Evaluation Points

To set an evaluation point:

1. Display the Action Point > Properties Dialog Box. You can do this, for example, by right-clicking on 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.
3. Select the button (if it isn't already selected) for the language in which you will code the fragment.
4. Type the code fragment. For information on supported C, Fortran, and assembler language constructs, refer to "*Writing Code Fragments*" on page 301.
5. For multiprocess programs, decide whether to share the evaluation point among all processes in the program's share group. By default, TotalView selects the Plant in share group check box for multiprocess programs, but you can override this by clearing it.

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.

CLI: dbreak –e
dbarrier –e

Creating Conditional Breakpoint Examples

Here are some examples:

- To define a breakpoint that is reached whenever the counter variable is greater than 20 but less than 25:

```
if (counter > 20 && counter < 25)
$stop;
```

- To define a breakpoint that will stop execution every tenth time that TotalView executes the $count function

```
$count 10
```

- To define a breakpoint with a more complex expression, consider:

```
$count my_var * 2
```

When the my_var variable equals 4, the process stops the eight time it executes the $count function. After the process stops, TotalView reevaluates the expression. If my_var now equals 5, the process will stop again after the process executes the $count function ten more times.

For complete descriptions of the $stop and $count statements, refer to "*Built-In Statements*" on page 302.

Patching Programs

You can use expressions in evaluation 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:

- Branch around code that you don't want your program to execute.
- Add new pieces of code.

In many cases, correcting an error means that you will do both operations: you patch out incorrect lines and patch in corrections.

Conditionally Patching Out 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 calling 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 evaluation 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 TotalView-generated line numbers.

Patching in a Function Call

Instead of routing around the problem, you could patch in a printf() statement that displays the value of the global_error variable created in the preceding program. You would set an evaluation point on line 4 and enter:

```
printf ("global_error is %d\n", global_error);
```

TotalView executes this code fragment before the code on line 4; that is, it is executed before global_error is set to 0.

Correcting Code

The next 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  }
```

In his example, you would correct this error by adding the following code to an evaluation 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 evaluation point.

Interpreted vs. Compiled Expressions

On some platforms, TotalView executes interprets expressions. TotalView can also execute compiled expressions on the HP Alpha Tru64 UNIX, IBM AIX, and SGI IRIX platforms. On HP Alpha Tru64 UNIX and IBM AIX platforms, compiled expressions are enabled by default.

You can use the TV::compile_expressions CLI variable to enable or disable compiled expressions. See "*Operating Systems*" in the *TotalView Reference Guide* to find out how TotalView handles expressions on specific platforms.

Using any of the following functions forces TotalView to interpret the evaluation point instead of compiling it: $clid, $duid, $nid, $processduid, $systid, $tid, *and* $visualize. *In addition,* $pid *forces interpretation on AIX.*

Interpreted Expressions

Interpreted expressions are interpreted by TotalView. As is always the case, interpreted expressions run slower (and possibly much slower) than compiled expressions. With multiprocess programs, interpreted expressions run even more slowly because processes may need to wait for TotalView to execute the expression.

When you're debugging remote programs, interpreted expressions always run slower because the TotalView process on the host, not the TotalView debugger server (tvdsvr) on the client, interprets the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView debugger process on the host machine to evaluate one interpreted expression. In contrast, if TotalView compiles the expression, it evaluates them on each remote process.

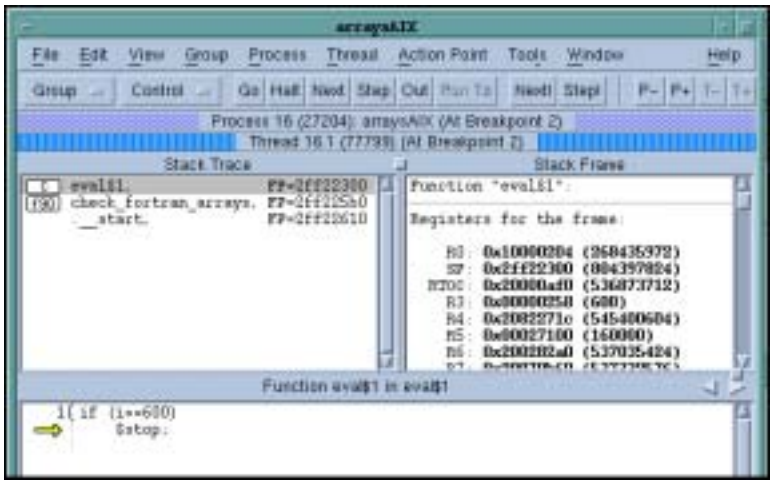
Whenever a thread hits an interpreted patch point, TotalView stops execution. This means that TotalView will create a new set of lockstep groups. Consequently, if goal threads contain interpreted patches, the results are unpredictable.

Compiled Expressions

TotalView compiles, links, and patches expressions into the target process. Because the target thread executes this code, evaluation points and conditional breakpoints execute very quickly. (Note that conditional watchpoints are always interpreted.) And, more importantly, 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. See Figure 194.

Figure 194: Stopped Execution of Compiled Expressions



If you will be using many compiled expressions or your expressions are long, you may need to think about allocating patch space. For more information, see *"Allocating Patch Space for Compiled Expressions*" on page 291.

Allocating Patch Space for Compiled Expressions

TotalView must either allocate or find space in your program to hold the code 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 may 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.

Dynamic Patch Space Allocation

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 following addresses:

Table 16: Dynamic Patch Space Allocation Default Addresses

Platform	Address range
HP Alpha Tru64 UNIX	0xFFFFF00000 – 0xFFFFFFFFFF
IBM AIX	0xCFF00000 – 0xCFFFFFFF
SGI IRIX (–n32)	0x4FF00000 – 0x4FFFFFFF
SGI IRIX (–64)	0x8FF00000 – 0x8FFFFFFF

You can only allocate dynamic patch space for these machines.

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:

- 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.

Static Patch Space Allocation

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 this 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. Table 17 shows sample assembler code for three platforms that support compiled patch points.

Table 17: Static Patch Space Assembler Code

Platform	Assembler Code

HP Alpha Tru64 UNIX	`.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:`

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:`

Here is how you would 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 –64 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 kind of action point called a *watchpoint*. Watchpoints are most often used to find a statement in your program that is writing to places where 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:

- "*Triggering Watchpoints*" on page 295
- "*Using Conditional Watchpoints*" on page 296

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 0 into this location, TotalView doesn't trigger the watchpoint even though your program wrote data into the memory location. See "*Triggering Watchpoints*" on page 295 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 will evaluate the expression when the watchpoint triggers. You can use conditional watchpoints for a number of purposes. For example, you can use one to test if a value changes its sign—that is, it becomes positive or negative—or if a value moves above or below some threshold value.

Architectures

The number of watchpoints, 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 Hewlett Packard (HP) machines running either Alpha Linux or HP-UX.

The following list describes constraints that exist on each platform:

HP Alpha Tru64
: Tru64 places no limitations on the number of watchpoints that you can create, and there are 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 containing watchpoints.

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.

IRIX6 MIPS
: Watchpoints are implemented on IRIX 6.2 and later operating systems. These systems allow you to create about 100 watchpoints. There are no alignment or size constraints. However, watchpoints can't overlap.

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 a 2-byte watchpoint must be aligned on a 2-byte boundary, and so on.

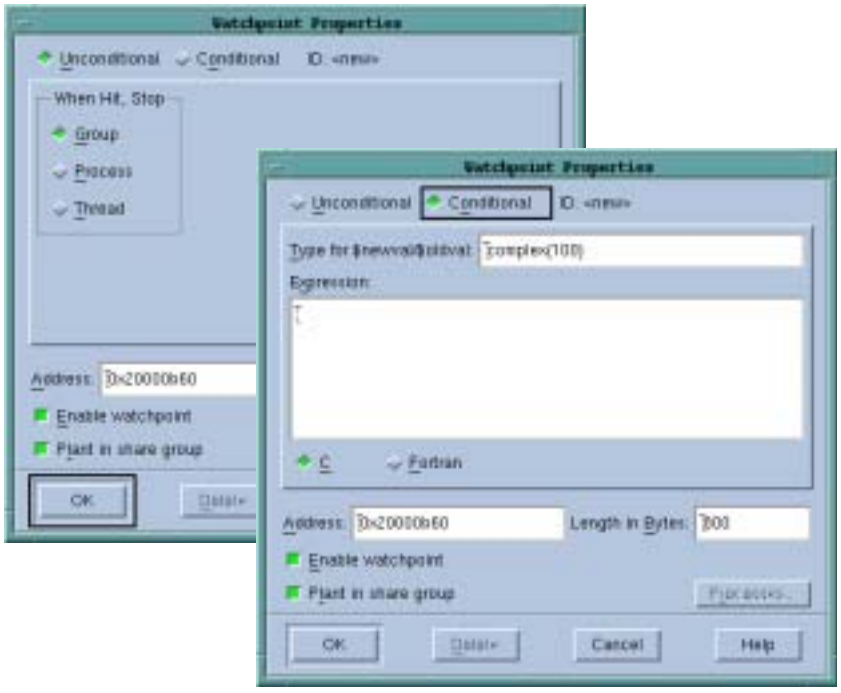
Solaris SPARC — TotalView supports watchpoints on Solaris 2.6 or later operating systems. These operating system allow you to create hundreds of watchpoints, and there are no alignment or size constraints. However, watchpoints can't overlap.

Typically, a debugging session doesn't use many watchpoints. In most cases, you are only monitoring one memory location at a time. So, restrictions on the number of values you can watch are seldom an issue.

Creating Watchpoints

Watchpoints are created by using the Tools > Watchpoint Dialog Box, which is only invocable from a Variable Window (If your platform doesn't support watchpoints, this menu item is dimmed.) See Figure 195.

Figure 195: Tools > Watchpoint Dialog Boxes



Controls within this dialog box let you create unconditional and conditional watchpoints. You will be setting a watchpoint for all of the information displayed in the Variable Window, not just for an element of it. If, for example, the Variable Window is displaying an array, you will be setting a watchpoint for the entire array (or as much of it as TotalView can watch.) If you only want to watch one element, dive on the element and then set the watchpoint. Similarly, if the Variable Window is displaying a structure and you only want to watch one element, dive on the element before setting the watchpoint.

The online Help contains 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 dive into a watchpoint, TotalView displays the Watchpoint Properties Dialog Box.

If you select a watchpoint, TotalView will toggle 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, assume 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. So, TotalView will not be able to monitor changes to the variable because it is at a different memory location.

All of this means that in most circumstances, you can'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.

In some circumstances, a subroutine will always be called from the same location. This means that its local variables will probably be in the same location, so trying can't hurt.

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.

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 will be pointing 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 may be modifying locations monitored by other watchpoints, only the watchpoint for the lowest memory location is triggered. This can occur when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these locations.

For example, assume that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also assume 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 and will not trigger at this time.

Here's a second example. Assume 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 assume 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.

Data Copies

TotalView keeps an internal copy of data in the watched memory locations for each process sharing the watchpoint. If you create watchpoints that cover a large area of memory or if your program has a large number or processes, you will increase TotalView's virtual memory requirements. Furthermore, TotalView refetches data for each memory location whenever it continues the process or thread. This can affect TotalView's performance.

Using Conditional Watchpoints

If you associate an expression with a watchpoint (by selecting the CDWP icon in the Tools > Watchpoint Dialog Box and entering an expression), TotalView will evaluate the expression after the watchpoint triggers. The programming statements that you can use are identical to those used when creating an evaluation 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.

Because memory locations are not scoped, the variable used in your expression must be globally accessible.

Fortran does not have global variables. Consequently, you can't directly refer to your program's variables.

TotalView has two function variables that are specifically designed to be used 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.

Here 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 will store 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.)

Here is a condition that triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And here's 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 "*Writing Code Fragments*" on page 301.

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 within 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 would 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.)

TotalView always interprets conditional watchpoints; it never compiles them. And, 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 into a file. TotalView will then use 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.

Watchpoints are not saved.

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.

CLI: dactions –save *filename*

If you're using a preference to automatically save breakpoints, TotalView will automatically save 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.

CLI: dactions –load *filename*

Automatic saving and loading is controlled by preferences (see File > Preferences in the online Help for more information).

CLI: dset TV::auto_save_breakpoints

Evaluating Expressions

TotalView lets you open a window for evaluating expressions in the context of a particular process and evaluate expressions 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, but here are two examples:

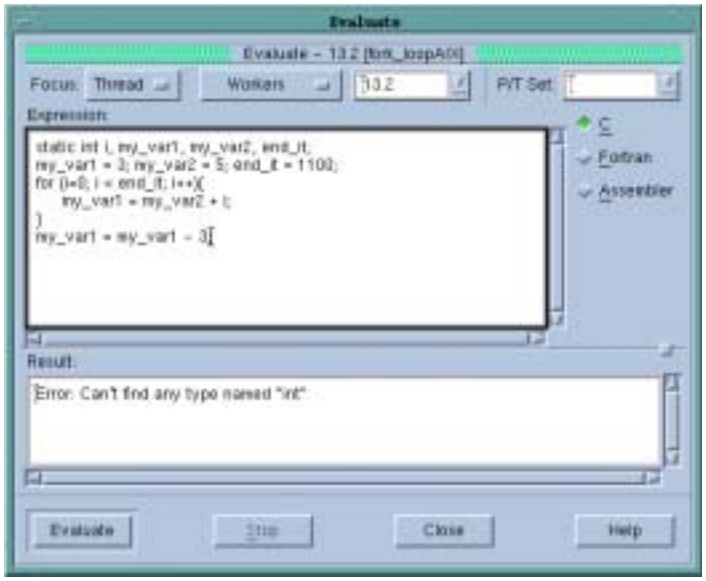
- Expressions can contain loops, so you could use a for loop to search an array of structures for an element set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression field.
- Because you can call subroutines, you can test and debug a single routine in your program without building a test program to call it.

To evaluate an expression:

1. Tell TotalView to 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.
2. Select a button for the programming language you're writing the expression in (if it isn't already selected).
3. Move to the Expression field and enter a code fragment. For a description of the supported language constructs, see "*Writing Code Fragments*" on page 301.

   TotalView returns the value of the last statement in the code fragment. This means that you don't have to assign the expression's return value to a variable. Figure 196 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 would be the same if you had just typed my_var1-3.

Figure 196: Tools > Evaluate Dialog Box



4. Select 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 may also display a message box that tells you that it is waiting for the command to complete. (See Figure 197.)

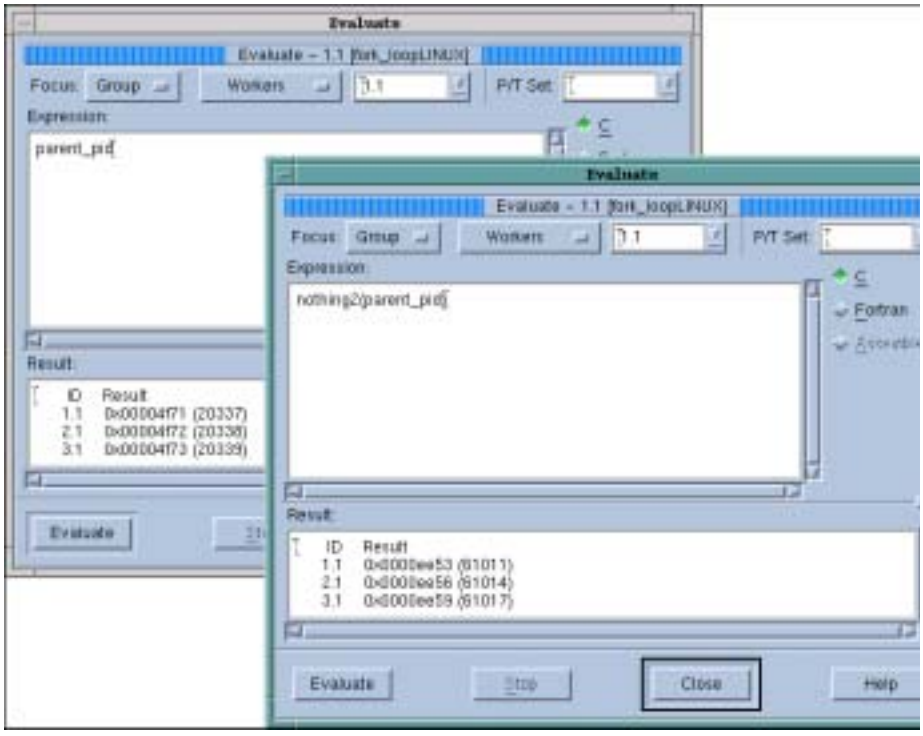
Figure 197: Waiting to Complete Message Box

If you select Cancel, TotalView stops execution.

Since TotalView evaluates code fragments in the context of the target process, it evaluates stack variables according to the currently selected stack frame. 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 could evaluate a function in more than one process. Figure 198 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.

Figure 198: Evaluating Information in Multiple Processes



See Chapter 11, "*Using Groups, Processes, and Threads*," on page 197 for information on using the P/T set controls at the top of this window.

Writing Code Fragments

You can use code fragments in evaluation points and in the Tools > Evaluate Dialog Box. This section describes the function variables, built-in statements, and language constructs supported by TotalView.

While the CLI does not have an "evaluate" command, the information in the following sections does apply to the expression argument of the dbreak, dbarrier, *and* dwatch *commands.*

Topics in this section are:

TotalView Variables

The TotalView expression system supports built-in variables that allow you to access special thread and process values. All variables are 32-bit integers, which is an int or a long on most platforms. Table 18 lists TotalView's built-in variables and their meanings.

Table 18: TotalView 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 of the process. (Interpreted expressions only.)
$systid	The system-assigned thread ID. When this is referenced from a process, TotalView throws an error.
$tid	The TotalView-assigned thread ID. When this is referenced from a process, TotalView throws an error.

TotalView's built-in variables allow you to 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 follows:

```
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 by 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 evaluation 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.

Built-In Statements

TotalView provides a set of built-in statements that you can use when writing code fragments. The statements are available in all languages, and are shown in the following table.

Statement	Use
$count *expression* $countprocess *expression*	Sets a process-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the process stops. The other processes in the control group continue to execute.
$countall *expression*	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 *expression*.
$countthread *expression*	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the thread stops. Other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs identically to $countprocess. A thread evaluates *expression* 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 expression. TotalView will not reevaluate until the expression actually stops the thread. This means that TotalView ignores changes in the value of *expression* 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.
$hold $holdprocess	Holds the current process. If all other processes in the group are already held at this Eval point, then TotalView will release all of them. If other processes in the group are running, they continue to run.

Statement	Use
$holdstopall $holdprocessstopall	Exactly like $hold, except any processes in the group which are running are *stopped.* Note that the other processes in the group are not automatically held by this call—they are just stopped.
$holdthread	Freezes the current thread, leaving other threads running.
$holdthreadstop $holdthreadstopprocess	Exactly like $holdthread except it *stops* the process. The other processes in the group are left running.
$holdthreadstopall	Exactly like $holdthreadstop except it stops the entire group.
$stop $stopprocess	Sets a process-level breakpoint. The process that executes this statement stops; other processes in the control group continue to execute.
$stopall	Sets a program-group-level breakpoint. All processes in the control group stop when any thread or process in the group executes this statement.
$stopthread	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 identically to $stopprocess.
$visualize(*expression*[,*slice*])	Visualizes the data specified by *expression* and modified by the optional *slice* value. *Expression* and *slice* must be expressed using the code fragment's language. The expression must return a dataset (after modification by *slice*) that can be visualized. *slice* is a quoted string containing a slice expression. For more information on using $visualize in an expression, see "*Visualizing Data Programmatically*" on page 135.

C Constructs Supported

When writing code fragments in C, keep these guidelines in mind:

- 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.

Data Types and Declarations

The following list describes the C data types and declarations that you can use:

- The data types that you can use are char, short, int, float, double, and pointers to any primitive type or any named type in the target program.
- Only simple declarations are permitted. Do not use struct, union, and array declarations.
- You can refer to variables of any type in the target program.
- Unmodified variable declarations are considered local. References to these declarations override references to similarly named global variables and other variables in the target program.

- (Compiled evaluation points only.) The global declaration makes a variable available to other evaluation points and expression windows in the target process.
- (Compiled evaluation points only.) The extern declaration references a global variable that was or will be defined elsewhere. If the global variable is not yet defined, TotalView displays a warning.
- Static variables are local and persist even after TotalView evaluates an evaluation point.
- TotalView only evaluates expressions that initialize static and global variables the first time it evaluates a code fragment. In contrast, it initializes local variables each time it evaluates a code fragment.

Statements

The following list describes the C language statements that you can use.

- The statements that you can use are assignment, break, continue, if/else structures, for, goto, and while.
- 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, here is a goto statement that branches to source line number 432 of the target program:

```
goto 432;
```

- Although function calls are permitted, you can't pass structures.
- Type casting is permitted.

All operators are permitted, with these limitations:

- TotalView doesn't support the ?: conditional operator.
- While you can use the sizeof operator, you can't use it for data types.
- The (*type*) operator can't cast data to fixed-dimension arrays by using C cast syntax.

Fortran Constructs Supported

When writing code fragments in Fortran, keep these guidelines in mind:

- Enter only one statement on a line. You can't continue a statement onto more than one line.
- You can use GOTO, GO TO, ENDIF, and END IF statements; while ELSEIF isn't allowed, you can use ELSE IF.
- Syntax is free-form. No column rules apply.
- You can enter comments in three ways: with a C in column 1 or as free-form comments using the /* ... */ delimiters or the // characters. In addition, anything typed after // characters is also ignored. The following example shows all three:

```
C I=I+1
/*
I=I+1
J=J+1
ARRAY1(I,J)= I * J
*/
k = 4 // This is also a comment
```

- 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	DO100I=1,10
CALL RINGBELL	CALL RING BELL
X .EQ. 1	X.EQ.1

Data Types and Declarations

The following is a list of data types and declarations that you can use in a Fortran expression.

- You can use the following data types: INTEGER (assumed to be long), REAL, DOUBLE PRECISION, and COMPLEX.
- You can't use implied data types.
- You can only use simple declarations. You can't use a COMMON, BLOCK DATA, EQUIVALENCE, STRUCTURE, RECORD, UNION, or an array declaration.
- Your can refer to variables of any type in the target program.

Statements

The following list describes the Fortran language statements that you can use.

- You can use the following statements: 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 an alternate return).
- A GOTO statement can refer to a line number in your program. This line number is the number displayed 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 TotalView's source line number rather than a statement label.
- The only expression operators that are not supported are the 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 array syntax.
- You can't use Fortran 90 pointer assignment (the => operator).
- You can't call Fortran 90 functions that require assumed shape array arguments.

Writing Assembler Code

On HP Alpha Tru64 UNIX, RS/6000 IBM AIX, and SGI IRIX operating systems, TotalView lets you use assembler code in evaluation points, conditional breakpoints, and in the Tools > Evaluate Dialog Box. However, if you want to use assembler constructs, you must enable compiled expressions. See "*Interpreted vs. Compiled Expressions*" on page 289 for instructions.

To indicate that an expression in the breakpoint or Evaluate Dialog Box is an assembler expression, click on the Assembler button in the Action Point > Properties Dialog Box, as shown in Figure 199.

Figure 199: Using Assembler

Assembler expressions are written in the TotalView Assembler Language. In this language, instructions are written in the target machine's native 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.

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 these 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. Labels can appear 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:

Table 19: TotalView Assembler Operators

Operators	Definition
+	Plus
–	Minus (also unary)

Operators	Definition
*	Times
#	Remainder
/	Quotient
&	Bitwise AND
^	Bitwise XOR
!	Bitwise OR NOT (also unary –, 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 *expr*
hi32 (*expr*)	High 32 bits of operand *expr*
lo16 (*expr*)	High 16 bits of operand *expr*
lo32 (*expr*)	Low 32 bits of operand *expr*

The TotalView Assembler pseudo-operations are as follows:

Table 20: TotalView Assembler Pseudo-Ops

Pseudo Ops	Definition
$debug [0 | 1]	*Internal debugging option.* With no operand, toggle debugging; 0 = > turn debugging off 1 = > turn debugging on
$hold $holdprocess	Hold the process
$holdstopall $holdprocessstopall	Hold the process and stop the control group
$holdthread	Hold the thread
$holdthreadstop $holdthreadstopprocess	Hold the thread and stop process
$holdthreadstopall	Hold the thread and stop the control group
$long_branch expr	Branch to location *expr* using a single instruction in an architecture-independent way; using registers is not required
$stop $stopprocess	Stop the process
$stopall	Stop the control group
$stopthread	Stop the thread
name=*expr*	Same as def *name*,*expr*
align *expr* [, *expr*]	Align location counter to an operand 1 alignment; use operand 2 (or 0) as the fill value for skipped bytes
ascii *string*	Same as *string*
asciz *string*	Zero-terminated string
bss *name*,*size-expr*[,*expr*]	Define *name* to represent *size-expr* bytes of storage in the bss section with alignment optional *expr*; the default alignment depends on the size:

Pseudo Ops	Definition
	if *size-expr* > = 8 then 8 else if *size-expr* > = 4 then 4 else if *size-expr* > = 2 then 2 else 1
byte *expr* [, *expr*] ...	Place *expr* values into a series of bytes
comm *name*,*expr*	Define *name* to represent *expr* bytes of storage in the bss section; *name* is declared global; alignment is as in bss without an alignment argument
data	Assemble code into data section (data)
def *name*,*expr*	Define a symbol with *expr* as its value
double *expr* [, *expr*] ...	Place *expr* values into a series of doubles
equiv *name*,*name*	Make operand 1 be an abbreviation for operand 2
fill *expr*, *expr*, *expr*	Fill storage with operand 1 objects of size operand 2, filled with value operand 3
float *expr* [, *expr*] ...	Place *expr* values into a series of floating point numbers
global *name*	Declare *name* as global
half *expr* [, *expr*] ...	Place *expr* values into a series of 16-bit words
lcomm *name*,*expr*[,*expr*]	Identical to bss
lsym name,*expr*	Same as def name,*expr* but allows redefinition of a previously defined name
org *expr* [, *expr*]	Set location counter to operand 1 and set operand 2 (or 0) to fill skipped bytes
quad *expr* [, *expr*] ...	Place *expr* values into a series of 64-bit words
string *string*	Place *string* into storage
text	Assemble code into text section (code)
word *expr* [, *expr*] ...	Place *expr* values into a series of 32-bit words
zero *expr*	Fill *expr* bytes with zeros

15 Debugging Memory Problems

TotalView provides two memory tools.

- The first monitors the amount of memory that your program uses by tracking the amount of memory each of your program's processes uses. Because TotalView displays which process is using the most and which process is using the least amount of memory, you can quickly identify processes that are not behaving as you expect them to be. Similarly, comparing memory use over time lets you identify programs that are leaking memory.
- The second monitors the way in which your program allocates and frees memory. If your program misuses that way it frees memory, TotalView will stop your program right *before* this problem occurs so that you can identify which statement will cause the problem.

For more information, see:

- "*Monitoring Memory Use*" on page 309
- "*Tracking Heap Problems*" on page 311

Monitoring Memory Use

The Tools > Memory Usage Window tells you how your program is using memory and where this memory is being used. One way to use this window is to compare memory use over time so that you can tell if your program is leaking memory. If a program is leaking memory, you'll see that the amount of memory being used steadily increases over time. Another way is to compare memory use between processes, which can tell you if a process is using more memory than you expect. Figure 200 on page 310 shows the By Process Pane of the Memory Usage Window.

When TotalView displays this information, it indicates the maximum value of an item is displayed in red and the minimum value in blue. It also indicates the amount of memory used by a process's text and data segments, and the TotalView process IDs.

Figure 200: Tools > Memory Usage Window

Clicking on a column heading sorts the information from maximum to minimum or vice versa. You can also sort information using the controls at the bottom of the window.

Notice that if you add the memory values of all columns but the last, the sum doesn't equal this last column's value. This are several reasons for this. For example, most operating systems divide segments into pages, and information in a segment does not cross page boundaries. Another reason is that a process could map a file or an anonymous region. Areas such as these are part of what you'll see in the VmSize column. However, they are not shown elsewhere.

CLI: dmstat

Here's the definition for most of these columns:

Process	The TotalView process ID.
Text	The amount of memory used for storing your program's machine code instructions.
Data	The amount of memory used for storing uninitialized and initialized data.
Heap	The amount of memory currently being used for data created at runtime.
Stack	The amount of memory used by the currently executing routine and all the routines in its backtrace. If you are looking at a multi-threaded process, TotalView only shows information for the main thread's stack. Note that the stacks of others threads doesn't change over time on some architectures.

On some systems, the space allocated for a thread is considered as being part of the heap.

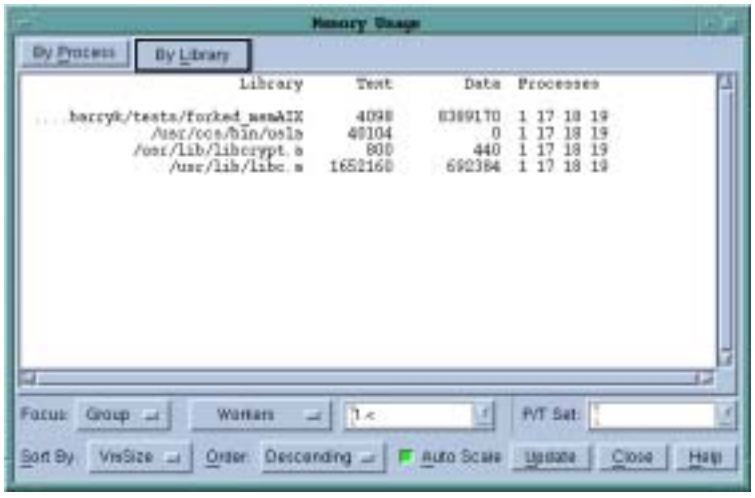
StackVm	The logical size of the stack is the difference between the current value of the stack pointer and the value reported under the Stack column. This value can differ from the size of the virtual memory mapping in which the stack resides.
VmSize	The sum of the sizes of the mappings in the process's address space.

The online Help has more information.

Using the P/T Set controls at the top of the window is discussed in Chapter 11, "*Using the P/T Set Browser*," on page 222.

The By Library Pane (shown in Figure 201) shows which library files are contained within your executable.

Figure 201: Tools > Memory Usage Window: By Library Pane



Tracking Heap Problems

TotalView can detect problems that occur when you allocate and free heap memory. (The *heap* is region of memory that your program uses when it needs to dynamically allocate space.) This memory is usually allocated by malloc(), calloc(), and realloc() and deallocated by free() and realloc(). See your systems man pages for definitions of what each of these do.

While your program can directly use these functions, it may also indirectly use them. For example, the strdup() function found on some operating systems will use malloc() to allocate memory for duplicated strings. Similarly, some libraries and programs place a wrapper around malloc() so that they can track, manipulate, and manage memory use. In these cases and others, TotalView's Memory Tracker is able to watch your program's heap operations. (See "*Limitations*" on page 316 for instances where problems can occur.)

The TotalView Memory Tracker is simple to use. In most cases, all you do is:

- Enable the Memory Tracker from within the GUI or the CLI.
- Either link your program with a TotalView library or set an environment variable.

As you start your enabled program, TotalView begins monitoring heap operations. If an error occurs, TotalView stops execution so that you can track down which statement within your program caused the problem. TotalView stops execution when your program executes the statement that would cause the heap error.

Linking your program with the Memory Tracker is almost always required for multiprocess programs. It is sometimes required for multithreaded programs. So, while there may be times when you don't have to do this, it's always better to link it and know that memory problems are being tracked when you expect them to be tracked. See "Linking and Environment Variables" on page 320 for more information.

Because the Memory Tracker intercepts calls to heap library functions, it can affect your application's performance. However, because the impact is small, you may not notice a difference.

The remaining topics in this section are:

- "*Quick Overview*" on page 312, which shows what occurs with a trivial program.
- "*Behind the Scenes*" on page 315. This section describes what TotalView is doing to your program.
- "*Kinds of Problems*" on page 316, which contains a look at small programs that illustrate the kinds of problems that the Memory Tracker can locate.
- "*Using the dheap Command*" on page 319, which looks at the GUI and CLI commands for using the Memory Tracker.
- "*Linking and Environment Variables*" on page 320. While adding the Memory Tracker to your program can be automatic, there are environments in which you will have to link it in manually. You'll find platform-specific discussions here.

Quick Overview

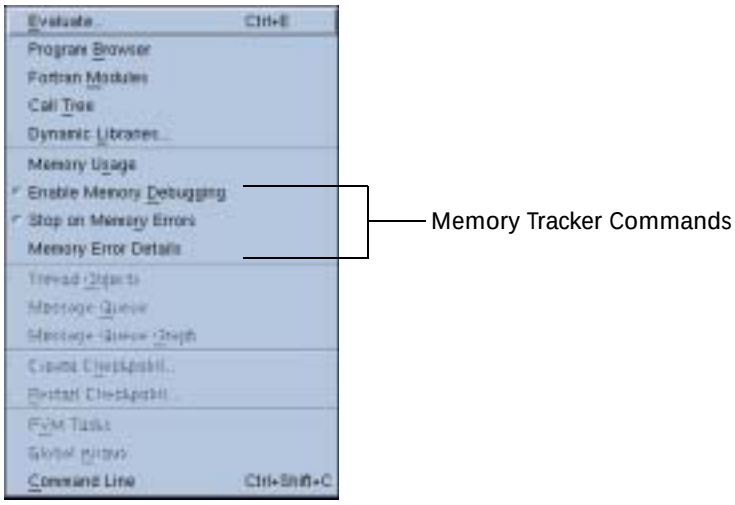
In some cases, you'll need to link your program with the Memory Tracker. Check the requirements in "*Linking and Environment Variables*" on page 320. If you have linked your program, notice that the Tools > Enable Memory Debugging command within the Process Window is selected. Figure 202 on page 313 shows this pulldown menu.

If you do need to link your program with the memory tracker, you'll need to select the Tools > Enable Memory Debugging command.

This section discusses how to use TotalView from within the GUI. CLI users should also read this section."Using the dheap Command" on page 319 discusses how to do the same things in the CLI in addition to describing operations that are unique to the CLI.

Selecting this command tells TotalView that it should stop when it detects a memory error. If you don't want it to stop, uncheck the Stop on Memory

Figure 202: Tools Pulldown



Errors command. If you've linked with the Memory Tracker, you'll need to select this command.

When TotalView detects a heap memory problem, it stops execution and displays its Memory Error Details Window. Figure 203 is an example.

Figure 203: Memory Error Details Window

Notice the following:

- The first line within the window tells you what kind of error occurred.
- The large central area contains a function backtrace if the memory error has an allocated block. You can select a stack frame in this pane to reset the Process Window to where the problem occurred.
- The bottom area contains the address in memory at which the memory block problem occurred and how many bytes were involved.

In some cases, the Memory Tracker will not display a backtrace. For example, if you try to free memory allocated on the stack or in a data section, the Memory Tracker will not display a backtrace. However, TotalView still displays a Memory Error Details Dialog Box. Instead of displaying a backtrace, it displays the following message: "No stack trace available for this memory error."

When execution stops, the Process Window will show one of TotalView's "magic" breakpoints—these are breakpoints that TotalView plants. You will also see a comment above the breakpoint that tells you how to obtain additional information from the TotalView CLI. Figure 204 on page 314 shows an example.

Figure 204: Memory Tracker Magic Breakpoint

```
27  void
28  TV_HEAP_notify_breakpoint_here ( TV_HEAP_event_info_t *event )
29  {
30
31    /* The process has been stopped because a heap event has been detected.  */
32    /* dheap -status will provide more information about what has happened.  */
33
34    TV_HEAP_event = *event;
35  } /* TV_HEAP_notify_breakpoint_here () */
36
```

Enabling, Stopping, and Starting

If your program is executing, you cannot enable the Memory Tracker. If you try, TotalView will display its Restart Now Dialog Box, which is shown in Figure 205.

Figure 205: Restart Now Dialog Box

Select Restart & Enable to tell TotalView to stop your program, enable the Memory Tracker, and then restart your program. If you select Cancel, your program will continue executing. However, TotalView does not enable the Memory Tracker at this time or when you restart your program.

If your program has begun executing and the Memory Tracker is enabled, you cannot disable it. If you try, TotalView displays a Restart Now dialog box that allows you to restart your program with the Memory Tracker disabled.

If all you want to do is stop TotalView from notifying you about heap problems, just reselect the Tools > Stop on Memory Errors command. After selecting this command, TotalView will continue tracking memory events. However, it will no longer stop execution if a problem occurs. In some cases, your operating system will terminate execute at this point. In some instances, however, your memory library may allow execution to continue. For example, some libraries allow execution to continue if you free memory that you've already freed.

The Stop on Memory Errors command has another, more important purpose. Suppose your program executes a section of code that you know has memory problems. For example, you are calling functions in a shared library you aren't interested in or can't debug and this library has heap problems. In this case, you'd uncheck the Stop on Memory Errors command. After setting a breakpoint at the place where you'll want TotalView to look for memory problems, you'd start execution. When TotalView hits this breakpoint, you would select the Tools > Stop on Memory Errors command. Now, if a problem occurs, TotalView will stop and tell you about it.

Behind the Scenes

The TotalView Memory Tracker intercepts calls made by your program to heap library functions that allocate and deallocate memory using the malloc() and free() functions. It also tracks related functions such as calloc() and realloc(). The Memory Tracker uses a technique called interposition in which an agent intercepts calls to functions.

There are two ways in this agent becomes part of your program:

- TotalView can preload it. Preloading means that loader will be told to load a specified object before any of the object's listed in the application's loader table. When a module makes a reference to a symbol defined in another module, the runtime linker searches for the first definition of that symbol. Because the Memory Tracker's agent is the first object loaded, references to routines in the program's malloc API become references to the agent's routines.

  On Linux, HP Tru64 Alpha, Sun, and SGI, TotalView will set an environment variable that contains the pathname of the agent's shared library within your local TotalView installation. For more information, see *"Attaching to Programs"* on page 321.

  The agent uses operations defined in the dynamic linker's API to find the original function definition. So, after the agent intercepts a call, it will call the original function. This lets TotalView use the Memory Tracker with most memory allocators.
- If the agent cannot be preloaded, you must explicitly link it into your program. You'll find complete details in "*Linking and Environment Variables*" on page 320. Also, if your program attaches to an already running program, you will need to explicitly link this other program with the agent.

Because TotalView uses interposition, it can be considered as being non-invasive. That is, it doesn't rewrite any of your program's code and you don't have to do anything within your program. Using interposition means that TotalView isn't changing your program or its logic. This also means that TotalView will not be the source of any bugs nor will it change your program's behavior.

Errors Detected

The following list describes the kinds of error that TotalView will detect:

- **free not allocated**: An application calls free() with an address that does not lie in any block allocated from the heap.
- **realloc not allocated**: An application calls realloc() with an address that does not lie at any block allocated from the heap.

- **Address not at start of block**: free() or realloc() receive a heap address that does not lie on the start of a previously allocated block.
- **Double allocation**: The Memory Tracker detects that an already allocated address is being returned by a new request. This would be returned by routines such as calloc(), malloc(), realloc(). This error indicates that a problem exists either with the heap manager or its data structures.
- **Allocation request returns null**: If a null value is returned by an allocation operation, no memory is available. While this is an expected problem for which a program should always check, not all do.

These operations may be disguised. For example, on many platforms, strdup() calls malloc() to create memory for a string. Since malloc() is being called, TotalView can track strdup's use of memory.

Limitations

Here are some limitations you should know about:

- The malloc API must reside in a shared library. It cannot be statically linked.
- If your application implements its own storage allocator that does not conform to the malloc API, problems will not be caught. If, however, your application provides a wrapper around malloc(), the Memory Tracker can monitor the heap.
- This Memory Tracker relies on preloading, and consequently, may not work with setuid programs if the loader does not allow preloading with this class of application.

Kinds of Problems

This section presents some trivial programs that illustrate the kinds of problems that TotalView detects. The errors shown in these programs are obvious. Errors in your program are, of course, more subtle.

Freeing Unallocated Space

The following section contains programs that free space that they cannot deallocate.

Freeing Stack Memory

In the following example, the memory for the stack_addr variable was created on the stack. Your program cannot deallocate this memory.

```
int main (int argc, char *argv[])
{
   void  *stack_addr  = &stack_addr;

   /* Error: freeing a stack address */
   free(stack_addr);

   return 0;
}
```

Freeing bss Data

Among other things, bss data includes static and global variables that your program does not initialize. Your program cannot free this memory. This example tries to free the bss_var variable.

```
      /* Not initialized; should be in bss */
static int bss_var;

int main (int argc, char *argv[])
{
   void *addr = (void *) (&bss_var);

   /* Error: address in bss section */
   free(addr);

   return 0;
}
```

Freeing Data Section Memory

If your program initializes static and global variables, they are found in your executable's data section. Your program cannot free this memory. This example tries to free the data_var variable.

```
       /* Initialized; should be in data section */
static int data_var = 9;

int main (int argc, char *argv[])
{
   void *addr = (void *) (&data_var);

   /* Error: adress in data section */
   free(addr);

   return 0;
}
```

Freeing Memory That Is Already Freed

The following program allocates some memory, then releases it twice. On some operating systems, your program may SEGV when it tries to free the memory a second time.

```
int main (int argc, char *argv[])
{
   char *prog_name = argv[0];
   void *s;
   int  region_size = 0xab;

   /* Get some memory */
   s = malloc(region_size);

   /* Now release the memory */
   free(s);

   /* Error: Release it again */
   free(s);

   return 0;
}
```

Tracking realloc Problems

The following program passes a misaligned address to realloc().

```
int main (int argc, char *argv[])
{
   char *s;
   char *misaligned_s;
   char *realloced_s;
   int  region_size  = 17;
   int  realloc_size = 256;

   /* Get some memory */
   s = malloc(region_size);

   /* Reallocate the memory using a misaligned address */
   misaligned_s = s + 8;
   realloced_s = realloc(misaligned_s, realloc_size);

   return 0;
}
```

In a similar fashion, TotalView will detect realloc() problems caused by passing addresses that are within parts of memory for which it cannot release old memory. For example, TotalView will detect problems if you try to:

- Reallocate stack memory.
- Reallocate memory within the data section.
- Reallocate memory within the bss section.

Freeing the Wrong Address

TotalView can detect when a program tries to free a block that does not correspond to the start of blocks allocated using malloc(). The following program begins by allocating memory. It then tries to free this memory using a misaligned address.

```
int main (int argc, char *argv[])
{
   char  *s;
   char  *misaligned_s;
   int    region_size  = 17;

   /* Get some memory */
   s = malloc(region_size);

   /*  Release the memory using a misaligned address */
   misaligned_s = s + 8;

   free(misaligned_s);

   free(s);

   return 0;
}
```

Using the dheap Command

The dheap command lets you track memory problems when you are using the CLI. While the dheap command lets you do everything that you can do using the GUI, there are also a few things that are unique to the CLI. The following list is a reprise of what has already been presented, differing in that the emphasis is on the CLI.

- To enable and disable the memory tracker, use the dheap –enable and dheap –disable commands.
- To start and stop error notification, use either the dheap –notify or dheap –nonotify commands. Unlike in the GUI, enabling the Memory Tracker does not also set up notification.
- To see the status of the Memory tracker, use the dheap command. You don't need to do this in the GUI because the buttons on the toolbar give you this information.

The most important difference between what you can do in the GUI and what you can do in the CLI is that you can set a focus using the CLI. This means that you can restrict the Memory Tracker to just the processes that you name. For example:

```
dfocus g dheap -notify
```

This command tells the Memory Tracker that it should only notify you if problems occur in those processes that are in control group within the process of interest. In most cases, you don't want to restrict the Memory Tracker to a particular focus. However, there are instances where this could simplify what you are doing.

The dheap -info command lets you display information about the heap. You can show information for the entire heap or limit what TotalView displays to just a portion. (See dheap in the *TotalView Reference Guide* for more information.)

The following example shows the kind of information the CLI will display after the Memory Tracker locates an error.

```
d1.<> dheap
        process:   Enable   Notify    Available
 1      (21361):      yes      yes          yes
 1.1 realloc: Address is not the start of any allocated block
.:
     realloc:  existing allocated block:
     realloc:  start=0x08049878  length=(17 [0x11])
     realloc:   malloc           PC=0x4001f60d \
                      [/home/../malloc_wrappers_dlopen.c#148]
     realloc:   main             PC=0x0804853a \
                      [realloc_test.c#33]
     realloc:   __libc_start_main PC=0x40054647 [/lib/i686/
libc.so.6]
     realloc:   _start            PC=0x08048421 \
                      [/nfs/.../realloc_test]

     realloc:  address passed to heap manager: 0x08049880
     realloc:  requested new length: 256 [0x100]
```

```
d1.<> dheap -info -backtrace
process     1   (21361):
   0x8049878 --        0x8049889       0x11 [         17]
  :   malloc           PC=0x4001f60d \
                      [/home/../malloc_wrappers_dlopen.c#148]
  :   main             PC=0x0804853a [tx_realloc_test.c#33]
  :   __libc_start_main PC=0x40054647 [/lib/i686/libc.so.6]
  :   _start           PC=0x08048421 [/nfs/.../realloc_test]
```

Linking and Environment Variables

The TotalView Memory Tracker *interposes* the malloc API with TotalView's heap agent. That is, the agent intercepts the calls your program is making, checks them for errors, and sends them on so they can be processed by the original library. The Memory Tracker's agent does not replace standard memory functions; it just monitors what they do.

You can incorporate the agent into your environment by:

- Linking your application with the agent.
- Requesting that the agent's library be preloaded by setting a runtime loader environment variable. This is only done when your program will attach to another program that it did not start and you want the Memory Tracker to locate problems in this second application.

AIX applications differ from applications running on other platforms as AIX does not support interposition. However, its malloc implementation allows it to be replaced. This allows TotalView to replace the AIX malloc with a replacement API.

Topics in this section are:

- "*Linking Your Application With the Agent*" on page 320
- "*Attaching to Programs*" on page 321

Linking Your Application With the Agent

In some situations, you will need to explicitly link the Memory Tracker's agent directly into your program. For example, if you are debugging an MPI program, your starter program may not propagate environment variables.

On AIX, you must always link your program so that malloc can find the malloc replacement and agent. In addition, you only set your LIBPATH environment variable when the tvheap_mr.a library is in your LIBPATH. If it isn't, your program may not load. You must use the –L *options shown in the following table.*

The following table shows additional linker command-line options that you must use when you link your program.

Platform	Compiler	ABI	Additional linker options
HP Tru64 Alpha (version 5)	Compaq/KCC	64	–L*path* –ltvheap –rpath *path*
	GCC	64	–L*path* –ltvheap –Wl,–rpath,*path*
IBM RS/6000 (all)	IBM/GCC	32/64	–L*path_mr* –L*path*
	KCC	32	–L*path_mr* –L*path* --static_libKCC
		64	–L*path_mr* –L*path*
AIX 4	IBM/KCC	32	–L*path_mr* –L*path path*/aix_malloctype.so –binitfini:aix_malloctype_init

Platform	Compiler	ABI	Additional linker options
AIX 5	IBM/GCC/KCC	64	–L*path_mr* –L*path* *path*/aix_malloctype64_4.so \ –binitfini:aix_malloctype_init
	GCC	32	–L*path_mr* –L*path* \ *path*/aix_malloctype.so –Wl, –binitfini:aix_malloctype_init
		64	–L*path_mr* –L*path* \ *path*/aix_malloctype64_4.so –Wl, –binitfini:aix_malloctype_init
	IBM/GCC/KCC	32	–L*path_mr* –L*path* *path*/aix_malloctype.o
		64	–L*path_mr* –L*path* *path*/aix_malloctype64_5.o
Linux x86	GCC/Intel	32	–L*path* –ltvheap –Wl,–rpath,*path*
	KCC	32	–L*path* –ltvheap –rpath *path*
Linux IA64	GCC/Intel	64	–L*path* –ltvheap –Wl,–rpath,*path*
SGI	SGI/GCC/KCC	32	–L*path* –ltvheap –rpath *path*
		64	–L*path* –ltvheap_64 –rpath *path*
Sun	Sun/KCC/ Apogee	32	–L*path* –ltvheap –R *path*
	Sun/KCC	64	–L*path* –ltvheap_64 –R *path*
	GCC	32	–L*path* –ltvheap –Wl,–R,*path*
		64	–L*path* –ltvheap_64 –Wl,–R,*path*

In this table:

path — The absolute path to the agent in the TotalView installation hierarchy. More precisely, this directory is:

installdir/toolworks/totalview.*version*/*platform*/lib

- *installdir* — The installation base directory name
- *version* — The TotalView version number
- *platform* — The platform tag

path_mr — The absolute path the malloc replacement. This value is determined by the person who installs the TotalView malloc replacement library.

As it is easy to misinterpret the path specifications, you may want to see what value TotalView uses when it sets a path. Here's the procedure:

1. Start TotalView.
2. Enable the Memory Tracker.
3. Select the Process > Startup Parameters command and then select the Environments Page. The value shown here should be the same or similar to what you will type. (See Figure 206 on page 322.)

Attaching to Programs

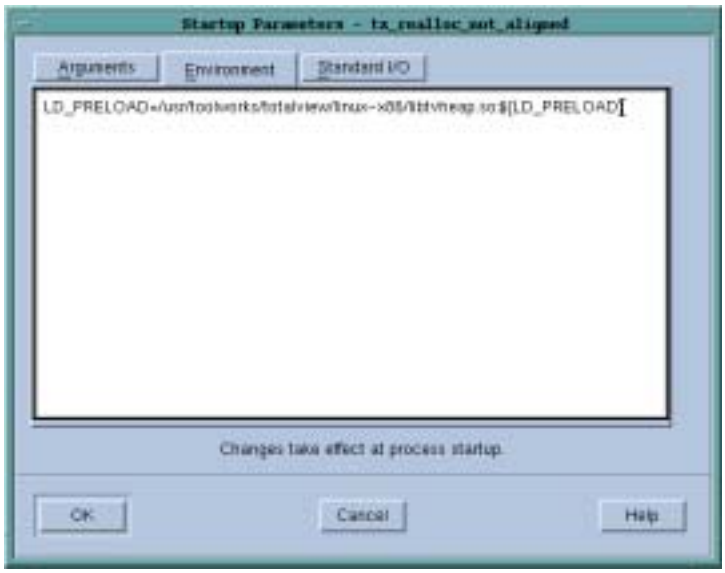
When your program attaches to a process that is already running, the Memory Tracker will not be able to locate heap problems in that process unless you've manually set a Memory Tracker environment variable. The

variable that you will use is unique (or relatively so) on each platform. Here's the name of these variables:

Platform	Variable
HP Tru64 Alpha	_RLD_LIST
IBM AIX	MALLOCTYPE
Linux IA64 and x86	LD_PRELOAD
SGI Irix	__RLDN32_LIST _RLD64_LIST
Sun	LD_PRELOAD

On all platforms, you can see the value that TotalView uses by displaying the Environment Page within the Process > Startup Parameters command. For example:

Figure 206: Process > Startup Parameter: Environment Page



If you need to set this variable:

1. Start TotalView and enable memory debugging.
2. Open this dialog and see what the value is for your environment.
3. Close TotalView.
4. Start the program to which you will be attaching as an argument to the env command. For example, here's how to set this variable on AIX:

```
env MALLOCTYPE user:tvheap_mr.a totalview my_prog
```

Do not set these environment variables so that the agent is available when any program that you invoke executes. For example, you should use env *to set this variable and run TotalView rather than* setenv*. If you set it so that it is exported to all programs in your shell, you may be running the agent against system programs such as* ls*.*

Using the Memory Tracker

This topic describes using the Memory Tracker within various environments. The sections within this topic are:

- MPICH
- IBM PE
- SGI MPI
- RMS MPI

MPICH

Here's how to use the Memory Tracker with MPICH MPI codes. Etnus has tested this only on Linux x86.

1. You must link your parallel application with the Memory Tracker's agent as described "*Linking and Environment Variables*" on page 320. On most Linux x86 systems, you'll type:

   ```
   mpicc -g test.o -o test -Lpath -ltvheap -Wl,-rpath,path
   ```

2. Start TotalView using the -tv command-line option to the mpirun script in the usual way. For example:

   ```
   mpirun -tv mpirun-args test args
   ```

   TotalView will start up on the rank 0 process.

3. Because you will have linked in the Memory Tracker's agent, Tools > Enable Memory Debugging is automatically selected in the Process Window showing the rank 0 process.

4. If you want TotalView to notify you when a heap error occurs in your application (and you probably do), select the Tools > Stop on Memory Errors command.

5. Run the rank 0 process.

IBM PE

Here's how to use the Memory Tracker with IBM PE MPI codes.There are two alternatives. The first is to place the following proc within your .tvdrc file:

```
# Automatically enable memory error notifications
# (without enabling memory debugging) for poe programs.
proc enable_mem {loaded_id} {
   set mem_prog poe
   set executable_name [TV::image get $loaded_id name]
   set file_component [file tail $executable_name]

   if {[string compare $file_component $mem_prog] == 0} {
      puts "Enabling Memory Tracker for $file_component"
      dheap -notify
   }
}

# Append this proc to Totalview's image load callbacks
# so that it runs this macro automatically.
dlappend TV::image_load_callbacks enable_mem
```

Here's the second method:

1. You must prepare your parallel application to use the Memory Tracker's agent in "*Linking and Environment Variables*" on page 320 and in "*Installing tvheap_mr.a on AIX*" on page 325. Here is an example that usually works:

   ```
   mpcc_r -g test.o -o test -Lpath_mr -Lpath \
         path/aix_malloctype.o
   ```

   "*Installing tvheap_mr.a on AIX*" on page 325 contains additional information.
2. Start TotalView on poe as usual:

   ```
   totalview poe -a test args
   ```

Because tvheap_mr.a is not in poe's LIBPATH, enabling the Memory Tracker upon the poe *process will cause problems because* poe *will not be able to locate the* tvheap_mr.a *malloc replacement library.*

3. If you want TotalView to notify you when a heap error occurs in your application (and you probably do), use the CLI to turn on notification, as follows:
   - Open a CLI window by selecting the Tools > Command Line command from the Process Window showing poe.
   - In the CLI window, enter the `dheap -notify` command. This command turns on notification in the poe process. The MPI processes to which TotalView will attach inherit notification.
4. Run the poe process.

SGI MPI

There are two ways to use the Memory Tracker on SGI MPI code. In most cases, all you need do is select the Tools > Memory Debugging command upon the mpirun process. Once in a while, this may cause a problem. If it does, here's what you should do:

1. Link your parallel application with the Memory Tracker's agent as described in the *Debugging Memory Problems* chapter of the *TotalView User's Guide*. Roughly:

   ```
   cc -n32 -g test.o -Lpath -ltvheap -rpath path \
         -lmpi -o test
   ```
2. Start TotalView on mpirun. For example:

   ```
   totalview mpirun -a mpirun-args test args
   ```
3. Select the Tools > Stop on Memory Errors command to turn on notification.
4. Run the mpirun process.

RMS MPI

Here's how to use the Memory Tracker with Quadrics RMS MPI codes. Etnus has tested this only on Linux x86.

1. There is no need to link the application with the Memory Tracker because prun propagates environment variables to the rank processes. However, if you'd like to link the application with the Memory Tracker's agent, you can.
2. Start TotalView on prun. For example:
   ```
   totalview prun -a prun-args test args
   ```
3. Enable memory debugging using the Tools > Enable Memory Debugging command from the Process Window showing prun. If you had linked in the agent, Tools > Enable Memory Debugging is automatically selected.
4. If you want TotalView to notify you when a heap error occurs in your application (and you probably do), select the Tools > Stop on Memory Errors command.
5. Run the prun process.

Installing tvheap_mr.a on AIX

You must install the tvheap_mr.a library on each node upon which you will be running the Memory Tracker's agent. One method is to place a symbolic link in /usr/lib to the tvheap_mr.a library. If you do this, you do not need to add special –L command line options to your build. In addition, In addition, there will not be any special requirements when using poe.

The rest of this section describes who you need to do if you cannot create symbolic links.

Most of what you need to do is encapsulated within the aix_install_-tvheap_mr.sh script. You'll find this script in the following directory:

toolworks/totalview.version/rs6000/lib/

For example, after you become root, enter the following commands:

```
cd toolworks/totalview.6.3.0-0/rs6000/lib
mkdir /usr/local/tvheap_mr
./aix_install_tvheap_mr.sh ./tvheap_mr.tar /usr/local/tvheap_mr
```

Use poe to create tvheap_mr.a on multiple nodes.

The pathname for the tvheap_mr.a library must be the same on each node. That means that you cannot install this library on a shared file system. Instead, you must install it on a file system that is private to the node. For example, because /usr/local is usually only accessible from the node upon which it is installed, you might want to install it there.

The tvheap_mr.a library depends heavily on the exact version of libc.a that is installed on a node. If libc.a changes, you must recreate tvheap_mr.a by re-executing the aix_install_tvheap_mr.sh script.

LIBPATH and Linking

This section discusses compiling and linking your AIX programs. To begin with, the following command adds *path_mr* and *path* to the your program's default LIBPATH:

```
xlc -Lpath_mr -Lpath -o a.out foo.o
```

When malloc() dynamically loads tvheap_mr.a, it should find the library in *path_mr*. When tvheap_mr.a dynamically loads tvheap.a, it should find it in *path*.

Note that the AIX linker allows you to relink executables. This means that you can make an already complete application ready for the Memory Tracker's agent; for example:

```
cc a.out -Lpath_mr -Lpath -o a.out.new
```

Here's an example that does not link in the malloc replacement. Instead, it allows you to dynamically set MALLOCTYPE:

```
xlC -q32 -g \
   -L/usr/local/tvheap_mr \
   -L/home/totalview/interposition/lib prog.o -o prog
```

The next example shows how you allow your program to access the Memory Tracker's agent by linking in the aix_malloctype.o module:

```
xlc -q32 -g \
   -L/usr/local/tvheap_mr \
   -L/home/totalview/interposition/lib prog.o \
     /home/totalview/interposition/lib/aix_malloctype.o \
   -o prog
```

You can check that the paths made it into the executable by running the dump command:

```
% dump -Hv tx_memdebug_hello

   tx_memdebug_hello:

           ***Loader Section***
         Loader Header Information
  VERSION#      #SYMtableENT    #RELOCent      LENidSTR
  0x00000001    0x0000001f      0x00000040     0x000000d3

  #IMPfilID     OFFidSTR        LENstrTBL      OFFstrTBL
  0x00000005    0x00000608      0x00000080     0x000006db

           ***Import File Strings***
  INDEX  PATH               BASE          MEMBER
  0      /home/totalview/interposition/lib:/usr/vacpp/
lib:/usr/lib:/lib
  1                         libc.a            shr.o
  2                         libC.a            shr.o
  3                         libpthreads.a     shr_comm.o
  4                         libpthreads.a     shr_xpg5.o
```

Index 0 within the Import File Strings section shows the search path the runtime loader uses when it dynamically loads a library. Some MPI systems propagate the preload library environment to the processes it will run. Other, do not. If they do not, you will need to manually link them with the tvheap library.

In some circumstances, you may want to link your program instead of setting MALLOCTYPE. If you set the MALLOCTYPE environment variable for

your program and it fork/execs a program that is not linked with the agent, then your program will terminate because it fails to find malloc().

Glossary

ACTION POINT: A debugger feature that allows a user to request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, evaluation points, and barriers.

ACTION POINT IDENTIFIER: A unique integer ID associated with an action point.

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 it can find information. These expressions are only used within the *type transformation facility* on page 339.

AFFECTED P/T SET: The set of process and threads that will be affected by the command. For most commands, this is identical to the target P/T set, but in some cases it may include additional threads. (See "*P/T (process/thread) set*" on page 336 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.

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 SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 334, *upper bound* on page 340, and *stride* on page 338. Displaying a slice of an array can be useful when you are working with very large arrays.

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*.

AUTOLAUNCHING: When a process begins executing on a remote computer, TotalView can also launch a tvdsvr (TotalView Debugger Sever) process on this 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. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition automatically starts the TotalView Debugger Server (the tvdsvr).

BARRIER: 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 where 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.

CALL FRAME: The memory area containing 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.

CHILD PROCESS: A process created by another process (*see* "*parent process*" on page 335) when that other process calls fork().

CLOSED LOOP: See *closed loop* on page 330.

CLUSTER DEBUGGING: The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

COMMAND HISTORY LIST: A debugger-maintained list storing copies of the most recent commands issued by the user.

CONDITION SYNCHRONIZATION: A process that delays thread execution until a condition is satisfied.

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 fork() are in the same control group.

CORE FILE: A file containing 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.

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 containing the current source location.

CURRENT LIST LOCATION: The location governing what source code will be displayed in response to a list command.

DATA SET: A set of array elements generated by TotalView and sent to the Visualizer. (*See visualizer process* on page 340.)

DBELOG LIBRARY: A library of routines for creating event points and generating event logs from within 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 multiprocess programs. If you link your program with TotalView's dbfork library, TotalView will be able to automatically attach to newly spawned processes.

DEBUGGING INFORMATION: Information relating an executable to the source code from which it was generated.

DEBUGGER INITIALIZATION FILE: An optional file establishing 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 .tvdrc.

DEBUGGER PROMPT: A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER: *See* ***tvdsvr process*** on page 339.

DEBUGGER STATE: Information that TotalView or the CLI maintains in order to interpret and respond to user commands. Includes debugger modes, user-defined commands, and debugger variables.

DEPRECATED: A feature that is still available but may be eliminated in a future release.

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 (PARMACS) run on more than one host.

DIVE STACK: A series of nested dives that were performed in the same Variable Window. The number of greater-than symbols (>) in the upper left-hand corner of a Variable Window indicates the number of nested dives on the dive stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of greater-than symbols shown in the Variable Window.

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.

DOPE VECTOR: This is a runtime 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
```

Assume that you initialize it as follows:

```
iptr => iarray (20:1:-2)
```

iptr is now 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.

EDITING CURSOR: A black rectangle that appears when a TotalView GUI field is selected for editing. You use field editor commands to move the editing cursor.

EVALUATION POINT: A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG: A file containing 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.

EXECUTABLE: A compiled and linked version of source files, containing a "main" entry point.

EXPRESSION: An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current 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's interface. 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 containing the information corresponding to a single invocation of a subprocedure. See *stack frame* on page 337.

FRAME POINTER: See *stack pointer* on page 338.

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.

GID: The TotalView group ID.

GRID: A collection of distributed computing resources available over a local or wide area network that appear as if it was one large virtual computing system.

IMAGE: All of the programs, libraries, and other components that make up your executable is called an image.

GOI: The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and the like.

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 the like.

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 MACHINE: The machine on which the TotalView debugger is running.

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.

INFINITE LOOP: See *loop, infinite* on page 334.

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.

LHS EXPRESSION: This is a synonym for lvalue.

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 333.

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.

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,” describing a type of parallel computing.

MISD: An acronym for “Multiple Instruction, Single Data,” describing a type of parallel computing.

MPI: This is 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 www.mcs.anl.gov/mpi.

MPMD (MULTIPLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving multiple executables, executed by multiple threads and processes.

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 is displayed. So, a nested dive is a *dive* within a dive. You can return to the previous display by selecting the left-facing arrow in the top-right corner of the window.

NODE: A machine on a network. Each machine has a unique network name and address.

OUT OF SCOPE: When symbol lookup is performed for a particular symbol name and it isn't found in the current scope or any containing scopes, the symbol is said to be out of scope.

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 may 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 fork() 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, filename and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PC: This is an abbreviation for *Program Counter*.

PID: Depending on context, this is either the "process ID" or the "program ID." In most cases, this will be a 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 the like.

PROCESS: An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP: A group of processes associated with a multiprocess 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 the like. This is abbreviated as POI.

PROGRAM EVENT: A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

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 execve()

(processes that don't share the same source code as the parent). Contrast with *share group* on page 337.

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.

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.

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 indicating 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 now be run. Prior to this event, they could not be run.

SERIAL EXECUTION: Execution of a program sequentially, one statement at a time.

SERIAL LINE DEBUGGING: A form of remote debugging where TotalView and the TotalView Debugger Server 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. If a service thread is not created by your program, you won't be interested in debugging it. If your program is creating a service thread, however, you will probably debug it separately from the rest of your program.

SHARE GROUP: All the processes in a control group that share the same code. In most cases, your program will have 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—and have no “main” entry point.

SIGNALS: Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not 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,” describing a type of parallel computing.

SISD: An acronym for “Single Instruction, Single Data,” describing a type of parallel computing.

SINGLE STEP: The action of executing a single statement and stopping (as if at a breakpoint).

SLICE: A subsection of an array, which is expressed in terms of a *lower bound* on page 334, *upper bound* on page 340, and *stride* on page 338. Displaying a slice of an array can be useful when you are working with very large arrays.

SOID: An acronym for “symbol object ID”. A soid uniquely identifies all information within TotalView. It also represents a handle by which this information can be accessed.

SOURCE FILE: Program file containing source language statements. TotalView allows you to debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler.

SOURCE LOCATION: For each thread, the source code line it will execute 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 (SINGLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving 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 pointing 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: A *TLA* on page 339 for Standard Template Library.

STOP SET: A set of threads that should be stopped once an action point has been triggered.

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) will be 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 the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is –1, every element between the upper bound and lower bound (reverse order) is displayed.

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 containing 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 such).

SYMBOL TABLE: A table of symbolic names (such as variables or functions) used in a program 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 MACHINE: The machine 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 will act 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 that will be affected by a command. This is abbreviated as TOI.

TID: The thread ID.

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 that will be 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 339.

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: A protocol that allows you to change the way information is displayed. For example, an STL vector can be displayed 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 on the undive icon in the upper right-hand corner of the window.

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 ^C (Ctrl+C).

VARIABLE WINDOW: A TotalView window displaying the name, address, data type, and value of a particular variable.

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 specifying that execution should stop whenever the value of a particular variable is updated.

WORKER THREAD: A thread in a workers group. These are threads created by your program that performs the actual “work.” However, you might want to distinguishes between threads that do the work and threads that assist the work. For example, you might not consider a thread that acts as a queue manager as being a worker thread even though TotalView considers it to be a worker thread. (This kind of thread might be called a “worker-manager” thread.) All worker thread is always part of a workers group.

WORKERS GROUP: All the worker threads in a control group. These threads can reside in more than one share group.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z