TotalView

Classic TotalView User Guide

Version 2023.4
November, 2023

PERFORCE

www.perforce.com

ACKNOWLEDGMENTS

TotalView by Perforce
http://totalview.io

Contents

About This Guide

Content Organization

This guide describes how to use the TotalView debugger, a source- and machine-level debugger for multi-process, multi-threaded programs. It is assumed that you are familiar with programming languages, a UNIX or Linux operating system, and the processor architecture of the system on which you are running TotalView and your program.

This user guide combines information for running the TotalView debugger either from within a Graphic User Interface (GUI), or the Command Line Interface (CLI), run within an xterm-like window for typing commands.

The information here emphasizes the GUI interface, as it is easier to use. Understanding the GUI will also help you understand the CLI.

Although TotalView doesn't change much from platform to platform, differences between platforms are mentioned.

The information in this guide is organized into these parts:

- Part I, Introduction to Debugging with TotalView contains an overview of TotalView features and an introduction to debugging with TotalView.
- Part II, Debugging Tools and Tasks describes the function and use of TotalView's primary set of debugging tools, such as stepping, setting breakpoints, and examining data including arrays.

  This part also includes detail on TotalView's process/thread model and working with multi-process, multi-threaded programs.
- Part III, Using the CLI discusses the basics of using the Command Line Interface (CLI) for debugging. CLI commands are not documented in this book but in the *Classic TotalView Reference Guide*.
- Part IV, Advanced Tools and Customization provides additional information required for setting up various MPI and other parallel programming environments, including high performance computing environments such as MPICH, OpenMP, UPC, and CAF. Setting Up Remote Debugging Sessions discusses how to get the TotalView Debugger Server (**tvdsvr**)

running and how to reconfigure the way that TotalView launches the **tvdsvr**. Group, Process, and Thread Control builds on previous process/thread discussions to provide more detailed configuration information and ways to work in multi-process, multi-threaded environments.

In most cases, TotalView defaults work fine and you won't need much of this information.

- Part V, Using the CUDA Debugger describes the CUDA debugger, including a sample application.

Audience

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

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

TotalView is a rather easy-to-use product. Nonetheless, we can't tell you how to use TotalView to solve your problems because your programs are unique and complex, and we can't anticipate what you want to do. So, what you'll find in this book is a discussion of the kinds of operations you can perform. This book, however, is not just a description of dialog boxes and what you should click on or type. Instead, it tells you how to control your program, see a variable's value, and perform other debugging actions.

Detailed information about dialog boxes and their data fields is in the context-sensitive Help available directly from the GUI. In addition, an HTML version of this information is shipped with the documentation and is available on our Web site. If you have purchased TotalView, you can also post this HTML documentation on your intranet.

Using the CLI

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

The following books are excellent sources of Tcl information:

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

There is also a rich set of resources available on the Web.

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

Resources

Appendix C contains information on:

- TotalView family differences, which details the differences among TotalView Enterprise, TotalView Team, and TotalView Individual
- a complete list of TotalView documentation
- conventions used in the documentation
- contact information

PART I

Introduction to Debugging with TotalView

This part of the *Classic TotalView User Guide* introduces TotalView's basic features and walks through a basic debugging session. Also included here is how to use the Remote Display Client, which allows you to connect to TotalView remotely.

About TotalView

TotalView® for HPC is a powerful tool for debugging, analyzing, and tuning the performance of complex serial, multi-process, multi-threaded, and network-distributed programs. It supports a broad range of platforms, environments, and languages.

TotalView is designed to handle most types of High Performance Computing (HPC) parallel applications, and can be used to debug programs, running processes, or core files.

This chapter introduces TotalView's primary components and features, including:

- **Sessions Manager** for managing and loading debugging sessions, Sessions Manager on page 6
- **Graphical User Interface** with powerful data visualization capabilities, The GUI on page 7
- **Command Line Interface (CLI)** for scripting and batch environments, The CLI on page 7
- **Stepping commands and specialized breakpoints** that provide fine-grained control, Stepping and Breakpoints on page 9
- **Examining complex data sets**, Data Display and Visualization on page 10
- **Controlling threads and processes**, Tools for Multi-Threaded and Parallel Applications on page 22
- **Automatic batch debugging**, Batch and Automated Debugging on page 26
- **Running TotalView remotely**, Remote Display on page 27
- **Debugging CUDA code** running on the host system and the NVIDIA® GPU, CUDA Debugger on page 28
- **Debugging remote programs**, Debugging on a Remote Host on page 27
- **Memory debugging capabilities** integrated into the debugger, Memory Debugging on page 29
- **Recording and replaying running programs**, Reverse Debugging on page 30

Sessions Manager

The Sessions Manager is a GUI interface to manage your debugging sessions. Use the manager to load a new program, to attach to a program, or to debug a core file. The manager keeps track of your debugging sessions, enabling you to save, edit or delete any previous session. You can also duplicate a session and then edit its configuration to test programs in a variety of ways.

RELATED TOPICS	
Managing debugging sessions	Managing Sessions on page 124
Loading programs into TotalView using the Session Manager	Loading Programs from the Sessions Manager on page 102

GUI and Command Line Interfaces

TotalView provides both an easy-to-learn graphical user interface (GUI) and a Command Line Interface (CLI). The CLI and GUI are well integrated, so you can use them both together, launching the CLI from the GUI and invoking CLI commands that display data in the GUI. Or you can use either separately without the other. However, because of the GUI's powerful data visualization capabilities and ease of use, we recommend using it (along with the CLI if you wish) for most tasks.

The GUI

The GUI is an easy and quick way to access most of TotalView's features, allowing you to *dive* on almost any object for more information. You can dive on variables, functions, breakpoints, or processes. Data is graphically displayed so you can easily analyze problems in array data, memory data, your call tree/graph, or a message queue graph.

RELATED TOPICS

GUI Basics and Customizations	Using and Customizing the GUI on page 145

The CLI

The Command Line Interface, or CLI, provides an extensive set of commands to enter into a command window. These commands are embedded in a version of the Tcl command interpreter. You can enter any Tcl statements from any version of Tcl into a CLI window, and you can also make use of TotalView-specific debugging commands. These additional commands are native to this version of Tcl, so you can also use Tcl to manipulate your programs. The result is that you can use the CLI to create your own commands or perform any kind of repetitive operation. For example, the following code shows how to set a breakpoint at line 1038 using the CLI:

```
dbreak 1038
```

When you combine Tcl and TotalView, you can simplify your job. For example, the following code sets a group of breakpoints:

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

RELATED TOPICS	
Using the CLI	Part III, Using the CLI on page 454
CLI commands and reference	"CLI Commands" in the *Classic TotalView Reference Guide*

Stepping and Breakpoints

In TotalView, breakpoints are just a type of *action point,* and there are four types:

- A *breakpoint* stops execution of processes and threads that reach it.
- An *eval point* executes a code fragment when it is reached.
- A *barrier point* synchronizes a set of threads or processes at a location (Synchronizing Execution with Barrier Points on page 25).
- A *watchpoint* monitors a location in memory and stops execution when it changes (Setting Watchpoints on page 16).

You can set action points 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 appears at the line number to indicate 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 resume or otherwise control program execution in any of the following ways:

- Use the single-step commands described in Using Stepping Commands on page 178.
- 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 187.
- Set breakpoints at lines you choose, and let your program execute to that breakpoint. See Setting Breakpoints and Barriers on page 194.
- Set conditional breakpoints that cause a program to stop after it evaluates a condition that you define, for example, “stop when a value is less than eight.” See Setting Eval Points on page 222.

RELATED TOPICS	
Detailed information on action points	Setting Action Points on page 188
Stepping commands	Stepping through and Executing your Program on page 177

Data Display and Visualization

TotalView provides comprehensive and flexible tools for developers to explore large and complex data sets. The TotalView data window supports browsing through complex structures and arrays. Powerful slicing and filtering helps developers manage arrays of thousands or even millions of elements. Data watchpoints provide answers to questions about how data is changing. Built-in graphical visualization displays a quick view of complex numerical data. Type transformations, especially C++ View, help you display data in a meaningful way.

This section includes:

RELATED TOPICS	
Viewing and editing data	Examining and Editing Data and Program Elements on page 240
Data Visualization	Visualizing Programs and Data on page 334

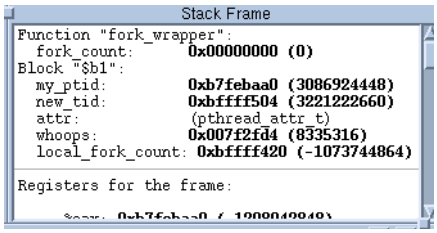
Data Display

All variables in your current routine are displayed in the Process Window's Stack Frame Pane in its upper right corner, Figure 1. If a variable's value is simple, it is visible here. If the value is not simple, *dive* on the variable to get more information.

NOTE: You can dive on almost any object in TotalView to display more information about that object. To dive, position the cursor over the item and click the middle mouse button or double-click the left mouse button.

Some values in the Stack Frame Pane are **bold**, meaning that you can click on the value and edit it.

Figure 1, Stack Frame Pane of the Process Window



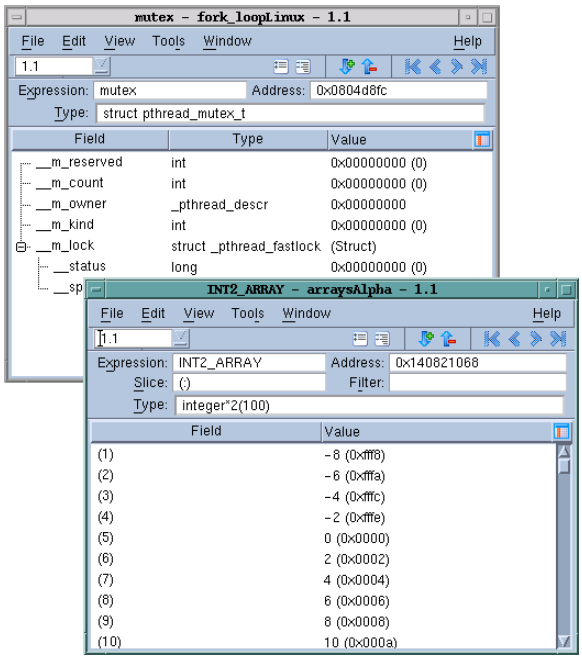
This section includes:

RELATED TOPICS	
Diving in general	About Diving into Objects on page 164
Displaying non-scalar variables	Displaying Variables on page 246
Editing variables	Changing the Values of Variables on page 281 and Changing a Variable's Data Type on page 283

Diving in a Variable Window

Figure 2 shows two Variable Windows, one created by diving on a structure and the second by diving on an array.

Figure 2, Diving on a Structure and an Array



If the displayed data is not scalar, you can redive on it for more information. When you dive in a Variable Window, TotalView replaces the window's contents with the new information, or you can use the **View > Dive Thread in New Window** command to open a separate window.

For pointers, diving on the variable dereferences the pointer and displays the data pointed to. In this way, you can follow linked lists.

Buttons in the upper right corner () support undives and redives. For example, if you're following a pointer chain, click the center-left arrow to go back to where you just were. Click the center-right arrow to move forward.

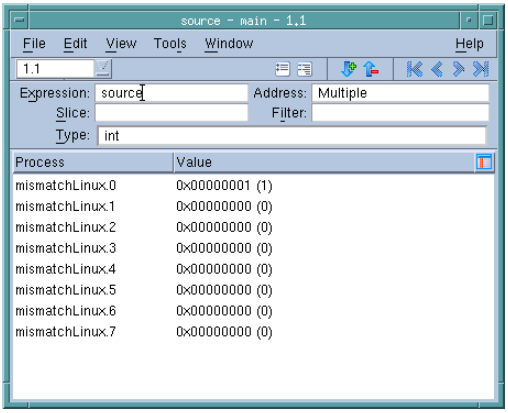
RELATED TOPICS

Diving in variable windows	Diving in Variable Windows on page 266
Duplicating variable windows	The **Window > Duplicate** command, in the in-product Help

Viewing a Variable Value across Multiple Processes or Threads

You can simultaneously see the value of a variable in each process or thread using the **View > Show Across > Thread** or **View > Show Across > Process** commands, Figure 3.

Figure 3, Viewing Across Processes



You can export data created by the **Show Across** command to the Array Visualizer (see The Array Visualizer on page 17).

RELATED TOPICS

The **View > Show Across**... command	Displaying a Variable in all Processes or Threads on page 330
Exporting a **Show Across** view to the Visualizer	Visualizing a "Show Across" Variable Window on page 333

Simplifying Array Display

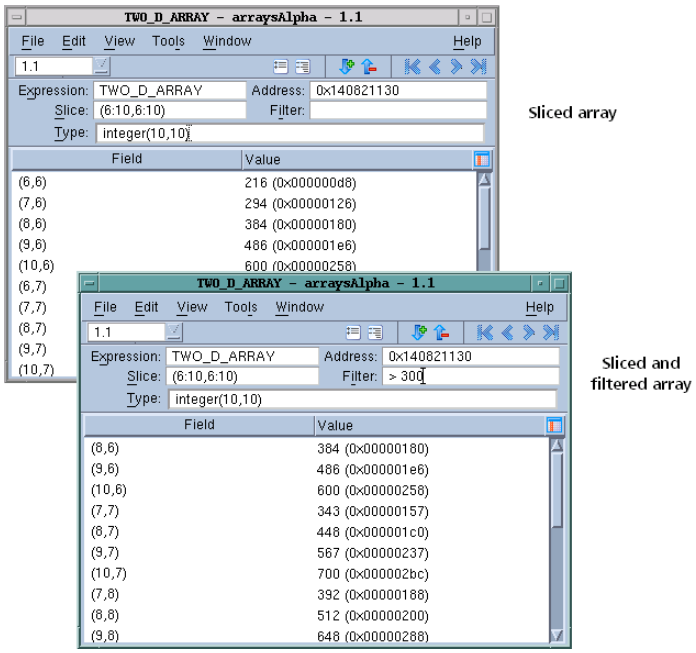
Because arrays frequently have large amounts of data, TotalView provides a variety of ways to simplify their display, including *slicing* and *filtering*, and a special viewer, the Array Viewer.

Slicing and Filtering

The top Variable Window of Figure 4 shows a basic slice operation that displays array elements at positions named by the slice. In this case, TotalView is displaying elements 6 through 10 in each of the array's two dimensions.

The other Variable Window combines a filter with a slice to display data according to some criteria. Here, the filter shows only elements with a value greater than 300.

Figure 4, Slicing and Filtering Arrays



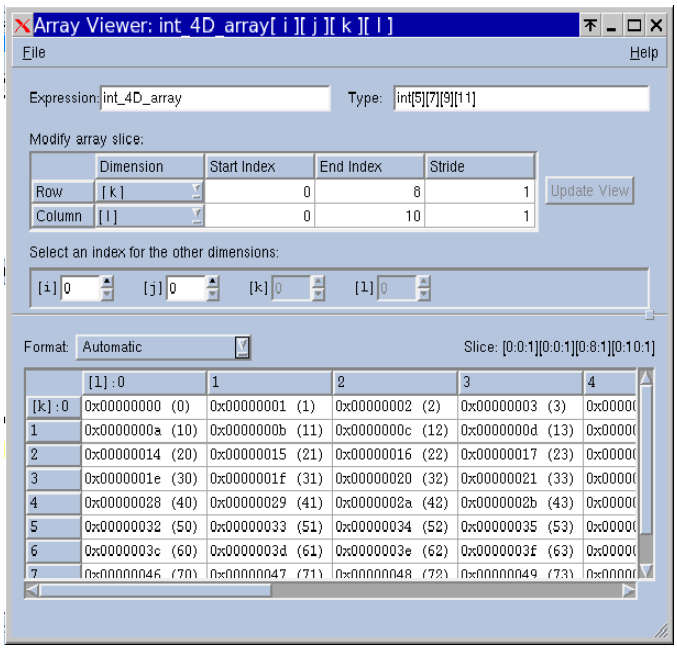
RELATED TOPICS

Arrays in general	Examining Arrays on page 312
Filtering in arrays	Filtering Array Data Overview on page 319
Array slices	Displaying Array Slices on page 313

The Array Viewer

Use the Array Viewer (from the Variable Window's **Tools > Array Viewer** command) for another graphical view of data in a multi-dimensional array, Figure 5. Think of this as viewing a "plane" of two-dimensional data in your array.

Figure 5, Array Viewer



The Array Viewer initially displays a slice of data based on values entered in the Variable Window. You can change the displayed data by modifying the Expression, Type, or Slice controls.

You can also see the shape of the data using the Visualizer, introduced in this chapter in The Array Visualizer on page 17.

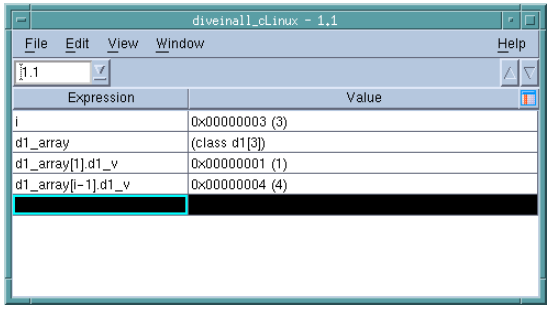
RELATED TOPICS

The Array Viewer	Viewing Array Data on page 317

Viewing a Variable's Changing Value

Variable Windows let you critically examine many aspects of your data. In many cases, however, you may be primarily interested in the variable's value. For this, use the Expression List Window, Figure 6, to display the values of many variables at the same time.

Figure 6, Tools > Expression List Window



This is particularly useful for viewing variable data about scalar variables in your program.

RELATED TOPICS	
Lists of variables in the Expression List Window	Viewing a List of Variables on page 272
TotalView's comprehensive expression system	Evaluating Expressions on page 360

Setting Watchpoints

The *watchpoint* — another type of action point — is yet another way to look at data. A TotalView watchpoint can stop execution when a variable's data changes, no matter the cause. That is, you could change data from within 30 different statements, triggering the watchpoint to stop execution after each of these 30 statements make a change. Or, if data is being overwritten, you could set a watchpoint at that location in memory and then wait until TotalView stops execution because of an overwrite.

If you associate an expression with a watchpoint (by selecting the Conditional button in the Watchpoint Properties dialog box to enter an expression), TotalView evaluates the expression after the watchpoint triggers.

RELATED TOPICS	
About watchpoints in general	Using Watchpoints on page 231
The **Tools > Create Watchpoint** command	**Tools > Create Watchpoint** in the in-product Help

Data Visualization

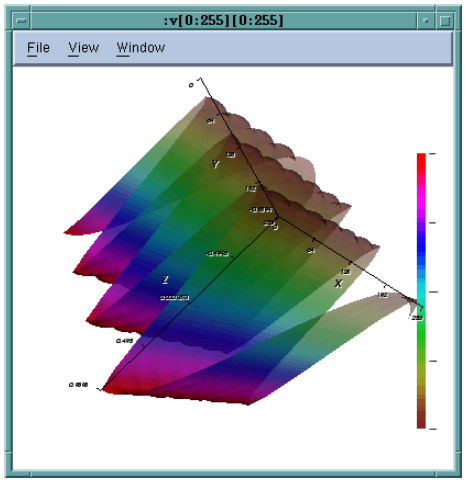
TotalView provides a set of tools to visualize your program activity, including its arrays and MPI message data. These include:

- The Array Visualizer on page 17
- The Parallel Backtrace View on page 18
- The Call Tree and Call Graph on page 18
- The Message Queue Graph on page 20

The Array Visualizer

The Variable Window's **Tools > Visualize** command shows a graphical representation of a multi-dimensional dataset. For instance, Figure 7 shows a sine wave in the Visualizer.

Figure 7, Visualizing an Array



This helps you to quickly see outliers or other issues with your data.

RELATED TOPICS

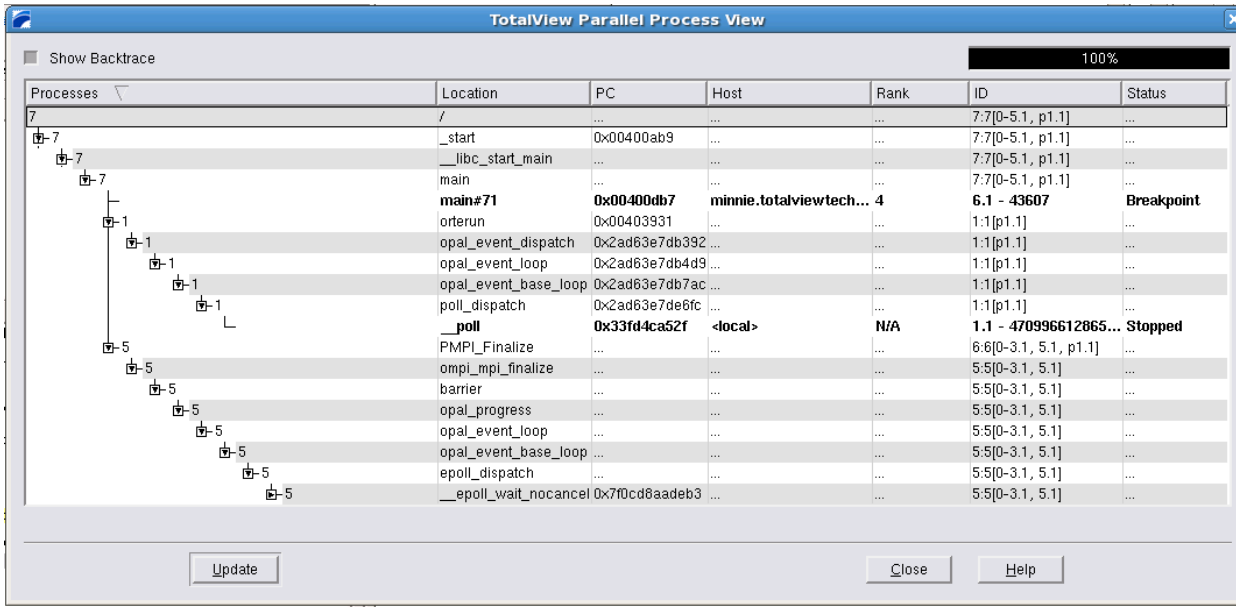
The Visualizer	Visualizing Programs and Data on page 334
Using the **Tools > Visualize** and the **Tools > Visualize Distribution** commands	The Visualizer Window in the in-product Help

The Parallel Backtrace View

The Parallel Backtrace View displays the state of every process and thread in a parallel job, allowing you to view thousands of processes at once, and helping you to identify stray processes.

Access the Parallel Backtrace View from the Tools menu of the Variable Window.

Figure 8, Parallel Backtrace View



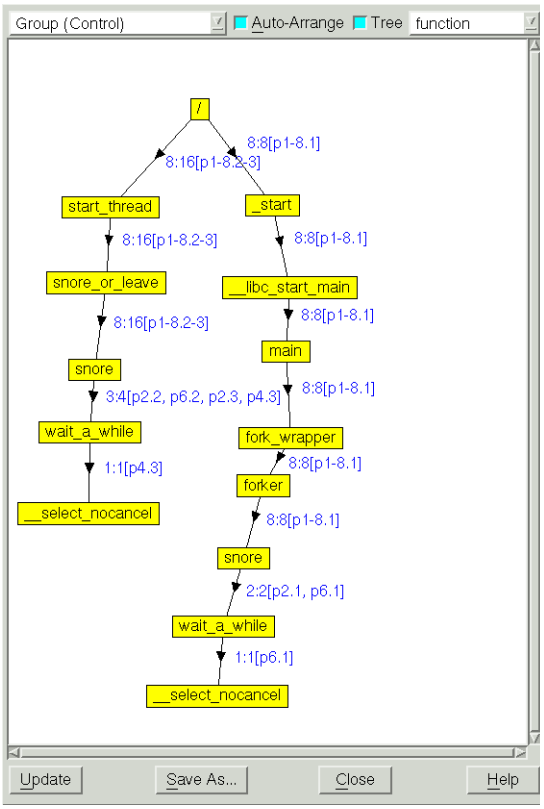
This view groups threads by common stack backtrace frames in a text-based tree. Expand or collapse elements to drill down and get more information.

RELATED TOPICS	
The Parallel Backtrace View	Parallel Backtrace View on page 338
Using the **Tools > Parallel Backtrace View** command	**Tools > Parallel Backtrace View** in the in-product Help

The Call Tree and Call Graph

The Call Tree or Call Graph, accessible from the Process Window using the command **Tools > Call Graph**, provides a quick view of application state and is especially helpful for locating outliers and bottlenecks.

Figure 9, Tools > Call Graph Dialog Box



By default, the Call Tree or Call Graph displays the Call Tree representing the backtrace of all the selected processes and threads.

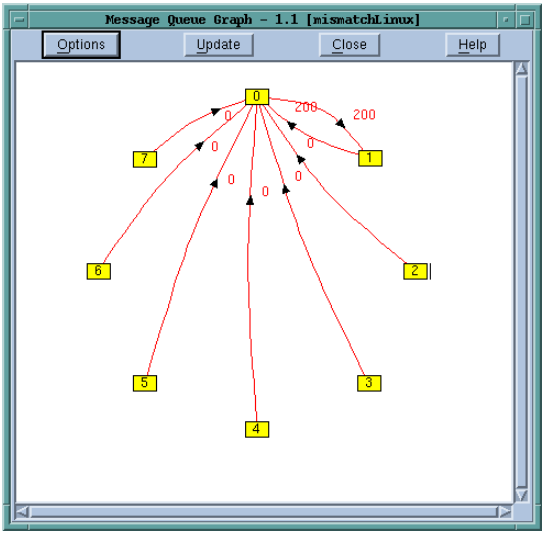
For multi-process or multi-threaded programs, a compressed process/thread list (**ptlist**) next to the arrows indicates which threads have a routine on their call stack. You can dive on a routine in the call tree/graph to create a group called `call_graph` that contains all the threads that have the routine you dived on in their call stack.

RELATED TOPICS	
The Call Tree or Call Graph in more detail	Displaying Call Trees and Call Graphs on page 335
Using the CLI's **dwhere -group_by** option to control and reduce the backtraces	**dwhere -group_by** in the *Classic TotalView Reference Guide*
About the **ptlist**	"Compressed List Syntax (ptlist)" in the *Classic TotalView Reference Guide*
Using the **Tools > Call Graph** command	**Tools > Call Graph** in the in-product Help

The Message Queue Graph

For MPI programs, use the Process Window's Message Queue Graph (**Tools > Message Queue Graph)** to display your program's message queue state.

Figure 10, A Message Queue Graph



The graph's Options window (available by clicking on the Options button, above) provides a variety of useful tools, such as Cycle Detection to generate reports about cycles in your messages, a helpful way to see when messages are blocking or causing deadlocks. Also useful is its filtering capability, which helps you identify pending send and receive messages, as well as unexpected messages.

RELATED TOPICS	
The Message Queue Graph	Displaying the Message Queue Graph Window on page 446
Using the **Tools > Message Queue Graph** command	**Tools > Message Queue Graph** in the in-product Help

C++ View

Using C++ View, you can format program data in a more useful or meaningful form than its concrete representation displayed in a running program. This allows you to inspect, aggregate, and check the validity of complex data, especially data that uses abstractions such as structures, classes, and templates.

RELATED TOPICS

More on C++ View	Creating Type Transformations" in the *Classic TotalView Reference Guide*

Tools for Multi-Threaded and Parallel Applications

TotalView is designed to debug multi-process, multi-threaded programs, with a rich feature set to support fine-grained control over individual or multiple threads and processes. This level of control makes it possible to quickly resolve problems like deadlocks or race conditions in a complex program that spawns thousands of processes and threads across a broad network of servers.

When your program creates processes and threads, TotalView can automatically bring them under its control, whether they are local or remote. If the processes are already running, TotalView can acquire them as well, avoiding the need to run multiple debuggers.

TotalView places a server on each remote processor as it is launched that then communicates with the main TotalView process. This debugging architecture gives you a central location from which you can manage and examine all aspects of your program.

This section introduces some of TotalView's primary tools for working with complex parallel applications, and includes:

- Program Using Almost Any Execution Model on page 22
- View Process and Thread State on page 22
- Control Program Execution on page 23

Program Using Almost Any Execution Model

TotalView supports the popular parallel execution models MPI and MPICH, OpenMP, ORNL, SGI shared memory (shmem), Global Arrays, UPC, and CAF.

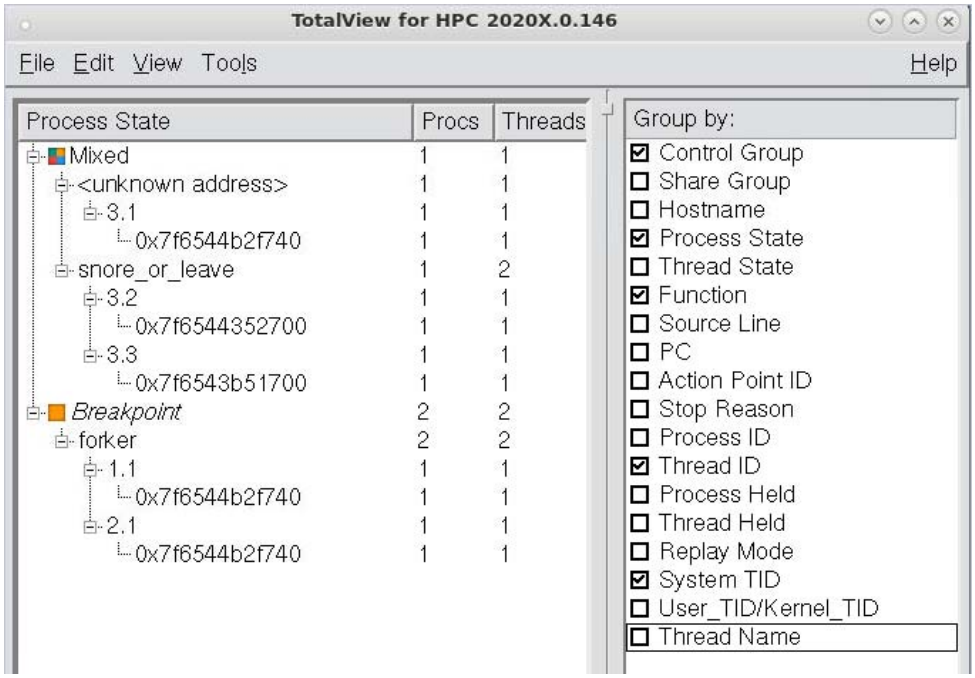
RELATED TOPICS	
MPI debugging sessions	Setting Up MPI Debugging Sessions on page 516
Other parallel environments (not MPI)	Setting Up Parallel Debugging Sessions on page 546

View Process and Thread State

You can quickly view process and thread state in both the Root Window and the Process Window. (You can also view the state of *all* processes and threads in a parallel job using the The Parallel Backtrace View on page 18.)

The Root Window contains an overview of all processes and threads being debugged, along with their process state (i.e. stopped, running, at breakpoint, etc.). You can dive on a process or a thread listed in the Root Window to quickly see detailed information.

Figure 11, The Root Window



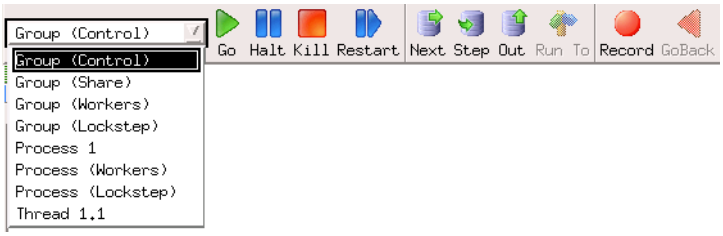
RELATED TOPICS	
More on process and thread state	Viewing Process and Thread States on page 409
The Root Window	Using the Root Window on page 147
The Process Window	Using the Process Window on page 157

Control Program Execution

Commands controlling execution operate on the current focus, or target -- either an individual thread or process, or a group of threads and processes. You can individually stop, start, step, and examine any thread or process, or perform these actions on a group.

Select the target of your action from the toolbar's pulldown menu, Figure 12.

Figure 12, Selecting a Target from the Toolbar Pulldown



You can also synchronize execution across threads or processes using a *barrier point*, which holds any threads or processes in a group until each reaches a particular point.

RELATED TOPICS	
Selecting a target (also called focus)	Using the Toolbar to Select a Target on page 416
Setting process and thread focus using the CLI	Setting Process and Thread Focus on page 579
Setting group focus	Setting Group Focus on page 585
Finely controlling focus	Defining the GOI, POI, and TOI on page 572
Introduction to barrier points	Synchronizing Execution with Barrier Points on page 25

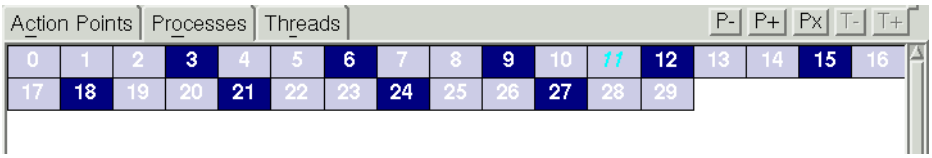
Using Groups

TotalView automatically organizes your processes and threads into groups, allowing you to view, execute and control any individual thread, process, or group of threads and processes. TotalView defines built-in groups, and you can define your own custom groups that help support full, asynchronous debugging control over your program.

For example, you can:

- Single step one or a small set of processes rather than all of them
- Use **Group > Custom Group** to create named groups
- Use **Run To** or breakpoints to control large groups of processes
- Use *Watchpoints* to watch for variable changes

For instance, here is the Processes Tab after a group containing 10 processes (in dark blue below) is selected in the Toolbar's Group pulldown list. This identifies the processes that will be acted on when you select a command such as **Go** or **Step**.



RELATED TOPICS	
Groups in TotalView	Organizing Chaos on page 394
How TotalView predefines groups	How TotalView Creates Groups on page 398
Introduction to setting watchpoints	Setting Watchpoints on page 16
Creating custom groups	Creating Custom Groups on page 602

Synchronizing Execution with Barrier Points

You can synchronize execution of threads and processes either manually using a *hold* command, or automatically by setting an action point called a *barrier point*. These two tools can be used together for fine-grained execution control. For instance, if a process or thread is held at a barrier point you can manually release it and then run it without first waiting for all other processes or threads in the group to reach that barrier.

When a process or a thread is *held*, it ignores any command to resume executing. For example, assume that you place a hold on a process in a control group that contains three processes. If you select **Group > Go**, two of the three processes resume executing. The held process ignores the **Go** command.

RELATED TOPICS	
Setting barrier points	Setting Breakpoints and Barriers on page 194 and
Using barrier points in a multi-threaded, multi-process program	Using Barrier Points on page 425
Using the CLI to set barrier points	Using Action Points on page 471
Strategies for using barrier points	Simplifying What You're Debugging on page 404 and Breakpoints, Stepping, and Program Execution on page 438

Batch and Automated Debugging

You can set up unattended batch debugging sessions using TotalView's powerful scripting tool **tvscript**. First, define a series of events that may occur within the target program. **tvscript** loads the program under its control, sets breakpoints as necessary, and runs the program. At each program stop, **tvscript** logs the data for your review when the job has completed.

A script file can contain CLI and Tcl commands (Tcl is the basis for TotalView's CLI).

Here, for example, is how **tvscript** is invoked on a program:

```
tvscript \
  -create_actionpoint "method1=>display_backtrace -show_arguments" \
  -create_actionpoint "method2#37=>display_backtrace -show_locals -level 1" \
  -display_specifiers "nowshow_pid,noshow_tid" \
  -maxruntime "00:00:30" \
      filterapp -a 20
```

You can also execute MPI programs using **tvscript**. Here is a small example:

```
tvscript -mpi "Open MP" -tasks 4 \
     -create_actionpoint \
       "hello.c#14=>display_backtrace" \
      ~/tests/MPI_hello
```

While batch debugging of large-scale MPI applications through **tvscript** has long been a powerful tool, **tvscript** has recently been enhanced and fully certified to handle 1024 process jobs, and 2048 threads per process, or more than two million running operations.

RELATED TOPICS	
About **tvscript** and batch scripting, including memory debugging	Batch Scripting and Using the CLI" in *Debugging Memory Problems with MemoryScape*
tvscript syntax and command line options	Batch Debugging Using tvscript" in the *Classic TotalView Reference Guide*

Remote Display

Using the Remote Display Client, you can easily set up and operate a TotalView debug session that is running on another system. A licensed copy of TotalView must be installed on the remote machine, but you do not need an additional license to run the Client.

The Client also provides for submission of jobs to batch queuing systems PBS Pro and Load Leveler.

RELATED TOPICS	
Using the Remote Display Client	Accessing TotalView Remotely on page 65

Debugging on a Remote Host

Using the TotalView Server, you can debug programs located on remote machines. Debugging a remote process is similar to debugging a native process, although performance depends on the load on the remote host and network latency. TotalView runs and accesses the process **tvdsvr** on the remote machine.

RELATED TOPICS	
The TotalView Server	Setting Up Remote Debugging Sessions on page 484
The **tdvsvr** process:	"The tvdsvr Command and Its Options" in the *Classic TotalView Reference Guide*

CUDA Debugger

The TotalView CUDA debugger is an integrated debugging tool capable of simultaneously debugging CUDA code that is running on the host system and the NVIDIA® GPU. CUDA support is an extension to the standard version TotalView, and is capable of debugging 64-bit CUDA programs. Debugging 32-bit CUDA programs is currently not supported.

Supported major features:

- Debug a CUDA application running directly on GPU hardware
- Set breakpoints, pause execution, and single step in GPU code
- View GPU variables in PTX registers, and in local, parameter, global, or shared memory
- Access runtime variables, such as threadIdx, blockIdx, blockDim, etc.
- Debug multiple GPU devices per process
- Support for the CUDA MemoryChecker
- Debug remote, distributed and clustered systems
- All host debugging features are supported, except ReplayEngine. For a list of supported hosts, please see the *TotalView Supported Platforms* guide.

RELATED TOPICS	
The CUDA debugger	Using the CUDA Debugger on page 626
The CLI **dcuda** command	**dcuda** in the *Classic TotalView Reference Guide*

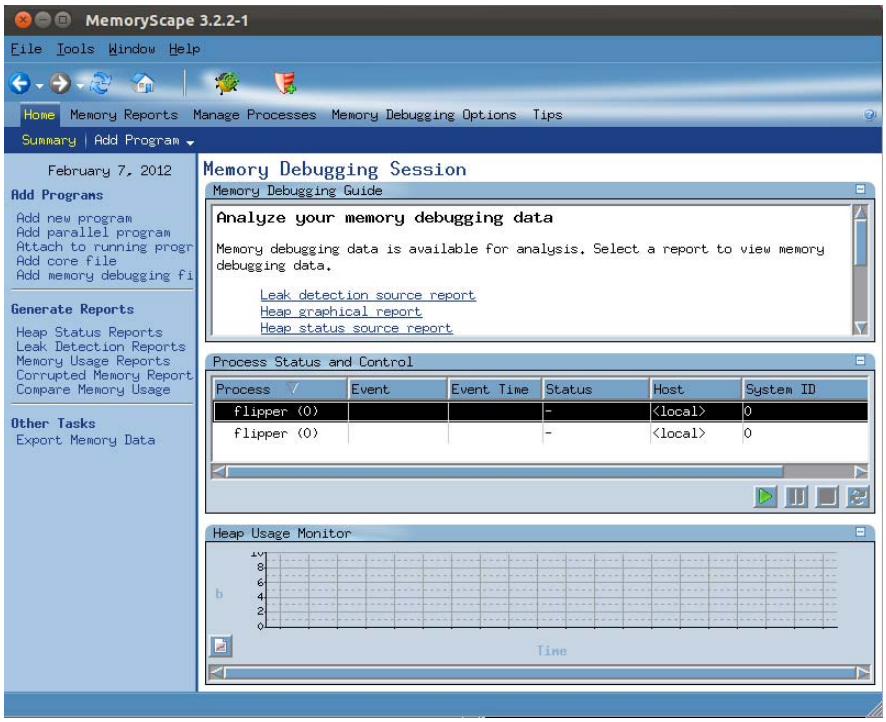
Memory Debugging

TotalView has a fully integrated version of the MemoryScape product for debugging memory issues. MemoryScape is also available as a standalone product.

MemoryScape can monitor how your program uses **malloc()** and **free()** and related functions such as **calloc()** and **realloc()**. For example, the C++ **new** operator is almost always built on top of the **malloc()** function. If it is, MemoryScape can track it.

You must enable memory debugging *before* you start running your program. Once you have loaded a program to debug in TotalView, select **Debug > Open MemoryScape** to launch the primary MemoryScape window.

Figure 13, MemoryScape home window



RELATED TOPICS

MemoryScape has its own user guide *Debugging Memory Problems with MemoryScape.*

Reverse Debugging

ReplayEngine records all your program's activities as it executes within TotalView. After recording information, you can move forward and backward by function, line, or instruction. You can examine data and state in the past, as if it were a live process.

Using ReplayEngine eliminates the cycle of starting and restarting so common in debugging and greatly helps in finding hard-to-reproduce bugs.

NOTE: ReplayEngine is a separately licensed product for Linux-x86-64.

When enabled, ReplayEngine commands are added to the toolbar (at right):

RELATED TOPICS

Reverse debugging is discussed in a separate user guide	*Reverse Debugging with ReplayEngine*

What's Next

This chapter has presented TotalView's primary features and tools, but a single chapter cannot provide a complete picture of everything you can do with TotalView. See the rest of this user guide and other books in the TotalView documentation for more about TotalView.

If you are a new TotalView user, we recommend reading *Getting Started with TotalView Products*, which provides basic information on TotalView's most commonly used tools.

You may also wish to work through the introductory tutorial in Basic Debugging on page 32.

Basic Debugging

This chapter illustrates some basic debugging tasks and is based on the shipped program, **wave_extended**, located in the directory *installdir*/**toolworks**/*totalview.version/platform*/**examples**. This is a simple program that creates an array and then increments its values to simulate a wave form which can then be viewed using the Visualizer. The program requires user input to provide the number of times to increment.

NOTE: TotalView has a new user interface with improved debugging workflows, features, and a modern look and feel. Existing TotalView users can opt to use the new UI by selecting the UI preference on the Display tab in the Preferences dialog.

For new TotalView users, the new UI is the default, but you can revert to the Classic TotalView UI, if necessary, by changing the Display preference on the Preferences tab. To learn more about using the new UI, see the new UI HTML documentation in the TotalView distribution at `<installdir>/totalview.<version>/help/html/TotalView_Help`, or the online TotalView documentation set and Getting Started Guide.

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

- Use the **-g** option to compile the program. (Compiling is not discussed here. Please see Compiling Programs on page 87.)
- Start the program under TotalView control.
- Start the debugging process, including setting breakpoints and examining your program's data.

The chapter introduces some of TotalView's primary tools, as follows:

Program Load and Navigation

This section discusses how to load a program and looks at the two primary TotalView windows, the Root and Process windows. It also illustrates some of TotalView's navigation tools.

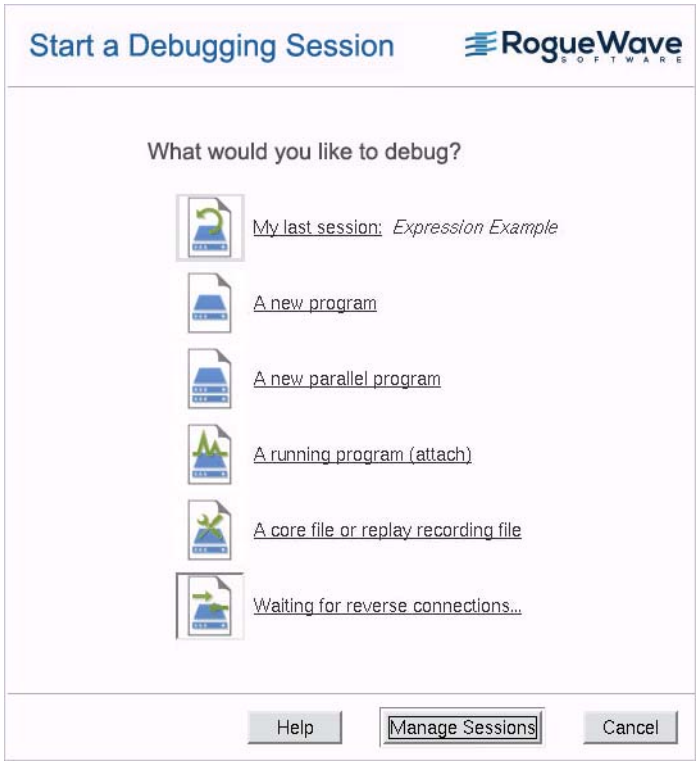
Load the Program to Debug

NOTE: Before starting TotalView, you must add TotalView to your PATH variable. For information on installing or configuring TotalView, see the *TotalView Installation Guide*.

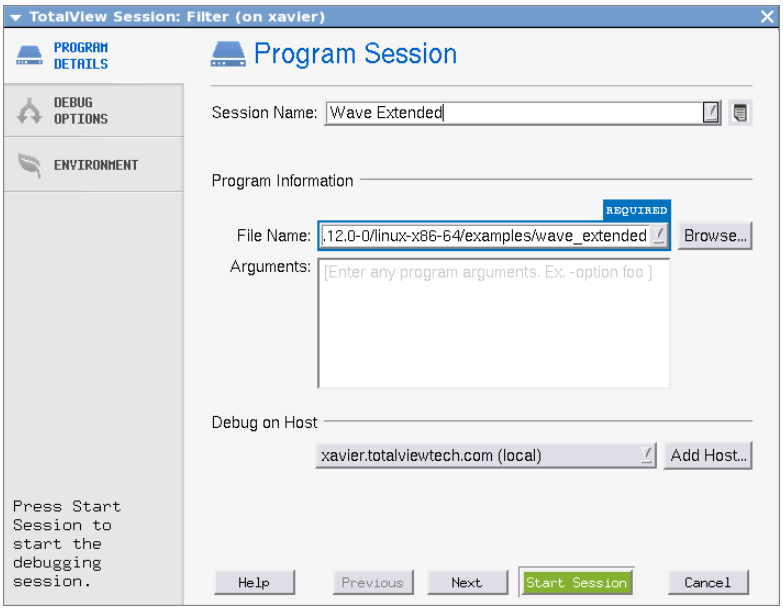
1. Start TotalView.

   ```
   totalview
   ```

   The Start a Debugging Session dialog launches.



2. Select **A new program** to launch the Program Session dialog.



3. Provide a name for the session in **Session Name** field. This can be any string.
4. In the **File Name** field, browse to and select the **wave_extended** program, located in the directory *install-dir*/**toolworks**/*totalview.version/platform*/**examples**. Leave all other fields and options as is. Click **Start Session** to load the program into TotalView.

Note that this is the same as entering the program name as an argument when starting TotalView:

```
totalview wave_extended
```

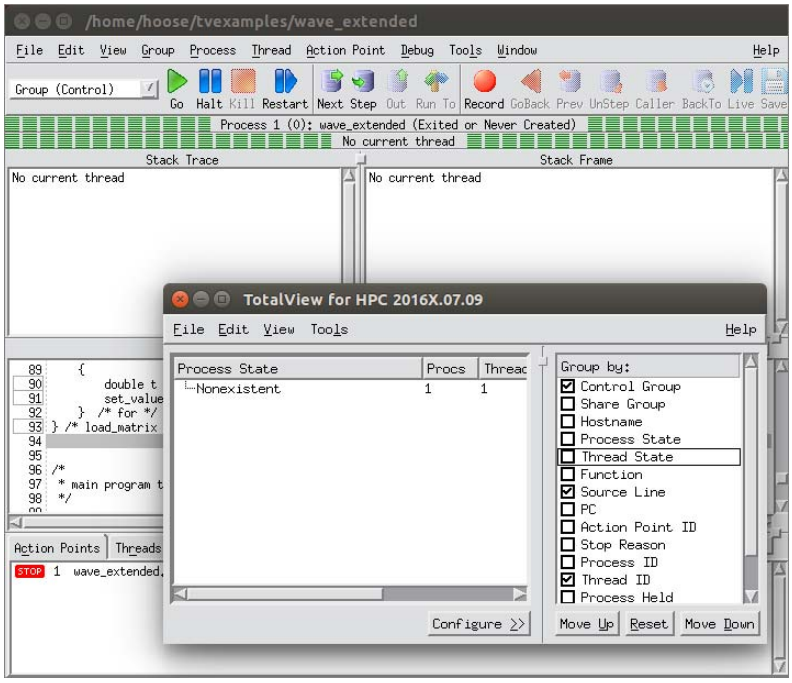
(Note that this invocation assumes that your **examples** directory is known to TotalView or that you are invoking TotalView from within the **examples** directory.)

RELATED TOPICS	
Compiling programs for debugging	Compiling Programs on page 87
Various ways to start TotalView	Starting TotalView on page 89
Loading programs	Loading Programs from the Sessions Manager on page 102

The Root and Process Windows

At startup, TotalView launches its two primary windows, the Root Window and the Process Window. With these two windows, you can navigate through the various elements of your program.

Figure 14, The Root and Process Windows



The Root Window

The Root Window (the smaller window above) lists all processes and threads under TotalView control. You can use the Configure pane, displayed by clicking the button on the bottom right, to specify the specific information you want to view.

Since the program has been created but not yet executed, there is just a single process and thread listed.

The Process Window

The Process Window displays a wide range of information about the state of a process and its individual threads.

- The Stack Trace pane displays the call stack with any active threads.
- The Stack Frame pane displays information on the current thread's variables.
- The Source Pane displays source code for the **main()** function. Note that the pane's header reports its focus as being in **main()**:
- Two tabs are visible at the bottom, Action Points, which displays any set action points, and Threads, which lists all active threads in the process. The Processes/Ranks tab, if enabled, displays processes within the current control group. The Processes/Ranks tab is disabled by default.

RELATED TOPICS	
The Root Window	Using the Root Window on page 147
The Process Window	Using the Process Window on page 157
Processes/Ranks tab	Using the Processes/Ranks and Threads Tabs on page 418

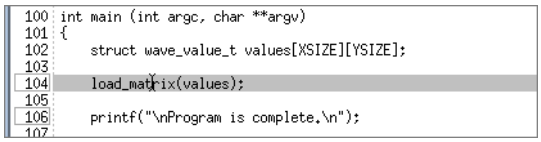
Program Navigation

From the Root and Process Windows, you can navigate anywhere in your program. Some examples:

1. **Dive on a function**

   - From the Process Window, in **main()**, **"**dive**"** on the function **load_matrix()** by double-clicking on it. (Click on the text, not on the line number, which would instead add an Action Point.)

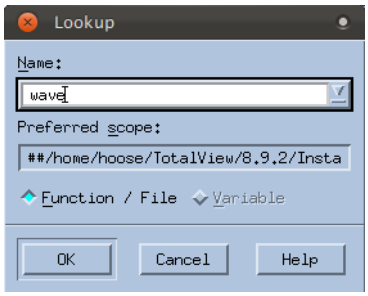
> **NOTE:** *Diving* simply means clicking on an object to launch a window with detailed information. Diving is integral to using TotalView and provides instant access to detailed data by drilling down into the object, routine, process, thread, etc.



The focus in the Source Pane shifts to this function. You can change the focus back to **main()** using the *dive stack* icons ( ) at the top right. If you click the left arrow, the focus returns to **main()** and the right arrow becomes enabled, allowing you to dive, undo a dive, and then redive.

2. **Look up a function**

   - From the View menu, select **Lookup Function**, then enter **wave**:

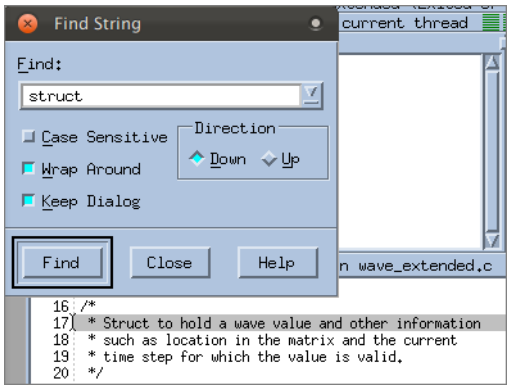


The focus shifts to the function **wave()**. This function happens to be in the same file, **wave_extended.c**, but finding, diving, and other navigation tools operate on any file in a project.

3. **Find any program element**

   - From the Edit menu, select **Find**.

     You can enter any search term, and TotalView returns results from anywhere in your program, including from assembler code if it is visible. For instance, a search on “struct” returns several instances:



RELATED TOPICS	
Diving on objects	About Diving into Objects on page 164
Finding program elements	Searching and Navigating Program Elements on page 168
The Lookup Function	Looking for Functions and Variables on page 169

Stepping and Executing

The Process Window's toolbar features buttons that control stepping and execution.

Figure 15, Toolbar

The following sections explore how these work using the **wave_extended** example.

NOTE: These procedures on stepping and execution can be performed independently of the other tasks in this chapter, but you must first load the program, as described in Load the Program to Debug on page 33.

Simple Stepping

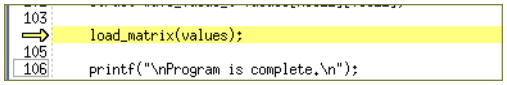
Here, we'll use the commands **Step**, **Run To**, and **Next**, and then note process and thread status.

1. **Step**

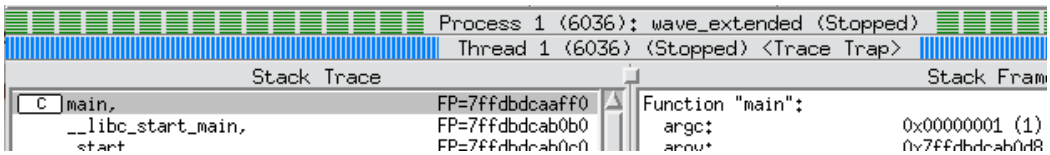
   - Select **Step**. TotalView stops the program just before the first executable statement, the method **load_matrix()**.

     All stepping functions are under the Process, Thread, and Group menus. So for the above, you could also select **Process > Step**, or just press the keyboard shortcut **s** — keyboard shortcuts are all listed under the above menus, and can considerably speed debugging tasks.

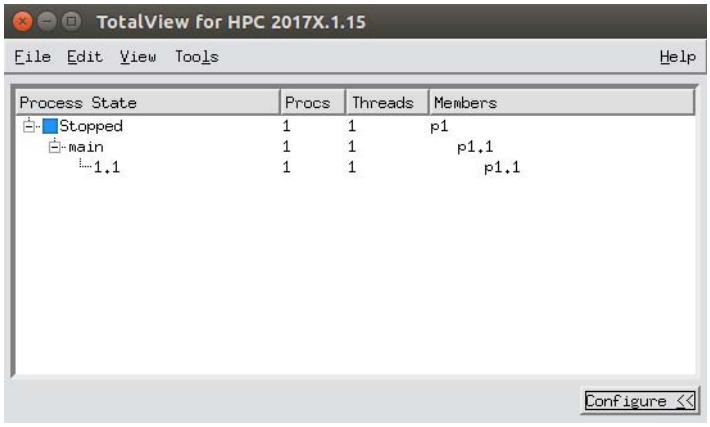
     Note the yellow arrow that shows the current location of the Program Counter, or **PC**, in the selected stack frame.

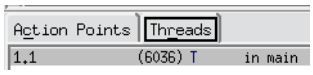
     The process and thread status are displayed in the status bars above the Stack Trace and Stack Frame panes:



The Root Window also displays process/thread status. The Process State column displays the state of the thread:



You can also see that a single Process and Thread have been launched in the Process Window's Thread tab at the bottom of the interface. (1.1 indicates one process and one thread.)



A single thread has been spawned, which reports that it is in **main()**.

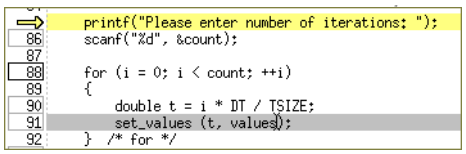
- Select **Step** again to *step into* the function. (**Next** would *step over,* or execute the function, as described in Step 3.)

  TotalView goes into the **load_matrix()** function.

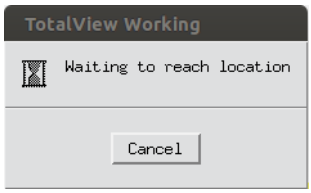
  The Source Pane header reports that the program is in **load_matrix()**, and the PC is at **printf()**.

2. Run To

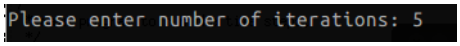
- Select the **set_values()** function at line 91, then click **Run To** in the toolbar.



The program attempts to run to the selected line. Note that the PC does not change, and TotalView launches a popup:



Because the method **set_values()** is called after **scanf()**, the program is waiting for user input. From the shell that launched TotalView, enter **5** at the prompt "Please enter number of iterations", then hit Return. (You can enter a different number, but a higher value will require you to wait over more iterations during later discussions.)

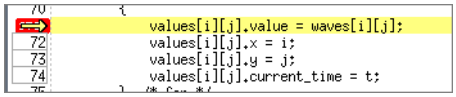


The PC stops at **set_values()**.

3. Next

- Select **Next**. The program executes the for loop the number of times you input in the previous step, and then completes the program by printing "Program is complete." to the console. (If you had instead selected **Step**, the program would have gone *into* the **set_values()** function.)

  The **Next** command simply executes any executable code at the location of the PC. If that is a function, it fully executes the function. If the PC is instead at a location within a function, it executes that line and then moves the PC to the next line. For instance, below the PC is setting a variable value. In this case, **Next** executes line 71, and then moves the PC to line 72.



(Note that the array building the wave is not visible, as there is no program output. To examine or visualize data, including array data, we'll use the Variable Window and the Visualizer, discussed in Examining Data on page 49 and Visualizing Arrays on page 59.)

To just run the program, select **Go**. This may be useful if you entered a larger number into the console, so you can avoid iterating through the for loop numerous times.

RELATED TOPICS

Detailed information on stepping	Stepping through and Executing your Program on page 177
Stepping instructions	Using Stepping Commands on page 178

Canceling

First, make sure the program has exited, by selecting **Kill**.

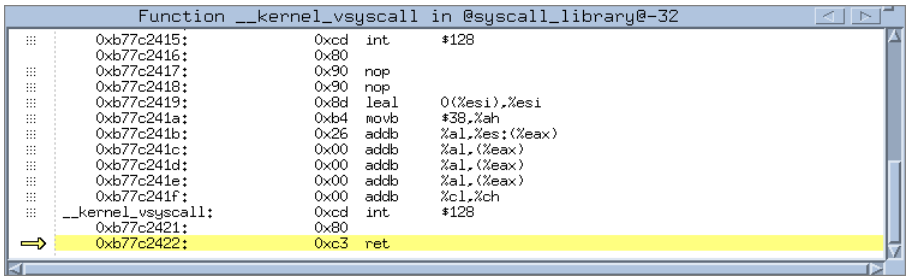
1. **Execute until user input is required**

   - Select **Next** twice. The "Waiting to reach location" dialog launches.

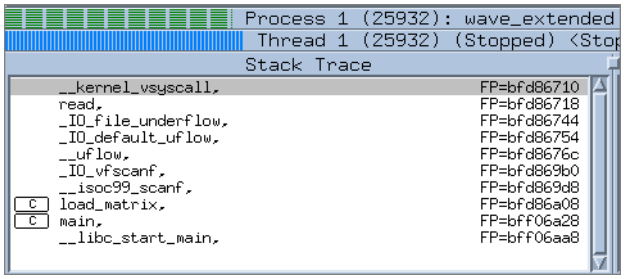
2. **Cancel the operation**

   - Rather than providing input, in the dialog, select **Cancel**.

     The Stack Trace Pane reports that the process is currently within a system call. The Source Pane displays assembler code, and its header identifies the library you're in, rather than the source file. This is because no debug information is present for system calls, and TotalView always focuses on the stack frame where your PC is, even if it was not built with debug information.



     In the Stack Trace Pane, **main** is preceded by C, meaning that TotalView has debug information for that frame, and the language is C.



     To execute out of the assembler code so you're back in your code, use the **Out** command.

   - Select **Out** several times until the program returns to your code and resumes execution. When the dialog "Waiting to reach location" launches, enter a number into the console, click **Next**, and let the program complete.

RELATED TOPICS

Viewing assembler code	Viewing the Assembler Version of Your Code on page 173 and "View > Assembler > By Address" in the in-product Help

Setting Breakpoints (Action Points)

In TotalView, a breakpoint is just one type of "action point" of which there are four types:

- **Breakpoint** - stops execution of the processes or threads that reach it.
- **Evaluation Point** - executes a code fragment when it is reached. Enables you to set "conditional breakpoints" and perform conditional execution.
- **Process Barrier Point** - holds each process when it reaches the barrier point until all processes in the group have reached the barrier point. Primarily for MPI programs.
- **Watchpoint** - monitors a location in memory and either stops execution or evaluates an expression when the value stored in memory is modified.

This section uses the **wave_extended** example to set a basic breakpoint as well as an evaluation point, called an "eval point."

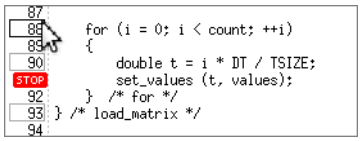
NOTE: These procedures on working with action points can be performed independently of the other sections in this chapter (which starts at Basic Debugging on page 32), but you must first load the program as described in Load the Program to Debug on page 33.

RELATED TOPICS	
Action points overview	About Action Points on page 189
Process barrier point	Setting Breakpoints and Barriers on page 194
Watchpoint	Using Watchpoints on page 231
Action Points Tab in the Process Window	Action Points Tab" in the in-product Help

Basic Breakpoints

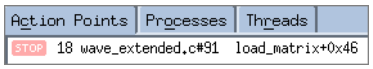
1. **Set a breakpoint**
   - Click a line number. for instance, select line 91, the call to **set_values()**. TotalView displays a STOP sign, both in the Source Pane at line 91 and in the Action Points tab where all action points in a program are listed.

NOTE: A breakpoint can be set if the line number is boxed in the Source Pane:



2. **Delete/disable/enable a breakpoint**

   - To delete the breakpoint, click the **Stop** icon in the Source Pane, and then re-add it by clicking again. You can also select it in the Action Points tab, right-click for a context menu, and select **Delete**.
   - To disable a breakpoint, click its icon in the Action Points tab. The icon dims to show it is disabled:

     Click it again to re-enable it. Again, you can also disable or re-enable a breakpoint using the context menu.

3. **Run the program**

   - Click the **Go** button in the toolbar.

     All panes in the Process Window report that the thread is running, or that it must be stopped for frame display. At this point, the program is waiting for user input.
   - Enter a number into the console, then click **Go** again.

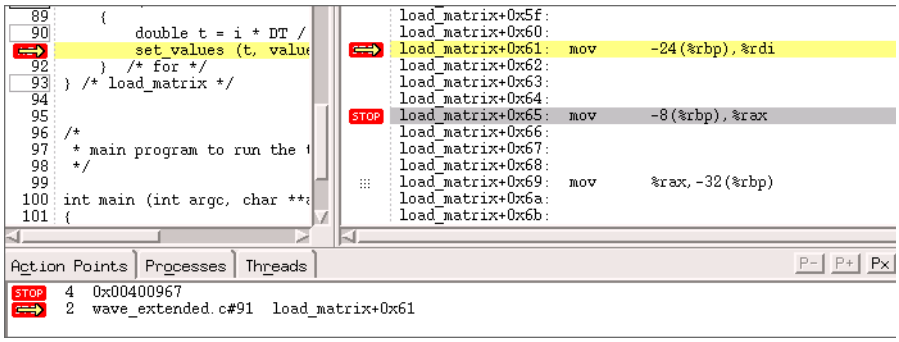
   The program halts execution at the breakpoint.

4. **Set a breakpoint in assembler code**

   You can also set a breakpoint in assembler code to view specific memory allocation.

   - Select **View > Source As > Both** to view both source and assembler code.
   - Set a breakpoint in some assembler code, such as the instruction immediately following the existing breakpoint.

     The Source Pane and Action Points tab display two breakpoints, one in source code and one in assembler code.

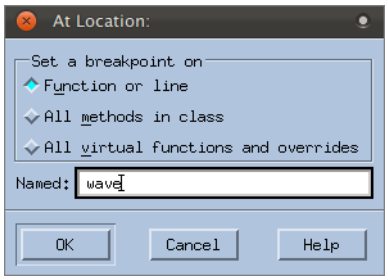


5. **Set a breakpoint at a particular location**

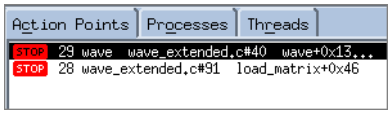
   In a complex program, it may be easier to set a breakpoint using the **At Location** dialog, which allows you to specify where you want the breakpoint without having to first find the source line in the Source Pane. Using this dialog, you can set breakpoints on all methods of a class or all virtual functions, a useful tool for C++ programs.

> **NOTE:** This dialog acts like a toggle, meaning that it *sets* a breakpoint if none exists at the specified location, or *deletes* an existing breakpoint at that location.

- Select **Action Point> At Location** and then enter **wave** to set a breakpoint at the function **wave()**.



The breakpoint is set and added to the Action Points tab. If a breakpoint already exists at that location, this action toggles the setting to *delete* the breakpoint.



RELATED TOPICS	
Action points properties	About Action Points on page 189 and "Action Point Properties" in the in-product Help.
Enabling/disabling action points	Displaying and Controlling Action Points on page 205
Suppressing action points	Suppressing Action Points on page 206
Breakpoints in assembler code	Setting Machine-Level Breakpoints on page 209

Evaluation Points

You can define an action point identified with a code fragment to be executed. This is called an *eval point*. This allows you to test potential fixes for your program, set the values of your program's variables, or stop a process based on some condition. You can also send data to the Visualizer to produce an animated display of changes to your program's data, discussed in Visualizing Arrays on page 59.

At each eval point, the code in the eval point is executed *before* the code on that line. One common use of an eval point is to include a **goto** statement that transfers control to a line number in your program, so you can test program patches.

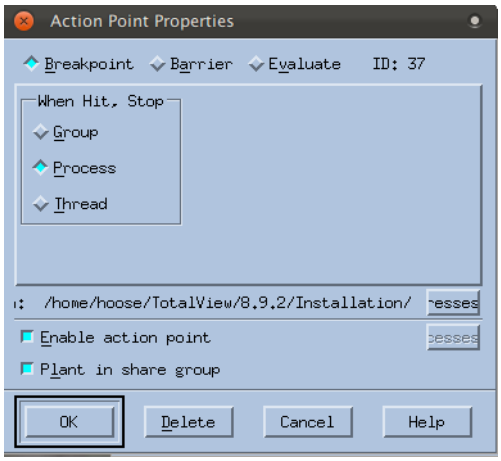
1. **Delete any breakpoints**

   Before setting an eval point, delete all other breakpoints you have set while working through this chapter.

   - Select **Action Points > Delete All**.

2. **Set an eval point**

   - Set a breakpoint on line 85 at the **printf()** function.
   - Open the **Action Point Properties** dialog by right-clicking on the **Stop** icon and selecting Properties.

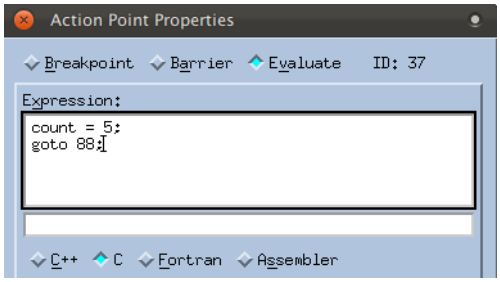


The **Action Point Properties** dialog box sets and controls an action point. Using this dialog, you can also change an action point's type to breakpoint, barrier point, or eval point, and define the behavior of threads and processes when execution reaches this action point.

3. **Add an expression**

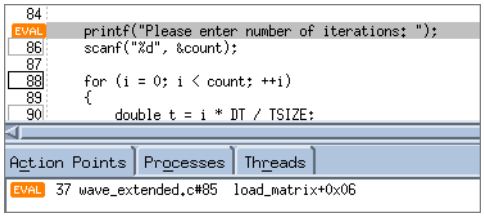
   - Select the **Evaluate** button, to display an Expression box. In the box, enter:

```
count = 5;
goto 88;
```



   - Click **OK** to close the dialog. This code will be executed before the **printf()** statement, and then will jump to line 88 where the for loop begins. This sets the count to **5** and avoids having to enter user input. (Code entered here is specific to TotalView debugging only, and is not persisted to your actual source code.)

Note that the **Stop** icon becomes an **Eval** icon, both in the Source Pane and in the Action Points tab:



4. **Execute the program to observe eval point behavior**

- Click **Go**. If no other breakpoints were planted in your code, the program simply executes and prints "Program is complete."

RELATED TOPICS	
Evaluation points in general	Defining Eval Points and Conditional Breakpoints on page 220
Writing expressions in eval points	Expressions in Eval Points and the Evaluate Window on page 363
Action Point Properties dialog box	About Action Points on page 189 and "Action Point Properties" in the in-product Help.

Saving and Reloading Action Points

You can save a set of action points to load into your program at a later time.

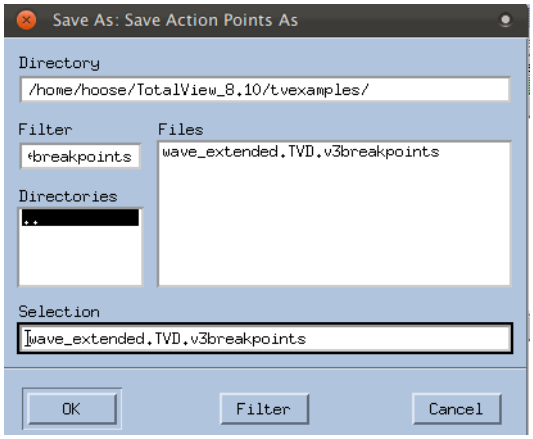
1. **Save Action Points**

   - Select **Action Point > Save All** to save your action points to a file in the same directory as your program. When you save action points, TotalView creates a file named *program_name*.**TVD.v4breakpoints**, where *program_name* is the name of your program.

     No dialog launches, but a file is created titled **wave_extended.TVD.v4breakpoints**.

   - Select **Action Point > Save As**, if you wish to name the file yourself and select a directory for its location.

     A dialog launches where you can enter a custom name and browse to a location.



2. **Load Saved Action Points**

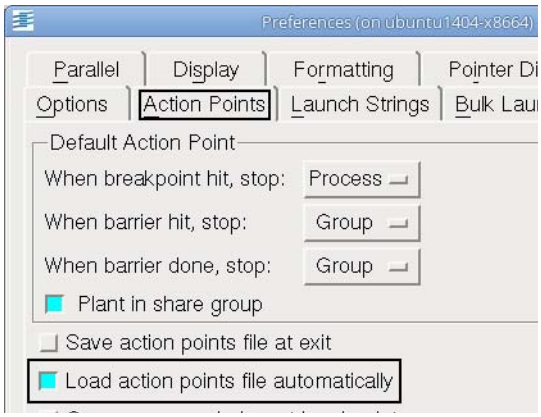
You can either explicitly load saved action points into a program when it is launched, or you can define a preference to load them automatically.

To manually load saved action points:

- After loading your program into TotalView, select **Action Point > Load All** to load these saved actions points into your program.

To automatically load saved action points:

- Select **File > Preferences** to launch the Preferences window, and then select the **Action Points** tab.
- Click the **Load Action Points File Automatically** button, then click OK.



If you close and then reload the program, your actions points are automatically loaded as well.

(Several other options exist to customize action points behavior. These are not discussed here. Please see the Related Topics table below.)

RELATED TOPICS

The CLI command **dbreak**	"dbreak" in the *Classic TotalView Reference Guide*
The Action Point Properties dialog box	About Action Points on page 189 and "Action Point Properties" in the in-product Help.
The **Action Point > At Location** command	Setting Breakpoints at Locations on page 201
Setting Action Points preferences	Setting Preferences on page 133 and "Action Points Page" in the in-product Help

Examining Data

Examining data is, of course, a primary focus of any debugging process. TotalView provides multiple tools to examine, display, and edit data.

This section discusses viewing built-in data in the Process Window and Expression List Window, and then using the Variable Window to look at compound data.

NOTE: These procedures on examining data can be performed independently of the tasks in other sections in this chapter, but you must first load the program (Load the Program to Debug on page 33). In addition, the discussion assumes an existing eval point has been set as described in Evaluation Points on page 45.

Viewing Built-in Data

For primitive, built-in types, you can quickly view data values from within the Process Window and can also add them to another window, the Expression List Window.

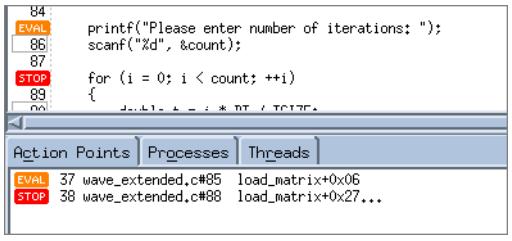
Viewing Variables in the Process Window

First, we'll add a breakpoint so the program will stop execution and we can view data.

1. **Set a breakpoint**

   - Add a breakpoint at line 88, the beginning of the for loop in the **load_matrix()** function.

   At this point, you should have two action points: the breakpoint just added, and the eval point added in the section Evaluation Points on page 45.



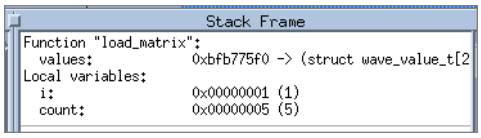
   - Click **Go**. The program should stop on the breakpoint you just added.

   Now let's view some data.

2. **View variables in the Stack Frame pane**

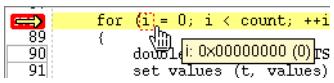
The Stack Frame pane lists function parameters, local variables, and registers. Scalar values are displayed directly, while aggregate types are identified with just type information.

In the Stack Frame pane, note the value of the local variables **i** and **count**: **i** is **1**, and **count** is **5**.



3. **View variables in a tool tip**

   - In the Source Pane, hover over the variable **i** to view a tool tip that displays its value:

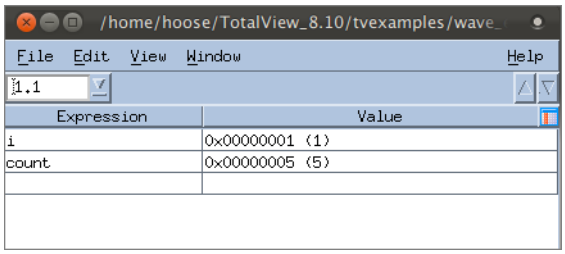


Viewing Variables in an Expression List Window

The Expression List window is a powerful tool that can list any variable in your program, along with its current or previous value and other information. This helps you to monitor variables as your program executes. For scalar variables, this is a particularly easy, compact way to view changing values.

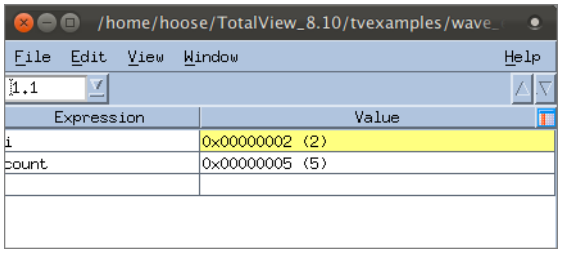
1. **Create an Expression List**

   - In the Stack Frame pane, right-click on the variable **i**, and select **Add to Expression List**. Then do the same for the variable **count**. The Expression List Window launches, displaying these two variables and their values.



2. **View the updated values**

   - Click **Go**. When the breakpoint is hit, the value of **i** increments to **2**, and this changed value is highlighted in yellow in the Expression List window:



- If you continue to hit **Go**, you can view the value of **i** increment to **5** before the program completes.

3. **Add additional columns to see more data**

   - Right-click on the column header and select **Type** and **Last Value**. These two columns are added to the table:

Expression	Type	Last Value	Value
i	int	0x00000001 (1)	0x00000002 (2)
count	int		0x00000005 (5)

RELATED TOPICS

Viewing variables in the Process Window	Displaying Variables on page 246
Viewing variables in the Expression List Window	Viewing a List of Variables on page 272

Viewing Compound Variables Using the Variable Window

For nonscalar variables, such as structures, classes, arrays, common blocks, or data types, you can dive on the variable to get more detail. This launches the Variable Window.

(For an overview on diving, see About Diving into Objects on page 164.)

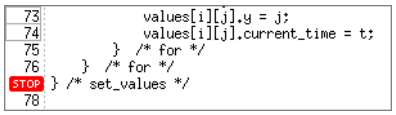
This section includes:

Basic Diving

First, delete any breakpoints you had entered previously except the eval point set in Evaluation Points on page 45. (Retaining this eval point simply allows the program to run without user input.)

1. **Set a breakpoint**

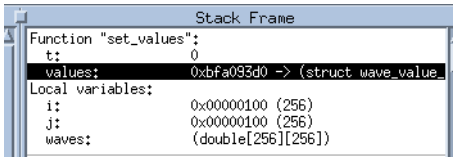
   - Add a breakpoint at line 77, at the completion of the **set_values()** method.



- Click **Go**. The program runs until the breakpoint.

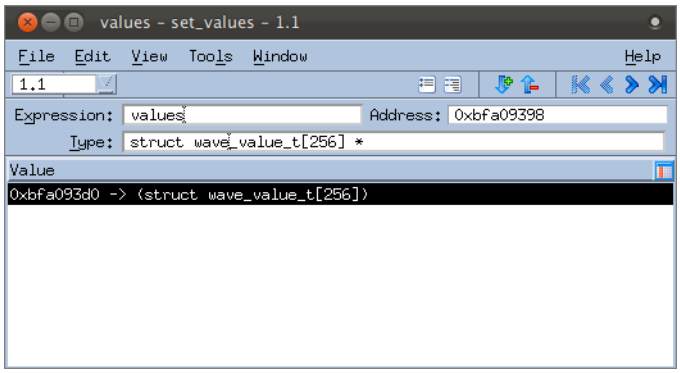
2. **Dive on a variable**

   - Dive on the variable **values** in the Stack Frame pane (by double-clicking on **values** or by right-clicking and selecting **Dive**).



The **values** variable is a struct of type **wave_value_t**, created to hold a copy of the variables that create the wave, as well as other data.

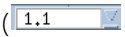
The Variable Window launches.



Elements of a Variable Window

The basic elements of the Variable Window include:

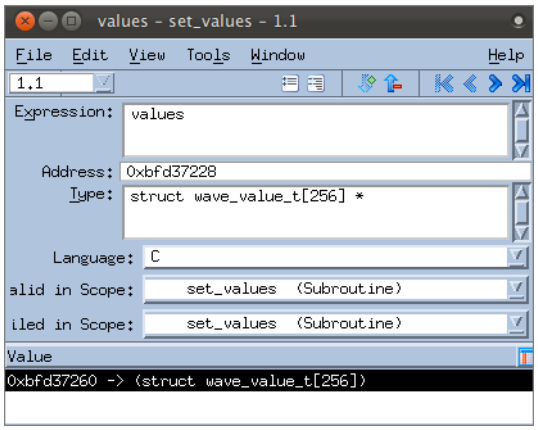
- A set of toolbar icons that provide navigation and customizations:

  Thread ID (

) icon to identify the current thread (in a single-threaded program, this is always 1.1, meaning process 1, thread 1).

  Collapse/expand () icons to expand or collapse the contents of a compound type in nested windows.

  Up/down () icons to control the level of information about your data. If you select the up arrow, more information about your data is displayed.



Redive/Undive buttons, discussed in Rediving and Undiving on page 54.

- The editable fields **Expression**, **Address**, and **Type**. You can add an expression or change the address and type for your variable here. Then select **Edit > Reset Defaults** when you are finished. (This is beyond the scope of this chapter. See the Related Topics table for more information.)

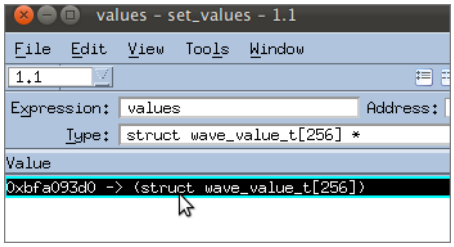
RELATED TOPICS	
Diving on objects	About Diving into Objects on page 164 and
The **View > Dive** command	View > Dive" in the Process Window in the in-product Help
More on the Variable Window	Diving in Variable Windows on page 266
Editing data in the Variable Window	Changing What the Variable Window Displays on page 270

Nested Dives

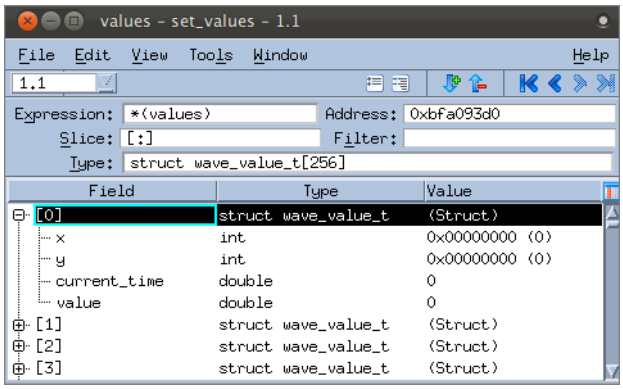
Since **values** is a compound type, you can dive again to get more detail about its components. This is called a *nested dive*.

1. **Dive on an array**

   - Dive on the array **wave_value_t** in the Value column, by double-clicking it:

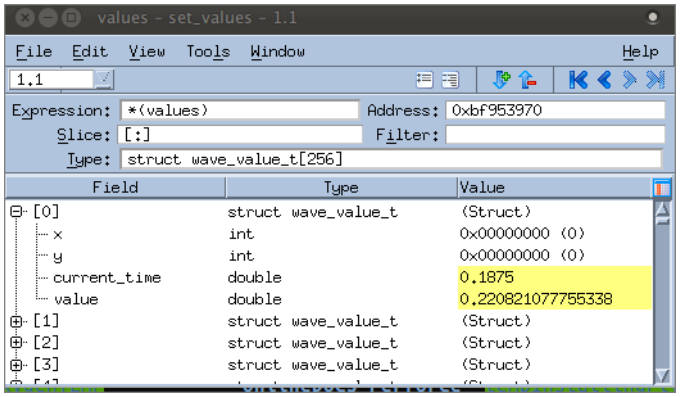


This replaces the existing display with the array's contents. Note the **+**(plus) sign on the left side of the Field column. For nonscalar types, you can click the plus sign to see the type's components:



2. **Run the program and observe changing variable values**

   - Click **Go** so that the program runs and again stops at the breakpoint. Note that the variables **current_time** and **value** have both changed:

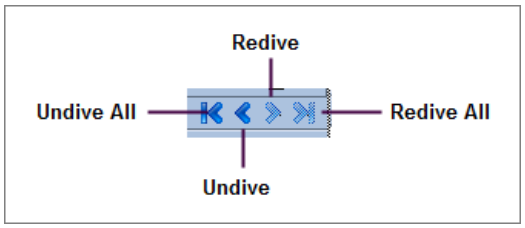


RELATED TOPICS

More on nested dives	Diving in Variable Windows on page 266

Rediving and Undiving

Note the arrow icons on the top right of the Variable Window. These are the **Undive/Redive** and **Undive all/ Redive** all buttons. Using these buttons, you can navigate up and down into your dive.



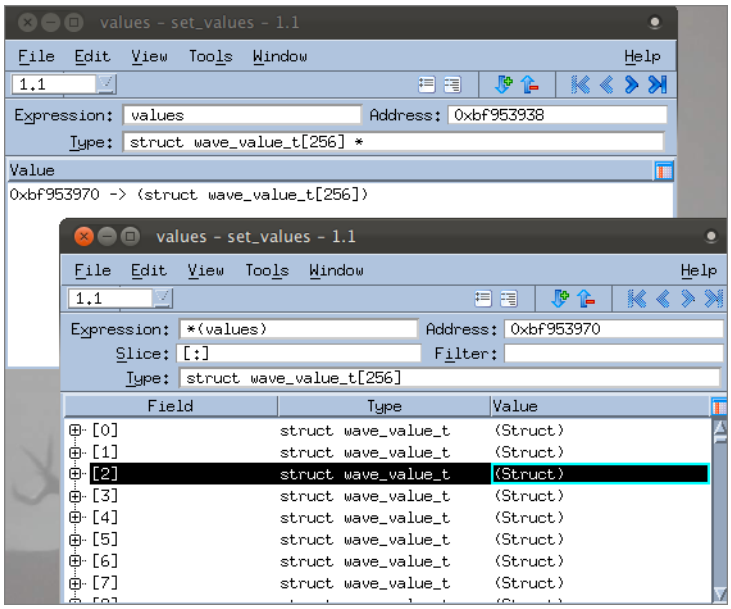
Click the Undive button, for instance, to return to the previous window.

Diving in a New Window

If you wish to have more than one dive level visible at the same time rather than having a dive replace the existing window's data, you can create a duplicate window.

1. **Undive**
   - Click the **Undive** arrow to return to the initial Variable Window.
2. **Launch a new window**
   - Right-click on the array and select **Dive in New Window**. Another window launches. Now, you can see both the original window and the new, nested dive.
3. **Duplicate a window**

   Alternatively you can create a duplicate of a window.
   - Select the command **Window > Duplicate** to duplicate the active Variable Window and then dive to the desired level in a new window.



Displaying an Element in an Array of Structures

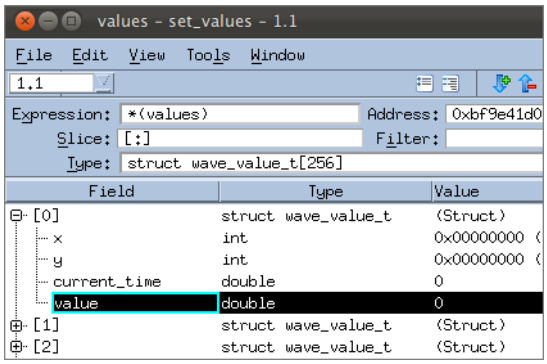
You can display an element in an array of structures as if it were a simple array using the **View > Dive In All** command.

Consider our **values** struct defined like this:

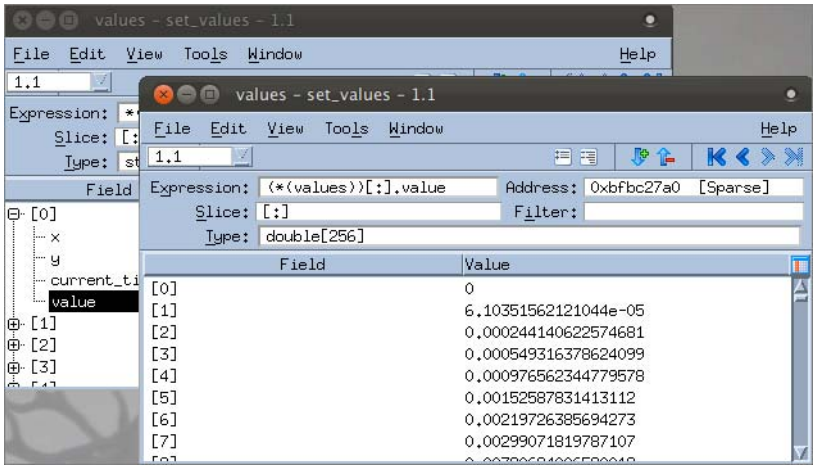
```
struct wave_value_t
{
    int x;
    int y;
    double current_time;
    double value;
};
```

We can select an element within that structure and view it as an array, allowing us to easily see the values of any individual element as they change throughout program execution.

1. **Dive in All on a variable**
    - In the nested Variable Window, select the double **value**.



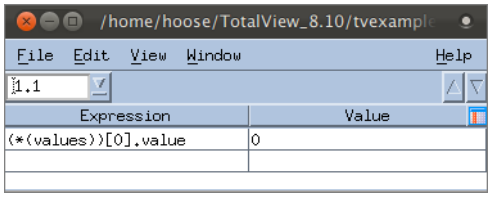
- Right-click and select **Dive In All**. TotalView displays all of the **value** elements of the **values** array as if they were a single array.



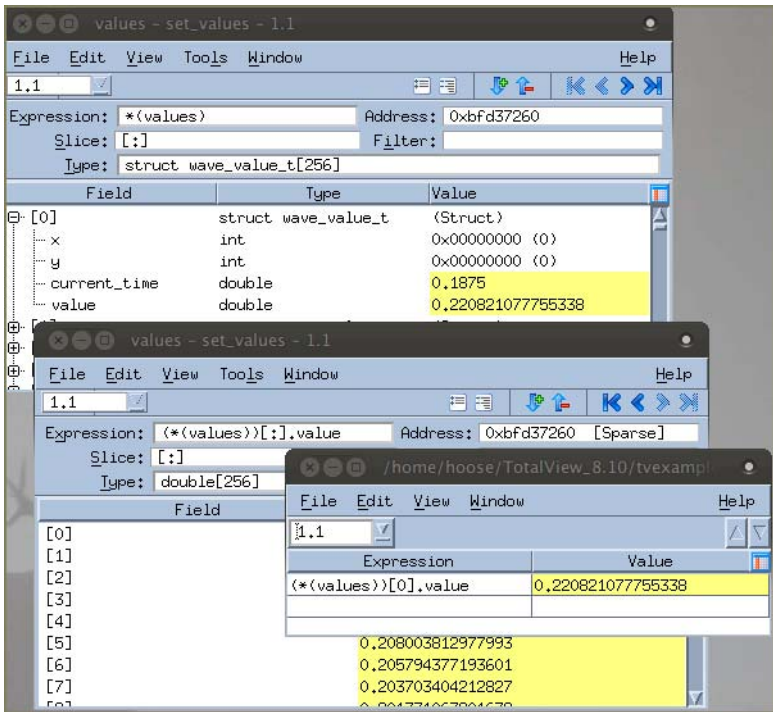
2. **Add the value to the Expression List**

   Remember that you can also view the scalar type value in an Expression List window.

   - In the window just launched, right-click again on **value** and select **Add to Expression List**. The Expression List window launches listing **value**:



3. Click **Go** to run your program. You can now view your variable values changing in three windows:



RELATED TOPICS

Displaying an array of any element	Displaying an Array of Structure's Elements on page 268
More on the **View > Dive in All** command	View > Dive in All" in the in-product Help

Visualizing Arrays

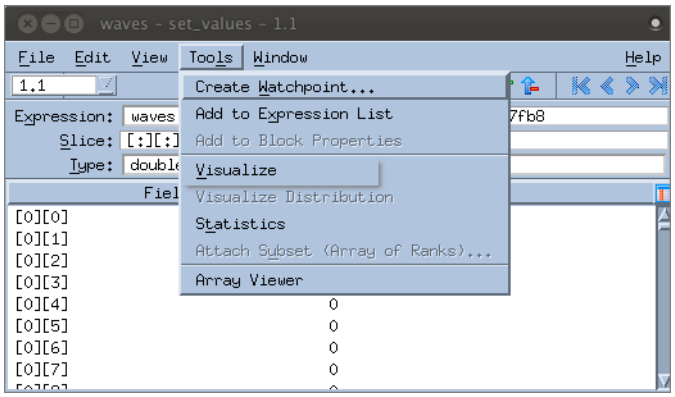
The Visualizer is a special tool to graphically visualize array data. This is a powerful and easy way to view your program's data during debugging and is useful in discovering anomalies in data value range, numerical spikes, and NaNs.

NOTE: These procedures on visualizing arrays can be performed independently of the other tasks discussed in this chapter, but you must first load the program (Load the Program to Debug on page 33). In addition, the discussion assumes an existing eval point has been set, as described in Evaluation Points on page 45.

You can launch the Visualizer either directly from the GUI or from within an eval point.

From the GUI

Select an array in a Variable Window, and then select **Tools > Visualizer**.



From within an eval point

Invoke the Visualizer using the **$visualize** command, with this syntax:

```
$visualize ( array [, slice_string ])
```

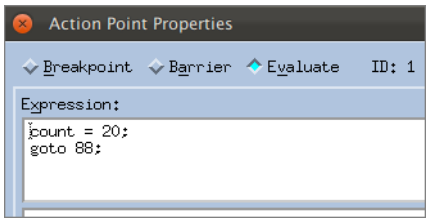
Launching from an eval point also provides the ability to stop program execution if desired.

This discussion uses the **$visualize** command in an eval point to launch the Visualizer to view the **waves** array. This array increments the value of **XSIZE** and **YSIZE** to create a visual wave.

Set Up

- Delete any breakpoints previously set except the eval point set at line 85 to suppress user input (Evaluation Points on page 45).

- Edit that eval point to provide a higher count, for instance, **20**. This will allow amore interesting wave to build as the values are incremented. (Right-click on its Eval icon, select Properties, and then edit the **count** value:

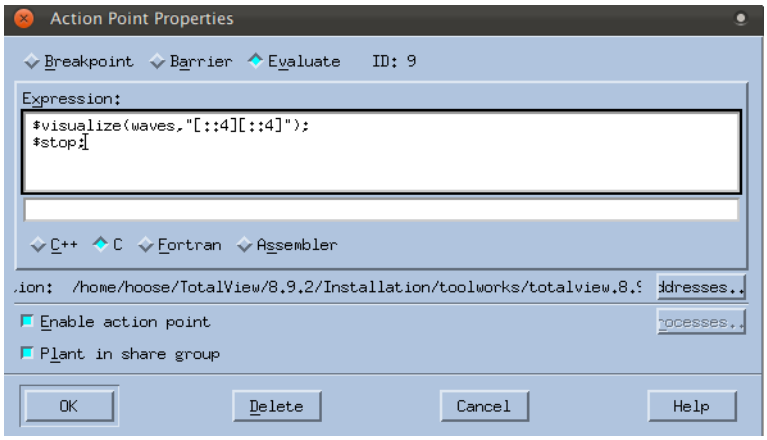


Launching the Visualizer from an Eval Point

1. **Add an eval point**

   - Click on line 63 at the function **wave()** to add a breakpoint, as the comments suggest.
   - Right-click on the breakpoint icon and select Properties to launch the **Action Point Properties** dialog.
   - Click the **Evaluate** button to open the Expression field. Enter the following code:

```
$visualize(waves,"[::4][::4]");
$stop;
```

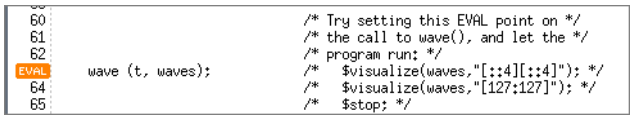


When the eval point is hit, this code will launch a single Visualizer window that will display every fourth element in the major dimension (the X axis), and then program execution will stop. To display the entire array, you could just write:

```
$visualize(waves);   // entire array
```

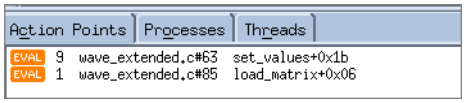
(Note that the code comments suggest launching two Visualizer windows. For purposes of this discussion, we'll add just one.)

- Click **OK**. The Eval Point icon appears:



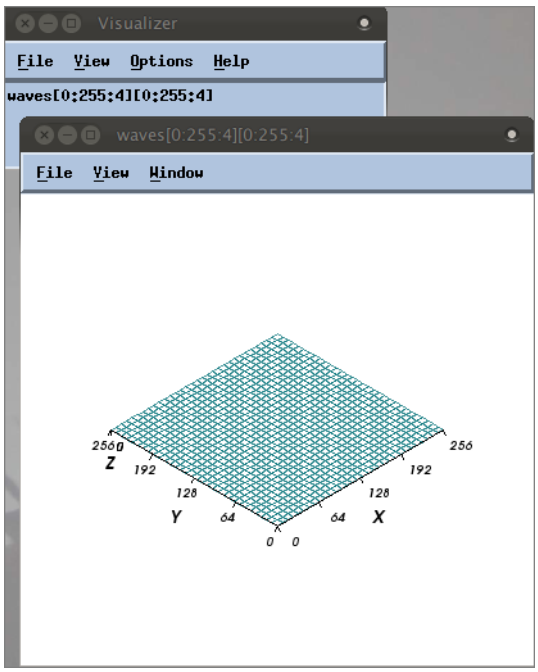
2. Run the program and view the data

At this point, the program should have no regular breakpoints and two eval points:



- Click **Go**.

  The program runs to the eval point at **wave()** and then stops. The Visualizer launches, reflecting the array's initial values:

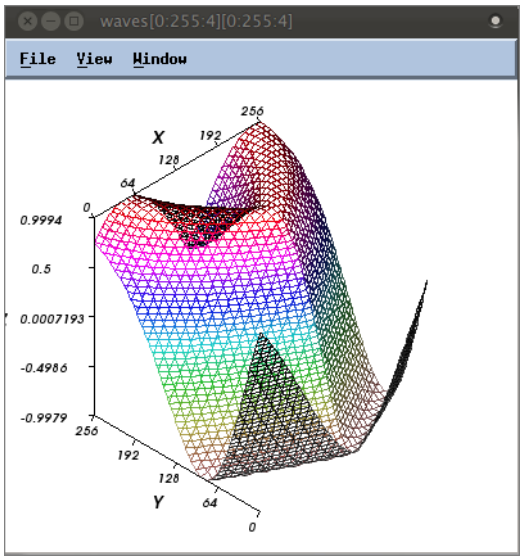


Above are the Visualizer's two windows: the top window is the Dataset window listing all datasets available to visualize (only one dataset has been loaded into the Visualizer at this point); the bottom is the View window where the graphical display appears.

3. Complete the program

- Click **Go** several more times (the program will complete once you have clicked **Go** as many times as the value for the variable **count** in the eval point).

  You can watch the wave build, for example:



4. **Run the program without stopping execution**

   An eval point does not have to stop execution. To let the program run without interruption, just remove the **$stop** command from the Expression field in the **Action Point Properties** dialog, then click **Go**.

RELATED TOPICS	
The Array Visualizer	Array Visualizer on page 341
More ways to use view arrays	Examining Arrays on page 312

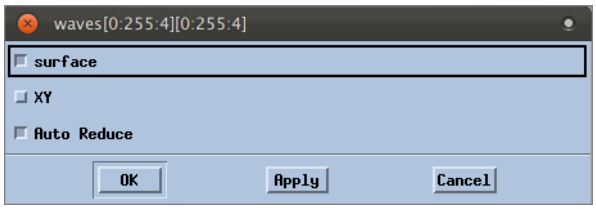
Viewing Options

The Visualizer shows either a Graph view (2-D plot) or Surface view (3-D plot) of your data. (If the array is one-dimensional, only the Graph view is available. The Graph view is not discussed here.)

By default, it shows a Surface view for most two-dimensional data, and that is what it shows for the **waves** array.

The Surface view 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, and the values map to the height (Z axis).

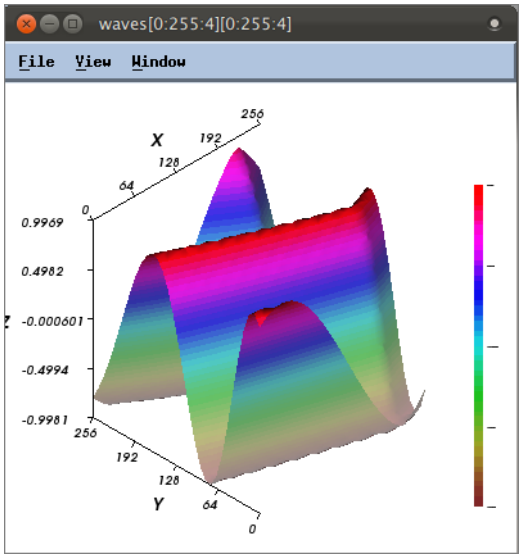
You can further refine the Surface view using the View window's options. Select **File > Option**s from the View window to launch the Options dialog.



Possible options are:

- **surface:** Displays the array as a three-dimensional surface (the default is to display it as a grid).
- **XY:** Reorients the view's XY axes. The Z axis is perpendicular to the display.
- **Auto Reduce:** Speeds visualization by averaging neighboring elements in the original dataset.

For example, click **surface**, then click OK. The view changes from a grid to a 3-D:



The Visualizer has many more options with various other viewing modes and tools. See the Related Topics below for references to further discussions.

This completes this tutorial on basic debugging.

RELATED TOPICS

More on the Visualizer	Array Visualizer on page 341
Visualizer options	Using the Graph Window on page 347 and Using the Surface Window on page 350
The Array Viewer (another way of looking at arrays)	Viewing Array Data on page 317
Displaying slices of arrays	Displaying Array Slices on page 313
Filtering array data	Filtering Array Data Overview on page 319

Moving On

- For an overview on TotalView's features, see About TotalView on page 5.
- To learn about parallel debugging tasks, see Manipulating Processes and Threads on page 407.
- For detailed information on TotalView's debugging tools and features, see Debugging Tools and Tasks on page 84.

Accessing TotalView Remotely

Using the TotalView Remote Display client, you can start and then view TotalView as it executes on another system, so that TotalView need not be installed on your local machine.

Remote Display Supported Platforms

Remote Display is currently bundled into all TotalView releases.

Supported platforms include:

- Linux x86-64
- Microsoft Windows
- Apple macOS Intel

No license is needed to run the Client, but TotalView running on any supported operating system must be a licensed version of TotalView 8.6 or greater.

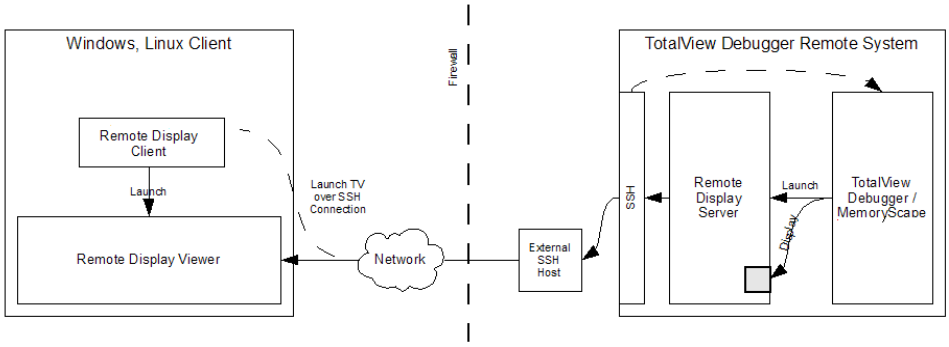
Remote Display Components

TotalView Remote Display has three components:

- The Client is a window running on a Remote Display supported platform (See Remote Display Supported Platforms on page 66).
- The Server is invisible, managing the movement of information between the Viewer, the remote host, and the Client. The Server can run on all systems that TotalView supports. For example, you can run the Client on a Windows system and set up a Viewer environment on an IBM RS/6000 machine.
- The Viewer is a window that appears on the Client system. All interactions between this window and the system running TotalView are handled by the Server.

Figure 16 shows how these components interact.

Figure 16, Remote Display Components



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

TotalView and the X Window system must be installed on the remote server machine containing the rgb and font files in order for the remote display server to start correctly. The bastion nodes (if any) between the remote client machine and remote server machine do not require TotalView or X Window file access.

Installing the Client

Before installing the Client, TotalView must already be installed.

The files used to install the client are in these locations:

- Remote Display Client files for each supported platform are in the **remote_display** subdirectory in your TotalView installation directory.
- Alternatively, request a Remote Display Client from TotalView's download page at https://totalview.io/downloads.

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

Installing on Linux

The Linux Client installer is **RDC_installer_<*release_number*>-linux-x86-64.run**.

Linux Requirements:

- The xterm application must be installed on both the RDC client and RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.

Installing on Microsoft Windows

Before installing the Client, TotalView must already be installed on your Linux or UNIX system. The Windows Client installer is **RDC_Installer.<*release_number*>-win.exe**.

Windows Requirements:

- The xterm application must be installed on the RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.

Installing on macOS

The macOS installer is **RDC_installer_<*release_number*>-macos.dmg**.

macOS Requirements:

- The xterm application must be installed on both the RDC client and RDC server host.
- The RDC server host must have a window manager installed. The RDC looks for **icewm**, **fvwm**, **twm** and **mwm**. You can override the window manager in use by providing the executable name of your window manager on the RDC's Advanced Options dialog.
- Catalina (macOS 10.14) and newer users must install the free Real VNC viewer from https://www.realvnc.com/en/connect/download/viewer/macos/.

Client Session Basics

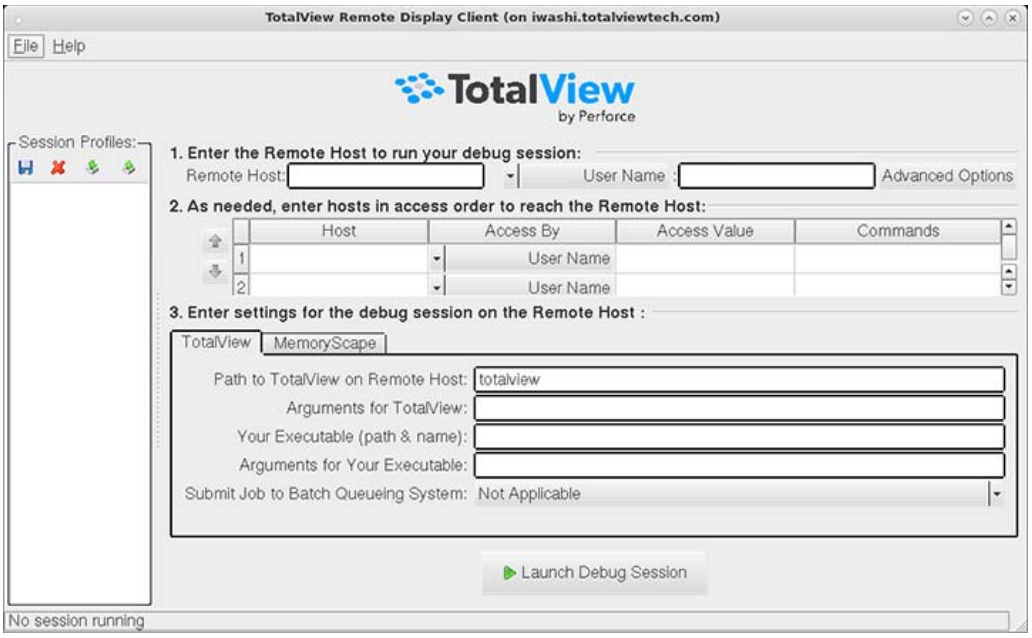
The TotalView Remote Display Client is simple to use. Just enter the required information, and the Client does the rest.

On Linux, invoke the Client with the following:

```
remote_display_client.sh
```

On Windows, either click the desktop icon or use the TVT Remote Display item in the start menu to launch the remote display dialog. On macOS, run the TVRemoteDisplayClient application from the Applications area.

Figure 17, Remote Display Client Window



The Client window displays similarly on Linux, Windows, or macOS.

Here are the basic steps:

1. **Enter the Remote Host**

    - **Remote Host**: The name of the machine upon which TotalView will execute. While the Client can execute only on specified systems (see Remote Display Supported Platforms), the remote system can be any system upon which you are licensed to run TotalView.
    - **User Name** dropdown: Your user name, a public key file, or other **ssh** options.

2. **(Optional) As needed, enter hosts in access order...**(depending on your network).

   If the Client system cannot directly access the remote host, specify the path. For more information, see Naming Intermediate Hosts on page 76.

3. **Enter settings for the debug session on the Remote Host**

   Settings required to start TotalView on the remote host. (The TotalView and MemoryScape tabs are identical.)

   - **Path to TotalView on the Remote Host**: The directory on the remote host in which TotalView resides, using either an absolute or relative path. "Relative" means relative to your home directory.
   - **(Optional) Your Executable**: Either a complete or relative pathname to the program being debugged. If you leave this empty, TotalView begins executing as if you had just typed **totalview** on the remote host.
   - **Other options**:

     You can add any command-line options for TotalView or your program.

     TotalView options are described in the "T*otalView Debugger Command Syntax*" chapter of the *Classic TotalView Reference Guide*.

     For arguments to your program, enter them in the same way as you would using the **-a** command-line option.

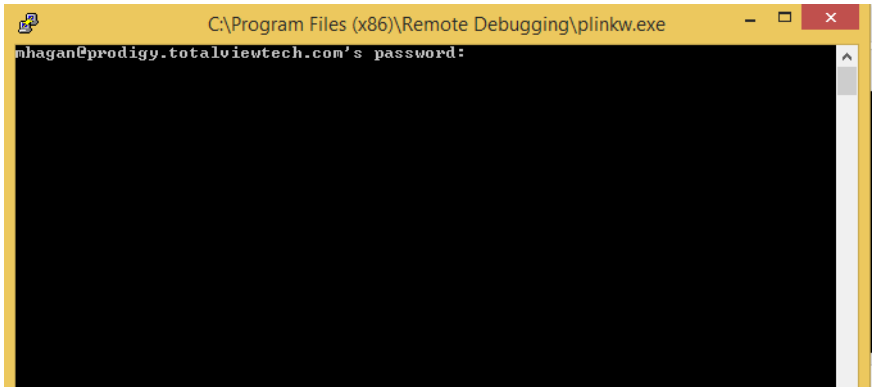
Additional options include:

- **Advanced Options:** Press the Advanced Options button to customize client/server interaction and server execution, Advanced Options on page 74.
- **Submit job to batch queueing system:** You can submit jobs to the PBS Pro and LoadLeveler batch queuing systems, Submitting a Job to a Batch Queuing System on page 77.

Launching the Remote Session

Next, press the **Launch Debug Session** button, which launches a password dialog box.

Figure 18, Asking for Password



Depending on how you have connected, you may be prompted twice for your password: first when Remote Display is searching ports on a remote system and another when accessing the remote host. You can often simplify logging in by using a public key file.

After entering the remote host password, a window opens on the local Client system containing TotalView as well as an xterm running on the remote host where you can enter operating system and other commands. If you do not add an executable name, TotalView displays its **File > New Debugging Session** dialog box. If you do enter a name, TotalView displays its **Process > Startup Parameters** dialog box.

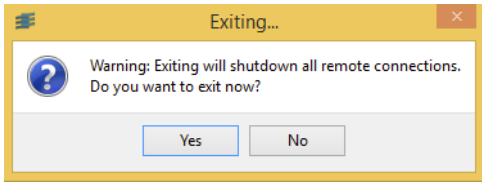
Closing the Remote Session

To close the session:

- From the Client, terminate the Viewer and Server by pressing the **End Debug Session** button. (The **Launch Debug Session** button changes to this button after you launch the session.)
- Click **Close** on the Viewer's window to remove the Viewer Window. This does not end the debugging session, so then select the Client's **End Debug Session** button. Using these two steps to end the session may be useful when many windows are running on your desktop, and the Viewer has obscured the Client.

Closing all Remote Sessions and the Client

To close all remote connections and shut down the Client window, select **File > Exit**.



Working on the Remote Host

After launching a remote session, the Client starts the Remote Display Server on the remote host where it creates a virtual window. The Server then sends the virtual window to the Viewer window running on your system. The Viewer is just another window running on the Client's system. You can interact with the Viewer window in the same way you interact with any window that runs directly on your system.

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

If you are running the Client on a Windows system, these are the icons available:

Figure 19, Remote Display Client commands on Windows

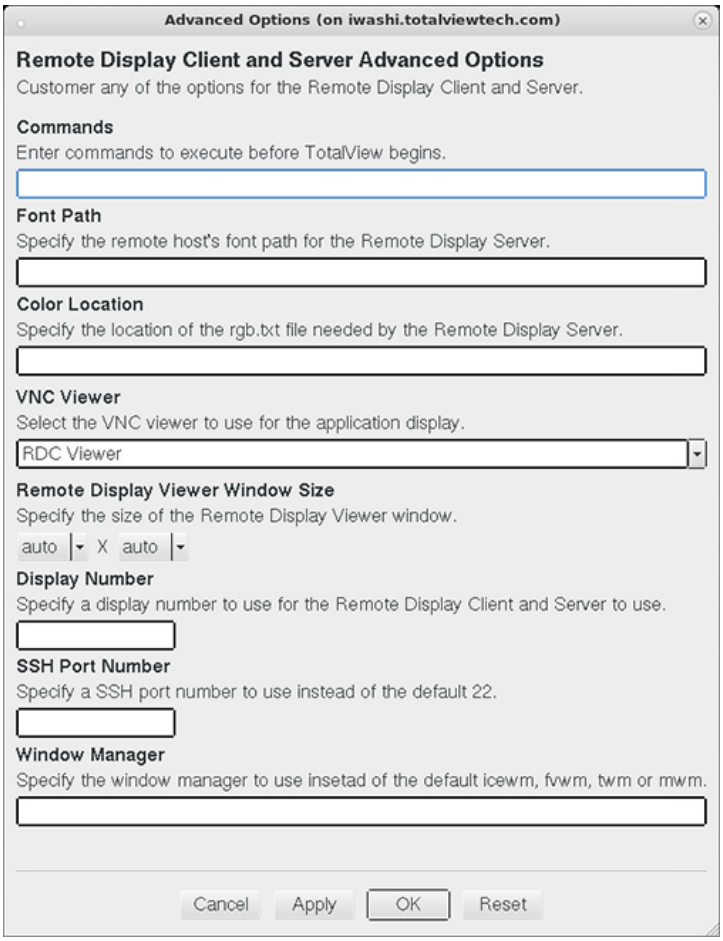
From left to right, the commands associated with these icons are:

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

Advanced Options

The Advanced Options window in Figure 20 is used to customize Remote Display Client and Server interaction and to direct the Server and Remote Display Viewer execution.

Figure 20, Advanced Options Window



Options are:

- **Commands**: Enter commands to execute before TotalView begins. For example, you can set an environment variable or change a directory location.
- **Font Path:** Specify the remote host's font path, needed by the Remote Display Server. Remote Display checks the obvious places for the font path, but on some architectures, the paths are not obvious.

- **Color Location:** Specify the location of the **rgb.txt** file needed by the Remote Display Server. Remote Display checks the obvious places for the location, but on some architectures, its location is not obvious. Providing the correct location may improve the startup time.
- **VNC Viewer:** Select the VNC viewer to use for application display.
- **Remote Display Viewer Window Size:** The default size of the Remote Display Viewer is dynamically computed, taking into account the size of the device on which the Remote Display Client is running. You can override this by selecting a custom size, which will be saved with the profile.
- **Display Number:** Specify a display number for Remote Display to use when the Client and Server connect. The Remote Display Client determines a free display number when connecting to the Server, requiring two password entries in some instances. Specifying the display number overrides the Remote Display Client determining a free number, and collisions may occur.
- **ssh Port Number:** On most systems, **ssh** uses port 22 when connecting, but in rare instances another port is used. This field allows you to override the default.
- **Window Manager:** Specify the name of the window manager. The path of the window manager you provide must be named in your **PATH** environment variable. The Server looks for (in order) the following window managers on the remote host: **icewm, fvwm, twm,** and **mwm**. Specifying a window manager may improve the startup time.

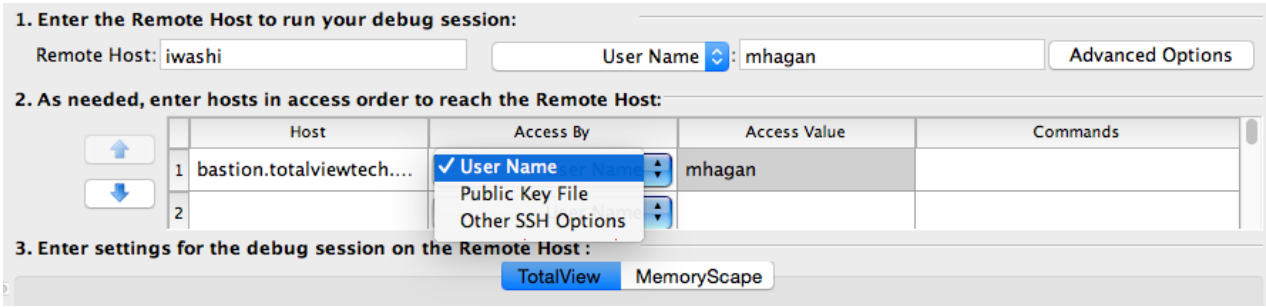
The buttons at the bottom are:

- **Cancel:** Closes the window without saving changes.
- **Apply:** Saves the changes with the profile, leaving the window open.
- **OK:** Closes the window and saves the changes with the profile.
- **Reset:** Reverts back to the previously saved values.

Naming Intermediate Hosts

If the Client system does not have direct access to the remote host, you must specify the path, or paths, along with how you will access the host. You can enter multiple hosts; the order in which you enter them determines the order Remote Display uses to reach your remote host. Use the arrow buttons on the left ( ) to change the order.

Figure 21, Access By Options

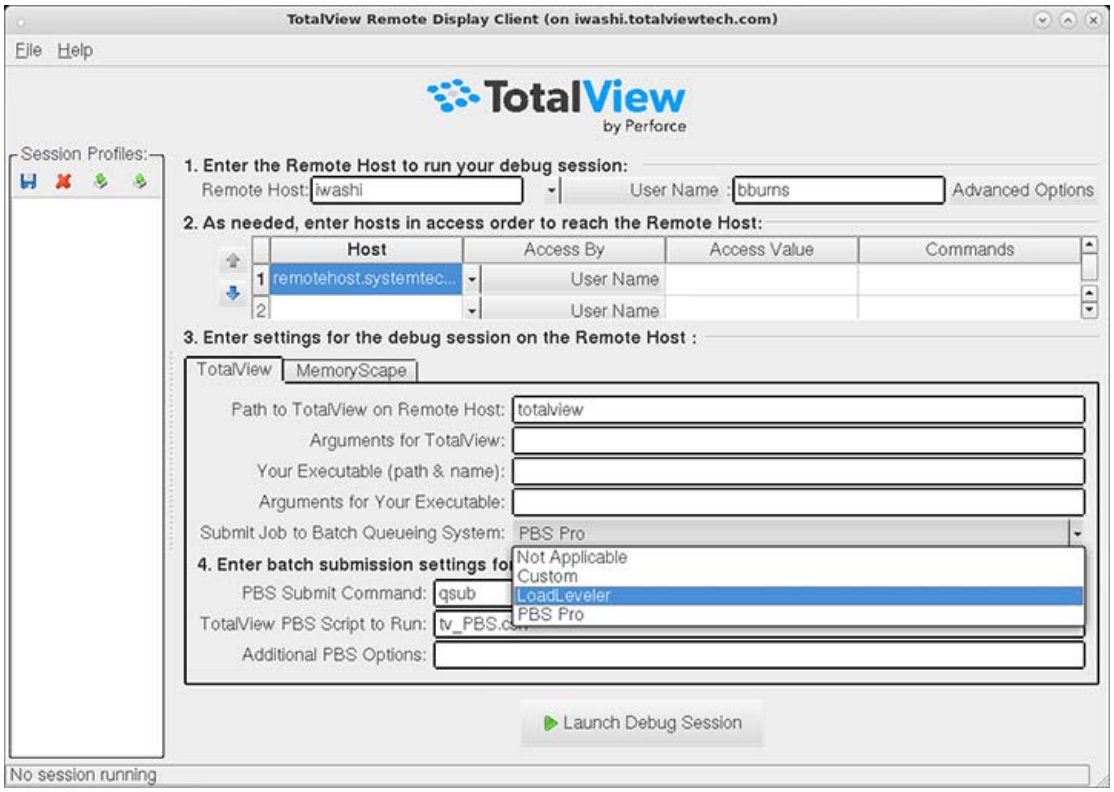


- **Host:** The route the Client should take to access the remote host. For instance, this can be a network path or an IP address. If your network has a gateway machine, you would name it here in addition to other systems in the path to the remote host.
- **Access By/Access Value:** The most common access method is by a user name, the default. If this is incorrect for your environment, use the dropdown menu to select the correct method:
  - **User Name**, i.e. the name you enter into a shell command such as ssh to log in to the host machine. Enter this in the **Access Value** field.
  - **Public Key File**, the file that contains access information, entered into the **Access Value** field.
  - **Other SSH Options**, the ssh arguments needed to access the intermediate host. These are the same arguments you normally add to the ssh command.
- **Commands:** Commands (in a comma-separated list) to execute when connected to the remote host, before connecting to the next host.

Submitting a Job to a Batch Queuing System

TotalView Remote Display can submit jobs to the PBS Pro and LoadLeveler batch queuing systems.

Figure 22, Remote Display Window: Showing Batch Options



1. Select a batch system from the **Submit job to Batch Queuing System** dropdown list, either **PBS Pro** or **LoadLeveler**.

   The default values are **qsub** for PBS Pro and **llsubmit** for LoadLeveler.

   The **Script to Run** field is populated with the default scripts for either system: **tv_PBS.csh** for PBS Pro and **tv_LoadLeveler.csh** for LoadLeveler. These scripts were installed with TotalView, but can of course be changed if your system requires it. For more information, see Batch Scripts on page 82.

2. (Optional) Select additional PBS or LoadLeveler options in the **Additional Options** field.

Any other required command-line options to either PBS or LoadLeveler. Options entered override those in the batch script.

3. Launch by pressing the **Launch Debug Session** button.

   Behind the scenes, a job is submitted that will launch the Server and the Viewer when it reaches the head of the batch queue.

Setting Up Your Systems and Security

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

As Remote Display connects to systems, a password is required. If you are allowed to use keyless ssh, you can simplify the connection process. Check with your system administrator to confirm that this kind of connection is allowed and the ssh documentation for how to generate and store key information.

Requirements for the Client to connect to the remote host:

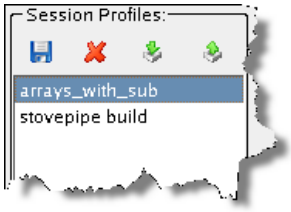
- If you use an LM_LICENSE_FILE environment variable to identify where your license is located, ensure that this variable is read in on the remote host. This is performed automatically if the variable's definition is contained within one of the files read by the shell when Remote Display logs in.
- ssh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

Session Profile Management

The Client saves your information into a profile based on the name entered in the remote host area. You can restore these settings by clicking on the profile's name in the Session Profiles area.

Figure 23 shows two saved profiles.

Figure 23, Session Profiles

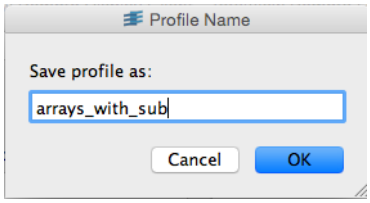


When you select a profile, the Client populates the right window with that profile's values.

If you edit the data in a text field, the Client automatically updates the profile information. If this is not what you want, click the **Create** icon to display a dialog box into which you can enter a new session profile name. The Client writes this existing data into a new profile instead of saving it to the original profile.

Saving a Profile

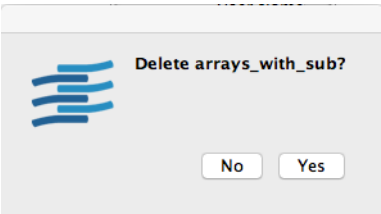
To save a profile, click the save button ( ) or select **File > Profile > Save**, then provide a profile name in the Profile Name popup.



This command saves the profile information currently displayed in the Client window to a name you provide, placing it in the **Session Profiles** area. You do not need to save changes to the current profile as the Client automatically saves them.

Deleting a Profile

To delete a profile, click the delete button ( ) or select **File > Profile > Delete**. This command deletes the currently selected profile and requires a confirmation.



Sharing Profiles

To import a profile, click the import button ( ) or select **File > Profile > Import**, and then browse to the profile to import. After you import a file, it remains in your Client profile until you delete it.

To export a profile, click the export button ( ) or select **File > Profile > Export**, browse to a directory where you want to export it, and then name the profile.

Batch Scripts

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

Place the script you create into *installation_dir*/*totalview_version*/**batch**. For example, you could place a new script file called **Run_Large.csh** into the *installation_dir*/**toolworks/totalview.8.6.0/batch** directory.

tv_PBS.csh Script

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

```
#!/bin/csh -f
#
# Script to submit using PBS
#
# These are passed to batch scheduler::
#
# account to be charged
##PBS -A VEN012
#
# pass users environment to the job
##PBS -V
#
# name of the job
#PBS -N TotalView
#
# input and output are combined to standard
##PBS -o PBSPro_out.txt
##PBS -e PBSPro_err.txt
#
##PBS -l feature=xt3
#
#PBS -l walltime=1:00:00,nodes=2:ppn=1
#
#
# Do not remove the following:
TV_COMMAND
exit
#
# end of execution script
```

```
#
```

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

```
TV_COMMAND
exit
```

tv_LoadLeveler.csh Script

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

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

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

```
TV_COMMAND
```

PART II

Debugging Tools and Tasks

This part introduces basic tools and features for debugging your programs using TotalView, including:

- Starting TotalView

  If you just enter `totalview` in a shell, the Sessions Manager launches where you can configure your debugging session. But you can also bypass the manager and launch TotalView directly. This chapter details the multiple options you have for starting TotalView.

- Loading and Managing Sessions

  You can set up a debugging session in several ways, depending on your platform. This chapter discusses common setup scenarios and configurations.

- Using and Customizing the GUI

  The TotalView GUI provides an extensive set of tools for viewing, navigating, and customization. This chapter discusses features specific to TotalView's interface.

- Stepping through and Executing your Program

  TotalView provides a wide set of tools for stepping through your program, using either the Process and Group menus, toolbar commands, or the CLI.

- Setting Action Points

  Action points 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.

- Examining and Editing Data and Program Elements

  This chapter discusses how to examine the value stored in a variable.

- Examining Arrays

Displaying information in arrays presents special problems. This chapter tells how TotalView solves these problems.

- Visualizing Programs and Data

  Some TotalView commands and tools are only useful if you're using the GUI. Here you will find information on the Call Graph and Visualizer.

- Evaluating Expressions

  Many TotalView operations such as displaying variables are actually operating upon expressions. Here's where you'll find details of what TotalView does. This information is not just for advanced users.

- About Groups, Processes, and Threads

  This chapter is the first of a three-chapter look at the TotalView process/thread model and how to manipulate threads and processes while debugging your multi-threaded applications. This chapter contains concept information on threads and processes in general. Manipulating Processes and Threads describes TotalView's hands-on tools for organizing and viewing thread and process activity and data, while Group, Process, and Thread Control includes advanced configuration and customization, useful for finely controlling execution in very complex applications.

- Manipulating Processes and Threads

  The second (of three) chapter focusing on threads and processes, with an emphasis on hands-on tasks and tools to control the view, execution, and focus of a single or group of threads and processes.

- Debugging Strategies for Parallel Applications

  Because debugging parallel applications can be so complex, this chapter offers a few strategies that can help streamline the task.

Starting TotalView

Before starting TotalView and loading a program to debug, first compile your program for debugging.

When you are ready to start debugging, you have many options for starting TotalView.

This chapter discusses:

Compiling Programs

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

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

You can also debug programs that you did not compile using the **-g** option, or programs for which you do not have source code. For more information, see

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

Compiler Option or Library	What It Does	When to Use It
Debugging symbols option (usually **-g**)	Generates debugging information in the symbol table.	Before debugging *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** option and the **-g** option at the same time. Even if your compiler lets you use the **-O** option, don't use it when debugging your program, since strange results often occur.	After you finish debugging your program.
multi-process programming library (usually **dbfork**)	Linking with **dbfork** defines TotalView-specific symbols that direct the debugger to follow **fork()** and **vfork()** system calls. In some cases, you need to use the -**lpthread** option. For more information about dbfork, see "*Linking with the dbfork Library*" contained in the "***Compilers and Platforms***" chapter of the *Classic TotalView Reference Guide*.	Before debugging a multi-process program that explicitly calls **fork()** or **vfork()** and for which you want TotalView to always attach to the child processes. See Debugging Processes That Call the fork() Function on page 213.

RELATED TOPICS

Compilers and platforms	"Compilers and Platforms" in the *Classic TotalView Reference Guide*
The dbfork library	"Linking with the dbfork Library" in the *Classic TotalView Reference Guide*

RELATED TOPICS	
Controlling TotalView's behavior for fork, vfork and execve handling	Controlling fork, vfork, and execve Handling on page 566
Assembler code	Viewing the Assembler Version of Your Code on page 173

Using File Extensions

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

Note that your installation may have its own guidelines for compiling programs.

Starting TotalView

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

NOTE: Starting TotalView with no arguments (just entering `totalview` in your shell) launches the Sessions Manager's Start a Debugging Session dialog, Starting a Debugging Session on page 102.

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

```
totalview [ executable [ core-files | recording-file ] ] [ options ]
```

where ***executable*** is the name of the executable file to debug, ***core-files*** is the name of one or more core files to examine, and recording-file is the name of a ReplayEngine recording to load.

CLI: totalviewcli [*executable* **[** *core-files* | *recording-file* **]] [** *options* **]**

Your environment may require you to start TotalView in another way. For example, if you are debugging an MPI program, you must invoke TotalView on **mpirun**. For details, see Setting Up Parallel Debugging Sessions on page 546.

Note that you can use the GUI and the CLI at the same time. Use the **Tools > Command Line** command to display the CLI's window.

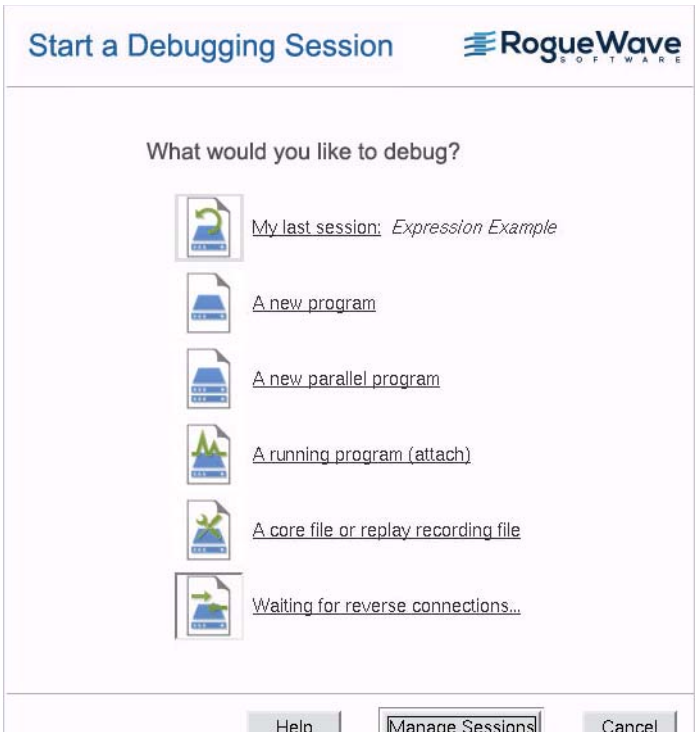
NOTE: Your installation may have its own procedures and guidelines for running TotalView.

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

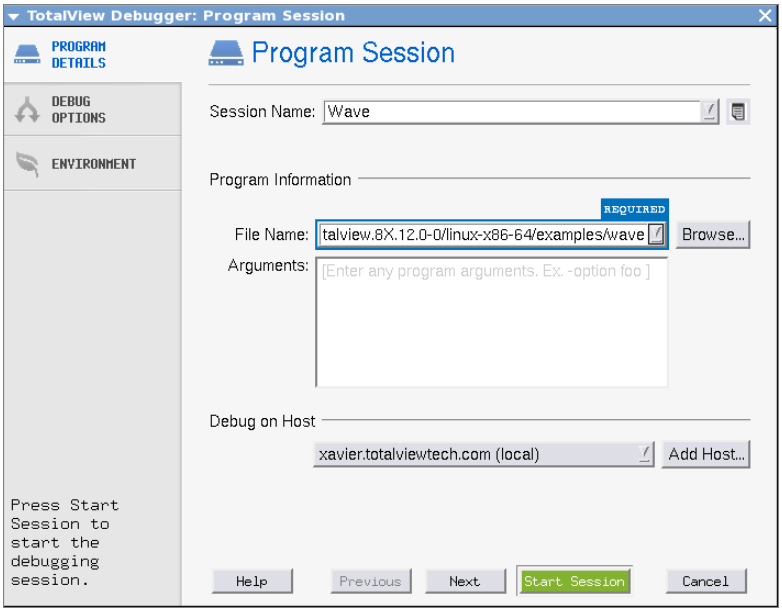
Starting TotalView

`totalview`

Starts TotalView without loading a program or core file. Instead, TotalView launches the Sessions Manager's Start a Debugging Session dialog where you can choose the type of session you plan to debug.



When you select your type of session, the relevant dialog launches. For instance, if you select A new program, the Program Session dialog launches.



Notice the **Debug Options** tab in the sidebar. Selecting this launches a dialog that enables reverse debugging with ReplayEngine and memory debugging with MemoryScape.

On the CLI, enter:

CLI: totalviewcli
dload *executable*

Starting on Mac OS X

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

*installdir***/TotalView.app/totalview**

where *installdir* is where TotalView is installed.

If TotalView was installed on your system without **procmod** permission, you will not be able to debug programs. If TotalView detects this problem, it displays a dialog box with information on how to fix it.

Creating or Loading a Session

totalview -load_session *session_name*

Creates a process based on the session values. Sessions that attach to an existing process cannot be loaded this way; use the `-pid` command line option instead

CLI: totalviewcli
dsession -load *session_name*

Debugging a Program

totalview *executable*

Starts TotalView and loads the ***executable*** program.

CLI: totalviewcli *executable*

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

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

Debugging a Core File

totalview *executable corefiles*

Starts TotalView and loads the ***executable*** program and one or more associated ***core-files***.

CLI: dattach -c *core-files* **-e** *executable*

The ***core-files*** argument represents one or more core files associated with this executable. You can use wild cards in the core file names.

Debugging with a Replay Recording File

totalview *executable recording-file*

Starts TotalView and loads the ***executable*** program and an associated ***recording-file***. The recording file was saved in a previous debugging session that used the Replay feature, and restores the state of that debugging session, including all Replay information.

CLI: dattach -c *recording-file* **-e** *executable*

Passing Arguments to the Program Being Debugged

`totalview` *`executable`* `-a` *`args`*

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

CLI: totalviewcli *executable* **-a** *args*

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

CLI: dset ARGS_DEFAULT {*value***}**

`totalview -args` *`executable args`*

Similar to above, but uses the command line option **-args** to specify that the *executable* program and arguments follow.

CLI: totalviewcli -args *executable args*

Debugging a Program Running on Another Computer

`totalview` *`executable`* `-remote` *`hostname_or_address[:port]`*

Starts TotalView on your local host and the **tvdsvr** command (which implements and controls debugging on remote machines) 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***]**

If TotalView fails to automatically load a remote executable, you may need to disable auto-launching for this connection and manually start the **tvdsvr**. (*Autolaunching* is the process of automatically launching **tvdsvr** processes.) To disable autolaunching, add the *hostname*:*portnumber* suffix to the name entered in the **Debug on Host** field of the **File > Debug New Program** dialog box. As always, the *portnumber* is the TCP/IP port number on which TotalView server is communicating with TotalView. See Starting the TotalView Server Manually on page 492 for more information.

NOTE: TotalView Individual does not allow remote debugging.

Debugging an MPI Program

The exact details for starting the debugger on an MPI program vary greatly from system to system, so consult with your local system documentation for details. The following are generic examples that may or may not work on your specific system.

`totalview`

Method 1: In many cases, you can start an MPI program in much the same way as you would start any other program. However, you need to select **A New Parallel program** from the **Start a Debugging Session** dialog box, and enter the MPI version and other information on the parallel program to debug.

`totalview -args` *`mpirun mpirun-args`*

Method 2: Invokes TotalView on the MPI starter program *mpirun*. The *mpirun-args* are the arguments to pass to the MPI starter program, such as the number of processes, number of nodes, the name of the MPI application program to debug, and arguments for the application. For example:

```
totalview -args mpirun -np 4 ./myMPIprog myDataFile
```

`mpirun -np` *`count`* `-tv` *`executable`*

Method 3: The MPI **mpirun** command starts the TotalView executable pointed to by the **TOTALVIEW** environment variable. TotalView then starts your program. This program runs using **count** processes.

Starting TotalView on a Script

It is sometimes convenient to start TotalView on a shell script. For example, a typical use case might be a script that calls into a shared library, and you need to debug the shared library code; another case is a shell script that sets environment variables, then execs the application to debug.

Anywhere in the examples above that an *executable* can be specified, an interpreter script can be specified instead. The underlying interpreter, which must be a valid executable object file for the platform, is debugged, not the script itself.

On Unix, an interpreter script starts with a line that is similar to the following:

```
#! interpreter [ interpreter-arg ]
```

Where:

- `#!` are the first two characters in the file.
- `interpreter` is the path to an executable object file or some other interpreter script.
- `interpreter-arg` is an optional argument to pass to the interpreter.

When the interpreter script is executed, the interpreter is invoked by the system as follows:

```
interpreter [ interpreter-arg ] script [ script-args ]
```

Here's a simple example:

```
% cat myscript.sh
#! /bin/sh -x
echo "$@"
% ./myscript.sh a b c
+ echo a b c
a b c
%
```

In the example above, the following command was executed:

```
/bin/sh -x ./myscript.sh a b c
```

Whenever TotalView is processing an executable file, it first checks to see if the file is an interpreter script. If the file starts with `#!`, it is treated as an interpreter script. The path to the interpreter is extracted from the script and used as the executable object file to debug. If the interpreter file is itself an interpreter script, TotalView repeats the procedure (up to 40 times) until it encounters an interpreter file that is not an interpreter script. If the procedure fails to find a valid executable object file for the platform, loading the script into the debugger will fail.

In most cases, the interpreter for the script does not directly contain the code you want to debug, and instead dynamically loads or executes the code to debug. TotalView contains several configuration settings that make it easier to plant breakpoints and stop in your code, as described by the Related Topics below. There are three common cases, where the interpreter script:

- Dynamically loads a shared library and calls into the code to debug (see the entries below regarding dynamic library handling and creating pending breakpoints)
- Execs the program containing the code to debug (relevant to exec handling)
- Runs the program containing the code to debug (relevant to fork handling)

RELATED TOPICS

Debugging parallel programs such as MPI, UPC, or CAF, including invoking TotalView on **mpirun**	Setting Up Parallel Debugging Sessions on page 546
Remote debugging	Setting Up Remote Debugging Sessions on page 484, and "TotalView Debugger Server (tvdsvr) Command Syntax" in the *Classic TotalView Reference Guide*.

RELATED TOPICS	
The **totalview** command	"TotalView Command Syntax" in the *Classic TotalView Reference Guide*
Separating debug information from an executable using a **gnu_debuglink** file.	"Using gnu_debuglink Files" in the *Compilers and Platforms* chapter of the *Classic TotalView Reference Guide*
Configuring dynamic library handling	"Debugging Your Program's Dynamically Loaded Libraries" in the *Classic TotalView Reference Guide*.
Creating a pending breakpoint	Pending Breakpoints on page 201
Configuring exec handling	Exec Handling on page 568
Configuring fork handling	Fork Handling on page 569

Initializing TotalView

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

NOTE: It is sometimes desirable to bypass defaults that have been set in either a global or a private initialization file. To bypass the default execution of startup scripts, you can specify `-no_startup_scripts` on the TotalView startup command line.

An *initialization file* stores CLI functions, set variables, and execute actions that TotalView interprets when it begins executing. This file, which you must name **tvdrc**, resides in the **.totalview** subdirectory contained in your home directory. TotalView creates this directory for you the first time it executes.

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

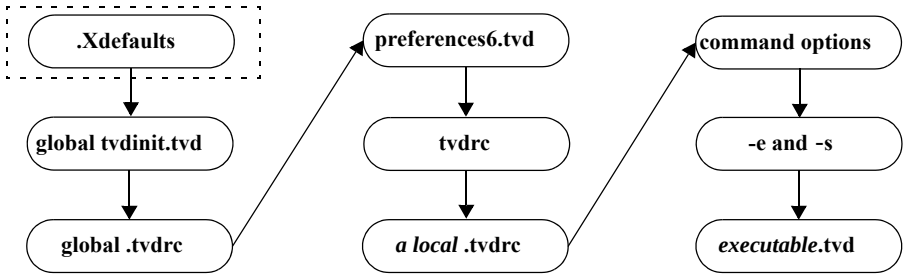
NOTE: Before Version 6.0, you placed your personal **.tvdrc** file in your home directory. If you do not move this file to the **.totalview** directory, TotalView will still find it. However, if you also have a **tvdrc** file in the **.totalview** directory, TotalView ignores the **.tvdrc** file in your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

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

If you add the **-s** ***filename*** option to either the **totalview** or **totalviewcli** shell command, TotalView executes the CLI commands contained in ***filename***. This startup file executes after a **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 create a shell command that starts the CLI.

Figure 24 shows the order in which TotalView executes initialization and startup files.

Figure 24, Startup and Initialization Sequence



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

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

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

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

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

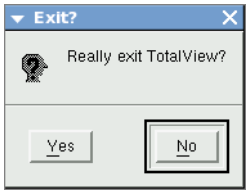
If you are debugging multi-process programs that run on more than one computer, TotalView caches library information in the **.totalview** subdirectory. If you want to move this cache to another location, set **TV::library_cache_directory** to this location. TotalView can share the files in this cache directory among users.

RELATED TOPICS	
The **TV::process_load_callbacks** variable	"TotalView Variables" in the *Classic TotalView Reference Guide*
Saving action points	Saving Action Points to a File on page 239

Exiting from TotalView

To exit from TotalView, select **File > Exit**. You can select this command in the Root, Process, and Variable Windows. After selecting this command, TotalView displays the dialog box shown in Figure 25.

Figure 25, File > Exit Dialog Box



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

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

CLI: exit

Note that if both the CLI and the GUI are open, and you want to exit only from the CLI, type Ctrl+D.

Loading and Managing Sessions

This chapter discusses how to set up a TotalView session, based on some of the most-used setup commands and procedures.

There are two primary ways to load programs into TotalView for debugging: the GUI via the Sessions Manager (Loading Programs from the Sessions Manager) or the CLI (Loading Programs Using the CLI) using its various commands. Both support all debugging session types.

For information on setting up remote debugging, see Setting Up Remote Debugging Sessions on page 484.

For information on setting up parallel debugging sessions, see Setting Up MPI Debugging Sessions on page 516 and Setting Up Parallel Debugging Sessions on page 546.

This chapter discusses:

Setting up Debugging Sessions

NOTE: Setting up parallel debugging sessions is not discussed in this chapter. Rather, see Setting Up MPI Debugging Sessions.

Additional Session Setup Options

Managing Debug Sessions

Other Configuration Options

Setting up Debugging Sessions

The easiest way to set up a new debugging session is to use the Sessions Manager, which provides an easy-to-use interface for configuring sessions and loading programs into TotalView. Alternatively, you can use the CLI.

Loading Programs from the Sessions Manager on page 102

Loading Programs Using the CLI on page 114

Loading Programs from the Sessions Manager

TotalView can debug programs on local and remote hosts, and programs that you access over networks and serial lines. The **File** menu of both the Root and Process windows contains a series of debug options to load local and remote programs, core files, and processes that are already running.

Figure 26, Debugging options from the File Menu

Each of these debug options launches the Sessions Manager where you can configure a new debug session or launch a previous session.

From this menu, you can also select **Manage Sessions** to edit or delete previously saved debug sessions.

NOTE: Your license limits the number of programs you can load. For example, TotalView Individual limits you to 16 processes or threads.

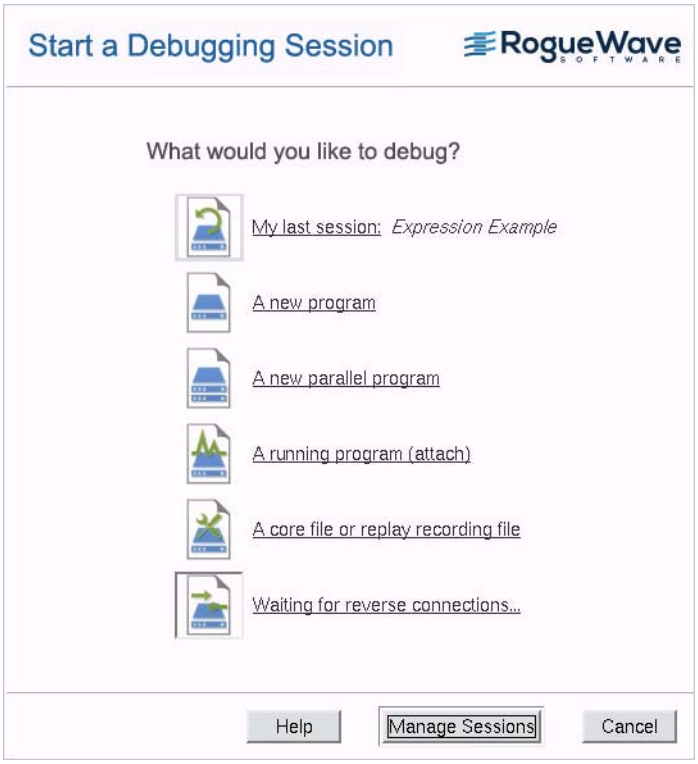
Starting a Debugging Session

Access the main page of the Sessions Manager either directly from your shell by just entering

```
totalview
```

or by selecting **File > New Debugging Session** in the Root and Process windows.

Figure 27, Start a Debugging Session dialog box



The Start a Debugging Session dialog of the Sessions Manager can configure various types of debugging sessions, depending on your selection. These are:

- Debugging a New Program on page 103
- Debugging a parallel application in Starting MPI Programs Using File > Debug New Parallel Program on page 518
- Attaching to a Running Program on page 105
- Debugging a Core File on page 110
- Debugging with a Replay Recording File on page 112
- Launching your Last Session on page 113
- Waiting for Reverse Connections on page 114

Debugging a New Program

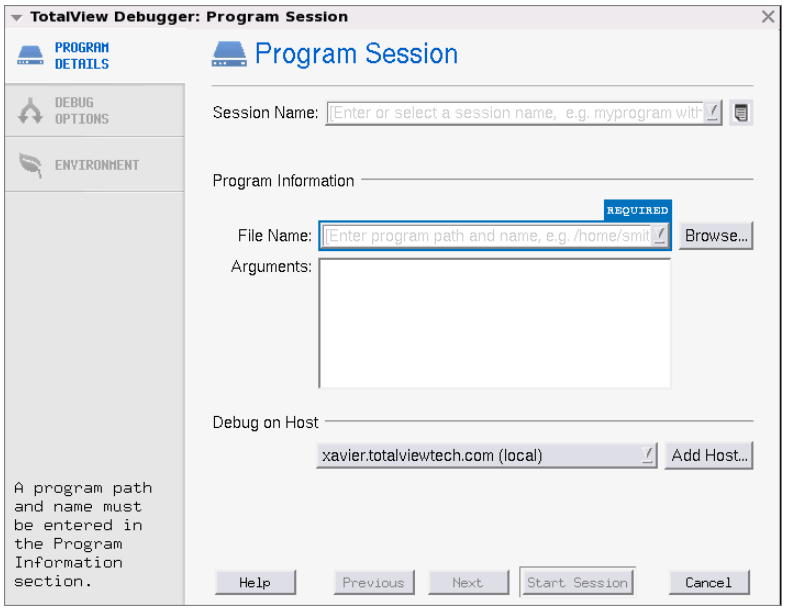
To configure a new debugging session, either:

- Select **A new program** to launch the Program Session dialog, or

- Select **File > Debug New Program** from the Root or Process windows, if TotalView is already running.

  The Program Session dialog launches.

Figure 28, Program Session dialog



1. Enter a session name in the **Session Name** text box.

> **NOTE:** Note that any previously entered sessions of the same type are available from the Session Name dropdown box. Once selected, you can change any session properties and start your debug session. See Editing or Starting New Sessions in a Sessions Window on page 126.

2. Enter the name of your program in the **File Name** box or press **Browse** to browse to and select the file. You can enter a full or relative path name. If you have previously entered programs here, they will appear in a drop-down list.

   If you enter a file name, TotalView searches for it in the list of directories named using the **File > Search Path** command or listed in your **PATH** environment variable.

   CLI: dset EXECUTABLE_PATH

3. *(Optional)* Add any custom configurations or options:

   - **Remote debugging**: Select or add a remote host, if the program is to be executed on a remote computer. See Adding a Remote Host on page 116.

- **Program arguments**: Enter any program arguments into the Arguments field.

  Because you are loading the program from within TotalView, you will not have entered the command-line arguments that the program needs. For detail, see **Program Arguments** in the In-Product Help.

- **Debugging Options:** See Options: Reverse Debugging, Memory Debugging, and CUDA on page 118.

- **Environment variables or standard I/O**: See Setting Environment Variables and Altering Standard I/O on page 120

- **Notes**: You can add any notes to the session by selecting the Note icon ( ). See Adding Notes to a Session on page 122.

4. Click **Start Session**. The Start Session button is enabled once all required information is entered.

Attaching to a Running Program

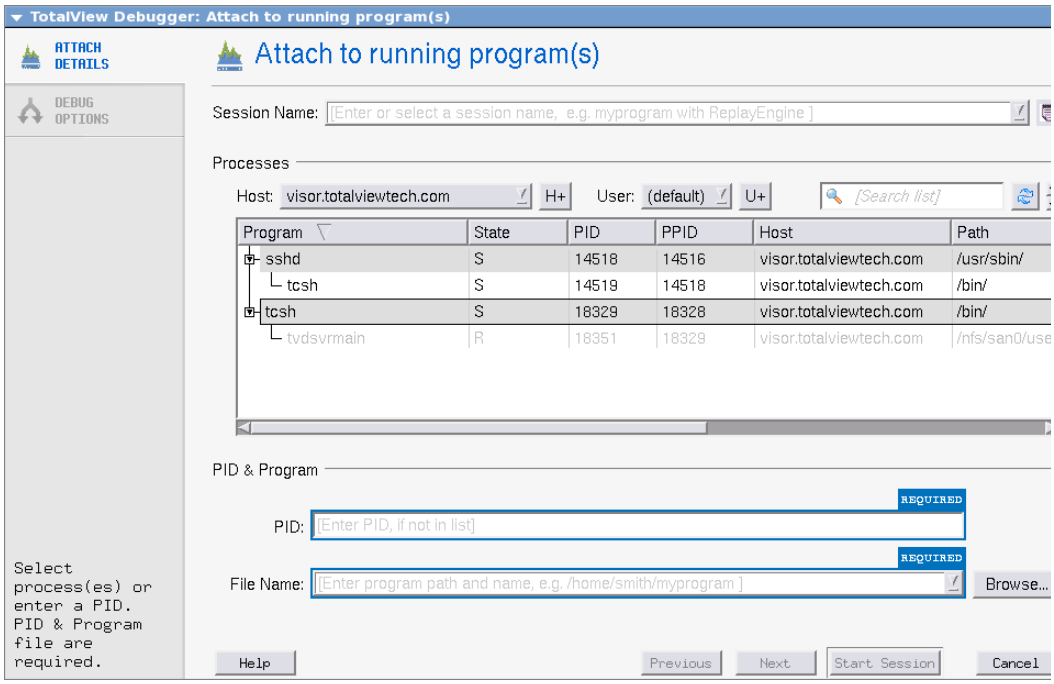
If a program you're testing is hung or looping (or misbehaving in some other way), you can attach to it while it is running. You can attach to single and multi-process programs, and these programs can be running remotely.

To open the Attach window, select either

- **A running program (attach)** on the Start a Debugging Session dialog, or
- **File > Attach to a Running Program** from the Root or Process window if TotalView is already running.

  A list of processes running on the selected host displays in the **Attach to running program(s)** dialog.

Figure 29, Attaching to an existing process



In the displayed list, processes to which TotalView is already attached are shown in gray text, while the processes displayed in black text are not currently running under TotalView control.

1. Enter a name for this session in the **Session Name** field.

> **NOTE:** Any previously entered sessions of the same type are available from the Session Name dropdown box. Once selected, you can change any session properties and start your debug session. See Editing or Starting New Sessions in a Sessions Window on page 126.

2. Click on the program's name under the Program column, and press **Start Session**.

CLI: dattach *executable pid*

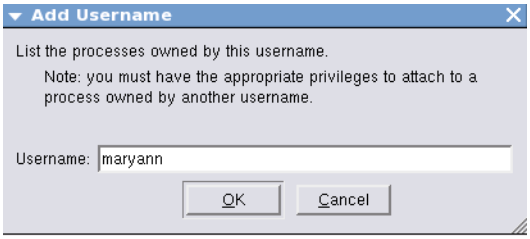
While you must link programs that use **fork()** and **execve()** with the TotalView dbfork library so that TotalView can automatically attach to them when your program creates them, programs that you attach to need not be linked with this library.

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

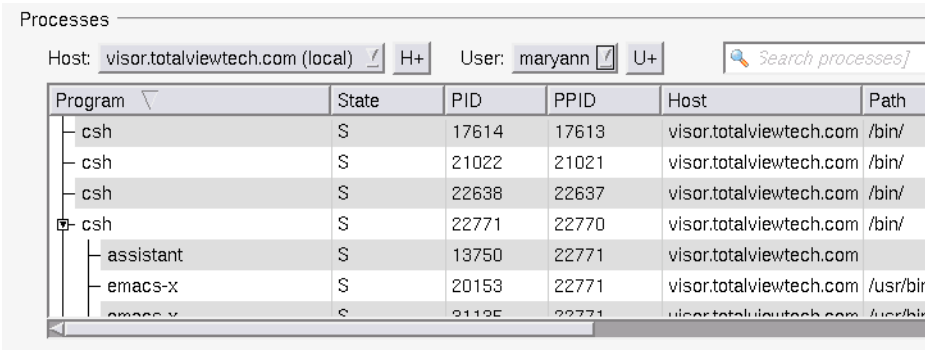
Adding a New User

You can enter a user name to see the processes owned by that user. If you wish to attach to a process owned by someone else, you need the proper permissions.

1. Click the U+ icon to launch the **Add username** dialog.
2. Enter a known username, then click OK.



If the username is not recognized, the system returns an error; otherwise, the user is added to the User drop-down and selected as the current user.

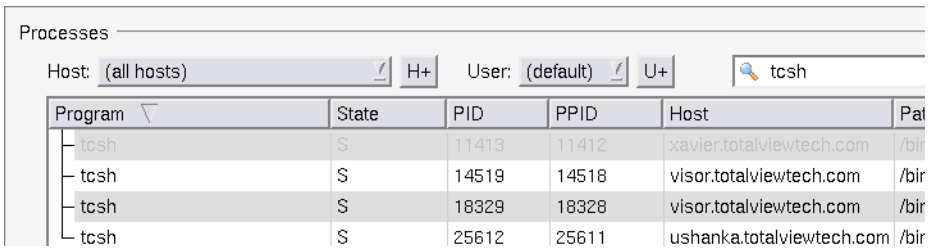


The selected user's processes are displayed. Attach to another user's processes just by clicking the process and selecting **Start Session**.

If you do not have permissions to attach to the process, an error is returned.

Searching for Processes

You can search for any process using the search box (Search processes]). If found, the process will display in the Processes pane.

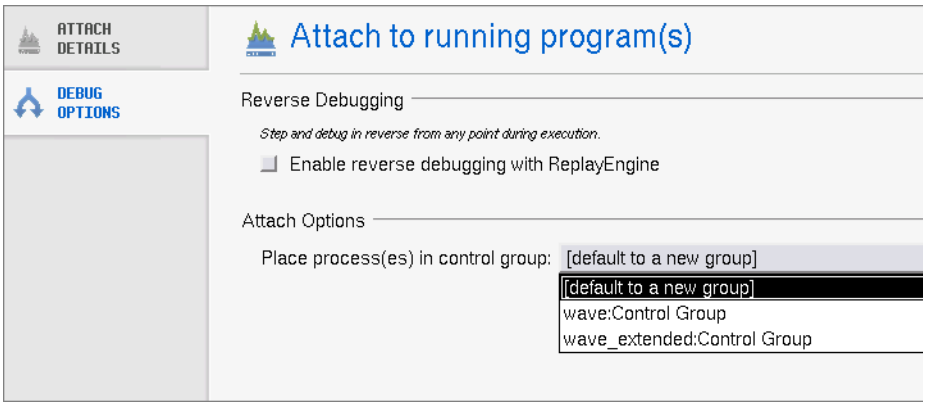


RELATED TOPICS	
Attached process states	Seeing Attached Process States on page 411
Starting TotalView	Starting TotalView on page 89
Using the Root Window	Using the Root Window on page 147
File > Attach to a Running Program	**Process > Detach** in the in-product Help

Attach Options

On the **Debug Options** tab, two options exist:

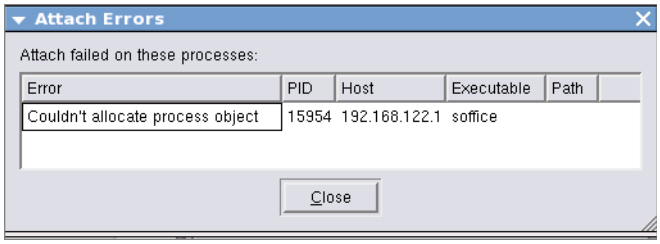
- Enabling ReplayEngine, which is an option available to all other debugging sessions (See Options: Reverse Debugging, Memory Debugging, and CUDA on page 118)
- Placing the processes to which you are attaching into a control group under the **Attach Options** area.



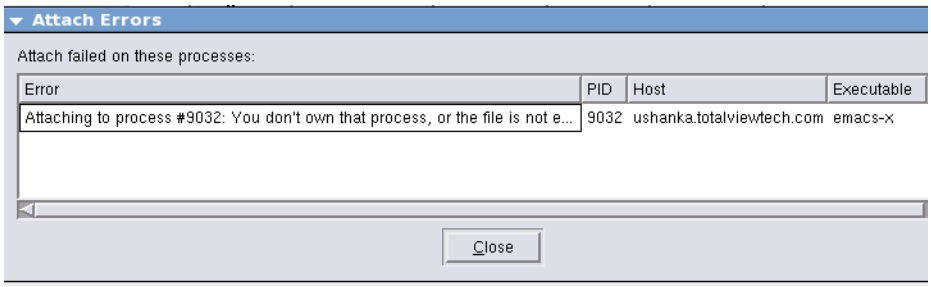
If you have selected a group in previous TotalView sessions and the group exists in the dropdown, it is selected for you. Otherwise, the default is to create a new group to contain all processes attached to in this session.

Attaching Errors

If TotalView returns an error while attempting to attach to a process, it is usually because you do not have permission to attach to that process. For example, the process could not be allocated:



Or you don't own the process:



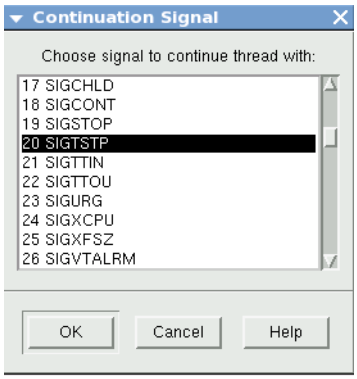
Detaching from Processes

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

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

To detach from processes that TotalView did not create:

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



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

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

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

   CLI: ddetach

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

RELATED TOPICS	
The **Process > Detach** command in detail	**Process > Detach** in the in-product Help
The CLI **ddetach** command	**ddetach** in the *Classic TotalView Reference Guide*
The continuation signal	**Thread > Continuation Signal** in the in-product Help

Debugging a Core File

If a process encounters a serious error and dumps a core file, you can load the file to examine it.

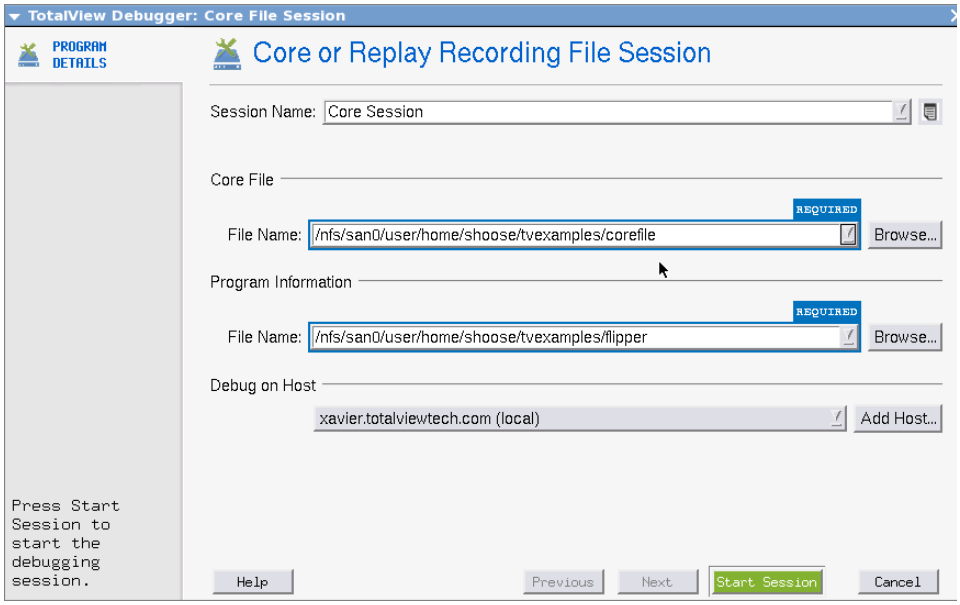
To debug a core file, select either

- **A core file or replay recording file** on the Start a Debugging Session dialog, or
- **File > Debug Core or Replay Recording** from the Root or Process window if TotalView is already running.

The "Core or Replay Recording File Session" dialog launches. Enter a name for the session and the program and core file's name.

NOTE: Any previously entered sessions of the same type are available from the Session Name drop-down box. Once selected, you can change any session properties and start your debug session. See Editing or Starting New Sessions in a Sessions Window on page 126.

Figure 30, Open a Core File



If the program and core file reside on another system, enter that system's name in the Debug on Host area (see Adding a Remote Host on page 116 for details).

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

When TotalView launches, 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. See Examining and Editing Data and Program Elements on page 240.

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

RELATED TOPICS

Debugging a Core File	The **File > Debug Core File** dialog in the in-product Help
The CLI **dattach** command's **`-c core-files`** option	**dattach** in the *Classic TotalView Reference Guide*

Debugging with a Replay Recording File

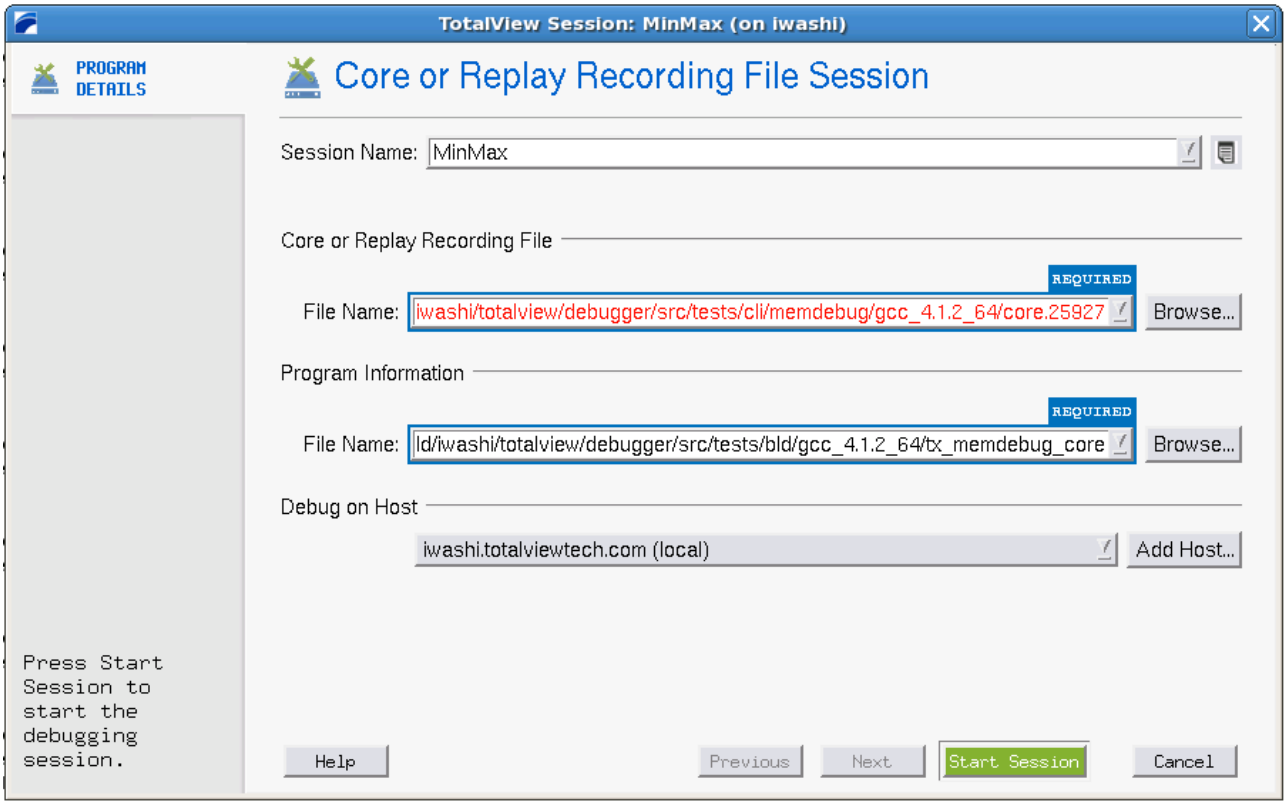
Previously saved Replay recording files can be loaded to further debug applications. These recording files contain the state of a previous debugging session, including all Replay information.

To debug with a replay recording file, select either

- A **core file or replay recording file** on the **Start a Debugging Session** dialog, *or*
- **File > Debug Core or Replay Recording File** from the Root or Process window if TotalView is already running.

The **Core or Replay Recording File Session** dialog launches. Enter a name for the session, the program, and replay recording file's name.

Figure 31, Open a Replay Recording File



If the program and replay recording file reside on another system, enter that system's name in the **Debug on Host** area (see Adding a Remote Host on page 116 for details).

When TotalView launches, the Process Window displays the replay recording file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when the replay session was saved. The title bar of the Process Window displays "Recording File". The right arrow in the line number area of the Source Pane indicates the value of the program counter (PC) when the process was saved.

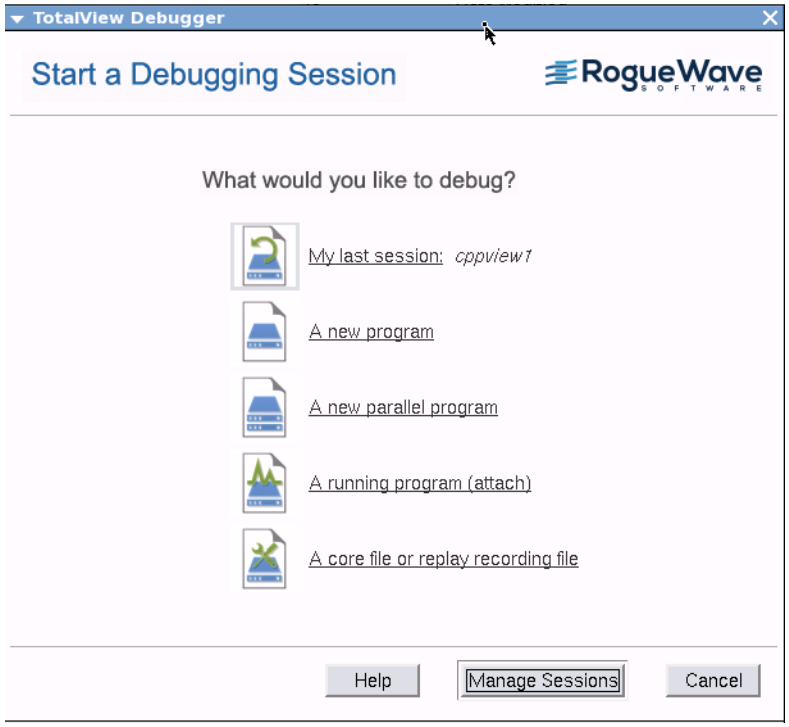
You can examine the state of all variables at the time the replay recording file was saved. See Examining and Editing Data and Program Elements on page 240.

RELATED TOPICS	
Debugging with a Replay recording file	The **File > Debug Core or Replay Recording File** dialog in the in-product Help
The CLI **dattach** command's **`-c recording-file`** option	**dattach** in the *Classic TotalView Reference Guide*

Launching your Last Session

The initial window of the Sessions Manager displays your most recent session so you can quickly continue a debugging session where you left off.

Figure 32, Start a Debugging Session



If you click on **My last session**, TotalView immediately launches based on your last session's settings, and displays the Process Window.

If you do wish to edit any properties of your last session, just select the **Manage Sessions** button to instead launch the Manage Debugging Sessions page.

Waiting for Reverse Connections

Using the Reverse Connect feature, you can run the TotalView UI on a front-end node to debug a job executing on compute nodes. For more information, see Reverse Connections.

Loading Programs Using the CLI

When using the CLI, you can load programs in a number of ways. Here are a few examples.

Load a session	**dsession -load** *session_name* If the preference "Show the Startup Parameters dialog on startup" is selected, this command launches the Sessions Manager so you can edit session properties; otherwise, it loads the session directly into Totalview and launches the Process and Root windows.

Start a new process

dload -e *executable*

Open a core file

dattach -c *core-files* **-e** *executable*

If TotalView is not yet running, you can also provide the core file as a startup argument, like so:

```
totalview executable core-files [ options ]
```

Open a replay recording session file

dattach -c *recording-file* **-e** *executable*

If TotalView is not yet running, you can also provide the replay recording session file as a startup argument, like so:

```
totalview executable recording-file [ options ]
```

Load a program using its process ID

dattach *executable pid*

Load a program on a remote computer

dload *executable* **-r** *hostname*

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

Load a poe program

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

RELATED TOPICS

CLI commands	"CLI Commands" in the *Classic TotalView Reference Guide*
Using the CLI	Using the CLI on page 454

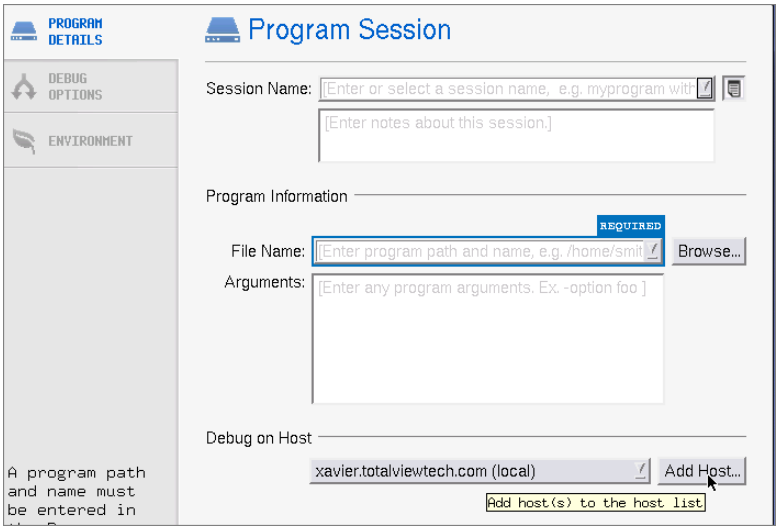
Debugging Options and Environment Setup

A debugging session can be customized in a variety of ways. This section discusses

Adding a Remote Host

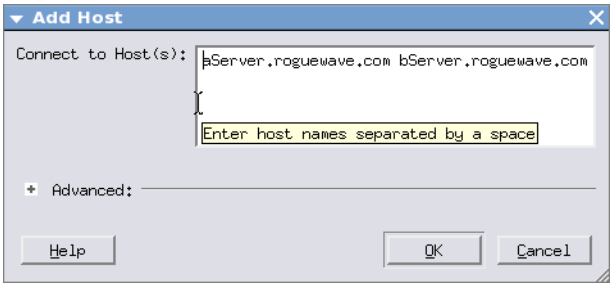
To debug a program running on a remote computer, enter the computer name in the Debug on Host area, or select it from the dropdown if already entered.

Figure 33, Debug on Host area



To enter a new host, select the **Add host** button and enter its name or IP address in the displayed dialog box.

Figure 34, Add Host dialog



You can add multiple hosts separated by a space. Alternatively, add the IP address. For example:

```
server1 server2 server3
```

or

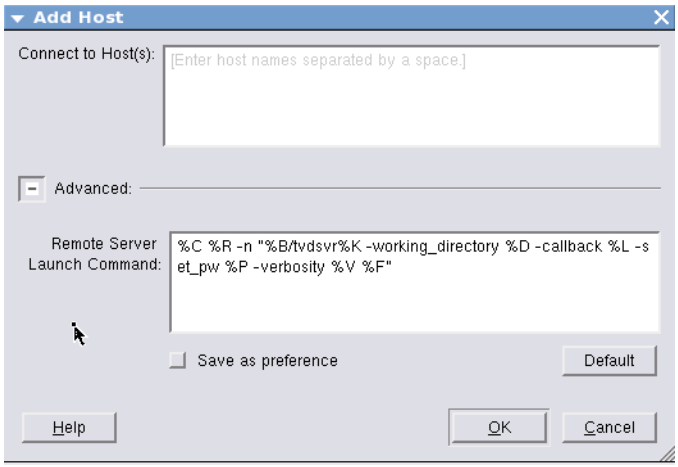
```
10.5.6.123 10.5.7.124
```

If TotalView cannot connect to the host, it displays an error dialog.

Figure 35, Add Host Failure popup

To modify the launch string or to view the default string TotalView will use to launch the remote debug session, select the **Advanced** button ( ) to open the **Remote Server Launch Command** field.

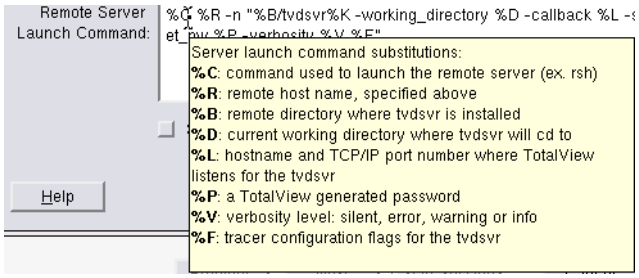
Figure 36, Remote server launch command



You can edit this string and then click **Save as preference** to have TotalView save it to your preferences. Once saved, you can view or edit it at **File > Preferences > Launch Strings** tab.

Note that if you hover your cursor inside the text field, a popup displays the substitutions used in the launch string.

Figure 37, Remote server launch command substitutions



If TotalView supports your program's parallel process runtime library (for example, MPI, UPC, or CAF), it automatically connects to remote hosts. For more information, see Setting Up Parallel Debugging Sessions on page 546.

RELATED TOPICS	
Editing the server launch command	Customizing Server Launch Commands on page 498
TotalView command line options	"Command-Line Options" in the *Classic TotalView Reference Guide*
The **tvdsr** command	"The TotalView Debugger Server Command Syntax" in the *Classic TotalView Reference Guide*
Remote debugging	Setting Up Remote Debugging Sessions on page 484

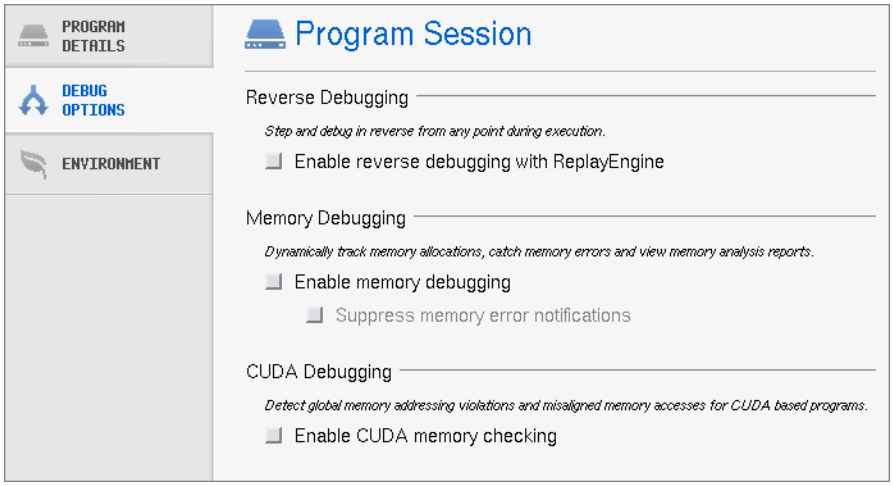
Options: Reverse Debugging, Memory Debugging, and CUDA

You can choose to enable various additional debugging features for a given session. Select the **Debug Options** tab to launch options.

Depending on the type of debug session, different options are available:

- The **New Program** and **New Parallel Program** windows offer reverse debugging, memory debugging, and CUDA debugging options:

Figure 38, Debug Options for Reverse, Memory or CUDA debugging



- **Reverse Debugging**. Record all program state while running and then roll back your program to any point.

  The **Enable ReplayEngine** check box is visible only on Linux-x86-64 platforms. If you do not have a license for ReplayEngine, enabling the check box has no effect, and TotalView displays an error message when your program begins executing. Selecting this check box tells TotalView that it should instrument your code so that you can move back to previously executed lines.

- **Memory Debugging**. Track dynamic memory allocations. Catch common errors, leaks, and show reports.

  Enabling memory debugging here is the same as enabling it within MemoryScape or using the Process Window's **Debug> Enable Memory Debugging** command.

  The **Enable memory debugging** and **Suppress memory error notifications** check boxes perform the same functions as the **Enable memory debugging** and **On memory event, halt execution** checkboxes do within the Advanced Options on MemoryScape's Memory Debugging Options page. This is the equivalent of the basic Low setting.

- **CUDA Debugging**. Detect global memory addressing violations and misaligned memory accesses for CUDA-based programs.

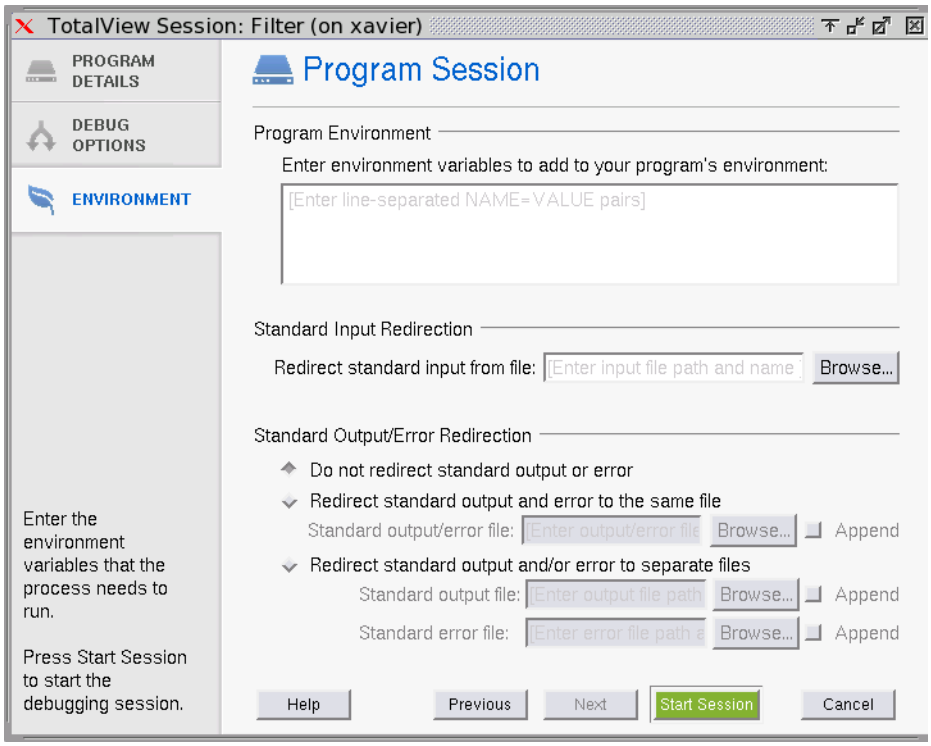
- The **Attach to a Running Program** window supports only ReplayEngine and a special Attach option. For more information, see Attaching to a Running Program on page 105.

RELATED TOPICS	
Reverse Debugging	"Understanding ReplayEngine" in the ReplayEngine User Guide
Memory Debugging	More on MemoryScape in *Debugging Memory Problems with MemoryScape*
CUDA Debugging	Part V, Using the CUDA Debugger on page 626
Attach options	Attach Options on page 108

Setting Environment Variables and Altering Standard I/O

When loading the program from within TotalView, you can add any necessary environment variables or alter standard I/O using the **Environment** tab.

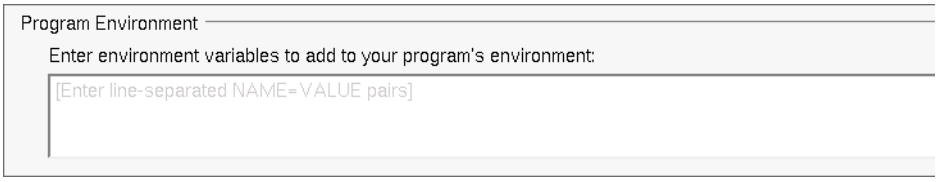
Figure 39, Setting Environment Variables and Altering Standard I/O



Environment Variables

Enter environment variables in the field in the Program Environment area.

Figure 40, Setting environment variables



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

At startup, TotalView reads in your environment variables to ensure that your program has access to them when the program begins executing. Use the Program Environment area to add additional environment variables or to override values of existing variables.

NOTE: TotalView does not display the variables that were passed to it when you started your debugging session. Instead, this field displays only the variables you added using this command.

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

```
DISPLAY=enterprise:0.0
```

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

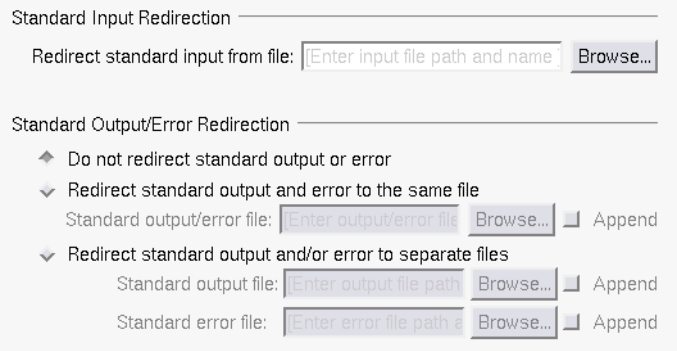
RELATED TOPICS

Environment options in the **File > Debug New Program** dialog	"Environment Variables" in the in-product Help

Standard I/O

Use the controls in the Input Processing and Standard and Error Output Processing to alter standard input, output, and error. In all cases, name the file to which TotalView will write or from which TotalView will read information. Other controls append output to an existing file if one exists instead of overwriting it or merge standard out and standard error to the same stream.

Figure 41, Resetting Standard I/O



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

RELATED TOPICS	
The standard I/O in the **File > Debug New Program** dialog	"Standard I/O" in the in-product Help

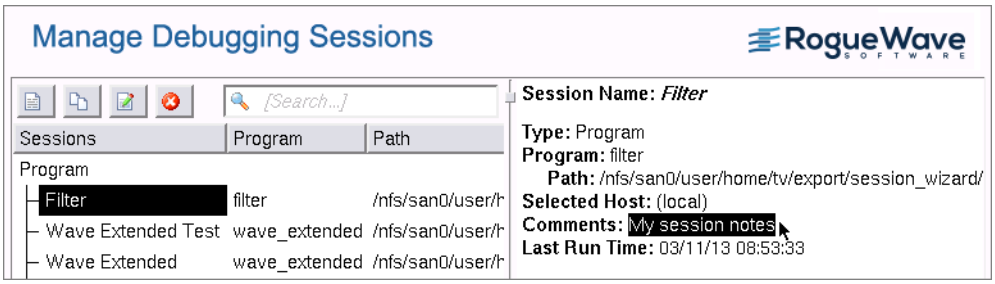
Adding Notes to a Session

On any of the Sessions Manager dialogs for configuring debugging sessions, you can add a note by selecting the **Note** icon ( ). This opens a simple text field where you can enter your note.

Figure 42, Adding a note to a session

Once added, your note is saved and viewable in the Manage Sessions dialog under Comments.

Figure 43, Viewing notes saved in a session



To edit your note, or any other option for this session, click the **Edit** button ( ) to launch the relevant session window.

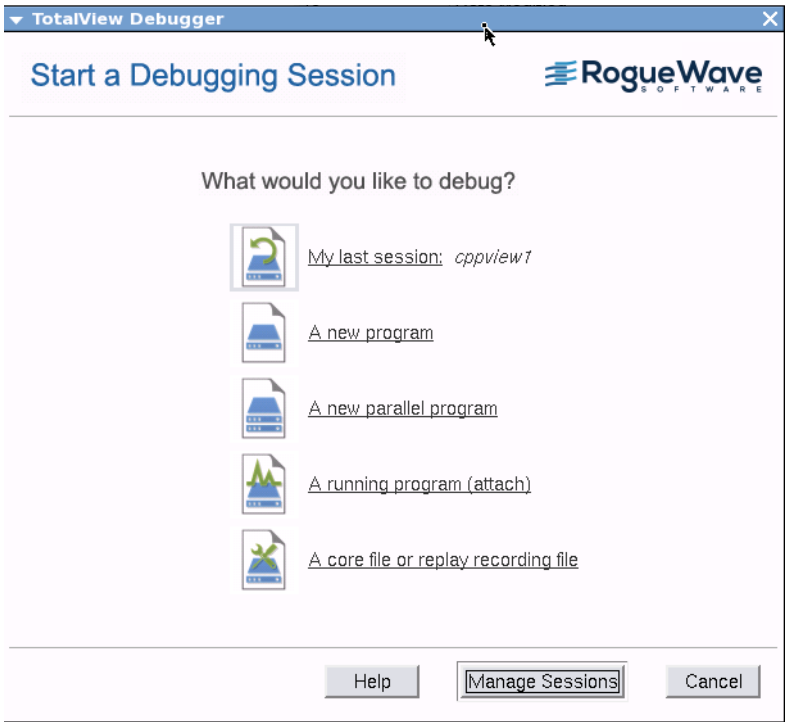
Managing Sessions

TotalView saves the settings for each of your previously-entered debugging sessions, available in the Manage Debugging Sessions window of the Sessions Manager. Here, you can edit, duplicate or delete sessions as well as start a session and create new sessions.

NOTE: You can also edit and create new sessions from any Sessions Window. See Editing or Starting New Sessions in a Sessions Window on page 126

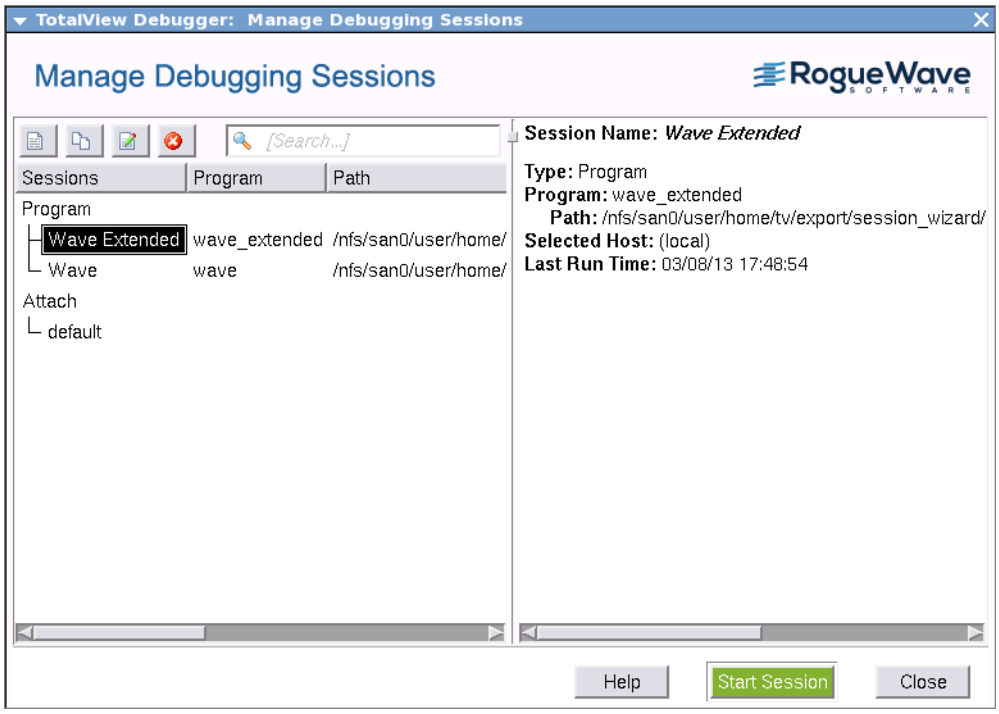
Access the Manage Debugging Sessions window, either from the **Start a Debugging Session** window of the Sessions Manager or from **File -> Manage Sessions** if TotalView is already running.

Figure 44, Accessing the Manage Sessions page



The Manage Debugging Sessions window launches. The left pane lists all sessions you have created. When you select a session in the left pane, the right pane displays data about that session.

Figure 45, Manage Debugging Sessions main page



If you have many sessions, you can search by keyword in the search box (). When found, TotalView immediately launches the session.

You can edit, delete and duplicate sessions using the icons in the left toolbar.

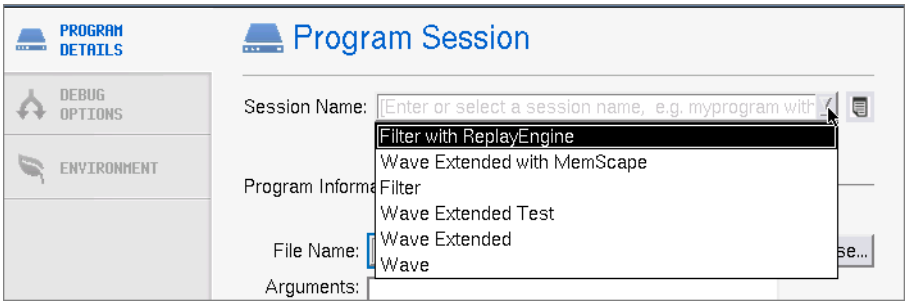
Icon	Action
	Creates a new debugging session, launching the Start a Debugging Session window of the Sessions Manager.
	Duplicates a session, naming it "<Session Name> Copy". You can rename and then edit this session.
	Edits a session, launching the appropriate window to change the session's configuration, either New Program, Parallel Program, Running Program or Core File.
	Deletes the session.

Editing or Starting New Sessions in a Sessions Window

In addition to editing a session using the Manage Debugging Sessions Window (Managing Sessions on page 124), you can also edit or even create a new session directly from any sessions window.

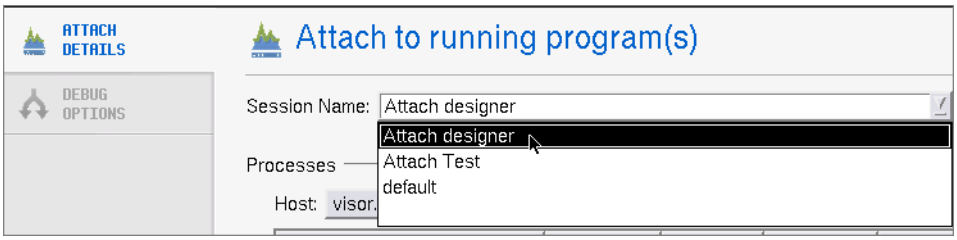
The **Session Name** field on each sessions window contains a dropdown that lists all previously created sessions of that type. For instance, from the Program Session window, you can access any session created in another Program Session:

Figure 46, Sessions Name dropdown of a Program Session window



Similarly, the Attach to a running program dialog displays any previous attach sessions:

Figure 47, Sessions Name dropdown of an Attach Session window



If you select a previous session, you can edit any field's values, even the Session Name to create an entirely new session. Then just click **Start Session** to launch that new debugging session.

Other Configuration Options

Handling Signals

If your program contains a signal handler routine, you may need to adjust the way TotalView handles signals. The following table shows how TotalView handles UNIX signals by default:

Signals Passed Back to Your Program		Signals Treated 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	

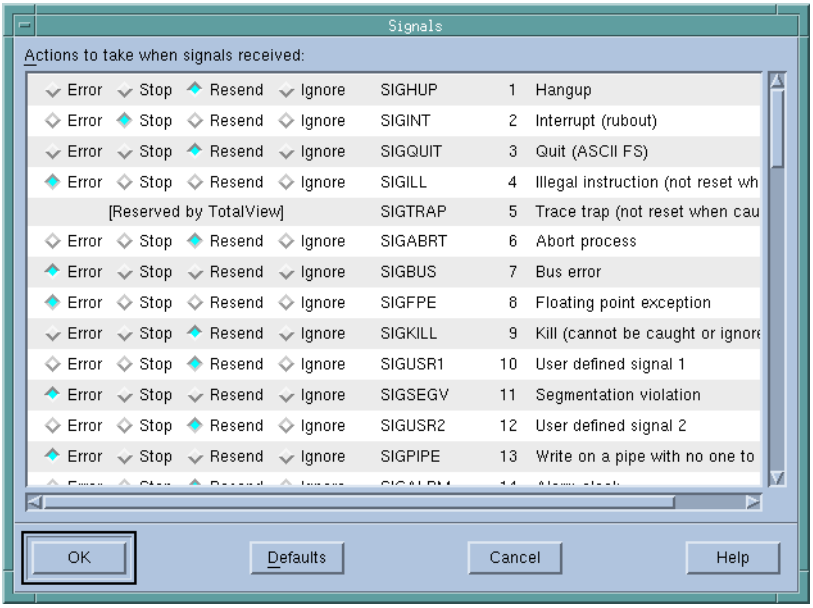
NOTE: TotalView uses the **SIGTRAP** and **SIGSTOP** signals internally. If a process receives either of these signals, TotalView neither stops the process with an error nor passes the signal back to your program. You cannot alter the way TotalView uses these signals.

On some systems, hardware registers affect how TotalView and your program handle signals such as **SIGFPE**. For more information, see Interpreting the Status and Control Registers on page 259 and the "Architectures" chapter in the *Classic TotalView Reference Guide*.

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

You can change the signal handling mode using the **File > Signals** command, Figure 48.

Figure 48, File > Signals Dialog Box



CLI: dset TV::signal_handling_mode

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

You can change the default way in which TotalView handles a signal by setting the **TV::signal_handling_mode** variable in a **.tvdrc** startup file. For more information, see "Command-Line Options" in the *Classic TotalView Reference Guide*.

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

CLI: dset TV::warn_step_throw

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

CLI: dset TV::GUI::pop_on_error

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

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

CLI: dset TV::GUI::pop_at_breakpoint

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

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

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

RELATED TOPICS

Thread continuation signal command	**Thread > Continuation Signal** in the in-product Help
The **TV::signal_handling_mode** variable in a **.tvdrc** startup file	"Command-Line Options" in the *Classic TotalView Reference Guide*
TotalView preferences	The **File > Preferences** dialog in the in-product Help and Setting Preferences on page 133

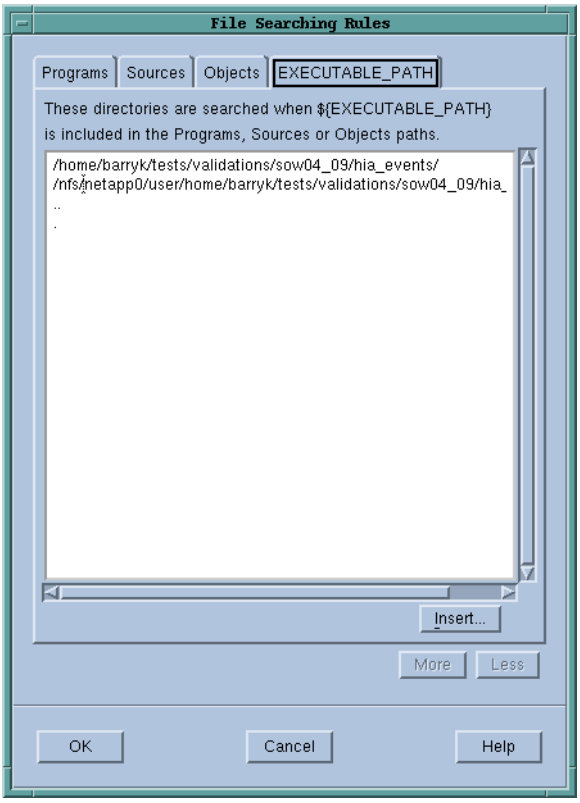
Setting Search Paths

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.

Figure 49, File > Search Path Dialog Box



TotalView searches the following directories in order:

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

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

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

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

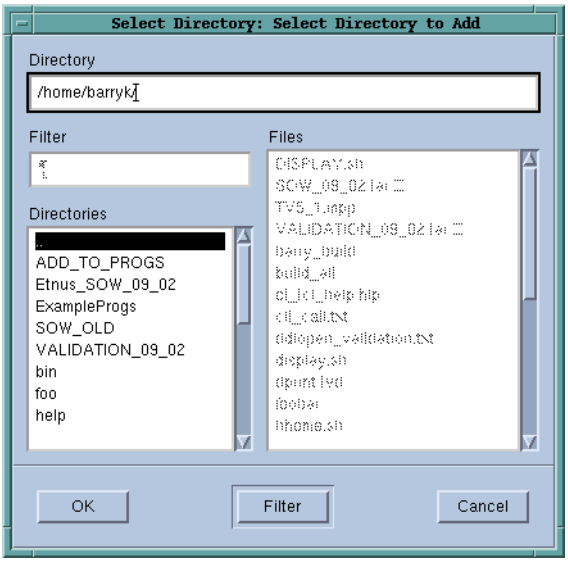
6. The **src** directory within your TotalView installation directory.

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

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

- Type path names directly.
- Cut and paste directory information.
- Click the **Insert** button to display the **Select Directory** dialog box that lets you browse through the file system, interactively selecting directories, as shown in Figure 50.

Figure 50, Select Directory Dialog Box



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

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

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

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

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

```
totalview prog_name
```

However, the following command does not:

```
totalview poe -a prog_name
```

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

RELATED TOPICS	
Starting TotalView	Starting TotalView on page 89
The **EXECUTABLE_PATH** environment variable	The **EXECUTABLE_PATH** variable in "TotalView Variables" in the *Classic TotalView Reference Guide*

Setting Startup Parameters

After you load a program, you may want to change a program's command-line arguments and environment variables or change the way standard input, output, and error behave. Do this using the **Process > Startup Parameters** command. The displayed dialog box is nearly identical to that displayed when you use the **File > Debug New Program** command, differing only in that it has an **Apply** button to save your entered parameters, rather than a **Start Session** button.

For information on other options you can edit here, see Options: Reverse Debugging, Memory Debugging, and CUDA on page 118 and Setting Environment Variables and Altering Standard I/O on page 120.

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

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

RELATED TOPICS	
The **ARGS_DEFAULT** variable	" TotalView Variables" in the *Classic TotalView Reference Guide* and **dset ARGS_DEFAULT {*value*}** on page 93
The **drun** command	**drun** n the *Classic TotalView Reference Guide*
The **drerun** command	**drerun** in the *Classic TotalView Reference Guide* and Restarting Programs on page 186

Setting Preferences

The **File > Preferences** command tailors many TotalView behaviors. This section contains an overview of these preferences. See **File > Preferences** in the in-product Help for more information.

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

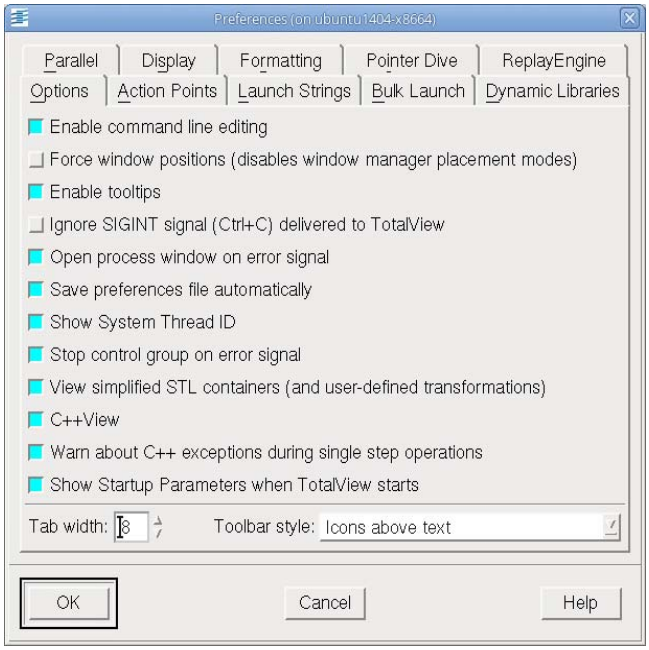
Every preference has a variable that you can set using the CLI. These variables are described in the"*Variables*" chapter of the *Classic TotalView Reference Guide*.

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

Options

This page contains check boxes that are either general in nature or that influence different parts of the system.

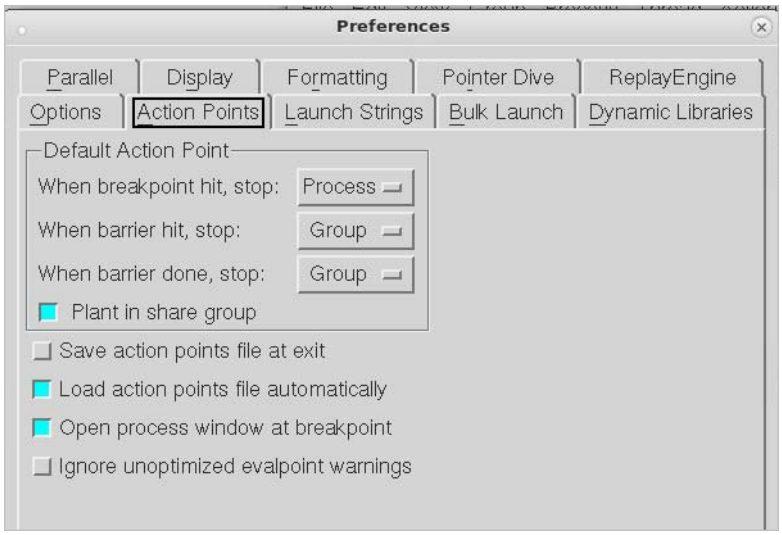
Figure 51, File > Preferences Dialog Box: Options Page



Action Points

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

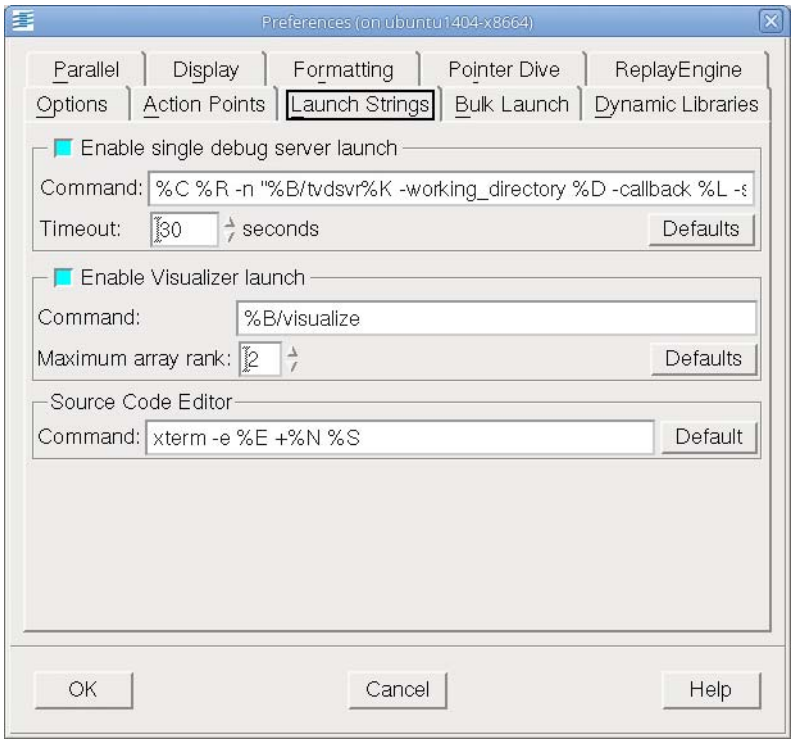
Figure 52, File > Preferences Dialog Box: Action Points Page



Launch Strings

This page sets the launch string that TotalView uses when it launches the **tvdsvr** remote debugging server, the Visualizer, and a source code editor. The initial values are the defaults

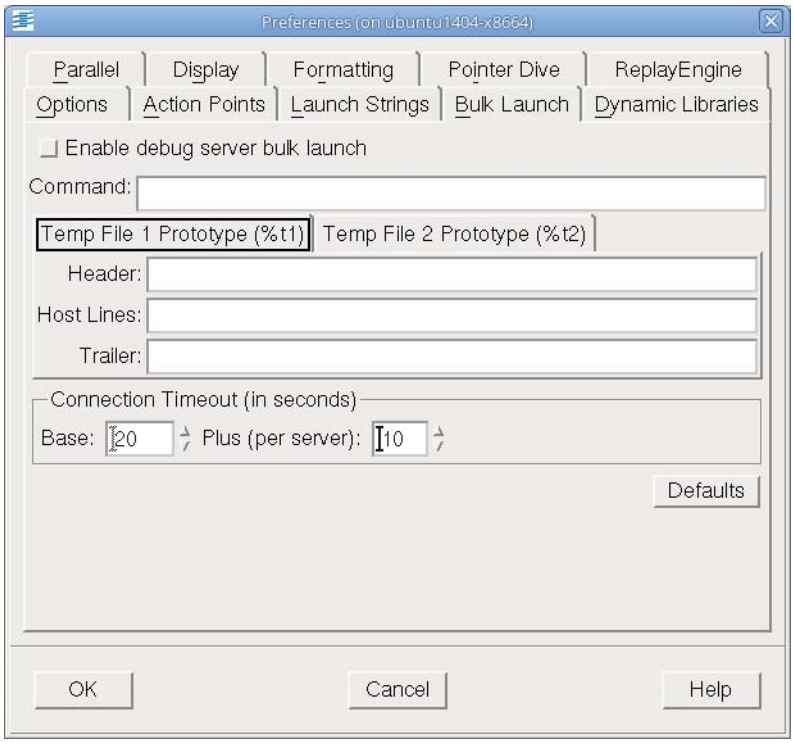
Figure 53, File > Preferences Dialog Box: Launch Strings Page



Bulk Launch

This page configures the TotalView bulk launch system which launches groups of processes simultaneously.

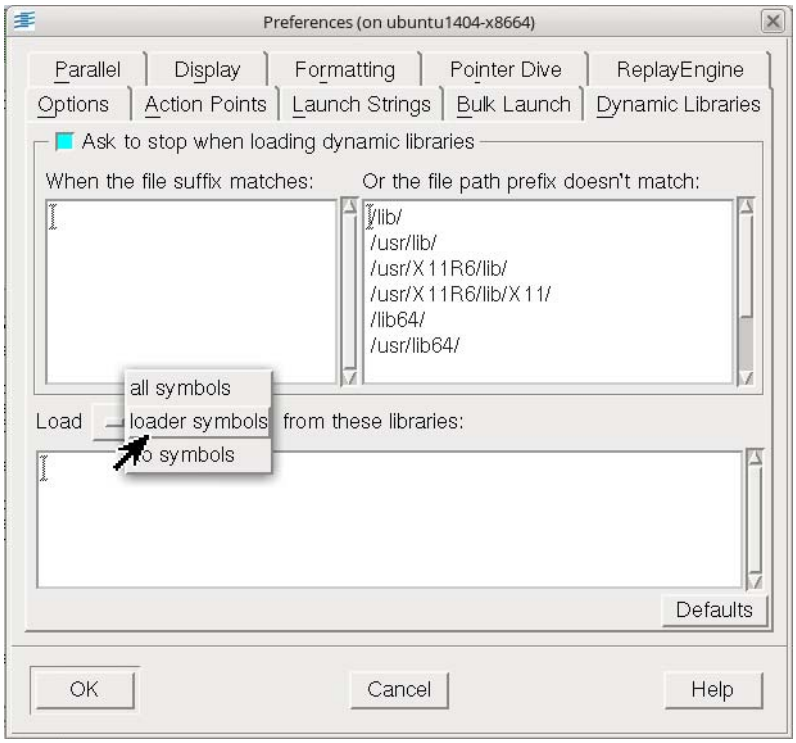
Figure 54, File > Preferences Dialog Box: Bulk Launch Page



Dynamic Libraries

When debugging large programs, you can sometimes increase performance by loading and processing debugging symbols. This page controls which symbols are added to TotalView when it loads a dynamic library, and how many of a library's symbols are read in.

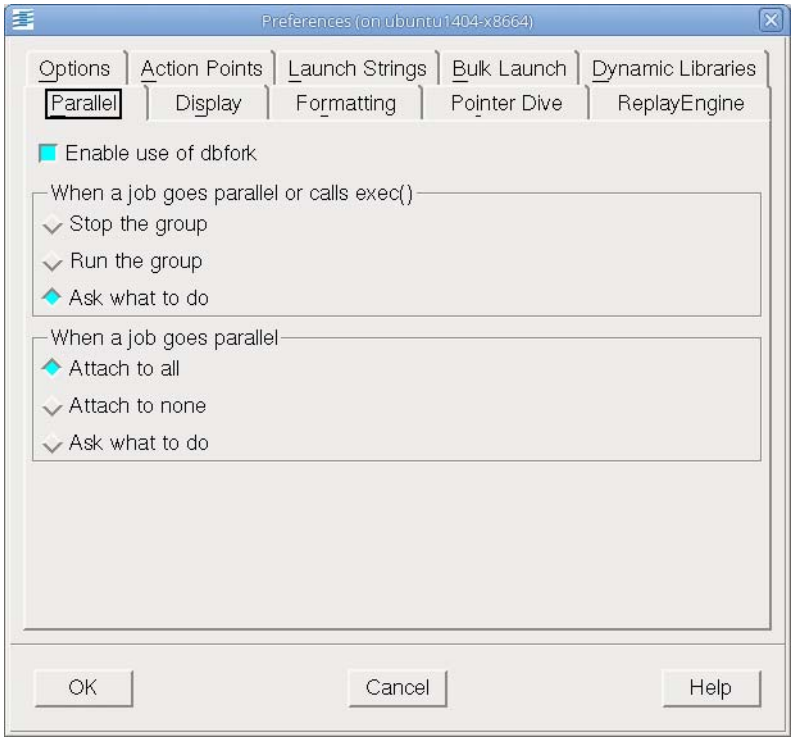
Figure 55, File > Preferences Dialog Box: Dynamic Libraries Page



Parallel

The options on this page control whether TotalView stops or continues executing when a process creates a thread or goes parallel. By stopping your program, you can set breakpoints and examine code before execution begins.

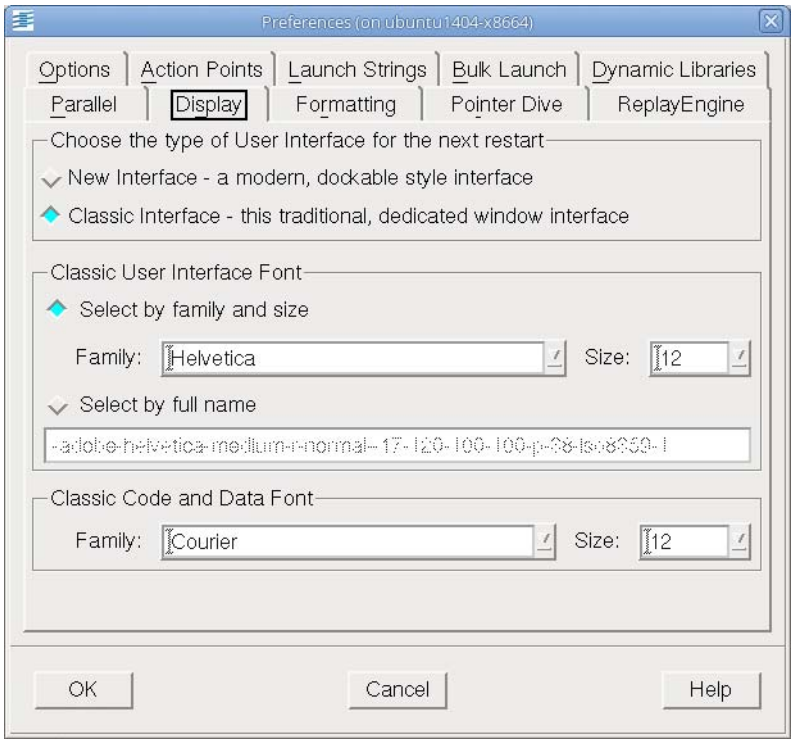
Figure 56, File > Preferences Dialog Box: Parallel Page



Display

This page specifies which user interface to use and which fonts are used in the classic user interface and how code is displayed.

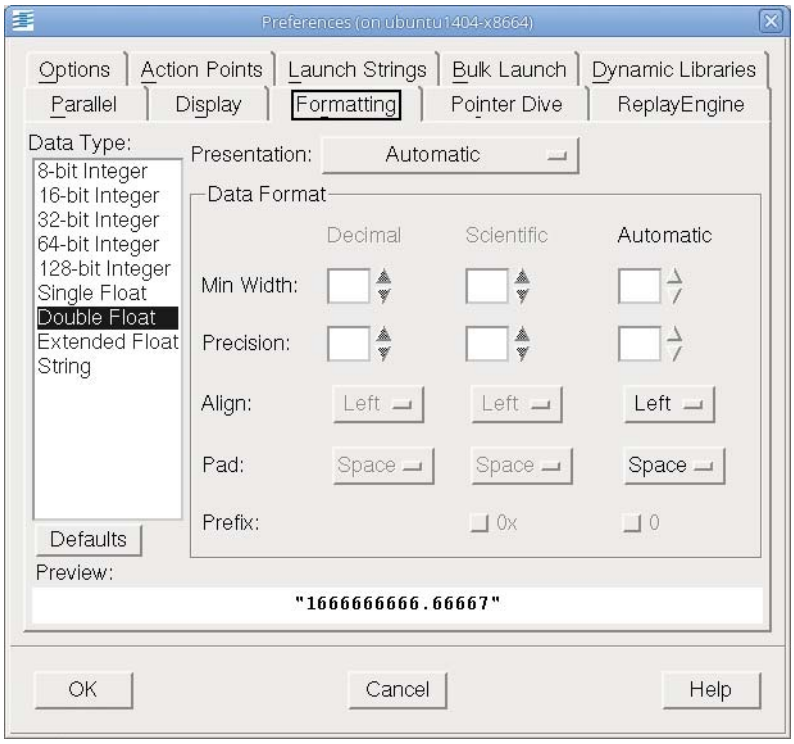
Figure 57, File > Preferences Dialog Box: Display Page



Formatting

This page controls how TotalView displays your program's variables.

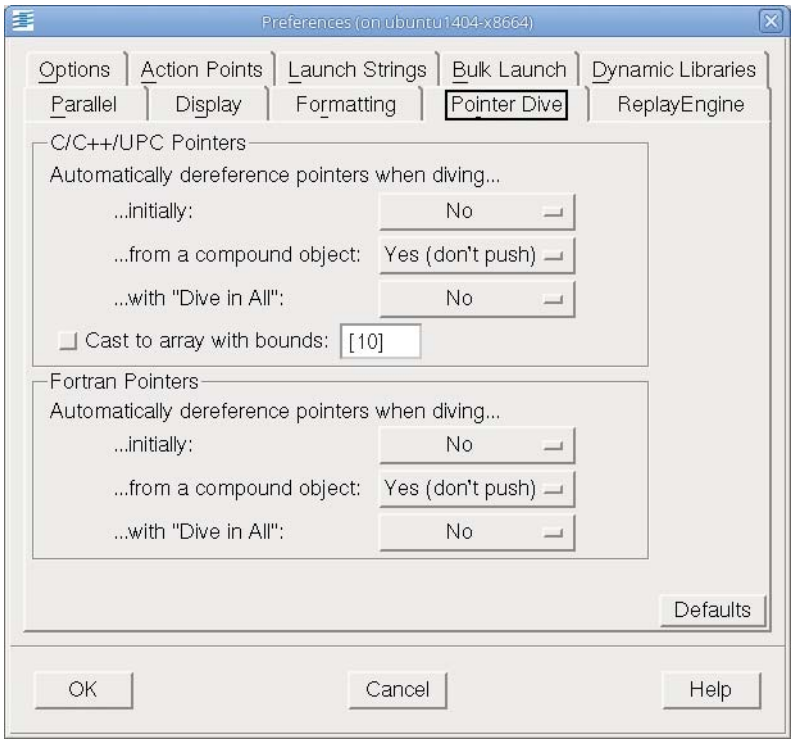
Figure 58, File > Preferences Dialog Box: Formatting Page



Pointer Dive

The options on this page control how TotalView dereferences pointers and casts pointers to arrays.

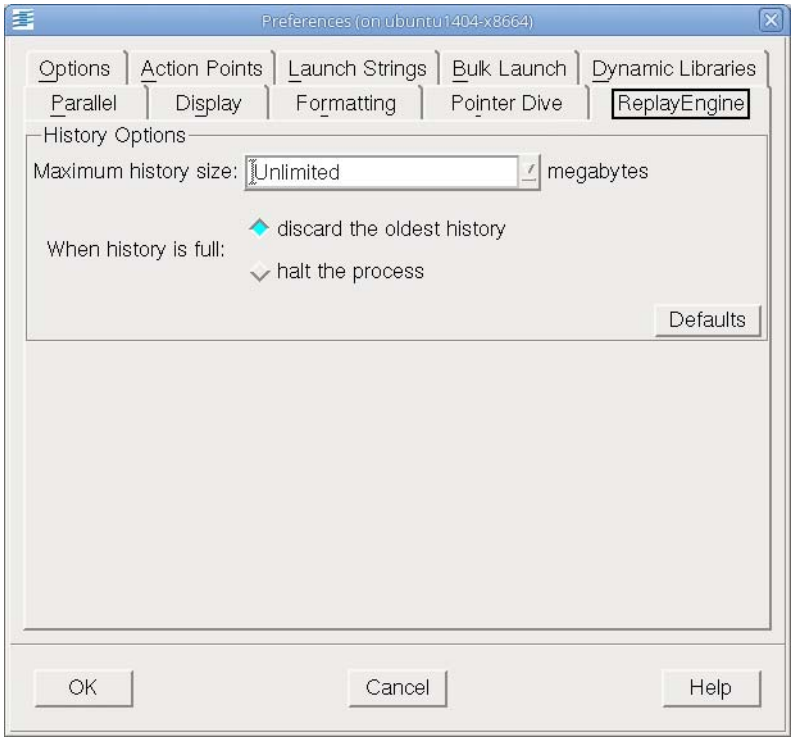
Figure 59, File > Preferences Dialog Box: Pointer Dive Page



ReplayEngine

This page controls how ReplayEngine handles recorded history.

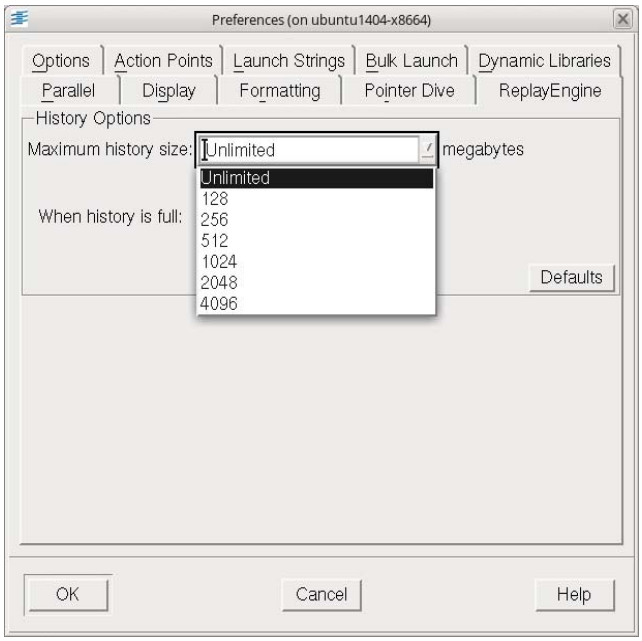
Figure 60, File > Preferences Dialog Box: ReplayEngine



The **Maximum history size** option sets the size in megabytes for ReplayEngine's history buffer. The default value, Unlimited, means ReplayEngine will use as much memory as is available to save recorded history. You can enter a new value into the text field or select from a drop-down list, as seen in Figure 61.

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

Figure 61, File > Preferences Dialog Box: ReplayEngine History Option



Setting Preferences, Options, and X Resources

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

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

NOTE: Deprecated means that while the feature still exists in the current release, there's no guarantee that it will continue to work at a later time. We have deprecated all "totalview" X default options. TotalView still fully supports Visualizer resources. Information on these Visualizer settings is in the document TotalView **XResources.pdf**, downloadable from the TotalView website at https://help.totalview.io/.

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

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

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

RELATED TOPICS	
Setting preferences in TotalView	Setting Preferences on page 133
TotalView variables	**"TotalView Variables"** in the *Classic TotalView Reference Guide*

Using and Customizing the GUI

This chapter contains information about using the TotalView GUI, including:

Using Mouse Buttons

The buttons on your three-button mouse work like this:

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.
Middle	Paste	Writes information previously copied or cut into the clipboard.	Move the cursor to the insertion point and click. Not all windows support pasting.
	Dive	Displays more information or replaces window contents.	Move the cursor over an object, then click.
Right	Context menu	Displays a menu with commonly used commands.	Move the cursor over an object and click. Most windows and panes have context menus; dialog boxes do not have context menus.

In most cases, a single-click selects an object while and a double-click dives on the object. However, if the field is editable, TotalView enters edit mode, so you can alter the selected item's value.

In some areas, such as the Stack Trace Pane, selecting a line performs an action. In this pane, TotalView dives on the selected routine. (In this case, *diving* means that TotalView finds the selected routine and shows it in the Source Pane.)

In the line number area of the Source Pane, a left mouse click sets a breakpoint at that line, displaying a  icon instead of a line number.

Selecting the  icon a second time deletes the breakpoint. If you change any of the breakpoint's properties or if you've created an eval point (indicated by an  icon), selecting the icon disables it. For more information on breakpoints and eval points, see Setting Action Points on page 188.

Using the Root Window

The Root Window appears when you start TotalView.

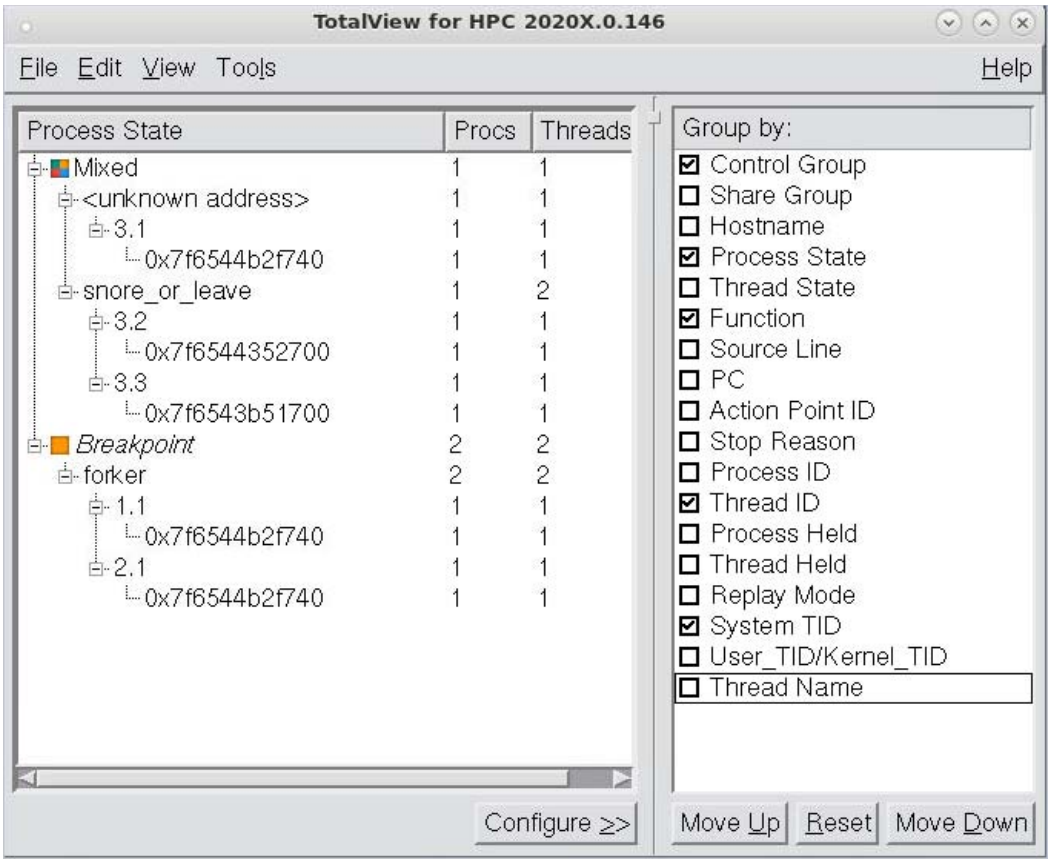
- If you type a program name immediately after the **totalview** command, TotalView also opens a Process Window with the program's source code.
- If you do not enter a program name when starting TotalView, TotalView displays its **File > New Debugging Session** dialog box. Use this dialog box to select the type of debug session you wish to configure.

The Root Window displays a list of aggregated processes and threads — instead of displaying one line per each process or thread, it groups them by common properties that you can configure. This provides a considerable performance boost when scaling to thousands — or more — threads or processes. The status of processes and threads is highlighted by colored icons for easy identification.

NOTE: To display the Root Window from versions prior to TotalView 8.15.0, see Using the Old Root Window. The old Root Window, however, does not perform as well for high-scale programs. See Scalability in HPC Computing Environments on page 604.

Figure 62 shows the Root Window for an executing multi-threaded, multi-process program.

Figure 62, Root Window



The Root Window groups threads and processes under common properties. The initial default view groups the display by control group, process state, function, and thread ID. You can regroup and reduce the display in a number of ways, based on either process or thread properties, as described in Table 1.

Table 1: Process and Thread Properties

Process	Property Level	Description
Control Group	Process	Control group of the processes in your job. Processes in the same job are placed in the same control group by default. If there is only one control group in the debug session, this property is omitted from the display.
Share Group	Process	Share group of the processes within a control group. Processes that are running the same main executable are placed in the same share group by default.

Table 1: Process and Thread Properties

Process	Property Level	Description
Hostname	Process	The hostname or IP address where the process is running.
Process State	Process	The process execution state, e.g., Nonexistent, Running, Stopped, Breakpoint, Watchpoint, etc. The process execution state derives from the execution state of the threads it contains.
Thread State	Thread	The thread execution state, e.g., Running, Stopped, Breakpoint, Watchpoint, etc.
Function	Thread	The function name of the location of the stopped thread. Displays the function name or "<unknown address>" if the thread is running or the function name is not known.
Source Line	Thread	The function name of the location of the stopped thread. Displays the source file name and line number or "<unknown line>" if the thread is running or the source line is not known.
PC	Thread	The PC of the location of the stopped thread. Displays the program counter value or "<unknown address>" if the thread is running.
Action Point ID	Thread	The action point (breakpoint or watchpoint) ID of the location of the stopped thread. Displays "Break (*ID*)", where ID is the action point ID or "none" if the thread is not stopped at an action point.
Stop Reason	Thread	If stopped, the reason, for example due to a breakpoint or barrier point.
Process ID	Process	The debugger process ID (dpid) of the process. Displays *dpid*.
Thread ID	Thread	The dpid and debugger thread ID (dtid) of the thread. Displays *dpid dtid*.
Process Held	Process	Indicates that a process is being held at a barrierpoint.
Thread Held	Thread	Indicates that a thread is being held at a barrierpoint.
Replay Mode	Process	Indicates that the process is in record mode.
System TID	Thread	The user thread ID (user TID) if it exists. Otherwise, the system kernel ID (Kernel TID).
User_TID/Kernel_TID	Thread	User thread ID / Kernel thread ID
Thread Name	Thread	The name of the thread, if set. If unset, “unnamed” is displayed.

The various properties here are either process or thread-level, which determines the kind of **ptlist** displayed in the Members column: the members of process-level properties are processes, and the members of thread-level properties are threads.

You can also use the CLI's **dstatus** command's `-group_by` switch for additional reduction options.

RELATED TOPICS

The CLI's **dstatus** command's `-group_by` switch **dstatus** in the *Reference Guide*

Controlling the Display of Processes and Threads

The Root Window's layout enables you to modify the grouping parameters while viewing the results.

Figure 63, Root Window and Process/Thread Display

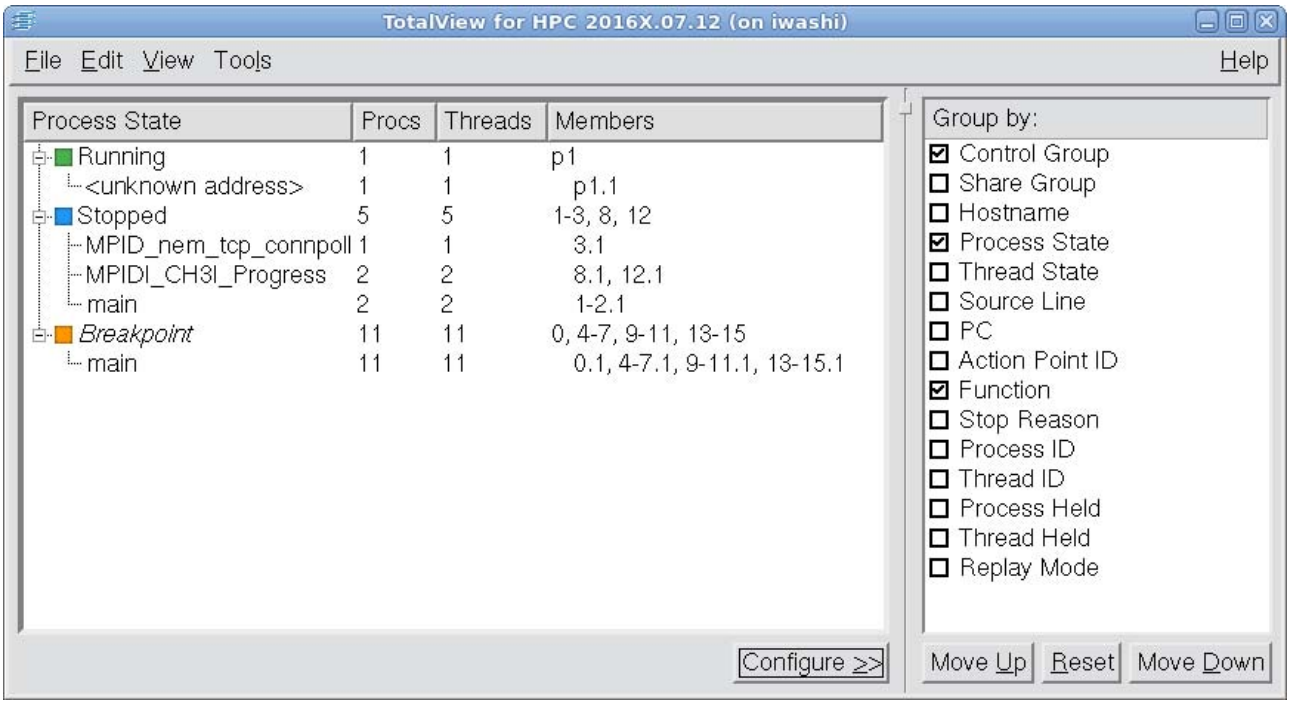


Figure 63 illustrates a 17-process MPI job comprised of a single MPI starter process, e.g., mpiexec (**p1**), and 16 MPI processes (0-15). At the highest level, processes are grouped by **Process State**, then by **Function**. The individual groupings are then sorted in ascending order (**Members**) by the process ID.

- **Configure**: Show or hide the configuration panel on the right using the **Configure** button or the **View > Show Configure Panel** menu item.

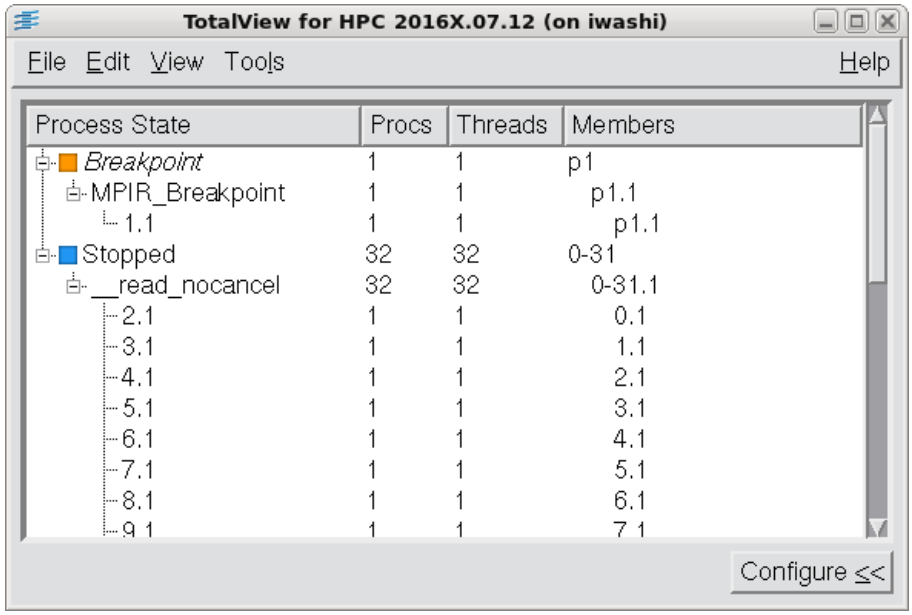
- **Move Up and Move Down**: By default, the properties are displayed in a hierarchical manner, such that the property at the top of the Configuration Panel forms the highest level grouping, the next property forms the second level of grouping subordinate to the first, and so on. Use **Move Up** and **Move Down** to control where to display a property in the hierarchy.
- **Reset**: The selected groupings and their relative order are automatically saved across TotalView sessions. To revert to the default order, press **Reset**.
- **Nested Attributes**: Instead of the default hierarchical display, the properties can be "flattened" into a single line in which each property is separated by a colon, using the **View > Nested Attribute** menu item. This menu item toggles between the hierarchical and flat display modes.
- **Show MPI Rank**: For MPI jobs, TotalView shows by default the rank in MPI_COMM_WORLD of MPI processes in the compressed **ptlist** in the Members column. Non-MPI processes are shown using the "**p***dpid*" where *dpid* is the debugger process ID of the process. Use the **View > Show MPI Rank** menu item to toggle between displaying MPI processes using the MPI rank or "**p***dpid*" notation.
- **Expand/Collapse All**: Minimize or fully expand the entire tree with **View > Expand All** and **View > Collapse All**.
- **Copy/Select All:** To copy data to an external program, use the clipboard: Select one or more rows using your computer's keyboard shortcuts, or select **Edit > Select All**, and copy to the clipboard using **Edit > Copy**.

Default View

By default, the Root Window displays Control Group, Process State, Function and Thread ID in a hierarchical or "nested" manner.

Figure 64 shows a 32-process, 32-thread MPI job, each process containing one thread, which is a single-threaded MPI starter process (e.g., mpiexec).

Figure 64, Root Window: Default View



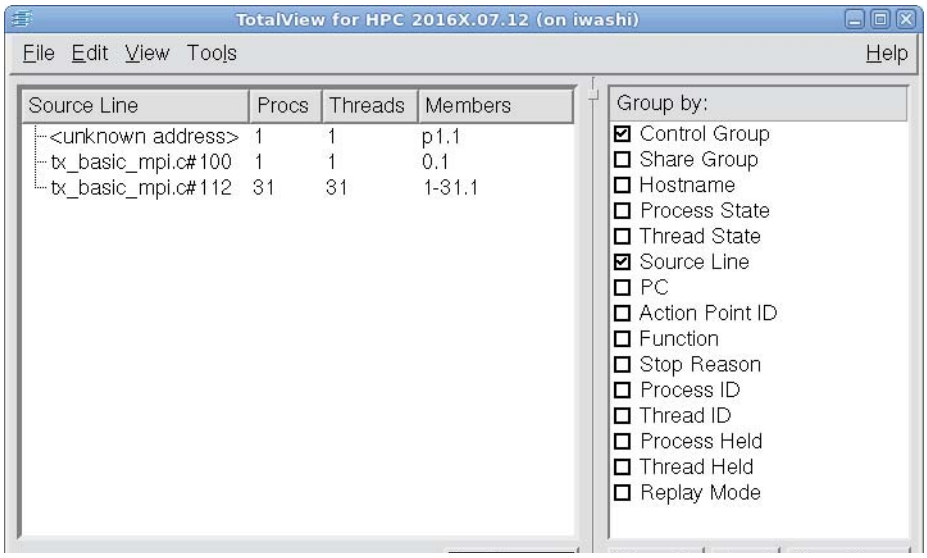
Since there is only one control group in the debug session, the control group has been omitted, and the Process state becomes the top-level property, in this case displaying two process states.

- **Breakpoint**: The first grouping lists the MPI starter process in **Breakpoint** process state, stopped in function **MPIR_Breakpoint**, and one thread with thread ID **1.1**. The **Procs** column shows the number of processes, and the **Threads** column shows the number of threads displayed in the Members column. The **Members** column shows the *dpid* the process and *dpid.dtid* of the threads, displayed as a compressed process/thread list **ptlist**.
- **Stopped**: This grouping displays all processes in a **Stopped** process state, of which there are 32, the MPI job size. The membership shows 0-31, which means MPI ranks 0 to 31, inclusive.

Changing the Display

To change the view, select the **Configure <<** button and select or de-select properties. Figure 65 shows a configuration where just the Source Line property is selected.

Figure 65, Root Window: Configure Pane, Group by Source Line



In this example, we've de-selected Process State, Function, and Thread ID and selected Source Line in order to group by Source Line instead.

Because Source Line is a thread property, the Members column now displays only threads. Here are the three lines:

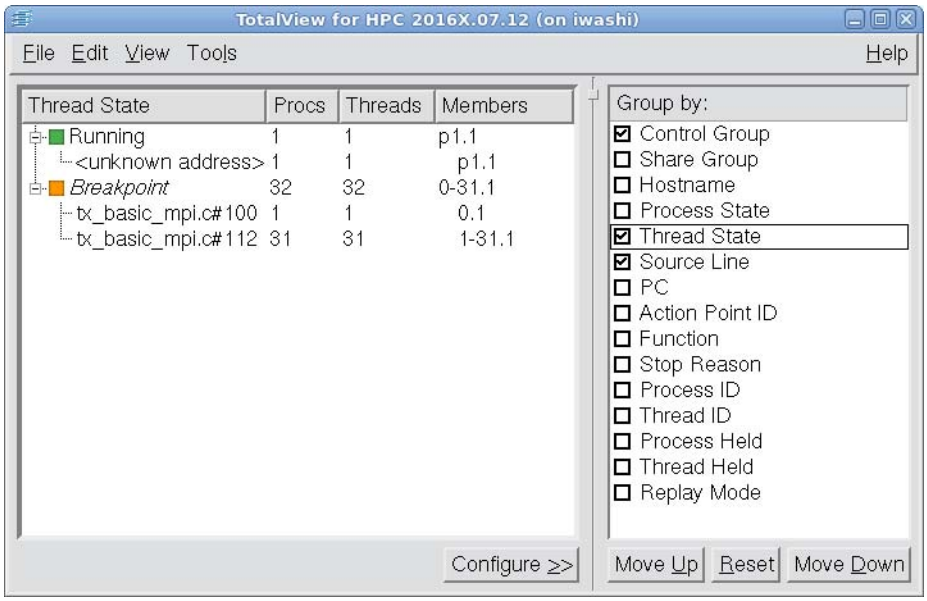
- Line 1 is the **mpirun** process used to launch the **tx_basic_mpi mpi** program.
- Line 2 displays all the threads at Line 100 in the source file **tx_basic_mpi.c**. The membership is 0.1, which means thread (dtid) 1 in MPI rank 0.
- Line 3 displays all the threads that are at line 112 inside that same file, representing the remaining 30 threads.

Note that even though the Control Group is selected, it has no grouping effect most of the time and is relevant only for debugging multiple jobs at once (which is uncommon), in which case the window would display a separate top-level control-group grouping for each job.

Grouping by Status and Source Line

If you select Thread State, leaving Source Line also selected, you can group by both properties, as in Figure 66.

Figure 66, Root Window: Grouped by Status *and* Source Line



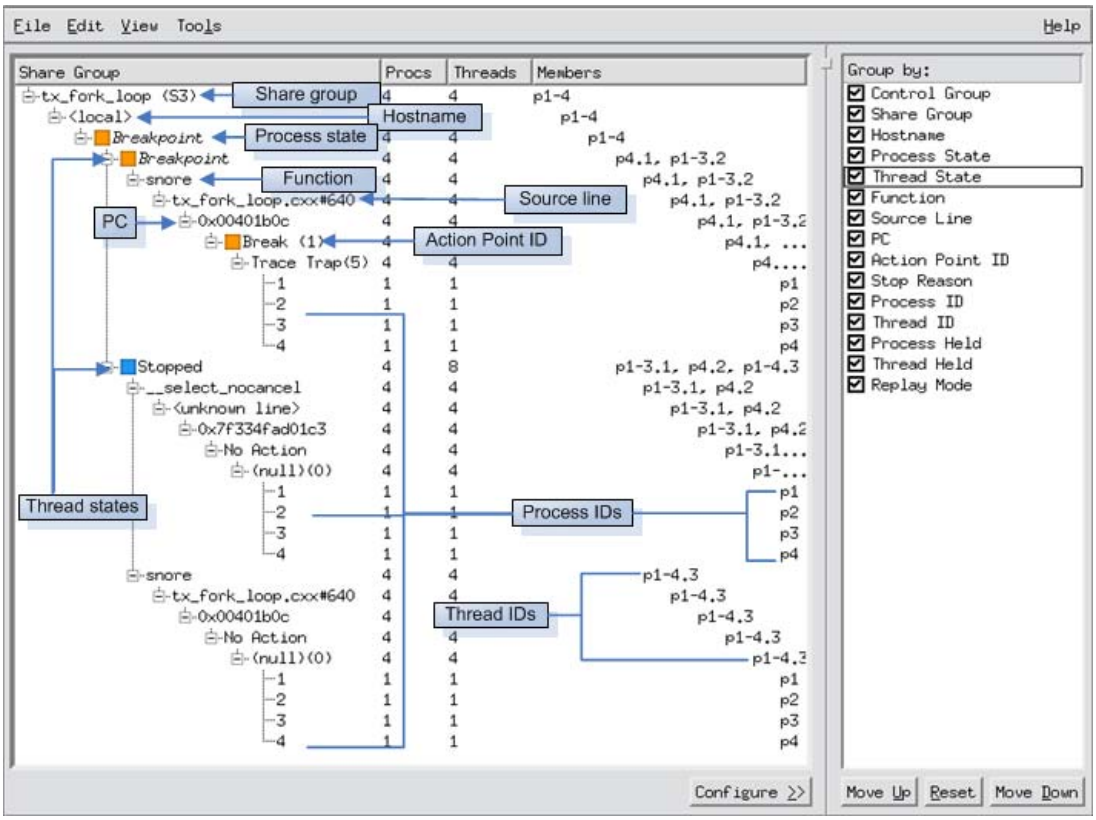
Again, there are two groupings in the list: the first is the **mpirun** process. Note that the second grouping is now multi-line:

- The first line of the second grouping displays a single thread, showing both that its status is at **Breakpoint** and that it is at line 100 in **tx_basic_mpi.c.**
- The second line displays 31 threads, all at a different **Breakpoint** at line 112 in **tx_basic_mpi.c**.

Grouping by All Properties

For illustration purposes, Figure 67 shows the Root Window configured to group by all properties in their default order. This figure also describes each of the groupings in this window.

Figure 67, Root Window Grouping by All Properties



When you dive on a line in this window, its source is displayed in a Process Window.

RELATED TOPICS	
The **dattach** command	The **dattach** command in the *Classic TotalView Reference Guide*
Displaying manager threads	The **View > Display Manager Threads** command in the in-product Help
Displaying exited threads	The **View > Display Exited Threads** command in the in-product Help

Using the Old Root Window

If debugger scalability is not a concern and you prefer the Root Window prior to version TotalView 8.15, you can reinstate it. Note, however, that the old Root Window is deprecated and may not be supported in future versions.

To start TotalView using the old Root Window, pass TotalView the **-oldroot** command option:

```
totalview  -oldroot
```

To always use the old Root Window by default, set the state variable `-TV::GUI::old_root_window` to **true** for use when initializing TotalView:

```
dset TV::GUI::old_root_window true
```

RELATED TOPICS	
Using the Root Window from previous TotalView versions	The **oldroot** command line option in the *Classic TotalView Reference Guide*

Suppressing the Root Window

To suppress opening the Root Window entirely when TotalView starts — whether the current or old Root Window — use the **--norootwin** option:

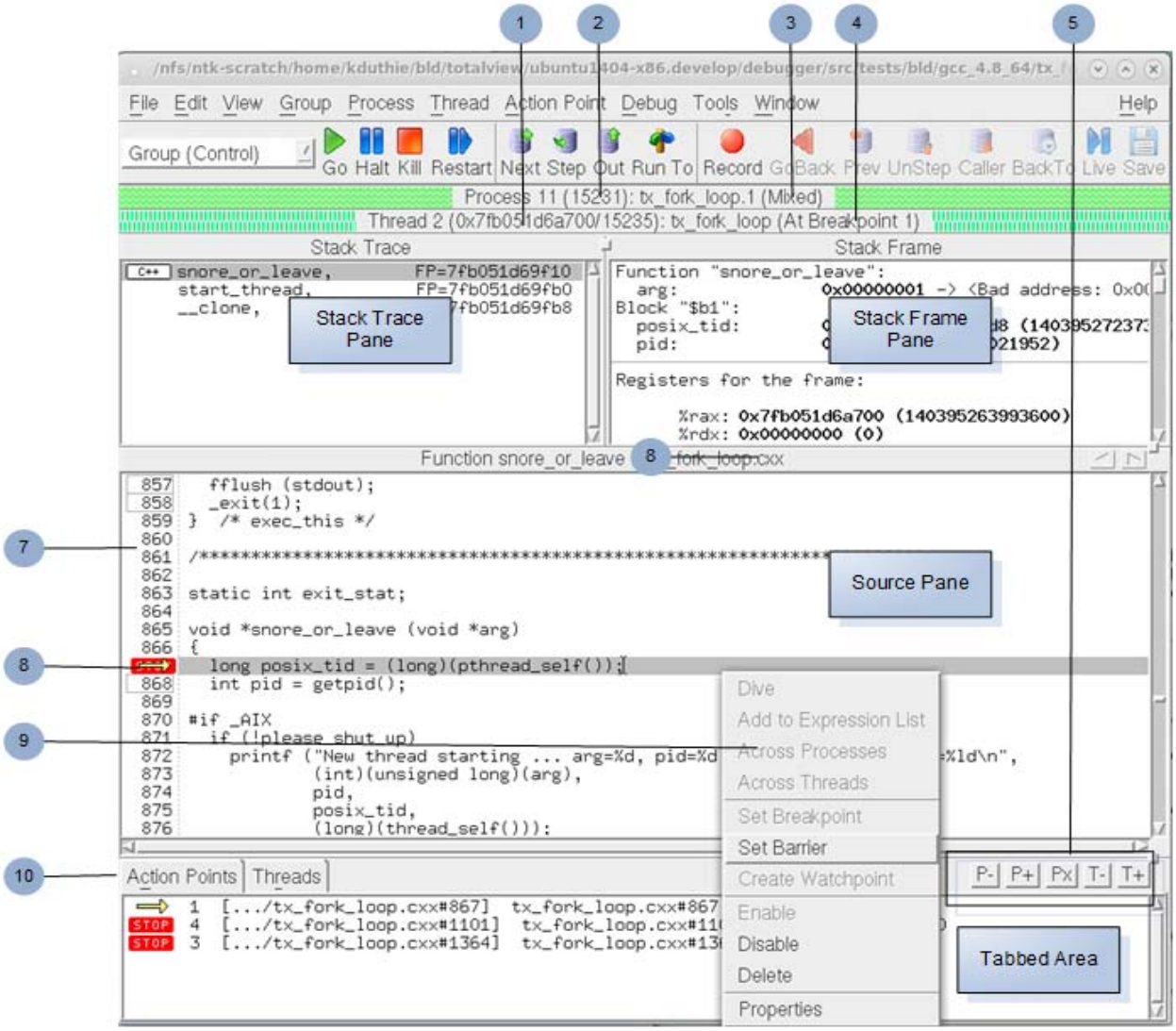
```
totalview  -norootwin
```

Note that you can open the Root Window at any time from the Process Window using the **Window > Root** menu item. The Root Window can also be closed at any time without quitting TotalView, if another window is open in the same debugging session.

Using the Process Window

The *Process Window* contains panes that display the code for the process or thread that you're debugging, as well as other related information. In the middle is the Source Pane. (The contents of these panes are discussed later in this section.

Figure 68, A Process Window



1. Thread ID (TID) / Kernel ID (KID)
2. Process ID (PID)
3. Process status
4. Thread status
5. Process/thread switching
6. Language of routine
7. Line number area
8. Current program counter
9. Context menu
10. Action Points tab displayed

As you examine the Process Window, notice the following:

- The thread ID shown in the Root Window and in the process's Threads Tab with the Tabs Pane is the logical thread ID (TID) assigned by TotalView and the system-assigned thread ID (SYSTID). On systems where the TID and SYSTID values are the same, TotalView displays only the TID value.

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

  The Threads Tab shows the threads that currently exist in a process. When you select a different thread in this list, TotalView updates the Stack Trace, Stack Frame, and Source 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 that displays information for that thread.

- The Stack Trace Pane shows the call stack of routines that the selected thread is executing. You can move up and down the call stack by clicking on the routine's name (stack frame). When you select a different stack frame, TotalView updates the Stack Frame and Source Pane with the information about the routine you just selected.

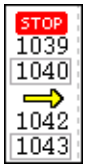
- The Stack Frame Pane displays all a routine's parameters, its local variables, and the registers for the selected stack frame.

- The information displayed in the Stack Trace and Stack Frame Panes reflects the state of a process when it was last stopped. Consequently, the information is not up-to-date while a thread is running.

- The left margin of the *Source Pane* displays line numbers and action point icons.

  You can place a breakpoint at any line whose line number is contained within a box indicating that this is executable code. You can also click on a line without a breakpoint box to create a sliding or pending breakpoint. (See Sliding Breakpoints and Pending Breakpoints).
  The Source Pane display unifies source-line breakpoint boxes across all image files within the share group that contain the source file. For example, the source-line breakpoint boxes will be unified for all instances of a source file or header file that is compiled into multiple shared libraries used by the process.

  When you place a breakpoint on a line, TotalView places a STOP icon over the line number. An arrow over the line number shows the current location of the program counter (PC) in the selected stack frame.

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

- The tabbed area at the bottom contains a set of tabs whose information you can hide or display as needed. In addition, the **P+**, **P-**, **Px**, **T+**, and **T-** buttons within this area allow you to change the Process Window's context by moving to another process or thread.

The *Action Points Tab* with the Tabs Pane shows the list of breakpoints, eval points, and watchpoints for the process.

The *Threads Tab* shows each thread and information about the thread. Selecting a process switches the context to that thread.

The *Processes/Ranks tab*, if present, displays a grid of all of your program's processes. The grid's elements show process status and indicate the selected group. Selecting a process switches the context to the first thread in that process.

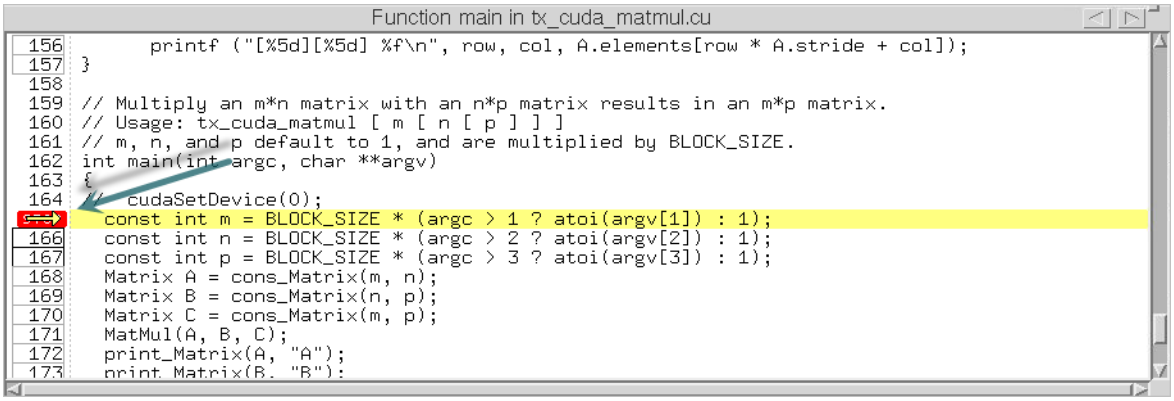
The Processes/Ranks Tab was displayed by default in previous versions of TotalView, but now it is off by default. This is because it can significantly affect performance, particularly for large, massively parallel applications. The tab can be turned back on with the command line switch **-processgrid** and/or by setting **TV::GUI::process_grid_wanted** to **true** in the `.tvdrc` file. If you enable this tab in the `.tvdrc` file, you can disable it for a particular session with the **-noprocessgrid** command line switch.

RELATED TOPICS	
More on using the Process Window	The **"Process Window"** in the in-product Help
More on the Processes/Ranks tab	The topic **Using the Processes/Ranks and Threads Tabs** in this user guide.

The Source Pane

The Source Pane located in the center of the Process Window displays the source code for source or header files that are compiled into the image files (e.g., executable or shared libraries) loaded into the process. The arrow in the left margin of the Source Pane indicates the location of the PC for that stack frame.

Figure 69, The Source Pane



Unified Source Pane Display

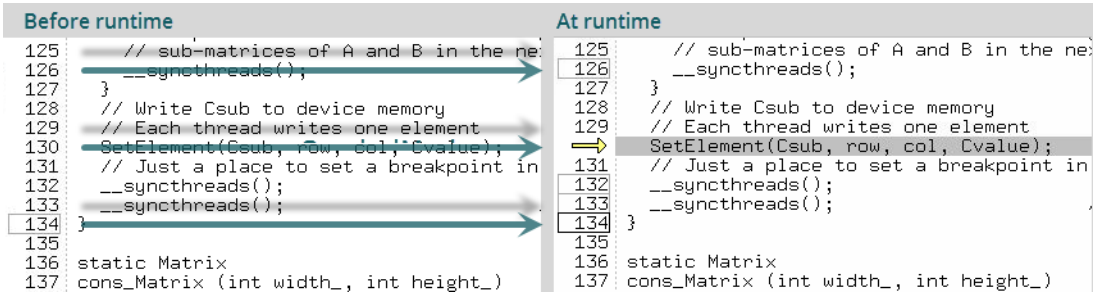
The Source Pane provides a unified view of source-line breakpoint boxes across all image files containing the source file, useful for programs in which the same source file or header file is compiled into multiple image files (e.g., executable and shared library files) used by the process.

To see how this works, remember that breakpoint boxes appear on lines where TotalView has identified executable code, i.e., source code lines where the compiler has generated one or more line number symbols in the debug information. **A gray box** denotes that there is exactly one line number symbol for that source line, while a **black box** denotes multiple line number symbols for that line.

For example, consider debugging a program that launches CUDA code running on a Graphics Processing Unit (GPU). When the host program is first loaded into TotalView, the CUDA threads have not yet launched, so the debugger has no symbol table information yet.

Figure 70 shows the Source Pane before and after a CUDA kernel launch. Before the CUDA threads exist (the left pane), lines **126**, **130**, **132**, and **133** have no boxed lines, meaning that TotalView can find no line number symbols associated with those lines. Additionally, line **134** has a gray line, meaning that TotalView has identified one line number symbol associated with the host (CPU code).

Figure 70, Unified Source Pane



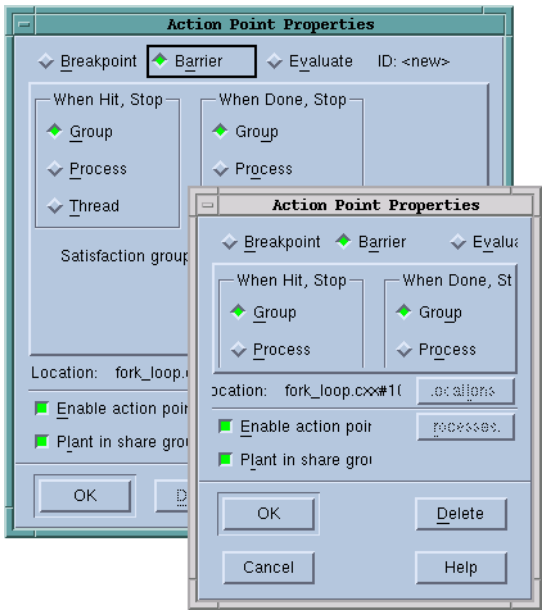
Once the program is running and the CUDA threads have started (the right pane), lines **126**, **132**, and **133** have gray lines, so now TotalView has been able to identify line number symbols at those locations. This is also true of line 130 which is obscured in the right pane by the PC icon. Further, line 134 has turned from gray to black, denoting that TotalView has found additional line number symbols, in this case one from the host (CPU) code and one from the CUDA (GPU) code.

RELATED TOPICS	
Setting breakpoints in CUDA code	Unified Source Pane and Breakpoint Display on page 634
Dynamically loading shared libraries using the **ddlopen** command	**ddlopen** in the TotalView Reference Guide
More on setting breakpoints	Setting Breakpoints and Barriers on page 194

Resizing and Positioning Windows

You can resize most TotalView windows and dialog boxes. While TotalView tries to do the right thing, you can push things to the point where shrinking doesn't work very well. Figure 71 shows a before-and-after look in which a dialog box was made too small.

Figure 71, Resizing (and Its Consequences)



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

- **Window > Memorize**: Tells TotalView to remember the position of the current window. The next time you bring up this window, it'll be in this position.
- **Window > Memorize All**: Tells TotalView to remember the positions of most windows. The next time you bring up any of the windows displayed when you used this command, it will be in the same position.

Most modern window managers such as KDE or Gnome do an excellent job managing window position. If you are using an older window manager such as **twm** or **mwm**, you may want to select the **Force window positions (disables window manager placement modes)** check box option located on the **Options** Page of the **File > Preferences** Dialog Box. This tells TotalView to manage a window's position and size. If it isn't selected, TotalView only manages a window's size.

About Diving into Objects

Diving is integral to the TotalView GUI and provides a quick and easy way to get more information about variables, processes, threads, functions, and other program elements.

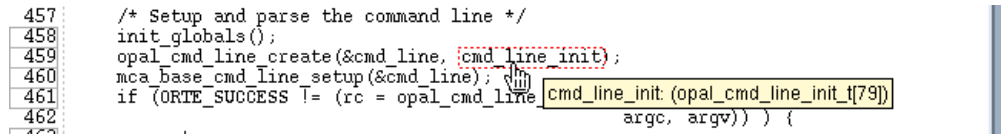
To dive on an element, just click your middle mouse button on it to launch another window with more information.

NOTE: In some cases, single-clicking performs a dive. For example, single-clicking on a function name in the Stack Trace Pane dives into the function. In other cases, double-clicking does the same thing.

Diving on processes and threads in the Root Window is the quickest way to launch a Process Window with more information. Diving on variables in the Process Window launches a Variable Window.

In the Process Window's Source Pane, if a global variable or function can be dived on, a red dotted box appears when your cursor hovers over it, Figure 72.

Figure 72, Diving on an object in the Source Pane



NOTE: If you prefer that the cursor remain an arrow when hovering over an element you can dive on, specify the option -nohand_cursor when starting TotalView, or set this permanently in .tvdrc as "TV::GUI::hand_cursor_enabled {false}".

The following table describes typical diving operations:

Items you dive on:	Information Displayed:
Process or thread	When you dive on a thread in the Root Window, TotalView finds or opens a Process Window for that process. If it doesn't find a matching window, TotalView replaces the contents of an existing window and shows you the selected process.
Variable	The variable displays in a Variable Window.

Items you dive on:	Information Displayed:
Expression List Variable	Same as diving on a variable in the Source Pane: the variable displays in a Variable Window.
Routine in the Stack Trace Pane	The stack frame and source code for the routine appear in a Process Window.
Array element, structure element, or referenced memory area	The contents of the element or memory area replace the contents that were in the Variable Window. This is known as a *nested* dive.
Pointer	TotalView dereferences the pointer and shows the result in a separate Variable Window. Given the nature of pointers, you may need to cast the result into the logical data type.
Subroutine	The source code for the routine replaces the current contents of the Source Pane. When this occurs, TotalView places a right angle bracket (>) in the process's title. Every time it dives, it adds another angle bracket, Figure 73.
	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 Window	TotalView replaces the contents of the Variable Window with information about the variable or element.
Expression List Window	TotalView displays information about the variable in a separate Variable Window.

Figure 73, Nested Dive

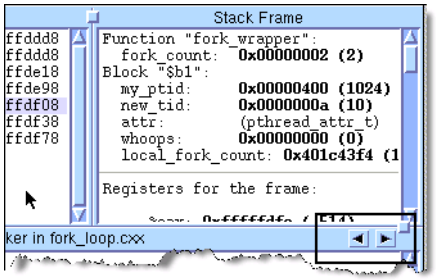
NOTE: Diving on a struct or class member that is out of scope does not work.

TotalView tries to reuse windows. For example, if you dive on a variable and that variable is already displayed in a window, TotalView pops the window to the top of the display. If you want the information to appear in a separate window, use the **View > Dive in New Window** command.

NOTE: Diving on a process or a thread might not create a new window if TotalView determines that it can reuse a Process Window. If you really want to see information in two windows, use the Process Window **Window > Duplicate** command.

When you dive into functions in the Process Window, or when you are chasing pointers or following structure elements in the Variable Window, you can move back and forth between your selections by using the *forward* and *backward* buttons. The boxed area of the following figure shows the location of these two controls.

Figure 74, Backward and Forward Buttons



For additional information about displaying variable contents, see Diving in Variable Windows on page 266.

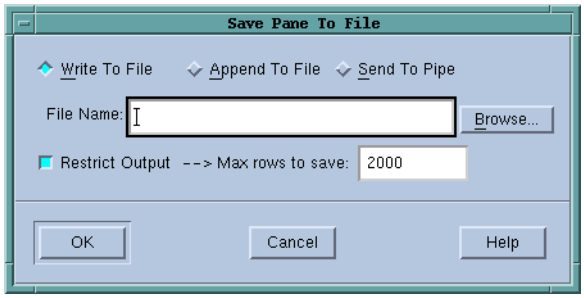
You can also use the following additional windowing commands:

- **Window > Duplicate**: (Variable and Expression List Windows) Creates a duplicate copy of the current Variable Window.
- **File > Close**: Closes an open window.
- **File > Close Relatives**: Closes windows that are related to the current window. The current window isn't closed.
- **File > Close Simila**r: Closes the current window and all windows similar to it.

Saving the Data in a Window

You can write an ASCII equivalent to most pages and panes by using the **File > Save Pane** command. This command can also pipe data to UNIX shell commands.

Figure 75, File > Save Pane Dialog Box



If the window or pane contains a lot of data, use the **Restrict Output** option to limit the information TotalView writes or sends. For example, you might not want to write a 100 x 100 x 10,000 array to disk. If this option is checked (the default), TotalView sends only the number of lines indicated in the **Max rows to save** box.

When piping information, TotalView sends the entered information to **/bin/sh**. This means that you can enter a series of shell commands. For example, the following is a command that ignores the top five lines of output, compares the current ASCII text to an existing file, and writes the differences to another file:

```
| tail +5 | diff - file > file.diff
```

Searching and Navigating 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 using the **Edit > Find** command, which launches the find dialog box.

Figure 76, Edit > Find Dialog Box

Controls in this dialog box let you:

- Perform case-sensitive searches.
- Wrap searches back to the beginning of the file.
- Keep the dialog box displayed.
- Search down or up.

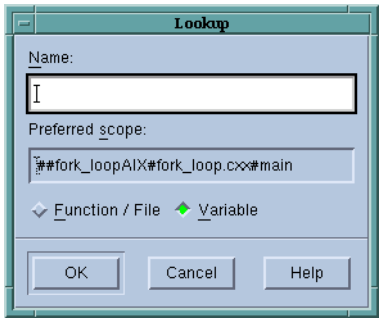
After you have found a string, you can find another instance of it by using the **Edit > Find Again** command.

If you searched for the same string previously, you can select it from the pulldown list on the right side of the **Find** text box.

Looking for Functions and Variables

Having TotalView locate a variable or a function is usually easier than scrolling through your sources to look for it. Do this with the **View > Lookup Function** and **View > Lookup Variable** commands. Here is the dialog set to look

Figure 77, View > Lookup Variable Dialog Box

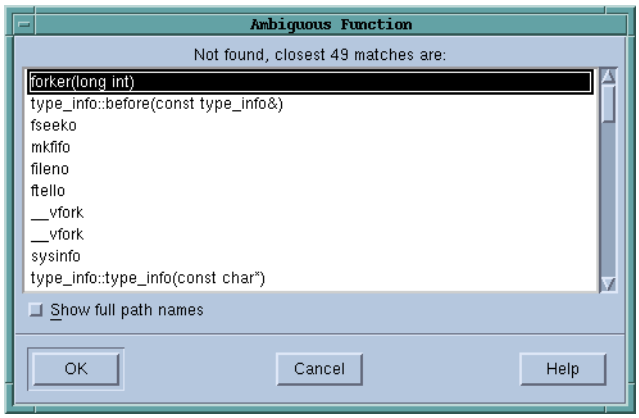


CLI: dprint *variable*

for variables:

If TotalView doesn't find the name and it can find something similar, it displays a dialog box that contains the names of functions that might match.

Figure 78, Ambiguous Function Dialog Box



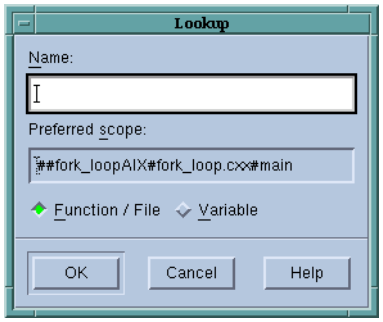
If the one you want is listed, click on its name and then choose **OK** to display it in the Source Pane.

Finding the Source Code for Functions

Use the **File > Open Source** command to search for a function's declaration.

CLI: dlist *function-name*

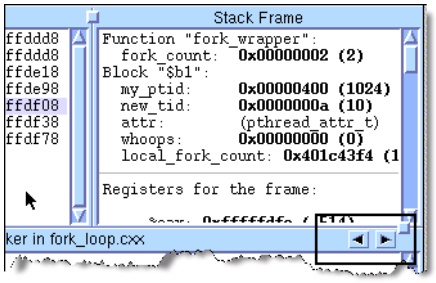
Figure 79, View > Lookup Function Dialog Box



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

When you want to return to the previous contents of the Source Pane, use the Backward button located in the upper-right corner of the Source Pane and just below the Stack Frame Pane. In Figure 80, a rectangle surrounds this button.

Figure 80, Undive/Dive Controls



You can also use the **View > Reset** command to discard the dive stack so that the Source Pane is displaying the PC it displayed when you last stopped execution.

Another method of locating a function's source code is to dive into a source statement in the Source Pane that shows the function being called. After diving, you see the source.

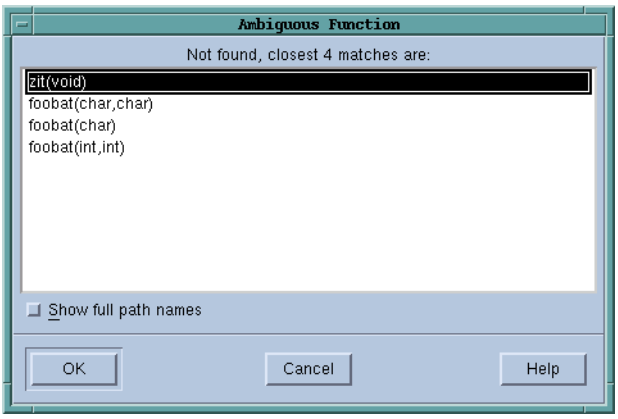
Resolving Ambiguous Names

Sometimes the function name you specify is ambiguous. For example, you might have specified the name of:

- A static function, and your program contains different versions of it.
- A member function in a C++ program, and multiple classes have a member function with that name.
- An overloaded function or a template function.

The following figure shows the dialog box that TotalView displays when it encounters an ambiguous function name. You can resolve the ambiguity by clicking the function name.

Figure 81, Ambiguous Function Dialog Box



If the name being displayed isn't enough to identify which name you need to select, select the **Show full path names** check box to display additional information.

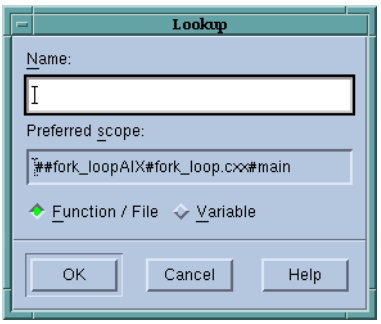
RELATED TOPICS

Using C++ in TotalView	Using C++ on page 364

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 82

Figure 82, View > Lookup Function Dialog Box



If a header file contains source lines that produce executable code, you can display the file's code by typing the file name here.

Resetting the Stack Frame

After moving around your source code to look at what's happening in different places, you can return to the executing line of code for the current stack frame by selecting the **View > Reset** command. This command places the PC arrow on the screen.

This command is also useful when you want to undo the effect of scrolling, or when you move to different locations using, for example, the **View > Lookup Function** command.

If the program hasn't started running, the **View > Reset** command displays the first executable line in your main program. This is useful when you are looking at your source code and want to get back to the first statement that your program executes.

Viewing the Assembler Version of Your Code

You can display your program in source or assembler using these commands:

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

Assembler code
: Select the **View > Source As > Assembler** command.

Both Source and assembler
: Select the **View > Source As > Both** command.

  The Source Pane divides into two parts. The left pane contains the program's source code and the right contains the assembler version. You can set breakpoints in either of these panes. Setting an action point at the first instruction after a source statement is the same as setting it at that source statement.

The following commands display your assembler code by using symbolic or absolute addresses:

Command	Display
View > Assembler > By Address	Absolute addresses for locations and references (default)
View > Assembler > Symbolically	Symbolic addresses (function names and offsets) for locations and references

NOTE: You can also display assembler instructions in a Variable Window. For more information, see Displaying Machine Instructions on page 263.

The following three figures illustrate the different ways TotalView can display assembler code. In Figure 83, 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 83, Address Only (Absolute Addresses)

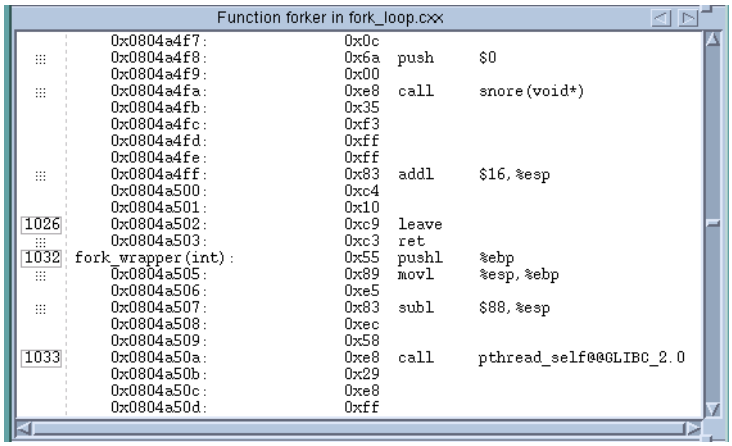


Figure 84 displays information symbolically. The second column shows locations using functions and offsets.

Figure 84, Assembly Only (Symbolic Addresses)

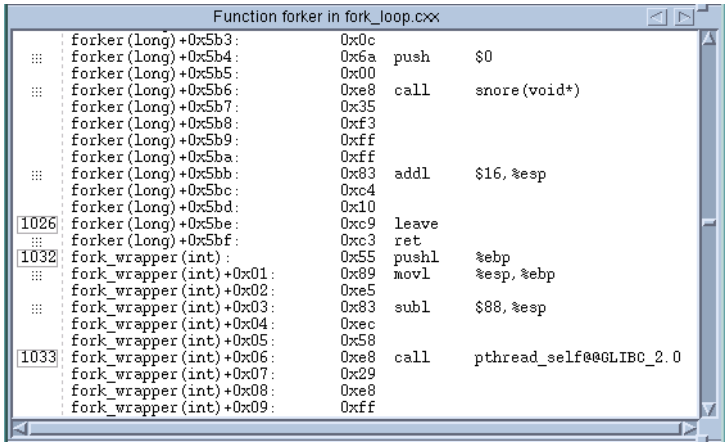
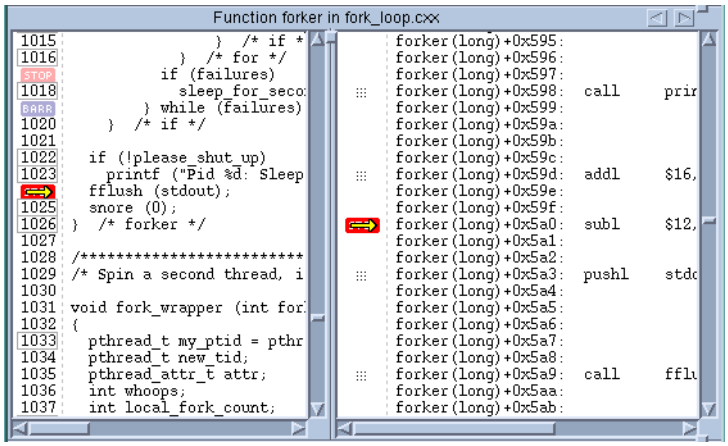


Figure 85 displays the split Source Pane, with the program's source code on the left and assembler version on the right. In this example, the assembler is shown symbolically (by selecting **View > Assembler > Symbolically**).

Figure 85, Both Source and Assembler (Symbolic Addresses)



NOTE: When TotalView displays instructions, the arguments are almost always in the following order: “source,destination”. On ~~Linux x86 (32 bit) and~~ Linux x86-64 platforms, this can be confusing as the order indicated in AMD and Intel technical literature indicates that the order is usually “destination,source”. The order in which TotalView displays this information conforms to the GNU assembler. This ordering is usually an issue only when you are examining a core dump.

RELATED TOPICS

Machine instructions	Displaying Machine Instructions on page 263
Memory with an unknown data type	Viewing Areas of Memory ($void Data Type) on page 292
Viewing the contents of a location as machine instruction	Viewing Instructions ($code Data Type) on page 292

Editing Source Text

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

The default editor is **vi**. However, TotalView uses the editor named in an **EDITOR** environment variable, or the editor you name in the Source Code Editor field of the **File > Preferences** Launch Strings Page. The online Help for this page contains information on setting this preference.

Stepping through and Executing your Program

This chapter discusses stepping and program execution, including these sections:

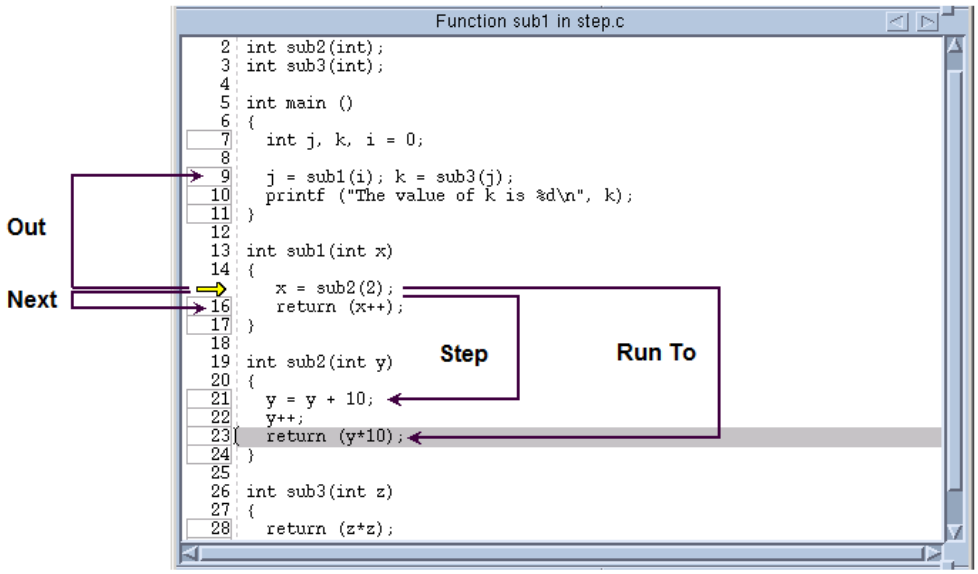
Using Stepping Commands

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

Figure 86 illustrates stepping commands.

Figure 86, Stepping Illustrated



The arrow indicates that the PC is at line 15. The four stepping commands do the following:

- **Next** executes line 15. After stepping, the PC is at line 16.
- **Step** moves into the **sub2()** function. The PC is at line 21.
- **Run To** executes all lines until the PC reaches the selected line, which is line 23.
- **Out** executes all statements within **sub1()** and exits from the function. The PC is at line 9. If you now execute a Step command, TotalView steps into **sub3()**.

Remember the following things about single-stepping commands:

- To cancel a single-step command, select **Group > Halt** or **Process > Halt**.

  CLI: dhalt

- If your program reaches a breakpoint while stepping over a function, TotalView cancels the operation and your program stops at the breakpoint.
- If you enter a source-line stepping command and the primary thread is executing in a function that has no source-line information, TotalView performs an assembler-level stepping command.
- When TotalView steps through your code, it steps one line at a time. This means that if you have more than one statement on a line, a step instruction executes all of the instructions on that line.
- To modify the way source-level single stepping works, use the **dskip** command to create and manage single-stepper "skip" rules. You can add rules that match a function, all functions in a source file, or a specific function in a specific source file.

RELATED TOPICS	
The **dfocus** command	The dfocus command in "CLI Commands" in the Classic TotalView Reference Guide
Detailed discussion on stepping	Stepping (Part I) on page 575, with examples in Stepping (Part II): Examples on page 598
The **dskip** command	The command in "CLI Commands" in the Classic TotalView Reference Guide

Stepping into Function Calls

The stepping commands execute one line in your program. If you are using the CLI, you can use a numeric argument that indicates how many source lines TotalView steps. For example, here's the CLI instruction for stepping three lines:

```
dstep 3
```

If the source line or instruction contains a function call, TotalView steps into it. If TotalView can't find the source code and the function was compiled with **-g**, it displays the function's machine instructions.

However, if a skip rule has been defined for a function call using the **dskip** command, it can change this stepping behavior by:

- Causing the step into commands to step over a call to a function that contains source-line information.
- Causing the step command to step through a function while ignoring any source-line information. This applies to both step into and step over commands.

You might not realize that your program is calling a function. For example, if you overloaded an operator, you'll step into the code that defines the overloaded operator.

NOTE: If the function being stepped into wasn't compiled with the -g command-line option, TotalView always steps over the function.

The GUI has eight **Step** commands and eight **Step Instruction** commands. These commands are located on the **Group**, **Process**, and **Thread** pulldowns. The difference between them is the focus.

CLI: **dfocus ... dstep**
dfocus ... dstepi

RELATED TOPICS	
The **dstep** command	**dstep** in "CLI Commands" in the Classic TotalView Reference Guide
The **dstepi** command	**dstepi** in "CLI Commands" in the Classic TotalView Reference Guide
Detailed discussion on skipping function calls	Skipping Function Calls on page 181

Stepping Over Function Calls

When you step over a function, TotalView stops execution when the primary thread returns from the function and reaches the source line or instruction after the function call.

The GUI has eight **Next** commands that execute a single source line while stepping over functions, and eight **Next Instruction** commands that execute a single machine instruction while stepping over functions. These commands are on the **Group**, **Process**, and **Thread** menus.

CLI: **dfocus ... dnext**
dfocus ... dnexti

If the PC is in assembler code—this can happen, for example, if you halt your program while it's executing in a library—a **Next** operation executes the next instruction. If you want to execute out of the assembler code so you're back in your code, select the **Out** command. You might need to select **Out** a couple of times until you're back to where you want to be.

RELATED TOPICS	
The **dnext** command	**dnext** in "CLI Commands" in the Classic TotalView Reference Guide
The **dnexti** command	**dnexti** in "CLI Commands" in the Classic TotalView Reference Guide

Skipping Function Calls

You can define skip rules that allow you to identify functions that you are not interested in debugging. The TotalView **dskip** command allows you to create and manage single-stepper skip rules that modify the way source-level single stepping works.

You can add rules that match a function, all functions in a source file, or a specific function in a specific source file. Functions can be matched by the function name or a regular expression (Tcl "regexp"). Files can be matched by the file name or a glob pattern (Tcl "string match").

TotalView implements two skip rule variants, **over** and **through**, as follows:

1. A matching and enabled skip **over** rule changes the behavior of all source-level step-into operations, such as the **dstep** command or the **Step** button or menu items in the graphical user interface.

   A skip over rule tells TotalView to not step into the function, but instead step over the function. Skip over is most useful to avoid stepping into library functions, such as C++ STL code.

2. A matching and enabled skip **through** rule changes the behavior of all source-level single-stepping operations, such as the **dstep** and **dnext** commands or the **Step** and **Next** buttons or menu items in the graphical user interface.

   A skip through rule tells TotalView to ignore any source-line information for the function, so that single stepping does not stop at source lines within the function. However, if the function being skipped through calls another function, that call is handled according to original single-stepping operation. Skip through is most useful for callback or thunk functions.

RELATED TOPICS	
The **dskip** command	**dskip** in "CLI Commands" in the Classic TotalView Reference Guide
The **dnext** command	**dnext** in "CLI Commands" in the Classic TotalView Reference Guide
The **dstep** command	**dstep** in "CLI Commands" in the Classic TotalView Reference Guide

Executing to a Selected Line

If you don't need to stop execution every time execution reaches a specific line, you can tell TotalView to run your program to a selected line or machine instruction. After selecting the line on which you want the program to stop, invoke one of the eight **Run To** commands defined within the GUI. These commands are on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... duntil

Executing to a selected line is discussed in greater depth in Group, Process, and Thread Control on page 571.

If your program reaches a breakpoint while running to a selected line, TotalView stops at that breakpoint.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane. When you do this, TotalView determines where to stop execution by looking at the following:

- The frame pointer (FP) of the selected stack frame.
- The selected source line or instruction.

CLI: dup and ddown

RELATED TOPICS

Detailed discussion on stepping and setting breakpoints	Stepping and Setting Breakpoints on page 434
The **duntil** command	**duntil** in "CLI Commands" in the Classic TotalView Reference Guide
The **dup** command	**dup** in "CLI Commands" in the Classic TotalView Reference Guide
The **ddown**command	**ddown** in "CLI Commands" in the Classic TotalView Reference Guide

Executing Out of a Function

You can step your program out of a function by using the **Out** commands. The eight **Out** commands in the GUI are located on the **Group**, **Process**, and **Thread** menus.

CLI: dfocus ... dout

If the source line that is the *goal* of the **Out** operation has more than one statement, TotalView will stop execution just after the routine from which it just emerged. For example, suppose that the following is your source line:

```
routine1; routine2;
```

Suppose you step into **routine1**, then use an **Out** command. While the PC arrow in the Source Pane still points to this same source line, the actual PC is just after **routine1**. This means that if you use a step command, you will step into **routine2**.

The PC arrow does not move when the source line only has one statement on it. The internal PC does, of course, change.

You can also return out of several functions at once, by selecting the routine in the Stack Trace Pane that you want to go to, and then selecting an **Out** command.

If your program calls recursive functions, you can select a nested stack frame in the Stack Trace Pane to indicate which instance you are running out of.

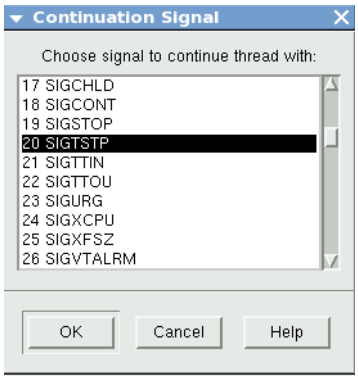
RELATED TOPICS	
The **dout** command	**dout** in "CLI Commands" in the Classic TotalView Reference Guide

Continuing with a Specific Signal

Letting your program continue after sending it a signal is useful when your program contains a signal handler. To set this up:

1. Select the Process Window's **Thread > Continuation Signal** command.

Figure 87, Thread > Continuation Signal Dialog Box



2. Select the signal to be sent to the thread and then select **OK**.

   The continuation signal is set for the thread contained in the current Process Window. If the operating system can deliver multi-threaded signals, you can set a separate continuation signal for each thread. If it can't, this command clears continuation signals set for other threads in the process.

3. Continue execution of your program with commands such as **Process > Go**, **Step**, **Next**, or **Detach**.

   TotalView continues the threads and sends the specified signals to your process.

NOTE: To clear the continuation signal, select **signal 0** from this dialog box.

You can change the way TotalView handles a signal by setting the **TV::signal_handling_mode** variable in a **.tvdrc** startup file. For more information, see Handling Signals on page 127

RELATED TOPICS	
The **TV::signal_handling_mode** command	The **TV::signal_handling_mode** variable in "TotalView Variables" in the Classic TotalView Reference Guide
Default settings for signals and how to change them	Handling Signals on page 127

Killing (Deleting) Programs

To kill (or delete) all the processes in a control group, use the **Group > Kill** command. The next time you start the program, for example, by using the **Process > Go** command, TotalView creates and starts a fresh master process.

CLI: dfocus g dkill

Restarting Programs

You can use the **Group > Restart** command to restart a program that is running or one that is stopped but hasn't exited.

CLI: drerun

If the process is part of a multi-process program, TotalView deletes all related processes, restarts the master process, and runs the newly created program.

The **Group > Restart** command is equivalent to the **Group > Kill** command followed by the **Process > Go** command.

Setting the Program Counter

TotalView lets you resume execution at a different statement than the one at which it stopped execution by resetting the value of the program counter (PC). For example, you might want to skip over some code, execute some code again after changing certain variables, or restart a thread that is in an error state.

Setting the PC can be crucial when you want to restart a thread that is in an error state. Although the PC symbol in the line number area points to the source statement that caused the error, the PC actually points to the failed machine instruction in the source statement. You need to explicitly reset the PC to the correct instruction. (You can verify the actual location of the PC before and after resetting it by displaying it in the Stack Frame Pane, or displaying both source and assembler code in the Source Pane.)

In TotalView, you can set the PC of a stopped thread to a selected source line or a selected instruction. When you set the PC to a selected line, the PC points to the memory location where the statement begins. For most situations, setting the PC to a selected line of source code is all you need to do.

To set the PC to a selected line:

1. If you need to set the PC to a location somewhere in a line of source code, select the **View > Source As > Both** command.

   TotalView responds by displaying assembler code.

2. Select the source line or instruction in the Source Pane.

   TotalView highlights the line.

3. Select the **Thread > Set PC** command.

   TotalView asks for confirmation, resets the PC, and moves the PC symbol to the selected line.

When you select a line and ask TotalView to set the PC to that line, TotalView attempts to force the thread to continue execution at that statement in the currently selected stack frame. If the currently selected stack frame is not the top stack frame, TotalView asks if it can unwind the stack:

```
This frame is buried. Should we attempt to unwind the stack?
```

If you select **Yes**, TotalView discards deeper stack frames (that is, all stack frames that are more deeply nested than the selected stack frame) and resets the machine registers to their values for the selected frame. If you select **No**, TotalView sets the PC to the selected line, but it leaves the stack and registers in their current state. Since you can't assume that the stack and registers have the right values, selecting **No** is almost always the wrong thing to do.

Setting Action Points

In TotalView, breakpoints are called "action points." TotalView has four kinds of action points:

- A *breakpoint* stops execution of processes and threads that reach it.
- A *barrier point* synchronizes a set of threads or processes at a location.
- An *eval point* executes a code fragment when it is reached.
- A *watchpoint* monitors a location in memory and stops execution when it changes.

This chapter contains the following sections:

About Action Points

Actions points specify an action to perform when a thread or process reaches a source line or machine instruction in your program. TotalView provides four types of action points:

- **Breakpoints**

  When a thread encounters a breakpoint, it stops at the breakpoint. Other threads in the process also stop. You can indicate that you want other related processes to stop, as well. Breakpoints are the simplest kind of action point.

- **Barrier points**

  Barrier points are similar to simple breakpoints, differing in that you use them to synchronize a group of processes or threads. A barrier point holds each thread or process that reaches it until all threads or processes reach it. Barrier points work together with the TotalView hold-and-release feature. TotalView supports thread barrier and process barrier points.

- **Eval points**

  An eval point is a breakpoint that has a code fragment associated with it. When a thread or process encounters an eval point, it executes this code. You can use eval points in a variety of ways, including conditional breakpoints, thread-specific breakpoints, countdown breakpoints, and patching code fragments into and out of your program.

- **Watchpoints**

  A watchpoint tells TotalView to either stop the thread so that you can interact with your program (unconditional watchpoint), or evaluate an expression (conditional watchpoint).

The different kinds of action points that you can use are shown in Figure 88 on page 190.

Action Point Properties

All action points share the following common properties.

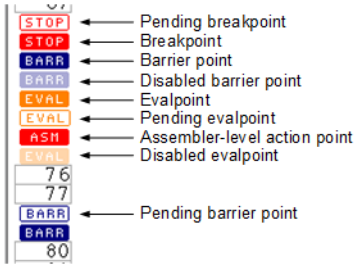
- You can independently enable or disable action points. A disabled action isn't deleted; however, when your program reaches a disabled action point, TotalView ignores it.
- You can share action points across multiple processes or set them in individual processes.
- Action points apply to the process. In a multi-threaded process, the action point applies to all of the threads associated with the process.

- TotalView assigns unique ID numbers to each action point. These IDs appear in several places, including the Root Window, the Action Points Tab of the Process Window, and the **Action Point > Properties** dialog.

Action Point Status Display

In the Process Window, each action point is identified by a symbol:

Figure 88, Action Point Symbols



The ASM icon is displayed when you create a breakpoint on an assembler statement.

For information on pending breakpoints, see Pending Breakpoints on page 201.

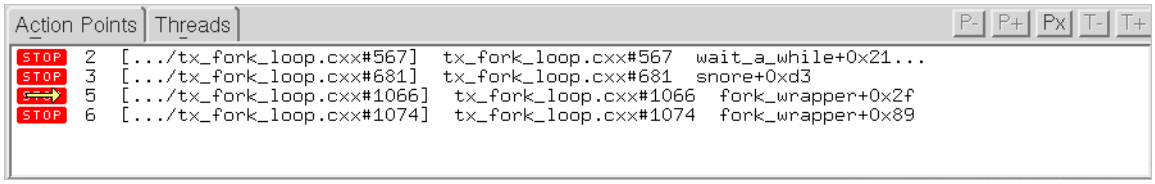
CLI: dactions -- shows information about action points

All action points display as "@" when you use the dlist command to display your source code. Use the**dactions** command to see what type of action point is set.

When your program halts because it encounters an action point, TotalView reports status in several locations. In the Root Window, the Process State is displayed with Breakpoint as well as the letters **ap** followed by a number if the Action Point ID checkbox is enabled in the Configure pane. This is the same number as in the Action Points

tab within the Process Window. In the Process Window, the status lines above the Source Pane also let you know that the thread is at a breakpoint. Finally, TotalView places a yellow arrow over the action point's icon in the Action Point tab. For example:

Figure 89, Action Points Tab



For templated code, an ellipsis (...) is displayed after the address, indicating that additional addresses are associated with the breakpoint.

Manipulating Action Points

When working with action points, you can use your mouse to quickly manipulate breakpoints. In the line number area of the Source Pane, a left mouse click sets a breakpoint at that line, 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 or if you've created an eval point (indicated by an EVAL icon), selecting the icon disables it.

RELATED TOPICS	
Modifying action point properties	The **Action Point > Properties** dialog box in the in-product Help
Using action points with the CLI	Using Action Points on page 471

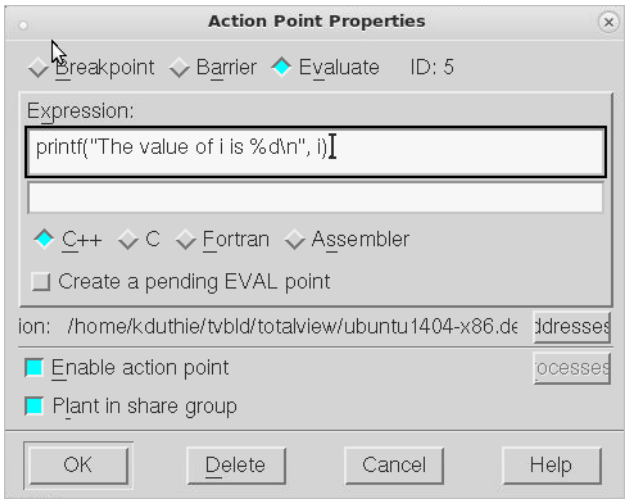
Print Statements vs. Action Points

Print statements are common in debugging, in which you insert **printf()** or **PRINT** statements in your code and then inspect the output. However, using print statements requires that you recompile your program; further, the output may be difficult to navigate as it is likely to be out of order when running multi-process, multi-threaded programs.

You can still use **printf()** statements if you wish — but more effectively and without recompiling your program. Simply add a breakpoint that prints information, using the **Action Point Properties,** dialog, Figure 90, which adds any code you want to a breakpoint.

NOTE: In this discussion, the term "breakpoint" is often used interchangeably with the broader TotalView-specific term "action point."

Figure 90, Action Point Properties Dialog Box

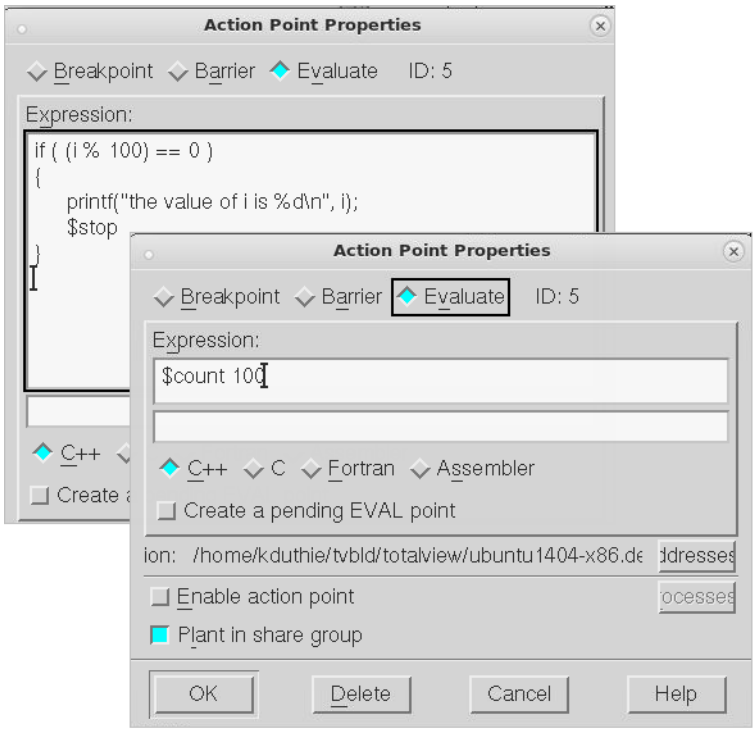


A breakpoint with associated code is an eval point. When your program reaches an eval point, TotalView executes the code. For instance, in the above case, TotalView prints the value of **i**.

Eval points do exactly what you tell them to do. Note that, in Figure 90, TotalView allows your program to continue to execute because you didn't tell it to stop. In other words, you don't have to stop program execution just to observe print statement output.

Figure 91 shows two eval points that do stop execution.

Figure 91, Setting Conditions



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

These are just a few ways that action points can define print statements. More examples can be seen throughout this chapter.

Setting Breakpoints and Barriers

TotalView has several options for setting breakpoints, including:

- Source-level breakpoints
- Breakpoints that are shared among all processes in multi-process programs
- Assembler-level breakpoints

You can also control whether TotalView stops all processes in the control group when a single member reaches a breakpoint.

Topics in this section are:

Setting Source-Level Breakpoints

Typically, you set and clear breakpoints before you start a process. To set a source-level breakpoint, select a line number in the Process Window.

Source Pane Line Number Indicators

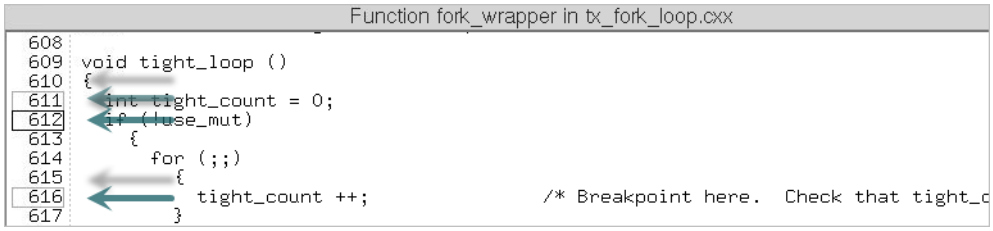
A boxed line number in the Source Pane indicates that the line is associated with executable code:

- **A gray box** denotes that the compiler generated exactly one line number symbol for the source line.

- A **black box** denotes that the compiler generated more than one line number symbol for the source line. The line number symbols might be within a single image file, for example on a "for" loop statement. Or, the line number symbols might be spread across multiple image files if the source file was compiled into the executable, shared libraries, and/or CUDA code.
- **No box** indicates that the compiler did not generate any line number symbols for the source line. However, you can still set a *sliding* or *pending* breakpoint at the line, which is useful if you know that code for that line will be dynamically loaded at runtime, for example, in a dynamically loaded shared library or a CUDA kernel launch.

  For example, Figure 92 illustrates that source lines **611** and **616** both have a single line number symbol, while line **612** has multiple line number symbols. Lines with no box indicate that no executable code exists at those source lines yet (although you can set a sliding or pending breakpoint at those lines, discussed in Pending Breakpoints on page 201 and Sliding Breakpoints on page 198).

Figure 92, Possible breakpoint locations in the Source Pane



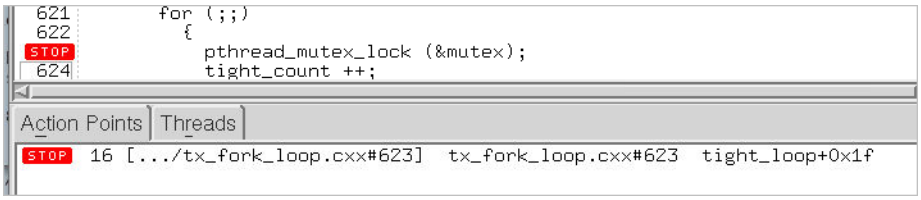
To set a breakpoint in several ways, either:

- Click the line number, *or*
- Right-click on the line number to access the context menu, and choose **Set Breakpoint**. (Choosing Set Barrier creates a barrier point, discussed in Setting Barrier Points on page 215.), *or*
- Select the source line text (not the source line number) and select the menu item **Action Point > Set Breakpoint**.

You can also set a breakpoint while a process is running by selecting a boxed line number in the Source Pane.

A STOP icon in both the Source Pane and the Action Points tab reports that a breakpoint has been set immediately before the source statement.

Figure 93, Breakpoints are identified by stop icons

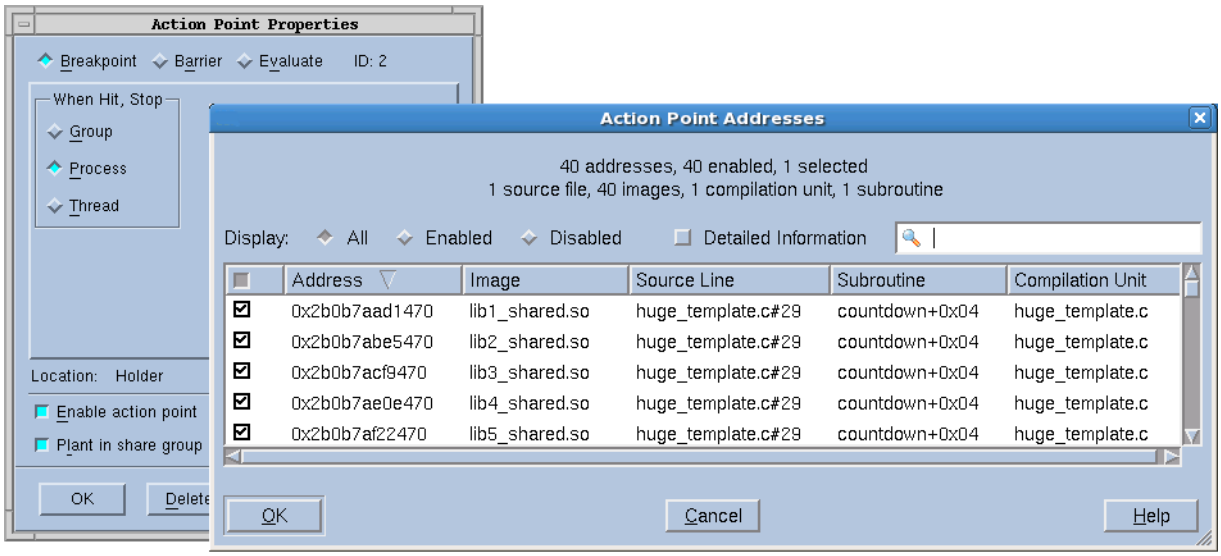


When you set a breakpoint or barrierpoint, it is defined by a *breakpoint expression*, also called a *breakpoint specification*, displayed in the Action Points tab for that breakpoint, or entered into the CLI (if created using the CLI). For more information, see **dbreak** in the *TotalView Reference Guide*.

Choosing Source Lines

If you're using C++ templates, TotalView sets a breakpoint in all instantiations of that template. If this isn't what you want, clear the button and then select the **Addresses** button in the Action Point Properties Dialog Box. You can now clear locations where the action point shouldn't be set.

Figure 94, Action Point and Addresses Dialog Boxes



Initially, addresses are either enabled or disabled, but you can change their state by clicking the checkbox in the first column. The checkbox in the columns bar enables or disables all the addresses. This dialog supports selecting multiple separate items (Ctrl-Click) or a range of items (Shift-Click or click and drag). Once the desired subset is selected, right-click one of the selected items and choose Enable Selection or Disable Selection from the context menu.

Filtering

In complex programs that use many shared libraries, the number of addresses can become very large, so the Addresses dialog has several mechanisms to manage the data. The search box filters the currently displayed data based on one or more space-separated strings or phrases (enclosed in quotes). Remember that data not currently displayed is not included in the filtering. It may be helpful to click the Detailed Information checkbox, which displays much more complete symbol table information, giving you more possibilities for filtering.

Sorting

Clicking on the column labels performs a sort based on the data in that column. Each click toggles between ascending and descending order. If entry values in a column are the same, the values of the column to the right of the sorted column are examined and sorted based on those values. If the values are the same, the next column is examined and so on, until different values are found. The Addresses dialog uses a stable sort, i.e. if all the entries are the same in the selected column and in the columns to the right, the list is not modified.

Displaying and rearranging columns

Finally, right-clicking in the columns bar presents a context menu for displaying or hiding columns. All are initially displayed except Image. You can reorder the columns by selecting a column label and dragging it to a new location.

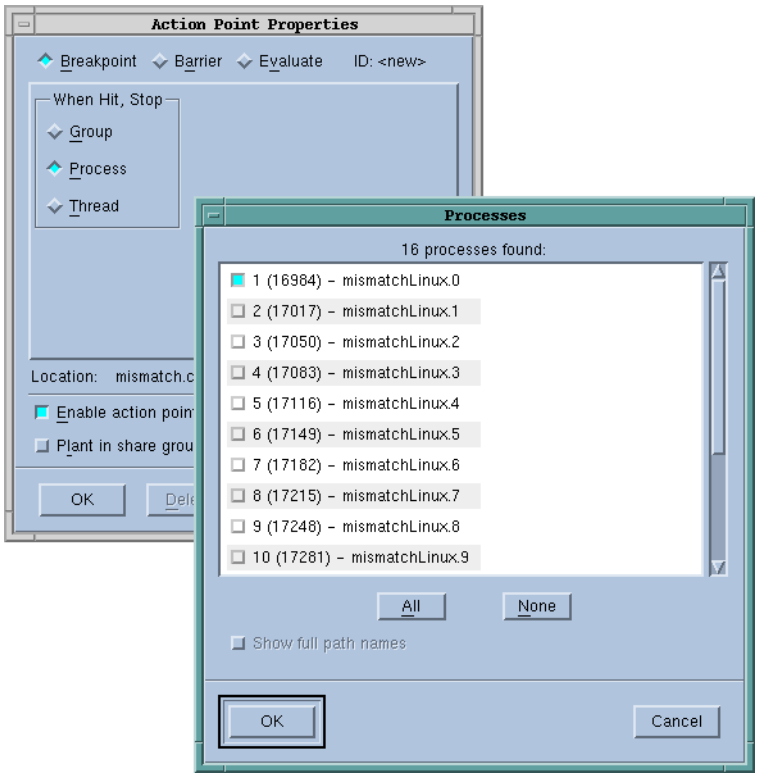
Keyboard Shortcuts

To provide easy access to the buttons at the bottom of the Addresses dialog, the following mnemonic keys have been assigned.

Button	Keyboard Sequence
OK	Alt-o
Cancel	Alt-c
Help	Alt-h

Similarly, in a multi-process program, you might not want to set the breakpoint in all processes. If this is the case, select the **Process** button.

Figure 95, Setting Breakpoints on Multiple Similar Addresses and on Processes

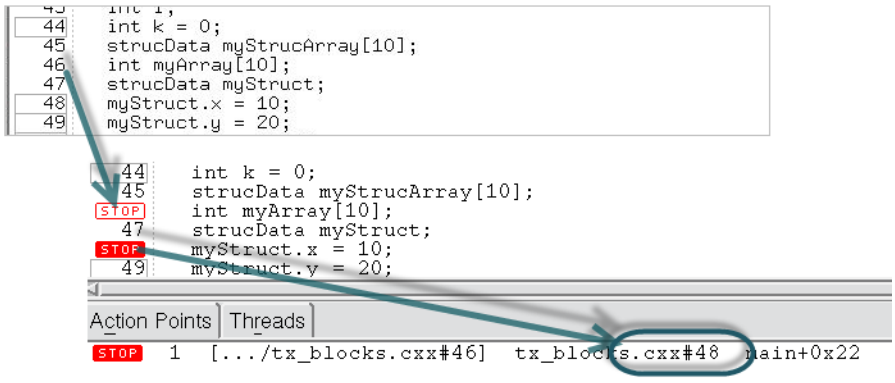


Sliding Breakpoints

If you try to set a breakpoint in the Source Pane at a location with no boxed line, i.e., if there are no line number symbols for that source code line yet, TotalView automatically "slides" the breakpoint to the next line number in the source file that does have a line number symbol.

For example, in Figure 96, a breakpoint was set at line 46 and slid to line 48 where there was a line number symbol. The Source Pane then displays a hollow stop icon indicating that it slid, along with a stop icon at the slid location.

Figure 96, Sliding breakpoint



NOTE: The Action Points tab always displays the full breakpoint expression (in brackets). It also displays the "best" source file and line number it can currently find. TotalView does *not* change the original breakpoint expression, in the event that dynamically loaded code would be a better match later.

The Action Points tab displays the full breakpoint expression in square brackets, abbreviating by-line breakpoints to save space in the display. Following the breakpoint expression, it also displays the "best" source file and line number it can currently find.

The breakpoint expression—pointing to line 46—is displayed in the Actions Points tab as well as the location of the actual breakpoint at line 48. Retaining the original expression supports the situation in which a library that is dynamically loaded does have line number symbols at that location. As the program runs and dynamically loads code, TotalView reevaluates the breakpoint expressions, factoring in any new line number symbols it finds. If better-matching line number information is found, the address blocks in the breakpoint are updated to add the addresses of the new line number symbols, and possibly disable or invalidate old address blocks. This ensures that the breakpoint triggers for the most relevant source line.

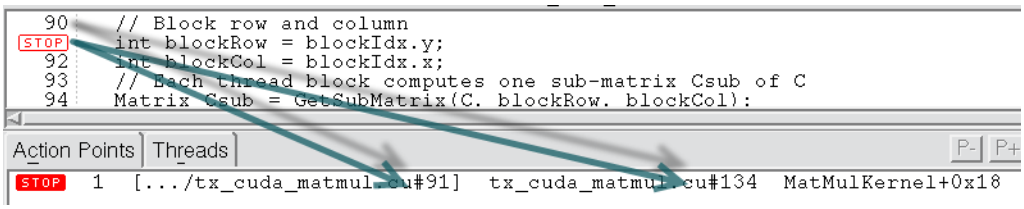
If TotalView cannot find a line number symbol following the line specified in the breakpoint expression, it creates a *pending* breakpoint. For example, this could occur when setting a breakpoint at the end of a source file. See Pending Breakpoints on page 201 for information.

Dynamic Code Loading Example

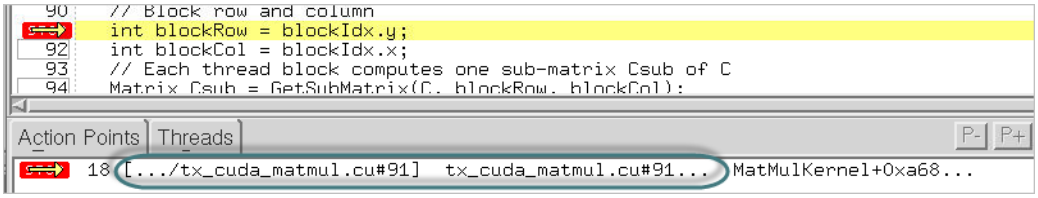
To see how this works, consider a program that will load code at runtime, such as when debugging CUDA code running on a GPU.

Figure 97 illustrates a breakpoint set at line 91 that has slid to line 134:

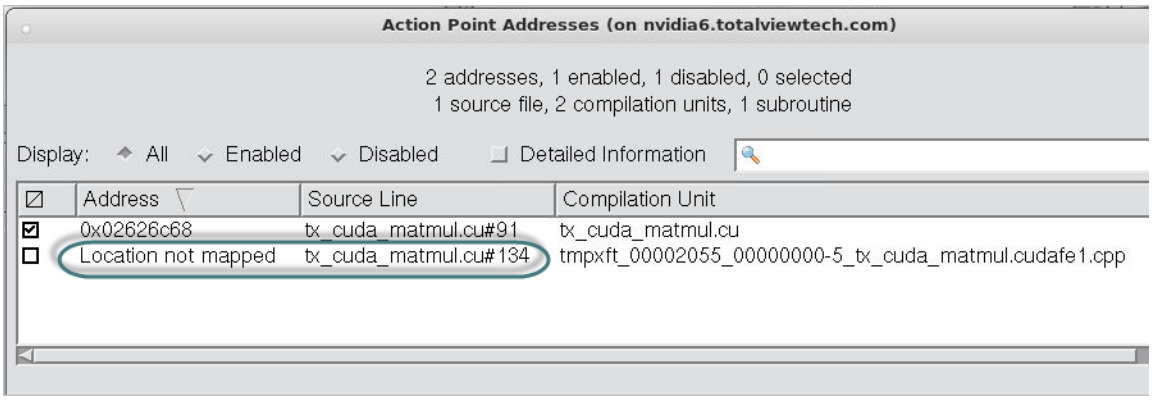
Figure 97, Sliding breakpoints when dynamically loading code



Once the program is running and the CUDA code is loaded, TotalView recalculates the breakpoint expression and is able to plant a breakpoint at line 91 in the CUDA code, which is an exact match for the breakpoint expression:



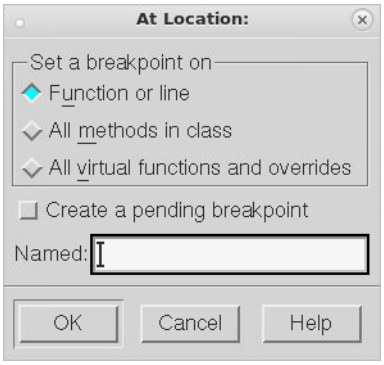
If you focus on the CUDA thread, then view the breakpoint address information (by selecting the breakpoint, right-clicking and selecting **Properties**, then clicking the **Addresses** button), the Action Point Addresses dialog reports that line 134 is not mapped to a breakpoint location, since it's in the host thread; TotalView disabled the address block at line 134 in the host code because line 91 was a better match for the breakpoint expression:



Setting Breakpoints at Locations

You can set or delete a breakpoint at a specific function or source-line number without having to first find the function or source line in the Source Pane. Do this by entering a line number or function name in the **Action Point > At Location** dialog.

Figure 98, Action Point > At Location Dialog Box



TotalView sets a breakpoint at the location. If you enter a function name, TotalView sets the breakpoint at the function's first executable line. In either case, if a breakpoint already exists at a location, TotalView deletes it.

CLI: **dbreak** sets a breakpoint
ddelete deletes a breakpoint

This dialog also allows the creation of a pending breakpoint. See Pending Breakpoints on page 201.

For detailed information about the kinds of information you can enter in this dialog box, see **dbreak** in the *Classic TotalView Reference Guide*.

Pending Breakpoints

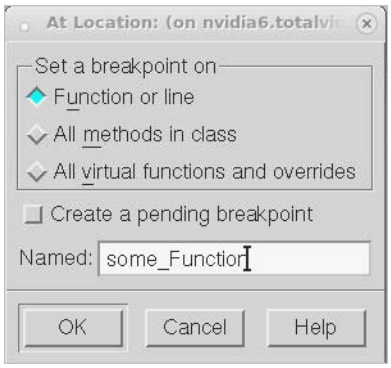
TotalView supports pending breakpoints, useful when setting a breakpoint on code contained in a library that has not yet been loaded into memory.

A pending action point is a breakpoint, barrierpoint, or eval point created with a breakpoint expression that does not yet correspond to any executable code. For example, a common use case is to create a pending function breakpoint with a breakpoint expression that matches the name of a function that will be loaded at runtime via **dlopen**(), CUDA kernel launch, or anything that dynamically loads executable code.

All four types of breakpoints can be pending (this includes line, function, methods in a class, and virtual function breakpoints). Further, a breakpoint may transition between pending to non-pending as image files are loaded, breakpoint expressions are reevaluated, address blocks are added, and invalid address blocks are nullified.

Pending Breakpoints on a Function

When creating a breakpoint on a function using the **Action Point > At Location** dialog box, you are prompted at various points to choose whether to set the breakpoint as pending.



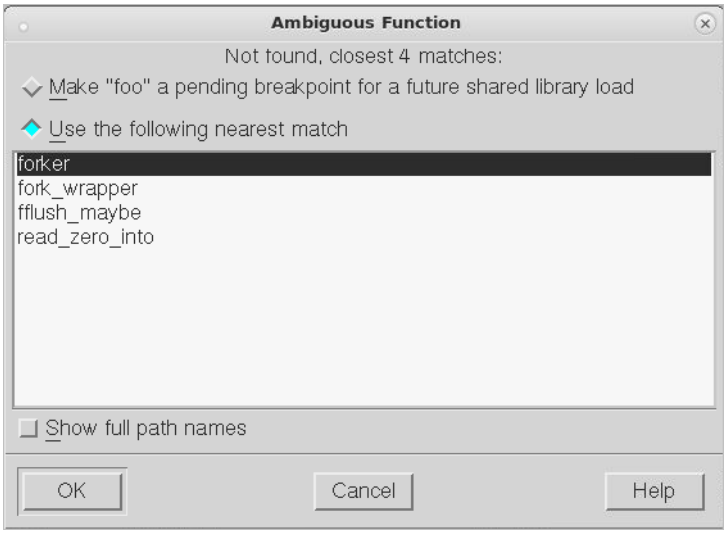
Directly, in the At Location dialog

To immediately set a pending breakpoint, click **Create a pending breakpoint** directly in the **At Location** dialog. This is useful if you are sure that the function name you are entering is correct (even if TotalView can't find it) because it will be dynamically loaded at runtime.

In the Ambiguous function dialog

If you type a function name for which TotalView has no information into the Named field, it assumes that you have either mistyped the function name or that the code containing the function has not yet been loaded into memory. If you don't check the **Create a pending breakpoint** box, and the entered name is similar to that of an existing function, TotalView launches its **Ambiguous Function** dialog, displaying existing functions that are the nearest match to the function in the breakpoint expression.

Figure 99, Ambiguous Function Dialog Box

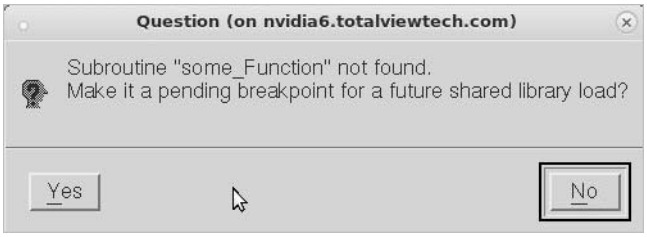


Choose either to create a pending breakpoint or to use one of the provided matches.

Pending breakpoint prompt

If the name you entered was not similar to any existing function, TotalView prompts to set a pending breakpoint.

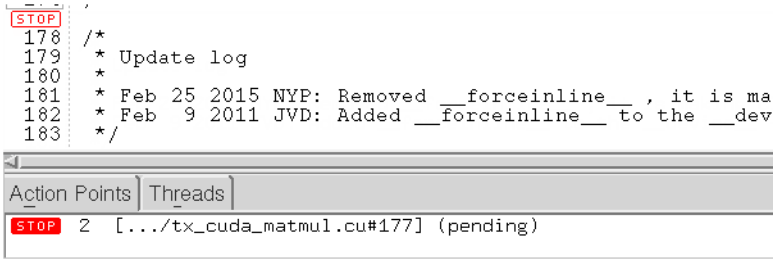
Figure 100, Pending breakpoint prompt



Pending Breakpoints on a Line Number

Because TotalView "slides" a line number breakpoint to the next valid location (see Sliding Breakpoints), explicitly setting a line number pending breakpoint is rarely necessary. If, however, you know that there will be code at that spot, you can explicitly set a pending breakpoint in only these ways:

- By creating a line number breakpoint at a line near the end of a source file where the following lines have no line number symbols, but where you expect there to be dynamically loaded code at runtime. For example, here is a breakpoint set at line 177 just before the end of a file:



- In the **At Location** dialog box, type the file name and line number of a source file that has not been loaded yet. For example, `dynaloaded.c#42` where `dynaloaded.c` is compiled into a dynamically loaded shared library. TotalView posts a dialog box to confirm, unless "Create a pending breakpoint" is selected.

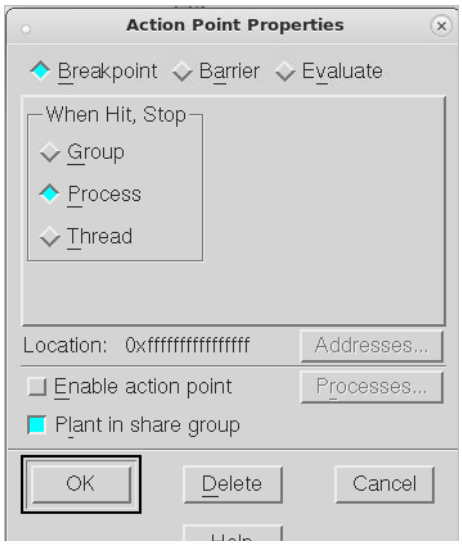
Conflicting Breakpoints

TotalView can place only one action point on an address. Because the breakpoints you specify are actually expressions, the locations to which these expressions evaluate can overlap or even be the same. Sometimes, and this most often occurs with pending breakpoints in dynamically loaded libraries, TotalView cannot predict when action points will overlap. If they do, TotalView enables only one of the action points and disables all others that evaluate to the same address. The action point that TotalView enables is that with the lowest actionpoint ID. The other overlapping action points are marked as "conflicted" in the Action Points pane and **dactions** output.

Displaying and Controlling Action Points

The **Action Point > Properties** Dialog Box sets and controls an action point. Controls in this dialog box also let you change an action point's type to breakpoint, barrier point, or eval point. You can also define what happens to other threads and processes when execution reaches this action point.

Figure 101, Action Point > Properties Dialog Box



The following sections explain how you can control action points by using the Process Window and the **Action Point > Properties** Dialog Box.

CLI: dset SHARE_ACTION_POINT
dset STOP_ALL
ddisable *action-point*

Disabling Action Points

TotalView can retain an action point's definition and ignore it while your program is executing. That is, disabling an action point deactivates it without removing it.

CLI: ddisable *action-point*

You can disable an action point by:

- Clearing **Enable action point** in the **Action Point > Properties** Dialog Box.
- Selecting the STOP or BARR symbol in the Action Points Tab.
- Using the context (right-click) menu.

- Clicking on the **Action Points > Disable** command.

Deleting Action Points

You can permanently remove an action point by selecting the STOP or BARR symbol or selecting the **Delete** button in the **Action Point > Properties** Dialog Box.

To delete all breakpoints and barrier points, use the **Action Point > Delete All** command.

CLI: ddelete

If you make a significant change to the action point, TotalView disables it rather than deleting it when you click the symbol.

Enabling Action Points

You can activate a previously disabled action point by selecting a dimmed STOP, BARR, or EVAL symbol in the Source or Action Points tab, or by selecting **Enable action point** in the **Action Point > Properties** Dialog Box.

CLI: denable

Suppressing Action Points

You can tell TotalView to ignore action points by using the **Action Point > Suppress All** command.

CLI: ddisable -a

The command **ddisable -a** is the closest CLI command to the GUI **Suppress All** feature. However, **ddisable -a** does not actually put TotalView into suppressed action point mode and you can still set and manipulate action points. Be aware that the **ddisable -a** command in the CLI operates only on the current focus. See the *Classic TotalView Reference Guide* for more discussion.

When you suppress action points, you disable them. After you suppress an action point, TotalView changes the symbol it uses within the Source Pane's line number area. In all cases, the icon's color is lighter. Selecting **Suppress All** in the Action Point menu places TotalView in a suppressed action point mode such that no action points are enabled in any process within the entire debugging session. While in this mode, you are unable to create new action points or enable any that are currently disabled.

You can make previously suppressed action points active and allow the creation of new ones by again selecting the **Action Point > Suppress All** command, which functions as a toggle.

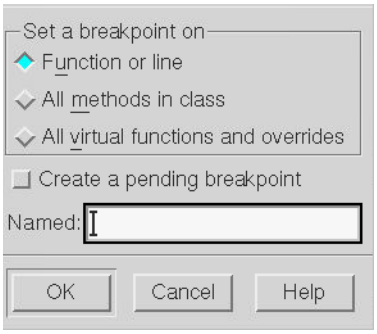
CLI: denable -a

The command **denable -a** is the closest CLI command to turning off **Suppress All** from within the GUI. However, the **denable -a** feature in the CLI operates only on the current focus. See the *Classic TotalView Reference Guide* for more discussion.

Setting Breakpoints on Classes and Functions

The **Action Point > At Location** dialog box lets you set breakpoints on all functions within a class or on a virtual function. The **All Methods in Class** and **All Virtual Functions and Overrides** sets a breakpoint that covers multiple source lines and functions. That is, the breakpoint has multiple address blocks covering all of the locations matching the breakpoint expression.

Figure 102, Action Point > At Location Dialog Box

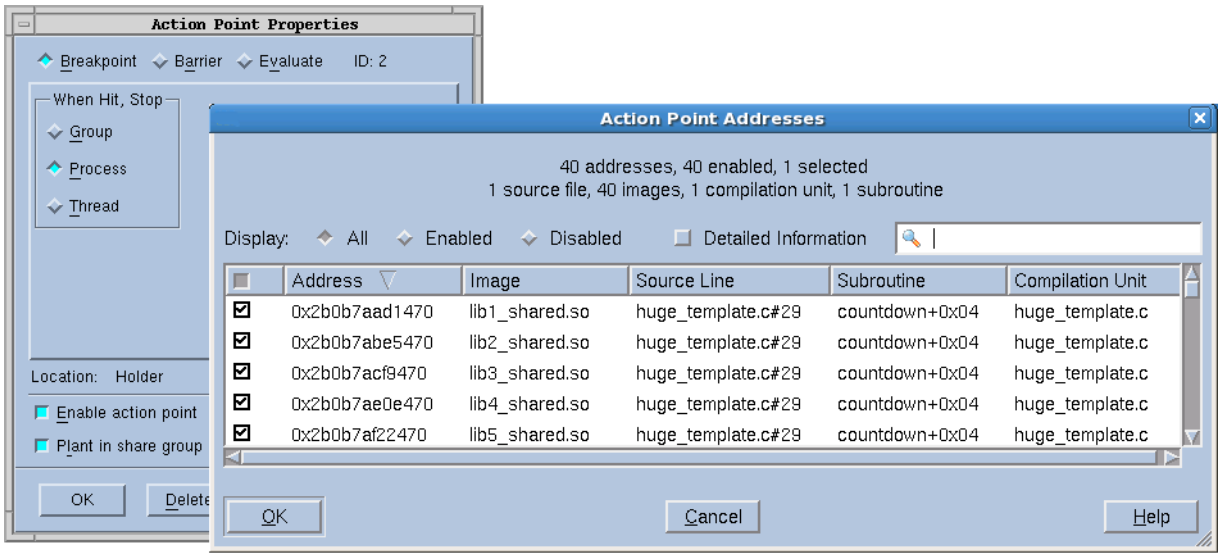


TotalView tells you that the action point is set on a virtual function or a class in the Action Points tab. If you dive on the action point in this tab, TotalView brings up its Ambiguous Line dialog box so that you can select which source line it should display. You may want to select the **Show full path names** check box if you can't tell which you want from the function's signature.

If a function name is overloaded, the debugger sets a breakpoint on each of these functions.

If you want the breakpoint to stop in only some functions, use the Action Point Properties dialog, accessed either by right-clicking on the breakpoint and clicking **Properties** or by selecting the **Action Point > Properties** command, and then clicking **Addresses** to open the **Action Point Addresses** dialog.

Figure 103, Action Point and Addresses Dialog Boxes



You can now enable or disable individual address blocks within the breakpoint. Initially, addresses are either enabled or disabled, but you can change their state by clicking the checkbox in the first column. The checkbox in the columns bar enables or disables all the addresses. This dialog supports selecting multiple separate items (Ctrl-Click) or a range of items (Shift-Click or click and drag). Once the desired subset is selected, right-click one of the selected items and choose Enable Selection or Disable Selection from the context menu.

Filtering

In complex programs that use many shared libraries, the number of addresses can become very large, so the Addresses dialog has several mechanisms to manage the data. The search box filters the currently displayed data based on one or more space-separated strings or phrases (enclosed in quotes). Remember that data not currently displayed is not included in the filtering. It may be helpful to click the Detailed Information checkbox, which displays much more complete symbol table information, giving you more possibilities for filtering.

Sorting

Clicking on the column labels performs a sort based on the data in that column. Each click toggles between ascending and descending order. If entry values in a column are the same, the values of the column to the right of the sorted column are examined and sorted based on those values. If the values are the same, the next column is examined and so on, until different values are found. The Addresses dialog uses a stable sort, i.e. if all the entries are the same in the selected column and in the columns to the right, the list is not modified.

Displaying and rearranging columns

Finally, right-clicking in the columns bar presents a context menu for displaying or hiding columns. All are initially displayed except Image. You can reorder the columns by selecting a column label and dragging it to a new location.

Keyboard Shortcuts

To provide easy access to the buttons at the bottom of the Addresses dialog, the following mnemonic keys have been assigned.

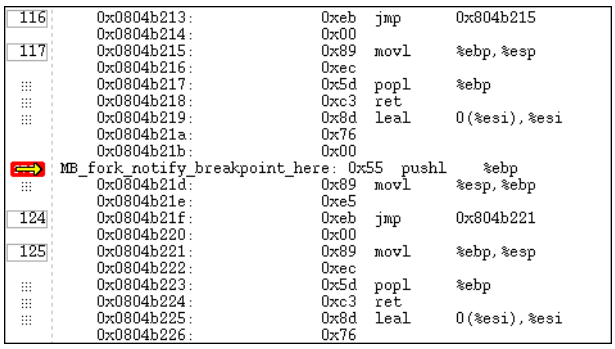
Button	Keyboard Sequence
OK	Alt-o
Cancel	Alt-c
Help	Alt-h

Setting Machine-Level Breakpoints

To set a machine-level breakpoint, you must first display assembler code. You can now select an instruction. TotalView replaces some line numbers with a dotted box (⁝⁝)—this indicates the line is the beginning of a machine instruction. If a line has a line number, this is the line number that appears in the Source Pane. Since instruction sets on some platforms support variable-length instructions, you might see a different number of lines associated with a single line contained in the dotted box. The STOP icon appears, indicating that the breakpoint occurs before the instruction executes.

If you set a breakpoint on the first instruction after a source statement, however, TotalView assumes that you are creating a source-level breakpoint, not an assembler-level breakpoint.

Figure 104, Breakpoint at Assembler Instruction

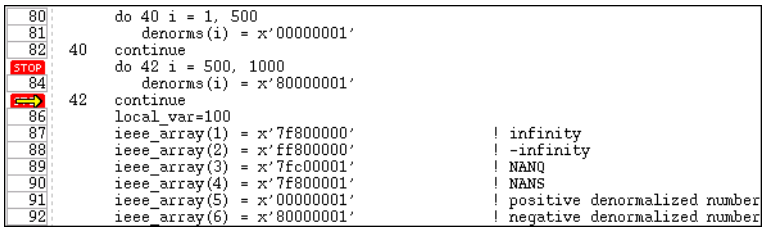


If you set machine-level breakpoints on one or more instructions generated from a single source line, and then display source code in the Source Pane, TotalView displays an ASM icon (Figure 88 on page 190) on the line number. To see the actual breakpoint, you must redisplay assembler instructions.

When a process reaches a breakpoint, TotalView does the following:

- Suspends the process.
- Displays the PC arrow icon ( ) over the stop sign to indicate that the PC is at the breakpoint.

Figure 105, PC Arrow Over a Stop Icon



- Displays **At Breakpoint** in the Process Window title bar and other windows.
- Updates the Stack Trace and Stack Frame Panes and all Variable windows.

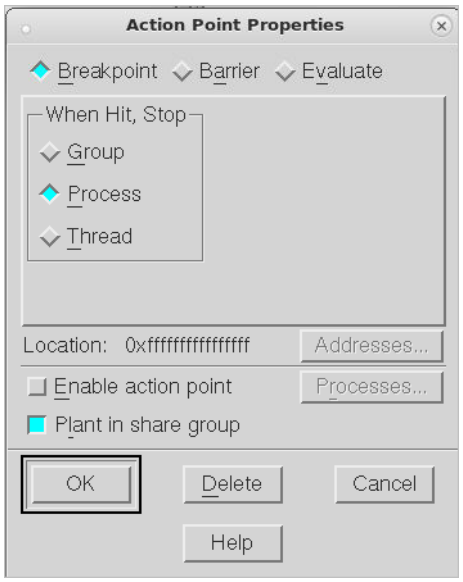
RELATED TOPICS

Displaying assembler code	Viewing the Assembler Version of Your Code on page 173
Barrier points	Setting Breakpoints and Barriers on page 194

Setting Breakpoints for Multiple Processes

In all programs, including multi-process programs, you can set breakpoints in parent and child processes before you start the program and while the program is executing. Do this using the **Action Point > Properties** Dialog Box.

Figure 106, 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**

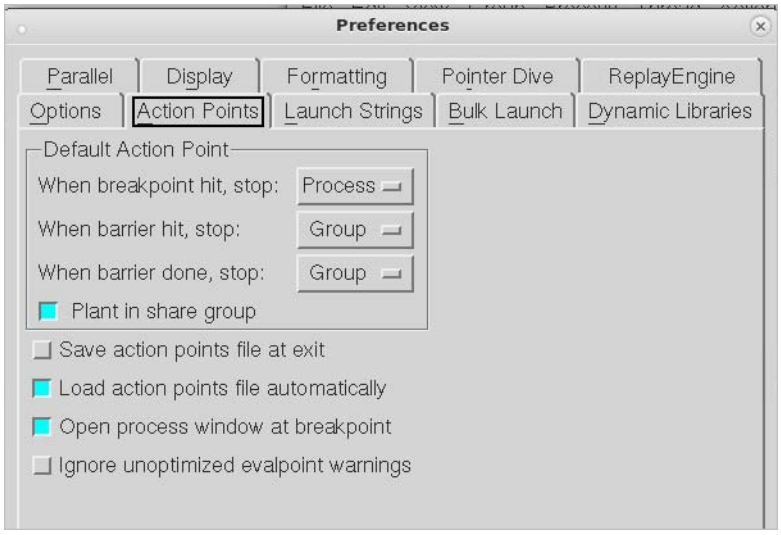
  When checked, TotalView enables the breakpoint in all members of this thread's share group at the same time. When unchecked, you must individually enable and disable breakpoints in each member of the share group.

  CLI: dset SHARE_ACTION_POINT

The **Processes** button specifies which process in a multi-process program will have enabled breakpoints. If **Plant in share group** is selected, this button is disabled since a breakpoint will be set in all of the processes.

You can preset many of the properties in this dialog box by selecting the **File > Preferences** command. Use the Action Points page to set action point preferences.

Figure 107, File > Preferences: Action Points Page



You can find additional information about this dialog box in the online Help.

If you select the **Evaluate** button in the **Action Point > Properties** Dialog Box, you can add an expression to the action point. This expression is attached to control and share group members.

If you're trying to synchronize your program's threads, you need to set a barrier point.

RELATED TOPICS

Using various programming languages in expressions	Using Programming Language Elements on page 367
Barrier points	Setting Breakpoints and Barriers on page 194 and Setting Barrier Points on page 215
Action Point > Properties dialog box	**Action Point > Properties** dialog box in the in-product Help

Setting Breakpoints When Using the fork()/execve() Functions

NOTE: If you are using ReplayEngine, note that it does not follow **fork()** or **vfork()** system calls. For more information on ReplayEngine, see *Reverse Debugging with ReplayEngine*.

Debugging Processes That Call the fork() Function

NOTE: You can control how TotalView handles system calls to **fork()**. See Fork Handling on page 569.

By default, TotalView places breakpoints in all processes in a share group. (For information on share groups, see Organizing Chaos on page 394.) When any process in the share group reaches a breakpoint, TotalView stops all processes in the control group. This means that TotalView stops the control group that contains the share group. This control can contain more than one share group.

To override these defaults:

1. Dive into the line number to display the **Action Point > Properties** Dialog Box.
2. Clear the **Plant in share group** check box and make sure that the **Group** radio button is selected.

   CLI: dset SHARE_ACTION_POINT false

Debugging Processes that Call the execve() Function

NOTE: You can control how TotalView handles system calls to **execve()**. See Exec Handling on page 568.

Shared breakpoints are not set in children that have different executables.

To set the breakpoints for children that call the **execve()** function:

1. Set the breakpoints and breakpoint options in the parent and the children that do not call the **execve()** function.
2. Start the multi-process program by displaying the **Group > Go** command.

   When the first child calls the **execve()** function, TotalView displays the following message:

```
Process name has exec'd name.
Do you want to stop it now?
```

CLI: G

3. Answer **Yes**.

   TotalView opens a Process Window for the process. (If you answer **No**, you won't have an opportunity to set breakpoints.)

4. Set breakpoints for the process.

   After you set breakpoints for the first child using this executable, TotalView won't prompt when other children call the **execve()** function. This means that if you do not want to share breakpoints in children that use the same executable, dive into the breakpoints and set the breakpoint options.

5. Select the **Group > Go** command.

Example: Multi-process Breakpoint

The following program excerpt illustrates the places where you can set breakpoints in a multi-process program:

```
1  pid = fork();
2  if (pid == -1)
3    error ("fork failed");
4  else if (pid == 0)
5    children_play();
6  else
7    parents_work();
```

The following table describes what happens when you set a breakpoint at different places:

Line Number	Result
1	Stops the parent process before it forks.
2	Stops both the parent and child processes.
3	Stops the parent process if **the fork()** function failed.
5	Stops the child process.
7	Stops the parent process.

RELATED TOPICS	
Share groups and TotalView's default design for organizing multiple processes into groups	Organizing Chaos on page 394
Controlling TotalView's behavior for fork, vfork and execve handling	Controlling fork, vfork, and execve Handling on page 566
The dbfork library	Compiling Programs on page 87, and "Linking with the dbfork Library" in the *Classic TotalView Reference Guide*

Setting Barrier Points

A barrier breakpoint is similar to a simple breakpoint, differing only in that it holds processes and threads that reach the barrier point. Other processes and threads continue to run. TotalView holds these processes or threads until all processes or threads defined in the barrier point reach this same place. When the last one reaches a barrier point, TotalView releases all the held processes or threads, but they do not continue executing until you explicitly restart execution. In this way, barrier points let you synchronize your program's execution.

CLI: dbarrier

Topics in this section are:

- About Barrier Breakpoint States on page 215
- Setting a Barrier Breakpoint on page 216
- Creating a Satisfaction Set on page 217
- Hitting a Barrier Point on page 218
- Releasing Processes from Barrier Points on page 218
- Deleting a Barrier Point on page 218
- Changing Settings and Disabling a Barrier Point on page 218

RELATED TOPICS	
How to hold and release threads and processes	Holding and Releasing Processes and Threads on page 422

About Barrier Breakpoint States

Processes and threads at a barrier point are held or stopped, as follows:

Held	A held process or thread cannot execute until all the processes or threads in its group are at the barrier, or until you manually release it. The various *go* and *step* commands from the **Group**, **Process**, and **Thread** menus cannot start held processes.
Stopped	When all processes in the group reach a barrier point, TotalView automatically releases them. They remain stopped at the barrier point until you tell them to resume executing.

You can manually release held processes and threads with the **Hold** and **Release** commands found in the **Group**, **Process**, and **Thread** menus. When you manually release a process, the *go* and *step* commands become available again.

CLI: dfocus ... dhold
dfocus ... dunhold

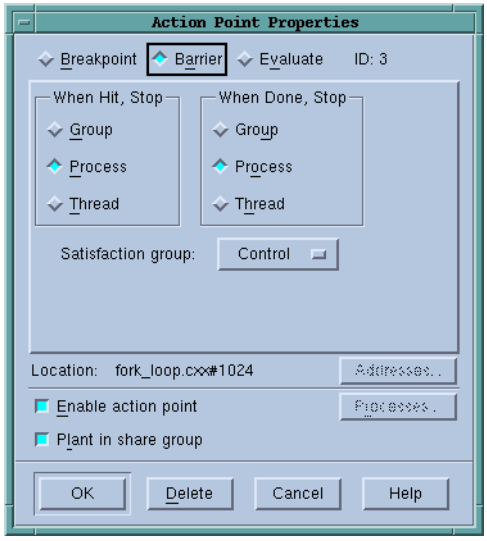
You can reuse the **Hold** command to again toggle the hold state of the process or thread. See Holding and Releasing Processes and Threads on page 422 for more information.

When a process or thread is held, TotalView displays **Stopped** next to the relevant process or thread in the Process State column of the Root Window.

Setting a Barrier Breakpoint

You can set a barrier breakpoint by using the **Action Point > Set Barrier** command or from the **Action Point > Properties** Dialog Box. As an alternative, you can right-click on the line. From the displayed context menu, you can select the **Set Barrier** command.

Figure 108, Action Point > Properties Dialog Box



You most often use barrier points to synchronize a set of threads. When a thread reaches a barrier, it stops, just as it does for a breakpoint. The difference is that TotalView prevents—that is, holds—each thread reaching the barrier from responding to resume commands (for example, ***step***, ***next***, or ***go***) until all threads in the affected set arrive at the barrier. When all threads reach the barrier, TotalView considers the barrier to be ***satisfied*** and releases all of the threads being held there. ***They are just released; they are not continued.*** That is, they are left stopped at the barrier. If you continue the process, those threads also run.

If you stop a process and then continue it, the held threads, including the ones waiting at an unsatisfied barrier, do not run. Only unheld threads run.

The **When Hit, Stop** radio buttons indicate what other threads TotalView stops when execution reaches the breakpoint, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: dbarrier -stop_when_hit

After all processes or threads reach the barrier, TotalView releases all held threads. *Released* means that these threads and processes can now run.

The **When Done, Stop** radio buttons tell TotalView what else it should stop, as follows:

Scope	What TotalView does:
Group	Stops all threads in the current thread's control group.
Process	Stops all threads in the current thread's process.
Thread	Stops only this thread.

CLI: dbarrier -stop_when_done

Creating a Satisfaction Set

For even more control over what TotalView stops, you can select a *satisfaction set*. This setting tells TotalView which processes or threads must be held before it can release the group. That is, the barrier is *satisfied* when TotalView has held all of the indicated processes or threads. The choices from the drop-down menu for the **Satisfaction group** are **Control**, **Share**, and **Workers**. The default setting, **Control**, affects all the processes controlled by

TotalView. The **Share** setting affects all the processes that share the same image as the current executable where the barrier point is set. For multi-threaded programs, to hold the threads at the barrier point, use the **Workers** setting, which holds at the thread level. **Control** and **Share** settings hold at the process level.

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, the Root Window displays **Stopped** in the Process State column for that process's or thread's entry, and the main Process Window displays **Held** in the title bar.

CLI: dstatus

If you create a barrier and all the process's threads are already at that location, TotalView won't hold any of them. However, if you create a barrier and all of the processes and threads are not at that location, TotalView holds any thread that is already there.

Releasing Processes from Barrier Points

TotalView automatically releases processes and threads from a barrier point when they hit that barrier point and all other processes or threads in the group are already held at it.

Deleting a Barrier Point

You can delete a barrier point in the following ways:

- Use the **Action Point > Properties** Dialog Box.
- Click the 

icon in the line number area.

CLI: ddelete

Changing Settings and Disabling a Barrier Point

Setting a barrier point at the current PC for a *stopped* process or thread holds the process there. If, however, all other processes or threads affected by the barrier point are at the same PC, TotalView doesn't hold them. Instead, TotalView treats the barrier point as if it were an ordinary breakpoint.

TotalView releases all processes and threads that are held and which have threads at the barrier point when you disable the barrier point. You can disable the barrier point in the **Action Point > Properties** Dialog Box by selecting **Enable action point** at the bottom of the dialog box.

CLI: ddisable

Defining Eval Points and Conditional Breakpoints

TotalView supports ***eval points***. These are action points at which you have added a code fragment that TotalView executes. You can write the code fragment in C, Fortran, or assembler.

NOTE: Assembler support is currently available on the IBM AIX operating systems. You can enable or disable TotalView's ability to compile eval points.

NOTE: When running on many AIX systems, you can improve the performance of compiled expressions by using the **-aix_use_fast_trap** command when starting TotalView. For more information, see the *TotalView Release Notes*, available from the Rogue Wave web site. Search for "fast trap."

Topics in this section are:

You can do the following when you use eval points:

- Include instructions that stop a process and its relatives. If the code fragment can make a decision whether to stop execution, it is called a ***conditional breakpoint***.
- Test potential fixes for your program.
- Set the values of your program's variables.
- Automatically send data to the Visualizer. This can produce animated displays of the changes in your program's data.

You can set an eval point at any source line that generates executable code (marked with a box surrounding a line number) or a line that contains assembler-level instructions. This means that if you can set a breakpoint, you can set an eval point.

At each eval point, TotalView or your program executes the code contained in the eval point before your program executes the code on that line. Although your program can then go on to execute this source line or instruction, it can do the following instead:

- Include a **goto** in C or Fortran that transfers control to a line number in your program. This lets you test program patches.
- Execute a TotalView function. These functions can stop execution and create barriers and countdown breakpoints. For more information on these statements, see Using Built-in Variables and Statements on page 378.

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.

NOTE: If you call a function from an eval point and there's a breakpoint within that function, TotalView will stop execution at that point. Similarly, if there's an eval point in the function, TotalView also evaluates that eval point.

Eval points only modify the processes being debugged—they do not modify your source program or create a permanent patch in the executable. If you save a program's action points, however, TotalView reapplies the eval point whenever you start a debugging session for that program.

NOTE: You should stop a process before setting an eval point in it. This ensures that the eval point is set in a stable context.

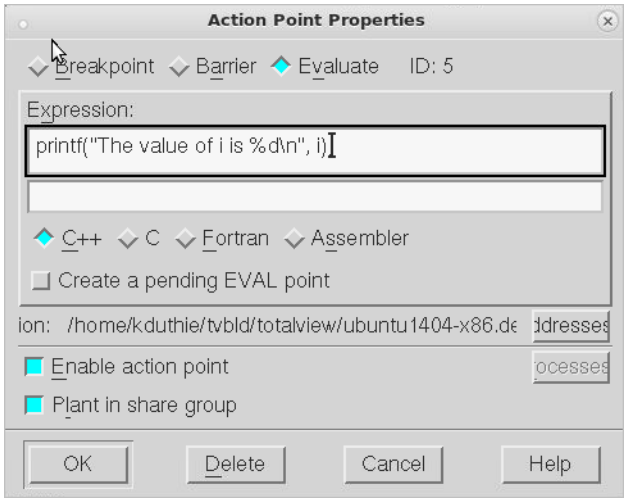
RELATED TOPICS

Saving eval points	Saving Action Points to a File on page 239
Writing code for an expression	Using Programming Language Elements on page 367
TotalView's expression system	Evaluating Expressions on page 360
Using built-in TotalView statements to control execution	Using Built-In Statements on page 379

Setting Eval Points

To create an eval point:

1. Display the **Action Point > Properties** dialog either by right-clicking a STOP icon and selecting **Properties** or by selecting a line and then selecting the **Action Points** menu, then **Properties**.



2. Select the **Evaluate** button.
3. Select the button for the language in which you plan to write the fragment.
4. Type the code fragment. For information on supported C, Fortran, and assembler language constructs, see Using Programming Language Elements on page 367.
5. For multi-process programs, decide whether to share the eval point among all processes in the program's share group. By default, TotalView selects the **Plant in share group** check box for multi-process programs, but you can override this by clearing this setting.
6. Select the **OK** button to confirm your changes.

   If the code fragment has an error, TotalView displays an error message. Otherwise, it processes the code, closes the dialog box, and places an EVAL icon on the line number in the Source Pane.

CLI: dbreak -e
dbarrier -e

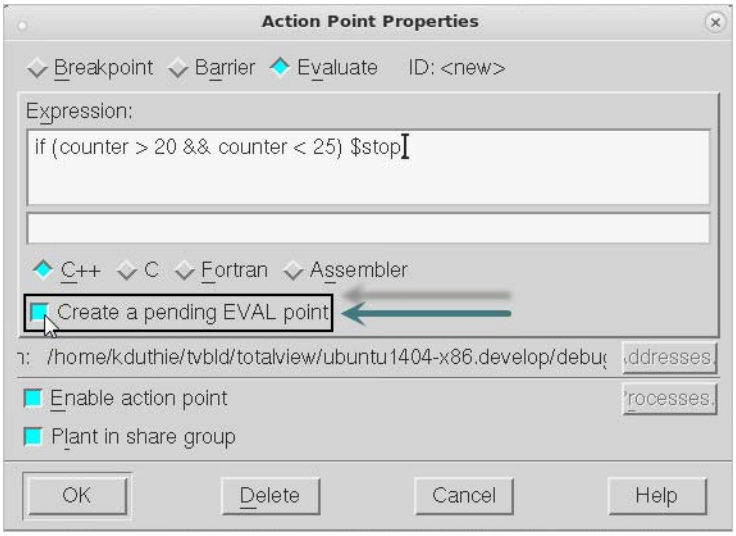
The variables that you refer to in your eval point can either have a global scope or be local to the block of the line that contains the eval point. If you declare a variable in the eval point, its scope is the block that contains the eval point unless, for example, you declare it in some other scope or declare it to be a static variable.

Creating a Pending Eval Point

You can create a *pending* eval point at a location in your code that hasn't yet been loaded, for instance, when your program will dynamically load libraries at runtime. Setting a pending eval point is, essentially, allowing its expression to fail compilation when it is created. For example, it may reference a local variable in the code that will not be defined in the symbol table until the code is loaded and TotalView reads the debug symbols. When your program loads new code at an eval point location, TotalView will attempt to compile the expression. If the eval point expression still fails to compile, the eval point is handled like a breakpoint.

To create a pending eval point, click "Create a pending EVAL point" on the **Action Points Properties** dialog.

Figure 109, Action Point Properties > Create Eval Point



CLI: dbreak -pending -e {expr}
dbarrier -pending -e {expr}

A pending eval point is one in which:

- **The underlying breakpoint is pending**. In this case, TotalView is unlikely to be able to compile the expression (since the breakpoint is not yet instantiated), so it creates a pending eval point.
- **A pending eval point has been explicitly created**. Explicitly creating a pending eval point is useful when an eval point is intended to be set in dynamically loaded code (such as CUDA GPU code), and so the breakpoint slides to the host code before runtime.

Note that the "Create a pending EVAL point" flag sticks to the eval point for the duration of the debug session. The flag is not saved with the eval point when TotalView saves action points; however, when restoring the action points, TotalView will set the flag if the underlying breakpoint needed to slide or was pending.

Creating Conditional Breakpoint Examples

The following are examples that show how you can create conditional breakpoints:

- The following example defines a breakpoint that is reached whenever the **counter** variable is greater than 20 but less than 25:
  ```
  if (counter > 20 && counter < 25) $stop;
  ```
- The following example defines a breakpoint that stops execution every tenth time that TotalView executes the **$count** function
  ```
  $count 10
  ```
- The following example defines a breakpoint with a more complex expression:
  ```
  $count my_var * 2
  ```
  When the **my_var** variable equals **4**, the process stops the eighth time it executes the **$count** function. After the process stops, TotalView reevaluates the expression. If **my_var** equals 5, the process stops again after the process executes the **$count** function ten more times.

The TotalView internal counter is a static variable, which means that TotalView remembers its value every time it executes the eval point. Suppose you create an eval point within a loop that executes 120 times and the eval point contains **$count 100**. Also assume that the loop is within a subroutine. As expected, TotalView stops execution the 100th time the eval point executes. When you resume execution, the remaining 20 iterations occur.

The next time the subroutine executes, TotalView stops execution after 80 iterations because it will have counted the 20 iterations from the last time the subroutine executed.

There is good reason for this behavior. Suppose you have a function that is called from lots of different places from within your program. Because TotalView remembers every time a statement executes, you could, for example, stop execution every 100 times the function is called. In other words, while **$count** is most often used within loops, you can use it outside of them as well.

For descriptions of the **$stop**, **$count**, and variations on **$count**, see Using Built-in Variables and Statements on page 378.

Patching Programs

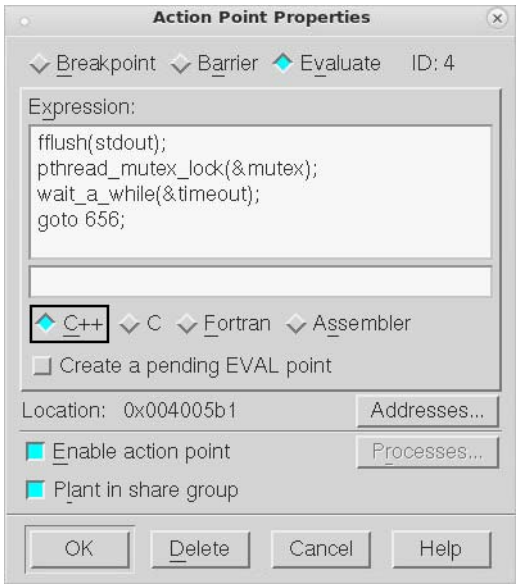
Eval points let you patch your programs and route around code that you want replaced, supporting:

- Branching around code that you don't want your program to execute.
- Adding new statements.

In many cases, correcting an error means that you will do both operations: use a **goto** to branch around incorrect lines and add corrections.

For example, suppose you need to change several statements. Just add these to an action point, then add a **goto** (C) or **GOTO** (Fortran) statement that jumps over the code you no longer want executed. For example, the eval point in Figure 110 executes three statements and then skips to line 656.

Figure 110, Patching Using an Eval Point



Branching Around Code

The following example contains a logic error where the program dereferences a null pointer:

```
1  int check_for_error (int *error_ptr)
2  {
3    *error_ptr = global_error;
4    global_error = 0;
5    return (global_error != 0);
6  }
```

The error occurs because the routine that calls this function assumes that the value of **error_ptr** can be 0. The **check_for_error()** function, however, assumes that **error_ptr** isn't null, which means that line 3 can dereference a null pointer.

You can correct this error by setting an eval point on line 3 and entering:

```
if (error_ptr == 0) goto 4;
```

If the value of **error_ptr** is null, line 3 isn't executed. Note that you are not naming a label used in your program. Instead, you are naming one of the line numbers generated by TotalView.

Adding a Function Call

The example in the previous section routed around the problem. If all you wanted to do was monitor the value of the **global_error** variable, you can add a **printf()** function call that displays its value. For example, the following might be the eval point to add to line 4:

```
printf ("global_error is %d\n", global_error);
```

TotalView executes this code fragment before the code on line 4; that is, this line executes before **global_error** is set to 0.

Correcting Code

The following example contains a coding error: the function returns the maximum value instead of the minimum value:

```
1  int minimum (int a, int b)
2  {
3    int result;        /* Return the minimum */
4    if (a < b)
5      result = b;
6    else
7      result = a;
8    return (result);
9  }
```

Correct this error by adding the following code to an eval point at line 4:

```
if (a < b) goto 7; else goto 5;
```

This effectively replaces the **if** statement on line 4 with the code in the eval point.

About Interpreted and Compiled Expressions

On all platforms, TotalView can interpret your eval points. On IBM AIX platforms, TotalView can also compile them. Compiling the expressions in eval points is *not* the default so you must explicitly request it.

With compiled eval points, performance will be significantly better, particularly if your program is using multi-processors. This is because interpreted eval points are single-threaded through the TotalView process. In contrast, compiled eval points execute on each processor.

The **TV::compile_expressions** CLI variable enables or disables compiled expressions. See "*Operating Systems*" in the *Classic TotalView Reference Guide* for information about how TotalView handles expressions on specific platforms.

NOTE: Using any of the following functions forces TotalView to interpret the eval point rather than compile it: **$clid**, **$duid**, **$nid**, **$processduid**, **$systid**, **$tid**, $pid, and **$visualize**.

About Interpreted Expressions

Interpreted expressions are interpreted by TotalView. Interpreted expressions run slower, possibly much slower, than compiled expressions. With multi-process programs, interpreted expressions run even more slowly because processes may need to wait for TotalView to execute the expression.

When you debug remote programs, interpreted expressions always run slower because the TotalView process on the host, not the TotalView server (**tvdsvr**) on the client, interprets the expression. For example, an interpreted expression could require that 100 remote processes wait for the TotalView process on the host machine to evaluate one interpreted expression. In contrast, if TotalView compiles the expression, it evaluates them on each remote process.

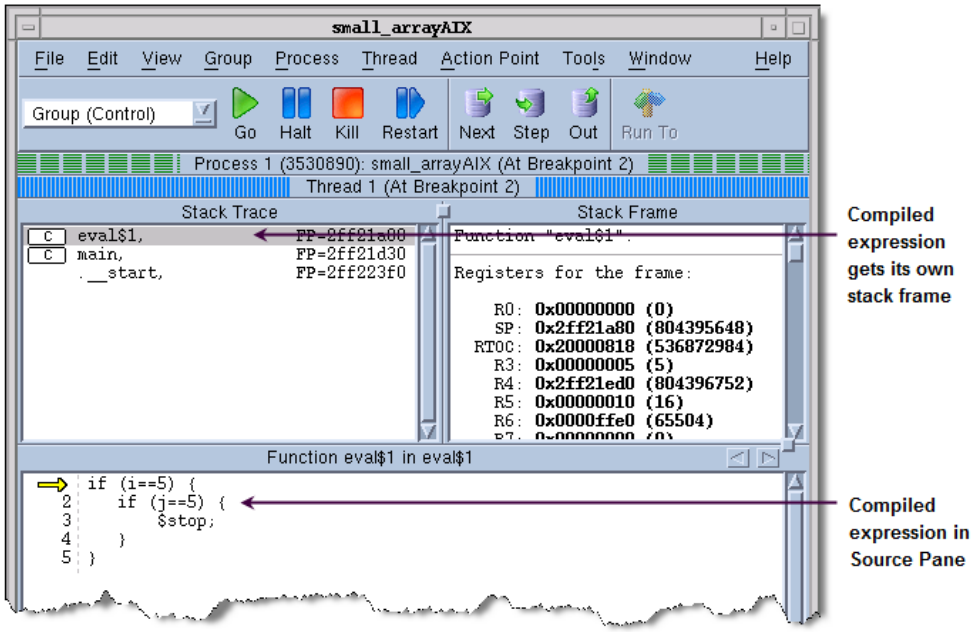
NOTE: Whenever a thread hits an interpreted eval point, TotalView stops execution. This means that TotalView creates a new set of lockstep group*s*. Consequently, if goal threads contain interpreted patches, the results are unpredictable.

About Compiled Expressions

TotalView compiles, links, and patches expressions into the target process. Because the target thread executes this code, eval points and conditional breakpoints execute very quickly. (Conditional watchpoints are always interpreted.) Also, this code doesn't communicate with the TotalView host process until it needs to.

If the expression executes a **$stop** function, TotalView stops executing the compiled expression. At this time, you can single-step through it and continue executing the expression as you would the rest of your code.

Figure 111, Stopped Execution of Compiled Expressions



If you plan to use many compiled expressions or your expressions are long, you may need to think about allocating patch space. For more information, see he section Allocating Patch Space for Compiled Expressions on page 228.

RELATED TOPICS

The **TV::compile_expressions** variable	**TV::compile_expressions** in "TotalView Variables" in the *Classic TotalView Reference Guide*

Allocating Patch Space for Compiled Expressions

TotalView must either allocate or find space in your program to hold the code that it generates for compiled expressions. Since this patch space is part of your program's address space, the location, size, and allocation scheme that TotalView uses might conflict with your program. As a result, you may need to change how TotalView allocates this space.

You can choose one of the following patch space allocation schemes:

- **Dynamic patch space allocation**: Tells TotalView to dynamically find the space for your expression's code.
- **Static patch space allocation**: Tells TotalView to use a statically allocated area of memory.

Allocating Dynamic Patch Space

Dynamic patch space allocation means that TotalView dynamically allocates patch space for code fragments. If you do not specify the size and location for this space, TotalView allocates 1 MB. TotalView creates this space using system calls.

TotalView allocates memory for read, write, and execute access in the addresses shown in the following table:

Platform	Address Range
IBM AIX (-q32)	`0xEFF00000 - 0xEFFFFFFF`
IBM AIX (-q64)	`0x07f0000000000000 - 0x07ffffffffffffff`

NOTE: You can allocate dynamic patch space only for the computers listed in this table.

If the default address range conflicts with your program, or you would like to change the size of the dynamically allocated patch space, you can change the following:

- *Patch space base address* by using the **-patch_area_base** command-line option.
- *Patch space length* by using the **-patch_area_length** command-line option.

RELATED TOPICS	
The**TV::comline_patch_area_base** variable	**TV::comline_patch_area_base** in "TotalView Variables" in the *Classic TotalView Reference Guide*
The **TV::comline_patch_area_length** variable	**TV::comline_patch_area_length** in "TotalView Variables" in the *Classic TotalView Reference Guide*

Allocating Static Patch Space

TotalView can statically allocate patch space if you add a specially named array to your program. When TotalView needs to use patch space, it uses the space created for this array.

You can include, for example, a 1 MB statically allocated patch space in your program by adding the **TVDB_-patch_base_address** data object in a C module. Because this object must be 8-byte aligned, declare it as an array of doubles; for example:

```
          /* 1 megabyte == size TV expects */
  #define PATCH_LEN 0x100000
  double TVDB_patch_base_address [PATCH_LEN / sizeof(double)]
```

If you need to use a static patch space size that differs from the 1 MB default, you must use assembler language. The following shows sample assembler code for IBM AIX-Power:

```
  .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:
```

To use the static patch space assembler code:

1. Use an ASCII editor and place the assembler code into a file named **tvdb_patch_space.s**.
2. Replace the **PATCH_SIZE** tag with the decimal number of bytes you want. This value must be a multiple of 8.
3. Assemble the file into an object file by using a command such as:

   ```
   cc -c tvdb_patch_space.s
   ```

4. Link the resulting **tvdb_patch_space.o** into your program.

Using Watchpoints

TotalView lets you monitor the changes that occur to memory locations by creating a special type of action point called a *watchpoint*. You most often use watchpoints to find a statement in your program that is writing to places to which it shouldn't be writing. This can occur, for example, when processes share memory and more than one process writes to the same location. It can also occur when your program writes off the end of an array or when your program has a dangling pointer.

Topics in this section are:

TotalView watchpoints are called *modify watchpoints* because TotalView only *triggers* a watchpoint when your program modifies a memory location. If a program writes a value into a location that is the same as what is already stored, TotalView doesn't trigger the watchpoint because the location's value did not change.

For example, if location 0x10000 has a value of 0 and your program writes a value of 0 to this location, TotalView doesn't trigger the watchpoint, even though your program wrote data to the memory location. See Triggering Watchpoints on page 236 for more details on when watchpoints trigger.

You can also create *conditional watchpoints*. A conditional watchpoint is similar to a conditional breakpoint in that TotalView evaluates the expression when the value in the watched memory location changes. You can use conditional watchpoints for a number of purposes. For example, you can use one to test whether a value changes its sign—that is, it becomes positive or negative—or whether a value moves above or below some threshold value.

RELATED TOPICS	
Breakpoints and barrier points	Setting Breakpoints and Barriers on page 194
Defining eval points and conditional breakpoints	Defining Eval Points and Conditional Breakpoints on page 220

Using Watchpoints on Different Architectures

The number of watchpoints, and their size and alignment restrictions, differ from platform to platform. This is because TotalView relies on the operating system and its hardware to implement watchpoints.

Watchpoint support depends on the target platform where your application is running, not on the host platform where TotalView is running. For example, if you are running TotalView on host platform "H" (where watchpoints are not supported), and debugging a program on target platform "T" (where watchpoints are supported), you can create a watchpoint in a process running on "T", but not in a process running on "H".

NOTE: Watchpoints are not available on the following target platforms: Mac OS X, and Linux-Power.

The following list describes constraints that exist on each platform:

Computer	Constraints
IBM AIX	You can create one watchpoint on AIX 4.3.3.0-2 (AIX 4.3R) or later systems running 64-bit chips. These are Power3 and Power4 systems. (AIX 4.3R is available as APAR IY06844.) A watchpoint cannot be longer than 8 bytes, and you must align it within an 8-byte boundary. If your watchpoint is less than 8 bytes and it doesn't span an 8-byte boundary, TotalView figures out what to do. You can create compiled conditional watchpoints when you use this system. When watchpoints are compiled, they are evaluated by the process rather than having to be evaluated in TotalView where all evaluations are single-threaded and must be sent from separately executing processes. Only systems having fast traps can have compiled watchpoints.
Linux x86-64 (AMD and Intel)	You can create up to four watchpoints and each must be 1, 2, 4, or 8 bytes in length, and a memory address must be aligned for the byte length. For example, you must align a 4-byte watchpoint on a 4-byte address boundary.

Computer	Constraints
Linux-PowerLE	On Linux-PowerLE platforms (but **not** Linux-Power big-endian platforms) TotalView uses the Linux kernel's `ptrace()` PowerPC hardware debug extension to plant watchpoints. The `ptrace()` interface implements a "hardware breakpoint" abstraction that reflects the capabilities of PowerPC BookE and server processors. If supported at all, the number of watchpoints varies by processor type. Typically, the PowerPC supports at least 1 watchpoint up to 8 bytes long. Systems with the DAWR feature support a watchpoint up to 512 bytes long. The watchpoint triggers if the referenced data address is greater than or equal to the watched address and less than the watched address plus length. Alignment constraints may apply. For example, the watched length may be required to be a power of 2, and the watched address may need to be aligned to that power of 2; that is, `(address % length) == 0`.
Linux ARM64	TotalView supports watchpoints for ARMv8 processors using the hardware's debug watchpoint registers. You can typically create up to four watchpoints (although some processors may have different limits, allowing from 2 to 16 watchpoints, or none at all). Each must be 1, 2, 4, or 8 bytes in length, and the watched memory address must be aligned for the byte length. Watchpoints cannot overlap.
Solaris SPARC and Solaris x86	TotalView supports watchpoints on Solaris 7 or later operating systems. These operating systems let you create hundreds of watchpoints, and there are no alignment or size constraints. However, watchpoints can't overlap.

Typically, a debugging session doesn't use many watchpoints. In most cases, you are only monitoring one memory location at a time. Consequently, restrictions on the number of values you can watch seldom cause problems.

Creating Watchpoints

Watchpoints are created by using either the **Action Points> Create Watchpoint** command in the Process Window or the **Tools > Create Watchpoint** Dialog Box. (If your platform doesn't support watchpoints, TotalView dims this menu item.) Here are some things you should know:

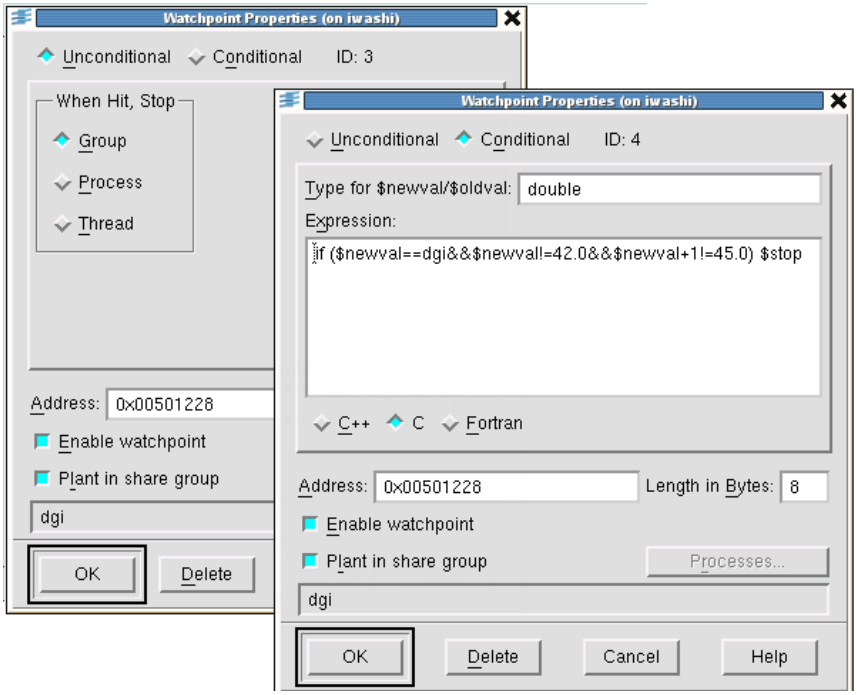
- You can also set watchpoints by right-clicking within the Process and Variable Windows and then select **Create Watchpoint** from the context menu.
- You can select an expression within the Source and Stack Frame panes and then use a context menu or select the **Action Points > Create Watchpoint** command. If you invoke either of these commands and TotalView cannot determine where to set the expression, it displays a dialog box into which you type the variable's name.

- If you select the **Tools > Create Watchpoint** command and a compound variable such an array or structure is being displayed, TotalView sets the watchpoint on the first element. However, if you select an element before invoking this command, TotalView sets the watchpoint on that element.

If you set a watchpoint on a stack variable, TotalView reports that you're trying to set a watchpoint on "non-global" memory. For example, the variable is on the stack or in a block and the variable will no longer exist when the stack is popped or control leaves the block. In either of these cases, it is likely that your program will overwrite the memory and the watchpoint will no longer be meaningful. See Watching Memory on page 235 for more information.

After you select a **Create Watchpoint** command, TotalView displays its Watchpoint Properties dialog box.

Figure 112, Tools > Watchpoint Dialog Boxes



Controls in this dialog box create unconditional and conditional watchpoints. When you set a watchpoint, you are setting it on the complete contents of the information being displayed in the Variable Window. For example, if the Variable Window displays an array, you can set a watchpoint only on the entire array (or as many bytes as TotalView can watch.) If you only want to watch one array element, dive on the element and then set the watchpoint. Similarly, if the Variable Window displays a structure and you only want to watch one element, dive on the element before you set the watchpoint.

RELATED TOPICS

The **Tools > Create Watchpoint** command **Tools > Create Watchpoint** in the in-product Help

Displaying Watchpoints

The watchpoint entry, indicated by UDWP (Unconditional Data Watchpoint) and CDWP (Conditional Data Watchpoint), displays the action point ID, the amount of memory being watched, and the location being watched.

If you select a watchpoint, TotalView toggles the enabled/disabled state of the watchpoint.

Watching Memory

A watchpoint tracks a memory location—it does not track a variable. This means that a watchpoint might not perform as you would expect it to when watching stack or automatic variables. For example, suppose that you want to watch a variable in a subroutine. When control exits from the subroutine, the memory allocated on the stack for this subroutine is deallocated. At this time, TotalView is watching unallocated stack memory. When the stack memory is reallocated to a new stack frame, TotalView is still watching this same position. This means that TotalView triggers the watchpoint when something changes this newly allocated memory.

Also, if your program reinvokes a subroutine, it usually executes in a different stack location. TotalView cannot monitor changes to the variable because it is at a different memory location.

All of this means that in most circumstances, you shouldn't place a watchpoint on a stack variable. If you need to watch a stack variable, you will need to create and delete the watchpoint each time your program invokes the subroutine.

This doesn't mean you can't place a watchpoint on a stack or heap variable. It just means that what happens is undefined after this memory is released. For example, after you enter a routine, you can be assured that memory locations are always tracked accurately until the memory is released.

NOTE: In some circumstances, a subroutine may be called from the same location. This means that its local variables might be in the same location. So, you might want to try.

If you place a watchpoint on a global or static variable that is always accessed by reference (that is, the value of a variable is always accessed using a pointer to the variable), you can set a watchpoint on it because the memory locations used by the variable are not changing.

Triggering Watchpoints

When a watchpoint triggers, the thread's program counter (PC) points to the instruction *following* the instruction that caused the watchpoint to trigger. If the memory store instruction is the last instruction in a source statement, the PC points to the source line *following* the statement that triggered the watchpoint. (Breakpoints and watchpoints work differently. A breakpoint stops *before* an instruction executes. In contrast, a watchpoint stops *after* an instruction executes.)

Using Multiple Watchpoints

If a program modifies more than one byte with one program instruction or statement, which is normally the case when storing a word, TotalView triggers the watchpoint with the lowest memory location in the modified region. Although the program might be modifying locations monitored by other watchpoints, TotalView only triggers the watchpoint for the lowest memory location. This can occur when your watchpoints are monitoring adjacent memory locations and a single store instruction modifies these locations.

For example, suppose that you have two 1-byte watchpoints, one on location 0x10000 and the other on location 0x10001. Also suppose that your program uses a single instruction to store a 2-byte value at locations 0x10000 and 0x10001. If the 2-byte storage operation modifies both bytes, the watchpoint for location 0x10000 triggers. The watchpoint for location 0x10001 does not trigger.

Here's a second example. Suppose that you have a 4-byte integer that uses storage locations 0x10000 through 0x10003, and you set a watchpoint on this integer. If a process modifies location 0x10002, TotalView triggers the watchpoint. Now suppose that you're watching two adjacent 4-byte integers that are stored in locations 0x10000 through 0x10007. If a process writes to locations 0x10003 and 0x10004 (that is, one byte in each), TotalView triggers the watchpoint associated with location 0x10003. The watchpoint associated with location 0x10004 does not trigger.

Copying Previous Data Values

TotalView keeps an internal copy of data in the watched memory locations for each process that shares the watchpoint. If you create watchpoints that cover a large area of memory or if your program has a large number of processes, you increase TotalView's virtual memory requirements. Furthermore, TotalView refetches data for each memory location whenever it continues the process or thread. This can affect performance.

Using Conditional Watchpoints

If you associate an expression with a watchpoint (by selecting the **Conditional** button in the Watchpoint Properties dialog box entering an expression), TotalView evaluates the expression after the watchpoint triggers. The programming statements that you can use are identical to those used when you create an eval point, except that you can't call functions from a watchpoint expression.

The variables used in watchpoint expressions must be global. This is because the watchpoint can be triggered from any procedure or scope in your program.

NOTE: Fortran does not have global variables. Consequently, you can't directly refer to your program's variables.

TotalView has two variables that are used exclusively with conditional watchpoint expressions:

$oldval	The value of the memory locations before a change is made.
$newval	The value of the memory locations after a change is made.

The following is an expression that uses these values:

```
if (iValue != 42 && iValue != 44) {
    iNewValue = $newval; iOldValue = $oldval; $stop;}
```

When the value of the **iValue** global variable is neither 42 nor 44, TotalView stores the new and old memory values in the **iNewValue** and **iOldValue** variables. These variables are defined in the program. (Storing the old and new values is a convenient way of letting you monitor the changes made by your program.)

The following condition triggers a watchpoint when a memory location's value becomes negative:

```
if ($oldval >= 0 && $newval < 0) $stop
```

And, here is a condition that triggers a watchpoint when the sign of the value in the memory location changes:

```
if ($newval * $oldval <= 0) $stop
```

Both of these examples require that you set the **Type for $oldval/$newval** field in the **Watchpoint Properties** Dialog Box.

For more information on writing expressions, see Using Programming Language Elements on page 367.

If a watchpoint has the same length as the **$oldval** or **$newval** data type, the value of these variables is apparent. However, if the data type is shorter than the length of the watch region, TotalView searches for the first changed location in the watched region and uses that location for the **$oldval** and **$newval** variables. (It aligns data in the watched region based on the size of the data's type. For example, if the data type is a 4-byte integer and byte 7 in the watched region changes, TotalView uses bytes 4 through 7 of the watchpoint when it assigns values to these variables.)

For example, suppose you're watching an array of 1000 integers called **must_be_positive**, and you want to trigger a watchpoint as soon as one element becomes negative. You declare the type for **$oldval** and **$newval** to be **int** and use the following condition:

```
if ($newval < 0) $stop;
```

When your program writes a new value to the array, TotalView triggers the watchpoint, sets the values of **$oldval** and **$newval**, and evaluates the expression. When **$newval** is negative, the **$stop** statement halts the process.

This can be a very powerful technique for range-checking all the values your program writes into an array. (Because of byte length restrictions, you can only use this technique on Solaris.)

NOTE: On all platforms except for IBM AIX, TotalView always interprets conditional watchpoints; it never compiles them. Because interpreted watchpoints are single-threaded in TotalView, every process or thread that writes to the watched location must wait for other instances of the watchpoint to finish executing. This can adversely affect performance.

Saving Action Points to a File

You can save a program's action points to a file. TotalView then uses this information to reset these points when you restart the program. When you save action points, TotalView creates a file named *program_name*.**TVD.v4breakpoints**, where *program_name* is the name of your program.

NOTE: TotalView does not save watchpoints because memory addresses can change radically every time you restart TotalView and your program.

Use the **Action Point > Save All** command to save your action points to a file. TotalView places the action points file in the same directory as your program. In contrast, the **Action Point > Save As** command lets you name the file to which TotalView saves this information.

CLI: dactions -save *filename*

If you're using a preference to automatically save breakpoints, TotalView automatically saves action points to a file. Alternatively, starting TotalView with the **-sb** option (see "*TotalView Command Syntax*" in the *Classic TotalView Reference Guide*) also tells TotalView to save your breakpoints.

At any time, you can restore saved action points if you use the **Action Points > Load All** command. After invoking this command, TotalView displays a File Explorer Window that you can use to navigate to or name the saved file.

CLI: dactions -load *filename*

You control automatic saving and loading by setting preferences. (See **File > Preferences** in the online Help for more information.)

CLI: dset TV::auto_save_breakpoints

RELATED TOPICS

The **TV::auto_save_breakponts** variable	**TV::auto_save_breakponts** in "TotalView Variables" in the *Classic TotalView Reference Guide*
The **TV::auto_load_breakpoints** variable	**TV::auto_load_breakpoints** in "TotalView Variables" in the *Classic TotalView Reference Guide*

Examining and Editing Data and Program Elements

This chapter explains how to examine and edit data and view the various elements of your program. It does not discuss array data. For that information, see Examining Arrays on page 312.

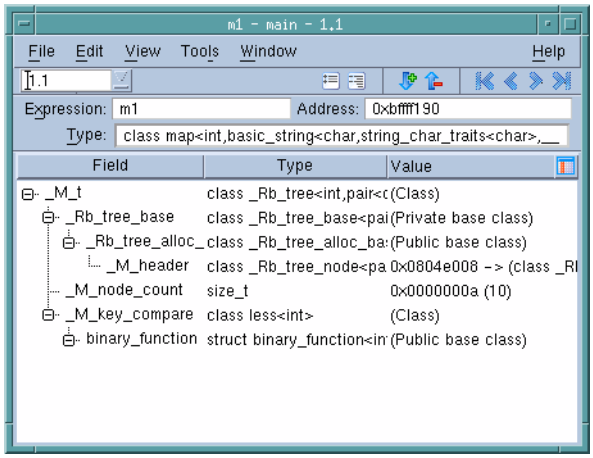
Changing How Data is Displayed

When a debugger displays a variable, it relies on the definitions of the data used by your compiler. The following two sections show how you can change the way TotalView displays this information:

Displaying STL Variables

The C++ STL (Standard Template Library) greatly simplifies access to data. Since it offers standard and prepackaged ways to organize data, you do not have to be concerned with the mechanics of the access method. The disadvantage to using the STL while debugging is that the information debuggers display is organized according to the compiler's view of the data, rather than the STL's logical view. For example, here is how your compiler sees a map compiled using the GNU C++ compiler (gcc):

Figure 113, An Untransformed Map



Most of the information is generated by the STL template and, in most cases, is not interesting. In addition, the STL does not aggregate the information in a useful way.

STLView solves these problems by rearranging (that is, *transforming*) the data so that you can easily examine it. For example, here is the transformed map.

Figure 114, A Transformed Map

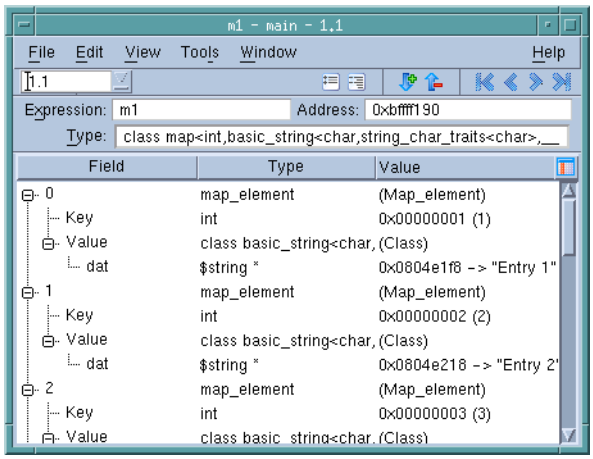
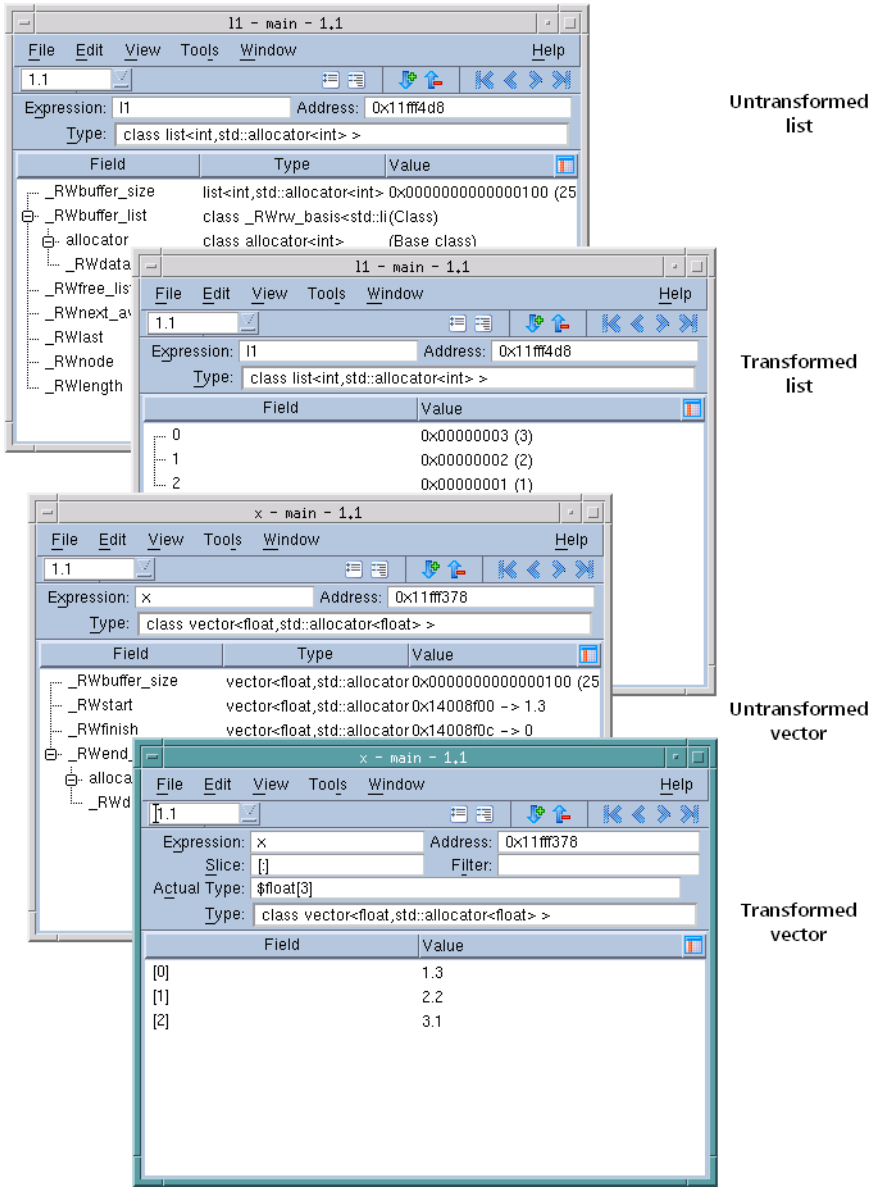


Figure 115 shows an untransformed and transformed list and vector.

Figure 115, List and Vector Transformations



NOTE: By default, TotalView transforms STL strings, vectors, lists, maps, multimaps, sets, and multi-sets. You can create transformations for other STL containers. See "Creating Type Transformations" in the *Classic TotalView Reference Guide* for more information.

By default, TotalView transforms STL types. If you need to look at the untransformed data structures, clear the **View simplified STL containers (and user-defined transformations)** checkbox on the Options Page of the **File > Preference** Dialog Box.

CLI: dset TV::ttf { true | false }

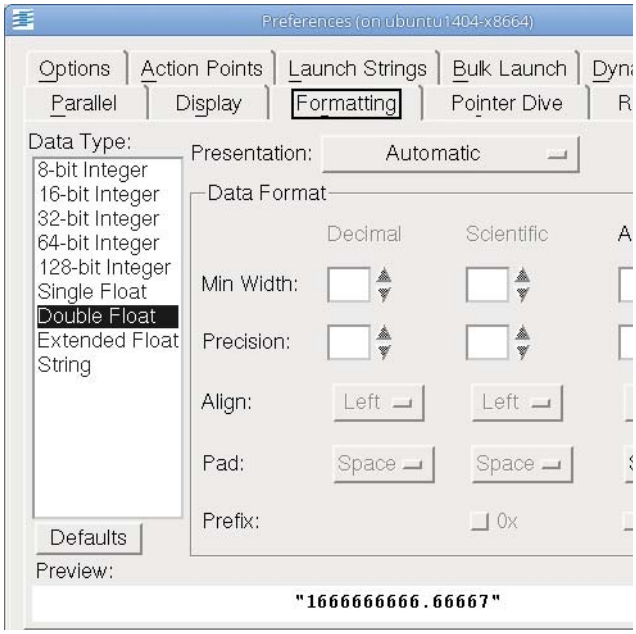
Following pointers in an STL data structure to retrieve values can be time-consuming. By default, TotalView only follows 500 pointers. You can change this by altering the value of the **TV::ttf_ max_length** variable.

RELATED TOPICS	
General information on creating custom type transformations	"Creating Type Transformations" in the *Classic TotalView Reference Guide*
Transforming C++ types	Displaying C++ Types on page 296

Changing Size and Precision

If the default formats that TotalView uses to display a variable's value doesn't meet your needs, you can use the **Formatting** Page of the **File > Preferences** Dialog Box to indicate the precision for simple data types.

Figure 116, File > Preferences Formatting Page



After selecting one of the data types listed on the left side of the Formatting Page, you can set how many character positions a value uses when TotalView displays it (**Min Width**) and how many numbers to display to the right of the decimal place (**Precision**). You can also tell TotalView how to align the value in the **Min Width** area, and if it should pad numbers with zeros or spaces.

Although the way in which these controls relate and interrelate may appear to be complex, the **Preview** area shows you the result of a change. Play with the controls for a minute or so to see what each control does. You may need to set the **Min Width** value to a larger number than you need it to be to see the results of a change. For example, if the **Min Width** value doesn't allow a number to be justified, it could appear that nothing is happening.

CLI: You can set these properties from within the CLI. To obtain a list of variables that you can set, type "**dset TV::data_format***".

RELATED TOPICS

The Formatting Page in the **File > Preferences** menu	The **File > Preferences Formatting Page** in the in-product Help
Data format CLI variables	A list of the TotalView data format variables in the Classic TotalView Reference Guide

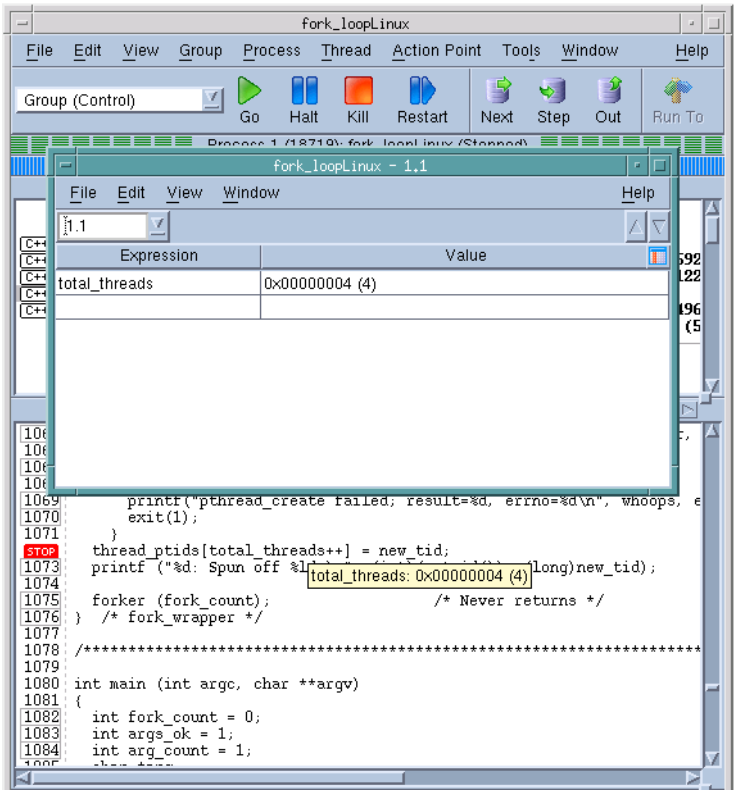
Displaying Variables

The Process Window Stack Frame Pane displays variables that are local to the current stack frame. This pane does not show the data for nonsimple variables, such as pointers, arrays, and structures. To see this information, dive on the variable.

NOTE: Dive on a variable by clicking your middle mouse button on it. If your mouse doesn't have three buttons, you can single- or double-click on an item.

If you place your mouse cursor over a variable or an expression, TotalView displays its value in a tooltip window.

Figure 117, A Tooltip



If TotalView cannot evaluate the object moused over, it still displays basic information. For example, if you place the mouse over a structure, the tooltip reports the kind of structure. In all cases, the displayed information is similar to the same information entered in the Expression List Window.

If you dive on simple variables or registers, TotalView still brings up a Variable Window and you do see some additional information about the variable or register.

Although a Variable Window is the best way to see all of an array's elements or all elements in a structure, using the Expression List Window is easier for variables with one value. Using it also cuts down on the number of windows that are open at any one time. For more information, see Viewing a List of Variables on page 272.

The following sections discuss how you can display variable information:

RELATED TOPICS	
Diving in variable windows	Diving in Variable Windows on page 266
More on examining and editing data	Examining and Editing Data and Program Elements on page 240
Details on the Variable Window	The "Variable Window" in the in-product Help
Viewing lists of variables	Viewing a List of Variables on page 272

Displaying Program Variables

You can display local and global variables by:

- Diving into the variable in the Source or Stack Panes.

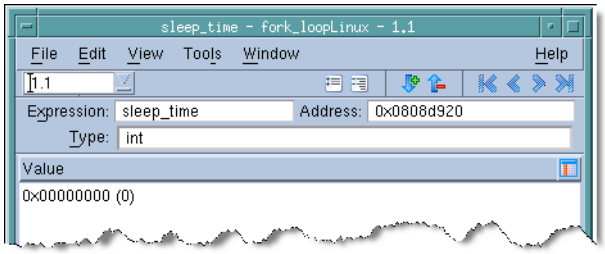
- Selecting the **View > Lookup Variable** command. When prompted, enter the name of the variable.

  CLI: dprint *variable*

- Using the **Tools > Program Browser** command.

After using one of these methods, TotalView displays a Variable Window that contains the information you want. The Variable Window can display simple variables, such as **int**s, sets of like elements such as arrays, or more complicated variables defined as structures and arrays of structures.

Figure 118, Variable Window for a Global Variable



If you keep a Variable Window open while a process or thread is running, the information being displayed might not be accurate. TotalView updates the window when the process or thread stops. If TotalView can't find a stack frame for a displayed local variable, the variable's status is **sparse**, since the variable no longer exists. The **Status** area can contain other information that alerts you to issues and problems with a variable.

When you debug recursive code, TotalView doesn't automatically refocus a Variable Window onto different instances of a recursive function. If you have a breakpoint in a recursive function, you need to explicitly open a new Variable Window to see the local variable's value in that stack frame.

CLI: dwhere, dup, and dprint

Use **dwhere** to locate the stack frame, use **dup** to move to it, and then use **dprint** to display the value.

Select the **View > Compilation Scope > Floating** command to tell TotalView that it can refocus a Variable Window on different instances. For more information, see Viewing Variables in Different Scopes as Program Executes on page 252.

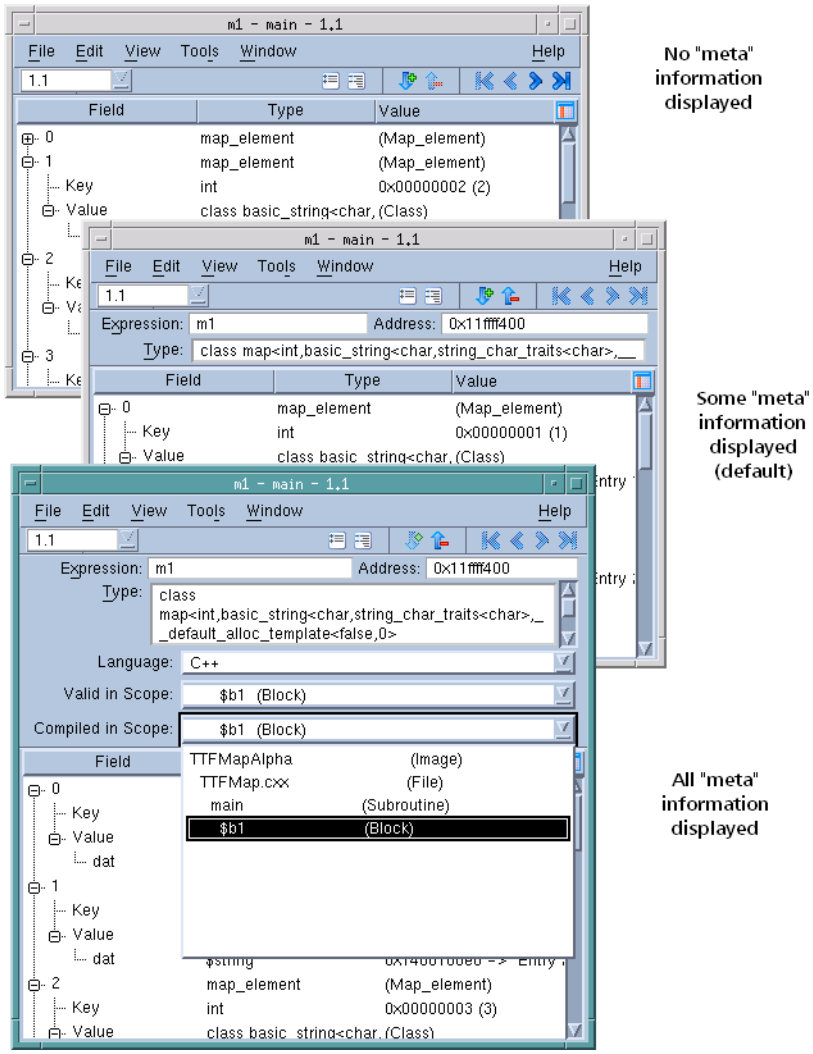
RELATED TOPICS

Using the Process Window	Using the Process Window on page 157
Viewing lists of variables	Viewing a List of Variables on page 272

Controlling the Displayed Information

TotalView can display more information about your variable than its value. This information is sometimes called *meta-information*. You can control how much of this meta-information it displays by clicking on the **More** and **Less** buttons.

Figure 119, Variable Window: Using More and Less



As the button names indicate, clicking **More** displays more meta-information and clicking **Less** displays less of it.

The two most useful fields are **Type**, which shows you what your variable's actual type is, and **Expression**, which allows you to control what is being displayed. This is sometimes needed because TotalView tries to show the type in the way that it thinks you declared it in your program.

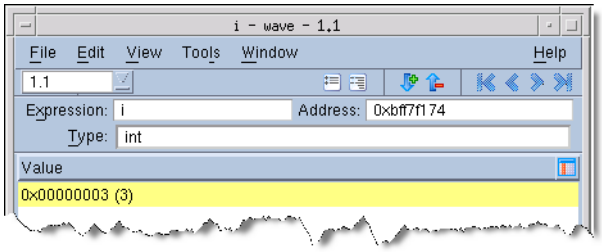
The online help describes all the meta-information fields.

Seeing Value Changes

TotalView reports when a variable's value changes in several ways.

- When your program stops at a breakpoint, TotalView adds a yellow highlight to the variable's value if it has changed,Figure 120

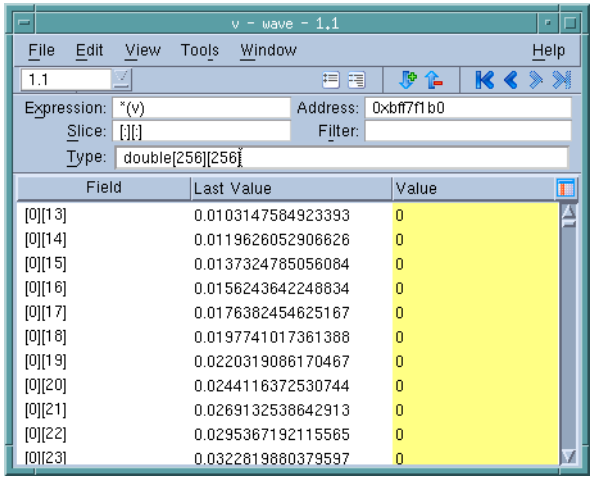
Figure 120, Variable Window With "Change" Highlighting



If the thread is stopped for another reason—for example, you've stepped the thread—and the value has changed, TotalView does not add yellow highlighting to the line.

- You can tell TotalView to display the **Last Value** column. Do this by selecting **Last Value** in the column menu, which is displayed after you click on the column menu ( ) icon, Figure 121.

Figure 121, Variable Window Showing Last Value Column



Notice that TotalView has highlighted all items that have changed within an array. In a similar fashion it can show the individual items that have changed within a structure.

In general, TotalView only retains the value for data items displayed within the Variable Window. At times, TotalView may track adjacent values within arrays and structures, but you should not rely on additional items being tracked.

NOTE: When you scroll the Variable Window, TotalView discards the information it is tracking and fetches new information. So, while the values may have changed, TotalView does not have information about this change. That is, TotalView only tracks what is visible. Similarly, when you scroll back to previously displayed values, TotalView needs to refetch this information. Because it is "new" information, no "last values" exist.

The Expression List window, described in Viewing a List of Variables on page 272, also highlights data and can display a **Last Value** column.

Seeing Structure Information

When TotalView displays a Variable Window, it displays structures in a compact form, concealing the elements within the structure. Click the **+** button to display these elements, or select the **View > Expand All** command to see all entries. If you want to return the display to a more compact form, you can click the **-** button to collapse one structure, or select the **View > Collapse All** command to return the window to what it was when you first opened it.

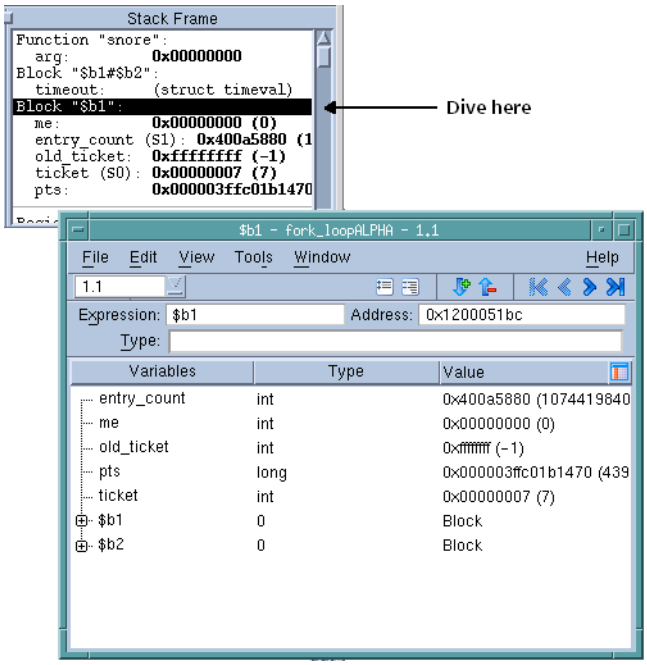
If a substructure contains more than about forty elements, TotalView does not let you expand it in line. That is, it does not place a **+** symbol in front of the substructure. To see the contents of this substructure, dive on it.

Similarly, if a structure contains an array as an element, TotalView only displays the array within the structure if it has fewer than about forty elements. To see the contents of an embedded array, dive on it.

Displaying Variables in the Current Block

In many cases, you may want to see all of the variables in the current block. If you dive on a block label in the Stack Frame Pane, TotalView opens a Variable Window that contains just those variables.

Figure 122, Displaying Scoped Variables



After you dive on a variable in this block window, TotalView displays a Variable Window for that scoped variable. In this figure, block **$b1** has two nested blocks.

RELATED TOPICS

Using the Process Window	Using the Process Window on page 157

Viewing Variables in Different Scopes as Program Executes

When TotalView displays a Variable Window, it understands the scope in which the variable exists. As your program executes, this scope doesn't change. In other words, if you're looking at variable **my_var** in one routine, and you then execute your program until it is within a second subroutine that also has a **my_var** variable, TotalView does not change the scope so that you are seeing the *in scope* variable.

If you would like TotalView to update a variable's scope as your program executes, select the **View > Compilation Scope > Floating** command. This tells TotalView that, when execution stops, it should look for the variable in the current scope. If it finds the variable, it displays the variable contained within the current scope.

Select the **View > Compilation Scope > Fixed** command to return TotalView to its default behavior, which is not to change the scope.

Selecting floating scope can be very handy when you are debugging recursive routines or have routines with identical names. For example, **i**, **j**, and **k** are popular names for counter variables.

Scoping Issues

When you dive into a variable from the Source Pane, the scope that TotalView uses is that associated with the current frame's PC; for example:

```
1: void f()
2: {
3:    int x;
4: }
5:
6: int main()
7: {
8:    int x;
9:}
```

If the PC is at line 3, which is in **f()**, and you dive on the **x** contained in **main()**, TotalView displays the value for the **x** in **f()**, not the **x** in **main()**. In this example, the difference is clear: TotalView chooses the PC's scope instead of the scope at the place where you dove. If you are working with templated and overloaded code, determining the scope can be impossible, since the compiler does not retain sufficient information. In all cases, you can click the **More** button within the Variable window to see more information about your variable. The **Valid in Scope** field can help you determine which instance of a variable you have located.

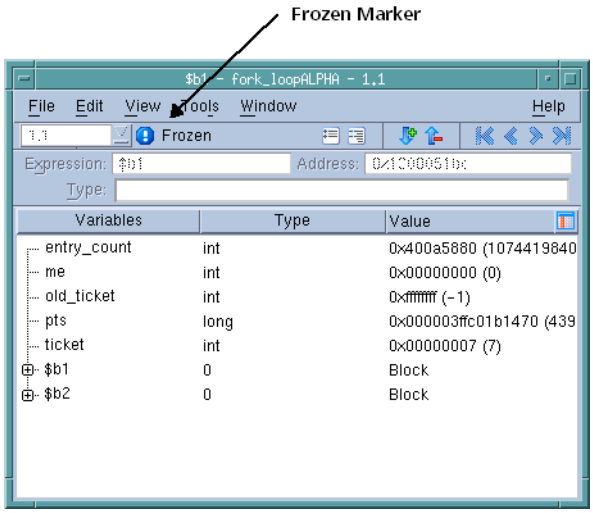
You can use the **View > Lookup Variable** command to locate the instance you are interested in.

Freezing Variable Window Data

Whenever execution stops, TotalView updates the contents of Variable Windows. More precisely, TotalView reevaluates the data based on the Expression field. If you do not want this reevaluation to occur, use the Variable Window's **View > Freeze** command. This tells TotalView that it should not change the information that is displaying.

After you select this command, TotalView adds a marker to the window indicating that the data is frozen.

Figure 123, Variable Window Showing Frozen State



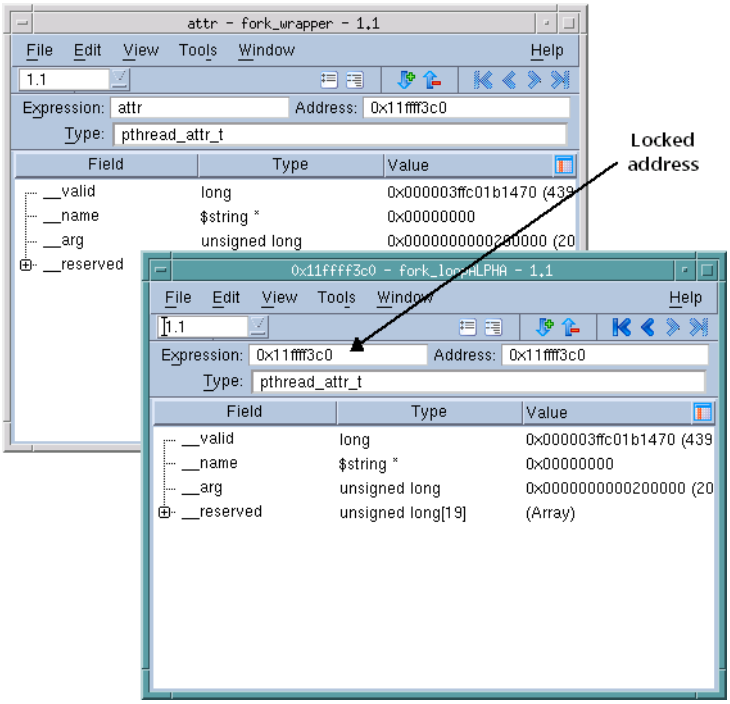
Selecting the **View > Freeze** command a second time removes the freeze. TotalView again evaluates this window's expression whenever execution stops.

In most cases, you'll want to compare the frozen information with an unfrozen copy. Do this by selecting the **Window > Duplicate** command before you freeze the display. As these two windows are identical, it doesn't matter which one you freeze. However, if you use the **Duplicate** command after you freeze the display, be aware that the new duplicated window will continue to update normally. The 'freeze' state of a window is not retained when using the **Window > Duplicate** command.

Locking the Address

Sometimes you want only to freeze the address, not the data at that address. Do this by selecting the **View > Lock Address** command. Figure 124 shows two Variable Windows, one of which has had its address locked.

Figure 124, Locked and Unlocked Variable Windows



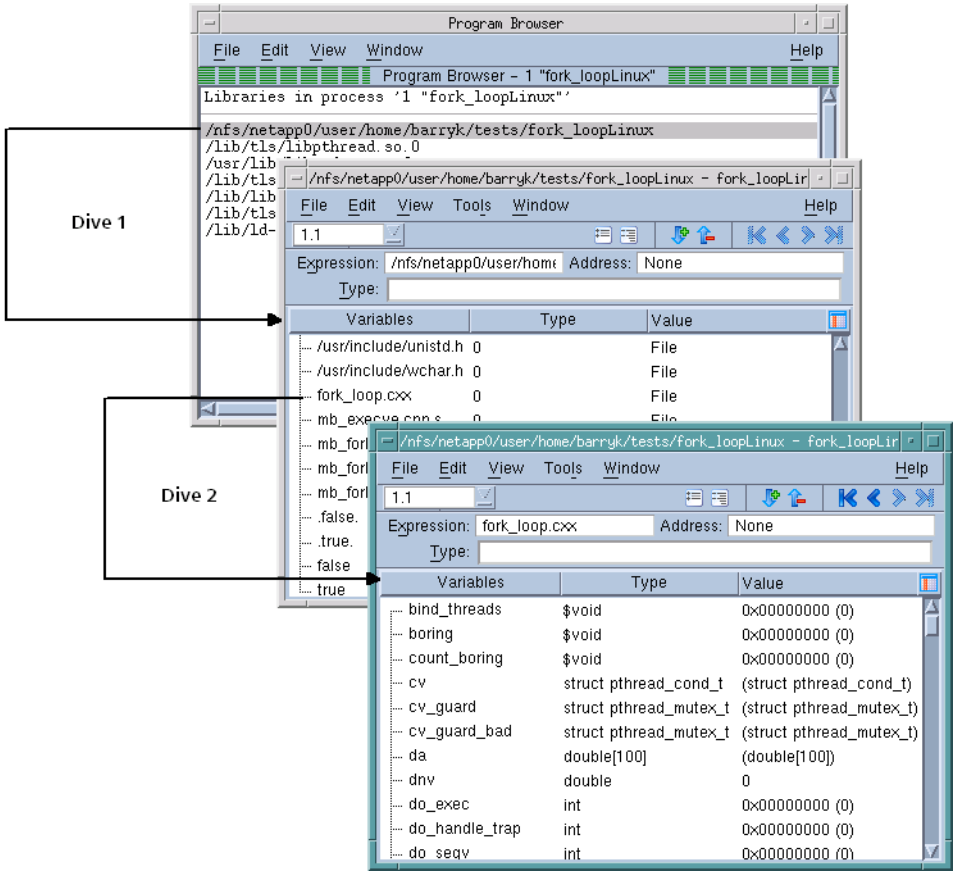
Freezing the address lets you continually reevaluate what is at that address as execution progresses. Here are two situations where you might want to do this:

- You need to look at a heap address access through a set of dive operations rooted in a stack frame that has become stale.
- You dive on a ***this** pointer to see the actual value after ***this** goes stale.

Browsing for Variables

The Process Window **Tools > Program Browser** command displays a window that contains all your executable's components. By clicking on a library or program name, you can access all of the variables contained in it.

Figure 125, Program Browser and Variable Windows (Part 1)

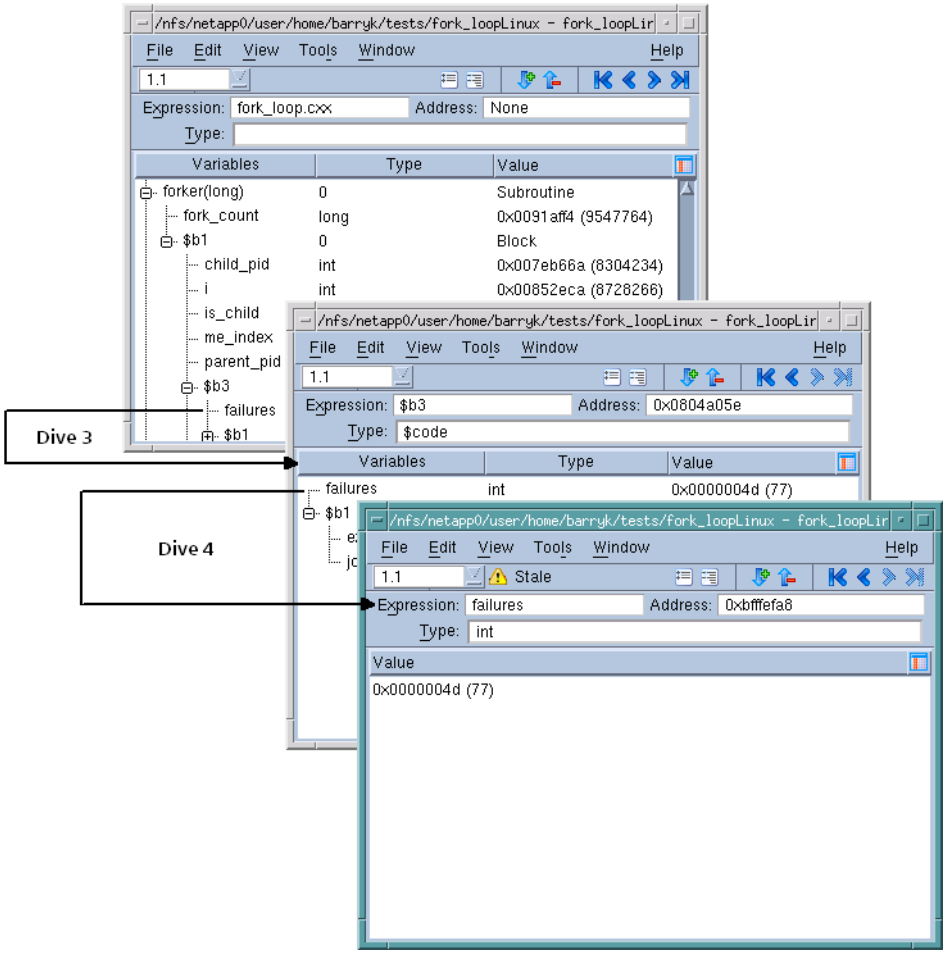


The window at the top of the figure shows programs and libraries that are loaded. If you have loaded more than one program with the **File > Debug New Program** command, TotalView displays information only for the currently selected process. After diving on an entry in this window (labeled **Dive 1**), TotalView displays a Variable Window that contains a list of files that make up the program, as well as other related information.

Diving on an entry in this Variable Window (**Dive 2** in this figure) changes the display to contain variables and other information related to the file. A list of functions defined within the program is at the end of this list.

Diving on a function changes the Variable Window again. The window shown at the top of the next figure was created by diving on one of these functions. The window shown in the center is the result of diving on a block in that subroutine. The bottom window shows a variable.

Figure 126, Program Browser and Variable Window (Part 2)



If you dive on a line in a Variable Window, the new contents replace the old contents, and you can use the undive/redive buttons to move back and forth.

If you are examining a complex program with large numbers of subroutines at file scope, often a result of a large number of include files and/or template class expansions, you may experience a performance slowdown. By default, the windows in this view display as much information as possible, including all symbols for all subroutines in a file scope. You can restrict views to initially show only the names of subroutines within a file scope by adding this to your `.tvdrc` file:

```
dset TV::recurse_subroutines false
```

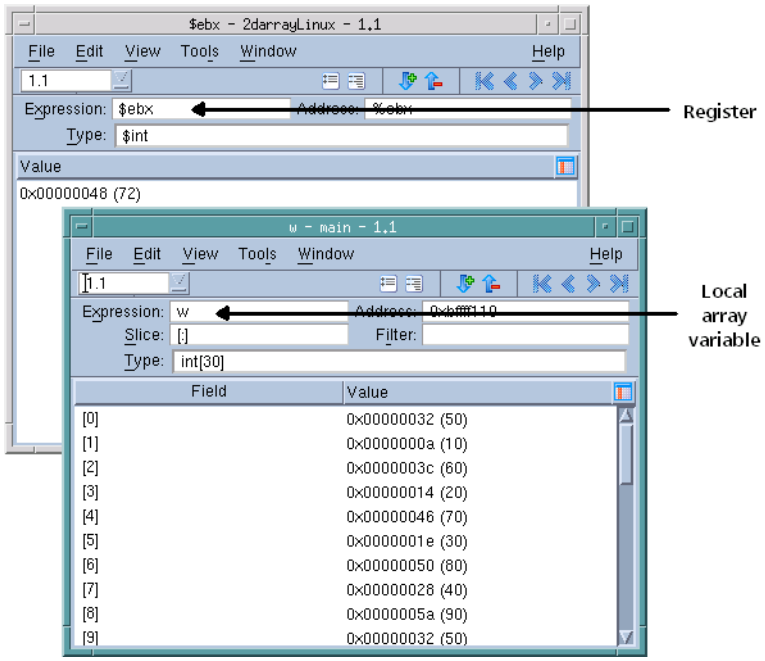
You can then still examine the symbols within a particular subroutine by diving on that subroutine.

RELATED TOPICS	
Diving in variable windows	Diving in Variable Windows on page 266
Details on the Variable Window	The "Variable Window" in the in-product Help

Displaying Local Variables and Registers

In the Stack Frame Pane, diving on a function's parameter, local variable, or register displays information in a Variable Window. You can also dive on parameters and local variables in the Source Pane. The displayed Variable Window shows the name, address, data type, and value for the object.

Figure 127, Diving on Local Variables and Registers



The window at the top of the figure shows the result of diving on a register, while the bottom window shows the results of diving on an array variable.

CLI: dprint *variable*
This command lets you view variables and expressions without having to select or find them.

You can also display local variables by using the **View > Lookup Variable** command.

RELATED TOPICS	
Diving in variable windows	Diving in Variable Windows on page 266
Using the Process Window	Using the Process Window on page 157
Details on the Variable Window	The "Variable Window" in the in-product Help

Interpreting the Status and Control Registers

The Stack Frame Pane in the Process Window lists the contents of CPU registers for the selected frame—you might need to scroll past the stack local variables to see them.

CLI: dprint *register*
You must quote the initial $ character in the register name; for example, **dprint \$r1**.

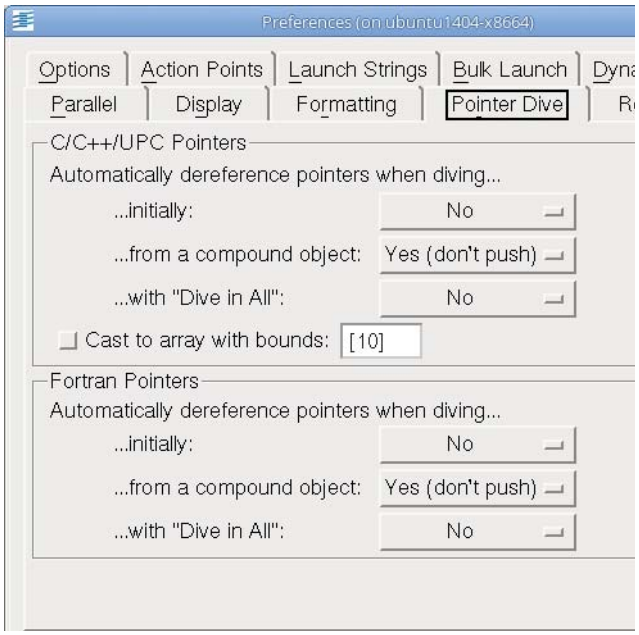
For your convenience, TotalView displays the bit settings of many CPU registers symbolically. For example, TotalView symbolically displays registers that control rounding and exception enable modes. You can edit the values of these registers and then resume program execution. For example, you might do this to examine the behavior of your program with a different rounding mode.

Since the registers that are displayed vary from platform to platform, see "*Architectures*" in the *Classic TotalView Reference Guide* for information on how TotalView displays this information on your CPU. For general information on editing the value of variables (including registers), see Displaying Areas of Memory on page 262. To learn about the meaning of these registers, see the documentation for your CPU.

Dereferencing Variables Automatically

In most cases, you want to see what a pointer points to, rather than what the value of its variable is. Using the controls on the **File > Preferences** Pointer Dive tab, you can tell TotalView to automatically dereference pointers (Figure 128).

Figure 128, File > Preferences Pointer Dive Page



Dereferencing pointers is especially useful when you want to visualize the data linked together with pointers, since it can present the data as a unified array. Because the data appears as a unified array, you can use TotalView's array manipulation commands and the Visualizer to view the data.

Each pulldown list on the Pointer Dive tab has three settings: **No**, **Yes**, and **Yes (don't push)**. **No** means do not automatically dereference pointers. **Yes** means automatically dereference pointers, and allow use of the **Back** command to see the undereferenced pointer value. **Yes (don't push)** also enables automatic dereferencing, but disallows use of the **Back** command to see the pointer value.

CLI: TV::auto_array_cast_bounds
TV::auto_deref_in_all_c
TV::auto_deref_in_all_fortran
TV::auto_deref_initial_c
TV::auto_deref_initial_fortran
TV::auto_deref_nested_c
TV::auto_deref_nested_fortran

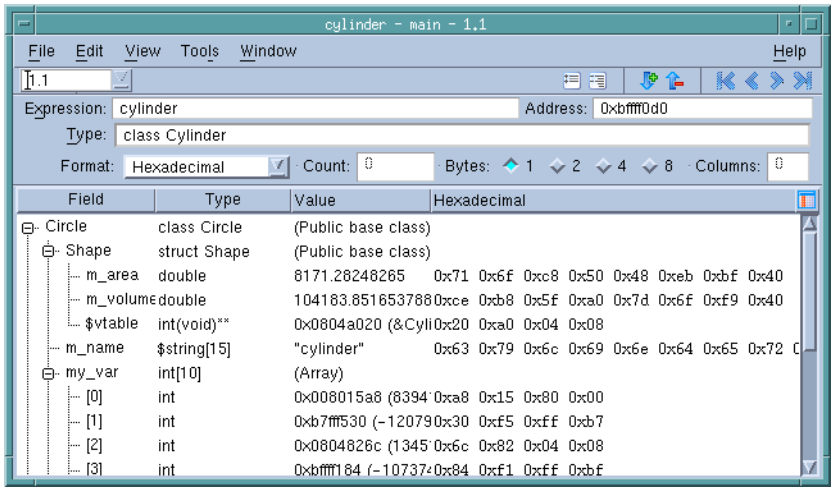
Automatic dereferencing can occur in the following situations:

- When TotalView *initially* displays a value.
- When you dive on a value in an aggregate or structure.
- When you use the **Dive in All** command.

Examining Memory

TotalView lets you display the memory used by a variable in different ways. If you select the **View > Examine Format > Structured** or **View > Examine Format > Raw** commands from within the Variable Window, TotalView displays raw memory contents. Figure 129 shows a structured view.

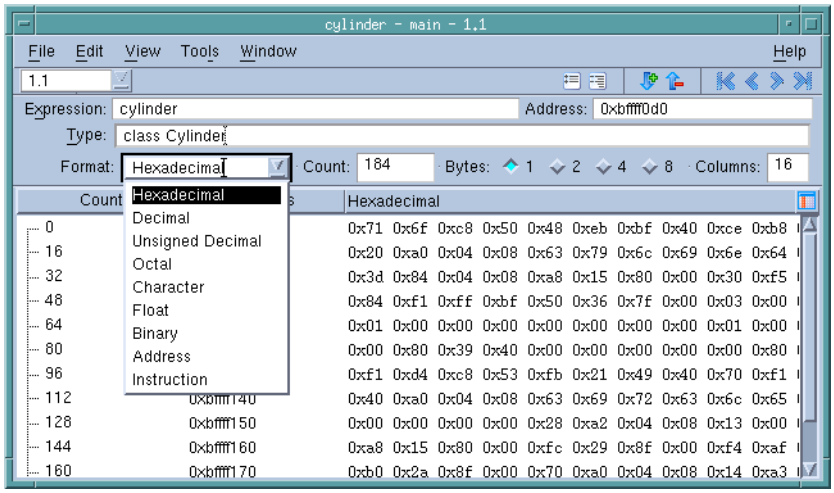
Figure 129, View > Examine Format > Stuctured Display



NOTE: The way this command displays data is similar to the way dump commands such as **od** that exist in your operating system display data.

When displaying a structured view, the left portion of the Variable Window shows the elements of the data, whether it be a structure or an array. The right portion shows the value of the data in the way that it is normally displayed within TotalView. The right-most column displays the raw memory data. By default, this information is displayed in hexadecimal. However, you can change it to other formats by selecting a representation within the **Format** pulldown. Figure 130 shows a raw display with this pulldown extended:

Figure 130, View > Examine Format > Raw Display



In either the raw or structured display, you can change the number of bytes grouped together and the range of memory being displayed.

If you select the **View > Block Status** command, TotalView will also give you additional information about memory. For example, you are told if the memory is in a **text**, **data**, or **bss** section. (If you see **unknown**, you are probably seeing a stack variable.)

In addition, if you right-click on the header area of the table, a context menu lets you add a **Status** column. This column contains information such as “Allocated”, “PostGuard”, “Corrupted PreGuard”, etc.

If you have enabled the Memory Debugger, this additional information includes whether memory is allocated or deallocated, or being used by a guard block, or hoarded.

Displaying Areas of Memory

You can display areas of memory using hexadecimal, octal, or decimal values. Do this by selecting the **View > Lookup Variable** command, and then entering one of the following in the dialog box that appears:

- **An address**

When you enter a single address, TotalView displays the word of data stored at that address.

CLI: dprint *address*

- **A pair of addresses**

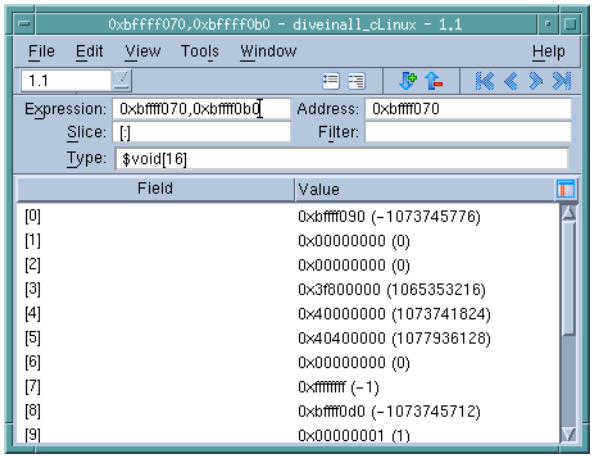
  When you enter a pair of addresses, TotalView displays the data (in word increments) from the first to the last address. To enter a pair of addresses, enter the first address, a comma, and the last address.

  CLI: dprint *address,address*

NOTE: All octal constants must begin with 0 (zero). Hexadecimal constants must begin with 0x.

The Variable Window for an area of memory displays the address and contents of each word.

Figure 131, Variable Window for an Area of Memory



TotalView displays the memory area's starting location at the top of the window's data area. In the window, TotalView displays information in hexadecimal and decimal notation.

If a Variable Window is already being displayed, you can change the type to **$void** and add an array specifier. If you do this, the results are similar to what is shown in this figure.

You can also edit the value listed in the **Value** field for each machine instruction.

Displaying Machine Instructions

You can display the machine instructions for entire routines as follows:

- Dive on the address of an assembler instruction in the Source Pane (such as **main+0x10** or **0x60**). A Variable Window displays the instructions for the entire function, and highlights the instruction you dove on.
- Dive on the PC in the Stack Frame Pane. A Variable Window displays the instructions for the entire function that contains the PC, and also highlights the instruction pointed to by the PC.
- Cast a variable to type **$code** or array of **$code**. For example:

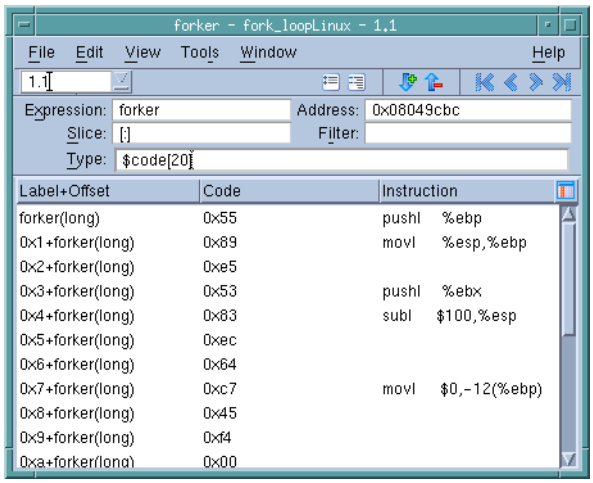
  ```
  $code[20]
  ```

  displays twenty code instructions, as shown in Figure 132.

The Variable Window lists the following information about each machine instruction:

Offset+Label	The symbolic address of the location as a hexadecimal offset from a routine name.
Code	The hexadecimal value stored in the location.
Instruction	The instruction and operands stored in the location.

Figure 132, Variable Window with Machine Instructions



Rebinding the Variable Window

When you restart your program, TotalView must identify the thread in which the variable existed. For example, suppose variable **my_var** was in thread 3.6. When you restart your program, TotalView tries to rebind the thread to a newly created thread. Because the order in which the operating system starts and executes threads can differ, there's no guarantee that the thread 3.6 in the current context is the same thread as what it was previously. Problems can occur. To correct rebinding issues, use the **Threads** box in the upper left-hand corner of the Variable Window to specify the thread to which you want to bind the variable.

Another way to use the **Threads** box is to change to a different thread to see the variable or expression's value there. For example, suppose variable **my_var** is being displayed in thread 3.4. If you type 3.5 in the Threads box, TotalView updates the information in the Expression List Window so that it is what exists in thread 3.5.

Closing Variable Windows

When you finish analyzing the information in a Variable Window, use the **File > Close** command to close the window. You can also use the **File > Close Similar** command to close all Variable Windows.

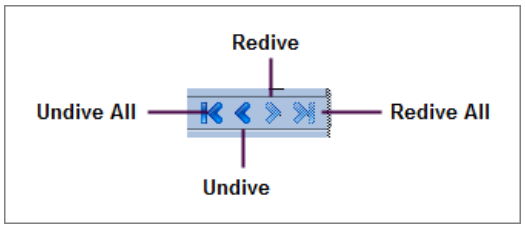
Diving in Variable Windows

If the variable being displayed in a Variable Window is a pointer, structure, or array, you can dive on the value. This new dive, which is called a *nested dive*, tells TotalView to replace the information in the Variable Window with information about the selected variable. If this information contains nonscalar data types, you can also dive on these data types. Although a typical data structure doesn't have too many levels, repeatedly diving on data lets you follow pointer chains. That is, diving lets you see the elements of a linked list.

TotalView can display a member of an array of structures as a single array across all the structures. See Displaying an Array of Structure's Elements on page 268 for more information.

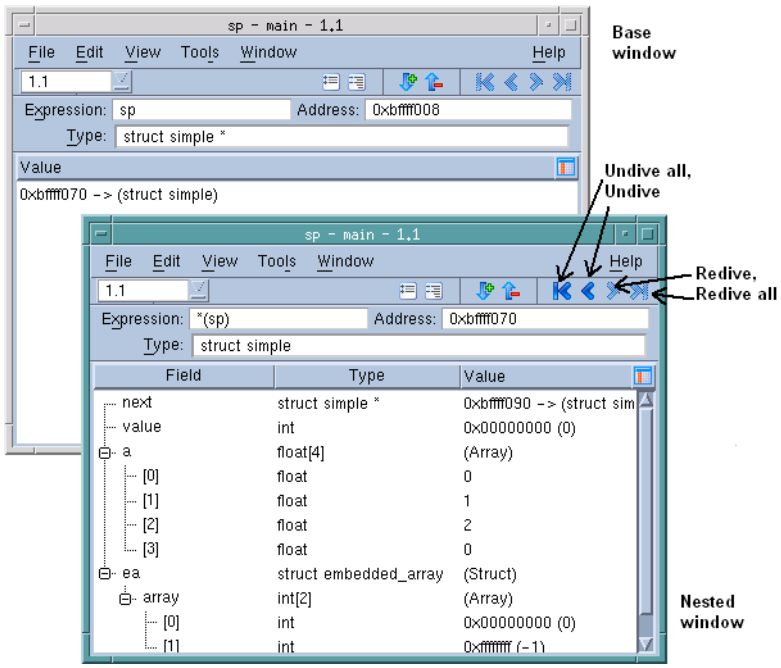
TotalView remembers your dives. This means that you can use the undive/redive buttons to view where you already dove.

Figure 133, Undive/Redive Buttons



The following figure shows a Variable Window after diving into a pointer variable named **sp** with a type of **simple***. The first Variable Window, which is called the *base window*, displays the value of **sp**. (This is **Window 1** in Figure 134.)

Figure 134, Nested Dives



The nested dive window (**Window 2** in this figure) shows the structure referenced by the **simple*** pointer.

You can manipulate Variable Windows and nested dive windows by using the undive/redive buttons, as follows:

- To undive from a nested dive, click the undive arrow button. The previous contents of the Variable Window appear.
- To undive from all your dive operations, click the undive all arrow button.
- To redive after you undive, click the redive arrow button. TotalView restores a previously executed dive operation.
- To redive from all your undive operations, click on the **Redive All** arrow button.

If you dive on a variable that already has a Variable Window open, the Variable Window pops to the top of the window display.

If you select the **Window > Duplicate** command, a new Variable Window appears, which is a duplicate of the current Variable Window.

RELATED TOPICS

Diving into objects	About Diving into Objects on page 164

Displaying an Array of Structure's Elements

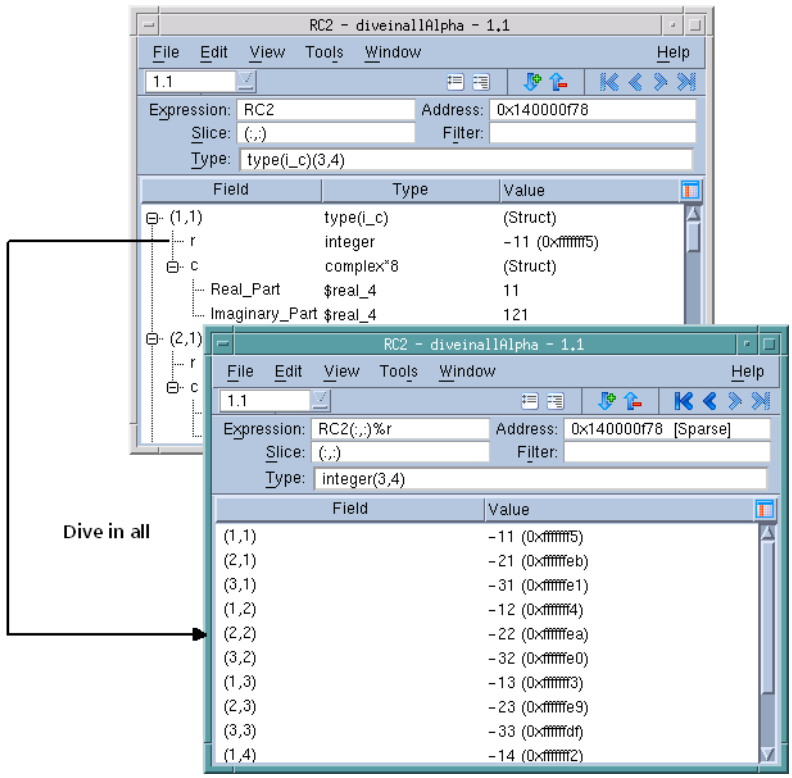
The **View > Dive In All** command, which is also available when you right-click on a field, can display an element in an array of structures as if it were a simple array. For example, suppose you have the following Fortran definition:

```
type i_c
   integer r
   complex c
end type i_C

type(i_c), target :: rc2(3,4)
```

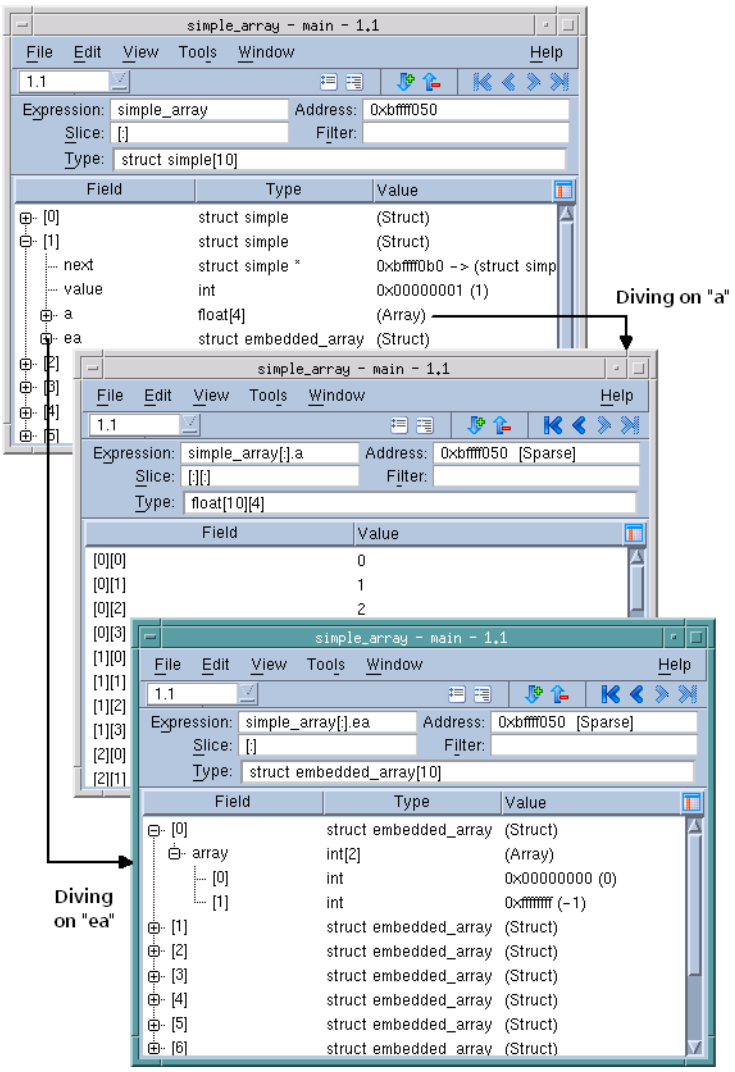
After selecting an **r** element, select the **View > Dive In All** command. TotalView displays all of the **r** elements of the **rc2** array as if they were a single array.

Figure 135, Displaying a Fortran Structure



The **View > Dive in All** command can also display the elements of a C array of structures as arrays. Figure 136 shows a unified array of structures and a multidimensional array in a structure.

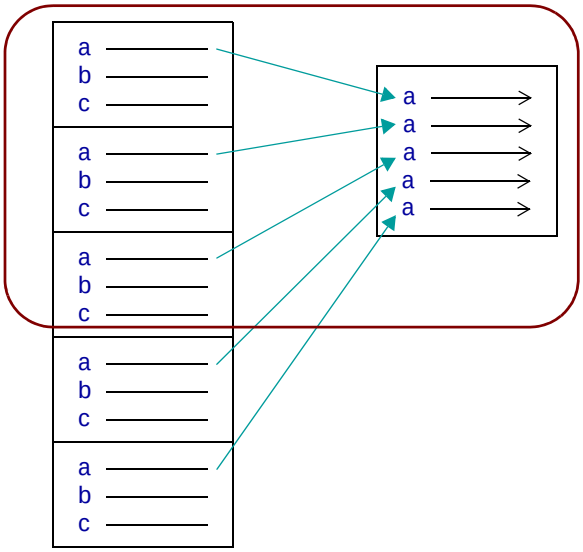
Figure 136, Displaying C Structures and Arrays



NOTE: As the array manipulation commands (described in Examining Arrays) generally work on what's displayed and not what is stored in memory, TotalView commands that refine and display array information work on this virtual array. For example, you can visualize the array, obtain statistics about it, filter elements in it, and so on.

Figure 137 is a high-level look at what a dive in all operation does.

Figure 137, Dive in All



In this figure, the rounded rectangle represents a Variable Window. On the left is an array of five structures. After you select the **Dive in All** command with element **a** selected, TotalView *replaces* the contents of your Variable Window with an array that contains all of these **a** elements.

RELATED TOPICS	
Arrays	Examining Arrays on page 312
Structures	Viewing Structures on page 287

Changing What the Variable Window Displays

When TotalView displays a Variable Window, the **Expression** field contains either a variable or an expression. Technically, a variable is also an expression. For example, **my_var.an_element** is actually an addressing expression. Similarly, **my_var.an_element[10]** and **my_var[10].an_element** are also expressions, since both TotalView and your program must figure out where the data associated with the element resides.

The expression in the **Expression** field is dynamic. That is, you can tell TotalView to evaluate what you enter before trying to obtain a memory address. For example, if you enter **my_var.an_element[i]**, TotalView evaluates the value of **i** before it redisplays your information. A more complicated example is **my_var.an_element[i+1].** In this example, TotalView must use its internal expression evaluation system to create a value before it retrieves data values.

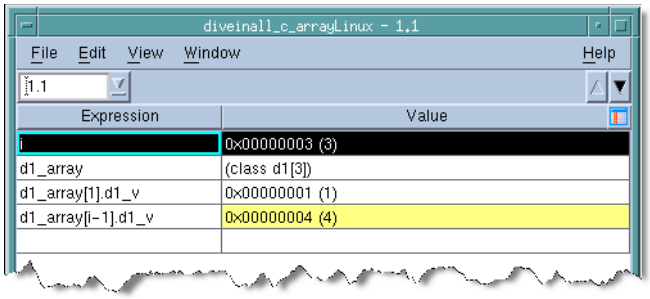
You can replace the variable expression with something completely different, such as **i+1**, and TotalView simply displays the value produced by evaluating the expression.

Evaluating Expressions on page 360 has a discussion of the evaluation system and typing expressions in an eval point in the **Tools > Evaluate** Window. In contrast, the expressions you can type in the Expression List Window are restricted, with the principal restriction being that what you type cannot have side effects. For example, you cannot use an expression that contains a function call or an operator that changes memory, such as **++** or **--**.

Viewing a List of Variables

As you debug your program, you may want to monitor a variable's value as your program executes. For many types of information, the Expression List Window offers a more compact display than the Variable Window for displaying scalar variables.

Figure 138, The Tools > Expression List Window



For more information, see the **Tools > Expression List Command**.

The topics discussing the Expression List Window are:

- Entering Variables and Expressions on page 272
- Seeing Variable Value Changes in the Expression List Window on page 274
- Entering Expressions into the Expression Column on page 275
- Using the Expression List with Multi-process/Multi-threaded Programs on page 277
- Reevaluating, Reopening, Rebinding, and Restarting on page 277
- Seeing More Information on page 278
- Sorting, Reordering, and Editing on page 279

Entering Variables and Expressions

To display an initial, empty window, select the **Tools > Expression List** command.

You can place information in the first column of the Expression List Window in the following ways:

- Enter it into a blank cell in the **Expression** column. When you do this, the context is the current PC in the process and thread indicated in the **Threads** box. If you type **my_var** in the window shown in the previous section, you would type the value of **my_var** in process 1, thread 1.
- Right-click on a line in the Process Window Source or Stack Frame Panes. From the displayed context menu, select **Add to Expression List**. Here is the context menu that TotalView displays in the Source Pane:

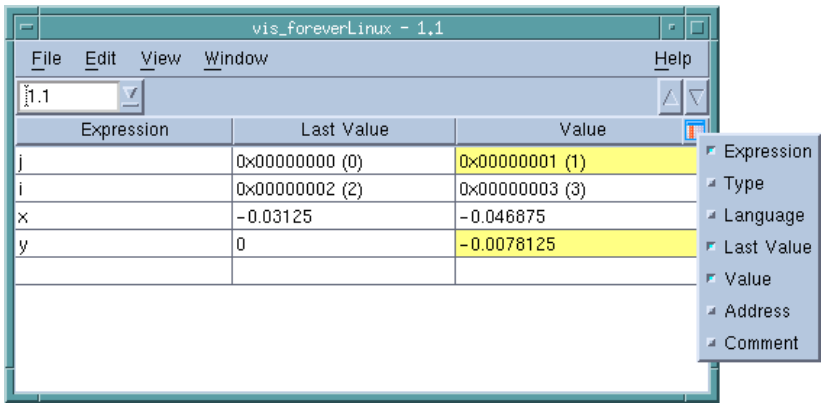
- Right-click on something in a Variable Window. Select **Add to Expression List** from the displayed context menu. You can also use the **View > Add to Expression List** command.

When you enter information in the **Tools > Expression List** Window, where you place the cursor and what you select make a difference. If you click on a variable or select a row in the Variable Window, TotalView adds that variable to the Expression List Window. If you instead select text, TotalView adds that text. What's the difference? Figure 138 on page 272 shows three variations of **d1_array**, each obtained in a different way, as follows:

- The first entry was added by selecting just part of what was displayed in the Source Pane.
- The second entry was added by selecting a row in the Variable Window.
- The third entry was added by clicking at a random point in the variable's text in the Source Pane.

You can tell TotalView to look for a variable in the scope that exists when your program stops executing, rather than keeping it locked to the scope from which it was added to the **Tools > Expression List** Window. Do this by right-clicking an item, then selecting **Compilation Scope > Floating** from the context menu.

Figure 139, Expression List Window Context Menu



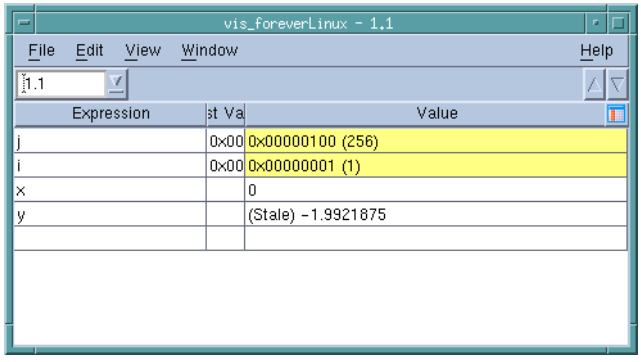
For more information, see Viewing Variables in Different Scopes as Program Executes on page 252.

Seeing Variable Value Changes in the Expression List Window

TotalView can tell you when a variable's value changes in several ways.

- When your program stops at a breakpoint, TotalView adds a yellow highlight to the variable's value if it has changed, Figure 140.

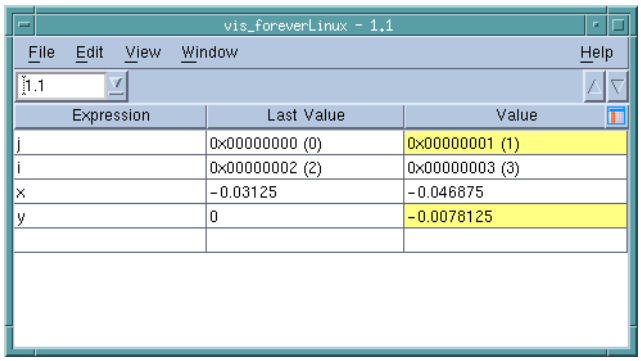
Figure 140, Expression List Window With "Change" Highlighting



If the thread is stopped for another reason—for example, you've stepped the thread—and the value has changed, TotalView does not add yellow highlighting to the line.

- You can tell TotalView to display the **Last Value** column. Do this by selecting **Last Value** in the column menu, which is displayed after you click on the column menu ( ) icon.

Figure 141, Variable Window Showing Last Value Column

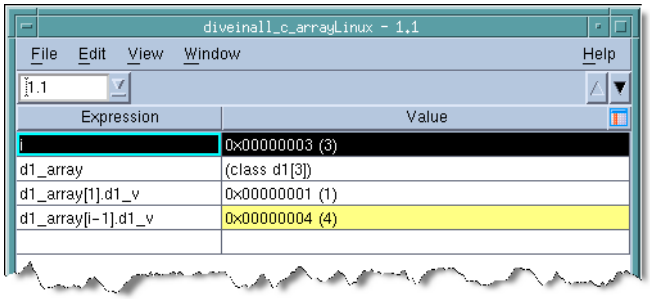


Notice that TotalView has highlighted all items that have changed within an array. In a similar fashion it can show the individual items that have changed within a structure.

Entering Expressions into the Expression Column

The following Expression List Window shows four different types of expressions.

Figure 142, The Tools > Expression List Window



The expressions in this window are:

i — A variable with one value. The **Value** column shows its value.

d1_array — An aggregate variable; that is, an array, a structure, a class, and so on. Its value cannot be displayed in one line. Consequently, TotalView just gives you some information about the variable. To see more information, dive on the variable. After diving, TotalView displays the variable in a Variable Window.

When you place an aggregate variable in the **Expression** column, you need to dive on it to get more information.

d1_array[1].d1_v

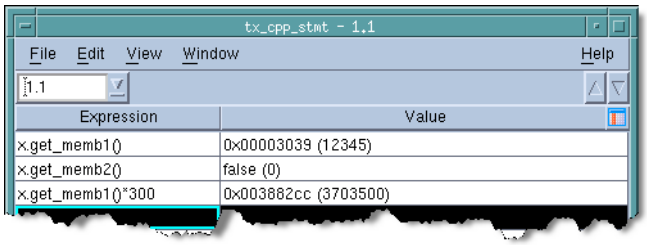
An element in an array of structures. If TotalView can resolve what you enter in the **Expression** column into a single value, it displays a value in the **Value** column. If TotalView can't, it displays information in the same way that it displays information in the **d1_array** example.

An element in an array of structures. This expression differs from the previous example in that the array index is an expression. Whenever execution stops in the current thread, TotalView re-evaluates the **i-1** expression. This means that TotalView might display the value of a different array item every time execution stops.

The expressions you enter cannot include function calls.

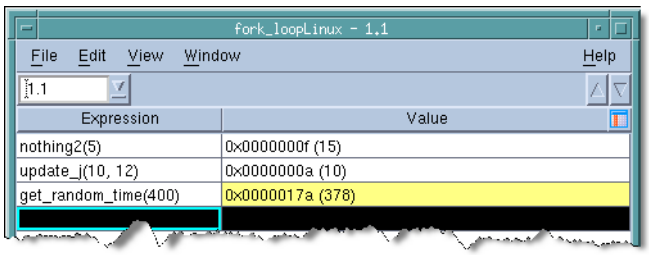
You can also enter methods and functions within an Expression. Figure 143 shows two get methods and a get method used in an expression.

Figure 143, Using Methods in the Tools > Expression List Window



In a similar fashion, you can even directly enter functions, Figure 144.

Figure 144, Using Functions in the Tools > Expression List Window



Using the Expression List with Multi-process/Multi-threaded Programs

You can change the thread in which TotalView evaluates your expressions by typing a new thread value in the **Threads** box at the top of the window. A second method is to select a value by using the drop-down list in the **Threads** box.

When you use an **Add to Expression List** command, TotalView checks whether an Expression List Window is already open for the current thread. If one is open, TotalView adds the variable to the bottom of the list. If an Expression List Window isn't associated with the thread, TotalView duplicates an existing window, changes the thread of the duplicated window, and then adds the variable to all open **Tools > Expression List** Windows. That is, you have two **Tools > Expression List** Windows. Each has the same list of expressions. However, the results of the expression evaluation differ because TotalView is evaluating them in different threads.

In all cases, the list of expressions in all **Tools > Expression List** Windows is the same. What differs is the thread in which TotalView evaluates the window's expressions.

Similarly, if TotalView is displaying two or more **Tools > Expression List** Windows, and you send a variable from yet another thread, TotalView adds the variable to all of them, duplicates one of them, and then changes the thread of the duplicated window.

Reevaluating, Reopening, Rebinding, and Restarting

This section explains what happens in the **Tools > Expression List** Window as TotalView performs various operations.

Reevaluating Contents

TotalView reevaluates the value of everything in the **Tools > Expression List** Window **Expression** column whenever your thread stops executing. More precisely, if a thread stops executing, TotalView reevaluates the contents of all **Tools > Expression List** Windows associated with the thread. In this way, you can see how the values of these expressions change as your program executes.

You can use the **Window > Update All** command to update values in all other **Tools > Expression List** Windows.

Reopening Windows

If you close all open **Tools > Expression List** Windows and then reopen one, TotalView remembers the expressions you added previously. That is, if the window contains five variables when you close it, it has the same five variables when you open it. The thread TotalView uses to evaluate the window's contents is based on the Process Window from which you invoked the **Tools > Expressions List** command.

Rebinding Windows

The values displayed in an **Expression List** Window are the result of evaluating the expression in the thread indicated in the **Threads** box at the top of the window. To change the thread in which TotalView evaluates these expressions, you can either type a new thread value in the **Threads** box or select a thread from the pulldown list in the **Threads** box. (Changing the thread to evaluate expressions in that thread's context is called *rebinding*.)

Restarting a Program

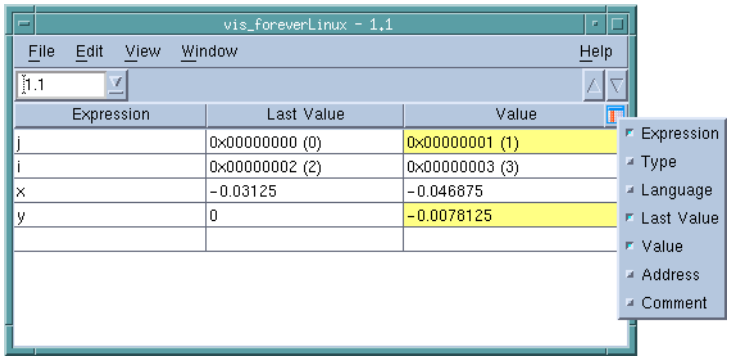
When you restart your program, TotalView attempts to rebind the expressions in a **Tools > Expression List** Window to the *correct* thread. Unfortunately, it is not possible to select the right thread with 100% accuracy. For example, the order in which your operating system creates threads can differ each time you run your program. Or, your program's logic can cause threads to be created in a different order.

You may need to manually change the thread by using the **Threads** box at the top of the window.

Seeing More Information

When you first open the **Tools > Expression List** Window, it contains two columns, but TotalView can display other columns. If you right-click on a column heading line, TotalView displays a context menu that indicates all possible columns. Clicking on a heading name listed in the context menu changes if from displayed to hidden or vice versa.

Figure 145, The Tools > Expression List Window Showing Column Selector



Even when you add additional columns, the **Expression List** Window might not show you what you need to know about a variable. If you dive on a row (or select **Dive** from a context menu), TotalView opens a Variable Window for what you just dove on.

You can combine the **Expression List** Window and diving to bookmark your data. For example, you can enter the names of structures and arrays. When you want to see information about them, dive on the name. In this way, you don't have to clutter up your screen with the Variable Windows that you don't need to refer to often.

Sorting, Reordering, and Editing

This section describes operations you can perform on **Tools > Expression List** Window data.

Sorting Contents

You can sort the contents of the **Tools > Expression List** Window by clicking on the column header. After you click on the header, TotalView adds an indicator that shows that the column was sorted and the way in which it was sorted. In the figure in the previous topic, the **Value** column is sorted in ascending order.

Reordering Row Display

The up and down arrows () on the right side of the **Tools > Expression List** Window toolbar let you change the order in which TotalView displays rows. For example, clicking the down arrow moves the currently selected row (indicated by the highlight) one row lower in the display.

Editing Expressions

You can change an expression by clicking in it, and then typing new characters and deleting others. Select **Edit > Reset Defaults** to remove all edits you have made. When you edit an expression, TotalView uses the scope that existed when you created the variable.

Changing Data Type

You can edit an expression's data type by displaying the **Type** column and making your changes. Select **Edit > Reset Defaults** to remove all edits you have made.

Changing an Expression's Value

You can change an expression's value if that value is stored in memory by editing the contents of the **Value** column.

About Other Commands

You can also use the following commands when working with expressions:

Edit > Delete Expression

Deletes the selected row. This command is also on a context (right-click) menu. If you have more than one **Expression List** Window open, deleting a row from one window deletes the row from all open windows.

Edit > Delete All Expressions

Deletes all of the **Expression List** Window rows. If you have more than one **Expression List** Window open, deleting all expressions from one window deletes all expressions in all windows.

View > Dive

Displays the expression or variable in a Variable Window. Although this command is also on a context menu, you can just double-click or middle-click on the variable's name instead.

Edit >Duplicate Expression

Duplicates the selected column. You would duplicate a column to see a similar variable or expression. For example, if **myvar_looks_at[i]** is in the **Expression** column, duplicating it and then modifying the new row is an easy way to see **myvar_looks_at[i]** and **myvar_looks_at[i+j-k]** at the same time.

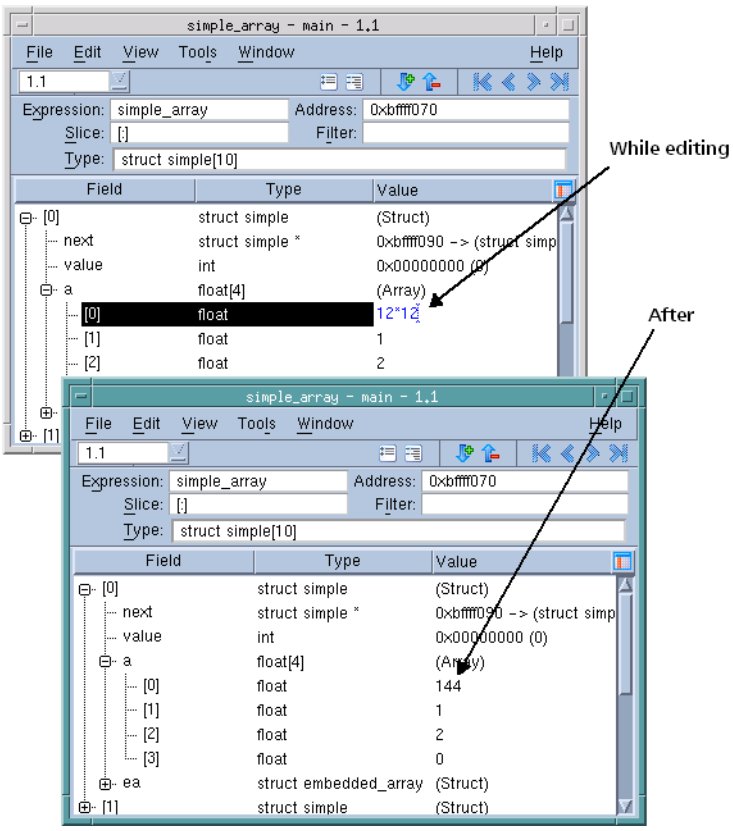
This command is also on a context menu.

Changing the Values of Variables

You can change the value of any variable or the contents of any memory location displayed in a Variable Window, Expression List Window, or Stack Frame Pane by selecting the value and typing the new value. In addition to typing a value, you can also type an expression. For example, you can enter **12*12** as shown in the following figure. You can include logical operators in all TotalView expressions.

CLI: set my_var [expr 1024*1024]
dassign int8_array(3) $my_var

Figure 146, Using an Expression to Change a Value



In most cases, you can edit a variable's value. If you right-click on a variable and the **Change Value** command isn't faded, you can edit the displayed value.

TotalView does not let you directly change the value of bit fields; you can use the **Tools > Evaluate** Window to assign a value to a bit field. See Evaluating Expressions on page 360.

CLI: Tcl lets you use operators such as **&** and | to manipulate bit fields on Tcl values.

RELATED TOPICS

Editing text in source code	Editing Source Text on page 176
Details on the Variable Window	The "Variable Window" in the in-product Help

Changing a Variable's Data Type

The data type declared for the variable determines its format and size (amount of memory). For example, if you declare an **int** variable, TotalView displays the variable as an integer.

The following sections discuss the different aspects of data types:

You can change the way TotalView displays data in the Variable Window and the Expression List Window by editing the data type. This is known as *casting*. TotalView assigns types to all data types, and in most cases, they are identical to their programming language counterparts.

When a C or C++ variable is displayed in TotalView, the data types are identical to their C or C++ type representations, except for pointers to arrays. TotalView uses a simpler syntax for pointers to arrays. (See Viewing Pointers to Arrays on page 286.) Similarly, when Fortran is displayed in TotalView, the types are identical to their Fortran type representations for most data types, including **INTEGER**, **REAL**, **DOUBLE PRECISION**, **COMPLEX**, **LOGICAL**, and **CHARACTER**.

If the window contains a structure with a list of fields, you can edit the data types of the listed fields.

NOTE: When you edit a data type, TotalView changes how it displays the variable in the current window. Other windows listing the variable do not change.

Displaying C and C++ Data Types

The syntax for displaying data is identical to C and C++ language cast syntax for all data types except pointers to arrays. That is, you use C and C++ cast syntax for data types. For example, you can cast using types such as **int**, **char**, **unsigned**, **float**, **double**, **union**, all named **struct** types, and so on. In addition, TotalView has a built-in type called **$string**. Unless you tell it otherwise, TotalView maps **char** arrays to this type. (For information on wide characters, see Viewing Wide Character Arrays ($wchar Data Types) on page 291.)

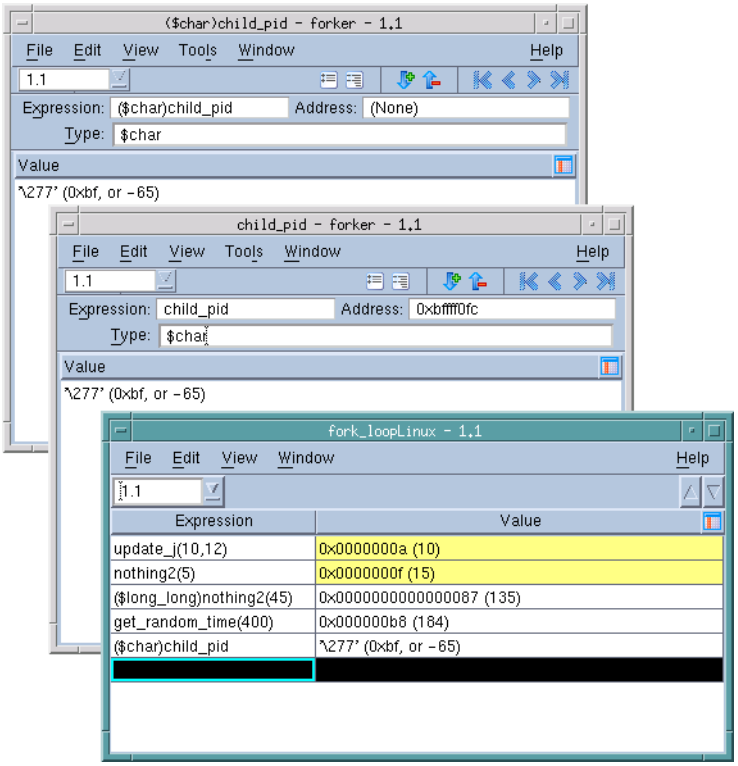
Read TotalView types from right to left. For example, **$string*[20]*** is a pointer to an array of 20 pointers to **$string**.

The following table shows some common TotalView data types:

Data Type String	Description
int	Integer
int*	Pointer to an integer
int[10]	Array of 10 integers
$string	Null-terminated character string
$string**	Pointer to a pointer to a null-terminated character string
$string*[20]*	Pointer to an array of 20 pointers to null-terminated strings

You can enter C and C++ Language cast syntax in the **Type** field. Figure 147 shows three different ways to cast:

Figure 147, Three Casting Examples



The two Variable Windows cast the same data in the same way. In the top-left window, a cast was used in the **Expression** field. In the other Variable Window, the data type was changed from **int** to **$char**. In the first cast, TotalView changed the **Type** for you. In the second, it did not alter the **Expression** field.

The Expression List Window contains two casting examples. The first casts a function's returned value to **long long**. The second is the same cast as was made in the two Variable Windows.

TotalView also lets you cast a variable into an array. In the GUI, add an array specifier to the **Type** declaration. For example, adding **[3]** to a variable declared as an **int** changes it into an array of three **int**s.

When TotalView displays some complex arrays and structures, it displays the compound object or array types in the Variable Window.

NOTE: Editing a compound object or array data type can produce undesirable results. TotalView tries to give you what you ask for, so if you get it wrong, the results are unpredictable. Fortunately, the remedy is quite simple: close the Variable Window and start over again.

The following sections discuss more complex data types.

Viewing Pointers to Arrays

Suppose you declared a variable **vbl** as a pointer to an array of 23 pointers to an array of 12 objects of type **mytype_t**. The C language declaration for this is:

```
mytype_t (*(*vbl)[23]) [12];
```

Here is how you would cast the **vbl** variable to this type:

```
(mytype_t (*(*)[23])[12])vbl
```

The TotalView cast for **vbl** is:

```
mytype_t[12]*[23]*
```

Viewing Arrays

When you specify an array, you can include a lower and upper bound separated by a colon (:).

NOTE: See Examining Arrays on page 312 for more information on arrays.

By default, the lower bound for a C or C++ array is **0**, and the lower bound for a Fortran array is **1**. In the following example, an array of ten integers is declared in C and then in Fortran:

```
int a[10];
integer a(10)
```

The elements of the array range from **a[0]** to **a[9]** in C, while the elements of the equivalent Fortran array range from **a(1)** to **a(10)**.

TotalView also lets you cast a variable to an array. In the GUI, just add an array specifier to the **Type** declaration. For example, adding **(3)** to a variable declared as an **integer** changes it to an array of three **integer**s.

When the lower bound for an array dimension is the default for the language, TotalView displays only the extent (that is, the number of elements in the dimension). Consider the following Fortran array declaration:

```
integer a(1:7,1:8)
```

Since both dimensions of this Fortran array use the default lower bound, which is **1**, TotalView displays the data type of the array by using only the extent of each dimension, as follows:

```
integer(7,8)
```

If an array declaration doesn't use the default lower bound, TotalView displays both the lower bound and upper bound for each dimension of the array. For example, in Fortran, you declare an array of integers with the first dimension ranging from -1 to 5 and the second dimension ranging from 2 to 10, as follows:

```
integer a(-1:5,2:10)
```

TotalView displays this the same way.

When editing an array's dimension, you can enter just the extent (if using the default lower bound), or you can enter the lower and upper bounds separated by a colon.

TotalView also lets you display a subsection of an array, or filter a scalar array for values that match a filter expression. See Displaying Array Slices on page 313 and Filtering Array Data Overview on page 319 for more information.

Viewing typedef Types

TotalView recognizes the names defined with **typedef**, and displays these user-defined types; for example:

```
typedef double *dptr_t;
dptr_t p_vbl;
```

TotalView displays the type for **p_vbl** as **dptr_t**.

Viewing Structures

TotalView lets you use the **struct** keyword as part of a type string. In most cases, this is optional.

NOTE: This behavior depends upon which compiler you are using. In most cases, you'll see what is described here.

If you have a structure and another data type with the same name, however, you must include the **struct** keyword so that TotalView can distinguish between the two data types.

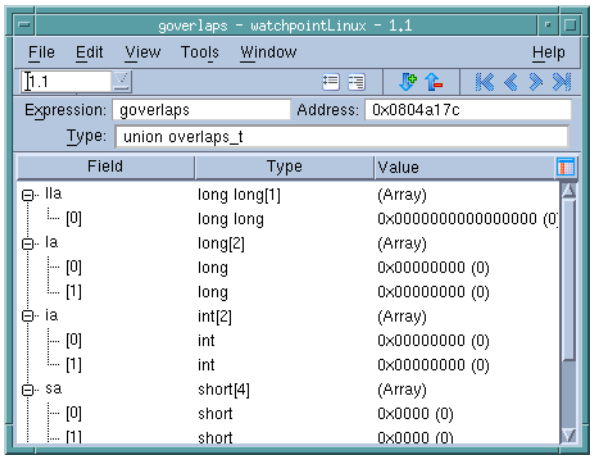
If you use a **typedef** statement to name a structure, TotalView uses the **typedef** name as the type string. Otherwise, TotalView uses the structure tag for the **struct**.

Viewing Unions

TotalView displays a union in the same way that it displays a structure. Even though the fields of a union are overlaid in storage, TotalView displays the fields on separate lines.

CLI: dprint *variable*

Figure 148, Displaying a Union



Casting Using the Built-In Types

TotalView provides a number of predefined types. These types are preceded by a **$**. You can use these built-in types anywhere you can use the ones defined in your programming language. These types are also useful in debugging executables with no debugging symbol table information. The following table describes the built-in types:

Type String	Language	Size	Description
$address	C	**void***	Void pointer (address).
$char	C	**char**	Character.
$character	Fortran	**character**	Character.
$code	C	**architecture-dependent**	Machine instructions. The size used is the number of bytes required to hold the shortest instruction for your computer.

Type String	Language	Size	Description
$complex	Fortran	**complex**	Single-precision floating-point complex number. The **complex** types contain a real part and an imaginary part, which are both of type **real**.
$complex_8	Fortran	**complex*8**	A **real*4**-precision floating-point complex number. The **complex*8** types contain a real part and an imaginary part, which are both of type **real*4**.
$complex_16	Fortran	**complex*16**	A **real*8**-precision floating-point complex number. The **complex*16** types contain a real part and an imaginary part, which are both of type **real*8**.
$double	C	**double**	Double-precision floating-point number.
$double_precision	Fortran	**double precision**	Double-precision floating-point number.
$extended	C	**architecture-dependent;** often **long double**	Extended-precision floating-point number. Extended-precision numbers must be supported by the target architecture. In addition, the format of extended floating point numbers varies depending on where it's stored. For example, the x86 register has a special 10-byte format, which is different than the in-memory format. Consult your vendor's architecture documentation for more information.
$float	C	**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.

Type String	Language	Size	Description
$long_long	C	**long long**	Long long integer.
$real	Fortran	**real**	Single-precision floating-point number. 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**, since different results can 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.
$wchar	C	*platform-specific*	Platform-specific wide character used by **wchar_t** data types
$wchar_s16	C	16 bits	Wide character whose storage is signed 16 bits (not currently used by any platform)
$wchar_u16	C	16 bits	Wide character whose storage is unsigned 16 bits
$wchar_s32	C	32 bits	Wide character whose storage is signed 32 bits
$wchar_u32	C	32 bits	Wide character whose storage is unsigned 32 bits
$wstring	C	*platform-specific*	Platform-specific string composed of **$wchar** characters
$wstring_s16	C	16 bits	String composed of **$wchar_s16** characters (not currently used by any platform)
$wstring_u16	C	16 bits	String composed of **$wchar_u16** characters
$wstring_s32	C	32 bits	String composed of **$wchar_s32** characters
$wstring_u32	C	32 bits	String composed of **$wchar_u32** characters

Viewing Character Arrays ($string Data Type)

If you declare a character array as **char vbl[*n*]**, TotalView automatically changes the type to **$string[*n*]**; that is, a null-terminated, quoted string with a maximum length of *n*. This means that TotalView displays an array as a quoted string of *n* characters, terminated by a null character. Similarly, TotalView changes **char*** declarations to **$string*** (a pointer to a null-terminated string).

Since most character arrays represent strings, the TotalView **$string** type can be very convenient. But if this isn't what you want, you can change the **$string** type back to a **char** (or **char[*n*]**), to display the variable as you declared it.

Viewing Wide Character Arrays ($wchar Data Types)

If you create an array of **wchar_t** wide characters, TotalView automatically changes the type to **$wstring[*n*]**; that is, it is displayed as a null-terminated, quoted string with a maximum length of *n*. For an array of wide characters, the null terminator is **L'0'**. Similarly, TotalView changes **wchar_t*** declarations to **$wstring*** (a pointer to a null-terminated string).

Figure 149, Displaying wchar_t Data

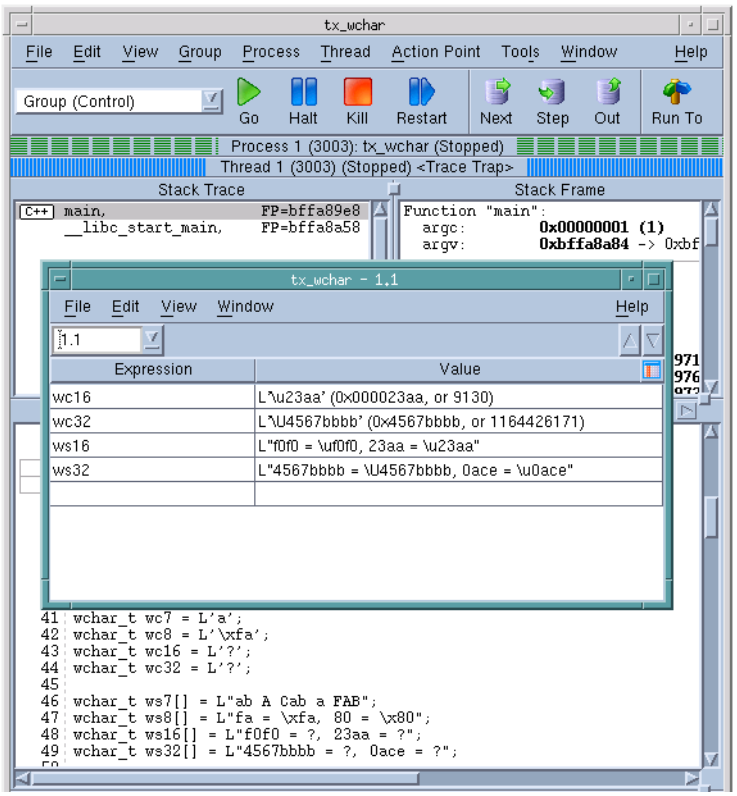


Figure 149 shows the declaration of two wide characters in the Process Window. The Expression List Window shows how TotalView displays their data. The **L** in the data indicates that TotalView is displaying a wide literal.

Since most wide character arrays represent strings, the **$wstring** type can be very convenient. But if this isn't what you want, you can change the **$wstring** type back to a **wchar_t** (or **wchar[*n*]** or **$wchar** or **$wchar[*n*]**), to display the variable as you declared it.

If the wide character uses from 9 to 16 bits, TotalView displays the character using the following universal-character code representation:

\uXXXX

X represents a hexadecimal digit. If the character uses from 17 to 32 bits, TotalView uses the following representation:

\UXXXXXXXX

NOTE: Platforms and compilers differ in the way they represent wchar_t. In consequence, TotalView allows you to see this information in platform-specific ways. For example, you can cast a string to $wstring_s16 or $wstring_s32. In addition, many compilers have problems either using wide characters or handing off information about wide characters so they can be interpreted by any debugger (not just TotalView). For information on supported compilers, see the document TotalView Supported Platforms in the TotalView distribution at `<installdir>/totalview.<version>/doc/pdf`, or on the TotalView documentation website.

Viewing Areas of Memory ($void Data Type)

TotalView uses the **$void** data type for data of an unknown type, such as the data contained in registers or in an arbitrary block of memory. The **$void** type is similar to the **int** type in the C Language.

If you dive on registers or display an area of memory, TotalView lists the contents as a **$void** data type. If you display an array of **$void** variables, the index for each object in the array is the address, not an integer. This address can be useful when you want to display large areas of memory.

If you want, you can change a **$void** to another type. Similarly, you can change any type to a **$void** to see the variable in decimal and hexadecimal formats.

Viewing Instructions ($code Data Type)

TotalView uses the **$code** data type to display the contents of a location as machine instructions. To look at disassembled code stored at a location, dive on the location and change the type to **$code**. To specify a block of locations, use **$code[*n*]**, where *n* is the number of locations being displayed.

RELATED TOPICS

Viewing assembler code	Viewing the Assembler Version of Your Code on page 173

Viewing Opaque Data

An opaque type is a data type that could be hidden, not fully specified, or defined in another part of your program. For example, the following C declaration defines the data type for **p** to be a pointer to **struct foo**, and **foo** is not yet defined:

```
struct foo;
struct foo *p;
```

When TotalView encounters a variable with an opaque type, it searches for a **struct**, **class**, **union**, or **enum** definition with the same name as the opaque type. If TotalView doesn't find a definition, it displays the value of the variable using an opaque type name; for example:

```
(Opaque foo)
```

Some compilers do not store sufficient information for TotalView to locate the type. This could be the reason why TotalView uses the opaque type.

You can tell TotalView to use the correct data type by having it read the source file. For example, if TotalView is showing you **(Opaque foo)** and you know that **struct foo** is defined in source file **foo.c**, use the **File > Open Source** Command. While this command's primary purpose is to display the file within the Process Window, it also causes TotalView to read the file's debugging information. As a side-effect, **struct foo** should now be defined. Because TotalView now knows its definition, it can resolve the opaque type.

Type-Casting Examples

This section contains three type-casting examples:

- Displaying Declared Arrays
- Displaying Allocated Arrays
- Displaying the argv Array

Displaying Declared Arrays

TotalView displays arrays the same way it displays local and global variables. In the Stack Frame or Source Pane, dive on the declared array. A Variable Window displays the elements of the array.

CLI: dprint array-name

Displaying Allocated Arrays

The C Language uses pointers for dynamically allocated arrays; for example:

```
int *p = malloc(sizeof(int) * 20);
```

Since TotalView doesn't know that **p** actually points to an array of integers, you need to do several things to display the array:

1. Dive on the variable **p** of type **int***.
2. Change its type to **int[20]***.
3. Dive on the value of the pointer to display the array of 20 integers.

Displaying the argv Array

Typically, **argv** is the second argument passed to **main()**, and it is either a **char **argv** or **char *argv[ ]**. Suppose **argv** points to an array of three pointers to character strings. Here is how you can edit its type to display an array of three pointers:

1. Select the type string for **argv**.

   CLI: dprint argv

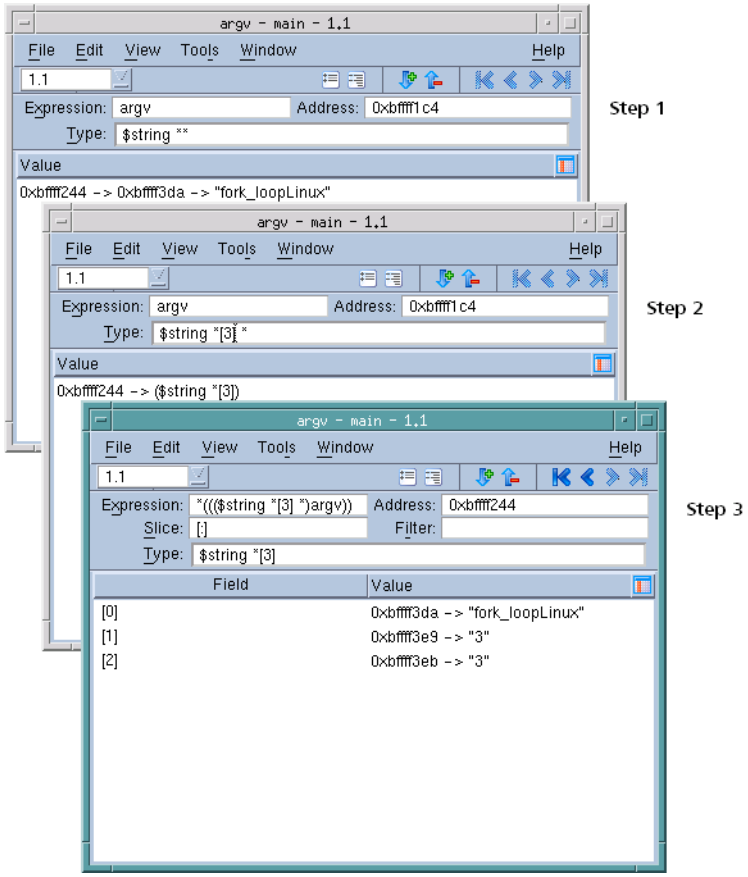
2. Edit the type string by using the field editor commands. Change it to:

   `$string*[3]*`

   CLI: dprint {($string*[3]*)argv}

3. To display the array, dive on the value field for **argv**.

Figure 150, Editing the argv Argument



Changing the Address of Variables

You can edit the address of a variable in a Variable Window by editing the value shown in the **Address** field. When you edit this address, the Variable Window shows the contents of the new location.

You can also enter an address expression such as **0x10b8 - 0x80** in this area.

Displaying C++ Types

RELATED TOPICS	
STL variable display	Displaying STL Variables on page 241
Changing the data type of a variable	Changing a Variable's Data Type on page 283
A variable's scope	Scoping and Symbol Names on page 309

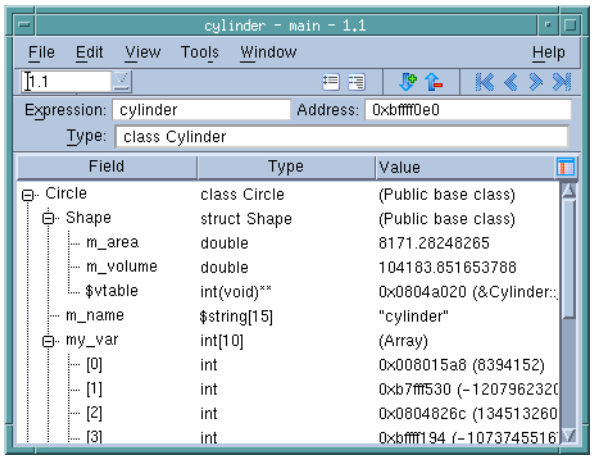
Viewing Classes

TotalView displays C++ classes and accepts **class** as a keyword. When you debug C++, TotalView also accepts the *unadorned* name of a **class**, **struct**, **union**, or **enum** in the type field. TotalView displays nested classes that use inheritance, showing derivation by indentation.

NOTE: Some C++ compilers do not write accessibility information. In these cases, TotalView cannot display this information.

For example, Figure 151 displays an object of a **class c**.

Figure 151, Displaying C++ Classes That Use Inheritance



Its definition is as follows:

```
class b {
```

```
   char * b_val;
public:
   b() {b_val = “b value“;}
};

class d : virtual public b {
   char * d_val;
public:
   d() {d_val = “d value“;}
};

class e {
   char * e_val;
 public:
   e() {e_val = “e value“;}
};

class c : public d, public e {
   char * c_val;
 public:
   c() {c_val = “c value“;}
};
```

TotalView tries to display the correct data when you change the type of a Variable Window while moving up or down the derivation hierarchy. Unfortunately, many compilers do not contain the information that TotalView needs so you might need to cast your class.

RELATED TOPICS

More on using C++ with TotalView Using C++ on page 364

C++View

C++View (CV) is a facility that allows you to format program data in a more useful or meaningful form than the concrete representation that you see in TotalView when you inspect data in a running program. To use C++View, you must write a function for each type whose format you would like to control. The signature of the function must be:

```
int TV_ttf_display_type ( const T *p )
```

where T is the type. Your function must use a TotalView-provided API to communicate the formatted representation of your data to TotalView.

When TotalView needs to display data, it checks to see if there is a function registered for the type to which the data belong. If there is, TotalView calls that function, and uses the results generated. Otherwise, if there is no matching function defined, TotalView presents the data in their raw form.

For complete details on using C++View, refer to the C++View chapter in the *Classic TotalView Reference Guide*.

C++View is available from the Preferences window. (See Setting Preferences on page 133.)

Displaying Fortran Types

TotalView lets you display FORTRAN 77 and Fortran 90 data types.

The topics in this section describe the various types and how the debugger handles them:

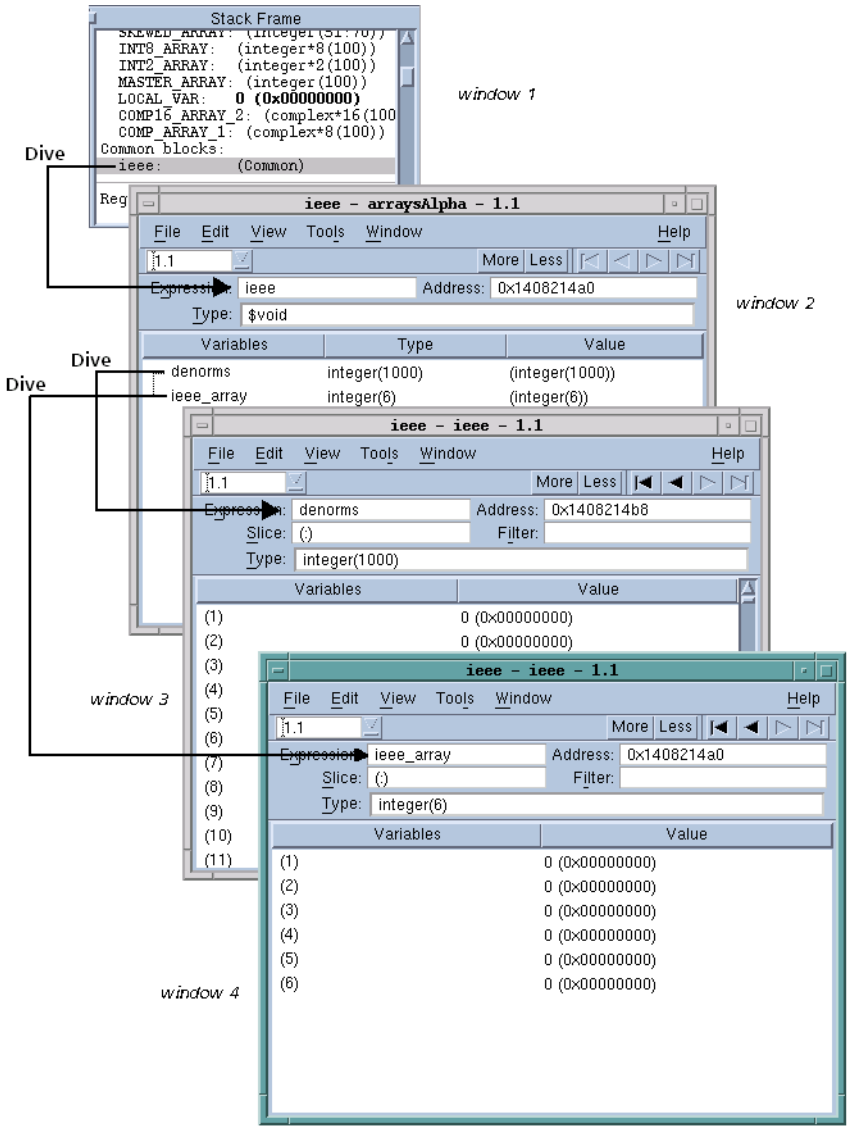
Displaying Fortran Common Blocks

For each common block defined in the scope of a subroutine or function, TotalView creates an entry in that function's common block list. The Stack Frame Pane displays the name of each common block for a function. The names of common block members have function scope, not global scope.

CLI: dprint *variable-name*

If you dive on a common block name in the Stack Frame Pane, the debugger displays the entire common block in a Variable Window, as shown in Figure 152.)

Figure 152, Diving on a Common Block List in the Stack Frame Pane



Window 1 in this figure shows a common block list in a Stack Frame Pane. After several dives, **Window 2** contains the results of diving on the common block.

If you dive on a common block member name, TotalView searches all common blocks in the function's scope for a matching member name, and displays the member in a Variable Window.

Displaying Fortran Module Data

TotalView tries to locate all data associated with a Fortran module and display it all at once. For functions and subroutines defined in a module, TotalView adds the full module data definition to the list of modules displayed in the Stack Frame Pane.

TotalView only displays a module if it contains data. Also, the amount of information that your compiler gives TotalView can restrict what is displayed.

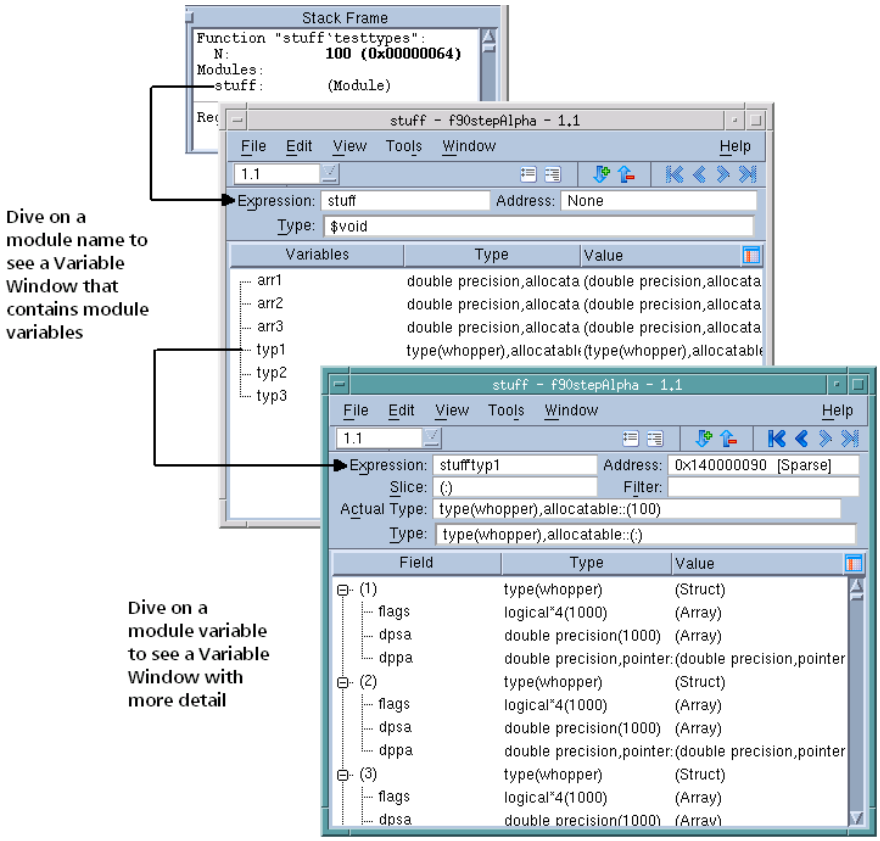
Although a function may use a module, TotalView doesn't always know if the module was used or what the true names of the variables in the module are. If this happens, either of the following occurs:

- Module variables appear as local variables of the subroutine.
- A module appears on the list of modules in the Stack Frame Pane that contains (with renaming) only the variables used by the subroutine.

CLI: dprint *variable-name*

Alternatively, you can view a list of all the known modules by using the **Tools > Fortran Modules** command. Because Fortran modules display in a Variable Window, you can dive on an entry to display the actual module data, as shown in Figure 153.

Figure 153, Fortran Modules Window



NOTE: If you are using the SUNPro compiler, TotalView can only display module data if you force TotalView 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 170.

Debugging Fortran 90 Modules

Fortran 90 lets you place functions, subroutines, and variables inside modules. You can then include these modules elsewhere by using a **USE** command. When you do this, the names in the module become available in the *using* compilation unit, unless you either exclude them with a **USE ONLY** statement or rename them. This means that you don't need to explicitly qualify the name of a module function or variable from the Fortran source code.

When debugging this kind of information, you need to know the location of the function being called. Consequently, TotalView uses the following syntax when it displays a function contained in a module:

modulename `functionname

You can also use this syntax in the **File > Debug New Program** and **View > Lookup Variable** commands.

Fortran 90 also lets you create a contained function that is only visible in the scope of its parent and siblings. There can be many contained 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*

Viewing Fortran 90 User-Defined Types

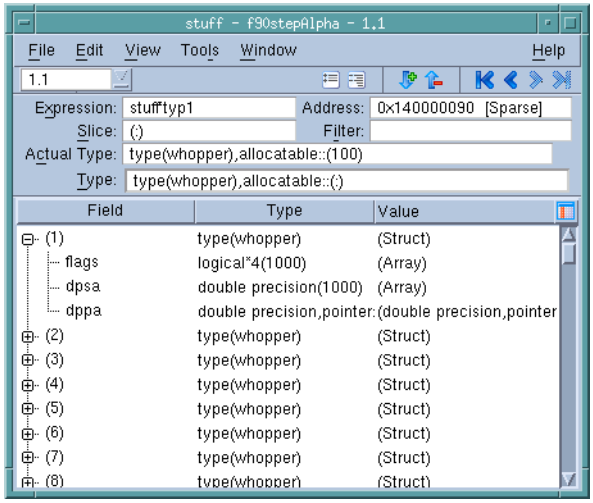
A Fortran 90 user-defined type is similar to a C structure. TotalView displays a user-defined type as **type(*name*)**, which is the same syntax used in Fortran 90 to create a user-defined type. For example, the following code fragment defines a variable **typ2** of **type(*whopper*)**:

```
TYPE WHOPPER
   LOGICAL, DIMENSION(ISIZE) :: FLAGS
   DOUBLE PRECISION, DIMENSION(ISIZE) :: DPSA
   DOUBLE PRECISION, DIMENSION(:), POINTER :: DPPA
END TYPE WHOPPER

TYPE(WHOPPER), DIMENSION(:), ALLOCATABLE :: TYP2
```

TotalView displays this type, as shown in Figure 154.

Figure 154, Fortran 90 User-Defined Type



Viewing Fortran 90 Deferred Shape Array Types

Fortran 90 lets you define deferred shape arrays and pointers. The actual bounds of a deferred shape array are not determined until the array is allocated, the pointer is assigned, or, in the case of an assumed shape argument to a subroutine, the subroutine is called. TotalView displays the type of deferred shape arrays as **type(:)**.

When TotalView displays the data for a deferred shape array, it displays the type used in the definition of the variable and the actual type that this instance of the variable has. The actual type is not editable, since you can achieve the same effect by editing the definition's type. The following example shows the type of a deferred shape rank 2 array of **real** data with runtime lower bounds of **-1** and **2**, and upper bounds of **5** and **10**:

```
        Type: real(:,:)
Actual Type: real(-1:5,2:10)
      Slice: (:,:)
```

Viewing Fortran 90 Pointer Types

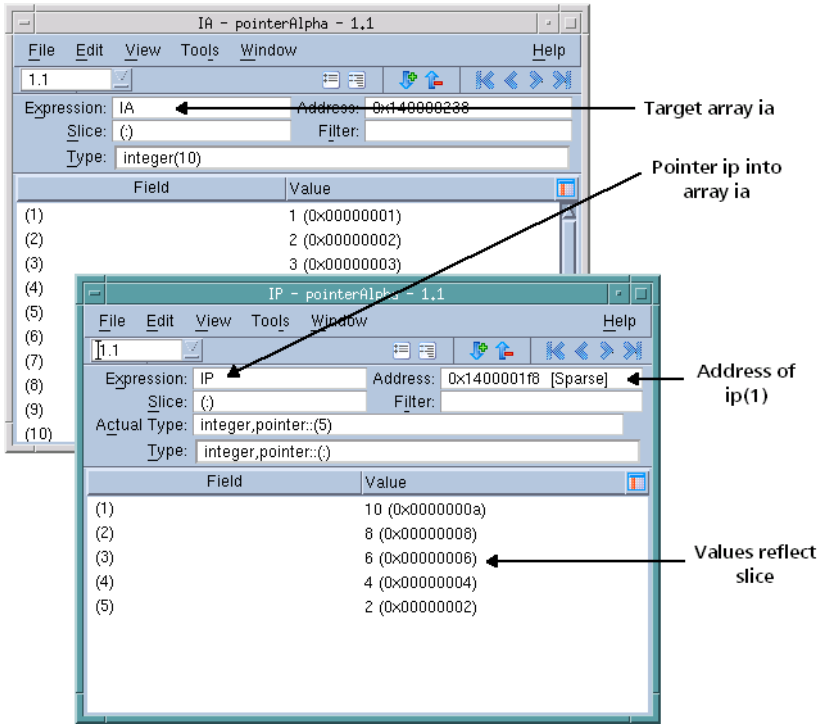
A Fortran 90 pointer type lets you point to scalar or array types.

TotalView implicitly handles slicing operations that set up a pointer or assumed shape subroutine argument so that indices and values it displays in a Variable Window are the same as in the Fortran code; for example:

```
integer, dimension(10), target :: ia
integer, dimension(:), pointer :: ip
do i = 1,10
   ia(i) = i
end do
ip => ia(10:1:-2)
```

After diving through the **ip** pointer, TotalView displays the windows shown in Figure 155:

Figure 155, Fortran 90 Pointer Value



The address displayed is not that of the array's base. Since the array's stride is negative, array elements that follow are at lower absolute addresses. Consequently, the address displayed is that of the array element that has the lowest index. This might not be the first displayed element if you used a slice to display the array with reversed indices.

Displaying Fortran Parameters

A Fortran **PARAMETER** defines a named constant. If your compiler generates debug information for parameters, they are displayed in the same way as any other variable. However, some compilers do not generate information that TotalView can use to determine the value of a **PARAMETER**. This means that you must make a few changes to your program if you want to see this type of information.

If you're using Fortran 90, you can define variables in a module that you initialize to the value of these **PARAMETER** constants; for example:

```
INCLUDE 'PARAMS.INC'
MODULE CONSTS
SAVE
INTEGER PI_C = PI
```

```
...
END MODULE CONSTS
```

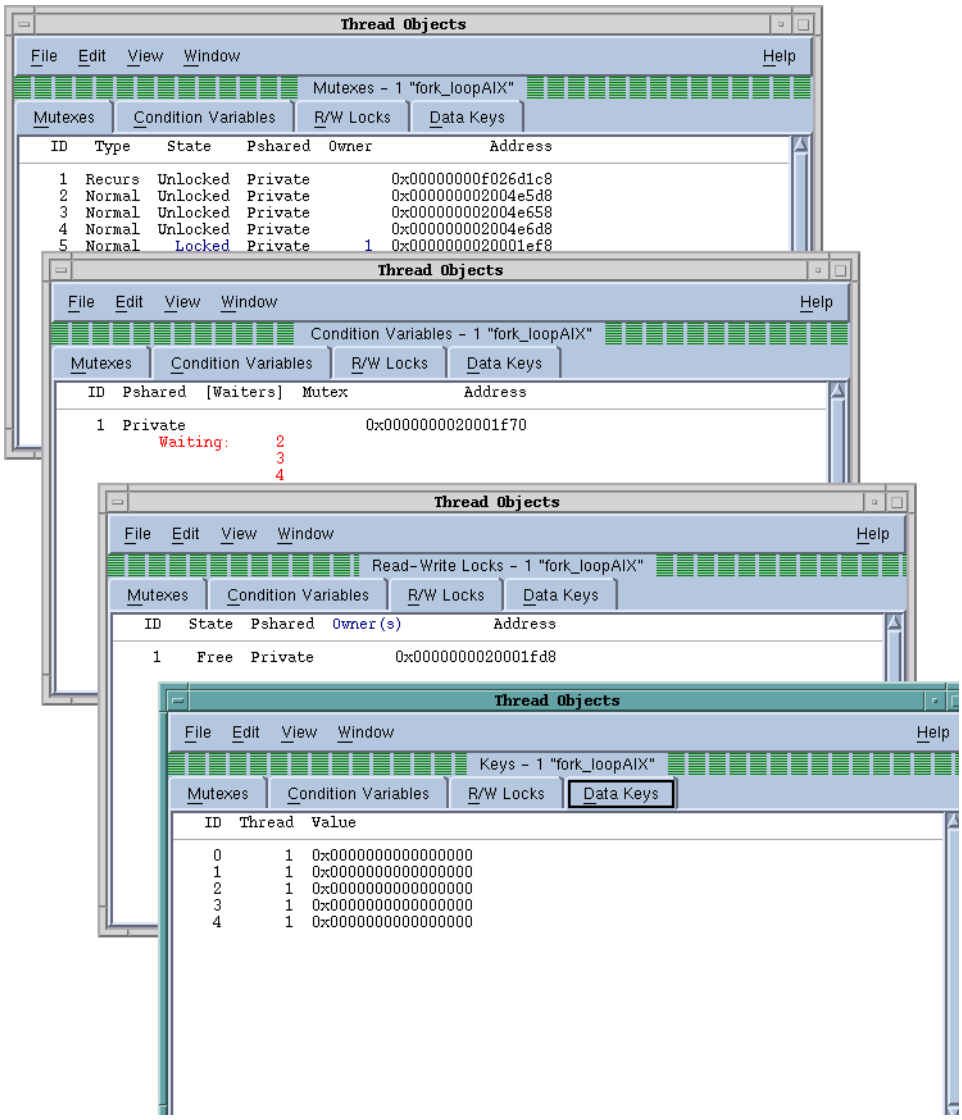
The **PARAMS.INC** file contains your parameter definitions. You then use these parameters to initialize variables in a module. After you compile and link this module into your program, the values of these parameter variables are visible.

If you're using FORTRAN 77, you can achieve the same results if you make the assignments in a common block and then include the block in **main()**. You can also use a block data subroutine to access this information.

Displaying Thread Objects

On IBM AIX systems, TotalView can display information about mutexes and conditional variables, read/write locks and data keys. You can obtain this information by selecting the **Tools > Thread Objects** command. After selecting this command, TotalView displays a window that contains four tabs. Figure 156 shows examples based on AIX.

Figure 156, Thread Objects Page on an IBM AIX Computer



Diving on any line in these windows displays a Variable Window that contains additional information about the item. Some notes:

- 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 in New Window** commands to display a Process Window for that thread ID.

The online Help contains information on the contents of the displayed windows.

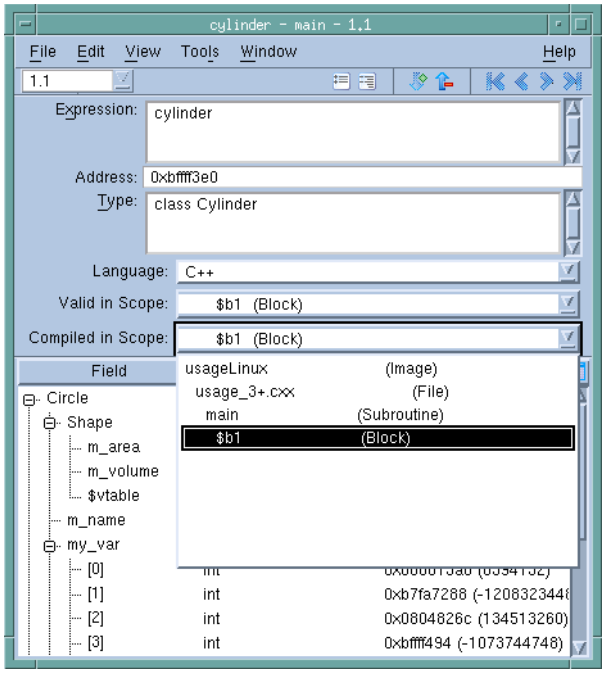
Scoping and Symbol Names

TotalView assigns a unique name to every element in your program based on the scope in which the element exists. A *scope* defines the part of a program that knows about a symbol. For example, the scope of a variable that is defined at the beginning of a subroutine is all the statements in the subroutine. The variable's scope does not extend outside of this subroutine. A program consists of multiple *scopes*. Of course, a block contained in the subroutine could have its own definition of the same variable. This would *hide* the definition in the enclosing scope.

All scopes are defined by your program's structure. Except for the simplest of programs, scopes are embedded in other scopes. The only exception is the outermost scope, which is the one that contains **main()**, which is not embedded. Every element in a program is associated with a scope.

To see the scope in which a variable is valid, click the **More** button in the Variable Window until the scope fields are visible. The Variable Window now includes additional information about your variable, as is shown in Figure 157.

Figure 157, Variable Window: Showing Variable Properties



The **Valid in Scope** list indicates the scope in which the variable resides. That is, when this scope is active, the variable is defined. The **Compiled in Scope** list can differ if you modify the variable with an expression. It indicates where variables in this expression have meaning.

When you tell the CLI or the GUI to execute a command, TotalView consults the program's symbol table to discover which object you are referring to—this process is known as *symbol lookup*. Symbol lookup is performed with respect to a particular context, and each context uniquely identifies the scope to which a symbol name refers.

RELATED TOPICS	
Issues with scoping	Scoping Issues on page 253
Variables in a current block	Displaying Variables in the Current Block on page 251

Qualifying Symbol Names

The way you describe a scope is similar to the way you specify a file. The scopes in a program form a tree, with the outermost scope (which is your program) as the root. At the next level are executable files and dynamic libraries; further down are compilation units (source files), procedures, modules, and other scoping units (for example, blocks) supported by the programming language. Qualifying a symbol is equivalent to describing the path to a file in UNIX file systems.

A symbol is fully scoped when you name all levels of its tree. The following example shows how to scope a symbol and also indicates parts that are optional:

```
[##executable-or-lib#][file#][procedure-or-line#]symbol
```

The pound sign (**#**) separates elements of the fully qualified name.

NOTE: Because of the number of different types of elements that can appear in your program, a complete description of what can appear and their possible order is complicated and unreadable. In contrast, after you see a name in the Stack Frame Pane, it is easy to read a variable's scoped name.

TotalView interprets most programs and components as follows:

- If a qualified symbol begins with ##, the name that follows indicates the name of the executable or shared library (just as an absolute file path begins with a directory immediately in the root directory). If you omit the executable or library component, the qualified symbol doesn't begin with #.
- The source file's name can appear after the possibly omitted executable or shared library.

- Because programming languages typically do not let you name blocks, that portion of the qualifier is specified using the symbols **$b** followed by a number that indicates which block. For example, the first unnamed block is named **$b1**, the second is **$b2**, and so on.

RELATED TOPICS	
Issues with scoping	Scoping Issues on page 253
The **dbreak** command	**dbreak** command description
Breakpoints at locations	Setting Breakpoints at Locations on page 201
Lookup Function	The **View > Lookup Function** topic in the in-product help
Lookup Variable	The **View > Lookup Variable** topic in the in-product help

Examining Arrays

This chapter explains how to examine and change array data as you debug your program. Since arrays also appear in the Variable Window, you need to be familiar with the information in Examining and Editing Data and Program Elements on page 240.

The topics in this chapter are:

Examining and Analyzing Arrays

TotalView can quickly display very large arrays in Variable Windows. An array can be the elements that you define in your program, or it can be an area of memory that you cast into an array.

If an array extends beyond the memory section in which it resides, the initial portion of the array is formatted correctly. If memory isn't allocated for an array element, TotalView displays **Bad Address** in the element's subscript.

Topics in this section are:

Displaying Array Slices

TotalView lets you display array subsections by editing the **Slice** field in an array's Variable Window. (An array subsection is called a *slice*.) The **Slice** field contains placeholders for all array dimensions. For example, the following is a C declaration for a three-dimensional array:

```
integer an_array[10][20][5]
```

Because this is a three-dimensional array, the initial slice definition is **[:][:][:]**. This lets you know that the array has three dimensions and that TotalView is displaying all array elements.

The following is a deferred shape array definition for a two-dimensional array variable:

```
integer, dimension (:,:) :: another_array
```

The TotalView slice definition is **(:,:)**.

TotalView displays as many colons (**:**) as there are array dimensions. For example, the slice definition for a one-dimensional array (a vector) is **[:]** for C arrays and **(:)** for Fortran arrays.

CLI: **dprint -slice "\[n:m\]" an_array**
dprint -slice "(n:m,p:q)" an_array

Using Slices and Strides

A slice has the following form:

lower_bound:upper_bound[:stride]

The *stride*, which is optional, tells TotalView to skip over elements and not display them. Adding a *stride* to a slice tells the debugger to display every *stride* element of the array, starting at the *lower_bound* and continuing through the *upper_bound*, inclusive.

For example, a slice of **[0:9:9]** used on a ten-element C array tells TotalView to display the first element and last element, which is the ninth element beyond the lower bound.

If the stride is negative and the lower bound is greater than the upper bound, TotalView displays a dimension with its indices reversed. That is, TotalView treats the slice as if it was defined as follows:

[*upperbound* **:** *lowerbound* **:** *stride***]**

CLI: dprint *an_array(n:m:p,q:r:s)*

For example, the following definition tells TotalView to display an array beginning at its last value and moving to its first:

```
[::-1]
```

This syntax differs from Fortran 90 syntax in that Fortran 90 requires that you explicitly enter the upper and lower bounds when you're reversing the order for displaying array elements.

Because the default value for the stride is **1**, you can omit the stride (and the colon that precedes it) from your definition. For example, the following two definitions display array elements 0 through 9:

```
[0:9:1]
[0:9]
```

If the lower and upper bounds are the same, just use a single number. For example, the following two definitions tell TotalView to display array element 9:

```
[9:9:1]
[9]
```

NOTE: The lower_bound, upper_bound, and stride must be constants. They cannot be expressions.

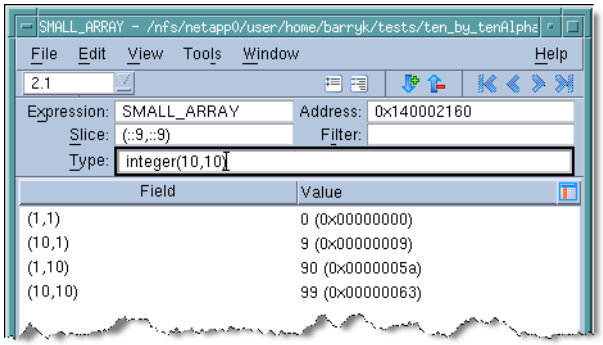
Example 1

A slice declaration of **[::2]** for a C or C++ array (with a default lower bound of **0**) tells TotalView to display elements with even indices of the array; that is, 0, 2, 4, and so on. However, if this were defined for a Fortran array (where the default lower bound is **1**), TotalView displays elements with odd indices of the array; that is, 1, 3, 5, and so on.

Example 2

Figure 158 displays a stride of **(::9,::9).** This definition displays the four corners of a ten-element by ten-element Fortran array.

Figure 158, Stride Displaying the Four Corners of an Array



Example 3

You can use a stride to invert the order *and* skip elements. For example, the following slice begins with the upper bound of the array and displays every other element until it reaches the lower bound of the array:

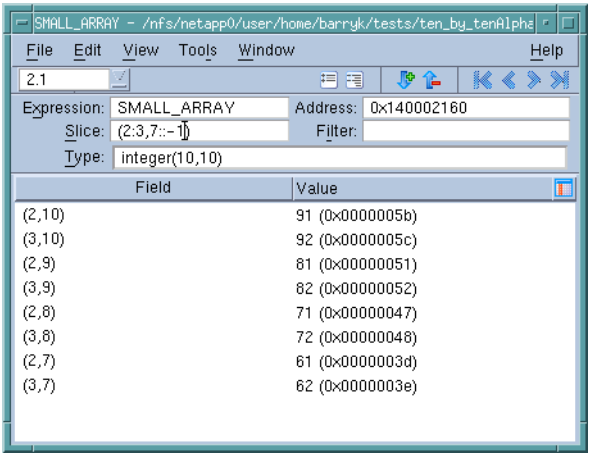
```
(::-2)
```

Using **(::-2)** with a Fortran **integer(10)** array tells TotalView to display the elements 10, 8, 6, 4, and 2.

Example 4

You can simultaneously invert the array's order and limit its extent to display a small section of a large array. The following figure shows how to specify a **(2:3,7::-1)** slice with an **integer*4(-1:5,2:10)** Fortran array.

Figure 159, 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 dialog box, TotalView initializes the **Slice** field in the Variable Window to this slice description.

If you add subscripts to an array name in the **View > Lookup Variable** dialog box, TotalView will look up just that array element.

```
CLI: dprint small_array(5,5)
```

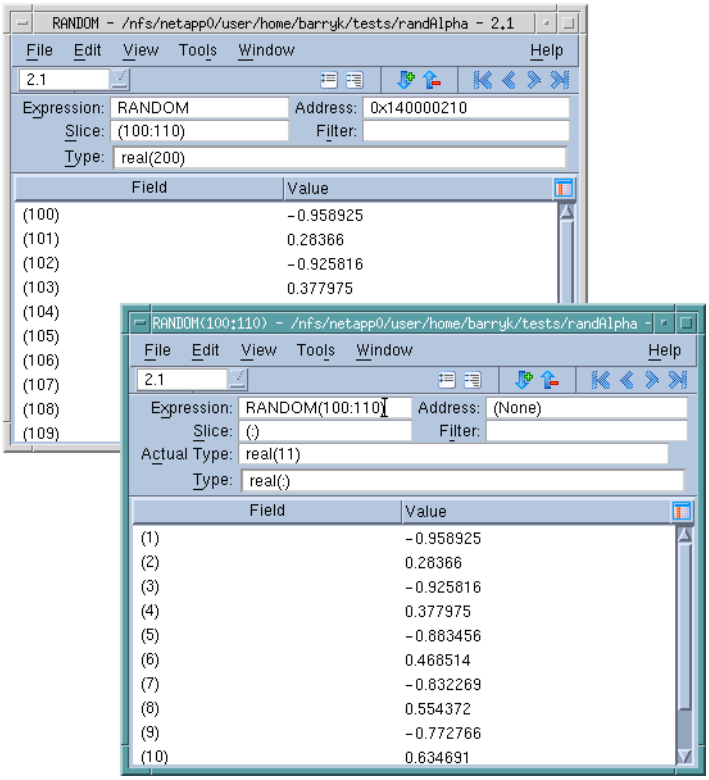
You can, of course, type an expression into the **View > Lookup Variable** dialog box; for example, you could type **small_array(i-1,j-1)**.

Array Slices and Array Sections

An array slice allows you to see a part of an array. The slice allows you to remove parts of the array you do not want to see. For example, if you have a 10,000 element array, you could tell TotalView that it should only display 100 of these elements. Fortran has introduced the concept of an array section. When you create an array section, you are creating a new array that is a subset of the old array. Because it is a new array, its first array index is 1.

In Figure 160, the top left Variable Window displays an eleven-element array slice. The bottom right Variable Window displays an eleven-element array.

Figure 160, An Array Slice and an Array Section

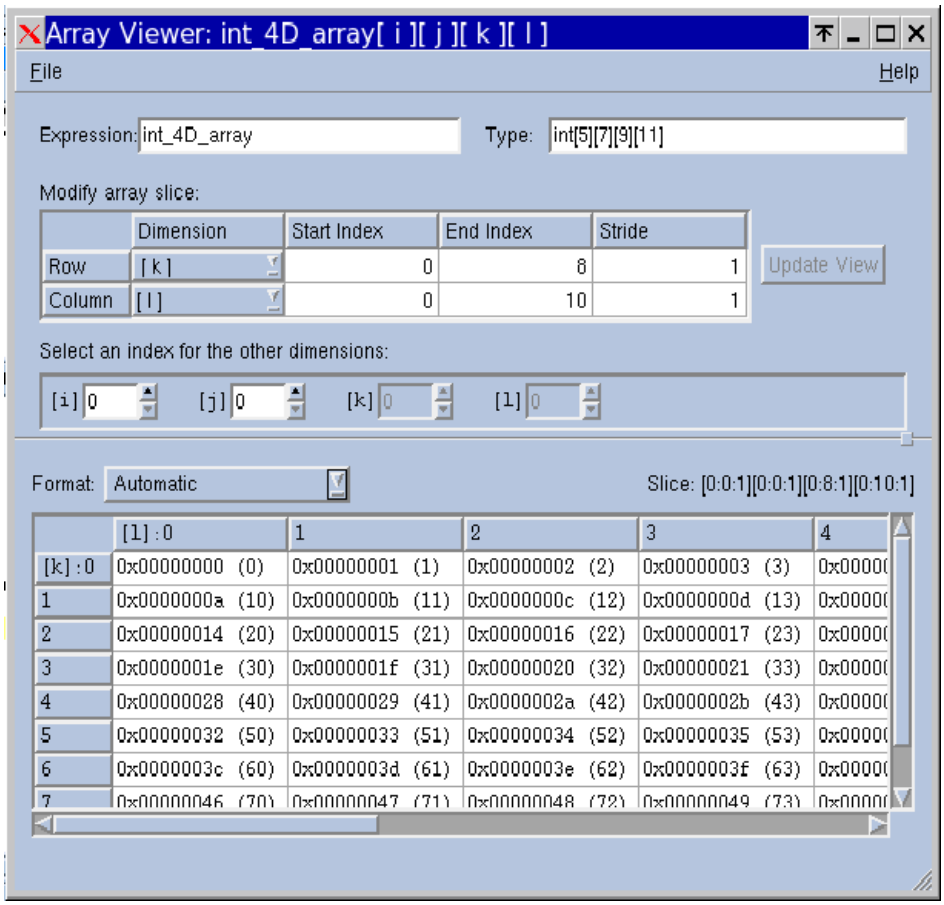


While the data in both is identical, notice that the array numbering is different. In addition, the array slice shows an address for the array. The section, however, only exists within TotalView. Consequently, there is no address associated with it.

Viewing Array Data

TotalView provides another way to look at the data in a multi-dimensional array. The Variable Window's **Tools > Array Viewer** command opens a window that presents a slice of array data in a table format, Figure 161. You can think of this as viewing a "plane" of two-dimensional data in your array.

Figure 161, Array Viewer



When the Array Viewer opens, the initial slice of displayed data depends on the values you entered in the Variable Window. You can change the displayed data by modifying the Expression, Type, or slice controls in the Array Viewer and then pressing the Update View button.

Expression Field

The Expression field contains an array expression based on the value you entered in the Variable Window. You can control the display by changing the value of this field; for example, you can cast the array to another array expression.

Type Field

The Type field also reflects the data you initially entered in the Variable Window. You can modify the type to cast the array to a different array type.

Slice Definition

Initially, TotalView selects the array slice by placing the appropriate array dimension as the row and the column, setting the indices for the lower and upper bounds of the dimensions with a stride of one. Any additional dimensions are held at 0. This is the slice or plane of data that is displayed in the table.

You have full control over all settings, including the ability to change which dimensions appear as rows and columns, as well as their indices and strides. As you change the row and column dimensions, the controls for the other dimensions are enabled/disabled accordingly. You can change the indices of the other dimensions to further refine the slice of data. The section Using Slices and Strides on page 314 provides more information on slicing arrays.

Update View Button

When you have finished making changes to the expression, type, and/or slice settings, press the **Update View** button to update the data in the table display.

Data Format Selection Box

The selection box at the top left corner of the data table allows you to select the format for displaying the data. The table automatically refreshes in the selected format.

The Slice field at the top right corner of the data table reflects the displayed slice of data.

Filtering Array Data Overview

You can restrict what TotalView displays in a Variable Window by adding a filter to the window. You can filter arrays of type character, integer, or floating point. Your filtering options are:

- Arithmetic comparison to a constant value
- Equal or not equal comparison to IEEE NaNs, Infs, and Denorms
- Within a range of values, inclusive or exclusive
- General expressions

When an element of an array matches the filter expression, TotalView includes the element in the Variable Window display.

The following topics describe filtering options:

- Filtering Array Data on page 320
- Filtering by Comparison on page 320

Filtering Array Data

The procedure for filtering an array is simple: select the **Filter** field, enter the array filter expression, and then press Enter.

TotalView updates the Variable Window to exclude elements that do not match the filter expression. TotalView only displays an element if its value matches the filter expression and the slice operation.

If necessary, TotalView converts the array element before evaluating the filter expression. The following conversion rules apply:

- If the filter operand or array element type is floating point, TotalView converts the operand to a double-precision floating-point value. TotalView truncates extended-precision values to double precision. Converting integer or unsigned integer values to double-precision values might result in a loss of precision. TotalView converts unsigned integer values to nonnegative double-precision values.
- If the filter operand or the array element is an unsigned integer, TotalView converts the operand to an unsigned 64-bit integer.
- If both the filter operand and array element are of type integer, TotalView converts the values to a 64-bit integer.

TotalView conversion operations modify a copy of the array's elements—conversions never alter the actual array elements.

To stop filtering an array, delete the contents of the **Filter** field in the Variable Window and press Enter. TotalView then updates the Variable Window so that it includes all elements.

Filtering by Comparison

The simplest filters are ones whose formats are as follows:

operator value

where ***operator*** is either a C/C++ or Fortran-style comparison operator, and ***value*** is a signed or unsigned integer constant or a floating-point number. For example, the filter for displaying all values greater than 100 is:

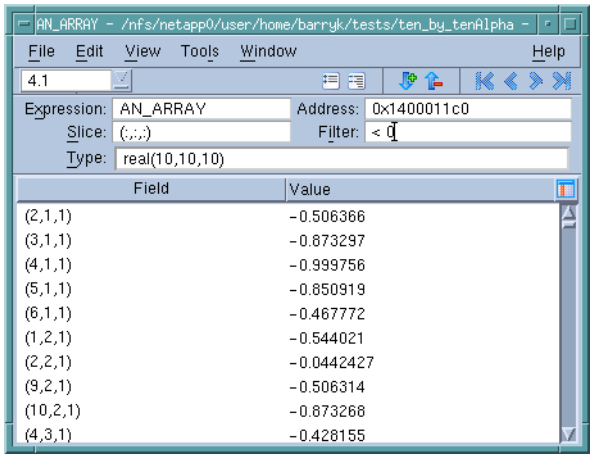
```
> 100
```

The following table lists the comparison operators:

Comparison	C/C++ Operator	Fortran Operator
Equal	==	**.eq.**
Not equal	!=	**.ne.**
Less than	<	**.lt.**
Less than or equal	<=	**.le.**
Greater than	>	**.gt.**
Greater than or equal	>=	**.ge.**

Figure 162 shows an array whose filter is **< 0**. This tells TotalView to display only array elements whose value is less than 0 (zero).

Figure 162, Array Data Filtering by Comparison



If the *value* you are using in the comparison is an integer constant, TotalView performs a signed comparison. If you add the letter **u** or **U** to the constant, TotalView performs an unsigned comparison.

Filtering for IEEE Values

You can filter IEEE NaN, Infinity, or denormalized floating-point values by specifying a filter in the following form:

operator ieee-tag

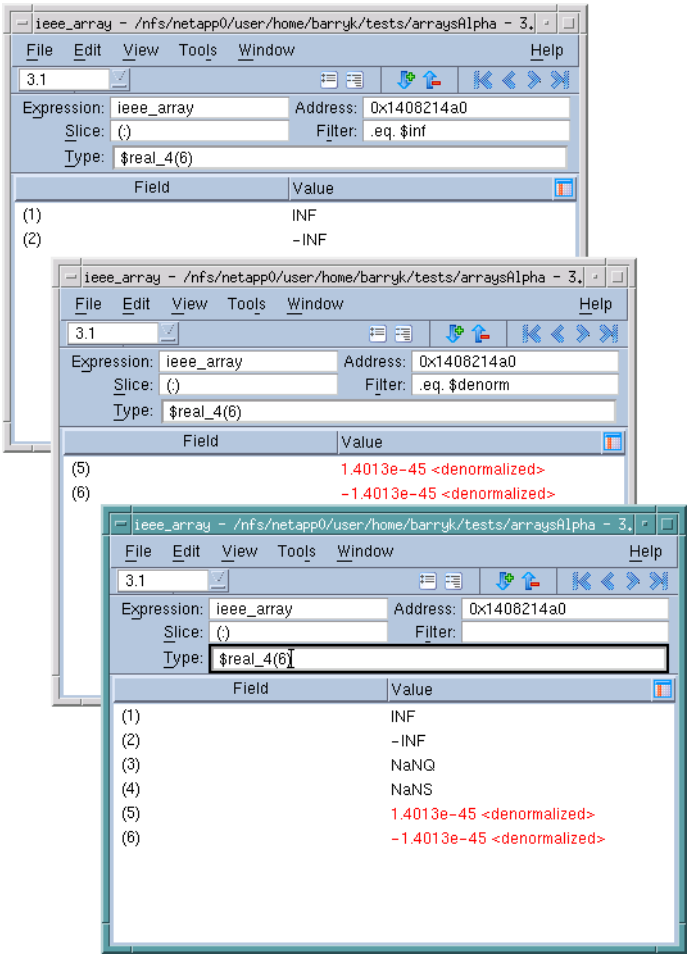
The only comparison operators you can use are ***equal*** and ***not equal***.

The ***ieee-tag*** represents an encoding of IEEE floating-point values, as the following table describes:

IEEE Tag Value	Meaning
$nan	NaN (Not a number), either quiet or signaling
$nanq	Quiet NaN
$nans	Signaling NaN
$inf	Infinity, either positive or negative
$pinf	Positive Infinity
$ninf	Negative Infinity
$denorm	Denormalized number, either positive or negative
$pdenorm	Positive denormalized number
$ndenorm	Negative denormalized number

Figure 163 shows an example of filtering an array for IEEE values. The bottom window in this figure shows how TotalView displays the unfiltered array. Notice the NaNQ, and NaNS, INF, and -INF values. The other two windows show filtered displays: the top window shows only infinite values; the second window only shows the values of denormalized numbers.

Figure 163, Array Data Filtering for IEEE Values



If you are writing an expression, you can use the following Boolean functions to check for a particular type of value:

IEEE Intrinsic	Meaning
$is_denorm(*value*)	Is a denormalized number, either positive or negative
$is_finite(*value*)	Is finite
$is_inf(*value*)	Is Infinity, either positive or negative
$is_nan(*value*)	Is a NaN (Not a number), either quiet or signaling
$is_ndenorm(*value*)	Is a negative denormalized number
$is_ninf(*value*)	Is negative Infinity

IEEE Intrinsic	Meaning
$is_nnorm(*value*)	Is a negative normalized number
$is_norm(*value*)	Is a normalized number, either positive or negative
$is_nzero(*value*)	Is negative zero
$is_pdenorm(*value*)	Is a positive denormalized number
$is_pinf(*value*)	Is positive Infinity
$is_pnorm(*value*)	Is a positive normalized number
$is_pzero(*value*)	Is positive zero
$is_qnan(*value*)	Is a quiet NaN
$is_snan(*value*)	Is a signaling NaN
$is_zero(*value*)	Is zero, either positive or negative

Filtering a Range of Values

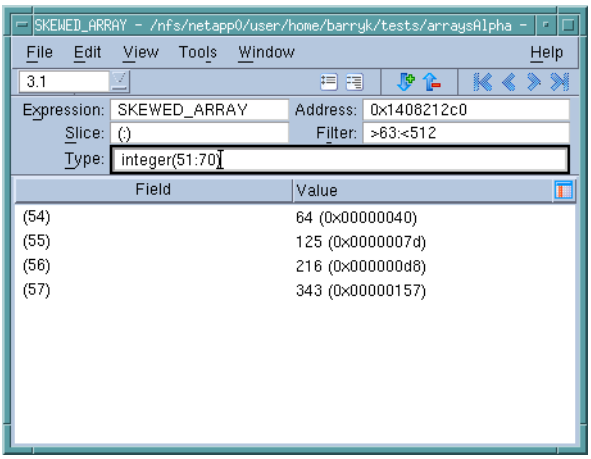
You can also filter array values by specifying a range, as follows:

[**>**] ***low-value*** **:** [**<**] ***high-value***

where ***low-value*** specifies the lowest value to include, and ***high-value*** specifies the highest value to include, separated by a colon. The high and low values are inclusive unless you use less-than (**<)** and greater-than (**>**) symbols. If you specify a **>** before ***low-value***, the low value is exclusive. Similarly, a **<** before ***high-value*** makes it exclusive.

The values of ***low-value*** and ***high-value*** must be constants of type integer, unsigned integer, or floating point. The data type of ***low-value*** must be the same as the type of ***high-value***, and ***low-value*** must be less than ***high-value***. If ***low-value*** and ***high-value*** are integer constants, you can append the letter **u** or **U** to the value to force an unsigned comparison. The following figure shows a filter that tells TotalView to only display values greater than 63, but less than 512. (See Figure 164.)

Figure 164, Array Data Filtering by Range of Values



Creating Array Filter Expressions

The filtering capabilities described in the previous sections are those that you use most often. In some circumstances, you may need to create a more general expression. When you create a filter expression, you're creating a Fortran or C Boolean expression that TotalView evaluates for every element in the array or the array slice. For example, the following expression displays all array elements whose contents are greater than 0 and less than 50, or greater than 100 and less than 150:

```
($value > 0 && $value < 50) ||
          ($value > 100 && $value < 150)
```

Here's the Fortran equivalent:

```
($value .gt. 0 && $value .lt. 50) .or.
          ($value .gt. 100 .and. $value .lt.150)
```

The **$value** variable is a special TotalView variable that represents the current array element. You can use this value when creating expressions.

Notice how the **and** and **or** operators are used in these expressions. The way in which TotalView computes the results of an expression is identical to the way it computes values at an eval point. For more information, see Defining Eval Points and Conditional Breakpoints on page 220.

Using Filter Comparisons

TotalView provides several different ways to filter array information. For example, the following two filters display the same array items:

```
> 100
$value > 100
```

The following filters display the same array items:

```
>0:<100
$value > 0 && $value < 100
```

The only difference is that the first method is easier to type than the second, so you're more likely to use the second method when you're creating more complicated expressions.

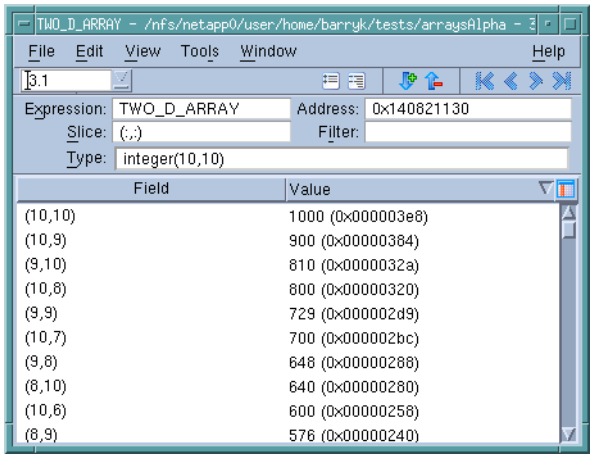
Sorting Array Data

TotalView lets you sort the displayed array data into ascending or descending order. (It does not sort the actual data.) To sort (or remove the sort), click the **Value** label.

- The first time you click, TotalView sorts the array's values into ascending order.
- The next time you click on the header, TotalView reverses the order, sorting the array's values into descending order.
- If you click again on the header, TotalView returns the array to its unsorted order.

Here is an example that sorts an array into descending order:

Figure 165, Sorted Variable Window



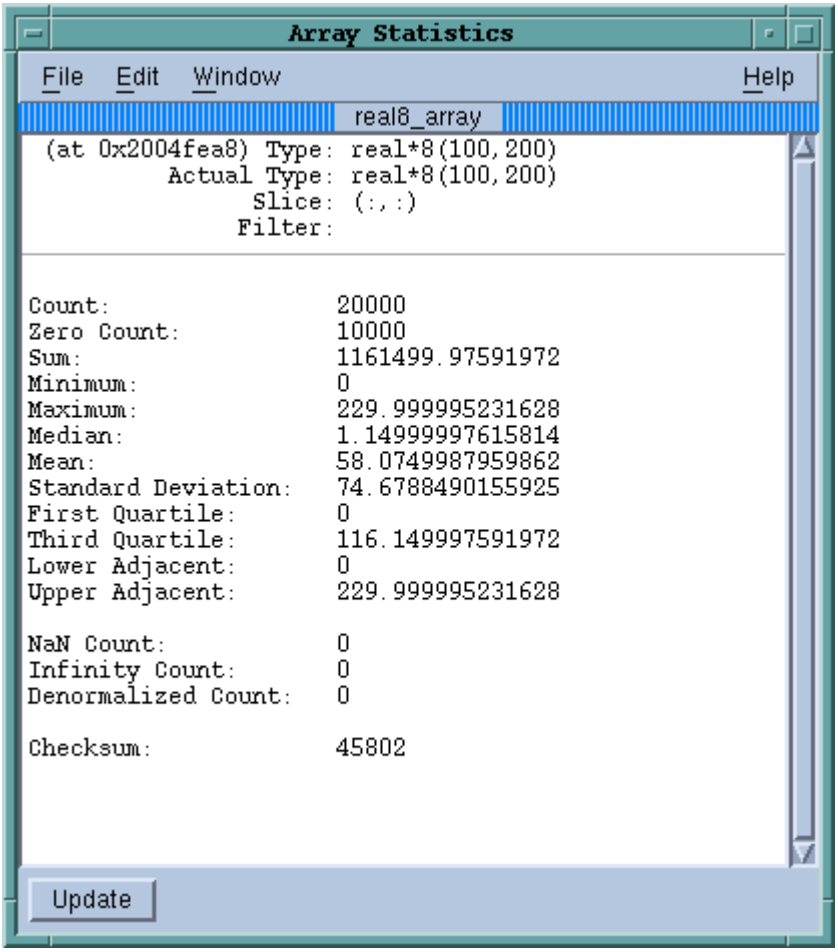
When you sort an array's values, you are just rearranging the information that's displayed in the Variable Window. Sorting does not change the order in which values are stored in memory. If you alter what TotalView is displaying by using a filter or a slice, TotalView just sorts the values that could be displayed; it doesn't sort all of the array.

If you are displaying the array created by a **Show across** command—see Displaying a Variable in all Processes or Threads on page 330 for more information—you can sort your information by process or thread.

Obtaining Array Statistics

The **Tools > Statistics** command displays a window that contains information about your array. Figure 166 shows an example.

Figure 166, Array Statistics Window



If you have added a filter or a slice, these statistics describe only the information currently being displayed; they do not describe the entire unfiltered array. For example, if 90% of an array's values are less than 0 and you filter the array to show only values greater than 0, the median value is positive even though the array's real median value is less than 0.

NOTE: Array statistics are available through the CLI, as switches to the dprint command. See the dprint description in the Reference Guide for details.

TotalView displays the following statistics:

- **Checksum**

  A checksum value for the array elements.

- **Count**

  The total number of displayed array values. If you're displaying a floating-point array, this number doesn't include NaN or Infinity values.

- **Denormalized Count**

  A count of the number of denormalized values found in a floating-point array. This includes both negative and positive denormalized values as defined in the IEEE floating-point standard. Unlike other floating-point statistics, these elements participate in the statistical calculations.

- **Infinity Count**

  A count of the number of infinity values found in a floating-point array. This includes both negative and positive infinity as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

- **Lower Adjacent**

  This value provides an estimate of the lower limit of the distribution. Values below this limit are called *outliers*. The lower adjacent value is the first quartile value minus the value of 1.5 times the difference between the first and third quartiles.

- **Maximum**

  The largest array value.

- **Mean**

  The average value of array elements.

- **Median**

  The middle value. Half of the array's values are less than the median, and half are greater than the median.

- **Minimum**

  The smallest array value.

- **NaN Count**

  A count of the number of NaN (not a number) values found in a floating-point array. This includes both signaling and quiet NaNs as defined in the IEEE floating-point standard. These elements do not participate in statistical calculations.

- **Quartiles, First and Third**

Either the 25th or 75th percentile values. The first quartile value means that 25% of the array's values are less than this value and 75% are greater than this value. In contrast, the third quartile value means that 75% of the array's values are less than this value and 25% are greater.

- **Standard Deviation**

  The standard deviation for the array's values.

- **Sum**

  The sum of all the displayed array's values.

- **Upper Adjacent**

  This value provides an estimate of the upper limit of the distribution. Values above this limit are called *outliers*. The upper adjacent value is the third quartile value plus the value of 1.5 times the difference between the first and third quartiles.

- **Zero Count**

  The number of elements whose value is 0.

Displaying a Variable in all Processes or Threads

When you're debugging a parallel program 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.

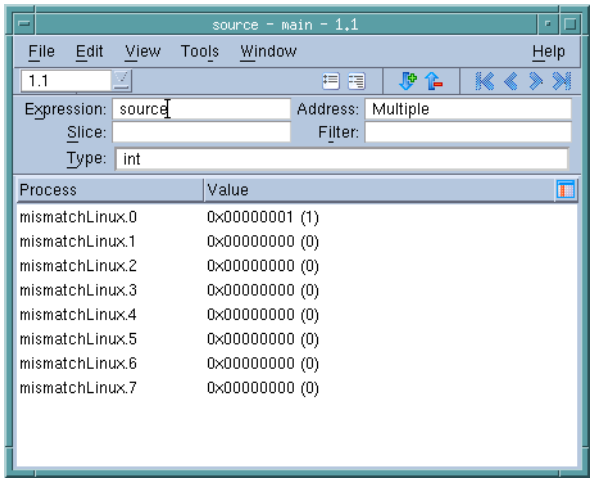
Before displaying a variable's value in all threads or processes, you must display an instance of the variable in a Variable Window. In this window, use one of the following commands:

- **View > Show Across > Process**, displays the value of the variable in all processes.
- **View > Show Across > Thread**, displays the value of a variable in all threads within a single process.
- **View > Show Across > None**, returns the window to what it was before you used other Show Across commands.

NOTE: You cannot simultaneously Show Across processes and threads in the same Variable Window.

After selecting a command, the Variable Window provides an array-like display of the value of the variable in each process or thread. Figure 167 shows a simple, scalar variable in each of the processes in an OpenMP program.

Figure 167, Viewing Across Threads



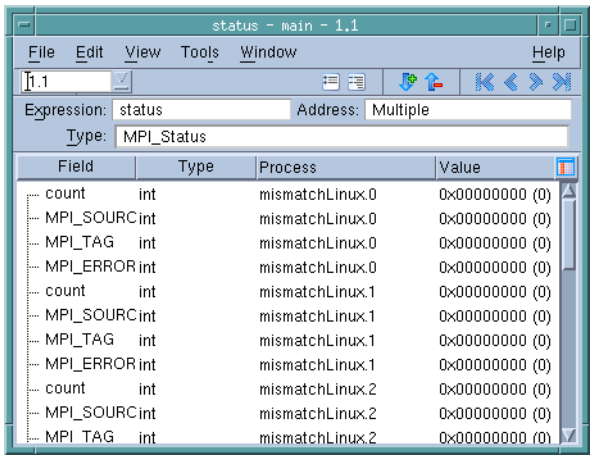
When looking for a matching stack frame, TotalView matches frames starting from the top frame, and considers calls from different memory or stack locations to be different calls. For example, the following definition of **recurse()** contains two additional calls to **recurse()**. Each of these calls generates a nonmatching call frame.

```
void recurse(int i) {
    if (i <= 0)
       return;
    if (i & 1)
       recurse(i - 1);
    else
       recurse(i - 1);
}
```

If the variables are at different addresses in the different processes or threads, the field to the left of the **Address** field displays **Multiple,** and the unique addresses appear with each data item.

TotalView also lets you Show Across arrays and structures. When you Show Across an array, TotalView displays each element in the array across all processes. You can use a slice to select elements to be displayed in an "across" display. The following figure shows the result of applying a **Show Across > Processes** command to an array of structures.

Figure 168, Viewing across an Array of Structures



RELATED TOPICS

Viewing a structure's elements as an array Displaying an Array of Structure's Elements on page 268

Diving on a "Show Across" Pointer

You can dive through pointers in a Show Across display. This dive applies to the associated pointer in each process or thread.

Editing a “Show Across” Variable

If you edit a value in a “Show Across” display, TotalView asks if it should apply this change to all processes or threads or only the one in which you made a change. This is an easy way to update a variable in all processes.

Visualizing Array Data

The Visualizer lets you create graphical images of array data. This presentation lets you see your data in one glance and can help you quickly find problems with your data while you are debugging your programs.

You can execute the Visualizer from within TotalView, or you can run it from the command line to visualize data dumped to a file in a previous TotalView session.

For information about running the Visualizer, see Visualizing Programs and Data on page 334.

Visualizing a "Show Across" Variable Window

You can export data created by using a *Show Across* command to the Visualizer by using the **Tools > Visualize** command. When visualizing this kind of data, the process (or thread) index is the first axis of the visualization. This means that you must use one less data dimension than you normally would. If you do not want the process/thread axis to be significant, you can use a normal Variable Window, since all of the data must be in one process.

Visualizing Programs and Data

TotalView provides a set of tools to visualize your program activity, including its arrays, and MPI message data. This chapter describes:

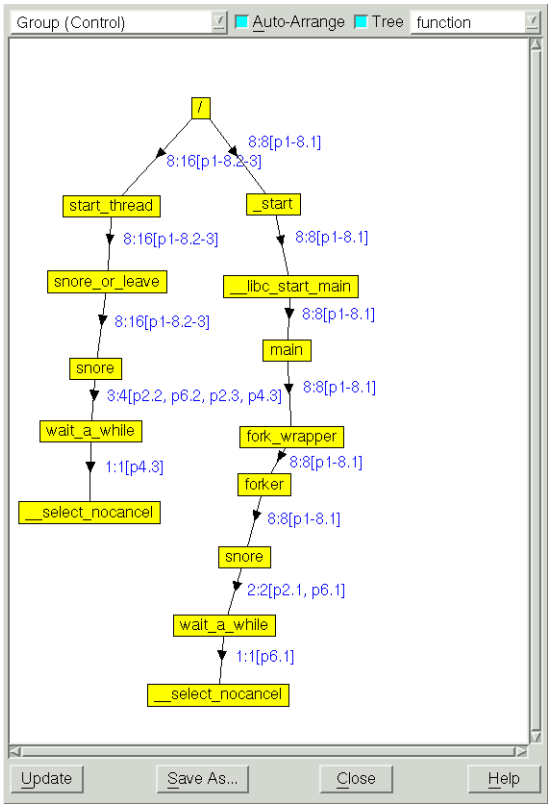
Displaying Call Trees and Call Graphs

Debugging is an art, not a science. Debugging often requires the intuition to guess what a program is doing and where to look for problems. Just locating a problem can be 90% or more of the effort. A call tree or call graph can help you understand what your program is doing so that you can understand how your program is executing.

To display a call tree or call graph, select **Tools > Call Graph** from the Process Window. A sample call tree is shown in Figure 169.

The call tree or call graph shows all currently active routines linked by arrows indicating if one routine is called by another. The display is *dynamic* in that it shows activity at the moment it is created. The **Update** button recreates the display.

Figure 169, Tools > Call Graph Dialog Box



You can toggle between displaying a call tree or call graph for specific processes and threads using the controls at the top of this window. By default, TotalView displays a tree representing the backtrace of all the selected processes and threads. To change to a Graph Style display, deselect the **Tree** button.

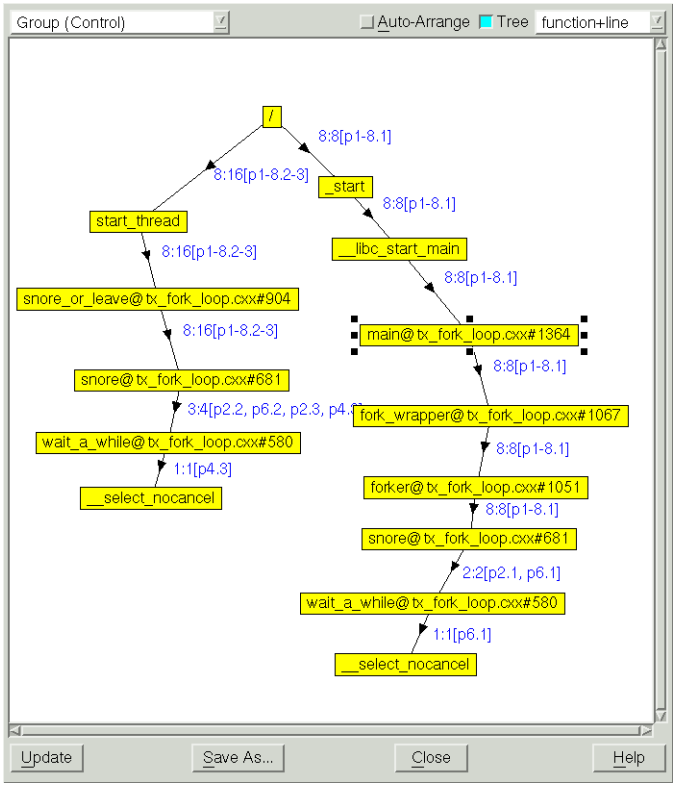
For multi-process or multi-threaded programs, a compressed process/thread list **(ptlist)** next to the arrows indicates which threads have a routine on their call stack.

Similar to the CLI's **dwhere -group_by** option, the dropdown in the call tree window enables you to aggregate the backtraces according to different properties, as follows:

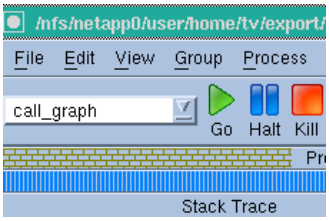
- **function**: Equivalence based on the name of the function containing the PC for the frame. This is the default.
- **function+line**: Equivalence based on the name of the function and the file and line number containing the PC for the frame.
- **function+offset**: Equivalence based on the name of the function containing the PC for the frame and offset from the beginning of the function to the PC for the frame.

For example, Figure 170 displays the call tree grouped by function and line:

Figure 170, Tools > Call Graph grouped by function and line



Diving on a routine within the call tree or call graph creates a group called **call_graph**, containing all the threads that have the routine you dived on in its call stack. If you look at the Process Window's Processes tab, you'll see that the **call_graph** set is selected in the scope pulldown.



In addition, the context of the Process Window changes to the first thread in the set.

As you begin to understand your program, you will see that this diagram reflects your program's rhythm and dynamic. As you examine and understand this structure, you will sometimes see things that don't look right — these are often places where you should look for problems.

Diving on a routine that doesn't look right can isolate the processes into their own group so that you can find out what is occurring there. Be aware that diving on a routine overwrites the group, so if you want to preserve the group, use the **Groups > Custom Groups** command to make a copy.

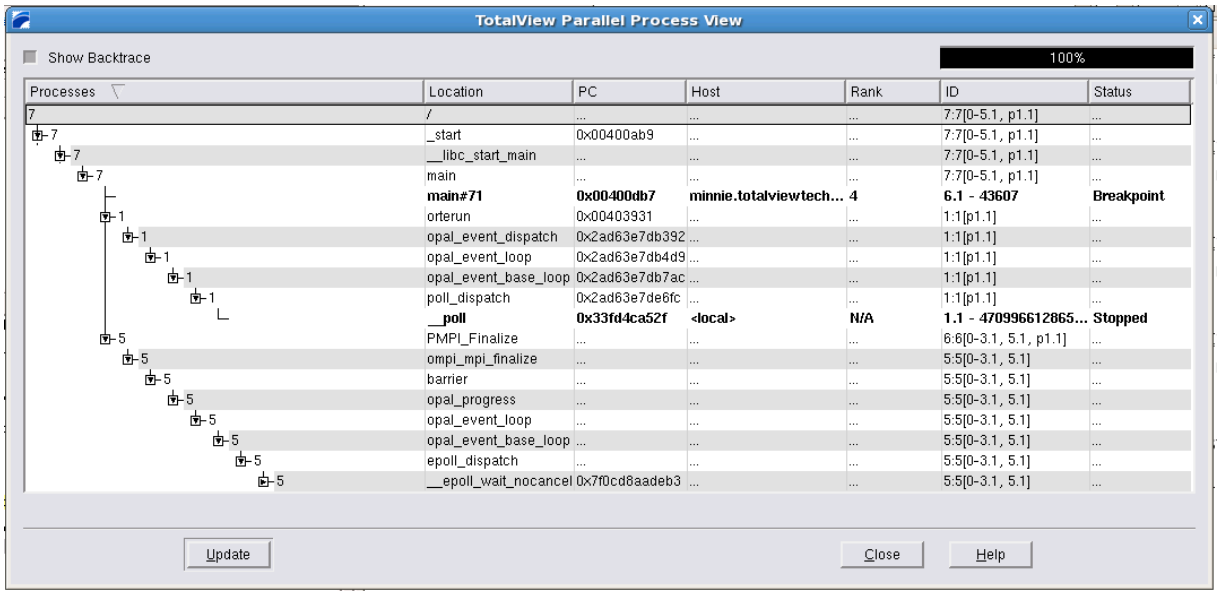
A call tree or call graph can also reveal bottlenecks. For example, if one routine is used by many other routines and controls a shared resource, this thread might be negatively affecting performance.

Parallel Backtrace View

The Parallel Backtrace View displays in a single window the state of every process and thread in a parallel job, including the host, status, process ID, rank, and location. In this way, you can view thousands of processes at once, helping identify stray processes.

Access the Parallel Backtrace View from the Tools menu.

Figure 171, Parallel Backtrace View

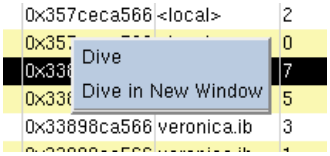


The Parallel Backtrace View shows the position of a program's processes and threads at the same time, displayed as a branching tree with the number and location of each process or thread at each point, as follows:

- **Processes:** the number of processes/threads at a particular location, shown as a branching tree. Expanding the branch shows the next level of the call hierarchy, eventually down to the line of source code. At each level the number of processes in the first column may change.
- **Location:** the location of the process/thread with line number if applicable.
- **PC:** the program counter of the process/thread.
- **Host:** the node on which the process/thread is executing.
- **Rank:** the thread rank of a parallel program. N/A indicates no rank.

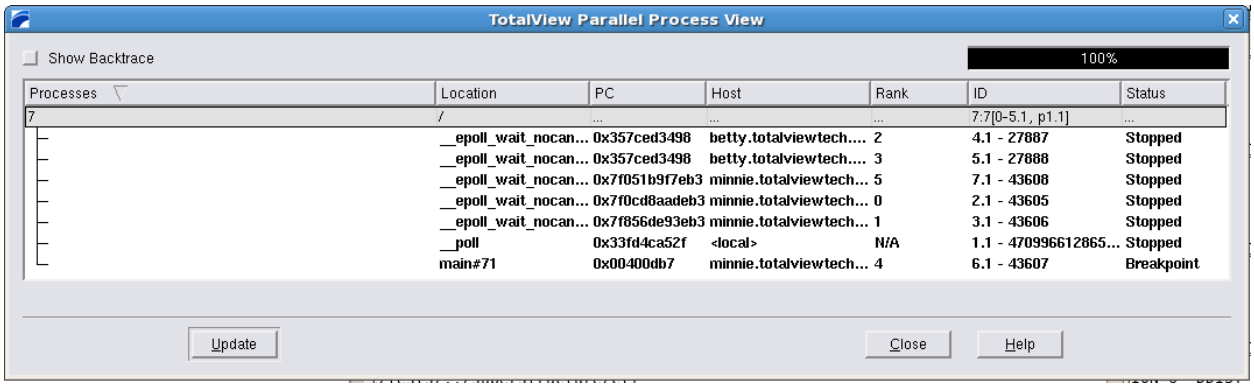
- **ID:** a compressed **ptlist** composed of a process and thread count, followed by square-bracket-enclosed list of process and thread ranges separated by dot (.). See **ptlist** in the *Reference Guide* for more information.
- **Status:** process status.

Diving (with the right mouse button) on each expanded item displays its process window



The progress indicator in the upper right reports the progress of collecting and displaying information.

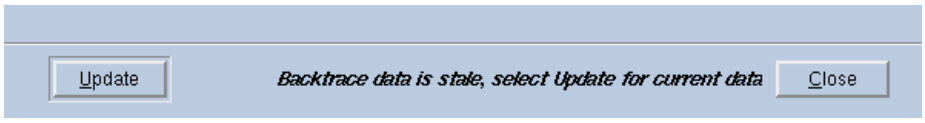
Figure 172, Parallel Backtrace View without Branches



Using the Show Backtrace toggle in the upper left hides the intervening branches and displays the start routine and current execution location of the processes or threads. This removes some of the clutter in the display, as shown above.

If a thread/process state changes, the data becomes stale, and an alert is displayed at the bottom of the window, Figure 173.

Figure 173, Stale Data Message



Use the Update button to refresh the display.

RELATED TOPICS

The **dcalltree** command	**dcalltree** in "CLI Commands" in the *Classic TotalView Reference Guide*

Array Visualizer

The TotalView Visualizer creates graphic images of your program's array data. Topics in this section are:

Command Summary

This section summarizes Visualizer commands.

Action		Click or Press
Camera mode	Actor mode	
Rotate camera around focal point (surface only)	Rotate actor around focal point (surface only)	Left mouse button
Zoom	Scale	Right mouse button
Pan	Translate	Middle mouse button or Shift-left mouse button
Other Functions		
Pick (show value)		**p**
Camera mode: mouse events affect the camera position and focal point. (The axis moves and you don't.)		**c**

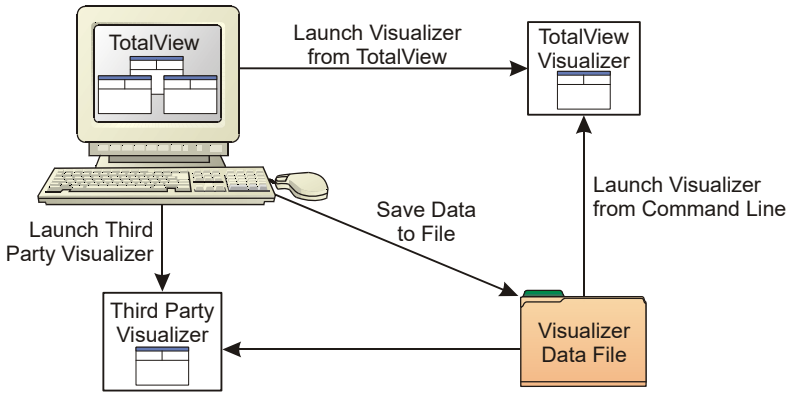
Action	Click or Press
Actor mode: mouse events affect the actor that is under the mouse pointer. (You move and the axis doesn't.)	**a**
Joystick mode: motion occurs continuously while a mouse button is pressed	**j**
Trackball mode: motion occurs only when the mouse button is pressed and the mouse pointer moves.	**t**
Wireframe view	**w**
Surface view	**s**
Reset	**r**
Initialize	**I**
Exit or Quit	**Ctrl-Q**

How the Visualizer Works

The Visualizer is a stand-alone program to which TotalView sends information. Because it is separate, you can use it in multiple ways:

- You can see your program's data while debugging in TotalView.
- You can save the data that would be sent to the Visualizer, and view it later by invoking the Visualizer from the command line.

Figure 174, TotalView Visualizer Relationships



- You can use a third party tool to read the datastream sent by TotalView, rather than using the Visualizer.

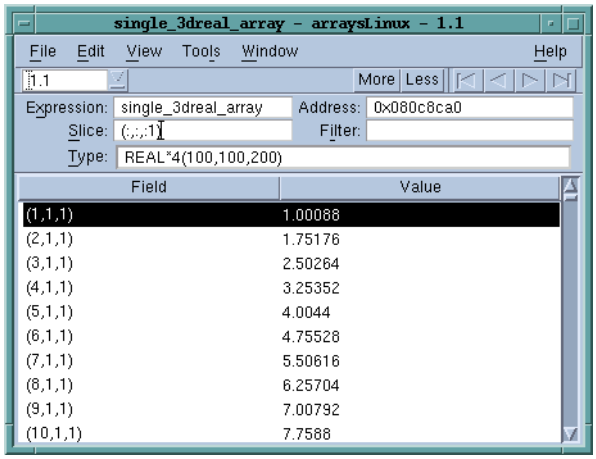
NOTE: For more information on adapting a third-party visualizer so that it can be used with TotalView, see Adapting a Third Party Visualizer on page 357.

Viewing Data Types in the Visualizer

The data selected for visualization is called a dataset. TotalView treats stack variables at different recursion levels or call paths as different datasets.

TotalView can visualize one- and two-dimensional arrays of integer or floating-point data. If an array has more than two dimensions, you can visualize part of it using an array slice that creates a subarray with fewer dimensions. Figure 175 shows a three-dimensional variable sliced so that one of the dimensions is invariant.

Figure 175, A Three-Dimensional Array Sliced into Two Dimensions



RELATED TOPICS

Other ways to examine arrays	Examining Arrays on page 312

Viewing Data

Different datasets can require different views to display their data. For example, a graph is more suitable for displaying one- or two-dimensional datasets if one of the dimensions has a small extent. However, a surface view is better for displaying a two-dimensional dataset.

When TotalView launches the Visualizer, one of the following actions occurs:

- If the Visualizer is displaying the dataset, it raises the dataset's window to the top of the desktop. If you had minimized the window, the Visualizer restores it.
- If you previously visualized a dataset but you've killed its window, the Visualizer creates a new window using the most recent visualization method.
- If you haven't visualized the dataset, the Visualizer chooses an appropriate method. You can disable this feature by using the **Options > Auto Visualize** command in the Visualizer Directory Window.

Visualizing Data Manually

Before you can visualize an array:

- Open a Variable Window the array.
- Stop program execution when the array's values reflect what you want to visualize.

You can restrict the visualized data by editing the **Slice** field. (See Displaying Array Slices on page 313.) Limiting the amount of data increases the speed of the Visualizer.

After selecting the Variable Window **Tools > Visualize** command, the Visualizer creates its window.

NOTE: As you step through your program, be aware that the data sent to the Visualizer is not automatically updated; explicitly update the display using *Tools* > *Visualize*.

TotalView can visualize variables across threads or processes. (See Visualizing a "Show Across" Variable Window on page 333.) In this case, the Visualizer uses the process or thread index as one dimension, meaning that you can visualize only scalar or vector information. If you do not want the process or thread index to be a dimension, do not use a **Show Across** command.

Using the Visualizer

The Visualizer uses two types of windows:

- **Dataset Window**

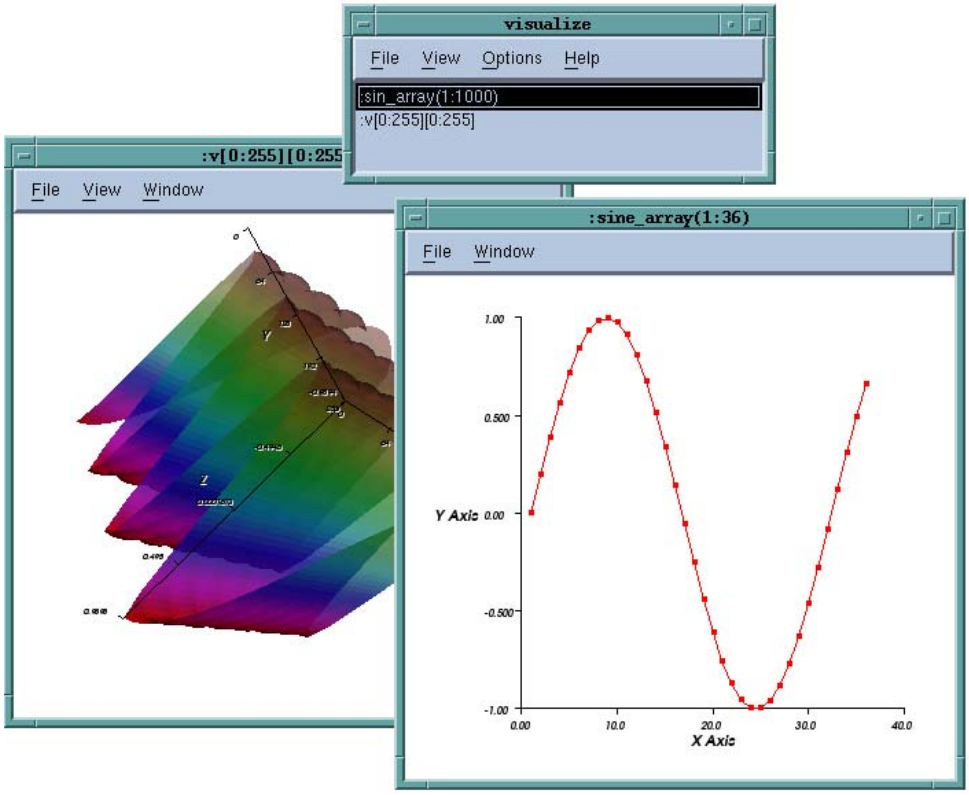
  This window contains the datasets that you can visualize. Use this window to set global options and to create views of your datasets. Commands in this window provide different views of the same data by allowing you to open more than one View Window.

- **View Window**

These windows actually display your data. The commands in a View Window set viewing options and change the way the Visualizer displays your data.

In Figure 176, the top window is a Dataset Window. The two remaining windows show a surface and a graph view.

Figure 176, Sample Visualizer Windows



Using Dataset Window Commands

The Dataset Window lists the datasets you can display. Double-click on a dataset to display it.

The **View** menu supports either **Graph** or **Surface** visualization. When TotalView sends a new dataset to the Visualizer, the Visualizer updates its dataset list. To delete a dataset from the list, click on it, display the File menu, and then select Delete. (It's usually easier to just close the Visualizer.)

The following commands are in the Dataset Window menu bar:

File > Delete Deletes the currently selected dataset. It removes the dataset from the list and *destroys* the View Window that displays it.

File > Exit Closes all windows and exits the Visualizer.

View > Graph Creates a new Graph Window; see Using the Graph Window on page 347.

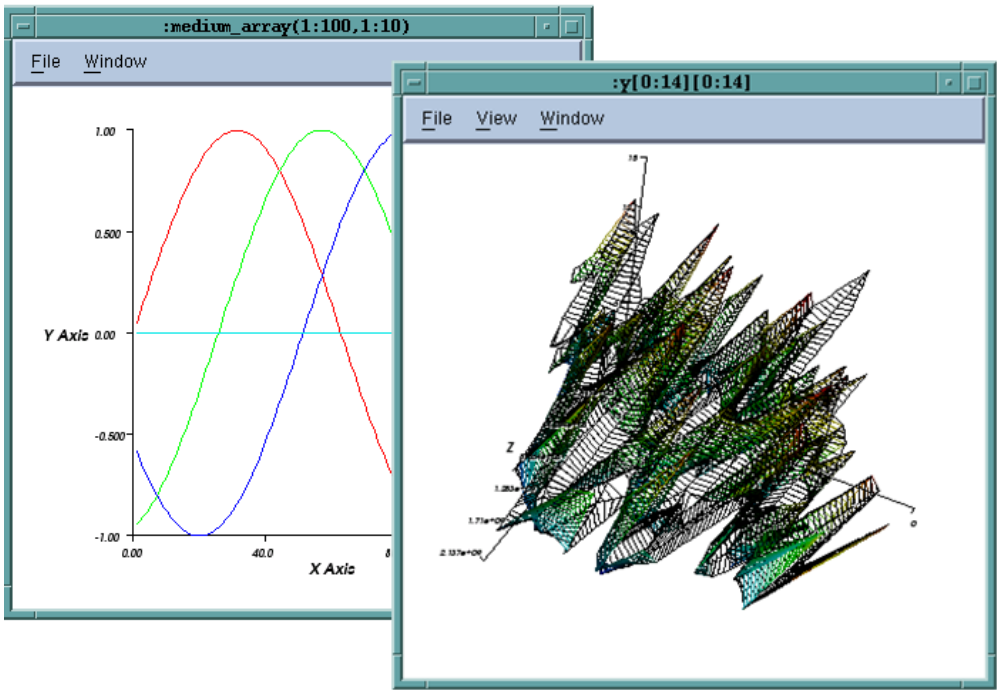
View > Surface Creates a new Surface Window; see Using the Surface Window on page 350.

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 want to configure how the Visualizer displays the graph, disable this option.

Using View Window Commands

View Windows display graphic images of your data. Figure 177 shows a graph view and a surface view. The View Window's title is the text that appears in the Dataset Window.

Figure 177, Graph and Surface Visualizer Windows



The View Window menu commands are:

File > Close Closes the View Window.

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

File > Delete Deletes the View Window dataset from the list. This also destroys other View Windows that view the dataset.

File > Options Pops up a window of viewing options.

Window > Duplicate Base Window
Creates a new View Window with the same visualization method and dataset as the current View Window.

Ways to view data

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 displaying a surface, you can rotate the surface to view it from different angles.
- You can get the value and indices of the dataset element nearest the cursor by clicking on it and typing "**P**". A pop-up window displays the information.

  These operations are discussed in Using the Graph Window on page 347 and Using the Surface Window on page 350.

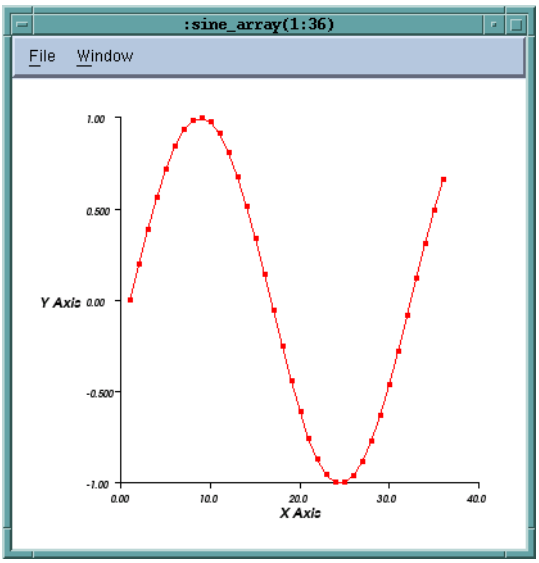
Using the Graph Window

The Graph Window displays a two-dimensional graph of one- or two-dimensional datasets. If the dataset is two-dimensional, the Visualizer displays multiple graphs. When you first create a Graph Window on a two-dimensional dataset, the Visualizer uses the dimension with the larger number of elements for the **X** axis. It then draws a separate graph for each subarray that has the smaller number of elements. If you don't like this choice, you can transpose the data by selecting a checkbox within the **File > Options** Dialog Box.

NOTE: You probably don't want to use a graph to visualize two-dimensional datasets with large extents in both dimensions as the display can be very cluttered. If you try, the Visualizer shows only the first ten.

You can display graphs with points for each element of the dataset, with lines connecting dataset elements, or with both lines and points, as demonstrated in Figure 178.

Figure 178, Visualizer Graph View Window



If the Visualizer is displaying more than one graph, each is a different color. The **X** axis is annotated with the indices of the long dimension. The **Y** axis shows you the data value.

Displaying Graph Views

The **File > Options** Dialog Box controls graph display. (A different dialog box appears if the Visualizer is displaying a surface view.)

Figure 179, Graph Options Dialog Box

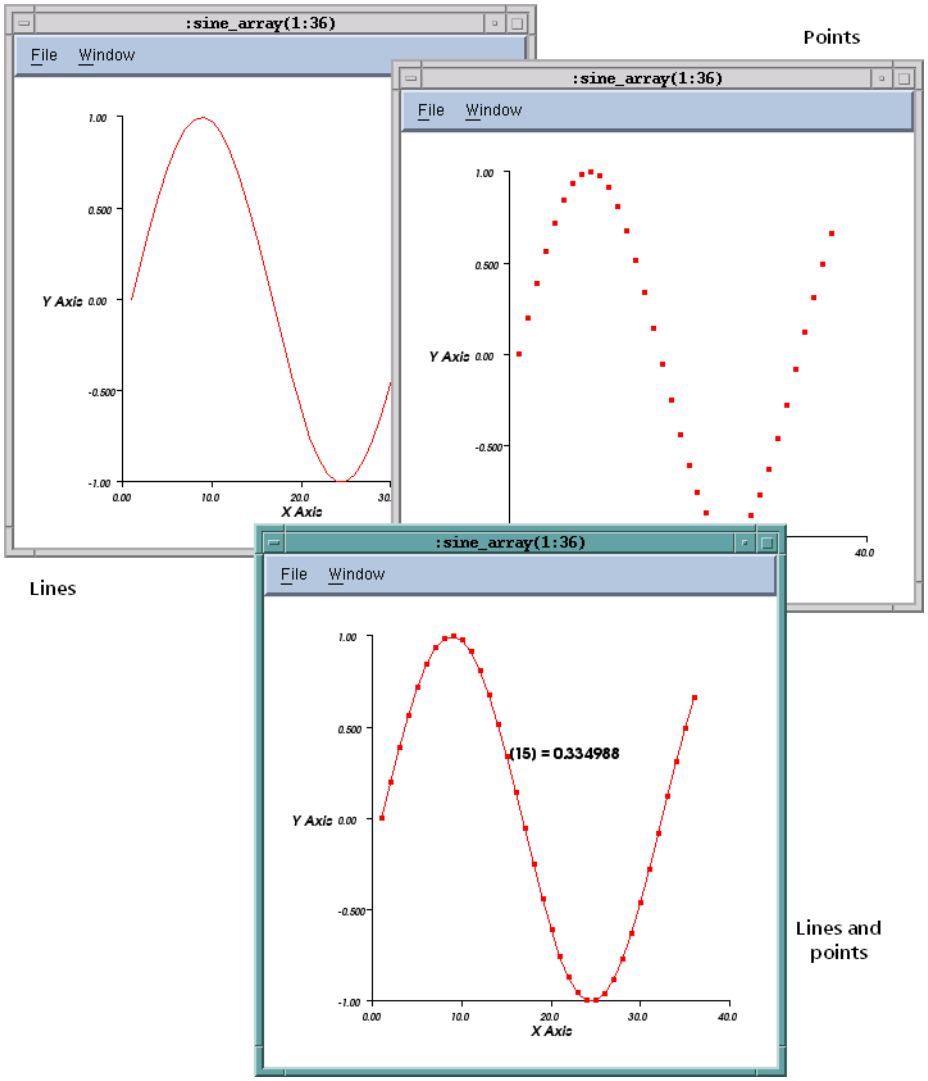
Options:

Lines	Displays lines connecting dataset elements.
Points	Displays points (markers) for dataset elements.

Transpose	Inverts which axis is held constant when generating a graph of a two-dimensional object. For other than two dimensions, this option is not available.

Figure 180 shows a sine wave displayed in three different ways:

Figure 180, Sine wave Displayed in Three Ways



To see the value of a dataset's element, place your cursor near a graph marker, and type "**P**". The bottom graph in Figure 180 shows the value of a data point.

Using the Surface Window

The Surface Window displays two-dimensional datasets as a surface in two or three dimensions. The dataset's array indices map to the first two dimensions (**X** and **Y** axes) of the display. Figure 181 shows a surface view:

Figure 181, A Surface View

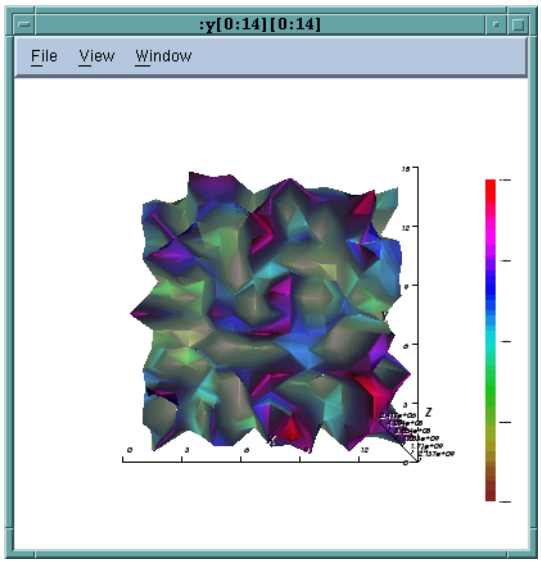
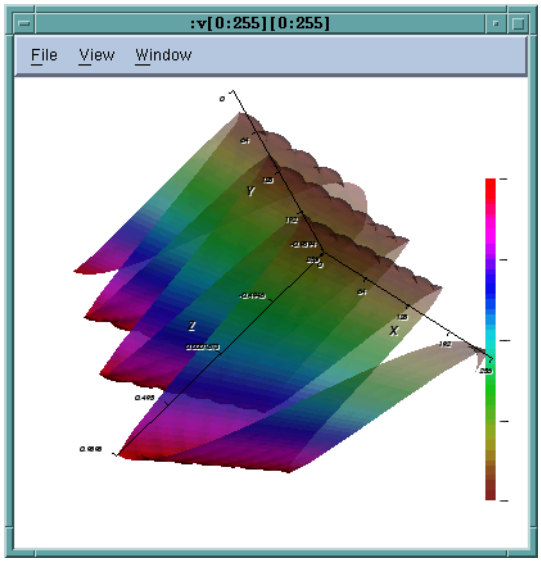


Figure 182 shows a three-dimensional surface that maps element values to the height (**Z** axis).

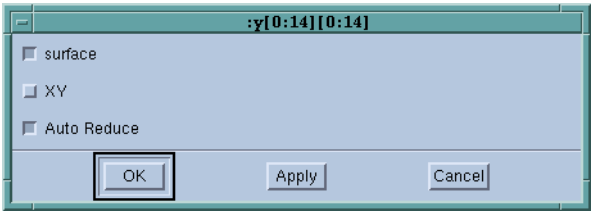
Figure 182, A Surface View of a Sine Wave



Displaying Surface Views

The Surface Window **File > Options** command controls surface display, Figure 183 (A different dialog box controls Graph View.)

Figure 183, Surface Options Dialog Box

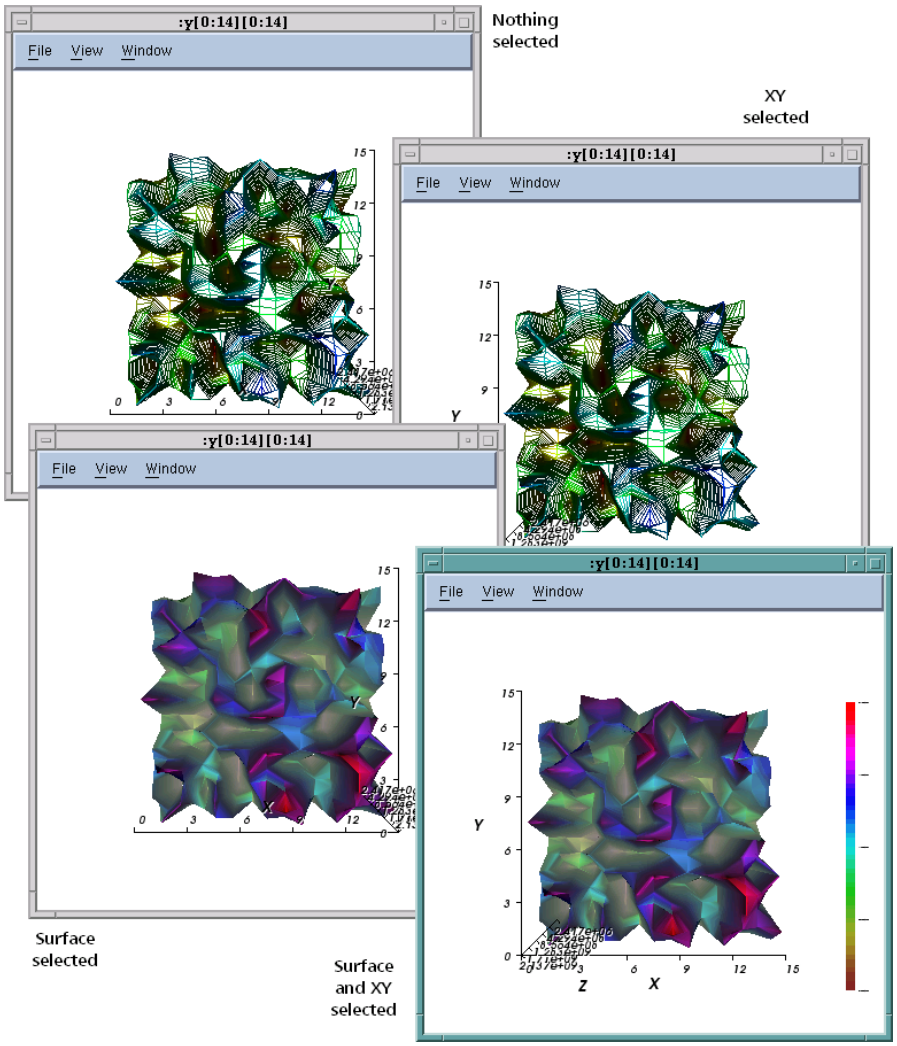


Options:

Surface	Displays the array's data as a three-dimensional surface; otherwise, displays the surface as a grid.
XY	Reorients the view's XY axes. The Z axis is perpendicular to the display.
Auto Reduce	Derives the displayed surface by averaging neighboring elements in the original dataset, in order to speed visualization by reducing surface resolution. Clear this option to accurately visualize all dataset elements. This option supports either viewing all your data points — which takes longer to appear in the display — or viewing the data average over a number of nearby points.

Figure 184 shows different views of the same data, based on Surface and XY options.

Figure 184, Four Surface Views



To restore initial state of translation, rotation, and scaling options, select **View > Initialize View**.

Manipulating Surface Data

The Surface Window supports various viewing modes. Camera mode is the default, in which the Visualizer behaves as a “camera” moving around the object. Actor mode, by contrast, displays the object as if you, the viewer, were changing position. The difference between these is subtle. In some circumstances, actions such as pan and zoom in camera mode can also add a slight rotation to the object.

From within TotalView, you can see only one array at a time. However, if you combine multiple datasets and visualize them externally, the differences between camera and actor mode can help differentiate the objects.

The following table defines all surface view general commands. Command letters can be typed in either upper- or lower-case.

Action	Press
Pick (show value): Displays the value of the data point at the cursor.	**p**
Camera mode: Mouse events affect the camera position and focal point. (Axes move, and you don't.)	**c**
Actor mode: Mouse events affect the actor under the mouse pointer. (You move, not the axes.)	**a**
Joystick mode: Motion occurs continuously while you press a mouse button.	**j**
Trackball mode: Motion occurs only when you press the mouse button *and* you move the mouse pointer.	**t**
Wireframe view: Displays the surface as a mesh. (This is the same as not checking the Surface option.)	**w**
Surface view: Displays the surface as a solid. (This is the same as having checked the Surface option.)	**s**
Reset: Removes the changes you've made to the way the Visualizer displays an object.	**r**
Initialize: Restores the object to its initial state before you interacted with the Visualizer. As this is a menubar accelerator, the window must have focus.	**i**
Exit or **Quit**: Close the Visualizer.	**Ctrl-Q**

The following table defines the actions you can perform using your mouse:

Action		Click or Press
Camera mode	**Actor mode**	
Rotate camera around focal point (surface only)	**Rotate** actor around focal point (surface only)	Left mouse button

Action		Click or Press
Zoom: Zooms in on the object.	**Scale**: the object appears to get larger	Right mouse button
Pan: Moves the "camera". For example, moving the camera up means the object moves down.	**Translate**: The object moves in the direction you pull it.	Middle mouse button or Shift-left mouse button

Visualizing Data Programmatically

The **$visualize** function supports data visualization from within eval points and the **Tools > Evaluate** Window. Because you can enter more than one **$visualize** function within an eval point or Evaluate Window, you can simultaneously visualize multiple variables.

If you enter the **$visualize** function in an eval point, TotalView interprets rather than compiles the expression, which can greatly decrease performance. See Defining Eval Points and Conditional Breakpoints on page 220 for information about compiled and interpreted expressions.

Using the **$visualize** function in an eval point lets you animate the changes that occur in your data, because the Visualizer updates the array's display every time TotalView reaches the eval point. Here is this function's syntax:

$visualize (*array* [, *slice_string*])

The *array* argument names the dataset being visualized. The optional *slice_string* argument is a quoted string that defines a constant slice expression that modifies the *array* parameter's dataset. In Fortran, you can use either a single (') or double (") quotation mark. You must use a double quotation mark in C or C++.

The following examples show how you can use this function. Notice that the array's dimension ordering differs between C/C++ and Fortran.

C and C++

```
$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 knows what the array's dimensions are. For example, here is a C function that passes a two-dimensional array parameter that does not specify the major dimension's extent.

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

You would need to cast this before TotalView can visualize it. For example:

```
$visualize (*(double[32][32]*)my_array);
```

Sometimes, it's hard to know what to specify. You can quickly refine array and slice arguments, for example, by entering the **$visualize** function into the **Tools > Evaluate** Dialog Box. When you select the **Evaluate** button, you quickly see the result. You can even use this technique to display several arrays simultaneously.

RELATED TOPICS	
Eval points and conditional breakpoints	Defining Eval Points and Conditional Breakpoints on page 220
Writing expressions in various TotalView-supported languages	Using Programming Language Elements on page 367

Launching the Visualizer from the Command Line

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

visualize [**-file** *filename* | **-persist**]

where:

-file *filename*	Reads data from *filename* instead of reading from standard input. For information on creating this file, see Setting the Visualizer Launch Command on page 357.
-persist	Continues to run after encountering an EOF (End-of-File) on standard input. If you don't use this option, the Visualizer exits as soon as it reads all the data.

By default, the Visualizer reads its datasets from standard input and exits when it reads an EOF. When started by TotalView, the Visualizer reads its data from a pipe, ensuring that the Visualizer exits when TotalView does. If you want the Visualizer to continue to run after it exhausts all input, invoke it by using the **-persist** option.

If you want to read data from a file, invoke the Visualizer with the **-file** option:

visualize -file *my_data_set_file*

The Visualizer reads all the datasets in the file. This means that the images you see represent the last versions of the datasets in the file.

The Visualizer supports the generic X toolkit command-line options. For example, you can start the Visualizer with the Directory Window minimized by using the **-iconic** option. Your system manual page for the X server or the *X Window System User's Guide* by O'Reilly & Associates lists the generic X command-line options in detail.

You can also customize the Visualizer by setting X resources in your resource files or on the command line with the **-xrm resource_setting** option.

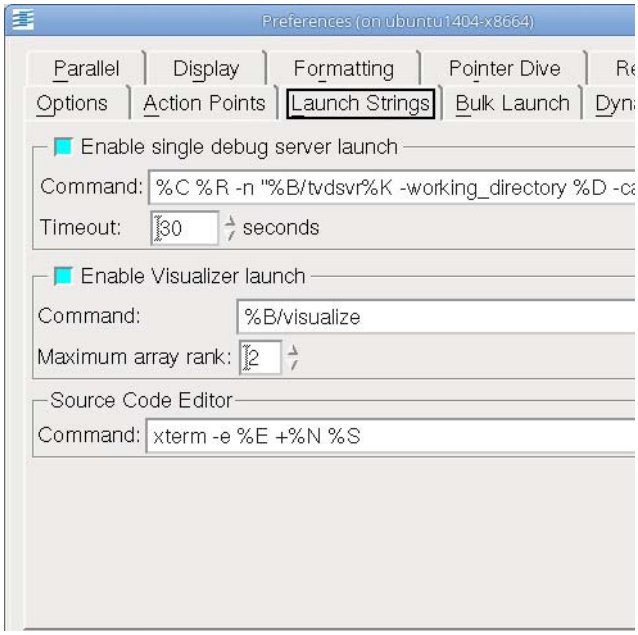
Configuring TotalView to Launch the Visualizer

TotalView launches the Visualizer when you select the **Tools > Visualize** command from the Variable Window. It also launches it when using a **$visualize** function in an eval point and the **Tools > Evaluate** Dialog Box.

You can disable visualization entirely. This lets you turn off visualization when your program executes code that contains eval points, without having to individually disable them all.

To change the Visualizer launch options interactively, select **File > Preferences**, and then select the Launch Strings Tab.

Figure 185, File > Preferences Launch Strings Page



Options:

- Customize the command used to start a visualizer by entering the visualizer's start up command in the **Command** edit box.

- Change the autolaunching option. If you want to disable visualization, clear the **Enable Visualizer launch** check box.
- Change the maximum permissible rank. Edit the value in the **Maximum array rank** field to save the data exported from TotalView or display it in a different visualizer. A rank's value can range from **1** to **16**.

  Setting the maximum permissible rank to either 1 or 2 (the default is 2) ensures that the Visualizer can use your data—the Visualizer displays only two dimensions of data. This limit doesn't apply to data saved in files or to third-party visualizers that can display more than two dimensions of data.
- Clicking the **Defaults** button returns all values to their default values. This reverts options to their default values even if you have used X resources to change them.

If you disable visualization while the Visualizer is running, TotalView closes its connection to the Visualizer. If you reenable visualization, TotalView launches a new Visualizer process the next time you visualize something.

RELATED TOPICS

The **File > Preferences** command	**File > Preferences** in the in-product Help

Setting the Visualizer Launch Command

You can change the shell command that TotalView uses to launch the Visualizer by editing the Visualizer launch command. (In most cases, the only reason you'd do this is if you're having path problems or you're running a different visualizer.) You can also change what's entered here so that you can view this information at another time; for example:

```
cat > your_file
```

Later, you can visualize this information by typing either:

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

You can preset the Visualizer launch options by setting X resources.

Adapting a Third Party Visualizer

TotalView passes a stream of datasets to the Visualizer encoded in the format described below, thus supporting the use of this data with other programs, with these requirements:

- TotalView and the Visualizer must be running on the same machine architectures; that is, TotalView assumes that word lengths, byte order, and floating-point representations are identical. While sufficient information in the dataset header exists to detect when this is not the case (with the exception of floating-point representation), no method for translating this information is supplied.

- TotalView transmits datasets down the pipe in a simple unidirectional flow. There is no handshaking protocol in the interface. This requires the Visualizer to be an eager reader on the pipe. If the Visualizer does not read eagerly, the pipe will back up and block TotalView.

Visualizer dataset format

The dataset format is described in the TotalView distribution in a header file named **include/visualize.h** in the TotalView installation directory. Each dataset is encoded with a fixed-length header followed by a stream of array elements. The header contains the following fields:

vh_axis_order	Contains one of the constants **vis_ao_row_major** or **vis_ao_column_major**.
vh_dims	Contains information on each dimension of the dataset. This includes a base, count, and stride. Only the count is required to correctly parse the dataset. The base and stride give information only on the valid indices in the original data. Note that all **VIS_MAXDIMS** of dimension information is included in the header, even if the data has fewer dimensions.
vh_effective_rank	Contains the number of dimensions that have an extent larger than 1.
vh_id	Contains the dataset ID. Every dataset in a stream of datasets is numbered with a unique ID so that updates to a previous dataset can be distinguished from new datasets.
vh_item_count	Contains the total number of expected elements.
vh_item_length	Contains the length (in bytes) of a single element of the array.
vh_magic	Contains **VIS_MAGIC**, a symbolic constant to provide a check that this is a dataset header and that byte order is compatible.
vh_title	Contains a plain text string of length **VIS_MAXSTRING** that annotates the dataset.
vh_type	Contains one of the constants **vis_signed_int**, **vis_unsigned_int**, or **vis_float**.
vh_version	Contains **VIS_VERSION**, a symbolic constant to provide a check that the reader understands the protocol.

Types in the dataset are encoded by a combination of the **vh_type** field and the **vh_item_length** field. This allows the format to handle arbitrary sizes of both signed and unsigned integers, and floating-point numbers.

The **vis_float** constant corresponds to the default floating-point format (usually, IEEE) of the target machine. The Visualizer does not handle values other than the default on machines that support more than one floating-point format.

Although a three-byte integer is expressible in the Visualizer's dataset format, it is unlikely that the Visualizer will handle one. The Visualizer handles only data types that correspond to the C data types permitted on the machine where the Visualizer is running.

Similarly, the long double type varies significantly depending on the C compiler and target machine. Therefore, visualization of the long double type is unlikely to work if you run the Visualizer on a machine different from the one where you extracted the data.

In addition, be aware of these data type differences if you write your own visualizer and plan to run it on a machine that is different from the one where you extract the data.

The data following the header is a stream of consecutive data values of the type indicated in the header. Consecutive data values in the input stream correspond to adjacent elements in **vh_dims[0]**.

You can verify that your reader's idea of the size of this type is consistent with TotalView by checking that the value of the n_bytes field of the header matches the product of the size of the type and the total number of array elements.

Evaluating Expressions

Whether you realize it or not, you've been telling TotalView to evaluate expressions and you've even been entering them. In every programming language, variables are actually expressions—actually they are lvalues—whose evaluation ends with the interpretation of memory locations into a displayable value. Structure, pointer and array variables, particularly arrays where the index is also a variable, are slightly more complicated.

While debugging, you also need to evaluate expressions that contain function calls and programming language elements such as **for** and **while** loops.

This chapter discusses what you can do evaluating expressions within TotalView:

Why is There an Expression System?

Either directly or indirectly, accessing and manipulating data requires an evaluation system. When your program (and TotalView, of course) accesses data, it must determine where this data resides. The simplest data lookups involve two operations: looking up an address in your program's symbol table and interpreting the information located at this address based on a variable's datatype. For simple variables such as an integer or a floating point number, this is all pretty straightforward.

Looking up array data is slightly more complicated. For example, if the program wants **my_var[9]**—this chapter will most often use C and C++ notation rather than Fortran—it looks up the array's starting address, then applies an offset to locate the array's 10th element. In this case, if each array element uses 32 bits, **my_var[9]** is located 9 times 32 bits away.

In a similar fashion, your program obtains information about variables stored in structures and arrays of structures.

Structures complicate matters slightly. For example **ptr->my_var** requires three operations: extract the data contained within address of the **my_var** variable, use this information to access the data at the address being pointed to, then display the data according to the variable's datatype.

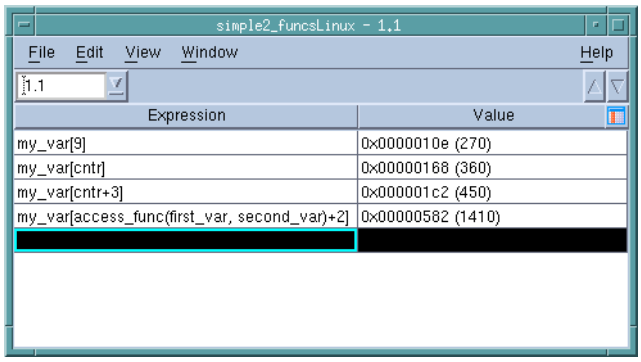
Accessing an array element such as **my_var[9]** where the array index is an integer constant is rare in most programs. In most cases, your program uses variables or expressions as array indices; for example, **my_var[cntr]** or **my_var[cntr+3]**. In the later case, TotalView must determine the value of **cntr+3** before it can access an array element.

Using variables and expressions as array indices are common. However, the array index can be (and often is) an integer returned by a function. For example:

```
my_var[access_func(first_var, second_var)+2]
```

In this example, a function with two arguments returns a value. That returned value is incremented by two, and the resulting value becomes the array index. Here is an illustration showing TotalView accessing the **my_var** array in the four ways discussed in this section:

Figure 186, Expression List Window: Accessing Array Elements



In Fortran and C, access to data is usually through variables with some sort of simple evaluation or a function. Access to variable information can be the same in C++ as it is in these languages. However, accessing private variables within a class almost always uses a method. For example:

```
myDataStructureList.get_current_place()
```

TotalView built-in expression evaluation system is able to understand your class inheritance structure in addition to following C++ rules for method invocation and polymorphism. (This is discussed in Using C++ on page 364.)

Calling Functions: Problems and Issues

Unfortunately, calling functions in the expression system can cause problems. Some of these problems are:

- What happens if the function has a side effect For example, suppose you have entered **my_var[cntr]** in one row in an Expression List Window, followed by **my_var[++cntr]** in another. If **cntr** equals 3, you'll be seeing the values of **my_var[3]** and **my_var[4]**. However, since cntr now equals 4, the first entry is no longer correct.
- What happens when the function crashes (after all you are trying to debug problems), doesn't return, returns the wrong value, or hits a breakpoint?
- What does calling functions do to your debugging interaction if evaluation takes an excessive amount of time?
- What happens if a function creates processes and threads? Or worse, kills them?

In general, there are some protections in the code. For example, if you're displaying items in an **Expression List** Window, TotalView avoids being in an infinite loop by only evaluating items once. This does mean that the information is only accurate at the time at which TotalView made the evaluation.

In most other cases, you're basically on your own. If there's a problem, you'll get an error message. If something takes too long, you can press the Halt button. But if a function alters memory values or starts or stops processes or threads and you can't live with it, you'll need to restart your program. However, if an error occurs while using the **Evaluate** Window, pressing the **Stop** button pops the stack, leaving your program in the state it was in before you used the **Evaluate** command. However, changes made to heap variables will, of course, not be undone.

Expressions in Eval Points and the Evaluate Window

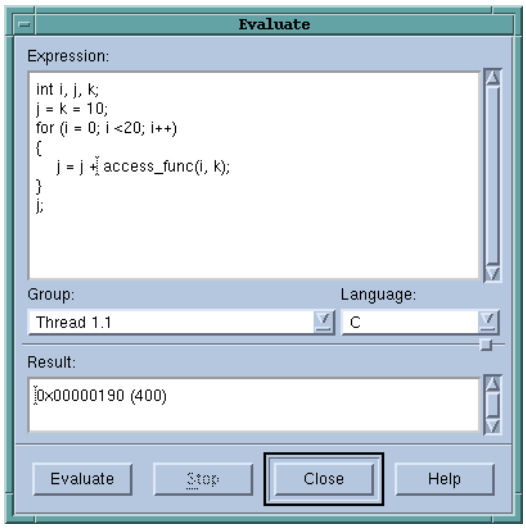
Expression evaluation is not limited to a Variable Window or an Expression List Window. You can use expressions within eval points and in the **Tools > Evaluate** Window. The expressions you type here also let you use programming language constructs. For example, here's a trivial example of code that can execute within the **Evaluate** Window:

```
int i, j, k;
j = k = 10;
for (i=0; i< 20; i++)
{
    j = j + access_func(i, k);
}
j;
```

This code fragment declares a couple of variables, runs them through a **for** loop, then displays the value of **j**. In all cases, the programming language constructs being interpreted or compiled within TotalView are based on code within TotalView. TotalView is not using the compiler you used to create your program or any other compiler or interpreter on your system.

Notice the last statement inFigure 187. TotalView displays the value returned by the last statement. This value is displayed. (See Displaying the Value of the Last Statement on page 364.)

Figure 187, Displaying the Value of the Last Statement



TotalView assumes that there is always a return value, even if it's evaluating a loop or the results of a subroutine returning a void. The results are, of course, not well-defined. If the value returned is not well-defined, TotalView displays a zero in the **Result** area.

The code within eval points and the **Evaluate** Window does not run in the same address space as that in which your program runs. Because TotalView is a debugger, it knows how to reach into your program's address space. The reverse isn't true: your program can't reach into the TotalView address space. This forces some limitations upon what you can do. In particular, you can not enter anything that directly or indirectly needs to pass an address of a variable defined within the TotalView expression into your program. Similarly, invoking a function that expects a pointer to a value and whose value is created within TotalView can't work. However, you can invoke a function whose parameter is an address and you name something within that program's address space. For example, you could say something like **adder(an_array)** if **an_array** is contained within your program.

Using C++

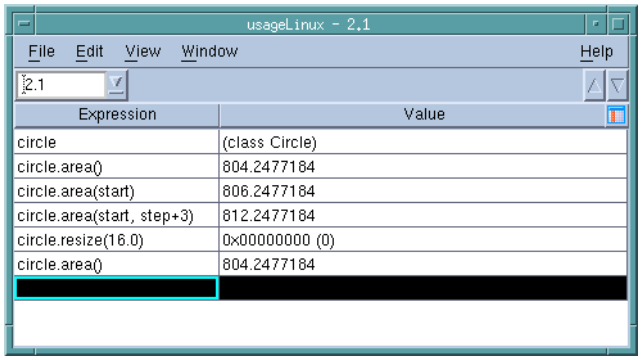
The TotalView expression system is able to interpret the way you define your classes and their inheritance hierarchy. For example, if you declare a method in a base class and you invoke upon an object instantiated from a derived class, TotalView knows how to access the function. It also understands when a function is virtual. For example, assume that you have the following declarations:

```
class Circle : public Shape {
public:
    ...
```

```
virtual double area();
virtual double area(int);
double area(int, int);
```

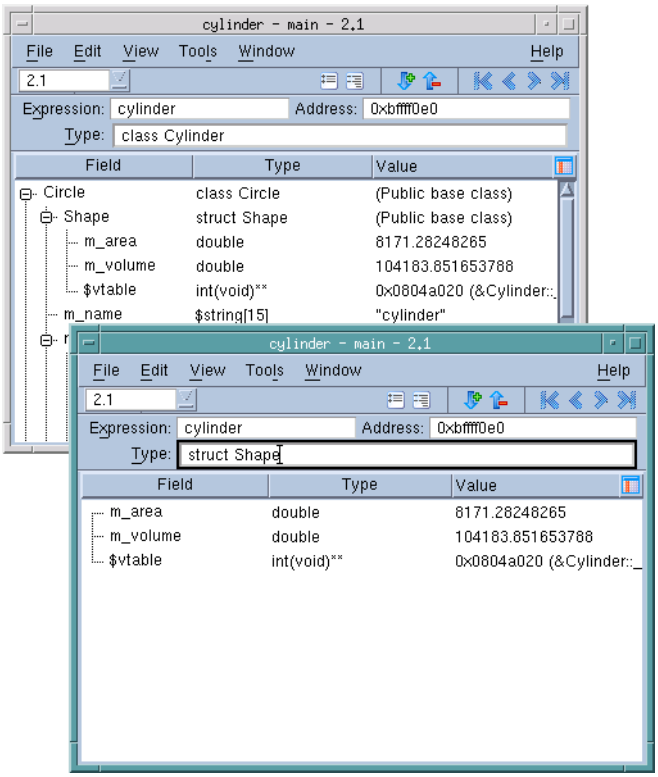
Figure 188 shows an expression list calling an overloaded function. It also shows a setter (mutator) that changes the size of the circle object. A final call to area shows the new value.

Figure 188, Expression List Window: Showing Overloads



If your object is instantiated from a class that is part of an inheritance hierarchy, TotalView shows you the hierarchy when you dive on the object.

Figure 189, Class Casting



Using Programming Language Elements

Using C and C++

This section contains guidelines for using C and C++ in expressions.

- You can use C-style (**/* comment */**) and C++-style (**// comment)** comments; for example:

  ```
  // This code fragment creates a temporary patch
  i = i + 2; /* Add two to i */
  ```

- You can omit semicolons if the result isn't ambiguous.
- You can use dollar signs ($) in identifiers. However, we recommend that you do not use dollar signs in names created within the expression system.

NOTE: If your program does not use a templated function within a library, your compiler may not include a reference to the function in the symbol table. That is, TotalView does not create template instances. In some cases, you might be able to overcome this limitation by preloading the library. However, this only works with some compilers. Most compilers only generate STL operators if your program uses them.

You can use the following C and C++ data types and declarations:

- You can use all standard data types such as **char**, **short**, **int**, **float**, and **double**, modifiers to these data types such as **long int** and **unsigned int**, and pointers to any primitive type or any named type in the target program.
- You can only use simple declarations. Do not define **stuct**, **class**, **enum** or **union** types or variables.

  You can define a pointer to any of these data types. If an **enum** is already defined in your program, you can use that type when defining a variable.
- The **extern** and **static** declarations are not supported.

You can use the following the C and C++ language statements.

- You can use the **goto** statement to define and branch to symbolic labels. These labels are local to the window. You can also refer to a line number in the program. This line number is the number displayed in the Source Pane. For example, the following **goto** statement branches to source line number 432 of the target program:

  ```
  goto 432;
  ```

- Although you can use function calls, you can't pass structures.
- You can use type casting.
- You can use assignment, **break**, **continue**, **if**/**else** structures, **for**, **goto**, and **while** statements. Creating a **goto** that branches to another TotalView evaluation is undefined.

Using Fortran

When writing code fragments in Fortran, you need to follow these guidelines:

- In general, you can use free-form syntax. You can enter more than one statement on a line if you separate the statements with semi-colons (;). However, you cannot continue a statement onto more than one line.
- You can use **GOTO**, **GO TO**, **ENDIF**, and **END IF** statements; Although **ELSEIF** statements aren't allowed, you can use **ELSE IF** statements.
- Syntax is free-form. No column rules apply.
- The space character is significant and is sometimes required. (Some Fortran 77 compilers ignore all space characters.) For example:

Valid	Invalid
`DO 100 I=1,10`	`DO100I=1,10`
`CALL RINGBELL`	`CALL RING BELL`
`X .EQ. 1`	`X.EQ.1`

You can use the following data types and declarations in a Fortran expression:

- You can use the **INTEGER**, **REAL**, **DOUBLE PRECISION**, and **COMPLEX** data types.
- You can't define or declare variables that have implied or derived data types.
- You can only use simple declarations. You can't use a **COMMON**, **BLOCK DATA**, **EQUIVALENCE**, **STRUCTURE**, **RECORD**, **UNION**, or array declaration.
- You can refer to variables of any type in the target program.
- TotalView assumes that **integer (kind=n)** is an n-byte integer.

Fortran Statements

You can use the Fortran language statements:

- You can use assignment, **CALL** (to subroutines, functions, and all intrinsic functions except **CHARACTER** functions in the target program), **CONTINUE**, **DO**, **GOTO**, **IF** (including block **IF**, **ENDIF**, **ELSE**, and **ELSE IF**), and **RETURN** (but not alternate return) statements.
- If you enter a comment in an expression, precede the comment with an exclamation point (!).
- You can use array sections within expressions. For more information, see Array Slices and Array Sections on page 316.
- A **GOTO** statement can refer to a line number in your program. This line number is the number that appears in the Source Pane. For example, the following **GOTO** statement branches to source line number 432:

  ```
  GOTO $432;
  ```

  You must use a dollar sign (**$**) before the line number so that TotalView knows that you're referring to a source line number rather than a statement label.

  You cannot branch to a label within your program. You can instead branch to a TotalView line number.
- The following expression operators are not supported: **CHARACTER** operators and the **.EQV.**, **.NEQV.**, and **.XOR.** logical operators.
- You can't use subroutine function and entry definitions.
- You can't use Fortran 90 pointer assignment (the **=>** operator).
- You can't call Fortran 90 functions that require assumed shape array arguments.

Fortran Intrinsics

TotalView supports some Fortran intrinsics. You can use these supported intrinsics as elements in expressions. The classification of these intrinsics into groups is that contained within Chapter 13 of the *Fortran 95 Handbook*, by Jeanne C. Adams, *et al.*, published by the MIT Press.

TotalView does not support the evaluation of expressions involving complex variables (other than as the arguments for **real** or **aimag**). In addition, we do not support function versions. For example, you cannot use **dcos** (the double-precision version of **cos**).

The supported intrinsics are:

- Bit Computation functions: **btest**, **iand**, **ibclr**, **ibset**, **ieor**, **ior**, and **not**.
- Conversion, Null and Transfer functions: **achar**, **aimag**, **char**, **dble**, **iachar**, **ichar**, **int**, and **real**.
- Inquiry and Numeric Manipulation Functions: **bit_size**.

- Numeric Computation functions: **acos**, **asin**, **atan**, **atan2**, **ceiling**, **cos**, **cosh**, **exp**, **floor**, **log**, **log10**, **pow**, **sin**, **sinh**, **sqrt**, **tan**, and **tanh**.

  Complex arguments to these functions are not supported. In addition, on MacIntosh and AIX, the **log10**, **ceiling**, and **floor** intrinsics are not supported.

The following are not supported:

- Array functions
- Character computation functions.
- Intrinsic subroutines

NOTE: If you statically link your program, you can only use intrinsics that are linked into your code. In addition, if your operating system is Mac OS X, AIX, or Linux/Power, you can only use math intrinsics in expressions if you directly linked them into your program. The ** operator uses the **pow** function. Consequently, it too must either be used within your program or directly linked. In addition, **ceiling** and **log10** are not supported on these three platforms.

Using the Evaluate Window

TotalView lets you open a window to evaluate expressions in the context of a particular process and evaluate them in C, Fortran, or assembler.

NOTE: Not all platforms let you use assembler constructs. See "Architectures" in the *Classic TotalView Reference Guide* for details.

You can use the **Tools > Evaluate** Dialog Box in many different ways. The following are two examples:

- Expressions can contain loops, so you can use a **for** loop to search an array of structures for an element set to a certain value. In this case, you use the loop index at which the value is found as the last expression in the expression field.
- Because you can call subroutines, you can test and debug a single routine in your program without building a test program to call it.

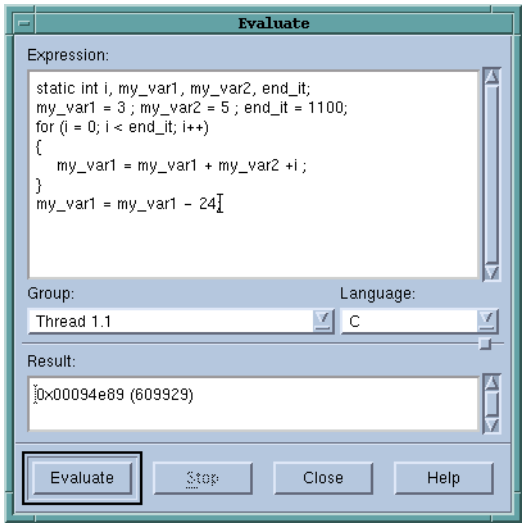
NOTE: Although the CLI does not have an evaluate command, the information in the following sections does apply to the expression argument of the dbreak, dbarrier, dprint, and dwatch commands.

To evaluate an expression: Display the **Evaluate** Dialog Box by selecting the **Tools > Evaluate** command.

An **Evaluate** Dialog Box appears. If your program hasn't yet been created, you won't be able to use any of the program's variables or call any of its functions.

1. Select a button for the programming language you're writing the expression in (if it isn't already selected).
2. Move to the **Expression** field and enter a code fragment. For a description of the supported language constructs, see Using Built-in Variables and Statements on page 378.

   Below is a sample expression. The last statement in this example assigns the value of **my_var1-3** back to **my_var1**. Because this is the last statement in the code fragment, the value placed in the **Result** field is the same as if you had just typed **my_var1-3**.

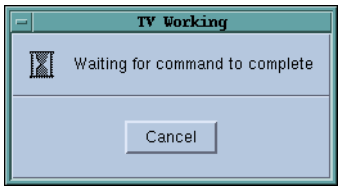


3. Click the **Evaluate** button.

   If TotalView finds an error, it places the cursor on the incorrect line and displays an error message. Otherwise, it interprets (or on some platforms, compiles and executes) the code, and displays the value of the last expression in the **Result** field.

   While the code is being executed, you can't modify anything in the dialog box. TotalView might also display a message box that tells you that it is waiting for the command to complete, Figure 190.

Figure 190, Waiting to Complete Message Box



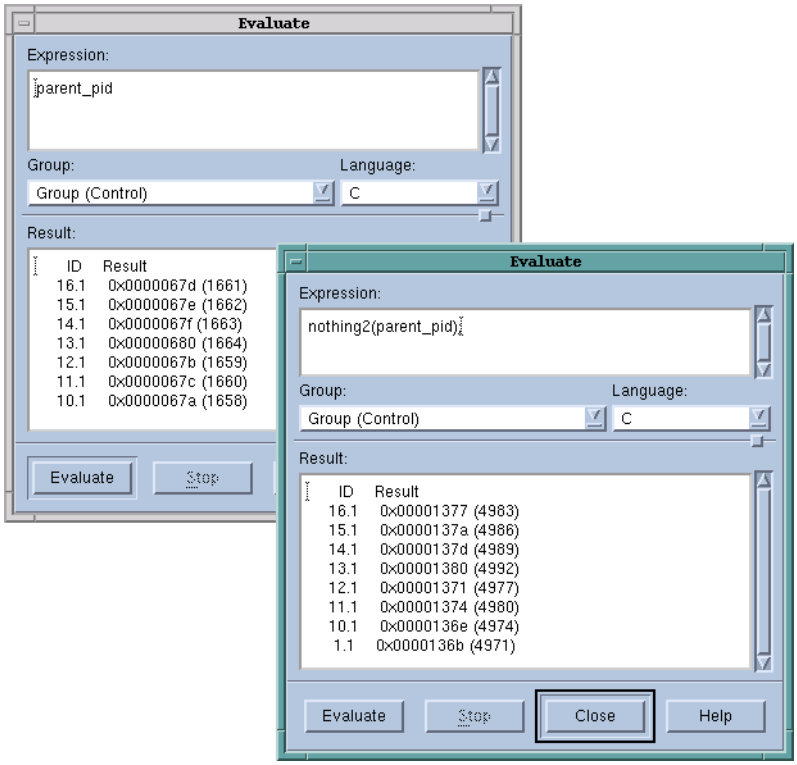
   If you click **Cancel**, TotalView stops execution.

Since TotalView evaluates code fragments in the context of the target process, it evaluates stack variables according to the current program counter. If you declare a variable, its scope is the block that contains the program counter unless, for example, you declare it in some other scope or declare it to be a static variable.

If the fragment reaches a breakpoint (or stops for any other reason), TotalView stops evaluating your expression. Assignment statements in an expression can affect the target process because they can change a variable's value.

The controls at the top of the dialog box let you refine the scope at which TotalView evaluates the information you enter. For example, you can evaluate a function in more than one process. The following figure shows TotalView displaying the value of a variable in multiple processes, and then sending the value as it exists in each process to a function that runs on each of these processes.

Figure 191, Evaluating Information in Multiple Processes



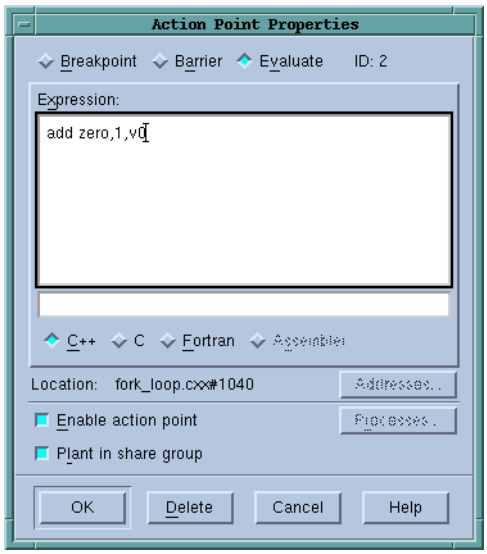
See Group, Process, and Thread Control on page 571 for information on using the P/T set controls at the top of this window.

Writing Assembler Code

On the RS/6000 IBM AIX operating system, TotalView lets you use assembler code in eval points, conditional breakpoints, and in the **Tools > Evaluate** Dialog Box. However, if you want to use assembler constructs, you must enable compiled expressions. See About Interpreted and Compiled Expressions on page 226 for instructions.

To indicate that an expression in the breakpoint or **Evaluate** Dialog Box is an assembler expression, click the **Assembler** button in the **Action Point > Properties** Dialog Box.

Figure 192, Using Assembler Expressions



You write assembler expressions in the target machine's native assembler language and in a TotalView assembler language. However, the operators available to construct expressions in instruction operands, and the set of available pseudo-operators, are the same on all machines, and are described below.

The TotalView assembler accepts instructions using the same mnemonics recognized by the native assembler, and it recognizes the same names for registers that native assemblers recognize.

Some architectures provide extended mnemonics that do not correspond exactly with machine instructions and which represent important, special cases of instructions, or provide for assembling short, commonly used sequences of instructions. The TotalView assembler recognizes mnemonics if:

- They assemble to exactly one instruction.
- The relationship between the operands of the extended mnemonics and the fields in the assembled instruction code is a simple one-to-one correspondence.

Assembler language labels are indicated as *name*: and appear at the beginning of a line. You can place a label alone on a line. The symbols you can use include labels defined in the assembler expression and all program symbols.

The TotalView assembler operators are described in the following table:

Operators	Description
+	Plus
-	Minus (also unary)
*****	Multiplication
#	Remainder
/	Division
&	Bitwise AND
^	Bitwise XOR
!	Bitwise OR NOT (also unary minus, bitwise NOT)
|	Bitwise OR
(*expr*)	Grouping
<<	Left shift
>>	Right shift
"*text*"	Text string, 1-4 characters long, is right-justified in a 32-bit word
hi16 (*expr*)	Low 16 bits of operand ***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:

Pseudo Ops	Description
$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 the process

Pseudo Ops	Description
$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
$ptree	Internal debugging option. Print assembler tree
$stop **$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:
	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 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

Pseudo Ops	Description
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

Using Built-in Variables and Statements

TotalView contains a number of built-in variables and statements that can simplify your debugging activities. You can use these variables and statements in eval points and in the **Tools > Evaluate** Dialog Box.

Topics in this section are:

RELATED TOPICS	
Creating an eval or conditional breakpoint	Defining Eval Points and Conditional Breakpoints on page 220
How to use watchpoints	Using Watchpoints on page 231

Using TotalView Variables

TotalView variables that let you access special thread and process values. All variables are 32-bit integers, which is an **int** or a **long** on most platforms. The following table describes built-in variables:

Name	Returns
$clid	The cluster ID. (Interpreted expressions only.)
$duid	The TotalView-assigned Debugger Unique ID (DUID). (Interpreted expressions only.)
$newval	The value just assigned to a watched memory location. (Watchpoints only.)
$nid	The node ID. (Interpreted expressions only.)
$oldval	The value that existed in a watched memory location before a new value modified it. (Watchpoints only.)
$pid	The process ID.
$processduid	The DUID (debugger ID) of the process. (Interpreted expressions only.)
$systid	The thread ID assigned by the operating system. When this is referenced from a process, TotalView throws an error.
$tid	The thread ID assigned by TotalView. When this is referenced from a process, TotalView throws an error.

The built-in variables let you create thread-specific breakpoints from the expression system. For example, the **$tid** variable and the **$stop** built-in function let you create a thread-specific breakpoint, as the following code shows:

```
if ($tid == 3)
   $stop;
```

This tells TotalView to stop the process only when the third thread evaluates the expression.

You can also create complex expressions using these variables; for example:

```
if ($pid != 34 && $tid > 7)
   printf ("Hello from %d.%d\n", $pid, $tid);
```

Using any of the following variables means that the eval point is interpreted instead of compiled: **$clid**, **$duid**, **$nid**, **$processduid**, **$systid**, **$tid**, and **$visualize**. In addition, **$pid** forces interpretation on AIX.

You can't assign a value to a built-in variable or obtain its address.

Using Built-In Statements

TotalView statements help you control your interactions in certain circumstances. These statements are available in all languages, and are described in the following table. The most commonly used statements are **$count**, **$stop**, and **$visualize**.

Statement	Use
$count *expression* **$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***.

Statement	Use
$countthread *expression*	Sets a thread-level countdown breakpoint. When any thread in a process executes this statement for the number of times specified by *expression*, the thread stops. Other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as **$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 does 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. See Holding and Releasing Processes and Threads. If all other processes in the group are already held at this eval point, TotalView releases all of them. If other processes in the group are running, they continue to run.
$holdstopall **$holdprocessstopall**	Like **$hold**, except that any processes in the group which are running are ***stopped***. 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. See Holding and Releasing Processes and Threads.
$holdthreadstop **$holdthreadstopprocess**	Like **$holdthread,** except that it *stops* the process. The other processes in the group are left running.
$holdthreadstopall	Like **$holdthreadstop,** except that 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.

Statement	Use
$stopthread	Sets a thread-level breakpoint. Although the thread that executes this statement stops, all other threads in the process continue to execute. If the target system cannot stop an individual thread, this statement performs the same as to **$stopprocess**.
$visualize(*expression*[,*slice*])	Visualizes the data specified by ***expression*** and modified by the optional ***slice*** value. ***Expression*** and ***slice*** must be expressed using the code fragment's language. The *expression* must return a dataset (after modification by ***slice***) that can be visualized. ***slice*** is a quoted string that contains a slice expression. For more information on using **$visualize** in an expression, see Using the Visualizer on page 344.

Expression Evaluation with ReplayEngine

When you enable ReplayEngine, you still have the ability to evaluate expressions, but the behavior is different. In regular, forward debugging, your expression may change the state of your program, for example, by changing the value of a variable. But when ReplayEngine is enabled, expression evaluation takes place in a separate, temporary space, and the results have no side effects in your program. When the evaluation is complete, the temporary space is released and any changes resulting from the evaluation no longer exist.

This is important to remember if you are actually counting on an expression evaluation to change something in your program. Note, too, that this is true even when ReplayEngine is in Record mode. If you want to regain the ability to affect your program state through expressions, you need to disable ReplayEngine.

With ReplayEngine enabled and in Record mode, there are still two ways to change memory or registers: with the CLI `dassign` command, and by directly editing the value in the TotalView user interface. However, an attempt to modify memory or registers in this way in Replay mode results in an error or the new value being discarded.

Expressions can call functions when ReplayEngine is enabled, but if the called function stops for any reason, for example, hits a breakpoint or receives a signal, the expression is suspended and limitations are imposed. You can continue to debug forward in a function called from an expression, but you cannot debug backwards until the expression evaluation is complete. Using an expression to write to stdout and stderr (file descriptors 1 and 2) is allowed with the following limitations: Writes to those file descriptors work for any type of file in Record mode. However, writes to those file descriptors fail in Playback mode unless the file is a TTY.

All of the above also applies to the transformations in C++View.

About Groups, Processes, and Threads

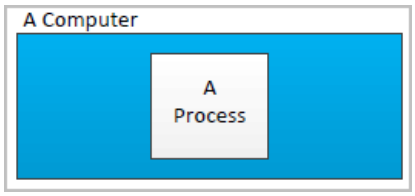
While the specifics of how multi-process, multi-threaded programs execute differ greatly between hardware platforms, operating systems, and compilers, all share some general characteristics. This chapter defines a general model for conceptualizing the way processes and threads execute and introduces the concepts of *threads*, *processes*, and *groups*. Group, Process, and Thread Control on page 571 is a more exacting and comprehensive look at these topics.

This chapter contains the following sections:

A Couple of Processes

When programmers write single-threaded, single-process programs, they can almost always answer the question "Do you know where your program is?" These types of programs are rather simple, looking something like Figure 193.

Figure 193, A Uniprocessor



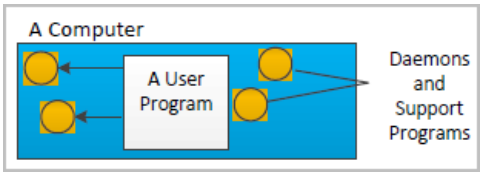
If you use any debugger on these types of programs, you can almost always figure out what's going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, and then inspect variables to see their values. If you suspect that there's a logic problem,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

What is actually occurring, however, is a lot more complicated, since other 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.

These additional processes can simplify your program because it can hand off some tasks and not have to focus on how that work gets done.

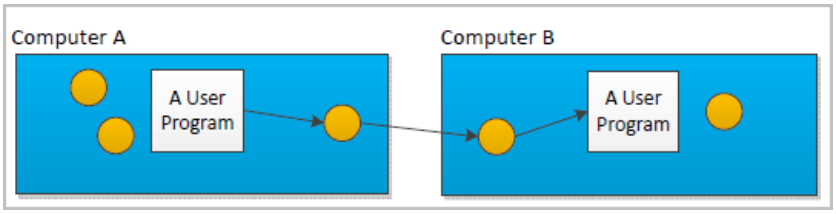
Figure 194 shows a very simple architecture in which the application program just sends requests to a daemon.

Figure 194, A Program and Daemons



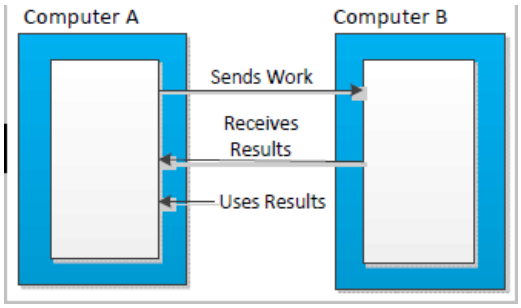
The type of architecture shown in Figure 195 is more typical. In this example, an email program communicates with a daemon on one computer. After receiving a request, this daemon sends data to an email daemon on another computer, which then delivers the data to another mail program.

Figure 195, Mail Using Daemons to Communicate



This architecture has one program handing off work to another. After the handoff, the programs do not interact. The program handing off the work just assumes that the work gets done. Some programs can work well like this. Most don't. Most computational jobs do better with a model that allows a program to divide its work into smaller jobs, and parcel this work to other computers. Said in a different way, this model has other machines do some of the first program's work. To gain any advantage, however, the work a program parcels out must be work that it doesn't need right away. In this model, the two computers act more or less independently. And, because the first computer doesn't have to do all the work, the program can complete its work faster.

Figure 196, Two Computers Working on One Problem



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

The TotalView Server Solution to Debugging Across Computers

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

NOTE: TotalView can also attach to programs 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 initially write your program so that it only uses one computer. After it is working, you can split up its work so that it uses other computers. It is likely that any problems you find will occur in the code that splits 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.

NOTE: Initially designing a multi-process application as a single-process program may not always be practical. For instance, some algorithms may take weeks to execute a program on one computer.

RELATED TOPICS

How TotalView organizes groups, processes, and threads	Group, Process, and Thread Control on page 571
Debugging remotely	Setting Up Remote Debugging Sessions on page 484
Attaching to a running program	Attaching to a Running Program on page 105

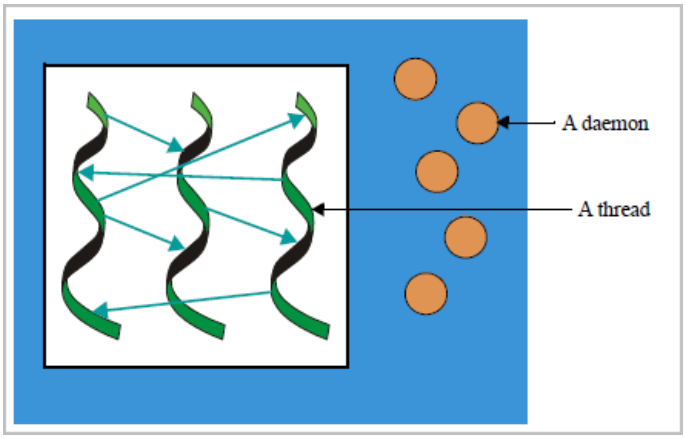
Threads

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

While operating systems can have many independently executing components, a program can as well, accomplished in various ways. One programming model splits the work off into somewhat independent tasks within the same process. This is the *threads* model.

Figure 197 also shows the daemon processes that are executing. (The figures in the rest of this chapter won't show these daemons.)

Figure 197, 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 issue here is similar to the problem of processes running on different machines. In both, a debugger must intervene with more than one executing entity, having to understand multiple address spaces and multiple contexts.

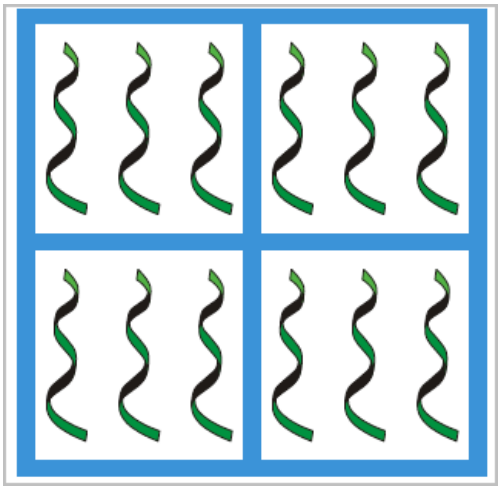
NOTE: Little difference exists between a multi-threaded or a multi-process program when using TotalView. The way in which TotalView displays process information is very similar to how it displays thread information.

RELATED TOPICS	
TotalView's design on organizing groups, processes, and threads	Group, Process, and Thread Control on page 571
Debugging multi-threaded, multi-process programs	Manipulating Processes and Threads on page 407
Setting breakpoints	Setting Breakpoints for Multiple Processes on page 211 Setting Breakpoints When Using the fork()/execve() Functions on page 213
Barrier points in multi-threaded programs	Setting Barrier Points on page 215

Complicated Programming Models

While most computers have one or two processors, high-performance computing often uses computers with many more. And as hardware prices decrease, this model is starting to become more widespread. Having more than one processor means that the threads model in Figure 197 changes to something similar to that shown in Figure 198.

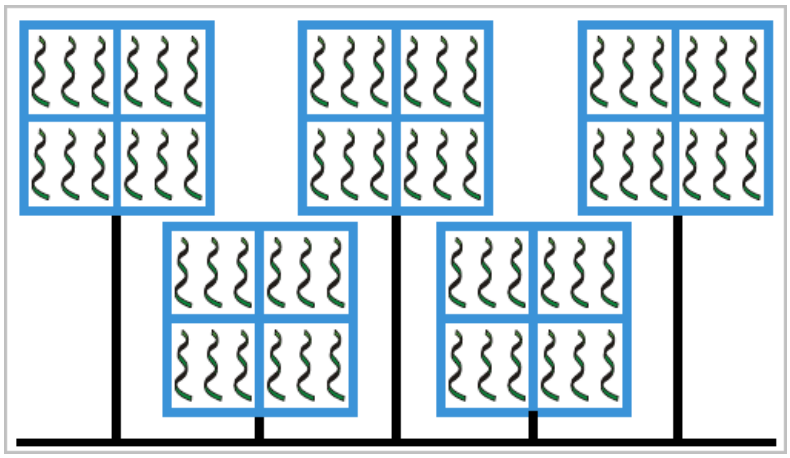
Figure 198, Four-Processor Computer



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

The next step is to join many multi-processor computers together. Figure 199 shows five computers, each with four processors, with each processsor running three threads. If this figure shows the execution of one program, then the program is using 60 threads.

Figure 199, Four Processors on a Network



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

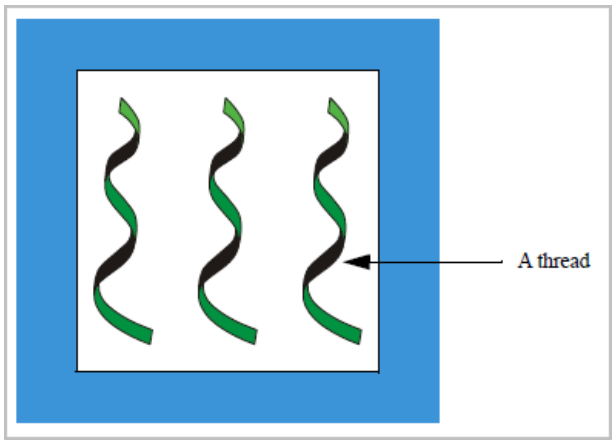
At any time, it is next to impossible to guess which threads are executing and what a thread is actually doing. Even more complex, many multi-processor programs begin by invoking a process such as **mpirun** or IBM **poe**, whose function is to distribute and control the work being performed. In this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

In this model, traditional debuggers and solutions don't work. TotalView, on the other hand, organizes this mass of executing procedures for you, distinguishing between threads and processes that the operating system uses from those that your program uses.

Types of Threads

All threads aren't the same. Figure 200 shows a program with three threads.

Figure 200, Threads (again)



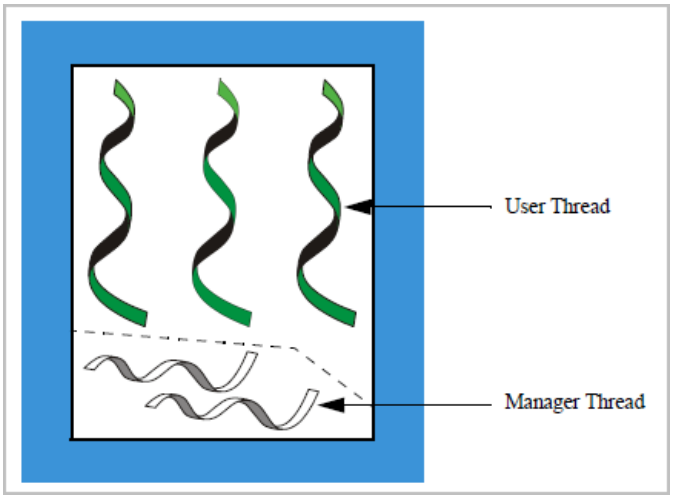
Assume that all these threads are *user threads*; that is, they are threads that perform some activity that you've programmed.

NOTE: Many computer architectures have something called user mode, user space, or something similar. In TotalView, the definition of a user thread is simply a unit of execution created by a program.

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

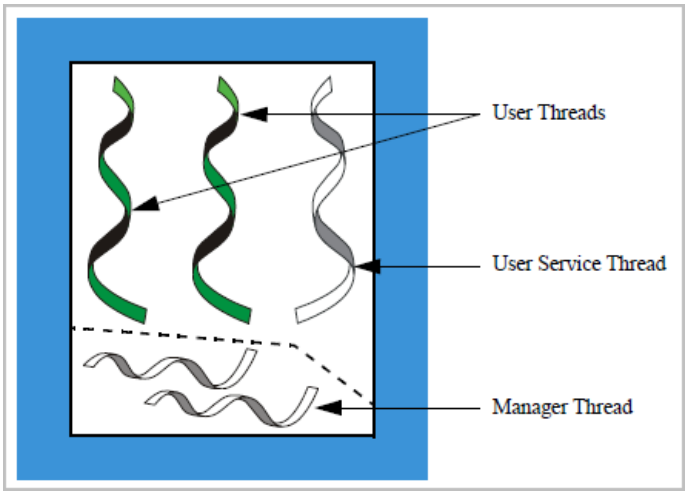
Other threads can also be executing. For example, there are always threads that are part of the operating environment. These threads are called *manager threads*. Manager threads exist to help your program get its work done. In Figure 201, the horizontal threads at the bottom are user-created manager threads.

Figure 201, User and Service Threads



All threads are not created equal, and all threads do not execute equally. Many programs also create manager-like threads. Since these user-created manager threads perform services for other threads, they are called *service threads*, Figure 202.

Figure 202, User, Service, and Manager Threads



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

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

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

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

Although Figure 202 shows five threads, most of your debugging effort will focus on just two threads.

RELATED TOPICS

TotalView's design on organizing groups, processes, and threads	Group, Process, and Thread Control on page 571
Setting the focus	Setting Process and Thread Focus on page 579 and Setting Group Focus on page 585

Organizing Chaos

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

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

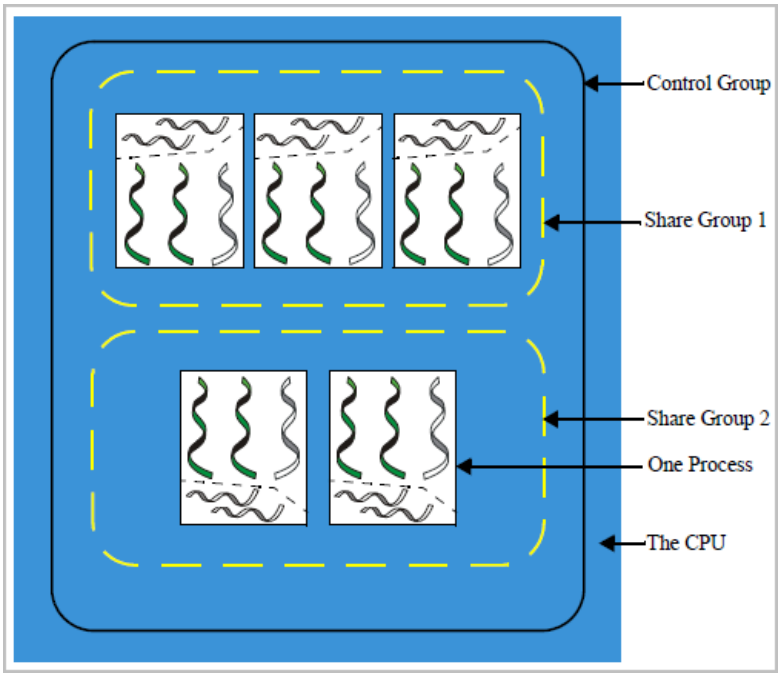
- **Control Group**: All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program that has two distinct executables that run independently of one another has each executable in a separate control group. In contrast, processes created by **fork()/exec()** are in the same control group.
- **Share Group**: All the processes within a control group that share the same code. *Same code* means that the processes have the same executable file name and path. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.
- **Workers Group**: All the worker threads within a control group. These threads can reside in more than one share group.
- **Lockstep Group**: All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. By definition, all members of a lockstep group are within the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group. A lockstep group only means anything when the threads are stopped.

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

TotalView lets you manipulate processes and threads individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent). For more information, see Group, Process, and Thread Control on page 571.

Figure 203 shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within the processes, along with a control group and two share groups within the control group.

Figure 203, Five-Processes: Their Control and Share Groups



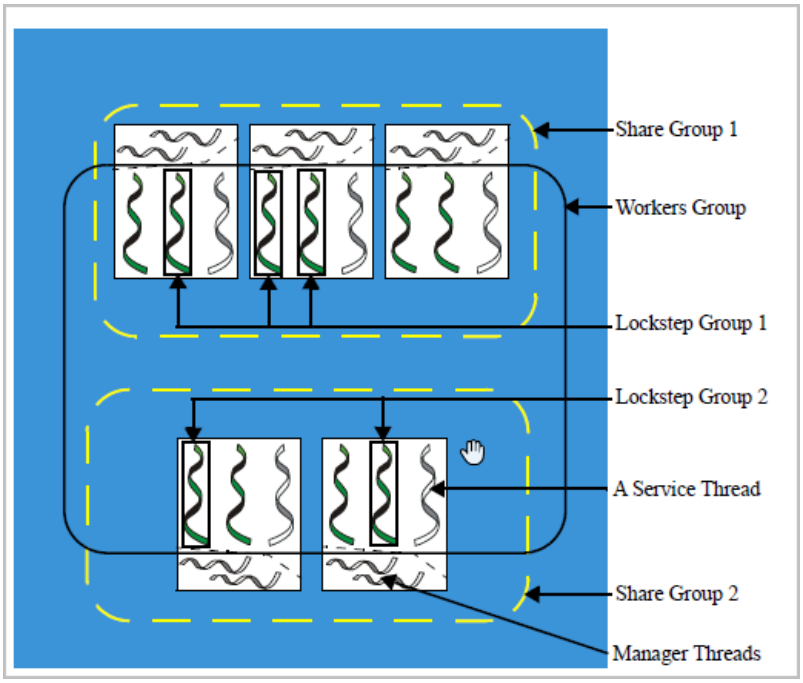
Many of the elements in this figure are used in other figures in this book. These elements are as follows:

CPU	The one outer square represents the CPU. All elements in the drawing operate within one CPU.
Processes	The five white inner squares represent processes being executed by the CPU.
Control Group	The large rounded rectangle that surrounds the five processes shows one control group. This diagram doesn't indicate which process is the main procedure.
Share Groups	The two smaller rounded rectangles having yellow 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 contain only processes.

Figure 204 shows how TotalView organizes the threads in the previous figure, adding a workers group and two lockstep groups.

Figure 204, Five Processes: Adding Workers and Lockstep Groups



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

The additional elements in this figure are as follows:

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

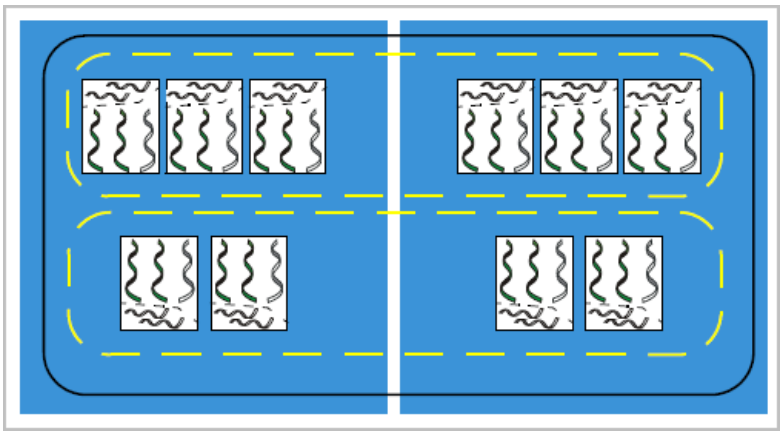
Lockstep Groups Each share group has its own lockstep group. The previous figure shows two lockstep groups, one in each share group.

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

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

Figure 205 extends Figure 204 to show the same kinds of information executing on two processors.

Figure 205, Five Processes and Their Groups on Two Computers



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

RELATED TOPICS	
TotalView's design on organizing groups, processes, and threads	Group, Process, and Thread Control on page 571
Setting the focus	Setting Process and Thread Focus on page 579 and Setting Group Focus on page 585

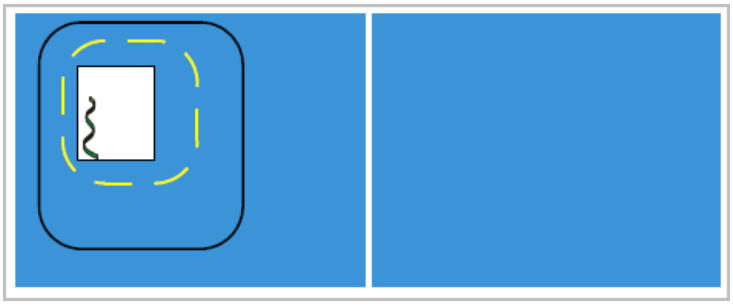
How TotalView Creates Groups

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

Step 1

TotalView and your program are launched, and your program begins executing.

Figure 206, Step 1: A Program Starts

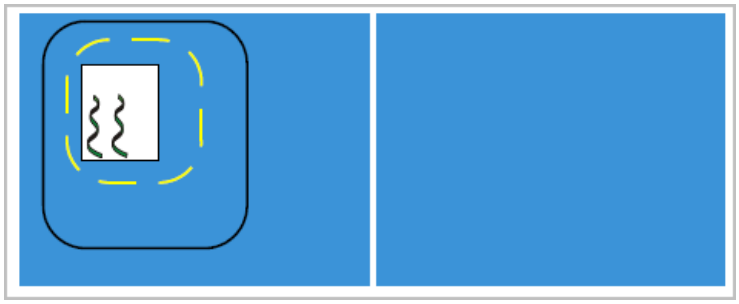


- **Control group**: The program is loaded and creates a group.
- **Share group**: The program begins executing and creates a group.
- **Workers group**: The thread in the **main()** routine is the workers group.
- **Lockstep group**: There is no lockstep group because the thread is running. (Lockstep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

Step 2

The program creates a thread.

Figure 207, Step 2: A Thread is Started

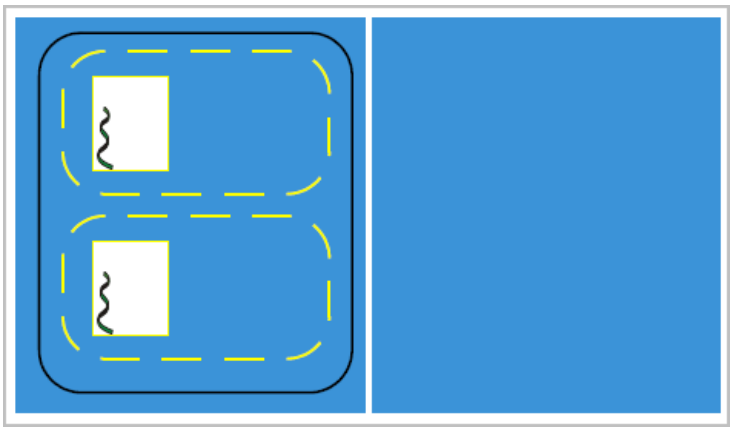


- **Control group**: The control group is unchanged.
- **Share group**: The share group is unchanged.
- **Workers group**: TotalView adds the thread to the existing group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 3

The first process uses the **exec()** function to create a second process, Figure 208.

Figure 208, Step 3: Creating a Process using exec()



- **Control group**: The group is unchanged.

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

Step 4

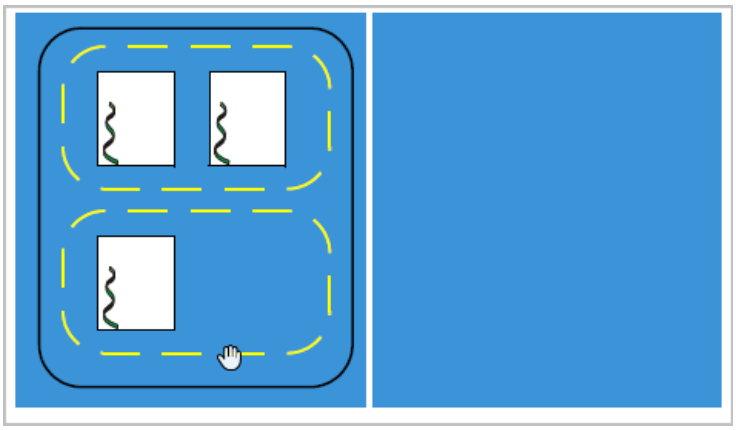
The first process hits a breakpoint.

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

Step 5

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

Figure 209, Step 5: Creating a Second Version

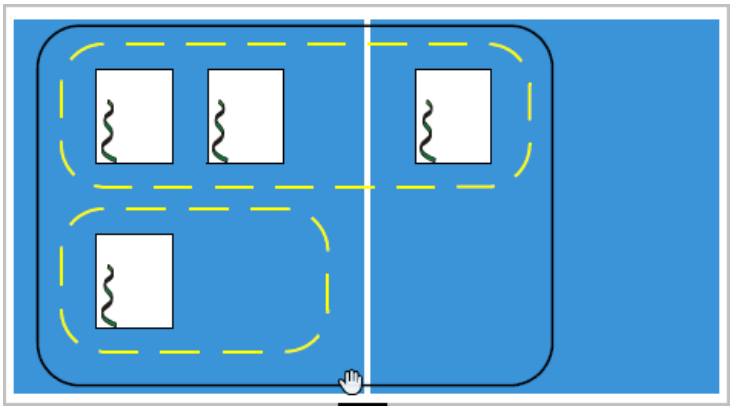


- **Control group**: TotalView adds a third process.
- **Share group**: TotalView adds this third process to the first share group.
- **Workers group**: TotalView adds the thread in the third process to the group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 6

Your program creates a process on another computer.

Figure 210, Step 6: Creating a Remote Process

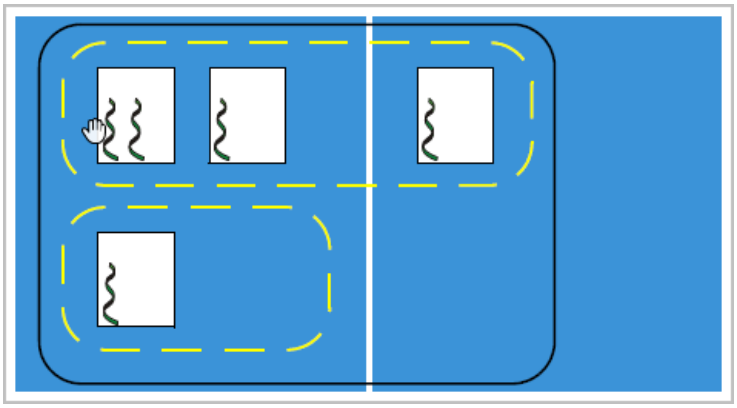


- **Control group**: TotalView extends the control group so that it contains the fourth process, which is running on the second computer.
- **Share group**: The first share group now contains this newly created process, even though it is running on the second computer.
- **Workers group**: TotalView adds the thread within this fourth process to the workers group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 7

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

Figure 211, Step 7: Creating a Thread

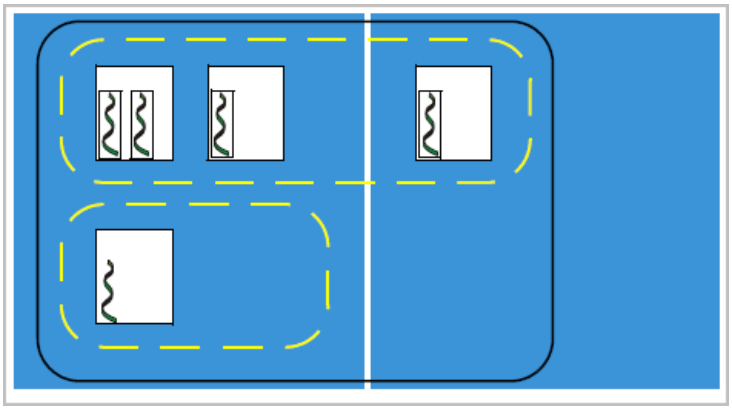


- **Control group**: The group is unchanged.
- **Share group**: The group is unchanged.
- **Workers group**: TotalView adds a fifth thread to this group.
- **Lockstep group**: There are no lockstep groups because the threads are running.

Step 8

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

Figure 212, Step 8: Hitting a Breakpoint

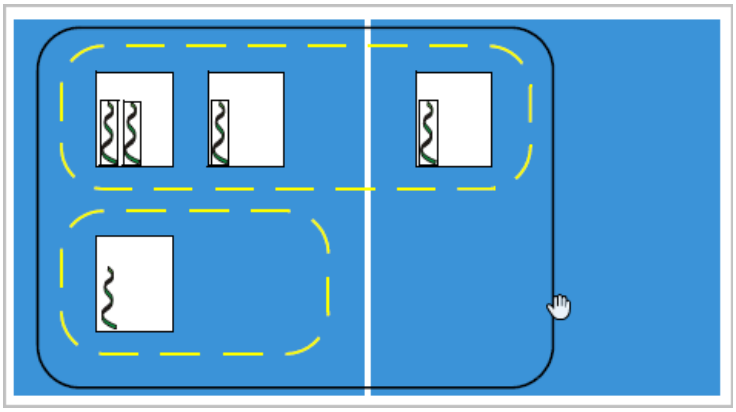


- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.
- **Workers group**: The group is unchanged.
- **Lockstep groups**: TotalView creates a lockstep group whose members are the four threads in the first share group.

Step 9

You tell TotalView to step the lockstep group.

Figure 213, Step 9: Stepping the Lockstep Group



- **Control group**: The group is unchanged.
- **Share group**: The groups are unchanged.
- **Workers group**: The group is unchanged.
- **Lockstep group**: The lockstep groups are unchanged. (There are other lockstep groups as explained in Group, Process, and Thread Control on page 571.)

What Comes Next

This example could continue to create a more complicated system of processes and threads. However, adding more processes and threads would not change the described behavior.

Simplifying What You're Debugging

The reason you're using a debugger is because your program isn't operating correctly, and the method you think will solve the problem is to stop your program's threads, examine the values assigned to variables, and step your program so you can observe execution.

Unfortunately, your multi-process, multi-threaded program and the computers upon which it executes are running several threads or processes that you want TotalView to ignore. For example, you don't want to examine 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, only a few problems 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 the program is running under TotalView control, run the process being debugged to an action point so that you can inspect the program's state at that point. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing.

NOTE: TotalView lets you control as many groups, processes, and threads as you need to control. Although you can control each one individually, it would be very complicated to try to control large numbers of these independently. TotalView creates and manages groups so that you can focus on portions of your program.

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

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

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

   CLI: dgroups -remove

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

> **NOTE:** Don't step your program unless you need to individually look at a thread. Using barrier points is much more efficient. Barrier points are discussed in Setting Barrier Points on page 215.

3. After execution stops at a barrier point, look at the contents of your variables. Verify that your program state is actually correct.

4. Begin stepping your program through its code. In most cases, step your program synchronously or set barriers so that everything isn't running freely.

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

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

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

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

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

  ```
  dfocus t3.7 dstep
  ```

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

RELATED TOPICS	
Detailed information on TotalView threads, processes, and groups	Group, Process, and Thread Control on page 571
Solving problems when starting MPI applications	Starting MPI Issues on page 542
Setting barrier points	Setting Barrier Points on page 215
More specific debugging tips for parallel applications	Debugging Strategies for Parallel Applications on page 437

Manipulating Processes and Threads

This chapter illustrates some foundational parallel debugging tasks and is based on the shipped program, **wave_extended_threads**, located in the directory *installdir*/**toolworks**/*totalview.version/platform*/**examples**. This is a simple program that creates an array and then increments its values to simulate a wave form which can then be viewed using the Visualizer. The program requires user input to provide the number of times to increment.

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

- Use the **-g** option to compile the program. (Compiling is not discussed here. Please see Compiling Programs on page 87.)
- Start the program under TotalView control.
- Start the debugging process, including setting breakpoints and examining your program's data.

When working with multi-process, multi-threaded programs, you have many options for controlling thread and process execution, viewing specific threads and processes, and organizing processes in to groups in order to better view the various elements of your program.

This chapter includes:

Viewing Process and Thread States

Process and thread states are displayed in the following:

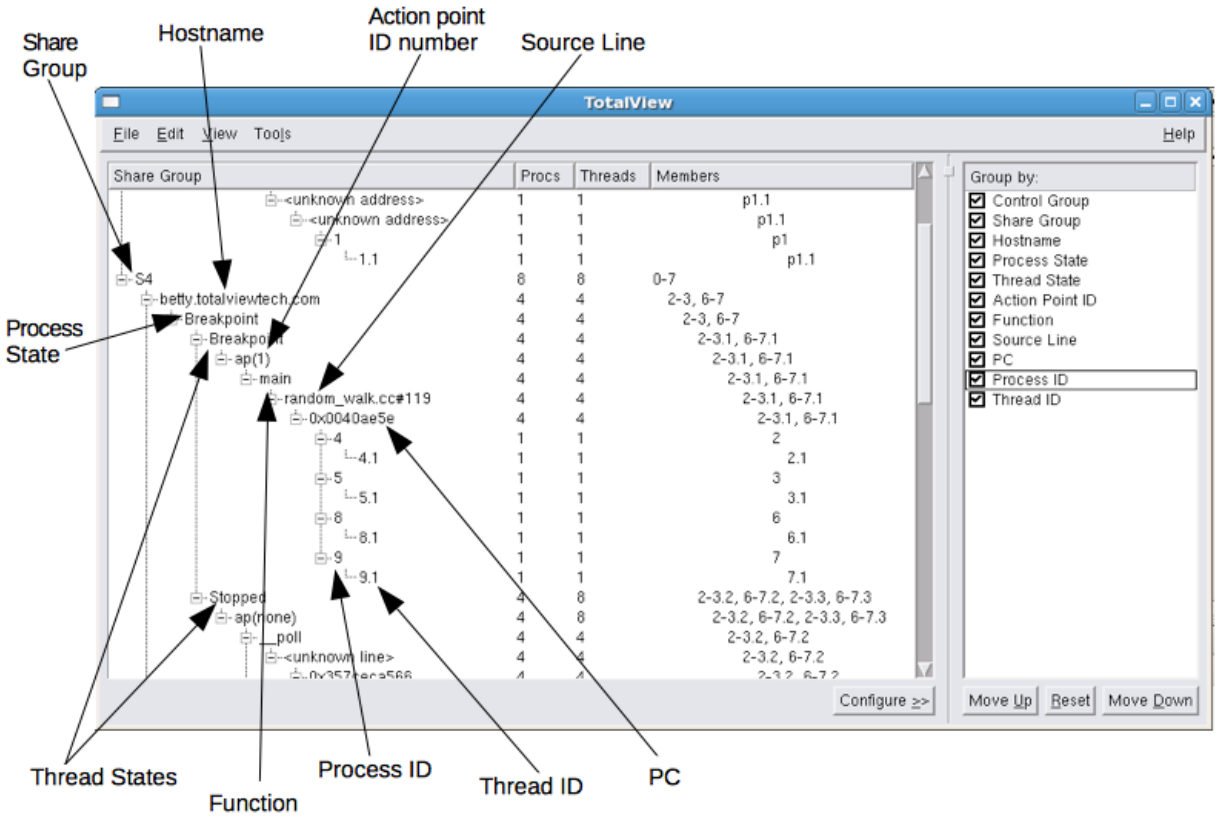
- The Root Window.
- The information within the **File > Attach to a Running Program** dialog.
- The process and thread status bars of the Process Window.
- The Threads tab of the Process Window.

Figure 214 shows TotalView displaying process state information in the Root Window.

CLI: dstatus and **dptsets**

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

Figure 214, Root Window Showing Process and Thread Status



The **Status** of a process includes the process location, the process ID, and the state of the process. (These characters are explained in Seeing Attached Process States on page 411.)

If you need to attach to a process that is not yet being debugged, open the **File > Attach to a Running Program** dialog. TotalView displays all processes associated with your username. Notice that some of the processes will be dim (drawn in a lighter font). This indicates either you cannot attach to the process or you're already attached to it.

Notice that the status bars in the Process Window also display status information, Figure 215.

Figure 215, Process and Thread Labels in the Process Window

NOTE: TotalView displays both the user thread id (if one exists) assigned by the **pthread** runtime, as well as the kernel thread id assigned by the operating system (user_tid/kernel_tid). If you are debugging an MPI program, TotalView displays the thread's rank number.

RELATED TOPICS	
The Root Window	Using the Root Window on page 147
The Process Window	Using the Process Window on page 157
Process state definition and display	Seeing Attached Process States on page 411

Seeing Attached Process States

The Root Window displays the Process and/or Thread State. To view these states, enable the "Process State" and/or "Thread State" check boxes in the Configure pane. Once enabled, the states display in the first column.

.

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

Seeing Unattached Process States

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

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

Displaying a Thread Name

In complex, multi-threaded programs with perhaps thousands of threads, it may be useful to name certain threads, for instance, if particular threads are dedicated to performing special functions. This can be helpful when sorting or identifying threads in your programs.

If you set a thread name in your program, the name is displayed in the TotalView UI:

- In the Root Window (if Thread Name is selected in the **Group By** sidebar)
- In the Process Window's thread status bar
- In the Threads Pane Tab

To display a thread name in the TotalView UI, first set the name in your program using the `pthread_setname_np()` method.

For example:

```
int rc = pthread_setname_np(thread, "MyThreadName");
```

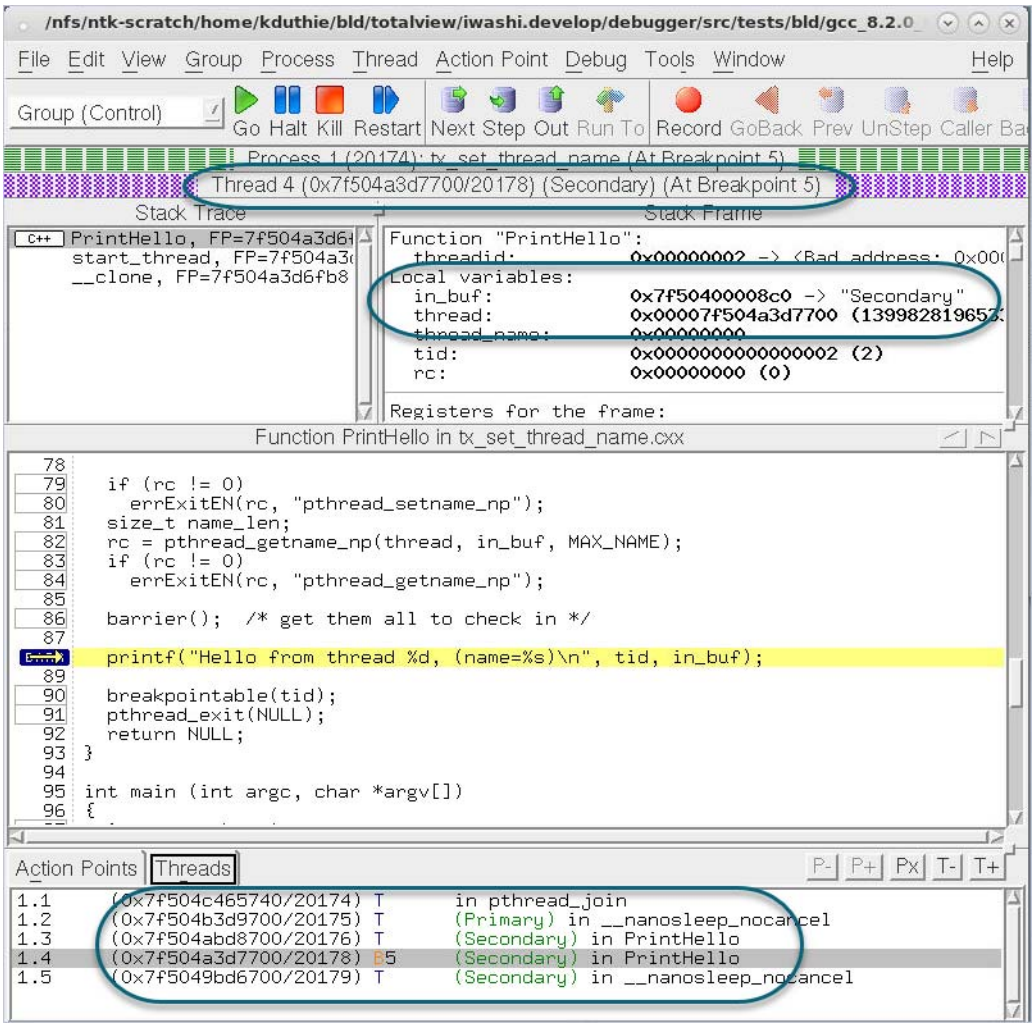
Unless explicitly set by the program, threads are not named.

RELATED TOPICS	
TV::thread properties	**TV::thread** in the *TotalView Reference Guide*
Thread names displayed in the UI	Thread Names in the UI on page 412
dstatus options relating to thread names	**dstatus** in the *TotalView Reference Guide*

Thread Names in the UI

When set, thread names are displayed in both the Process Window and the Root Window.

Thread names in the Process Window



This program sets these thread names:

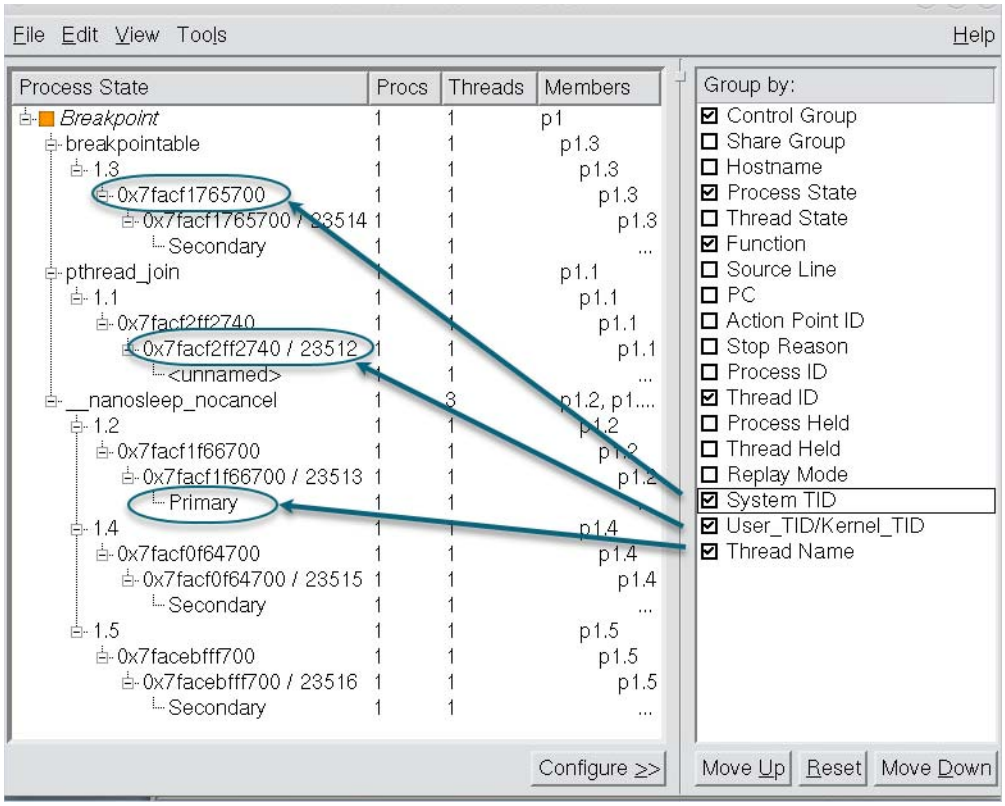
```
pthread_mutex_lock(&name_mutex);
  int rc;
  if (!first_in)
    {
      first_in = true;
      rc = pthread_setname_np(thread, "Primary");
    }
  else
    rc = pthread_setname_np(thread, "Secondary");
```

The first thread to enter a function is named “Primary”, and all subsequent threads are called “Secondary.”

Note that the thread names are displayed in:

1. The Thread status bar at the top, which also displays the **utid / ktid** ("thread user ID" / "thread kernel ID")
2. The Stack Frame view, if relevant
3. The Threads View tab, again also displaying the **utid / ktid**.

Thread names in the Root Window



Thread Properties

TV::thread includes these properties relevant to thread naming:

- **thread_name**: The name given to a thread by the application.
- **thread_ktid**: The kernel thread id)
- **thread_utid**: User thread ID (**pthread_t**)

These properties are read-only, so can be accessed but not set.

Thread Options on dstatus

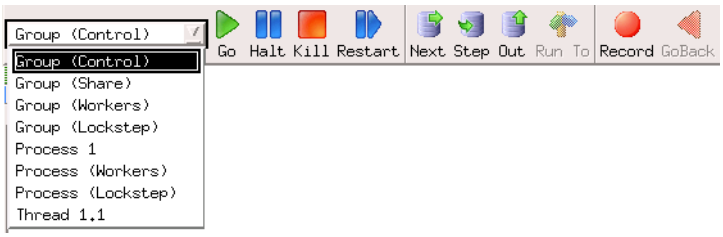
The command **dstatus** has options and properties related to thread names:

- **-thread_name**: Displays any thread names in a program, if they exist
- Properties on the **-group_by** option:
  - **systid**: Either the user thread ID (**utid**) or the kernel thread ID (**ktid**) if no **utid** exists
  - **utid_ktid**: "**utid / ktid**" or just **ktid** if no **utid** exists.
  - **tname**: thread name or "<unnamed>" if no thread name exists.

Using the Toolbar to Select a Target

The Process Window toolbar has a dropdown list that controls process and thread focus. The selection in this dropdown list defines the *focus*, or target of the toolbar commands. (The selected target in this pulldown is also called a *scope modifier*.)

Figure 216, The Toolbar



For example, if you select a thread and then select **Step**, TotalView steps the current thread. If **Process (workers)** is selected and you select **Halt**, TotalView halts all processes associated with the current thread's workers group. If you are running a multi-process program, other processes continue to execute.

In a multi-process, multi-threaded program, this is important, as TotalView needs to know which processes and threads to act on.

In the CLI, specify this target using the **dfocus** command.

NOTE: Group, Process, and Thread Control on page 571 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.

RELATED TOPICS	
The Processes/Ranks tab in the Process Window	Using the Processes/Ranks and Threads Tabs on page 418
How to create custom groups	Creating Custom Groups on page 602

Stopping Processes and Threads

To stop a group, process, or thread, select a **Halt** command from the **Group**, **Process**, or **Thread** pulldown menus in the menubar.

CLI: dhalt
Halts a group, process, or thread. Setting the focus changes the scope.

The three **Halt** commands differ in the scope of what they halt. In all cases, TotalView uses the current thread, which is called the thread of interest or TOI, to determine what else it will halt. For example, selecting **Process > Halt** tells TotalView to determine the process in which the TOI is running. It then halts this process. Similarly, if you select **Group > Halt**, TotalView determines what processes are in the group in which the current thread participates. It then stops all of these processes.

NOTE: For more information on the Thread of Interest, see Defining the GOI, POI, and TOI on page 572.

When you select the **Halt** button in the toolbar instead of the commands in the menubar, TotalView decides what it should stop based on what is set in the toolbar pulldown list.

After entering a **Halt** command, TotalView updates any windows that can be updated. When you restart the process, execution continues from the point where TotalView stopped the process.

Using the Processes/Ranks and Threads Tabs

The Processes Tab

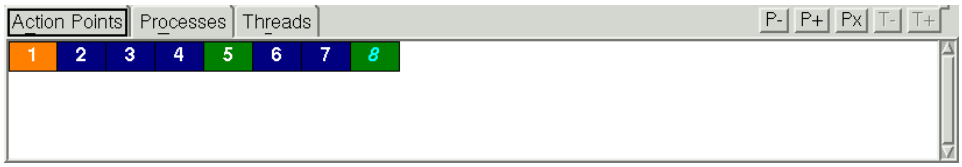
The Processes Tab was displayed by default in previous versions of TotalView, but now it is off by default. This is because it can significantly affect performance, particularly for large, massively parallel applications. The tab can be turned back on with the command line switch **-processgrid** and/or by setting **TV::GUI::process_grid_wanted** to **true** in the `.tvdrc` file. If you enable this tab in the `.tvdrc` file, you can disable it for a particular session with the **-noprocessgrid** command line switch.

The Processes tab, which is called a Ranks tab if you are running an MPI program, contains a grid. Each block in the grid represents one process. The color that TotalView uses to display a process indicates the process's state, as follows:

Color	Meaning
Blue	Stopped; usually due to another process or thread hitting a breakpoint.
Orange	At breakpoint.
Green	All threads in the process are running or can run.
Red	The **Error** state. Signals such as **SIGSEGV**, **SIGBUS**, and **SIGFPE** can indicate an error in your program.
Gray	The process has not begun running.

Figure 217 shows a tab with processes in three different states:

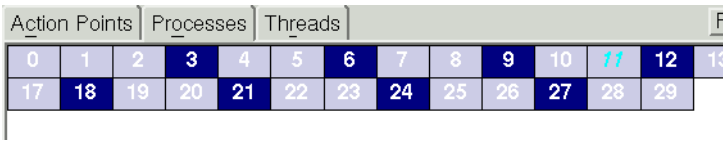
Figure 217, The Processes Tab



If you select a group by using the Process Window's group selector pulldown (see Using the Toolbar to Select a Target on page 416 for information), TotalView dims the blocks for processes not in the group, Figure 218.

CLI: dptsets

Figure 218, The Processes Tab: Showing Group Selection

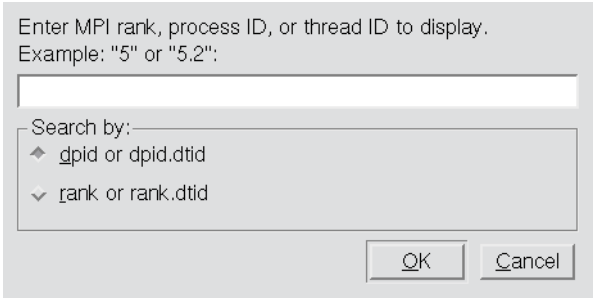


If you click on a block, the context within the Process Window changes to the first thread in that process.

CLI: dfocus

Clicking on the **P+** and **P-** buttons in the tab bar changes the process being displayed within the Process Window. Click on **Px** to launch a **Jump To** dialog in which you can specify a particular process or thread to focus on, Figure 219.

Figure 219, The Jump To Dialog

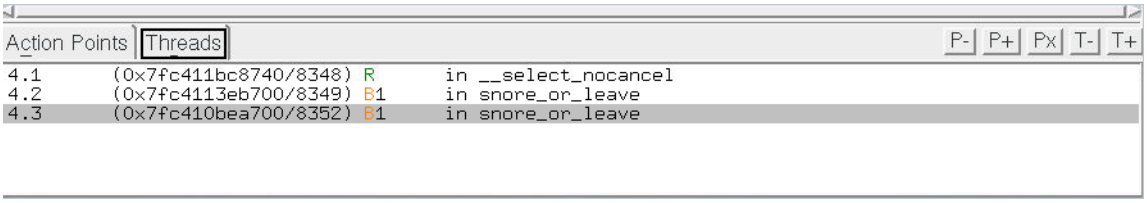


RELATED TOPICS	
Custom group creation	Creating Custom Groups on page 602
More on controlling processes and threads	Using the Toolbar to Select a Target on page 416

The Threads Tab

The Threads Tab displays information about the state of your threads. Clicking on a thread shifts the focus within the Process Window to that thread.

Figure 220, The Threads Tab



Clicking on the **T+** and **T-** buttons in the tab bar also changes the thread being displayed within the Process Window.

RELATED TOPICS

More on the Threads Tab and its display	The Threads Tab in the online Help

Updating Process Information

Normally, TotalView updates information only when the thread being executed stops executing. You can force TotalView to update a window by using the **Window > Update** command. You need to use this command if you want to see what a variable's value is while your program is executing.

NOTE: When you use this command, TotalView momentarily stops execution to obtain update information, then restarts the thread.

Holding and Releasing Processes and Threads

Many times when you are running a multi-process or multi-threaded program, you want to synchronize execution to the same place. You can do this manually using a *hold* command, or automatically by setting a barrier point.

When a process or a thread is *held*, it ignores any command to resume executing. For example, assume that you place a hold on a process in a control group that contains three processes. If you select **Group > Go**, two of the three processes resume executing. The held process ignores the **Go** command.

Use the **Release** command to remove the hold. When you release a process or a thread, it can resume execution, but you still need to tell it to do so. That is, you must resume execution with a command such as **Go**, **Out**, or **Step**.

Manually holding and releasing processes and threads is useful when:

- You need to run a subset of the processes and threads. You can manually hold all but the ones you want to run.
- A process or thread is held at a barrier point and you want to run it without first running all the other processes or threads in the group to that barrier. In this case, you release the process or the thread manually and then run it.

See Setting Barrier Points on page 215 for more information on manually holding and releasing barrier breakpoints.

When TotalView is holding a process, the Root Window displays **Stopped**, and the Process Window displays a held indicator, which is the uppercase letter **H**. When TotalView is holding a thread, it displays a lowercase **h**.

You can hold or release a thread, process, or group of processes in one of the following ways:

- You can hold a group of processes using the **Group > Hold** command.
- You can release a group of processes using the **Group > Release** command.
- You can toggle the hold/release state of a process by selecting and clearing the **Process > Hold** command.
- You can toggle the hold/release state of a thread by selecting and clearing the **Thread > Hold** command.

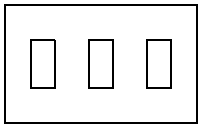
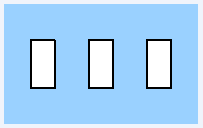
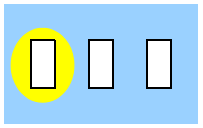
CLI: dhold and dunhold
Setting the focus changes the scope.

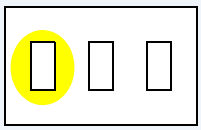
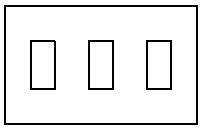
If a process or a thread is running when you use a hold or release command, TotalView stops the process or thread and then holds it. TotalView lets you hold and release processes independently from threads.

The Process pulldown menu contains the commands **Hold Threads** and **Release Threads**, which act on all the threads in a multi-process program. The result is seldom what you actually want as you really do want something to run. You can select one or more threads and use the **Thread > Hold** toggle command to clear them so that TotalView lets them run. This may appear awkward, but it is actually an easy way to run just one or more threads when your program has a lot of threads. You can verify that you're doing the right thing by looking at the thread status in the Root Window.

CLI: dhold -thread
dhold -process
dunhold -thread

Here are some examples of using hold commands:

Held/Release State	What Can Be Run Using Process > Go
	This figure shows a process with three threads. Before you do anything, all threads in the process can be run.
	Select the **Process > Hold** toggle. The blue shading indicates that you held the process. Nothing runs when you select **Process > Go**.
	Go to the **Threads** menu. 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 indicates that thread 1 is held. At this time, there are two different holds on thread 1. One is at the process level; the other is at thread level. Nothing will run when you select **Process > Go**.

Held/Release State	What Can Be Run Using Process > Go
	Select the **Process > Hold** command. Select **Process > Go**. The second and third threads run.
	Select **Process > Release Threads**. This releases the hold placed on the first thread by the **Thread > Hold** command. You could also release the thread individually with **Thread > Hold**. When you select **Process > Go**, all threads run.

RELATED TOPICS

Barrier points	Setting Barrier Points on page 215
The CLI **dbarrier** command	**dbarrie**r in the "CLI Commands" in the *Classic TotalView Reference Guide*

Using Barrier Points

Because threads and processes are often executing different instructions, keeping threads and processes together is difficult. The best strategy is to define places where the program can run freely and places where you need control. This is where barrier points come in.

To keep things simple, this section only discusses multi-process programs. You can do the same types of operations when debugging multi-threaded programs.

Why breakpoints don't work (part 1)

If you set a breakpoint that stops all processes when it is hit and you let your processes run using the **Group > Go** command, you might get lucky and have all of your threads reach the breakpoint together. More likely, though, some processes won't have reached the breakpoint and TotalView will stop them wherever they happen to be. To get your processes synchronized, you would need to find out which ones didn't get there and then individually get them to the breakpoint using the **Process > Go** command. You can't use the **Group > Go** command since this also restarts the processes stopped at the breakpoint.

Why breakpoints don't work (part 2)

If you set the breakpoint's property so that only the process hitting the breakpoint stops, you have a better chance of getting all your processes there. However, you must be careful not to have any other breakpoints between where the program is currently at and the target breakpoint. If processes hit these other breakpoints, you are once again left to run processes individually to the breakpoint.

Why single stepping doesn't work

Single stepping is just too tedious if you have a long way to go to get to your synchronization point, and stepping just won't work if your processes don't execute exactly the same code.

Why barrier points work

If you use a barrier point, you can use the **Group > Go** command as many times as it takes to get all of your processes to the barrier, and you won't have to worry about a process running past the barrier.

The Root Window shows you which processes have hit the barrier, grouping all held processes under **Breakpoint** in the first column.

RELATED TOPICS	
Barrier points	Setting Barrier Points on page 215
The CLI **dbarrier** command	**dbarrie**r in "CLI Commands" in the *Classic TotalView Reference Guide*

Barrier Point Illustration

Creating a barrier point tells TotalView to *hold* a process when it reaches the barrier. Other processes that can reach the barrier but aren't yet at it continue executing. One-by-one, processes reach the barrier and, when they do, TotalView holds them.

When a process is *held*, it ignores commands that tell it to execute. This means, for example, that you can't tell it to go or to step. If, for some reason, you want the process to execute, you can manually release it using either the **Group > Release** or **Process > Release Threads** command.

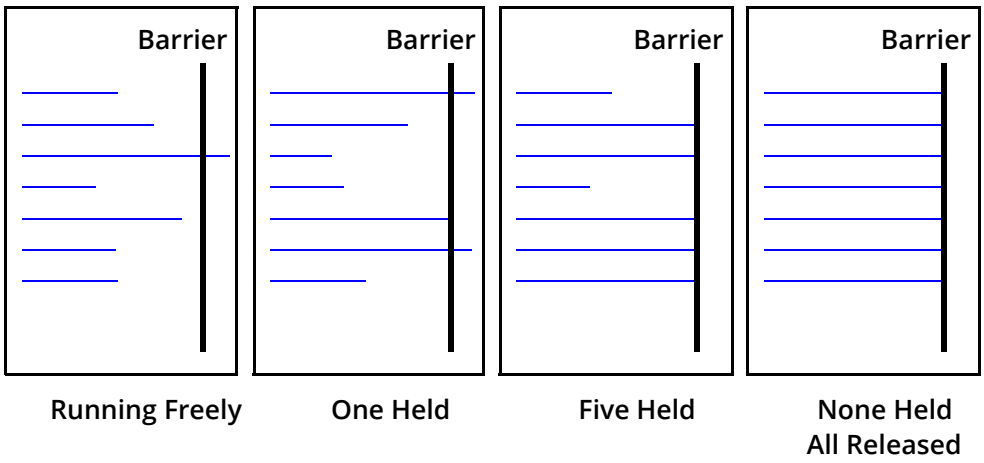
When all processes that share a barrier reach it, TotalView changes their state from *held* to *released*, which means they no longer ignore a command that tells them to begin executing.

The following figure shows seven processes that are sharing the same barrier. (Processes that aren't affected by the barrier aren't shown.)

- First block: All seven processes are running freely.
- Second block: One process hits the barrier and is held. Six processes are executing.
- Third block: Five of the processes have now hit the barrier and are being held. Two are executing.

- Fourth block: All processes have hit the barrier. Because TotalView isn't waiting for anything else to reach the barrier, it changes the processes' states to *released*. Although the processes are released, none are executing.

Figure 221, Running To Barriers



For more information on barriers, see Setting Barrier Points on page 215.

RELATED TOPICS

Barrier points	Setting Barrier Points on page 215
The CLI **dbarrier** command	**dbarrie**r in "CLI Commands" in the *Classic TotalView Reference Guide*

Examining Groups

When you debug a multi-process program, TotalView adds processes to both a control and a share group as the process starts. These groups are not related to either UNIX process groups. (See About Groups, Processes, and Threads on page 383 for information on groups.)

Because a program can have more than one control group and more than one share group, TotalView decides where to place a process based on the type of system call—which can either be **fork()** or **execve()**—that created or changed the process. The two types of process groups are:

Control Group	The parent process and all related processes. A control group includes children that a process forks (processes that share the same source code as the parent). It also includes forked children that subsequently call a function such as **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	The set of processes in a control group that shares the same source code. Members of the same share group share action points.

NOTE: See Group, Process, and Thread Control on page 571 for a complete discussion of groups.

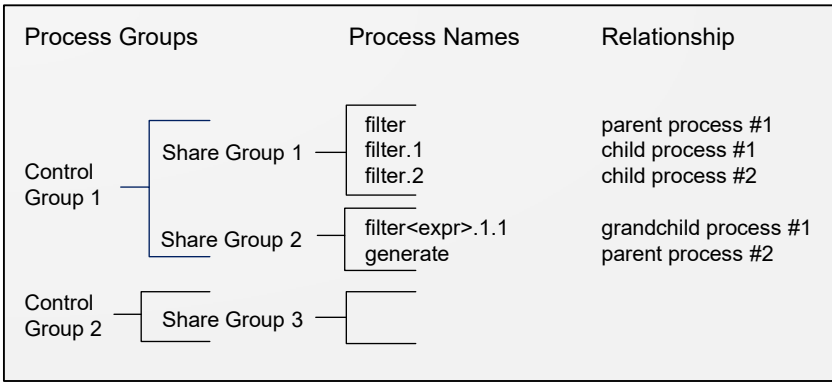
TotalView automatically creates share groups when your processes fork children that call the **execve()** function, or when your program creates processes that use the same code as some parallel programming models such as MPI do.

TotalView names processes according to the name of the source program, using the following naming rules:

- TotalView names the parent process after the source program.
- The name for forked child processes differs from the parent in that TotalView appends a numeric suffix (*.n*). If you're running an MPI program, the numeric suffix is the process's rank in **COMM_WORLD**.
- If a child process calls the **execve()** function after it is forked, TotalView places a new executable name in angle brackets (<>).

In Figure 222, assume that the **generate** process doesn't fork any children, and that the **filter** process forks two child processes. Later, the first child forks another child, and then calls the **execve()** function to execute the **expr** program. In this figure, the middle column shows the names that TotalView uses.

Figure 222, Control and Share Groups Example



RELATED TOPICS

Custom group creation	Creating Custom Groups on page 602
Understanding threads and processes and how TotalView organizes them	About Groups, Processes, and Threads
TotalView's process/thread model in detail	Group, Process, and Thread Control

Placing Processes in Groups

TotalView uses your executable's name to determine the share group that the program belongs to. If the path names are identical, TotalView assumes that they are the same program. If the path names differ, TotalView assumes that they are different, even if the file name in the path name is the same, and places them in different share groups.

RELATED TOPICS	
Using the **Group > Edit Group** command	**Group > Edit Group** in the in-product Help

Starting Processes and Threads

To start a process, select a **Go** command from the **Group**, **Process**, or **Thread** pulldown menus.

After you select a **Go** command, TotalView determines what to execute based on the current thread. It uses this thread, which is called the Thread of Interest (TOI), to decide other threads that should run. For example, if you select **Group > Go**, TotalView continues all threads in the current group that are associated with this thread.

CLI: dfocus g dgo
Abbreviation: G

The commands you will use most often are **Group > Go** and **Process > Go**. The **Group > Go** command creates and starts the current process and all other processes in the multi-process program. There are some limitations, however. TotalView only resumes a process if the following are true:

- The process is not being held.
- The process already exists and is stopped.
- The process is at a breakpoint.

Using a **Group > Go** command on a process that's already running starts the other members of the process's control group.

CLI: dgo

If the process hasn't yet been created, a **Go** command creates and starts it. *Starting* a process means that all threads in the process resume executing unless you are individually holding a thread.

NOTE: TotalView disables the **Thread > Go** command if asynchronous thread control is not available. If you enter a thread-level command in the CLI when asynchronous thread controls aren't available, TotalView tries to perform an equivalent action. For example, it continues a process instead of a thread.

For a single-process program, the **Process > Go** and **Group > Go** commands are equivalent. For a single-threaded process, the **Process > Go** and **Thread > Go** commands are equivalent.

Creating a Process Without Starting It

The **Process > Create** command creates a process and stops it before the first statement in your program executes. If you link a program with shared libraries, TotalView allows the dynamic loader to map into these libraries. Creating a process without starting it is useful when you need to do the following:

- Create watchpoints or change the values of global variables after a process is created, but before it runs.
- Debug C++ static constructor code.

CLI: dstepi
While there is no CLI equivalent to the Process > Create command, executing the dstepi command produces the same effect.

Creating a Process by Single-Stepping

The TotalView single-stepping commands let you create a process and run it to the beginning of your program. The single-stepping commands available from the **Process** menu are as shown in the following table:

GUI command	CLI command	Creates the process and ...
Process > Step	**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.
Process > Next Instruction	**dfocus p dnexti**	Runs it to the first line of the **main()** routine. This is the same as **Process > Step**.

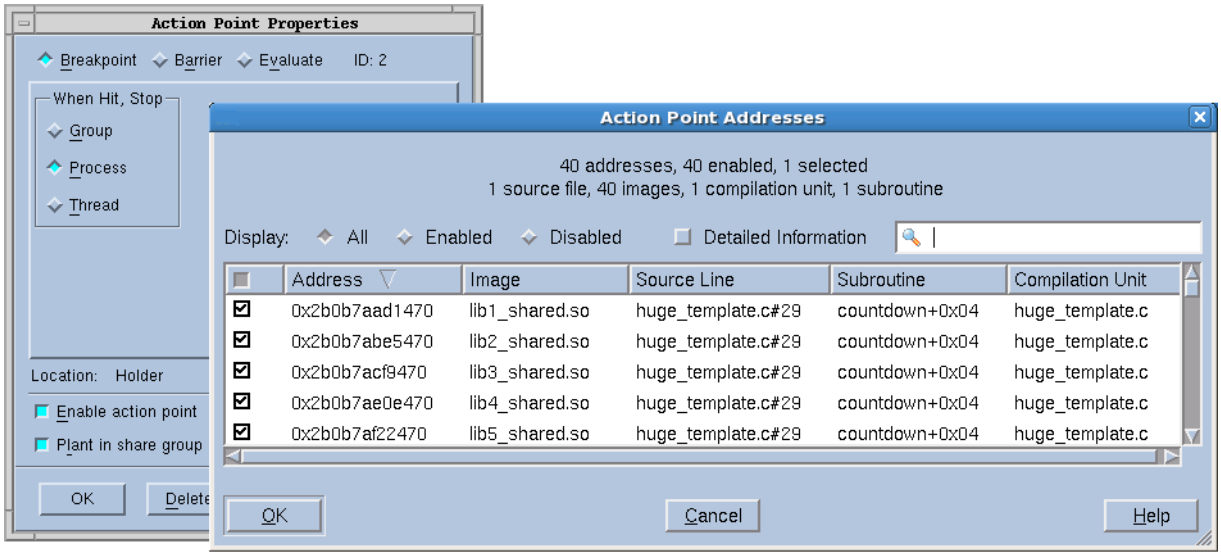
If a group-level or thread-level stepping command creates a process, the behavior is the same as if it were a process-level command.

NOTE: Group, Process, and Thread Control on page 571 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 **Addresses** button in the **Action Point > Properties** Dialog Box to change which instantiations will have a breakpoint.

Figure 223, Action Point and Addresses Dialog Boxes



Initially, addresses are either enabled or disabled, but you can change their state by clicking the checkbox in the first column. The checkbox in the columns bar enables or disables all the addresses. This dialog supports selecting multiple separate items (Ctrl-Click) or a range of items (Shift-Click or click and drag). Once the desired subset is selected, right-click one of the selected items and choose Enable Selection or Disable Selection from the context menu.

Filtering

In complex programs that use many shared libraries, the number of addresses can become very large, so the Addresses dialog has several mechanisms to manage the data. The search box filters the currently displayed data based on one or more space-separated strings or phrases (enclosed in quotes). Remember that data not currently displayed is not included in the filtering. It may be helpful to click the Detailed Information checkbox, which displays much more complete symbol table information, giving you more possibilities for filtering.

Sorting

Clicking on the column labels performs a sort based on the data in that column. Each click toggles between ascending and descending order. If entry values in a column are the same, the values of the column to the right of the sorted column are examined and sorted based on those values. If the values are the same, the next column is examined and so on, until different values are found. The Addresses dialog uses a stable sort, i.e. if all the entries are the same in the selected column and in the columns to the right, the list is not modified.

Displaying and rearranging columns

Finally, right-clicking in the columns bar presents a context menu for displaying or hiding columns. All are initially displayed except Image. You can reorder the columns by selecting a column label and dragging it to a new location.

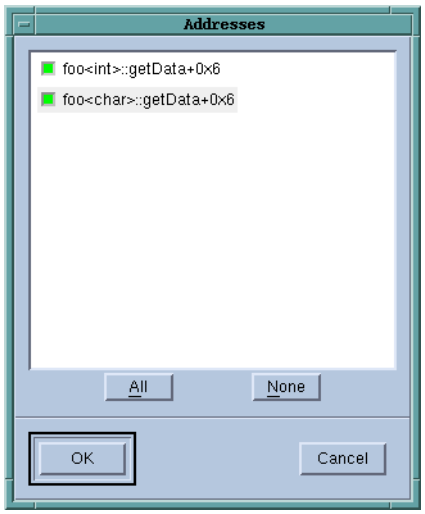
Keyboard Shortcuts

To provide easy access to the buttons at the bottom of the Addresses dialog, the following mnemonic keys have been assigned.

Button	Keyboard Sequence
OK	Alt-o
Cancel	Alt-c
Help	Alt-h

Similarly, if TotalView cannot figure out which instantiation to set a breakpoint at, it displays its **Address** Dialog Box.

Figure 224, Ambiguous Address Dialog Box



RELATED TOPICS

Action points	Setting Action Points on page 188

Debugging Strategies for Parallel Applications

This chapter provides tips and strategies for debugging parallel programs.

RELATED TOPICS

A general discussion on ways to simplify the debugging of complex, multi-threaded and multi-process applications	Simplifying What You're Debugging on page 404

General Parallel Debugging Tips

This section provides debugging tips relevant to most parallel programs.

Breakpoints, Stepping, and Program Execution

Setting Breakpoint Behavior

When you're debugging message-passing and other multi-process programs, it is usually easier to understand the program's behavior if you change the default stopping action of breakpoints and barrier breakpoints. By default, when one process in a multi-process program hits a breakpoint, TotalView stops all other processes.

To change the default stopping action of breakpoints and barrier breakpoints, you can set debugger preferences. The online Help contains information on these preference. These preferences tell TotalView whether to continue to run when a process or thread hits the breakpoint.

These options affect only the default behavior. You can choose a behavior for a breakpoint by setting the breakpoint properties in the **File > Preferences** Action Points Page. See Setting Breakpoints for Multiple Processes on page 211.

Synchronizing Processes

TotalView has two features that make it easier to get all of the processes in a multi-process program synchronized and executing a line of code. Process barrier breakpoints and the process hold/release features work together to help you control the execution of your processes. See Setting Barrier Points on page 215.

The Process Window **Group > Run To** command is a special stepping command. It lets you run a group of processes to a selected source line or instruction. See Stepping (Part I) on page 575.

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** to stop execution.

CLI: dfocus g dhalt
Abbreviation: H

The group-level single-stepping commands such as **Group > Step** and **Group > Next** let you single-step a group of processes in a parallel. See Stepping (Part I) on page 575.

CLI: dfocus g dstep
Abbreviation: S
dfocus g dnext
Abbreviation: N

Stepping at Process Level

If you use a process-level single-stepping command in a multi-process program, TotalView may appear to hang (it continuously displays the watch cursor). If you single-step a process over a statement that can't complete without allowing another process to run, and that process is stopped, the stepping process appears to hang. This can occur, for example, when you try to single-step a process over a communication operation that cannot complete without the participation of another process. When this happens, you can abort the single-step operation by selecting **Cancel** in the **Waiting for Command to Complete** Window that TotalView displays. As an alternative, consider using a group-level single-step command.

CLI: Type Ctrl+C

NOTE: Rogue Wave receives many bug reports on hung processes, usually because one process is waiting for another. Using the **Group** debugging commands almost always solves this problem.

Viewing Processes, Threads, and Variables

Identifying Process and Thread Execution

The Root Window helps you determine where various processes and threads are executing. When you select a line of code in the Process Window, the Root Window updates to show which processes and threads are executing that line.

Viewing Variable Values

You can view the value of a variable that is replicated across multiple processes or multiple threads in a single Variable Window. See Displaying a Variable in all Processes or Threads on page 330.

Restarting from within TotalView

You can restart a parallel program at any time. If your program runs past the point you want to examine, you can kill the program by selecting the **Group > Kill** command. This command kills the master process and all the slave processes. Restarting the master process (for example, **mpirun** or **poe**) recreates all of the slave processes. Start up is faster when you do this because TotalView doesn't need to reread the symbol tables or restart its **tvdsvr** processes, since they are already running.

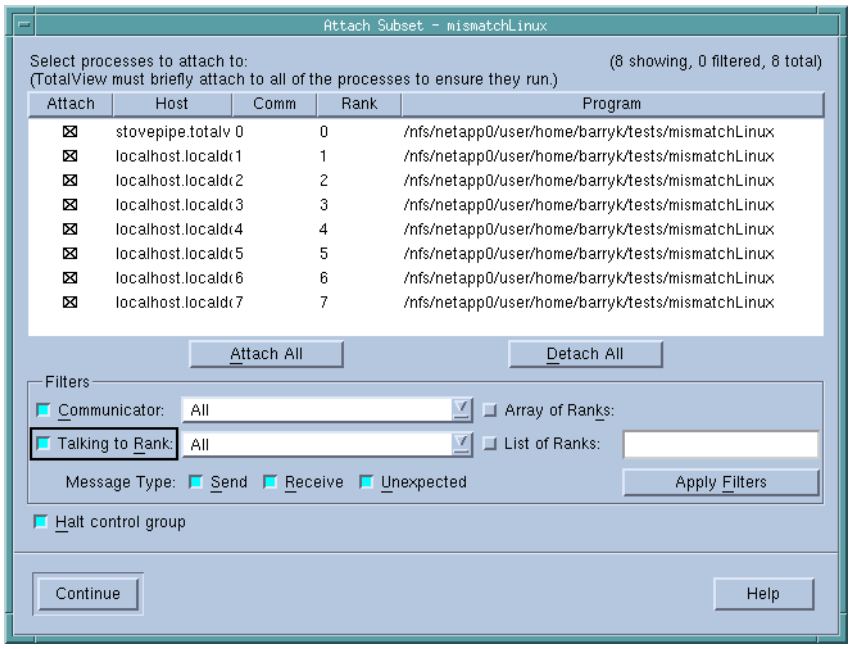
CLI: dfocus g dkill

Attaching to Processes Tips

In a typical multi-process job, you're interested in some processes and not as much in others. By default, TotalView tries to attach to all of the processes that your program starts. If there are a lot of processes, there can be considerable overhead involved in opening and communicating with the jobs.

You can minimize this overhead by using the **Attach Subset** dialog box, shown in Figure 225.

Figure 225, Group > Attach Subset Dialog Box



NOTE: You can start MPI jobs in two ways. One requires that the starter program be under TotalView control and have special instrumentation for TotalView, while the other does not. In the first case, you will enter the name of the starter program on the command line. The other requires that you enter information into the File > Debug New Program or File > Debug New Parallel Program > dialog boxes. The Attach Subset command is available only if you directly name a starter program on the command line.

The Subset Attach dialog box can be launched in multiple ways. It is automatically available when you launch your job with the parallel preference set to "Ask what to do." (See Figure 227). It is also available through other menu options after the job has been started, as discussed later in this section.

Selecting check boxes in the Attach column defines the processes to attach to. Although your program will launch all these processes, TotalView attaches only to the those you have selected.

The **Attach All** and **Detach All** buttons elect or deselect all the processes at once. You can then use the check boxes to select and deselect individual processes. For example, to attach to only a few processes in a lengthy list, use **Detach All** and then select those to which TotalView should attach.

The **Filter** controls restrict which processes are displayed; filtering is unrelated to attaching or detaching.

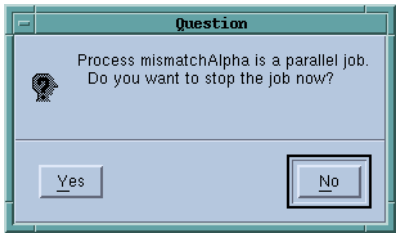
- The **Communicator** control specifies that the processes displayed must be involved with the communicators that you select. For example, if something goes wrong that involves a communicator, selecting it from the list displays only the processes that use that communicator. You can then use **Attach All** to attach to only those processes.
- The **Talking to Rank** control limits the processes displayed to those that receive messages from the indicated ranks. In addition to your rank numers, you can also select **All** or **MPI_ANY_SOURCE**.
- The **Array of Ranks** option is automatically selected and the array name displayed if you have invoked **Tools > Attach Subset (Array of Ranks)** from the Variable Window. In this case, the dialog box will only display the list of processes whose ranks match the array elements.
- The **List of Ranks** control allows you to enter rank numbers to filter on. Use a dash to indicate a range of ranks, and commas to indicate individual ranks. For example: 3, 10-16, 24.
- The three checkboxes in the **Message Type** area add yet another qualifier. Checking a box displays only communicators that are involved with a **Send**, **Receive**, or **Unexpected** message.

The **Halt Control Group** button is not active if the dialog box is launched after the job is already started. It is active only at the initial startup of a parallel job. You typically want to halt processes to allow the setting of breakpoints.

Many applications place values that indicate ranks in an array variable so that the program can refer to them as needed. 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. (See the **Array of Ranks** explanation above.)

You can use the **Group > Attach Subset** command at any time, but you would probably use it immediately before TotalView launches processes. Unless you have set preferences otherwise, TotalView stops and asks if you want it to stop your processes. When selected, the **Halt control group** check box also stops a process just before it begins executing.

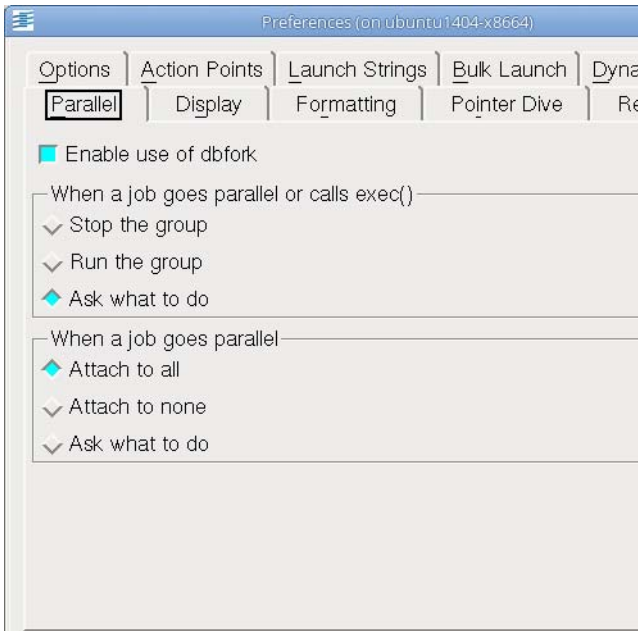
Figure 226, Stop Before Going Parallel Question Box



If you click Yes, when the job stops the starter process should be at a “magic breakpoint.” These are set by TotalView behind the scene, and usually not visible. The other processes may or may not be at a “magic breakpoint.”

The commands on the Parallel Page in the **File > Preferences** Dialog Box control what TotalView does when your program goes parallel.

Figure 227, File > Preferences: Parallel Page



NOTE: TotalView displays the preceding question box only when you directly name a starter program on the command line.

The radio buttons in the **When a job goes parallel or calls exec()** area:

- **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 whether TotalView should start the created processes.

CLI: dset TV::parallel_stop

The radio buttons in the **When a job goes parallel** area:

- **Attach to all**: Automatically attaches to all processes at executing.
- **Attach to none**: Does not attach to any created process at execution.

- **Ask what to do**: Asks what processes to attach to. For this option, the same dialog box opens as that displayed for **Group > Attach Subset**. TotalView then attaches to the processes that you have selected. Note that this dialog box isn't displayed when you set the preference; rather, it controls behavior when your program actually creates parallel processes.

CLI: dset TV::parallel_attach

MPI Debugging Tips and Tools

TotalView provides specific tools to view MPI program status, including rank and message queues.

This section discusses these display tools as well as any other information specific to an MPI program.

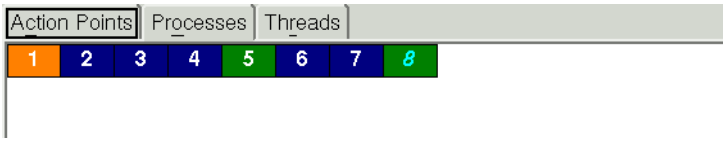
MPI Display Tools

The tools available for MPI display include the Processes/Ranks Tab and the Message Queue Graph Window.

MPI Rank Display

The Processes/Ranks Tab at the bottom of the Process Window displays the status of each rank. For example, in Figure 228, one rank is at a breakpoint, two are running, and five are stopped.

Figure 228, Ranks Tab

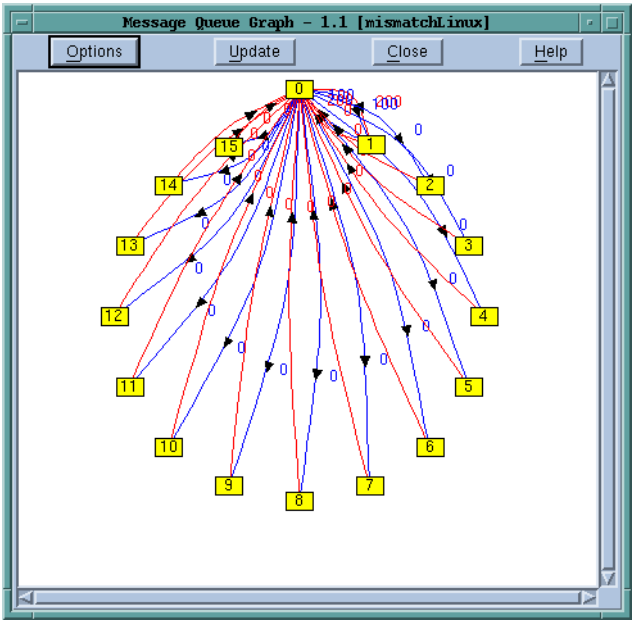


RELATED TOPICS	
Creating Custom Groups	Creating Custom Groups on page 602
The Processes/Rank Tab	Using the Processes/Ranks and Threads Tabs on page 418.

Displaying the Message Queue Graph Window

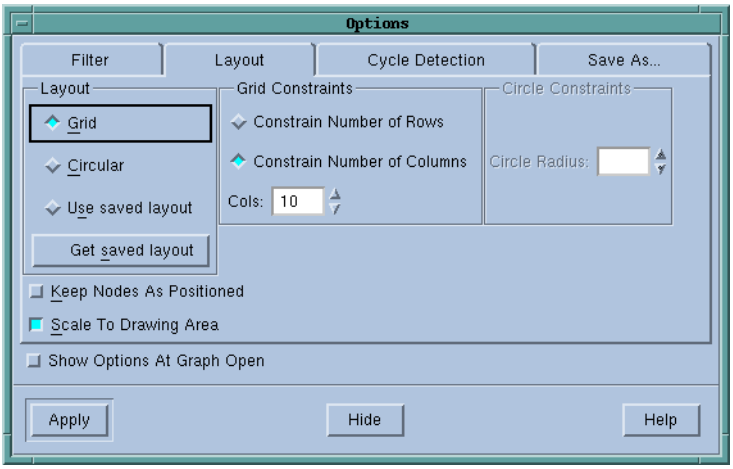
TotalView can graphically display your MPI program's message queue state. Select the Process Window **Tools > Message Queue Graph** command to display a graph of the current message queue state.

Figure 229, Tools > Message Queue Graph Window



If you want to restrict the display, select the **Options** button, Figure 230.

Figure 230, Tools > Message Queue Graph Options Window

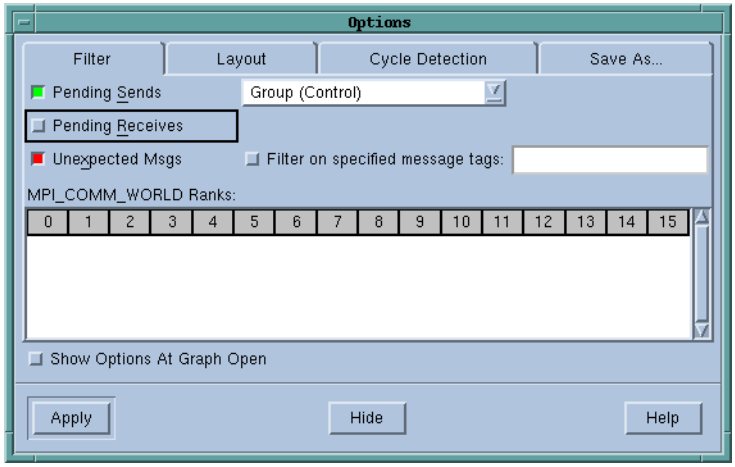


Here, you can alter the way in which TotalView displays ranks within this window—for example, as a grid or in a circle.

Use the commands within the **Cycle Detection** tab to receive reports about cycles in your messages. This is a quick and efficient way to detect when messages are blocking one another and causing deadlocks.

Perhaps the most used of these tabs is **Filter**.

Figure 231, Tools > Message Queue Graph Options. Filter Tab



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

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

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

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

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

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

The numbers next to the arrows indicate the MPI message tags that existed when TotalView created the graph. Diving on an arrow displays the **Tools > Message Queue** Window, with detailed information about the messages. If TotalView has not attached to a process, it displays this information in a grey box.

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

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

Displaying the Message Queue

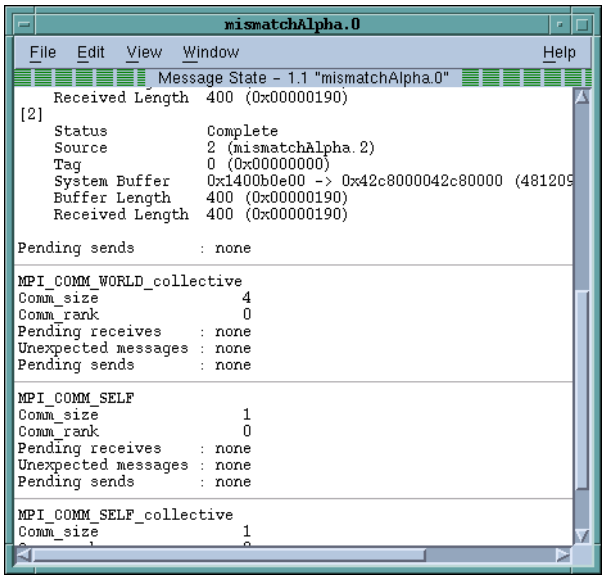
The **Tools > Message Queue** Window displays your MPI program's message queue state textually. This can be useful when you need to find out why a deadlock occurred.

MPI versions that support message queue display are described in the Platforms Guide in the product distribution at `<installdir>/totalview.<version>/doc/pdf`, or available on the TotalView documentation website at https://help.totalview.io/.

About the Message Queue Display

After an MPI process returns from the call to **MPI_Init()**, you can display the internal state of the MPI library by selecting the **Tools > Message Queue** command, Figure 232.

Figure 232, Message Queue Window



This window displays the state of the process' MPI communicators. If user-visible communicators are implemented as two internal communicator structures, TotalView displays both. One is used for point-to-point operations and the other is used for collective operations.

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

The contents of the Message Queue Window are valid only 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***]**).

RELATED TOPICS	
Message Queue field descriptions available for display	"Message Queue Window" in the in-product help
Message operations	"Message Operations" in the in-product Help

Diving on MPI Processes

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

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

Diving on MPI Buffers

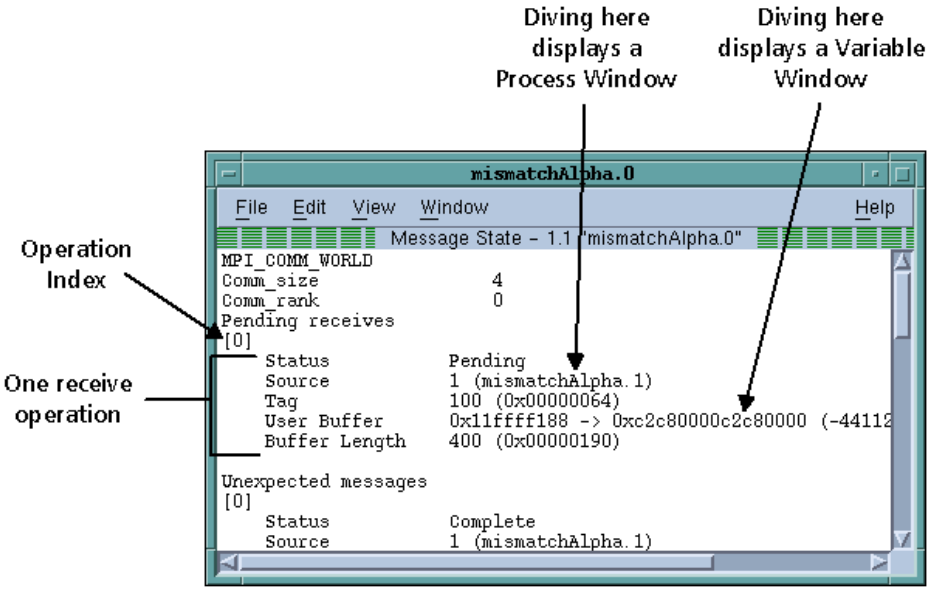
When you dive into the buffer fields, TotalView opens a Variable Window. It also guesses the correct format for the data based on the buffer length and data alignment. You can edit the **Type** field within the Variable Window, if necessary.

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

About Pending Receive Operations

TotalView displays each pending receive operation in the **Pending receives** list. Figure 233 shows an example of an MPICH pending receive operation.

Figure 233, Message Queue Window Showing Pending Receive Operation



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

About Unexpected Messages

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

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

About Pending Send Operations

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

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

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

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

```
Pending sends : no information available
```

MPICH Debugging Tips

These debugging tips apply only to MPICH:

- **Passing options to mpirun**

  You can pass options to TotalView using the MPICH **mpirun** command.

  To pass options to TotalView when running **mpirun**, you can use the **TOTALVIEW** environment variable. For example, you can cause **mpirun** to invoke TotalView with the **-no_stop_all** option, as in the following C shell example:

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

- **Using ch_p4**

  If you start remote processes with MPICH/**ch_p4**, you may need to change the way TotalView starts its servers.

  By default, TotalView uses **ssh** to start its remote server processes. This is the same behavior as **ch_p4** uses. If you configure **ch_p4** to use a different start-up mechanism from another process, you probably also need to change the way that TotalView starts the servers.

RELATED TOPICS	
MPICH configuration and session setup	MPICH Applications on page 523 and MPICH2 Applications on page 528
tvdsvr and **ssh**	TotalView Server Launch Options and Commands on page 495
ssh specifically	Setting the Single-Process Server Launch Command on page 498

IBM PE Debugging Tips

These debugging tips apply only to IBM MPI (PE):

- **Avoid unwanted timeouts**

  Timeouts can occur if you place breakpoints that stop other processes too soon after calling **MPI_Init()** or **MPL_Init()**. If you create "stop all" breakpoints, the first process that gets to the breakpoint stops all the other parallel processes that have not yet arrived at the breakpoint. This can cause a timeout.

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

  CLI: dbarrier *location* **-stop_when_hit process**

- **Control the poe process**

  Even though the **poe** process continues under debugger control, do not attempt to start, stop, or otherwise interact with it. Your parallel tasks require that **poe** continues to run. For this reason, if **poe** is stopped, TotalView automatically continues it when you continue any parallel task.

- **Avoid slow processes due to node saturation**

  If you try to debug a PE program in which more than three parallel tasks run on a single node, the parallel tasks on each node can run noticeably slower than they would run if you were not debugging them.

  In general, the number of processes running on a node should be the same as the number of processors in the node.

  This becomes more noticeable as the number of tasks increases, and, in some cases, the parallel tasks does not progress. This is because PE uses the **SIGALRM** signal to implement communications operations, and AIX requires that debuggers must intercept all signals. As the number of parallel tasks on a node increases, TotalView becomes saturated and can't keep up with the **SIGALRM** signals being sent, thus slowing the tasks.

RELATED TOPICS

Detail on IBM PE configuration and session setup	IBM MPI Parallel Environment (PE) Applications on page 532

PART III Using the CLI

This part deals exclusively with the CLI. Most CLI commands must have a process/thread focus for what they do. See Group, Process, and Thread Control on page 571 for more information.

- Using the Command Line Interface (CLI)

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

- Seeing the CLI at Work

  While you can use the CLI as a stand-alone debugger, using the GUI is usually easier. You will most often use the CLI when you need to debug programs using very slow communication lines or when you need to create debugging functions that are unique to your program. This chapter presents a few Tcl macros in which CLI commands are embedded.

  Most of these examples are simple, designed to give you a feel for what you can do.

Using the Command Line Interface (CLI)

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

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

This chapter contains the following sections:

About the Tcl and the CLI

The CLI is built in version 8.0 of Tcl, so TotalView CLI commands are built into Tcl. This means that the CLI is *not* a library of commands that you can bring into other implementations of Tcl. Because the Tcl you are running is the standard 8.0 version, the CLI supports all libraries and operations that run using version 8.0 of Tcl.

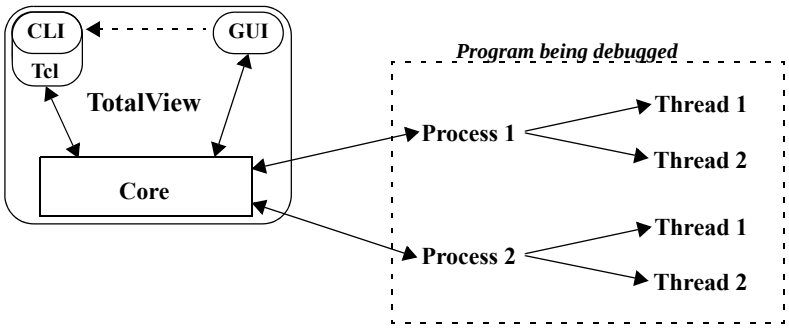
Integrating CLI commands into Tcl makes them intrinsic Tcl commands. This lets you enter and execute all CLI commands in exactly the same way as you enter and execute built-in Tcl commands. As CLI commands are also Tcl commands, you can embed Tcl primitives and functions in CLI commands, and embed CLI commands in sequences of Tcl commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, and then use a CLI command that operates on the elements of this list. You can also create a Tcl function that dynamically builds the arguments that a process uses when it begins executing.

About The CLI and TotalView

Figure 234 illustrates the relationship between the CLI, the GUI, the TotalView core, and your program:

Figure 234, The CLI, GUI and TotalView



The CLI and GUI are components that communicate with the TotalView core, which is what actually does the work. In this figure, the dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI. The reverse is not true: you can't invoke the GUI from the CLI.

In turn, the TotalView core communicates with the processes that make up your program, receives information back from these processes, and passes information back to the component that sent the request. If the GUI is also active, the core also updates the GUI's windows. For example, stepping your program from within the CLI changes the PC in the Process Window, updates data values, and so on.

Using the CLI Interface

You interact with the CLI by entering a CLI or Tcl command. (Entering a Tcl command does exactly the same thing in the CLI as it does when interacting with a Tcl interpreter.) Typically, the effect of executing a CLI command is one or more of the following:

- The CLI displays information about your program.
- A change takes place in your program's state.
- A change takes place in the information that the CLI maintains about your program.

After the CLI executes your command, it displays a prompt. Although CLI commands are executed sequentially, commands executed by your program might not be. For example, the CLI doesn't require that your program be stopped when it prompts for and performs commands. It only requires that the last CLI command be complete before it can begin executing the next one. In many cases, the processes and threads being debugged continue to execute after the CLI has finished doing what you asked it to do.

If you need to stop an executing command or Tcl macro, press Ctrl+C while the command is executing. If the CLI is displaying its prompt, typing Ctrl+C stops any executing processes.

Because actions are occurring constantly, state information and other kinds of messages that the CLI displays are usually mixed in with the commands that you type. You might want to limit the amount of information TotalView displays by setting the **VERBOSE** variable to **WARNING** or **ERROR**. (For more information, see the "*Variables*" chapter in the *Classic TotalView Reference Guide*.)

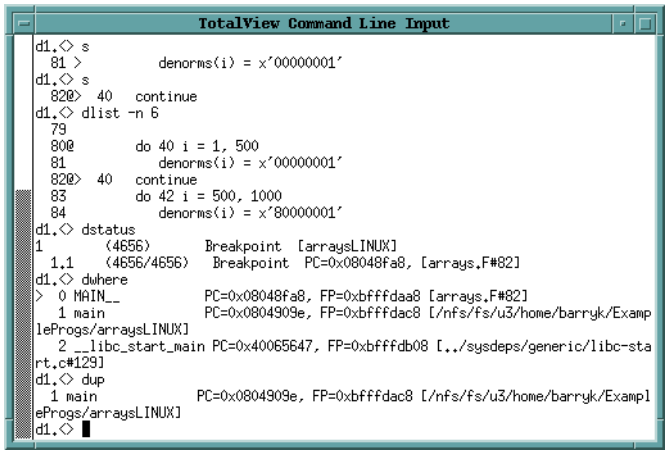
Starting the CLI

You can start the CLI in one of the following ways:

- You can start the CLI from the GUI by selecting the **Tools > Command Line** command in the Root or Process Windows. After selecting this command, TotalView opens a window into which you can enter CLI commands.
- You can start the CLI directly from a shell prompt by typing **totalviewcli**. (This assumes that the TotalView binary directory is in your path.)

Figure 235 is a snapshot of a CLI window that shows part of a program being debugged.

Figure 235, CLI xterm Window



If you have problems entering and editing commands, it might be because you invoked the CLI from a shell or process that manipulates your **stty** settings. You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option isn't available, you have to change values individually.)

If you start the CLI with the **totalviewcli** command, you can use all of the command-line options that you can use when starting TotalView, except those that have to do with the GUI. (In some cases, TotalView displays an error message if you try. In others, it just ignores what you did.)

Information on command-line options is in the "TotalView Command Syntax" chapter of the *Classic TotalView Reference Guide*.

RELATED TOPICS	
All the ways to start TotalView	Starting TotalView on page 89
How to perform remote debugging	Setting Up Remote Debugging Sessions on page 484
Setting up for MPI debugging	Setting Up MPI Debugging Sessions on page 516
Setting up for non-MPI parallel debugging	Setting Up Parallel Debugging Sessions on page 546

Startup Example

The following is a very small CLI script:

```
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

This script begins by loading and interpreting the **make_actions.tcl** file, which was described in Seeing the CLI at Work on page 472. It then loads the **fork_loop** executable, sets its default startup arguments, and steps one source-level statement.

If you stored this in a file named **fork_loop.tvd**, you can tell TotalView to start the CLI and execute this file by entering the following command:

```
totalviewcli -s fork_loop.tvd
```

The following example places a similar set of commands in a file that you invoke from the shell:

```
#!/bin/sh
# Next line executed by shell, but ignored by Tcl because: \
   exec totalviewcli -s "$0" "$@"
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

The only real difference between the last two examples is the first few lines in the file. In this second example, the shell ignores the backslash continuation character; Tcl processes it. This means that the shell executes the **exec** command while Tcl will ignore it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute your program from within the CLI, enter a **dload** command followed by the **drun** command.

NOTE: If your program is launched from a starter program such as srun or aprun, use the drerun command rather than drun to start your program. If you use drun, default arguments to the process are suppressed; drerun passes them on.

The following example uses the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. Since this was not the first time the file was run, breakpoints exist from a previous session.

NOTE: In this listing, the CLI prompt is "`d1.<>`". The information preceding the greater-than symbol (>) symbol indicates the processes and threads upon which the current command acts. The prompt is discussed in About the CLI Prompt on page 466.

```
% totalviewcli
d1.<> dload arraysAlpha   #load the arraysAlpha program
1
d1.<> dactions           # Show the action points
No matching breakpoints were found
d1.<> dlist -n 10 75
    75       real16_array (i, j) = 4.093215 * j+2
    76 #endif
    77 26    continue
    78 27    continue
    79
    80 do 40 i = 1, 500
    81    denorms(i) = x'00000001'
    82 40 continue
    83 do 42 i = 500, 1000
    84    denorms(i) = x'80000001'
d1.<> dbreak 80           # Add two action points
1
d1.<> dbreak 83
2
```

```
d1.<> drun        # Run the program to the action point
```

This two-step operation of loading and running supports setting action points before execution begins, as well as executing a program more than once. At a later time, you can use **drerun** to restart your program, perhaps sending it new 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 a new process. The new process is in addition to any existing processes for the program because the CLI does not shut down older processes when starting the new one.

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

```
d1.<> dkill             # kills process
d1.<> drun              # runs program from start
d1.<> dlist -e -n 3     # shows lines about current spot
   79
   80@>      do 40 i = 1, 500
   81            denorms(i) = x'00000001'
d1.<> dwhat master_array # Tell me about master_array
In thread 1.1:
Name: master_array; Type: integer(100);
    Size: 400 bytes; Addr: 0x140821310
    Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
    (Scope class: Any)
    Address class: proc_static_var
    (Routine static variable)
d1.<> dgo                    # Start program running
d1.<> dwhat denorms          # Tell me about denorms
In thread 1.1:
Name: denorms; Type: <void>; Size: 8 bytes;
    Addr: 0x1408214b8
    Scope: ##arraysAlpha#arrays.F#check_fortran_arrays
    (Scope class: Any)
    Address class: proc_static_var
    (Routine static variable)
d1.<> dprint denorms(0)      # Show me what is stored
 denorms(0) = 0x0000000000000001 (1)
d1.<>
```

Because information is interleaved, you may not realize that the prompt has appeared. It is always safe to use the Enter key to have the CLI redisplay its prompt. If a prompt isn't displayed after you press Enter, you know that the CLI is still executing.

About CLI Output

A CLI command can either print its output to a window or return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you won't care about the difference between *printing* and *returning-and-printing*. Either way, the CLI displays information in your window. And, in both cases, printed output is fed through a simple *more* processor. (This is discussed in more detail in the next section.)

In the following two cases, it matters whether the CLI directly prints output or returns and then prints it:

- When the Tcl interpreter executes a list of commands, the CLI only prints the information returned from the last command. It doesn't show information returned by other commands.
- You can only assign the output of a command to a variable if the CLI returns a command's output. You can't assign output that the interpreter prints directly to a variable, or otherwise manipulate it, unless you save it using the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed—unless, of course, the **dload** command appears in the middle of a list of commands; for example:

```
{dload test_program;dstatus}
```

In this example, the CLI doesn't display the ID of the loaded program, since the **dload** command was not the last command.

When information is returned, you can assign it to a variable. For example, the next command assigns the ID of a newly created process to a variable:

```
set pid [dload test_program]
```

Because you can't assign the output of the **help** command to a variable, the following doesn't work:

```
set htext [help]
```

This statement assigns an empty string to **htext** because the **help** command doesn't return text. It just prints it.

To save the output of a command that prints its output, use the **capture** command. For example, the following example writes the **help** command's output into a variable:

```
set htext [capture help]
```

NOTE: You can capture the output only from commands. You can't capture the informational messages displayed by the CLI that describe process state. If you are using the GUI, TotalView also writes this information to the Log Window. You can display this information by using the **Tools > Event Log** command.

‘more’ Processing

When the CLI displays output, it sends data through a simple *more*-like process. This prevents data from scrolling off the screen before you view it. After you see the **MORE** prompt, press Enter to see the next screen of data. If you type **q** (followed by pressing the Enter key), the CLI discards any data it hasn’t yet displayed.

You can control the number of lines displayed between prompts by using the **dset** command to set the **LINES_PER_SCREEN** CLI variable. (For more information, see the *Classic TotalView Reference Guide*.)

Using Command Arguments

The default command arguments for a process are stored in the **ARGS**(*num*) variable, where *num* is the CLI ID for the process. If you don't set the **ARGS**(*num*) variable for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. TotalView sets the **ARGS_DEFAULT** variable when you use the **-a** option when starting the CLI or the GUI.

NOTE: The **-a** option tells TotalView to pass everything that follows on the command line to the program.

For example:

```
totalviewcli -a argument-1, argument-2, ...
```

To set (or clear) the default arguments for a process, you can use the **dset** (or **dunset**) command to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, the following clears the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained in **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable; for example:

```
dunset ARGS_DEFAULT
```

All commands (except the **drun** command) that can create a process—including the **dgo**, **drerun**, **dcont**, **dstep**, and **dnext** commands—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

RELATED TOPICS	
The **ARGS** variable	**ARGS** in "TotalView Variables" in the *Classic TotalView Reference Guide*
The **ARGS_DEFAULT** variable	**ARGS_DEFAULT** in "TotalView Variables" in the *Classic TotalView Reference Guide*
The **Process > Startup Parameters** command	**Process > Startup Parameters** in the in-product Help

Using Namespaces

CLI interactive commands exist in the primary Tcl namespace (**::**). Some of the TotalView state variables also reside in this namespace. Seldom-used functions and functions that are not primarily used interactively reside in other namespaces. These namespaces also contain most TotalView state variables. (The variables that appear in other namespaces are usually related to TotalView preferences.) TotalView uses the following namespaces:

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

If you discover other namespaces beginning with **TV**, you have found a namespace that contains private functions and variables. These objects can (and will) disappear, so don't use them. Also, don't create namespaces that begin with **TV**, since you can cause problems by interfering with built-in functions and variables.

The CLI **dset** command lets you set the value of these variables. You can have the CLI display a list of these variables by specifying the namespace; for example:

```
dset TV::
```

You can use wildcards with this command. For example, **dset TV::au*** displays all variables that begin with "au".

RELATED TOPICS	
CLI namespace commands	"CLI Namespace Commands" in the *Classic TotalView Reference Guide*
TotalView variables	"TotalView Variables" in the *Classic TotalView Reference Guide*

About the CLI Prompt

The appearance of the CLI prompt lets you know that the CLI is ready to accept a command. This prompt lists the current focus, and then displays a greater-than symbol (**>**) and a blank space. (The *current focus* is the processes and threads to which the next command applies.) For example:

d1.<>	The current focus is the default set for each command, focusing on the first user thread in process 1.
g2.3>	The current focus is process 2, thread 3; commands act on the entire group.
t1.7>	The current focus is thread 7 of process 1.
gW3.>	The current focus is all worker threads in the control group that contains process 3.
p3/3	The current focus is all processes in process 3, group 3.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable; for example:

```
dset PROMPT "Kill this bug! > "
```

Using Built-in and Group Aliases

Many CLI commands have an alias that lets you abbreviate the command's name. (An alias is one or more characters that Tcl interprets as a command or command argument.)

NOTE: The **alias** command, which is described in the Classic TotalView Reference Guide, lets you create your own aliases.

For example, the following command tells the CLI to halt the current group:

```
dfocus g dhalt
```

Using an abbreviation is easier. The following command does the same thing:

```
f g h
```

You often type less-used commands in full, but some commands are almost always abbreviated. These commands include **dbreak** (**b**), **ddown** (**d**), **dfocus** (**f**), **dgo** (**g**), **dlist** (**l**), **dnext** (**n**), **dprint** (**p**), **dstep** (**s**), and **dup** (**u**).

The CLI also includes uppercase group versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is **s**; in contrast, **S** is the alias for **dfocus g dstep**. (The first command tells the CLI to step the process. The second steps the control group.)

Group aliases differ from the group-level command that you type interactively, as follows:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

How Parallelism Affects Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

- **Initial process**

  A pre-existing process from the normal run-time environment (that is, created outside TotalView), or one that was created as TotalView loaded the program.

- **Spawned process**

  A new process created by a process executing under CLI control.

TotalView assigns an integer value to each individual process and thread under its control. This *process/thread identifier* can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; in contrast, thread numbers are only unique while the process exists.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

Thread 1 of process 1
Thread 2 of process 1
Thread 1 of process 2
Thread 2 of process 2

You identify the four threads as follows:

1.1—Thread 1 of process 1
1.2—Thread 2 of process 1
2.1—Thread 1 of process 2
2.2—Thread 2 of process 2

RELATED TOPICS

An overview of threads and processes and how TotalView organizes them into groups	About Groups, Processes, and Threads on page 383
More detail on the TotalView thread/process model and how to create custom groups	Group, Process, and Thread Control on page 571

Types of IDs

Multi-threaded, multi-process, and distributed programs contain a variety of IDs. The following types are used in the CLI and the GUI:

System PID This is the process ID and is generally called the *PID*.

System TID This is the ID of the system kernel or user thread. On some systems (for example, AIX), the TIDs have no obvious meaning. On other systems, they start at 1 and are incremented by 1 for each thread.

TotalView thread ID This is usually identical to the system TID. On some systems (such as AIX) where the threads have no obvious meaning, TotalView uses its own IDs.

pthread ID This is the ID assigned by the Posix pthreads package. If this differs from the system TID, the TID is a pointer value that points to the pthread ID.

Debugger PID This is an ID created by TotalView that lets it identify processes. It is a sequentially numbered value beginning at 1 that is incremented for each new process. If the target process is killed and restarted (that is, you use the **dkill** and **drun** commands), the TotalView PID does not change. The system PID changes, however, since the operating system has created a new target process.

Controlling Program Execution

Knowing what's going on and where your program is executing is simple in a serial debugging environment. Your program is either stopped or running. When it is running, an event such as arriving at a breakpoint can occur. This event tells TotalView to stop the program. Sometime later, you tell the serial program to continue executing. Multi-process and multi-threaded programs are more complicated. Each thread and each process has its own execution state. When a thread (or set of threads) triggers a breakpoint, TotalView must decide what it should do about other threads and processes because it may need to stop some and let others continue to run.

RELATED TOPICS	
Tasks for working with a multi-process, multi-threaded application	Manipulating Processes and Threads on page 407
Stepping commands	Using Stepping Commands on page 178
The **dload** command	**dload** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **dattach** command	**dattach** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **drun** command	**drun** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **dkill** command	**dkill** in "CLI Commands" in the *Classic TotalView Reference Guide*

Advancing Program Execution

Debugging begins by entering a **dload** or **dattach** command. If you use the **dload** command, you must use the **drun** (or perhaps **drerun** if there's a starter program) command to start the program executing. These three commands work at the process level and you can't use them to start individual threads. (This is also true for the **dkill** command.)

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a P/T set. (P/T sets are discussed in About Groups, Processes, and Threads on page 383 and Group, Process, and Thread Control on page 571.) 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 a tool that lets you alter a program's state in a controlled way, and debugging is the process of stopping a process to examine its state. However, the term *stop* has a slightly different meaning in a multi-process, multi-threaded program. In these programs, *stopping* means that the CLI holds one or more threads at a location until you enter a command that tells them to start executing again. Other threads, however, may continue executing.

For more detailed information on debugging in general, see Part II, Debugging Tools and Tasks on page 84.

Using Action Points

Action points tell the CLI to stop a program's execution. You can specify the following types of action points:

- A *breakpoint* (see **dbreak** in the *Classic TotalView Reference Guide*) stops the process when the program reaches a location in the source code.
- A *watchpoint* (see **dwatch** in the *Classic TotalView Reference Guide*) stops the process when the value of a variable is changed.
- A *barrier point* (see **dbarrier** in the *Classic TotalView Reference Guide*), as its name suggests, effectively prevents processes from proceeding beyond a point until all other related processes arrive. This gives you a method for synchronizing the activities of processes. (You can only set a barrier point on processes; you can't set then on individual threads.)
- An *eval point* (see **dbreak** in the *Classic TotalView Reference Guide*) lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. An eval point typically does not stop the process; instead, it performs an action. In most cases, an eval point stops the process when some condition that you specify is met.

NOTE: For extensive information on action points, see Setting Action Points on page 188.

Each action point is associated with an *action point identifier*. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1; the second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and the GUI let you assign only one action point to a source code line, but you can make this action point as complex as you need it to be.

Seeing the CLI at Work

The CLI is a command-line debugger that is completely integrated with TotalView. You can use it and never use the TotalView GUI, or you can use it and the GUI simultaneously. Because the CLI is embedded in a Tcl interpreter, you can also create debugging functions that exactly meet your needs. When you do this, you can use these functions in the same way that you use TotalView's built-in CLI commands.

This chapter contains macros that show how the CLI programmatically interacts with your program and with TotalView. Reading examples without bothering too much with details gives you an appreciation for what the CLI can do and how you can use it. With a basic knowledge of Tcl, you can make full use of all CLI features.

In each macro in this chapter, all Tcl commands that are unique to the CLI are displayed in bold.

These macros perform the following tasks:

Setting the CLI EXECUTABLE_PATH Variable

The following macro recursively descends through all directories, starting at a location that you enter. (This is indicated by the ***root*** argument.) The macro ignores directories named in the ***filter*** argument. The result is set as the value of the CLI **EXECUTABLE_PATH** state variable.

See also the *Classic TotalView Reference Guide*'s entry for the **EXECUTABLE_PATH** variable

```
# Usage:
#
# rpath [root] [filter]
#
# If root is not specified, start at the current
# directory. filter is a regular expression that removes
# unwanted entries. If it is not specified, the macro
# automatically filters out CVS/RCS/SCCS directories.
#
# The search path is set to the result.

proc rpath {{root "."} {filter "/(CVS|RCS|SCCS)(/|$)"}} {

   # Invoke the UNIX find command to recursively obtain
   # a list of all directory names below "root".
   set find [split [exec find $root -type d -print] \n]

   set npath ""

   # Filter out unwanted directories.
   foreach path $find {
      if {! [regexp $filter $path]} {
         append npath ":"
         append npath $path
      }
   }

   # Tell TotalView to use it.
   dset EXECUTABLE_PATH $npath
}
```

In this macro, the last statement sets the **EXECUTABLE_PATH** state variable. This is the only statement that is unique to the CLI. All other statements are standard Tcl.

The **dset** command, like most interactive CLI commands, begins with the letter **d**. (The **dset** command is only used in assigning values to CLI state variables. In contrast, values are assigned to Tcl variables by using the standard Tcl **set** command.)

Initializing an Array Slice

The following macro initializes an array slice to a constant value:

```
array_set (var lower_bound upper_bound val) {
   for {set i $lower_bound} {$i <= $upper_bound} {incr i}{
      dassign $var\($i) $val
   }
}
```

The CLI **dassign** command assigns a value to a variable. In this case, it is setting the value of an array element. Use this function as follows:

```
d1.<> dprint list3
 list3 = {
   (1) = 1 (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)
 }
```

For more information on slices, see Displaying Array Slices on page 313.

Printing an Array Slice

The following macro prints a Fortran array slice. This macro, like others shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslice {anArray i1 i2 j1 j2 {i3 1} {j3 1} \
        {width 20}} {
   for {set i $i1} {$i <= $i2} {incr i $i3} {
      set row_out ""
      for {set j $j1} {$j <= $j2} {incr j $j3} {
         set ij [capture dprint $anArray\($i,$j\)]
         set ij [string range $ij \
               [expr [string first "=" $ij] + 1] end]
         set ij [string trimright $ij]
         if {[string first "-" $ij] == 1} {
            set ij [string range $ij 1 end]}
         append ij "                    "
         append row_out " " \
               [string range $ij 0 $width] " "
      }
      puts $row_out
   }
}
```

NOTE: The CLI's **dprint** command lets you specify a slice. For example, you can type: **dprint a(1:4,1:4)**.

After invoking this macro, the CLI prints a two-dimensional slice **(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 About CLI Output on page 462.

The following shows how this macro is used:

```
d1.<> pf2Dslice a 1 4 1 4
    0.841470956802 0.909297406673 0.141120001673-0.756802499294
    0.909297406673-0.756802499294-0.279415488243 0.989358246326
    0.141120001673-0.279415488243 0.412118494510-0.536572933197
   -0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslice a 1 4 1 4 1 1 17
    0.841470956802 0.909297406673 0.141120001673-0.756802499294
    0.909297406673-0.756802499294-0.279415488243 0.989358246326
    0.141120001673-0.279415488243 0.412118494510-0.536572933197
   -0.756802499294 0.989358246326-0.536572933197-0.287903308868
```

```
d1.<> pf2Dslice a 1 4 1 4 2 2 10
    0.84147095    0.14112000
    0.14112000    0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
   -0.75680249    0.98935824
    0.98935824   -0.28790330
d1.<>
```

Writing an Array Variable to a File

It often occurs that you want to save the value of an array so that you can analyze its results at a later time. The following macro writes array values to a file:

```
proc save_to_file {var fname} {
   set values [capture dprint $var]
   set f [open $fname w]

   puts $f $values
   close $f
}
```

The following example shows how you might use this macro. Using the **exec** command tells the shell's **cat** command to display the file that was just written.

```
d1.<> dprint list3
 list3 = {
   (1) = 1 (0x00000001)
   (2) = 2 (0x00000002)
   (3) = 3 (0x00000003)
 }
d1.<> save_to_file list3 foo
d1.<> exec cat foo
 list3 = {
   (1) = 1 (0x00000001)
   (2) = 2 (0x00000002)
   (3) = 3 (0x00000003)
 }
d1.<>
```

Automatically Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems are occurring. The following CLI macro parses comments that you can include in a source file and, depending on the comment's text, sets a breakpoint or an eval point.

(For detailed information on action points, see Setting Action Points on page 188.)

Following this macro is an excerpt from a program that uses it.

```
# make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions {{filename ""}} {

   if {$filename == ""} {
      puts "You need to specify a filename"
      error "No filename"
   }

      # Open the program's source file and initialize a
      # few variables.
   set fname [set filename]
   set fsource [open $fname r]
   set lineno 0
   set incomment 0

      # Look for "signals" that indicate the type of
      # action point; they are buried in the comments.
   while {[gets $fsource line] !=-1} {
      incr lineno
      set bpline $lineno

            # Look for a one-line eval point. The
            # format is ... /* EVAL: some_text */.
            # The text after EVAL and before the "*/" in
            # the comment is assigned to "code".
      if [regexp "/\\* EVAL: *(.*)\\*/" $line all code] {
         dbreak $fname\#$bpline -e $code
         continue
      }

            # Look for a multiline eval point.
      if [regexp "/\\* EVAL: *(.*)" $line all code] {
            # Append lines to "code".
```

```
         while {[gets $fsource interiorline] != -1} {
            incr lineno

            # Tabs will confuse dbreak.
            regsub -all \t $interiorline \
                  "  " interiorline

            # If "*/" is found, add the text to "code",
            # then leave the loop. Otherwise, add the
            # text, and continue looping.
            if [regexp "(.*)\\*/" $interiorline \
                  all interiorcode]{
               append code \n $interiorcode
               break
            } else {
               append code \n $interiorline
            }
         }
         dbreak $fname\#$bpline -e $code
         continue
      }
            # Look for a breakpoint.
      if [regexp "/\\* STOP: .*" $line] {
         dbreak $fname\#$bpline
         continue
      }
            # Look for a command to be executed by Tcl.
      if [regexp "/\\* *CMD: *(.*)\\*/" $line all cmd] {
         puts "CMD: [set cmd]"
         eval $cmd
      }
   }
   close $fsource
}
```

The only similarity between this macro and the previous three is that almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command that set eval points and breakpoints.

The following excerpt from a larger program shows how to embed comments in a source file that is read by the **make_actions** macro:

```
...
struct struct_bit_fields_only {
   unsigned f3 : 3;
   unsigned f4 : 4;
   unsigned f5 : 5;
   unsigned f20 : 20;
```

```
   unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
...
int main()
{
   struct struct_bit_fields_only *lbfop = &sbfo;
...
   int i;
   int j;

   sbfo.f3 = 3;
   sbfo.f4 = 4;
   sbfo.f5 = 5;
   sbfo.f20 = 20;
   sbfo.f32 = 32;
...
   /* TEST: Check to see if we can access all the
      values */
   i=i;   /* STOP: */
   i=1;   /* EVAL: if (sbfo.f3 != 3) $stop; */
   i=2;   /* EVAL: if (sbfo.f4 != 4) $stop; */
   i=3;   /* EVAL: if (sbfo.f5 != 5) $stop; */
   ...
   return 0;
}
```

The **make_actions** macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with **/* STOP**, **/* EVAL**, and **/* CMD**. After parsing the comment, it sets a breakpoint at a *stop* line, an eval point at an *eval* line, or executes a command at a *cmd* line.

Using eval points can be confusing because eval point syntax differs from that of Tcl. In this example, the **$stop** function is built into the CLI. Stated differently, you can end up with Tcl code that also contains C, C++, Fortran, and TotalView functions, variables, and statements. Fortunately, you only use this kind of mixture in a few places and you'll know what you're doing.

PART IV Advanced Tools and Customization

This part discusses tools and configurations that are either specific to a particular environment or setup, or that are used only in advanced customizations or other non-routine ways.

- Setting Up Remote Debugging Sessions

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

- Setting Up MPI Debugging Sessions

  Setting up an MPI debugging session may require special startup or environment configuration. This chapter details any non-default configuration information for individual platforms.

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

- Setting Up Parallel Debugging Sessions

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

- Controlling fork, vfork, and execve Handling

  You can control TotalView's behavior for system calls to fork, vfork, and execve.

- Group, Process, and Thread Control

  In a multi-process, multi-threaded program, you may need to finely control execution. This chapter discusses the TotalView process/thread model, how to direct a command to a specific process or thread, and how to create custom groups of processes.

- Scalability in HPC Computing Environments

When working in an HPC environment, you can configure TotalView for maximum scalability, including the use of MRNet, a tree-based overlay network that supports scalable communication.

Setting Up Remote Debugging Sessions

This chapter explains how to set up TotalView remote debugging sessions, detailed in the following sections:

- Automatically Launching a Process on a Remote Server on page 487.

  In most cases, you can easily perform this from the New Program dialog which launches the TotalView Server **tvdsvr** program automatically. If so, you will likely not need to read any of the following sections.

- Troubleshooting Server Autolaunch on page 488

  Some systems have requirements that may prohibit TotalView's default autolaunching capabilities. This section discusses various ways to customize autolaunch options and commands.

- Starting the TotalView Server Manually on page 492

  You can also just manually launch the **tvdsvr** program, discussed in this section.

- TotalView Server Launch Options and Commands on page 495

  The **File > Preferences** dialog box features several ways to customize both options and commands for single and bulk server launch. This section discusses these options as well as specific commands relevant to particular platforms.

- Debugging Over a Serial Line on page 503

  TotalView supports debugging programs over a serial line as well as TCP/IP sockets, discussed in this section.

About Remote Debugging

Debugging a remote process with TotalView is similar to debugging a native process, with these primary differences:

- The remote server hosting the processes to debug must be running the TotalView Server process **tvdsvr**, automatically launched by TotalView in most cases.
- TotalView performance depends on your network's performance. If the network is overloaded, debugging can be slow.

NOTE: You cannot debug remote processes using TotalView Individual.

TotalView can automatically launch **tvdsvr** either:

- Independently on each remote host, called *single-process server launch*.
- As a bulk job, launching all remote processes at the same time, called *bulk server launch*.

Because TotalView can automatically launch **tvdsvr**, programs that launch remote processes rarely require any special handling. When using TotalView, it doesn't matter whether a process is local or remote.

NOTE: When debugging programs remotely, the architecture of the remote machine must be compatible with that of the machine running TotalView. See Platform Issues when Remote Debugging on page 485 for more information.

Platform Issues when Remote Debugging

In general, when debugging programs remotely, the architecture of the remote machine must be compatible with that of the machine running TotalView. For example, you cannot perform remote debugging on a Linux x86-64 system if you launch TotalView from a Linux PowerLE system. In addition, the operating systems must also be compatible.

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

NOTE: The path to TotalView must be identical on the local and all remote systems so that TotalView can find the tvdsvr program.

TotalView assumes that you launch **tvdsvr** using **ssh -x**. If **ssh** is unavailable, set the **TVDSVRLAUNCHCMD** environment variable to the command that you use to remotely access the remote system.

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

Automatically Launching a Process on a Remote Server

In most cases, loading a process to debug on a remote server is no different than debugging a process on a local host. You can add or select a remote host from these debugging sessions:

- **File > Debug New Program**
- **File > Attach to a Running Program**
- **File > Debug Core File or Replay Recording File**

After you have set up a debug session, TotalView can automatically launch the process **tvdsvr** on the remote computer. For more information, see Adding a Remote Host on page 116. If this simple procedure does not work for you, your system may not support TotalView's default autolaunching. You can disable autolaunch or reconfigure some of your settings. See Troubleshooting Server Autolaunch on page 488.

Troubleshooting Server Autolaunch

Some systems do not support TotalView's default autolaunch behavior, requiring you to create your own autolaunch command or requiring special permissions or some other custom configuration.

If autolaunching of the TotalView Server is not working, you can

- Disable autolaunch and start the TotalView server manually (Starting the TotalView Server Manually on page 492)
- Customize either server options or commands, discussed here.

This section discusses how to edit the remote shell command as well as the arguments provided to TotalView at remote launch. For more information on the commands and options in general, see TotalView Server Launch Options and Commands on page 495 and **tvdsvr** in the *Classic TotalView Reference Guide*.

Changing the Remote Shell Command

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

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

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

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

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

After **telnet** establishes a connection to the remote host, you can use the **cd** and **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 computer 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 **ssh** and **rsh** commands, see the manual page supplied with your operating system.

For more information on editing server launch commands, see Customizing Server Launch Commands on page 498.

Changing Arguments

You can also change the command-line arguments passed to **ssh** (TotalView passes **-x** by default), or whatever command you use to invoke the remote process.

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

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

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

Each time TotalView launches **tvdsvr**, a new **xterm** opens to handle standard input and output for the remote program.

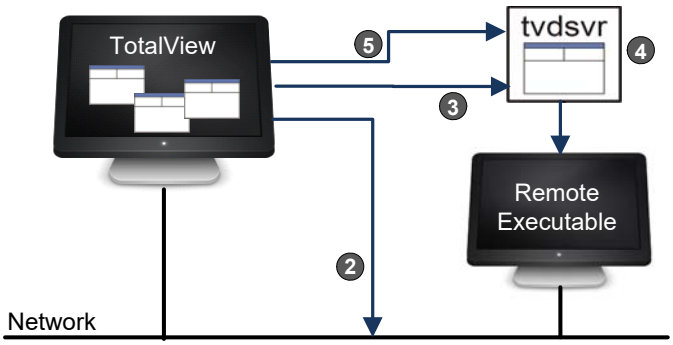
Autolaunching Sequence

This section describes the actions involved in autolaunching. This information is provided to help you troubleshoot autolaunching issues.

1. With the **File > Debug New Program** or **dload** commands, specify the host name of the computer on which you want to debug a remote process, as described in Starting the TotalView Server Manually on page 492.
2. TotalView begins listening for incoming connections.
3. TotalView launches the **tvdsvr** process with the server launch command. (See Setting the Single-Process Server Launch Command on page 498.)
4. The **tvdsvr** process starts on the remote computer.
5. The **tvdsvr** process establishes a connection with TotalView.

Figure 236 illustrates a single server launch. The numbers in the diagram refer to the numbers in the preceding procedure.

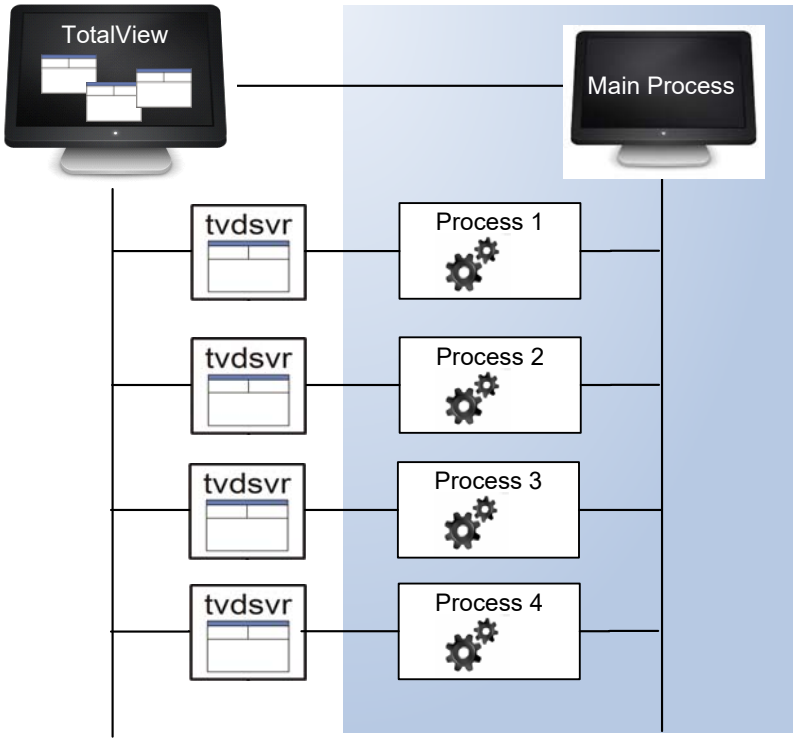
Figure 236, Launching tvdsvr



2. Listens
3. Invokes commands
4. tvdsvr starts
5. Makes connection

If you have more than one server process, Figure 237 shows what your environment might look like:

Figure 237, Multiple tvdsvr Processes



Starting the TotalView Server Manually

In some cases, TotalView is unable to automatically launch the TotalView Server on the remote host, and you will need to manually start the server.

NOTE: You cannot debug remote processes using TotalView Individual.

If TotalView can't automatically launch **tvdsvr**, start it manually:

- Disable both bulk launch and single server launch, set in the **File > Preferences** dialog box
- Enter a host name and port number into the relevant Sessions Manager window (see Automatically Launching a Process on a Remote Server on page 487 for where this is located on the various dialogs). 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.

For information on all the ways to start TotalView, see Starting TotalView on page 89.

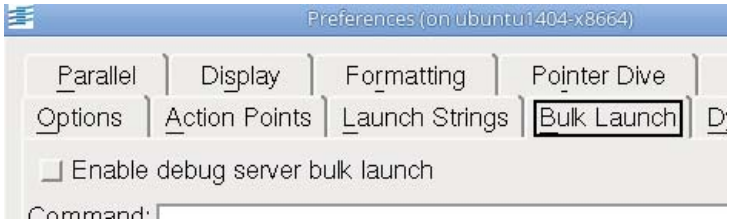
NOTE: Some parallel programs — MPI programs, for example — make use of a starter program such as *poe* or *mpirun* to create all the parallel jobs on your nodes. TotalView lets you start these programs in two ways. One requires that the starter program be under TotalView control, and the other does not.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

Here are the steps in detail to manually start **tvdsvr**:

1. Disable both bulk launch and single server launch, set in the **File > Preferences** dialog box from either the Root Window or the Process Window.

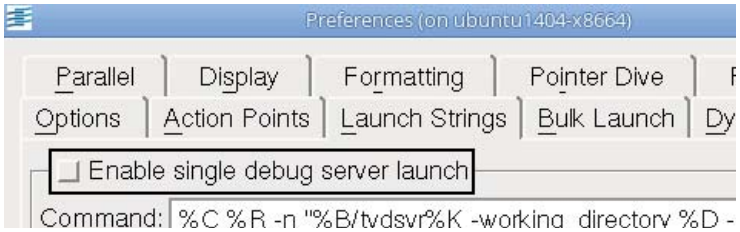
   NOTE: Bulk and single server launch options are discussed in detail in Server Launch Options on page 495.

- To disable bulk launch, select the Bulk Launch Tab and clear the **Enable debug server bulk launch** check box.



CLI: dset TV::bulk_launch_enabled

- To disable single server bulk launch, select the Launch Strings Tab and clear the **Enable single debug server launch** check box.



CLI: dset TV::server_launch_enabled

2. Log in to the remote computer and start **tvdsvr**:

   tvdsvr -server

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

   After printing the port number and the assigned password, the server begins listening for connections. Be sure to note the password, which must be entered in Step 3.

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

3. From the Root Window, select the **File > Debug New Program** command (or any other type of debugging session). Enter the program's name in the **File Name** field and the *hostname*:*portnumber* in the **Debug On Host > Add Host** dialog, and then select **OK**.

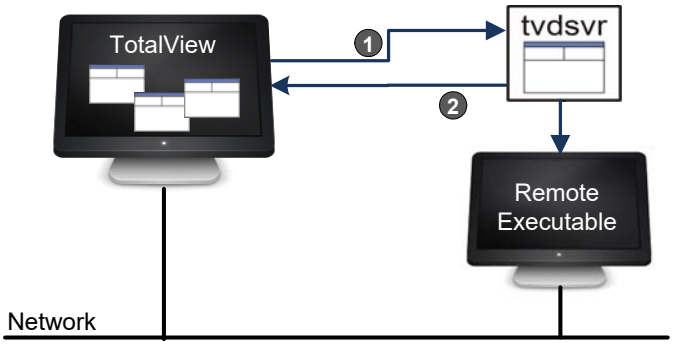
   CLI: dload *executable* **-r** *hostname*

   TotalView tries to connect to **tvdsvr**.

4. Enter the password at the prompt.

Figure 238 summarizes the steps for starting **tvdsvr** manually.

Figure 238, Manual Launching of Debugger Server



1. Makes connection
2. Listens

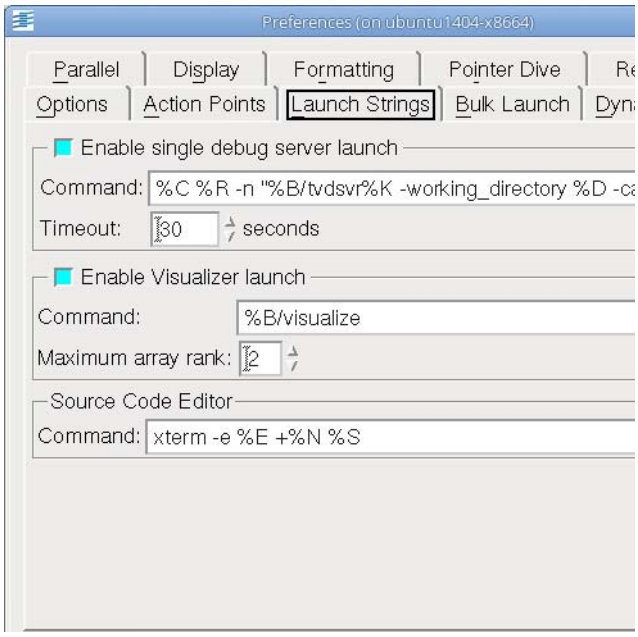
TotalView Server Launch Options and Commands

Server Launch Options

Setting Single-Process Server Launch Options

Use the **Enable single debug server launch** check box in the Launch Strings Page of the **File > Preferences** dialog box to disable autolaunching, change the command that TotalView uses to launch remote servers, and alter the amount of time TotalView waits when establishing connections to a **tvdsvr** process. (The **Enable Visualizer launch** and **Source Code Editor** areas are not used when setting launch options.)

Figure 239, File > Preferences: Launch Strings Page



Enable single debug server launch

Independently launches the **tvdsvr** on each remote system.

CLI: dset TV::server_launch_enabled

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

Command

The command to use when independently launching **tvdsvr**. For information on this command and its options, see TotalView Server Launch Options and Commands on page 495.

CLI: dset TV::server_launch_string

Timeout

The time TotalView waits for a connection after automatically launching the **tvdsvr** process. The default is 30 seconds. If the connection isn't made in this time, TotalView times out. Change the length of time by entering a value from 1 to 3600 seconds (1 hour).

CLI: dset TV::server_launch_timeout

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

Defaults

Reverts to the default settings.

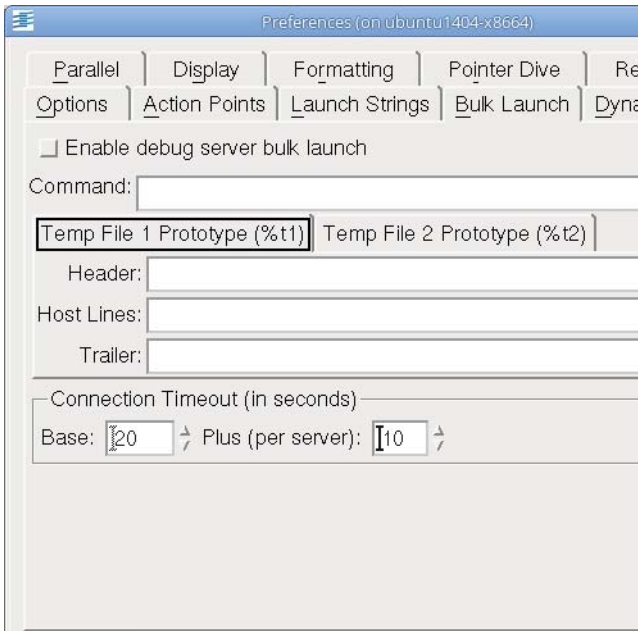
Clicking the **Defaults** button also discards all changes you made using a CLI variable. TotalView doesn't immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Setting Bulk Launch Window Options

Use the **File > Preferences** Bulk Launch Page to change the bulk launch command, disable bulk launch, and alter connection timeouts that TotalView uses when it launches **tvdsvr** programs.

CLI: dset TV::bulk_launch_enabled

Figure 240, File > Preferences: Bulk Launch Page



Enable debug server bulk launch

Uses the bulk launch procedure when launching the **tvdsvr**. By default, bulk launch is disabled; that is, TotalView uses its single-server launch procedure.

Command

Command used to launch **tvdsvr** if bulk launch is enabled. For information on this command and its options, see Setting the Bulk Server Launch Command on page 500 and IBM RS/6000 AIX on page 501.

CLI: dset TV::bulk_launch_string

Temp File 1 Prototype
Temp File 2 Prototype

Specifies the contents of temporary files that the bulk launch operation uses. For information on these fields, see "TotalView Debugger Server (tvdsvr) Command Syntax" in the *Classic TotalView Reference Guide*.

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

Connection Timeout (in seconds)

Sets the connection timeout TotalView uses after launching **tvdsvr** processes. The default is 20 seconds for responses from the process (the **Base** time) plus 10 seconds for each server process being launched.

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

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

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

Defaults

Returns to the default settings.

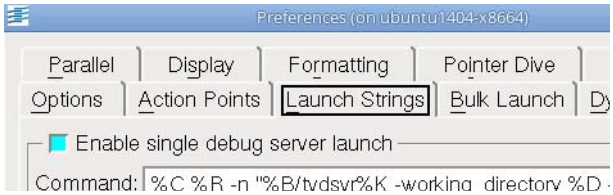
Clicking **Defaults** also discards any changes made using a CLI variable. TotalView doesn't immediately change settings after you click the **Defaults** button; instead, it waits until you click the **OK** button.

Customizing Server Launch Commands

If autolaunch is not working on your system, you may wish to check the default commands set for launching the TotalView Server on your system. You can edit customize these for both single and bulk server launch.

Setting the Single-Process Server Launch Command

You can customize the default command string that TotalView uses when it automatically launches TotalView server for a single process. This string is accessible via the **File >Preferences > Launch Strings** dialog in its Command text box:



This is the default command string:

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

where:

%C Expands to the name of the server launch command to use, which is the value of **TV::launch_command**. On most platforms, this is **ssh -x**. On Sun SPARC computers, it is **rsh**. If the **TVDSVRLAUNCHCMD** environment variable exists, **TV::launch_command** is initialized to its value.

%R Expands to the host name of the remote computer specified in the **File > Debug New Program** (and other Session Manager dialog boxes) or **dload** commands.

%B Expands to the **bin** directory in which **tvdsvr** is installed.

-n Tells the remote shell to read standard input from **/dev/null**; that is, the process immediately receives an EOF (End-Of-File) signal.

-**working_directory %D**

Makes **%D** the directory to which TotalView connects. **%D** expands to the absolute path name of the directory.

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

After changing to this directory, the shell invokes the **tvdsvr** command.

You must make sure that the **tvdsvr** directory is in your path on the remote computer.

-callback %L Establishes a connection from **tvdsvr** to TotalView. **%L** expands to the host name and TCP/IP port number (*hostname*:*portnumber*) on 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. TotalView expands **%P** to the password that it automatically generates. For more information on this password, see "TotalView Debugger Server (tvdsvr) Command Syntax" in the *Classic TotalView Reference Guide*.

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

%F Contains the tracer configuration flags that need to be sent to **tvdsvr** processes. These are system-specific startup options that the **tvdsvr** process needs.

You can also use the **%H** option with this command. See Setting the Bulk Server Launch Command on page 500 for more information.

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

Setting the Bulk Server Launch Command

The commands for bulk server launch settings vary according to platform.

SGI XE and SGI ICE

The bulk server launch string is as follows:

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

where:

-working_directory %D
: Specifies the directory to which TotalView connects. TotalView expands **%D** to this directory's absolute path name.

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

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

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

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

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

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

```
#
```

```
# Command that allows invocation of the TotalView
# Debugger server when performing a Bulk Server Launch.
#
command tvdsvr
    invoke /opt/totalview/bin/tvdsvr %ALLARGS
    user %USER
    group %GROUP
    project %PROJECT
```

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

Cray XT/XE/XK/XC Series

NOTE: Bulk server launch is not used when MRNet is enabled in TotalView, which is the default for Cray.

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

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

where the options unique to this command are:

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

IBM RS/6000 AIX

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

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

where the options unique to this command are:

%N The number of servers that TotalView launches.

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

TotalView generates this information by expanding the **%R** symbol. This is the information you enter in the **Temp File 2 Prototype** field on the Bulk Launch Page.

%t2 A file that contains the commands to start the **tvdsvr** processes on each computer. 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 *Classic TotalView Reference Guide*.

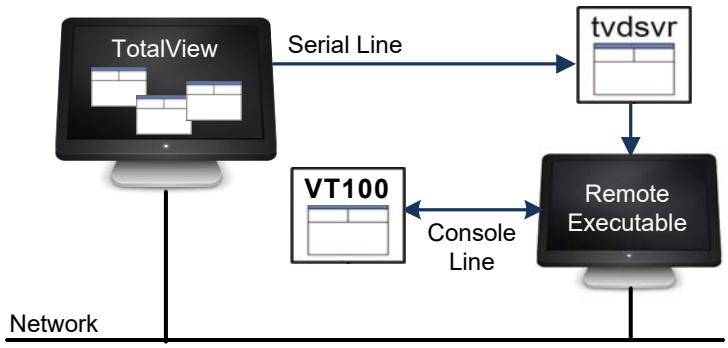
Debugging Over a Serial Line

TotalView supports debugging programs over a serial line as well as TCP/IP sockets. However, if a network connection exists, use it instead to improve performance.

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

Figure 241 illustrates a TotalView session using a serial line. In this example, TotalView is communicating over a dedicated serial line with a **tvdsvr** 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 241, Debugging Session Over a Serial Line



Starting the TotalView Debugger Server

To start a debugging session over a serial line, first start the **tvdsvr** from the command line.

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

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.

Reverse Connections

The organization of modern HPC systems often makes it difficult to deploy tools such as TotalView. For example, the compute nodes in a cluster may not have access to any X libraries or X forwarding, so launching a GUI on a compute node is not possible.

Using the Reverse Connect feature, you can run the TotalView UI on a front-end node to debug a job executing on compute nodes.

The basic process is to embed the **tvconnect** command in a batch script; when the batch job runs, the **tvconnect** process connects with the TotalView client to start the debugger server process on the batch node. The TotalView client would typically run on a front-end node, where the application is built and batch jobs are submitted.

About Reverse Connections

When using reverse connect, TotalView is started in two stages:

1. Run the **tvconnect** command to create a debugging request, typically from a batch job on a batch node or compute node in a cluster. The **tvconnect** command accepts the name of the program to debug, along with any arguments to pass to the program.

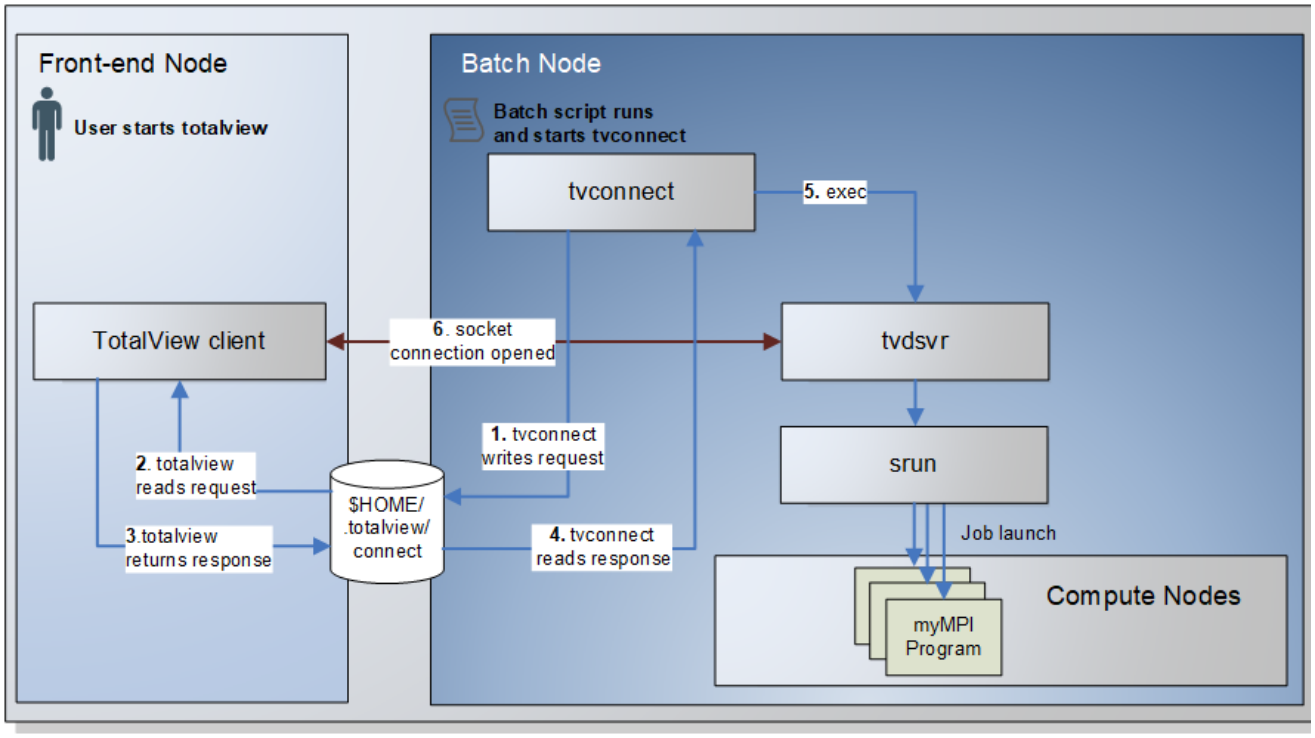
   The **tvconnect** process blocks for a TotalView session to accept the request.

2. Start TotalView on another node, which is typically a front-end node. When the UI opens, TotalView looks for a request, and if it finds one, confirms via a pop-up that the user wants to accept it. If the request is accepted, TotalView starts a debugger server on the node where the **tvconnect** process is running, and loads the program that was passed to the **tvconnect** command. If the request is rejected, the **tvconnect** process exits with an error.

At that point, debug the program in the normal way within the TotalView UI.

The process works as follows:

Figure 242, Reverse connect flow



Typically, a **tvconnect** command is added to a batch script, placed in front of the command to debug. For example:

```
tvconnect srun -n4 myMPIprogram
```

Once a batch script runs and starts the **tvconnect** command and a TotalView front-end UI is started:

1. The **tvconnect** command creates a request in the `$HOME/.totalview/connect` directory and blocks indefinitely until the request is either accepted or rejected. If the **tvconnect** process is killed with a **SIGINT** or **SIGTERM**, the **tvconnect** process deletes the request it created.
2. TotalView reads the request file written by the **tvconnect** process.
3. TotalView accepts or rejects the request, sending back a response.
4. **tvconnect** reads the response. If it was accepted:
5. **tvconnect** execs **tvdsvr**, the command that allows TotalView to control and debug a program on a remote machine.
6. **tvdsvr** opens a connection to the TotalView UI. TotalView then loads the program and any program arguments, using the parameters provided to **tvconnect**. In this example, **srun** was loaded to debug an MPI job.

NOTE: TotalView does not look for reverse connect requests once it starts to debug a program, i.e., it automatically listens only if no other debug session is active. You can choose, however, to continue to listen for connection requests while debugging. See Listening for Reverse Connections.

Reverse connections are also supported by the CLI **dload** command, which has options to either accept or reject reverse connections. In addition, some command line arguments and special environment variables are available that can be used to modify some behavior.

RELATED TOPICS	
dload command's reverse connect options	**-list_reverse_connect** in the *TotalView Reference Guide* **-reject_reverse_connect** in the *TotalView Reference Guide* **-accept_reverse_connect** in the *TotalView Reference Guide*
Environment variables specific to reverse connections	Reverse Connection Environment Variables
State variable **TV::reverse_connect_wanted**	**TV::reverse_connect_wanted** in the *TotalView Reference Guide*
Command line arguments specific to reverse connections	**-reverse_connect** and **-no_reverse_connect** in the *TotalView Reference Guide*
The **tvdsvr** command	"The tvdsvr Command and Its Options" in the *TotalView Reference Guide*

Reverse Connection Environment Variables

TotalView supports two special reverse-connection specific environment variables:

- **TV_REVERSE_CONNECT_DIR**
- **TVCONNECT_OPTIONS**

TV_REVERSE_CONNECT_DIR

The environment variable **TV_REVERSE_CONNECT_DIR** identifies the directory where the request and response files will be written and read.

The default location is the user's `$HOME/.totalview/connect` directory.

To customize the location for your reverse connection files, set this environment variable before starting **tvconnect** and TotalView:

```
setenv TV_REVERSE_CONNECT_DIR /home/tv-reverse-connect/tmp
```

Reverse Connection Directory Requirements

The directory that will contain the generated reverse connect files must:

- Be owned by the same user that is running the **tvconnect** process and the TotalView client.
- Have permissions that allow access only by the user. No "Group" or "Other" permissions are allowed.

By default, **tvconnect** creates the connect directory with the following permissions:

```
>ls -l ~/.totalview/
total 80
drwx------  2 smith tss  4096 Jul 23 12:11 connect
```

TV_CONNECT_OPTIONS

The environment variable **TVCONNECT_OPTIONS** supports the ability to add extra arguments to the **tvconnect** command. One such option might be `-ipv6_support`, which adds support for IPv6 addresses. For example:

```
setenv TVCONNECT_OPTIONS="-ipv6_support"
tvconnect ~/tx_hello
```

or just:

```
env TVCONNECT_OPTIONS="-ipv6_support" tvconnect ~/tx_hello
```

Starting a Reverse Connect Session

1. Run the **tvconnect** command on a "back-end" compute node. This node does not need access to X libraries to launch a UI. Provide as an argument the program to debug. For example, at its most simple:

   ```
   tvconnect /home/totalview/tests/myTest
   ```

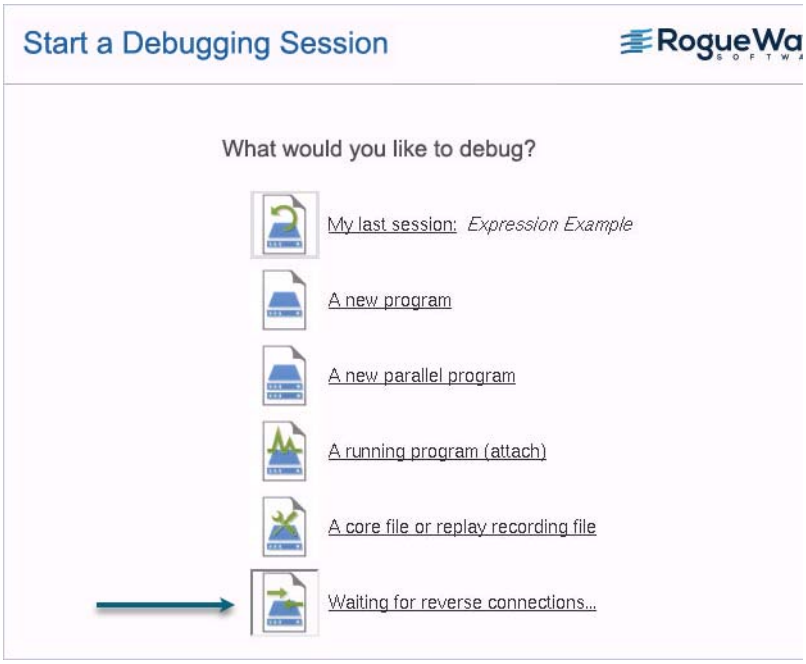
   This command creates a request file in the user's `$HOME/.totalview/connect` directory. The file contains all the information needed to launch a debugging session on a "front-end" where TotalView is installed with UI capabilities. The request includes such things as the remote host name, the IP address, the user's home directory, and other information required to launch the debugging session.

   This command then blocks waiting for a response.

2. Start TotalView with no arguments on the server where you will perform debugging:

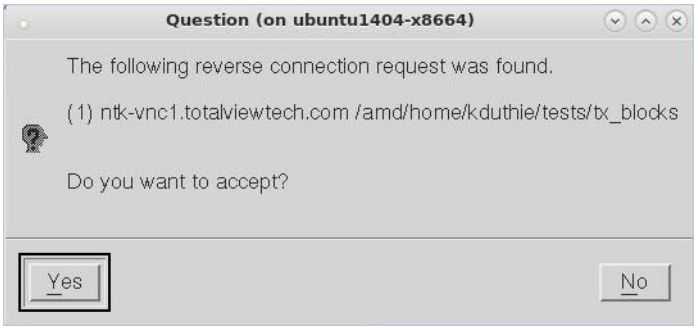
   ```
   totalview
   ```

   When TotalView launches, it automatically listens for reverse connection requests:



NOTE: TotalView automatically listens for connections at launch *only when invoked without specifying a debug target*. If TotalView starts debugging a different program, it does not listen for reverse connections.

If one or more requests are found, it then launches a pop-up to confirm that you want to accept the reverse debugging request:



If you select "No," the back-end **tvconnect** stops waiting and exits with the following error message:

"Reverse connect request was rejected."

If you select "Yes," your program to debug launches in the usual way, and you can start your debugging session.

Note that the UI displays the node where TotalView is running in parentheses in the Process Window's title bar:

Listening for Reverse Connections

Once you start a debugging session, TotalView automatically *stops* listening for reverse connection requests.

To turn back on listening mode:

1. From the File menu, choose **New Debugging Session** to open the Start a Debugging Session window.
2. Select **Accept reverse connections**:

Reverse Connect Examples

Initiate a reverse connection request by specifying any debug target program as the argument to **tvconnect**. The specified program must be accessible by the TotalView front-end UI that wishes to accept the request.

Here's a simple example:

```
tvconnect /home/totalview/tests/myTest
```

To start it on an MPI job, for example:

```
tvconnect srun -n 4 /home/fullpath/tx_mpi_test
```

CLI Example

This example illustrates the usage of the reverse connection **dload** options.

Assume that:

- **tvconnect** was started on machine1 specifying a program with a full path
- **tvconnect** was started on machine2 specifying the program **tx_hello**
- The program was in the current working directory and no path was added to the program.

```
d1.<>  dload -list_reverse_connect
(1) machine1.totalviewtech.com /home/user/tests/tx_blocks
(2) machine2.totalviewtech.com tx_hello

d1.<> dload -accept_rc 1
d1.<> dload -reject_rc 2
```

MPI Batch Script Example

This is a simple example for invoking an MPI job from a batch script.

1. Create your job script. For example, create a batch script containing the following:

```
#-----------------------
#!/bin/tcsh
#SBATCH -p pdebug
#SBATCH -J myJob
#SBATCH -N 2
# Wait for a front end TV to accept this reverse connection request
tvconnect srun -n4 myMPIprogram
echo 'DONE!'
#-----------------------
```

Once the script is run, the **tvconnect** command creates the request file with the necessary details, then holds and waits for a connection request.

2. Start TotalView on your front-end node and accept the request. The debugger begins debugging **srun**. Pressing **Go** starts the MPI job and TotalView will attach to the MPI processes running on the compute nodes in the normal way.

NOTE: If the application is in your system path, TotalView will find it. i.e., you do not need to enter the full path in your command.

it is not required that you include the full path to your application in the command, if the application is in your path. In the above case, `myMPIprogram` was in the current working directory when the batch job was submitted. The request file reports the current working directory, so that the front-end TotalView can find the application even if it was not started from same directory.

MPI Batch Script Example

This is a simple example for invoking an MPI job from a batch script.

1. Create your job script. For example, create a batch script containing the following:

```
#-----------------------
#!/bin/tcsh
#SBATCH -p pdebug
#SBATCH -J myJob
#SBATCH -N 2
# Wait for a front end TV to accept this reverse connection request
tvconnect srun -n4 myMPIprogram
echo 'DONE!'
#-----------------------
```

Once the script is run, the **tvconnect** command creates the request file with the necessary details, then holds and waits for a connection request.

2. Start TotalView on your front-end node and accept the request. The debugger begins debugging **srun**. Pressing **Go** starts the MPI job and TotalView will attach to the MPI processes running on the compute nodes in the normal way.

NOTE: If the application is in your system path, TotalView will find it. i.e., you do not need to enter the full path in your command.

it is not required that you include the full path to your application in the command, if the application is in your path. In the above case, `myMPIprogram` was in the current working directory when the batch job was submitted. The request file reports the current working directory, so that the front-end TotalView can find the application even if it was not started from same directory.

Troubleshooting Reverse Connections

Most of the issues that can result in a failed reverse connection have to do with the reverse connect directory where TotalView writes and reads connection requests.

Stale Files in the Reverse Connect Directory

In some cases, TotalView may leave stale request or response files in the reverse connect directory, which could result in a failed connection attempt.

If you have no pending reverse connect requests, it is safe to remove the entire directory or its contents. For example, use `rm -rf $HOME/.totalview/connect` to remove the default directory and all its files. The **tvconnect** process will recreate the directory when needed.

Directory Permissions

You must be the owner of the reverse connect directory, and its permissions must allow "user" access only. The directory cannot be owned by a different user, and cannot have any "group" or "other" permission bits set.

User ID Issues

The **tvconnect** process and the TotalView client must be running with the same effective user id (`euid`). The `euid` of the processes run by the client must match that of the reverse connect directory.

Reverse Connect Directory Environment Variable

If the TotalView client fails to find a **tvconnect** request, make sure that the **tvconnect** process is writing its request file to the same directory being read by the TotalView client.

For example, the client may read the wrong directory if your home directory on the node where **tvconnect** is running is different than the node where the TotalView client is running.

To work around this problem, set the environment variable TV_REVERSE_CONNECT_DIR.

When setting this variable, make sure that it points to a directory accessible by both the **tvconnect** and TotalView client processes, and that it meets all the ownership and permission requirements.

Setting Up MPI Debugging Sessions

This chapter discusses how to set up TotalView MPI debugging sessions for various environments and special use cases, as well as some application-specific debugging tasks. In most cases, you can just use the basic procedure, discussed in Starting MPI Programs Using File > Debug New Parallel Program on page 518.

For information on setting up non-MPI parallel programs, see Setting Up Parallel Debugging Sessions on page 546.

NOTE: For TotalView Individual, all your MPI processes must execute on the computer on which you installed TotalView. Further, you are limited to no more than 16 processes and threads.

This chapter describes the basics on setting up to debug an MPI system (Debugging MPI Programs on page 518), as well as the following MPI systems:

- MPICH Applications on page 523
- MPICH2 Applications on page 528
- Cray MPI Applications on page 531
- IBM MPI Parallel Environment (PE) Applications on page 532
- Open MPI Applications on page 536
- QSW RMS Applications on page 537

This chapter also includes these topics specific to MPI applications:

- Starting MPI Issues on page 542
- Using ReplayEngine with Infiniband MPIs on page 544

RELATED TOPICS	
Tips for debugging parallel applications	Debugging Strategies for Parallel Applications on page 437
Tools for displaying an MPI Message Queue	MPI Display Tools on page 445
Creating startup profiles for environments not defined by TotalView. These definitions will appear in the Additional Starter Arguments field of the Debug New Parallel Program dialog box.	"MPI Startup" in the *Classic TotalView Reference Guide*

Debugging MPI Programs

Starting MPI Programs

MPI programs use a starter program such as **mpirun** to start your program. You can start these MPI programs in two ways: with the starter program under TotalView control, or using the GUI, in which case the starter program is not under TotalView control. In the first case, you will enter the name of the starter program on the command line. In the latter, you will enter program information into the **File > Debug New Parallel Program** or **Process > Startup Parameters** dialog boxes.

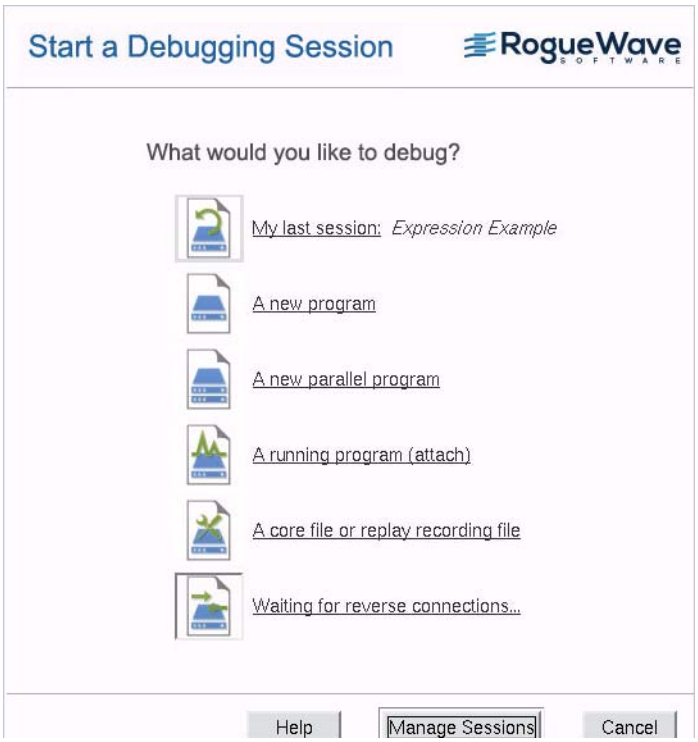
NOTE: Programs started using GUI dialog boxes have some limitations: program launch does not use the information you set for single-process and bulk server launching, and you cannot use the Attach Subset command.

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

Starting MPI Programs Using File > Debug New Parallel Program

In many cases, the way in which you invoke an MPI program within TotalView control differs little from discipline to discipline. If you invoke TotalView from the command line without an argument, TotalView displays its Start a Debugging Session dialog box. This is the same as choosing **File > New Debugging Session** from either the Root or Process windows.

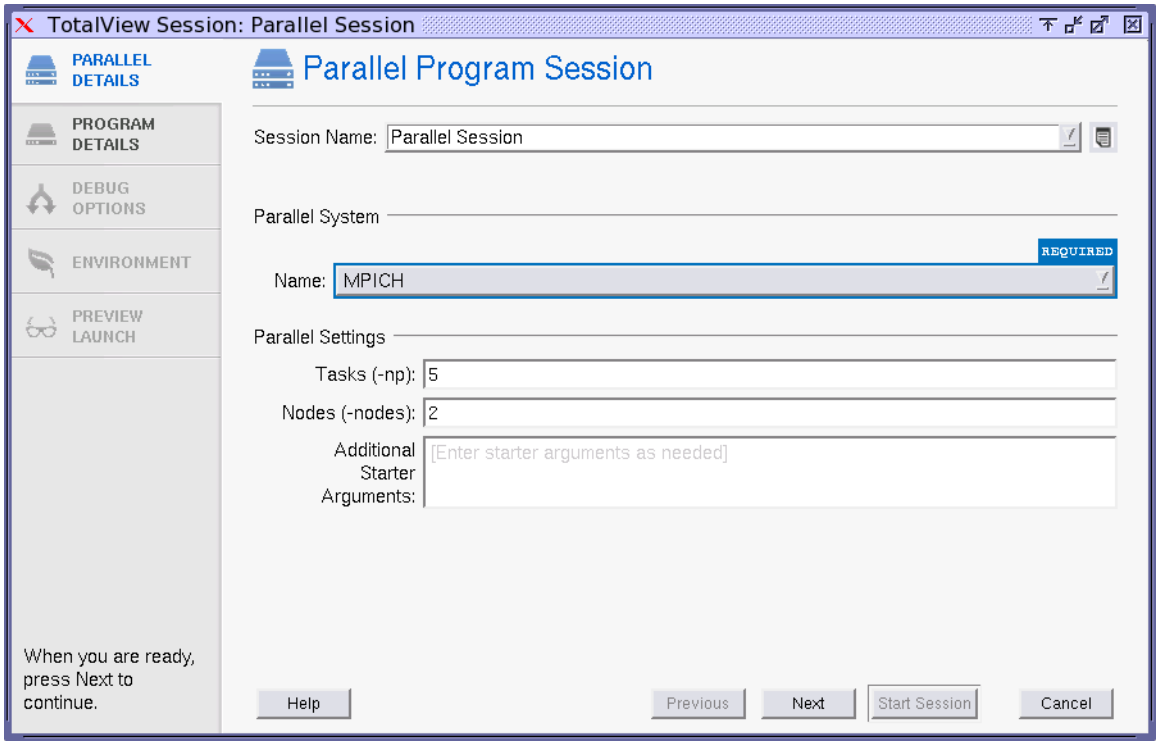
Figure 243, Start a Debugging Session dialog



From here, select **A new parallel program**. Alternatively, if TotalView is already running, choose **File > Debug New Parallel Program** from the Root or Process window. Both launch the Parallel Program Session dialog.

The Parallel Program Session Dialog

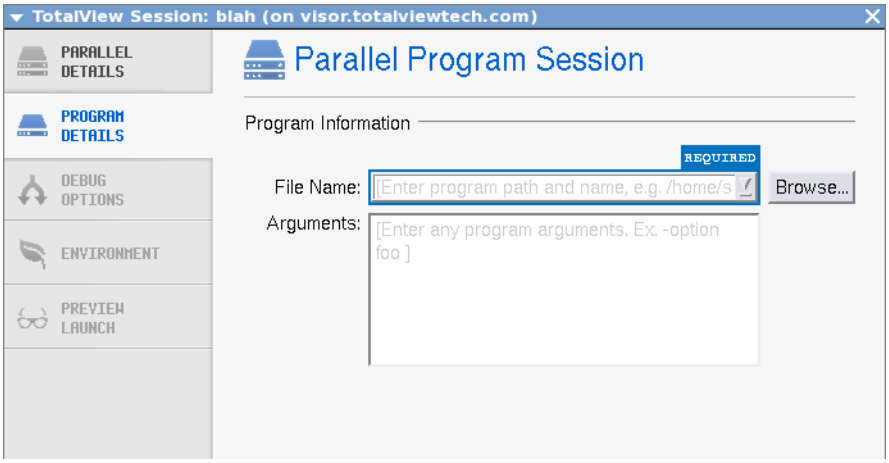
Figure 244, Parallel Program Session dialog



1. Enter a session name in the **Session Name** field.

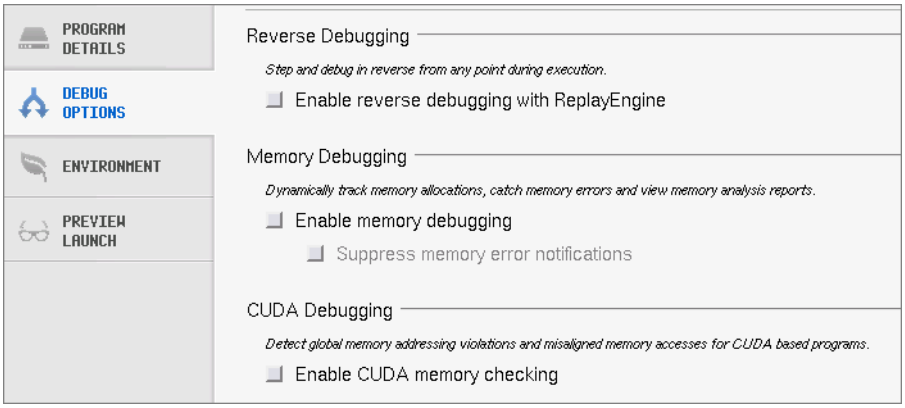
> **NOTE:** Any previously entered sessions of the same type are available from the Session Name dropdown box. Once selected, you can change any session properties and start your debug session. See Editing or Starting New Sessions in a Sessions Window on page 126.

2. Select the **Parallel** system, the number of **Tasks**, and **Nodes**.
3. (*Optional*) Enter any additional arguments required by the starter process into the **Arguments** area. Note that these arguments are those sent to a starter process such as **mpirun** or **poe**. They are not arguments sent to your program.
4. Select the **Program Details** tab to enter the file name of the program being debugged and any arguments to be sent to your program.

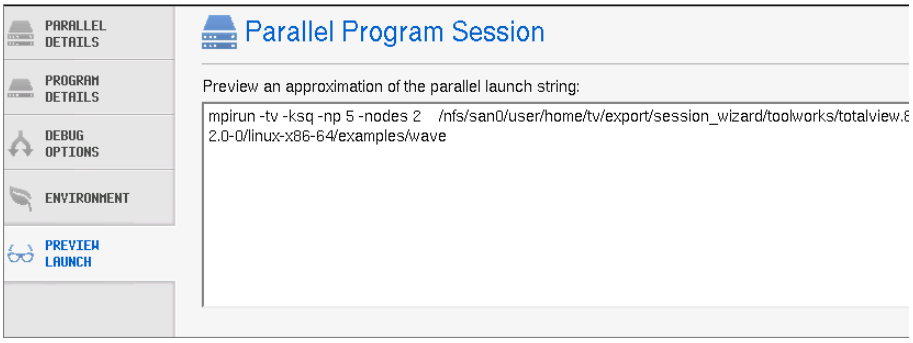


5. Select any optional settings:

   - Select **Debug Options** to enable reverse, memory or CUDA debugging. See Options: Reverse Debugging, Memory Debugging, and CUDA on page 118.



   - Select the **Environment** tab to add or initialize environment variables or customize standard I/O. See Setting Environment Variables and Altering Standard I/O on page 120.
   - Select the **Preview Launch** tab to view the launch string TotalView will use to open your debugging session.



6. Select the **Start Session** button to launch the TotalView.

Once created, a session named `my_foo` can be quickly launched later using the `-load` command line option, like so:

```
totalview -load_session my_foo
```

MPICH Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

To debug Message Passing Interface/Chameleon Standard (MPICH) applications, you must use MPICH version 1.2.3 or later on a homogeneous collection of computers. If you need a copy of MPICH, you can obtain it at no cost from Argonne National Laboratory at http://www.mpich.org/downloads/. (We strongly urge that you use a later version of MPICH. For versions that work with TotalView, see *TotalView Supported Platforms* document in the TotalView distribution at `<installdir>/totalview.<version>/doc/pdf` or TotalView Supported Platforms on the TotalView documentation website.)

The MPICH library should use the **ch_p4**, **ch_p4mpd**, **ch_shmem**, **ch_lfshmem**, or **ch_mpl** devices.

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

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

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

For more information on MPICH applications, see MPICH Debugging Tips on page 451.

Starting TotalView on an MPICH Job

Before you can bring an MPICH job under TotalView's control, both TotalView and the **tvdsvr** must be in your path, most easily set in a login or shell startup script.

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

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

For example:

```
mpirun -np 4 -tv sendrecv
```

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

For Version 1.2.4, the syntax changes to the following:

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

For example:

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

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

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

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

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

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

On the IBM SP computer with the **ch_mpl** device, the **mpirun** command uses the **poe** command to start an MPI job. While you still must use the MPICH **mpirun** (and its **-tv** option) command to start an MPICH job, the way you start MPICH differs. For details on using TotalView with **poe**, see Starting TotalView on a PE Program on page 533.

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

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

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

As your program executes, TotalView automatically acquires the processes that are part of your parallel job as your program creates them. Before TotalView begins to acquire them, it asks if you want to stop the spawned processes. If you click **Yes**, you can stop processes as they are initialized. This lets you check their states or set

breakpoints that are unique to the process. TotalView automatically copies breakpoints from the master process to the slave processes as it acquires them. Consequently, you don't have to stop them just to set these breakpoints.

If you're using the GUI, TotalView updates the Root Window to show these newly acquired processes. For more information, see Attaching to Processes Tips on page 440.

Attaching to an MPICH Job

You can attach to an MPICH application even if it was not started under TotalView control. To attach to an MPICH application:

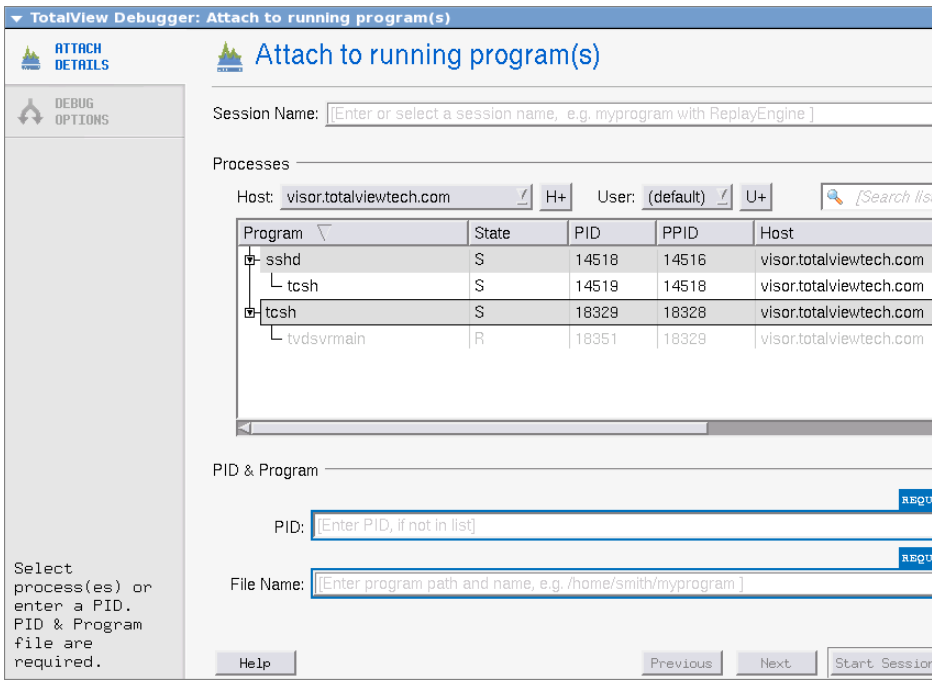
1. Start TotalView.

   Select **A running program (attach)** on the Start a Debugging Session dialog. A list of processes running on the selected host displays in the **Attach to running program(s)** dialog.

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

   CLI: dattach *executable pid*

3. On an IBM SP with the **ch_mpi** device, attach to the **poe** process that started your job. For details, see Starting TotalView on a PE Program on page 533.



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

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

4. After attaching to your program's processes, a dialog launches where you can choose to also attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

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

   As an alternative, you can use the **Group > Attach Subset** command to predefine what TotalView should do. For more information, see Attaching to Processes Tips on page 440.

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

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

Using MPICH P4 procgroup Files

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

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

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

In this example, TotalView does the following:

1. Reads the symbols from **mympichexe** only once.

2. Places MPICH processes in the same TotalView share group.
3. Names the processes **mypichexe.0**, **mympichexe.1**, **mympichexe.2**, and **mympichexe.3**.

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

MPICH2 Applications

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

Downloading and Configuring MPICH2

You can download the current MPICH2 version from:

http://www.mpich.org/downloads/versions/

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

- -enable-debuginfo

or

- - -enable-totalview

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

python2.x/config/Makefile

It fails if the file is not there.

The next steps are:

1. Run **make**
2. Run **make install**

   This places the binaries and libraries in the directory specified by the optional **- -prefix** option.
3. Set the PATH and LD_LIBRARY_PATH to point to the MPICH2 **bin** and **lib** directories.

Starting TotalView Debugging on an MPICH2 Hydra Job

As of MPICH2 1.4.1, the default job type for MPICH2 is Hydra. If you are instead using MPD, see Starting TotalView Debugging on an MPICH2 MPD Job on page 529.

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

Start a Hydra job as follows:

totalview -args mpiexec *mpiexec-args program program-args*

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

Starting TotalView Debugging on an MPICH2 MPD Job

You must start the **mpd** daemon before starting an MPICH2 MPI job.

NOTE: As of MPICH2 1.4.1, the default job type is Hydra, rather than MPD, so if you are using the default, there is no need to start the daemon. See Starting TotalView Debugging on an MPICH2 Hydra Job on page 528.

Starting the MPI MPD Job with MPD Process Manager

To start the mpd daemon, use the **mpdboot** command. For example:

```
mpdboot -n 4 -f hostfile
```

where:

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

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

You are now ready to start debugging your application.

Starting an MPICH2 MPD Job

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

Start an MPICH2 MPD job in one of the following ways:

mpiexec *mpi-args* -tv *program* -a *program-args*

This command tells MPI to start TotalView. You must have set the TOTALVIEW environment variable with the path to TotalView's executable when you start a program using **mpiexec**. For example:

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

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

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

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

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

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

Cray MPI Applications

In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518

Specific information on debugging Cray MPI applications is located in our discussion of running TotalView on Cray platforms. See Debugging Cray XT/XE/XK/XC Applications on page 554 for information.

IBM MPI Parallel Environment (PE) Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

You can debug IBM MPI Parallel Environment (PE) applications on the IBM RS/6000 and SP platforms.

To take advantage of TotalView's ability to automatically acquire processes, you must be using release 3,1 or later of the Parallel Environment for AIX.

Topics in this section are:

Preparing to Debug a PE Application

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

Using Switch-Based Communications

If you're using switch-based communications (either *IP over the switch* or *user space*) on an SP computer, configure your PE debugging session so that TotalView can use *IP over the switch* for communicating with the TotalView Server (**tvdsvr**). Do this by setting the **-adapter_use** option to **shared** and the **-cpu_use** option to **multiple**, as follows:

- If you're using a PE host file, add **shared multiple** after all host names or pool IDs in the host file.
- Always use the following arguments on the **poe** command line:
  ```
  -adapter_use shared -cpu_use multiple
  ```

If you don't want to set these arguments on the **poe** command line, set the following environment variables before starting **poe**:

```
setenv MP_ADAPTER_USE shared
setenv MP_CPU_USE multiple
```

When using *IP over the switch*, the default is usually **shared adapter use** and **multiple cpu use**; we recommend that you set them explicitly using one of these techniques. You must run TotalView on an SP or SP2 node. Since TotalView will be using *IP over the switch* in this case, you cannot run TotalView on an RS/6000 workstation.

Performing a Remote Login

You must be able to perform a remote login using the **ssh** command. You also need to enable remote logins by adding the host name of the remote node to the **/etc/hosts.equiv** file or to your **.rhosts** file.

When the program is using switch-based communications, TotalView tries to start the TotalView Server by using the **ssh** command with the switch host name of the node.

Setting Timeouts

If you receive communications timeouts, you can set the value of the **MP_TIMEOUT** environment variable; for example:

```
setenv MP_TIMEOUT 1200
```

If this variable isn't set, TotalView uses a **timeout** value of 600 seconds.

Starting TotalView on a PE Program

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

program [*arguments*] [*pe_arguments*]

You can also use the **poe** command to run programs as follows:

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

If, however, you start TotalView on a PE application, you must start **poe** as TotalView's target using the following syntax:

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

For example:

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

Setting Breakpoints

After TotalView is running, start the **poe** process using the **Process > Go** command.

CLI: dfocus p dgo

A dialog box launches in the GUI —in the CLI, it prints a question—to determine if you want to stop the parallel tasks.

If you want to set breakpoints in your code before they begin executing, answer **Yes**. TotalView initially stops the parallel tasks, which also allows you to set breakpoints. You can now set breakpoints and control parallel tasks in the same way as any process controlled by TotalView.

If you have already set and saved breakpoints with the **Action Point > Save All** command, and you want to reload the file, answer **No**. After TotalView loads these saved breakpoints, the parallel tasks begin executing.

CLI: dactions -save *filename*
dactions -load *filename*

Starting Parallel Tasks

After you set breakpoints, you can start all of the parallel tasks with the Process Window **Group > Go** command.

CLI: dfocus G dgo
Abbreviation: G

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

Be very cautious in placing breakpoints at or before a line that calls **MPI_Init()** or **MPL_Init()** because timeouts can occur while your program is being initialized. After you allow the parallel processes to proceed into the **MPI_Init()** or **MPL_Init()** call, allow all of the parallel processes to proceed through it within a short time. For more information on this, see IBM PE Debugging Tips on page 453.

Attaching to a PE Job

To take full advantage of TotalView's **poe**-specific automation, you need to attach to **poe** itself, and let TotalView automatically acquire the **poe** processes on all of its nodes. In this way, TotalView acquires the processes you want to debug.

Attaching from a Node Running poe

To attach TotalView to **poe** from the node running **poe**:

1. Start TotalView in the directory of the debug target.

   If you can't start TotalView in the debug target directory, you can start TotalView by editing the **tvdsvr** command line before attaching to **poe**. See Setting the Single-Process Server Launch Command on page 498.

2. In the **File > Attach to a Running Program**, then find the **poe** process list, and attach to it by diving into it. When necessary, TotalView launches **tvdsvr**s. TotalView also updates the Root Window and opens a Process Window for the **poe** process.

   CLI: dattach poe *pid*

3. Locate the process you want to debug and dive on it, which launches a Process Window for it. If your source code files are not displayed in the Source Pane, invoke the **File > Search Path** command to add directories to your search path.

Attaching from a Node Not Running poe

The procedure for attaching TotalView to **poe** from a node that is not running **poe** is essentially the same as the procedure for attaching from a node that is running **poe**. Since you did not run TotalView from the node running **poe** (the startup node), you won't be able to see **poe** on the process list in the Root Window and you won't be able to start it by diving into it.

To place **poe** in this list:

1. Connect TotalView to the startup node. For details, see Starting the TotalView Server Manually on page 492.
2. Select the **File > Attach to a Running Program.**
3. Look for the process named **poe** and continue as if attaching from a node that is running **poe**.

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

Open MPI Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

Open MPI is an open source implementation of both the MPI-1 and MPI-2 documents that combines some aspects of four different (and now no longer under active development) MPI implementations: FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI from the University of Stuttgart.

For more information on Open MPI, see https://www.open-mpi.org/.

Debug an Open MPI program similarly to most MPI programs, using the following syntax if TotalView is in your path:

mpirun -tv *args prog prog_args*

As an alternative, you can invoke TotalView on **mpirun**.

totalview -args mpirun *args ./prog*

For example, to start TotalView on a four-process MPI program:

```
totalview -args mpirun -np 4 ./mpi_program
```

Alternatively, you can use the Session Manager or Startup Parameter window (accessed via **Process > Startup Parameters**) and choose the **Parallel** option to enter the parallel session details in the GUI.

QSW RMS Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

Starting TotalView on an RMS Job

To start a parallel job under TotalView control, use TotalView as if you were debugging **prun**:

{ totalview | totalviewcli } prun -a *prun-command-line*

TotalView starts and shows you the machine code for RMS **prun**. Since you're not usually interested in debugging this code, use the **Process > Go** command to let the program run.

CLI: dfocus p dgo

The RMS **prun** command executes and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

Attaching to an RMS Job

To attach to a running RMS job, attach to the RMS **prun** process that started the job.

You attach to the **prun** process the same way you attach to other processes.

After you attach to the RMS **prun** process, you have the option to attach to slave MPICH processes. If you do, press **Return** or choose **Yes**. If you do not, choose **No**.

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

As an alternative, you can use the **Group > Attach Subset** command to predefine what TotalView should do.

RELATED TOPICS	
Attaching to processes using **prun**	Attaching to a Running Program on page 105
Using the **Group > Attach Subset** command to specify TotalView behavior when attaching to an RMS **prun** process	Attaching to Processes Tips on page 440

SGI MPI Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

TotalView can acquire processes started by SGI MPI applications. This MPI is part of the Message Passing Toolkit (MPT) 1.3 and 1.4 packages. TotalView can display the Message Queue Graph Window for these releases. See Displaying the Message Queue Graph Window on page 446 for message queue display.

Starting TotalView on an SGI MPI Job

You normally start SGI MPI programs by using the **mpirun** command. You use a similar command to start an MPI program under debugger control, as follows:

{ totalview | totalviewcli } mpirun -a *mpirun-command-line*

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

CLI: dfocus p dgo

The SGI MPI **mpirun** command runs and starts all MPI processes. After TotalView acquires them, it asks if you want to stop them at startup. If you answer **Yes**, TotalView halts them before they enter the main program. You can then create breakpoints.

If you set a verbosity level that allows informational messages, TotalView also prints a message that shows the name of the array and the value of the array services handle (**ash**) to which it is attaching.

Attaching to an SGI MPI Job

To attach to a running SGI MPI program, attach to the SGI MPI **mpirun** process that started the program. The procedure for attaching to an **mpirun** process is the same as that for attaching to any other process.

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

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

As an alternative, you can use the **Group > Attach Subset** command to predefine what to do.

RELATED TOPICS	
Attaching to an **mpirun** process	Debugging an MPI Program on page 94
Using the **Group > Attach Subset** command to specify TotalView behavior when attaching to a process	Attaching to Processes Tips on page 440

Using ReplayEngine with SGI MPI

SGI MPI uses the **xpmem** module to map memory from one MPI process to another during job startup. Memory mapping is enabled by default. The size of this mapped memory can be quite large, and can have a negative effect on TotalView's ReplayEngine performance. Therefore, mapped memory is limited by default for the **xpmem** module if Replay is enabled. The environment variable, **MPI_MEMMAP_OFF**, is set to 1 in the TotalView file **parallel_support.tvd** by adding the variable to the replay_env: specification as follows: **replay_env: MPI_MEMMAP_OFF=1**.

If full memory mapping is required, set the startup environment variable in the Arguments field of the Program Session dialog. Add the following to the environment variables: **MPI_MEMMAP_OFF=0.**

Be aware that the default mapped memory size may prove to be too large for ReplayEngine to deal with, and it could be quite slow. You can limit the size of the mapped heap area by using the **MPI_MAPPED_HEAP_SIZE** environment variable documented in the SGI documentation. After turning off **MEMMAP_OFF** as described above, you can set the size (in bytes) in the TotalView startup parameters.

For example:

```
MPI_MAPPED_HEAP_SIZE=1048576
```

NOTE: SGI has a patch for an *MPT/XPMEM* issue. Without this patch, *XPMEM* can crash the system if ReplayEngine is turned on. To get the *XPMEM* fix for the munmap problem, either upgrade to ProPack 6 SP 4 or install SGI patch 10570 on top of ProPack 6 SP 3.

Sun MPI Applications

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

TotalView can debug a Sun MPI program and can display Sun MPI message queues. This section describes how to perform *job startup* and *job attach* operations.

To start a Sun MPI application:

1. Enter the following command:

   totalview mprun [*totalview_args* **] -a [** *mpi_args* **]**

   For example:

   ```
   totalview mprun -g blue -a -np 4 /usr/bin/mpi/conn.x
   ```

   CLI: totalviewcli mprun [*totalview_args* **] -a [** *mpi_args* **]**

   When the TotalView Process Window appears, select the **Go** button.

   CLI: dfocus p dgo

   TotalView may display a dialog box with the following text:

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

2. If you compiled using the **-g** option, click **Yes** to open a Process Window that shows your source. All processes are halted.

Attaching to a Sun MPI Job

To attach to an already running **mprun** job:

1. Find the host name and process identifier (PID) of the **mprun** job by typing **mpps -b**. For more information, see the **mpps(1M)** manual page.

   The following is sample output from this command:

   ```
   JOBNAME     MPRUN_PID   MPRUN_HOST
   cre.99      12345       hpc-u2-9
   cre.100     12601       hpc-u2-8
   ```

2. After selecting **File > Attach to a Running Program**, type **mprun** in the **File Name** 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, select the host or add a new host in the **Host** field.

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

Starting MPI Issues

NOTE: In many cases, you can bypass the procedure described in this section. For more information, see Debugging MPI Programs on page 518.

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 verify if you can start remote processes on all of the computers in your computers file. (See **tstmachines** in **mpich/util**.)

- You won't get a message queue display if you get the following warning:

  ```
  The symbols and types in the MPICH library used by TotalView to extract the message queues are not as expected in the image <your image name>. This is probably an MPICH version or configuration problem.
  ```

  You need to check that you are using MPICH Version 1.1.0 or later and that you have configured it with the **-debug** option. (You can check this by looking in the **config.status** file at the root of the MPICH directory tree.)

- Does the TotalView Server (**tvdsvr**) fail to start?

  tvdsvr must be in your **PATH** when you log in. Remember that TotalView uses **ssh** 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 at https://help.totalview.io/ for up-to-date information.

- Under some circumstances, MPICH kills TotalView with the **SIGINT** signal. You can see this behavior when you use the **Group > Kill** command as the first step in restarting an MPICH job.

  CLI: dfocus g ddelete

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

RELATED TOPICS

Tips for debugging MPI applications — MPI Debugging Tips and Tools on page 445

RELATED TOPICS	
The TotalView server, **tvdsvr**	" The tvdsvr Command and Its Options" in the *Classic TotalView Reference Guide*
MPI version information	The *TotalView Release Notes* on the TotalView documentation page

Using ReplayEngine with Infiniband MPIs

In general, using ReplayEngine with MPI versions that communicate over Infiniband is no different than using it with other MPIs, but its use requires certain environment settings, as described here. If you are launching the MPI job from within TotalView, these are set for you; if instead, you start the MPI program from outside TotalView, you must explicitly set your environment.

Required Environment Settings

When you start the MPI program from within TotalView with ReplayEngine enabled, TotalView inserts environment variable settings into the MPI processes to disable certain RDMA optimizations. (These are optimizations that hinder ReplayEngine's ability to identify the memory regions being actively used for RDMA, and their use can therefore result in unreasonably slow execution in record mode.) These variables are set for you, requiring no extra tasks compared to using a non-Infiniband MPI.

The inserted settings are:

- `VIADEV_USE_DREG_CACHE=0` (addresses MVAPICH1 versions)
- `MV2_DREG_CACHE_LIMIT=1` (addresses MVAPICH2 versions)
- `MV2_RNDV_PROTOCOL=R3` (addresses Intel MPI versions, also affects MVAPICH2)
- `OMPI_MCA_mpool_rdma_rcache_size_limit=1` (addresses Open MPI versions)

When the MPI program is started outside TotalView (for example, when using a command like **mpirun -tv**, or when you attach TotalView to an MPI program that is already running), you must set the relevant environment variable for your MPI version, as described above. Also, two additional environment variables are required to make the MPI program's use of RDMA memory visible to ReplayEngine, as follows:

- `IBV_FORK_SAFE`: Set to any value, for example `IBV_FORK_SAFE=1`
- `LD_PRELOAD`: Set to include a preload library, which can be found under the TotalView installation directory at **toolworks/totalview.<version>/linux-x86-64/lib/undodb_infiniband_preload_x64.so**.

For example, here's how to set the environment for the MVAPICH1 implementation of MPI:

```
mpirun_rsh -np 8 -hostfile myhosts \
VIADEV_USE_DREG_CACHE=0 IBV_FORK_SAFE=1 \
LD_PRELOAD=/<path>/undodb_infiniband_preload_x64.so myprogram
```

For more information, consult your MPI version documentation for specifics on setting environment variables.

Cray XT/XE/XK/XC MPIs

On Cray XT/XE/XK/XC (x86_64 only) systems, although Infiniband is not used, the MPIs do use RDMA techniques. As a result, using Replay on these systems requires some particular environmental settings. Briefly, the required settings are `MPICH_SMP_SINGLE_COPY_OFF = 1`, and `LD_PRELOAD` set to the location of the Infiniband pre-load library described above. Refer to the "Debugging Cray XT/XE/XK/XC Applications" section for details.

Possible Errors

ReplayEngine checks environment settings before it attaches to the MPI program, but in some cases, may not detect incompatible settings, reporting the following errors:

- If ReplayEngine finds that either the **IBV_FORK_SAFE** setting is absent, or that the preload library has not been loaded, it declines to attach and issues an error message citing unmet prerequisites. You can still attach TotalView to the program without ReplayEngine - for example, in the GUI by using the New Program dialog.
- If ReplayEngine cannot determine that the environment variable setting to disable an MPI optimization has been set, it continues to attach, but issues a warning message that it could not verify prerequisites. Depending on your program's use of memory for RDMA, you may find that it runs unreasonably slowly in record mode, or encounters errors that would not occur if ReplayEngine were not attached.

RELATED TOPICS	
Using ReplayEngine in general	*Getting Started with Replay Engine*
MPI programs	Debugging MPI Programs on page 518

Setting Up Parallel Debugging Sessions

This chapter explains how to set up TotalView parallel debugging sessions for applications that use the parallel execution models that TotalView supports and which do not use MPI.

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

This chapter discusses the following topics:

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

- Define the process you want TotalView to attach to. See Attaching to Processes Tips on page 440.
- See Debugging Strategies for Parallel Applications on page 437 for general hints on how to approach debugging parallel programs.

Debugging OpenMP Applications

TotalView supports many OpenMP compilers for the C, C++, and Fortran languages. Supported compilers and architectures are listed in the *TotalView Platforms and Systems Requirements* document.

The following are some 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 supported compilers.

Topics in this section are:

Debugging OpenMP Programs

Debugging OpenMP code is similar to debugging multi-threaded code. The major differences are in the way the OpenMP compiler alters your code. These alterations include:

- *Outlining*. The compiler pulls the body of a parallel region out of the original routine and places it in an *outlined routine*. In some cases, the compiler generates multiple outlined routines from a single parallel region. This allows multiple threads to execute the parallel region.

  The outlined routine's name is based on the original routine's name. In most cases, the compiler adds a numeric suffix.

- The compiler inserts calls to the OpenMP runtime library.

- The compiler splits variables between the original routine and the outlined routine. Normally, shared variables reside in the master thread's original routine, and private variables reside in the outlined routine.
- The master thread creates threads to share the workload. As the master thread begins to execute a parallel region in the OpenMP code, it creates the worker threads, dispatches them to the outlined routine, and then calls the outlined routine itself.

About TotalView OpenMP Features

TotalView interprets the changes that the OpenMP compiler makes to your code so that it can display your program in a coherent way. Here are some things you should know:

- The compiler can generate multiple outlined routines from a single parallel region. This means that a single line of source code can generate multiple blocks of machine code inside different functions.
- You can't single step into or out of a parallel region. Instead, set a breakpoint inside the parallel region and let the process run to it. After execution reaches the parallel region, you can single step in it.
- OpenMP programs are multi-threaded programs, so the rules for debugging multi-threaded programs apply.

About OpenMP Platform Differences

In general, TotalView smooths out the differences that occur when you execute OpenMP platforms on different platforms. The following list discusses these differences:

- The OpenMP master thread has logical thread ID number 1. The OpenMP worker threads have a logical thread ID number greater than 1.
- Select or dive on the stack parent token line to view the original routine's stack frame in the OpenMP master thread.
- When you stop the OpenMP worker threads in a **PARALLEL DO** outlined routine, the stack backtrace shows the following call sequence:
  - Outlined routine called from the special stack parent token line.
  - The OpenMP runtime library called from.
  - The original routine (containing the parallel region).

Viewing OpenMP Private and Shared Variables

You can 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 258. In contrast, the compiler maintains OpenMP shared variables in the master thread's original routine stack frame.

You can display shared variables through a Process Window focused on the OpenMP master thread, or through one of the OpenMP worker threads.

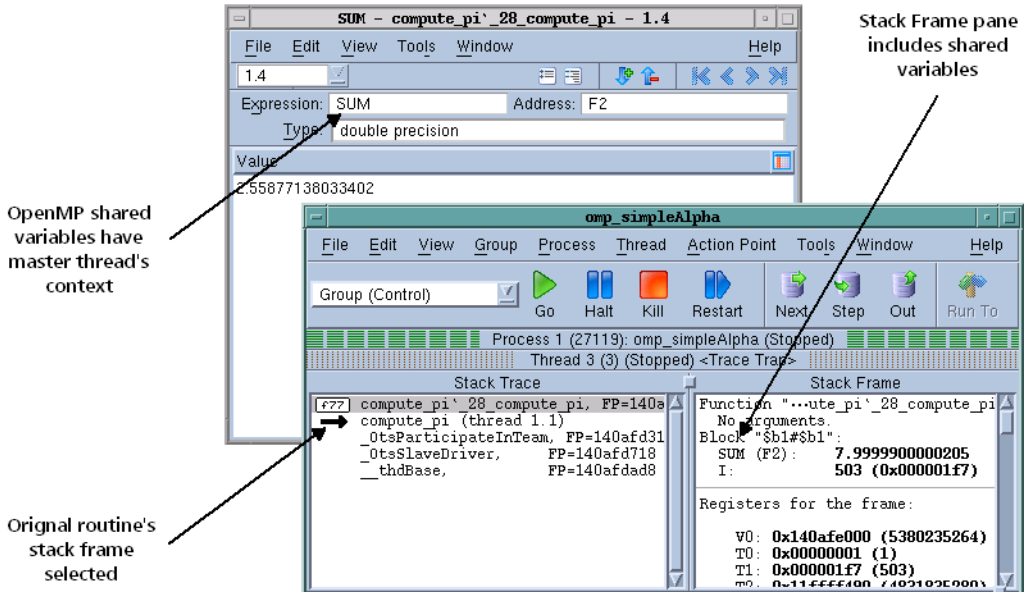
To see these variables:

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 need to first set your focus to the OpenMP master thread.

   A Variable Window is launched that displays the value of the OpenMP shared variable, as shown in Figure 245.

Figure 245, OpenMP Shared Variable



Shared variables reside in the OpenMP master thread's stack. When displaying shared variables in OpenMP worker threads, TotalView uses the stack context of the OpenMP master thread to find the shared variable. TotalView uses the OpenMP master thread's context when displaying the shared variable in a Variable Window.

You can also view OpenMP shared variables in the Stack Frame Pane by selecting either of the following:

- Original routine stack frame in the OpenMP master thread.
- Stack parent token line in the Stack Trace Pane of OpenMP worker threads.

Viewing OpenMP THREADPRIVATE Common Blocks

Some 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, IBM and other compilers use the pthread key facility.

To view a variable in an OpenMP **THREADPRIVATE** common block or the OpenMP **THREADPRIVATE** common block:

1. In the Threads Tab of the Process Window, select the thread that contains the private copy of the variable or common block you want to view.
2. In the Stack Trace Pane of the Process Window, select the stack frame that lets you access the OpenMP **THREADPRIVATE** common block variable. You can select either the outlined routine or the original routine for an OpenMP master thread. You must, however, select the outlined routine for an OpenMP worker thread.
3. From the Process Window, dive on the variable name or common block name, or select the **View > Lookup Variable** command. When prompted, enter the name of the variable or common block. You may need to append an underscore character (_) after the common block name.

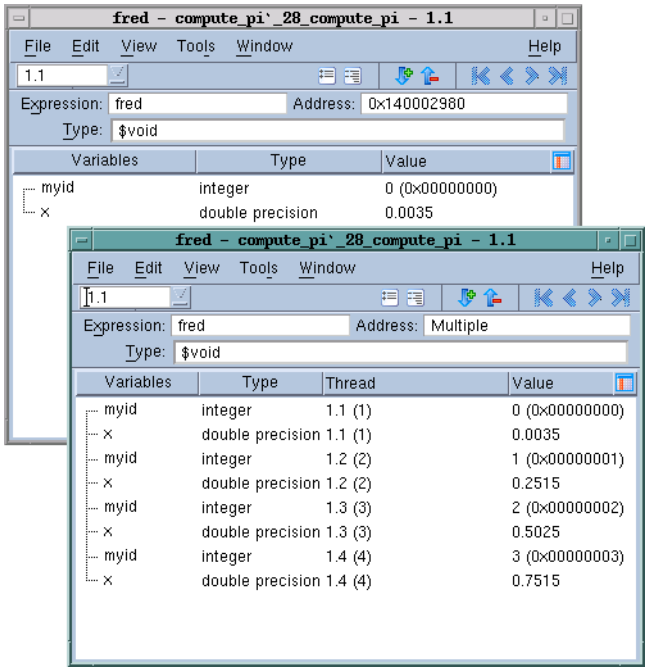
   CLI: dprint

   A Variable Window opens that displays the value of the variable or common block for the selected thread.

   See Displaying Variables on page 246 for more information on displaying variables.
4. To view OpenMP **THREADPRIVATE** common blocks or variables across all threads, use the Variable Window's **Show across > Threads** command. See Displaying a Variable in all Processes or Threads on page 330.

Figure 246 shows Variable Windows displaying OpenMP **THREADPRIVATE** common blocks. Because the Variable Window has the same thread context as the Process Window from which it was created, the title bar patterns for the same thread match. TotalView displays the values of the common block across all threads when you use the **View > Show Across > Threads** command.

Figure 246, OpenMP THREADPRIVATE Common Block Variables

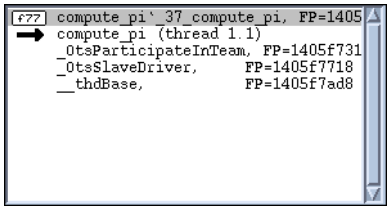


Viewing the OpenMP Stack Parent Token Line

TotalView inserts a special stack parent token line in the Stack Trace Pane of OpenMP worker threads when they are stopped in an outlined routine.

When you select or dive on the stack parent token line, the Process Window switches to the OpenMP master thread, allowing you to see the stack context of the OpenMP worker thread's routine.

Figure 247, OpenMP Stack Parent Token Line



This stack context includes the OpenMP shared variables.

Using SLURM

TotalView supports the SLURM resource manager. Here is some information copied from the SLURM website (https://computing.llnl.gov/tutorials/linux_clusters/).

> SLURM is an open-source resource manager designed for Linux clusters of all sizes. It provides three key functions. First it allocates exclusive and/or non-exclusive access to resources (computer nodes) to users for some duration of time so they can perform work. Second, it provides a framework for starting, executing, and monitoring work (typically a parallel job) on a set of allocated nodes. Finally, it arbitrates conflicting requests for resources by managing a queue of pending work.
>
> SLURM is not a sophisticated batch system, but it does provide an Applications Programming Interface (API) for integration with external schedulers such as the Maui Scheduler. While other resource managers do exist, SLURM is unique in several respects:

- Its source code is freely available under the GNU General Public License.
- It is designed to operate in a heterogeneous cluster with up to thousands of nodes.
- It is portable; written in C with a GNU autoconf configuration engine. While initially written for Linux, other UNIX-like operating systems should be easy porting targets. A plugin mechanism exists to support various interconnects, authentication mechanisms, schedulers, etc.
- SLURM is highly tolerant of system failures, including failure of the node executing its control functions.
- It is simple enough for the motivated end user to understand its source and add functionality.

Debugging Cray XT/XE/XK/XC Applications

The Cray XT/XE/XK/XC series of supercomputers are supported by the TotalView Linux x86_64 and Linux ARM64 (aarch64) distributions. The discussion here is based on running applications using the Cray Linux Environment (CLE). TotalView supports launching application programs using either PBS Pro with ALPS **aprun** or SLURM **srun**.

RELATED TOPICS	
Setting up an MPI debugging session	Setting Up MPI Debugging Sessions on page 516
Tips for parallel debugging	General Parallel Debugging Tips on page 438

Starting TotalView on Cray

Because the configuration of most Cray systems typically varies from site to site, the following provides only general guidelines for starting TotalView on your application. Please consult your site's documentation for the specific steps needed to debug a program using TotalView on your Cray system.

File System Considerations

Place your application to debug on a file system that is shared across all Cray node types, such as the service, elogin, login/MOM, and/or compute nodes. This allows you to compile and debug your application across node types.

Further, make sure that your `$HOME/.totalview` directory is on a shared file system that is common across all Cray node types to ensure that you can use the **tvconnect** feature in batch scripts. For more information, see Reverse Connections on page 505.

Starting TotalView

TotalView typically runs interactively. If your site has not designated any compute nodes for interactive processing, you can allocate compute nodes for interactive use. Use PBS Pro's **qsub -I** or SLURM's **salloc** command to allocate an interactive job. Be sure that your X11 **DISPLAY** environment variable is propagated or set properly. See "man qsub" or "man salloc" for more information on interactive jobs.

If TotalView is installed on your system, load it into your user environment:

```
module load totalview
```

Use the following command to start TotalView where *mpi_starter* is the MPI starter program for your system, such as **aprun** or **srun**.

The CLI:

totalviewcli *-tv_options* **-args** *mpi_starter [mpi_options] application_name [application_arguments]*

The GUI:

totalview *-tv_options* **-args** *mpi_starter [mpi_options] application_name [application_arguments]*

TotalView is not able to stop your program before it calls **MPI_Init()** when using ALPS. While this is typically at the beginning of **main()**, the actual location depends on how you've written the program. This means that if you set a breakpoint before the **MPI_Init()** call, your program will not hit it because the statement upon which you set the breakpoint will have already executed. On the other hand, SLURM will stop your program before it enters main(), which allows you to debug the statements before **MPI_Init()** is called.

Example 1: Interactive Jobs Using qsub and aprun

This example shows how you can start TotalView on a program named **a.out** running in an interactive job using **qsub** and **aprun**.

```
n9610@crystal:~> qsub -I -X -l nodes=4
qsub: waiting for job 599856.sdb to start
qsub: job 599856.sdb ready

Directory: /home/users/n9610
Mon Nov  4 17:23:28 CST 2019
n9610@crystal2:~> cd shared/rctest
n9610@crystal2:~/shared/rctest> module load totalview
n9610@crystal2:~/shared/rctest> totalview -verbosity errors -args aprun -n 4 ./a.out
```

Example 2: Interactive jobs using salloc and srun

Similarly, you can debug an interactive job using **salloc** and **srun** if your Cray system uses SLURM.

This example shows how to submit a SLURM batch job using **tvconnect** in the batch script. After the batch job starts running, TotalView is started to accept the reverse-connect request.

```
n9610@jupiter-elogin:~/shared/rctest> cat slurm-script.bash
#!/bin/bash -x
#SBATCH --qos=debug
#SBATCH --time=00:30:00
#SBATCH --nodes=4
#SBATCH --tasks-per-node=1
#SBATCH --constraint=haswell

module load totalview
tvconnect srun -n 4 tx_basic_mpi
n9610@jupiter-elogin:~/shared/rctest> sbatch -C BW28 slurm-script.bash
Submitted batch job 1374150
n9610@jupiter-elogin:~/shared/rctest> squeue -u $USER
   JOBID    USER ACCOUNT           NAME  ST REASON    START_TIME          TIME  TIME_LEFT NODES CPUS
 1374150   n9610  (null) slurm-script.b   R   None    2019-11-05T09:31:53 0:16      29:44     4    8
n9610@jupiter-elogin:~/shared/rctest> module load totalview
n9610@jupiter-elogin:~/shared/rctest> totalview
```

Support for Cray Abnormal Termination Processing (ATP)

Cray's ATP module stops a running job at the moment it crashes. This allows you to attach TotalView to the held job and begin debugging it. To hold a job as it is crashing you must set the `ATP_HOLD_TIME` environment variable before launching your job with **aprun** or **srun**.

When your job crashes, the MPI starter process outputs a message stating that your job has crashed and that ATP is holding it. You can now attach TotalView tothe **aprun** or **srun** process using the normal attach procedure (see Attaching to a Running Program on page 105.

For more information on ATP, see the Cray `intro_atp` man page.

Special Requirements for Using ReplayEngine

On Crayx86_64 systems, the MPIs use RDMA techniques, similar to Infiniband MPIs. When using ReplayEngine on MPI programs, certain environment variable settings must be in effect for the MPI rank processes. These settings ensure that memory mapping operations are visible to ReplayEngine. The required environment variable settings are:

- `MPICH_SMP_SINGLE_COPY_OFF=1`
- `LD_PRELOAD`: Set to include a preload library, which can be found under the TotalView installation directory at `toolworks/totalview.`*<version>*`/linux-x86-64/lib/undodb_infiniband_preload_x64.so`.

When using APLS, these settings may be applied with the **aprun -e** option. For example, to have TotalView launch an MPI program with ReplayEngine enabled, use a command similar to this:

```
totalview -replay -args aprun -n 8 \
-e MPICH_SMP_SINGLE_COPY_OFF=1 \
-e LD_PRELOAD=/<path>/undodb_infiniband_preload_x64.so \
myprogram
```

When using SLURM, these settings may be applied with the **srun --export** option. For example:

```
totalview -replay -args srun -n 8 \
--export=ALL,MPICH_SMP_SINGLE_COPY_OFF=1,LD_PRELOAD=/<path>/undodb_infiniband_pre-
load_x64.so \
myprogram
```

Debugging Global Arrays Applications

The following paragraphs, which are copied from the Global Arrays home site (http://hpc.pnl.gov/globalarrays), describe the global arrays environment:

> The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures (called "global arrays"). From the user perspective, a global array can be used as if it was stored in shared memory. All details of the data distribution, addressing, and data access are encapsulated in the global array objects. Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important. The primary target architectures for which GA was developed are massively-parallel distributed-memory and scalable shared-memory systems.
>
> GA divides logically shared data structures into "local" and "remote" portions. It recognizes variable data transfer costs required to access the data depending on the proximity attributes. A local portion of the shared memory is assumed to be faster to access and the remainder (remote portion) is considered slower to access. These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located. In addition, any processes can access a local portion of the shared data directly/in-place like any other data in process local memory. Access to other portions of the shared data must be done through the GA library calls.
>
> GA was designed to complement rather than substitute for the message-passing model, and it allows the user to combine shared-memory and message-passing styles of programming in the same program. GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.

The global arrays environment has a few unique attributes. Using TotalView, you can:

- Display a list of a program's global arrays.
- Dive from this list of global variables to see the contents of a global array in C or Fortran format.
- Cast the data so that TotalView interprets data as a global array handle. This means that TotalView displays the information as a global array. Specifically, casting to **$GA** forces the Fortran interpretation; casting to **$ga** forces the C interpretation; and casting to **$Ga** uses the language in the current context.

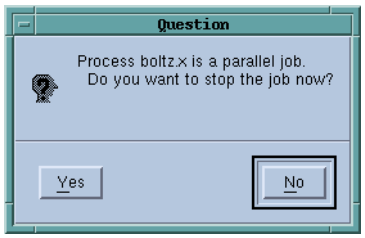
Within a Variable Window, the commands that operate on a local array, such as slicing, filtering, obtaining statistics, and visualization, also operate on global arrays.

The command used to start TotalView depends on your operating system. For example, the following command starts TotalView on a program invoked using **prun** using three processes:

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

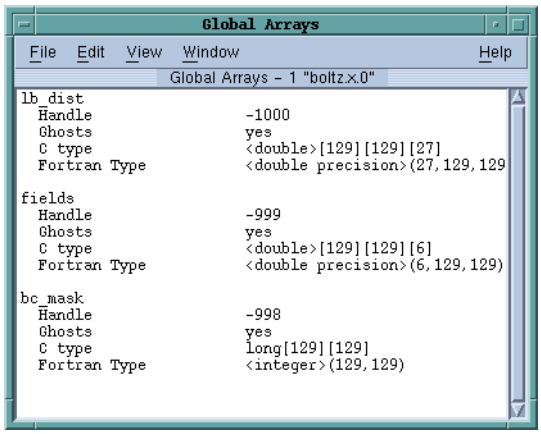
Before your program starts parallel execution, a Question dialog launches so you can stop the job to set breakpoints or inspect the program before it begins execution

Figure 248, Question Window for Global Arrays Program



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

Figure 249, Tools > Global Arrays Window



CLI: dga

The arrays named in this window are displayed using their C and Fortran type names. Diving on the line that contains the type definition displays Variable Windows that contain information about that array.

After TotalView displays this information, you can use other standard commands and operations on the array. For example, you can use the slice and filter operations and the commands that visualize, obtain statistics, and show the nodes from which the data was obtained.

If you inadvertently dive on a global array variable from the Process Window, TotalView does not know that it is a component of a global array. If, however, you do dive on the variable, you can cast the variable into a global array using either **$ga** for a C Language cast or **$GA** for a Fortran cast.

Debugging Shared Memory (SHMEM) Code

TotalView supports programs using the distributed memory access Shared Memory (SHMEM) library on Quadrics RMS systems and SGI Altix systems. The SHMEM library allows processes to read and write data stored in the memory of other processes. This library also provides collective operations.

Debugging a SHMEM RMS or SGI Altix program is no different than debugging any other program that uses a starter program. For example:

```
totalview srun -a my_program
```

Debugging UPC Programs

TotalView supports debugging UPC programs on Linux x86 platforms. This section discusses only the UPC-specific features of TotalView. It is not an introduction to the UPC Language. For an introduction to the UPC language, see https://www2.gwu.edu/~upc/.

NOTE: When debugging UPC code, TotalView requires help from a UPC assistant library that your compiler vendor provides. You need to include the location of this library in your **LD_LIBRARY_PATH** environment variable. TotalView also provides assistants that you can use.

Topics in this section are:

Invoking TotalView

The way in which you invoke TotalView on a UPC program is straight-forward. However, this procedure depends on the parallel technology you are using. Here are a couple of examples:

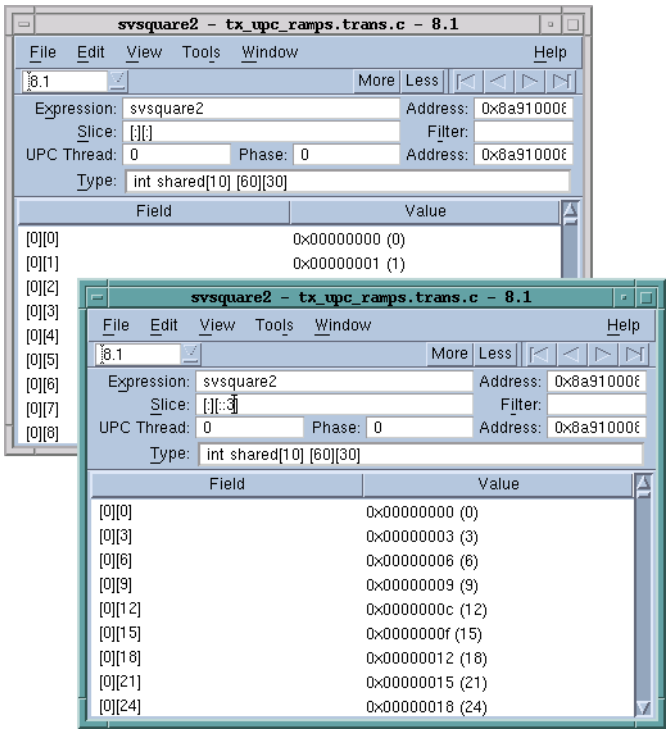
- For Quadrics RMS:
  ```
  totalview prun -a prog_upc_args
  ```
- For MPICH and LAM
  ```
  totalview mpirun -a -np 2 prog_upc_args
  ```

Viewing Shared Objects

TotalView displays UPC shared objects, and fetches data from the UPC thread with which it has an affinity. For example, TotalView always fetches shared scalar variables from thread 0.

The upper-left screen in Figure 250 displays elements of a large shared array. You can manipulate and examine shared arrays the same as any other array. For example, you can slice, filter, obtain statistical information, and so on. (For more information on displaying array data, see Examining Arrays on page 312.) The lower-right screen shows a slice of this array.

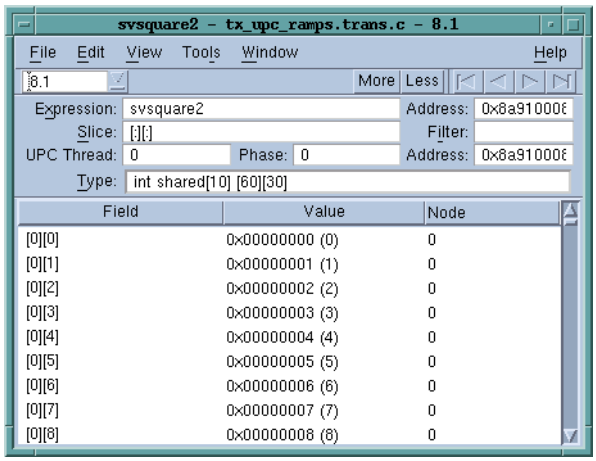
Figure 250, A Sliced UPC Array



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

Since the array is shared, it has an additional property: the element's affinity. You can display this information if you right-click your mouse on the header and tell TotalView to display Nodes.

Figure 251, UPC Variable Window Showing Nodes

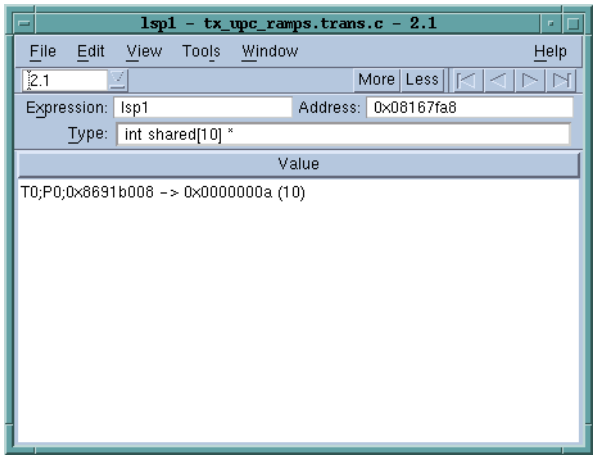


You can also use the **Tools > Visualize Distribution** command to visualize this array. For more information on visualization, see Array Visualizer on page 341.

Displaying Pointer to Shared Variables

TotalView understands pointer-to-shared data and displays the components of the data, as well as the target of the pointer to shared variables. For example,Figure 252 shows this data being displayed:

Figure 252, A Pointer to a Shared Variable



In this figure, notice the following:

- Because the **Type** field displays the full type name, this is a pointer to a shared **int** with a block size of 10.
- TotalView also displays the **upc_threadof ("T0")**, the **upc_phaseof ("P0"**), and the **upc_addrfield (0x0x10010ec4)** components of this variable.

In the same way that TotalView normally shows the target of a pointer variable, it also shows the target of a UPC pointer variable. When dereferencing a UPC pointer, TotalView fetches the target of the pointer from the UPC thread with which the pointer has affinity.

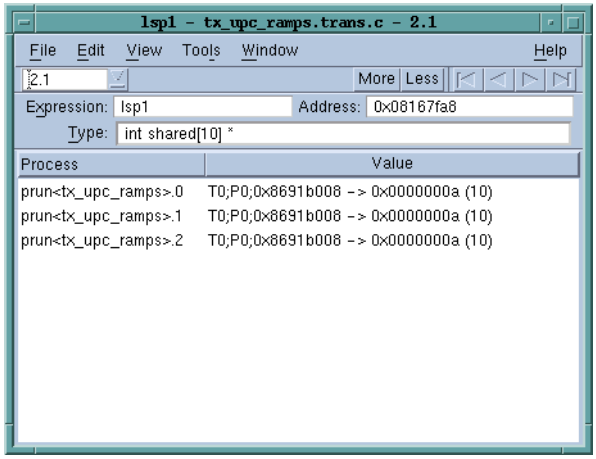
You can update the pointer by selecting the pointer value and editing the thread, phase, or address values. If the phase is corrupt, you'll see something like the following in the **Value** area:

```
T0;P6;0x3ffc0003b00 <Bad phase [max 4]> ->
                        0xc0003c80 (-1073726336)
```

In this example, the pointer is invalid because the phase is outside the legal range. TotalView displays a similar message if the thread is invalid.

Since the pointer itself is not shared, you can use the **TView > Show Across** commands to display the value from each of the UPC threads.

Figure 253, Pointer to a Shared Variable



Debugging CoArray Fortran (CAF) Programs

TotalView has partial support for debugging CoArray Fortran (CAF) programs on Cray platforms. This section discusses the parts of TotalView that support CAF-specific features. CoArray Fortran allows a programmer to distribute parts of an array over a set of processes using an augmented Fortran array syntax. The processes in a CAF job share the same executable. The processes are assigned "image ids" starting at image one.

NOTE: When debugging CAF code, TotalView requires help from a CAF assistant library that your compiler vendor provides. You need to include the location of this library in your LD_LIBRARY_PATH environment variable. TotalView also provides assistants that you can use.

Because currently TotalView support is partial, expressions that attempt to re-cast CAF types or change the visible slices of CAF types are likely to fail.

Invoking TotalView

CAF programs commonly rely on an underlying parallel protocol such as MPI. They are started the same way as other programs using the same parallel technology.

On Cray machines that use `aprun`, invoking a four-image job on TotalView may look like this:

```
totalview aprun -a -n 4 caf_program caf_program_args
```

Viewing CAF Programs

For a CAF program, the process id in the TotalView process window shows the CAF image id. TotalView shows the correct dimensions and co-dimensions of arrays and the co-dimensions of scalars.

When diving on a CAF array or scalar, TotalView shows the data local to the current image. Diving across processes shows the entire distributed array.

Figure 254, Diving on CAF array *y*

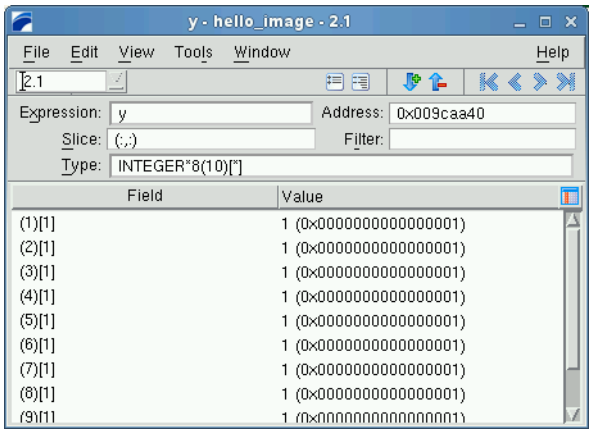
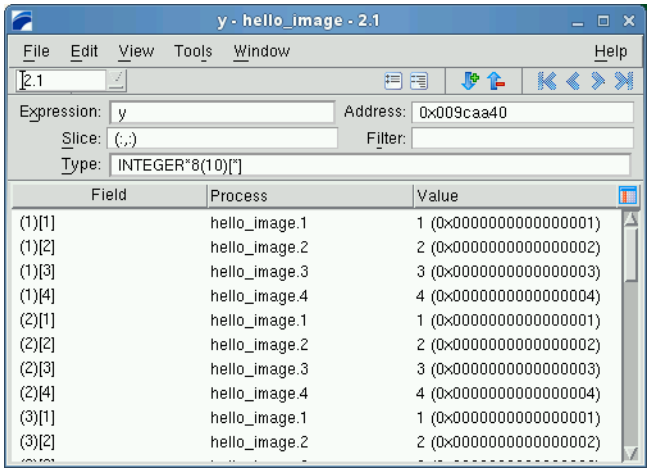


Figure 255, Diving on CAF array *y* across processes



If you use the **array viewer**, **statistics**, and **visualizer** commands from the Tools menu when viewing a CAF array across processes, the commands treat the co-array dimensions much like standard array dimensions.

Using CLI with CAF

The **dprint** command in the CLI displays the data in CAF arrays in a similar way to the above. When the focus is a process, **dprint** lists the local values. When the focus is the shared group containing the CAF images, **dprint** lists the entire co-array.

Controlling fork, vfork, and execve Handling

TotalView supports customizing how to handle system calls to system calls to **execve()**, **fork()**, **vfork()**, and **clone()** (when used without the **CLONE_VM** flag).

This chapter includes:

exec_handling and fork_handling Command Options and State Variables

TotalView allows you to control how the debugger handles system calls to **execve()**, **fork()**, **vfork()**, and **clone()** (when used without the **CLONE_VM** flag).

- When calling **fork()**, **vfork()** or **clone()**, choose to either attach or detach from the new child process.
- When calling **execve()**, choose either to continue the new process, halt it, or ask what action to take.

This behavior is controlled by two CLI state variables and two command options. Set the state variables to control the default behavior for TotalView. Use the command options when starting TotalView to control the behavior for a particular debugging session. The command options override the state variable settings.

Table 2: exec_handling and fork_handling Command Options and State Variables

Command Options	CLI State Variables
-exec_handling *exec-handling-list*	**TV::exec_handling** *exec-handling-list*
-fork_handling *fork-handling-list*	**TV::fork_handling** *fork-handling-list*

The lists *exec-handling-list and fork-handling-list* are Tcl lists of *regexp* and *action* pairs. Each *regexp* is matched against the process's name to find a matching *action*, which determines how to handle the exec or fork event.

RELATED TOPICS	
The state variables	**TV::exec_handling** *exec-handling-list* and **TV::fork_handling** *fork-handling-list*
The `totalview` command options	**-exec_handling** *exec-handling-list* and **-fork_handling** *fork-handling-list*
Setting breakpoints when using **fork()** and **execve()**	Setting Breakpoints When Using the fork()/execve() Functions on page 213
Linking with the dbfork library	Compiling Programs on page 87, and "Linking with the dbfork Library" in the *Classic TotalView Reference Guide*

Exec Handling

When a process being debugged execs a new executable, the debugger iterates over ***exec-handling-list*** to match the original process name (that is, the name of the process before it called exec) against each ***regexp*** in the list. When it finds a match, it uses the corresponding ***action***, as follows:

Action	Description
halt	Stop the process
go	Continue the process
ask	Ask whether to stop the process

If a matching process name is not found in the ***exec-handling-list***, the value of the **TV::parallel_stop** CLI state variable preference is used.

Fork Handling

When first launching or attaching to a process, the debugger iterates over *fork-handling-list* to match the process name against each *regexp* in the list. When it finds a match, it uses the corresponding *action* to determine how future fork system calls will be handled, as follows:

Action	Description
attach	Attach to the new child processes.
detach	Detach from the new child processes.

If a matching process name is not found in the *fork-handling-list* list, TotalView handles **fork()** based on whether the process was linked with the **dbfork** library and the setting of the **TV::dbfork** CLI state variable preference.

For more information, see"Linking with the dbfork Library" in the *Classic TotalView Reference Guide.*

Example

It's important to properly construct the *exec-handling-list* and *fork-handling-list* list of pairs, so that the list is properly quoted for Tcl or the shell. Generally, enclose the list in curly braces in the CLI, and enclose it in single quotes in the shell.

Note that the regular expressions are not anchored, so you must use "^" and "$" to match the beginning or end of the process name.

Calling exec:

This example configures TotalView to automatically continue the process (without asking) when bash calls exec, but to ask when other processes call exec, using the following **dset** CLI command or `totalview` command option:

```
dset TV::exec_handling {{{^bash$} go} {. ask}}
totalview -exec_handling '{{^bash$} go} {. ask}'
```

Above, the *regexp* is wrapped in an extra set of curly braces to make sure that Tcl does not process the "$" as a variable reference.

Calling fork:

This example configures TotalView to attach to the child process when a process containing the name "tx_-fork_exec" calls fork, but to detach from other forked processes, using the following **dset** CLI command or `totalview` command option:

```
dset TV::fork_handling {{tx_fork_exec attach} {. detach}}
totalview -fork_handling '{tx_fork_exec attach} {. detach}'
```

An example session:

```
% totalviewcli -verbosity errors \
  -exec_handling '{{^bash$} go} {. ask}' \
  -fork_handling '{tx_fork_exec attach} {. detach}' \
  -args \
    bash -c 'tx_fork_exec tx_hello'
d1.<> co
Parent done ....
Child is calling execve ...
Process bash<tx_fork_exec>.1 has exec'd /path/to/tx_hello.
  Do you want to stop it now?

: yes
d1.<> ST
1       (0)          Nonexistent [bash]
2       (20053)      Stopped     [bash<tx_fork_exec><tx_hello>.1]
  2.1   (20053/20053)   Stopped     PC=0x7f70517dd210
d1.<>
```

Group, Process, and Thread Control

The specifics of how multi-process, multi-threaded programs execute differ greatly from platform to platform and environment to environment, but all share some general characteristics. This chapter discusses the TotalView process/thread model. It also describes how you tell the GUI and the CLI what processes and threads to direct a command to.

This chapter contains the following sections:

Defining the GOI, POI, and TOI

This chapter consistently uses the following three related acronyms:

- GOI—Group of Interest
- POI—Process of Interest
- TOI—Thread of Interest

These terms are important in the TotalView process/thread model because TotalView must determine the scope of what it does when it executes a command. For example, About Groups, Processes, and Threads introduced the types of groups TotalView defines. That chapter ignored what happens when you execute a TotalView command on a group. For example, what does "stepping a group" actually mean? What happens to processes and threads that aren't in this group?

Associated with these three terms is a fourth term: *arena*. The *arena* is the collection of processes, threads, and groups that are affected by a debugging command. This collection is called an *arena list*.

In the GUI, the arena is most often set using the pulldown list in the toolbar. You can also set the arena using commands in the menubar. For example, there are eight *next* commands. The difference between them is the arena; that is, the difference between the *next* commands is the processes and threads that are the target of what the *next* command runs.

When TotalView executes any action command, the arena decides the scope of what can run. It doesn't, however, determine what does run. Depending on the command, TotalView determines the TOI, POI, or GOI, and then executes the command's action on that thread, process, or group. For example, suppose TotalView steps the current control group:

- TotalView needs to know what the TOI is so that it can determine what threads are in the lockstep group—TotalView only lets you step a lockstep group.
- The lockstep group is part of a share group.
- This share group in turn is part of a control group.

By knowing what the TOI is, the GUI also knows what the GOI is. This is important because, while TotalView knows what it will step (the threads in the lockstep group), it also knows what it will allow to run freely while it is stepping these threads. In the CLI, the P/T set determines the TOI.

RELATED TOPICS	
Concept information on threads and processes and how TotalView organizes them into groups	About Groups, Processes, and Threads on page 383
Selecting a focus	Using the Toolbar to Select a Target on page 416

Recap on 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  icon masking a line number indicates that a breakpoint is set on the line. Selecting the  icon clears the breakpoint.

When a program reaches a breakpoint, it stops. You can let the program resume execution in any of the following ways:

- Use the single-step commands described in Using Stepping Commands on page 178.
- 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 187.
- Set breakpoints at lines you choose, and let your program execute to that breakpoint. See Setting Breakpoints and Barriers on page 194.
- Set conditional breakpoints that cause a program to stop after it evaluates a condition that you define, for example, “stop when a value is less than eight." See Setting Eval Points on page 222.

TotalView provides additional features for working with breakpoints, process barrier breakpoints, and eval points. For more information, see Setting Action Points on page 188.

Stepping (Part I)

You can use TotalView stepping commands to:

- Execute one source line or machine instruction at a time; for example, **Process > Step** in the GUI and **dstep** in the CLI.

  CLI: dstep

- Run to a selected line, which acts like a temporary breakpoint; for example, **Process > Run To**.

  CLI: duntil

- Run until a function call returns; for example, **Process > Out**.

  CLI: dout

In all cases, stepping commands operate on the Thread of Interest (TOI). In the GUI, the TOI is the selected thread in the current Process Window. In the CLI, the TOI is the thread that TotalView uses to determine the scope of the stepping operation.

On all platforms except SPARC Solaris, TotalView uses *smart* single-stepping to speed up stepping of one-line statements that contain loops and conditions, such as Fortran 90 array assignment statements. *Smart stepping* occurs when TotalView realizes that it doesn't need to step through an instruction. For example, assume that you have the following statements:

```
integer iarray (1000,1000,1000)
iarray = 0
```

These two statements define one billion scalar assignments. If your computer steps every instruction, you will probably never get past this statement. *Smart stepping* means that TotalView single-steps through the assignment statement at a speed that is very close to your computer's native speed.

NOTE: To define a rule to skip over or through specific functions or files, use the **dskip** command. You can add rules that match a function, all functions in a source file, or a specific function in a specific source file.

Other topics in this section are:

- Understanding Group Widths on page 576
- Understanding Process Width on page 576

- Understanding Thread Width on page 577
- Using Run To and duntil Commands on page 577

RELATED TOPICS	
Stepping through your program	Using Stepping Commands on page 178
Stepping examples	Stepping (Part II): Examples on page 598

Understanding Group Widths

TotalView behavior when stepping at group width depends on whether the Group of Interest (GOI) is a process group or a thread group. In the following lists, *goal* means the place at which things should stop executing. For example, if you selected a *step* command, the goal is the next line. If you selected a *run to* command, the goal is the selected line.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView examines the group, and identifies which of its processes has a thread stopped at the same location as the TOI (a *matching* process). TotalView runs these matching processes until one of its threads arrives at the goal. When this happens, TotalView stops the thread's process. The command finishes when it has stopped all of these *matching* processes.
- *Thread group*—TotalView runs all processes in the control group. However, as each thread arrives at the goal, TotalView only stops that thread; the rest of the threads in the same process continue executing. The command finishes when all threads in the GOI arrive at the goal. When the command finishes, TotalView stops all processes in the control group.

  TotalView doesn't wait for threads that are not in the same share group as the TOI, since they are executing different code and can never arrive at the goal.

Understanding Process Width

TotalView behavior when stepping at process width (which is the default) depends on whether the Group of Interest (GOI) is a process group or a thread group.

The actions that TotalView performs on the GOI are dependent on the type of process group that is the focus, as follows:

- *Process group*—TotalView runs all threads in the process, and execution continues until the TOI arrives at its goal, which can be the next statement, the next instruction, and so on. Only when the TOI reaches the goal does TotalView stop the other threads in the process.

- *Thread group*—TotalView lets all threads in the GOI and all manager threads run. As each member of the GOI arrives at the goal, TotalView stops it; the rest of the threads continue executing. The command finishes when all members of the GOI arrive at the goal. At that point, TotalView stops the whole process.

Understanding Thread Width

When TotalView performs a stepping command, it decides what it steps based on the *width*. Using the toolbar, you specify width using the left-most pulldown. This pulldown has three items: **Group**, **Process**, and **Thread**.

Stepping at thread width tells TotalView to only run that thread. It does not step other threads. In contrast, process width tells TotalView to run all threads in the process that are allowed to run while the TOI is stepped. While TotalView is stepping the thread, manager threads run freely.

Stepping a thread isn't the same as stepping a thread's process, because a process can have more than one thread.

NOTE: 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 use process-width stepping instead.

Using Run To and duntil Commands

The **duntil** and **Run To** commands differ from other step commands when you apply them to a process group. (These commands tell TotalView to execute program statements *until* it reaches the selected statement.) When used with a process group, TotalView identifies all processes in the group that already have a thread stopped at the goal. These are the *matching* processes. TotalView then runs only nonmatching processes. Whenever a thread arrives at the goal, TotalView stops its process. The command finishes when it has stopped all members of the group. This lets you synchronize all the processes in a group in preparation for group-stepping them.

You need to know the following if you're running at process width:

Process group — If the Thread of Interest (TOI) is already at the goal location, TotalView steps the TOI past the line before the process runs. This lets you use the Run To command repeatedly in loops.

Thread group	If any thread in the process is already at the goal, TotalView temporarily holds it while other threads in the process run. After all threads in the thread group reach the goal, TotalView stops the process. This lets you synchronize the threads in the POI at a source line.

If you're running at group width:

Process group	TotalView examines each process in the process and share group to determine whether at least one thread is already at the goal. If a thread is at the goal, TotalView holds its process. Other processes are allowed to run. When at least one thread from each of these processes is held, the command completes. This lets you synchronize at least one thread in each of these processes at a source line. If you're running a control group, this synchronizes all processes in the share group.
Thread group	TotalView examines all the threads in the thread group that are in the same share group as the TOI to determine whether a thread is already at the goal. If it is, TotalView holds it. Other threads are allowed to run. When all of the threads in the TOI's share group reach the goal, TotalView stops the TOI's *control* group and the command completes. This lets you synchronize thread group members. If you're running a workers group, this synchronizes all worker threads in the share group.

The process stops when the TOI and at least one thread from each process in the group or process being run reach the command stopping point. This lets you synchronize a group of processes and bring them to one location.

You can also run to a selected line in a nested stack frame, as follows:

1. Select a nested frame in the Stack Trace Pane.
2. Select a source line or instruction in the function.
3. Enter a **Run To** command.

TotalView executes the primary thread until it reaches the selected line in the selected stack frame.

RELATED TOPICS

Stepping commands	Using Stepping Commands on page 178
Running to a specific line	Executing to a Selected Line on page 182
The **duntil** command	**duntil** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **Group > Run To** command	**Group > Run To** in the in-product Help
The**Process > Run To** command	**Process > Run To** in the in-product Help
The **Thread > Run To** command	**Thread > Run To** in the in-product Help

Setting Process and Thread Focus

NOTE: The previous sections emphasize the GUI; this section and the ones that follow emphasize the CLI. Here you will find information on how to have full asynchronous debugging control over your program. Fortunately, having this level of control is seldom necessary. In other words, don't read the rest of this chapter unless you have to.

When TotalView executes a command, it must decide which processes and threads to act on. Most commands have a default set of threads and processes and, in most cases, you won't want to change the default. In the GUI, the default is the process and thread in the current Process Window. In the CLI, this default is indicated by the focus, which is shown in the CLI prompt.

There are times, however, when you need to change this default. This section begins a rather intensive look at how you tell TotalView what processes and threads to use as the target of a command.

Topics in this section are:

Understanding Process/Thread Sets

All TotalView commands operate on a set of processes and threads. This set is called a *Process/Thread (P/T) set.* The right-hand text box in windows that contain P/T set controls lets you construct these sets. In the CLI, you specify a P/T set as an argument to a command such as **dfocus**. If you're using the GUI, TotalView creates this list for you based on which Process Window has focus.

Unlike a serial debugger in which each command clearly applies to the only executing thread, TotalView can control and monitor many threads with their PCs at many different locations. The P/T set indicates the groups, processes, and threads that are the target of the CLI command. No limitation exists on the number of groups, processes, and threads in a set.

A P/T set is a list that contains one or more P/T identifiers. (The next section, Specifying Arenas on page 581, explains what a P/T identifier is.) Tcl lets you create lists in the following ways:

- You can enter these identifiers within braces (**{ }**).
- You can use Tcl commands that create and manipulate lists.

These lists are then used as arguments to a command. If you're entering one element, you usually do not have to use the Tcl list syntax.

For example, the following list contains specifiers for process 2, thread 1, and process 3, thread 2:

```
{p2.1 p3.2}
```

If you do not explicitly specify a P/T set in the CLI, TotalView defines a target set for you. (In the GUI, the default set is determined by the current Process Window.) This set is displayed as the default CLI prompt. (For information on this prompt, see About the CLI Prompt on page 466.)

You can change the focus on which a command acts by using the **dfocus** command. If the CLI executes the **dfocus** command as a unique command, it changes the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

After TotalView executes this command, all commands that follow focus on process 2.

NOTE: In the GUI, you set the focus by displaying a Process Window that contains this process. Do this by either using the P+, Px and P- buttons in the tab bar at the bottom, by making a selection in the Processes/Ranks Tab, or by clicking on a process in the Root Window. Note that the **Px** button launches a dialog box that enables you to enter a specific Process or Thread to focus on.

If you begin a command with **dfocus**, TotalView changes the target only for the command that follows. After the command executes, TotalView restores the *former* default. The following example shows both of these ways to use the **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then step the threads in this group twice:

```
dfocus g2
dstep
dstep
```

In contrast, if the current focus is process 1, thread 1, the following commands step group 2 and then step process 1, thread 1:

```
dfocus g2 dstep
dstep
```

Some commands only operate at the process level; that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Specifying Arenas

A P/T identifier often indicates a number of groups, processes, and threads. For example, assume that two threads executing in process 2 are stopped at the same statement. This means that TotalView places the two stopped threads into lockstep groups. If the default focus is process 2, stepping this process actually steps both of these threads.

TotalView uses the term *arena* to define the processes and threads that are the target of an action. In this case, the arena has two threads. Many CLI commands can act on one or more arenas. For example, the following command has two arenas:

```
dfocus {p1 p2}
```

The two arenas are process 1 and process 2.

When there is an arena list, each arena in the list has its own GOI, POI, and TOI.

Specifying Processes and Threads

The previous sections described P/T sets as being lists; however, these discussions ignored what the individual elements of the list are. A better definition is that a P/T set is a list of arenas, where an *arena* consists of the processes, threads, and groups that are affected by a debugging command. Each *arena specifier* describes a single arena in which a command acts; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some CLI output commands, however, combine arenas and act on them as a single target.

An arena specifier includes a *width* and a TOI. (Widths are discussed later in this section.) In the P/T set, the TOI specifies a target thread, while the width specifies how many threads surrounding the thread of interest are affected.

Defining the Thread of Interest (TOI)

The TOI is specified as **p.t**, where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID). The **p.t** combination identifies the POI (Process of Interest) and TOI. The TOI is the primary thread affected by a command. This means that it is the primary focus for a TotalView command. For example, while the **dstep** command always steps the TOI, it may also run the rest of the threads in the POI and step other processes in the group.

In addition to using numerical values, you can also use two special symbols:

- The less-than character (**<**) indicates the *lowest numbered worker thread* in a process, and is used instead of the TID value. If, however, the arena explicitly names a thread group, the **<** symbol means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which is not necessarily thread 1.

- A dot (**.**) indicates the current set. Although you seldom use this symbol interactively, it can be useful in scripts.

About Process and Thread Widths

You can enter a P/T set in two ways. If you're not manipulating groups, the format is as follows:

[*width_letter***][***pid***][.***tid***]**

NOTE: Specifying Groups in P/T Sets on page 586 extends this format to include groups. When using P/T sets, you can create sets with just width indicators or just group indicators, or both.

For example, **p2.3** indicates process 2, thread 3.

Although the syntax seems to indicate that you do not need to enter any element, TotalView requires that you enter at least one. Because TotalView tries to determine what it can do based on what you type, it tries to fill in what you omit. The only requirement is that when you use more than one element, you use them in the order shown here.

You can leave out parts of the P/T set if what you do enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be <. For more information, see Naming Incomplete Arenas on page 596.

The *width_letter* indicates which processes and threads are part of the arena. You can use the following letters:

t *Thread width*

A command's target is the indicated thread.

p *Process width*

A command's target is the process that contains the TOI.

g *Group width*

A command's target is the group that contains 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.

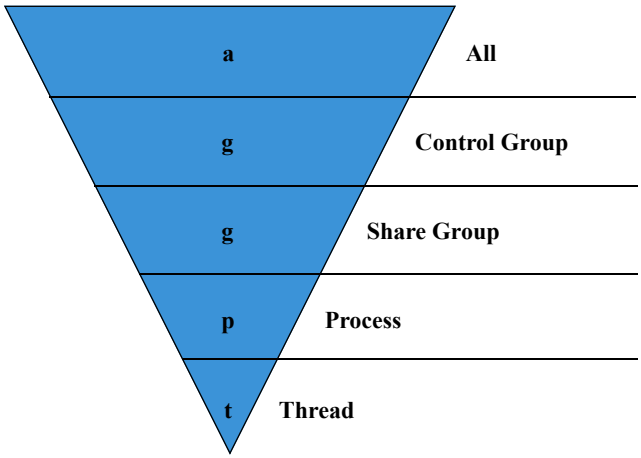
d *Default width*

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.

You must use lowercase letters to enter these widths.

Figure 256 illustrates how these specifiers relate to one another.

Figure 256, Width Specifiers



The **g** specifier indicates control and share groups. This inverted triangle indicates that the arena focuses on a greater number of entities as you move from **Thread** level at the bottom to **All** level at the top.

As mentioned previously, the TOI specifies a target thread, while the width specifies how many threads surrounding the TOI are also affected. For example, the **dstep** command always requires a TOI, but entering this command can do the following:

- Step just the TOI during the step operation (thread-level single-step).
- Step the TOI and step all threads in the process that contain the TOI (process-level single-step).
- Step all processes in the group that have threads at the same PC as the TOI (group-level single-step).

This list doesn't indicate what happens to other threads in your program when TotalView steps your thread. For more information, see Stepping (Part II): Examples on page 598.

To save a P/T set definition for later use, assign the specifiers to a Tcl variable; for example:

```
set myset {g2.3 t3.1}
dfocus $myset dgo
```

As the **dfocus** command returns its focus set, you can save this value for later use; for example:

```
set save_set [dfocus]
```

Specifier Examples

The following are some sample specifiers:

g2.3	Select process 2, thread 3, and set the width to *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 less-than symbol (**<**) sets the TID to the first user thread.

Setting Group Focus

TotalView has two types of groups: process groups and thread groups. Process groups only contain processes, and thread groups only contain threads. The threads in a thread group can be drawn from more than one process.

Topics in this section are:

For a general discussion on how TotalView organizes threads and processes into groups, see About Groups, Processes, and Threads on page 383.

TotalView has four predefined groups. Two of these only contain processes, while the other two only contain threads. TotalView also lets you create your own groups, and these groups can have elements that are processes and threads. The following are the predefined process groups:

- **Control Group**

  Contains the parent process and all related processes. A control group includes children that were forked (processes that share the same source code as the parent) and children that were forked but subsequently called the **execve()** function.

  Assigning a new value to the **CGROUP (**dpid**)** variable for a process changes that process's control group. In addition, the **dgroups -add** command lets you add members to a group in the CLI. In the GUI, you use the **Group > Custom Groups** command.

- **Share Group**

  Contains all members of a control group that share the same executable. TotalView automatically places processes in share groups based on their control group and their executable.

NOTE: You can't change a share group's members. However, the dynamically loaded libraries used by group members can be different.

In general, if you're debugging a multi-process program, the control group and share group differ only when the program has children that it forked by calling the **execve()** function.

The following are the predefined thread groups:

- **Workers Group**

  Contains all worker threads from all processes in the control group. The only threads not contained in a workers group are your operating system's manager threads.

- **Lockstep Group**

  Contains every stopped thread in a share group that has the same PC. TotalView creates one lockstep group for every thread. For example, suppose two threads are stopped at the same PC. TotalView creates two lockstep groups. While each lockstep group has the same two members, they differ in that each has a different TOI. While there are some circumstances where this is important, you can usually ignore this distinction. That is, while two lockstep groups exist if two threads are stopped at the same PC, ignoring the second lockstep group is almost never harmful.

  The group ID value for a lockstep group differs from the ID of other groups. Instead of having an automatic 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**.

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. The CLI only displays a target group in the focus string if you set it to something other than the default value.

NOTE: You most often use target group specifiers with the stepping commands, as they give these commands more control over what's being stepped.

Use the following format to add groups to an arena specifier:

[*width_letter***][***group_indicator***][***pid***][.***tid***]**

This format adds the *group_indicator* to what was discussed in Specifying Processes and Threads on page 581.

In the description of this syntax, everything appears to be optional. But, while no single element is required, you must enter at least one element. TotalView determines other values based on the current focus.

TotalView lets you identify a group by using a letter, number, or name.

A Group Letter

You can name one of TotalView's predefined sets. Each set is identified by a letter. For example, the following command sets the focus to the workers group:

```
dfocus W
```

The following are the group letters. These letters are in uppercase:

C *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 this letter tells the CLI not to display a group letter in 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 that contains all threads in the share group that have the same PC as the arena's TOI. If you step these threads as a group, they proceed in lockstep.

A Group Number

You can identify a group by the number TotalView assigns to it. The following example sets the focus to group 3:

```
dfocus 3/
```

The trailing slash tells TotalView that you are specifying a group number instead of a PID. The slash character is optional if you're using a *group_letter*. However, you must use it as a separator when entering a numeric group ID and a **pid.tid** pair. For example, the following example identifies process 2 in group 3:

```
p3/2
```

A Group Name

You can name a set that you define. You enter this name with slashes. The following example sets the focus to the set of threads contained in process 3 that are also contained in a group called **my_group**:

```
dfocus p/my_group/3
```

About Arena Specifier Combinations

The following table lists what's selected when you use arena and group specifiers to step your program:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."
gC	All threads in the Thread of Interest (TOI) control group.
gS	All threads in the TOI share group.
gW	All worker threads in the control group that contains the TOI.
gL	All threads in the same share group within the process that contains the TOI that have the same PC as the TOI.
pC	All threads in the control group of the Process of Interest (POI). This is the same as **gC**.
pS	All threads in the process that participate in the same share group as the TOI.
pW	All worker threads in the POI.
pL	All threads in the POI whose PC is the same as the TOI.
tC	Just the TOI. The **t** specifier overrides the group specifier, so all of these specifiers resolve to the current thread.
tS	
tW	
tL	

NOTE: Stepping commands behave differently if the group being stepped is a process group rather than a thread group. For example, **aC** and **aS** perform the same action, but **aL** is different.

If you don't add a PID or TID to your arena specifier, TotalView does it for you, taking the PID and TID from the current or default focus.

The following are some additional examples. These examples add PIDs and TIDs numbers to the raw specifier combinations listed in the previous table:

pW3 All worker threads in process 3.

pW3.< All worker threads in process 3. The focus of this specifier is the same as the focus in the previous example.

gW3 All worker threads in the control group that contains process 3. The difference between this and `pW3` is that **pW3** restricts the focus to just 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. The arena width, POI, and TOI are inherited from the existing P/T set, so the exact meaning of this specifier depends on the previous context.

While the slash is unnecessary because no group is indicated, it is syntactically correct.

g3.2/3 The **3.2** group ID is the name of the lockstep group for thread 3.2. This group includes all threads in the process 3 share group that are executing at the same PC as thread 2.

p3/3 Sets the process to process 3. The Group of Interest (GOI) is set to group 3. If group 3 is a process group, most commands ignore the group setting. If group 3 is a thread group, most commands act on all threads in process 3 that are also in group 3.

When you set the process using an explicit group, you might not be including all the threads you expect to be included. This is because commands must look at the TOI, POI, and GOI.

NOTE: It is redundant to specify a thread width with an explicit group ID as this width means that the focus is on one thread.

In the following examples, the first argument to the **dfocus** command defines a temporary P/T set that the CLI command (the last term) operates on. The **dstatus** command lists information about processes and threads. These examples assume that the global focus was **d1.<** initially.

dfocus g dstatus

Displays the status of all threads in the control group.

dfocus gW dstatus

Displays the status of all worker threads in the control group.

dfocus p dstatus

Displays the status of all worker threads in the current focus process. The width here, as in the previous example, is process, and the (default) group is the control group. The intersection of this width and the default group creates a focus that is the same as in the previous example.

dfocus pW dstatus

Displays the status of all worker threads in the current focus process. The width is process level, and the target is the workers group.

The following example shows how the prompt changes as you change the focus. In particular, notice how the prompt changes when you use the **C** and the **D** group specifiers.

```
d1.<> f C
dC1.<
dC1.<> f D
d1.<
d1.<>
```

Two of these lines end with the less-than symbol (<). These lines are not prompts. Instead, they are the value returned by TotalView when it executes the **dfocus** command.

'All' Does Not Always Mean 'All'

When you use stepping commands, TotalView determines the scope of what runs and what stops by looking at the TOI. This section looks at the differences in behavior when you use the **a** (all) arena specifier. The following table describes what runs when you use this arena:

Specifier	Specifies
aC	All threads.
aS	All threads.
aW	All threads in all workers groups.
aL	All threads. Every thread is a member of a control group and a member of a share group and a member of a lockstep group. Consequently, three of these definitions mean "all threads."

The following are some combinations:

f aC dgo

Runs everything. If you're using the **dgo** command, everything after the **a** is ignored: **a/aPizza/17.2**, **ac**, **aS**, and **aL** do the same thing. TotalView runs everything.

f aC duntil

While everything runs, TotalView must wait until something reaches a goal. It really isn't obvious what this focus is. Since **C** is a process group, you might guess that all processes run until at least one thread in every participating process arrives at a goal. The reality is that since this goal must reside in the current share group, this command completes as soon as all processes in the TOI share group have at least one thread at the goal. Processes in other control groups run freely until this happens.

	The TOI determines the goal. If there are other control groups, they do not participate in the goal.
f aS duntil	This command does the same thing as the **f aC duntil** command because the goals for **f aC duntil** and **f aS duntil** are the same, and the processes that are in this scope are identical. Although more than one share group can exist in a control group, these other share groups do not participate in the goal.
f aL duntil	Although everything will run, it is not clear what should occur. **L** is a thread group, so you might expect that the **duntil** command will wait until all threads in all lockstep groups arrive at the goal. Instead, TotalView defines the set of threads that it allows to run to a goal as just those threads in the TOI's lockstep group. Although there are other lockstep groups, these lockstep groups do not participate in the goal. So, while the TOI's lockstep threads are progressing towards their goal, all threads that were previously stopped run freely.
f aW duntil	Everything runs. TotalView waits until all members of the TOI workers group arrive at the goal.

Two broad distinctions between process and thread group behavior exist:

- When the focus is on a process group, TotalView waits until just one thread from each participating process arrives at the goal. The other threads just run.

  When focus is on a thread group, every participating thread must arrive at the goal.

- When the focus is on a process group, TotalView steps a thread over the goal breakpoint and continues the process if it isn't the right thread.

  When the focus is on a thread group, TotalView holds a thread even if it isn't the right thread. It also continues the rest of the process.

  If your system does not support asynchronous thread control, TotalView treats thread specifiers as if they were process specifiers.

With this in mind, **f aL dstep** does not step all threads. Instead, it steps only the threads in the TOI's lockstep group. All other threads run freely until the stepping process for these lockstep threads completes.

Setting Groups

This section presents a series of examples that set and create groups.

You can use the following methods to indicate that thread 3 in process 2 is a worker thread:

`dset WGROUP(2.3) $WGROUP(2)`
: Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a nonzero value to **WGROUP** indicates that this is a worker group.)

```
dset WGROUP(2.3) 1
```
This is a simpler way of doing the same thing as the previous example.

```
dfocus 2.3 dworker 1
```
Adds the groups in the indicated focus to a workers group.

```
dset CGROUP(2) $CGROUP(1)
dgroups -add -g $CGROUP(1) 2
dfocus 1 dgroups -add 2
```
These three commands insert process 2 into the same control group as process 1.

```
dgroups -add -g $WGROUP(2) 2.3
```
Adds process 2, thread 3 to the workers group associated with process 2.

```
dfocus tW2.3 dgroups -add
```
This is a simpler way of doing the same thing as the previous example.

Following are some additional examples:

```
dfocus g1 dgroups -add -new thread
```
Creates a new thread group that contains all the threads in all the processes in the control group associated with process 1.

```
set mygroup [dgroups -add -new thread $GROUP($SGROUP(2))]
dgroups -remove -g $mygroup 2.3
dfocus g$mygroup/2 dgo
```
The first command creates a new group that contains all the threads from the process 2 share group; the second removes thread 2.3; and the third runs the remaining threads.

RELATED TOPICS	
The **dfocus** command	**dfocus** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **dgroup** command	**dgroup** in "CLI Commands" in the *Classic TotalView Reference Guide*
The **dset** command	**dset** in "CLI Commands" in the *Classic TotalView Reference Guide*

Using the g Specifier: An Extended Example

The meaning of the **g** width specifier is sometimes not clear when it is coupled with a group scope specifier. Why have a **g** specifier when you have four other group specifiers? Stated in another way, isn't something like **gL** redundant?

The simplest answer, and the reason you most often use the **g** specifier, is that it forces the group when the default focus indicates something different from what you want it to be.

The following example shows this. The first step sets a breakpoint in a multi-threaded OMP program and executes the program until it hits the breakpoint.

```
d1.<> dbreak 35
Loaded OpenMP support library libguidedb_3_8.so :
                  KAP/Pro Toolset 3.8
1
d1.<> dcont
Thread 1.1 has appeared
Created process 1/37258, named "omp_prog"
Thread 1.1 has exited
Thread 1.1 has appeared
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.1 hit breakpoint 1 at line 35 in ".breakpoint_here"
```

The default focus is **d1.<**, which means that the CLI is at its default width, the POI is 1, and the TOI is the lowest numbered nonmanager thread. Because the default width for the **dstatus** command is process, the CLI displays the status of all processes. Typing **dfocus p dstatus** produces the same output.

```
d1.<> dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
d1.<> dfocus p dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Stopped     PC=0xd042c944
```

The CLI displays the following when you ask for the status of the lockstep group. (The rest of this example uses the **f** abbreviation for **dfocus**, and **st** for **dstatus**.)

```
d1.<> f L st
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

This command tells the CLI to display the status of the threads in thread, which is the 1.1 lockstep group since this thread is the TOI. The **f L focus** command narrows the set so that the display only includes the threads in the process that are at the same PC as the TOI.

NOTE: By default, the **dstatus** command displays information at process width. This means that you don't need to type **f pL dstatus**.

The **duntil** command runs thread 1.3 to the same line as thread 1.1. The **dstatus** command then displays the status of all the threads in the process:

```
d1.<> f t1.3 duntil 35
    35@>            write(*,*)"i= ",i,
                      "thread= ",omp_get_thread_num()
d1.<> f p dstatus
1:      37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

As expected, the CLI adds a thread to the lockstep group:

```
d1.<> f L dstatus
1:      37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3  Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

The next set of commands begins by narrowing the width of the default focus to thread width—notice that the prompt changes—and then displays the contents of the lockstep group:

```
d1.<> f t
t1.<> f L dstatus
1:      37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

Although the lockstep group of the TOI has two threads, the current focus has only one thread, and that thread is, of course, part of the lockstep group. Consequently, the lockstep group *in the current focus* is just the one thread, even though this thread's lockstep group has two threads.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group of thread 1.1. as follows:

```
t1.<> f pL dstatus
1:      37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
t1.<> f gL dstatus
1:      37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                     [./omp_prog.f#35]
```

```
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
```

NOTE: If the TOI is 1.1, **L** refers to group number 1.1, which is the lockstep group of thread 1.1.

Because this example only contains one process, the **pL** and **gL** specifiers produce the same result when used with the **dstatus** command. If, however, there were additional processes in the group, you only see them when you use the **gL** specifier.

Merging Focuses

When you specify more than one focus for a command, the CLI merges them. In the following example, the focus indicated by the prompt—this focus is called the *outer* focus—controls the display. This example shows what happens when **dfocus** commands are strung together:

```
t1.<> f d
d1.<
d1.<> f tL dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
d1.<> f tL f p dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
d1.<> f tL f p f D dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
    1.2: 37258.2 Stopped     PC=0xffffffffffffffff
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
d1.<> f tL f p f D f L dstatus
1:       37258   Breakpoint  [omp_prog]
    1.1: 37258.1 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
    1.3: 37258.3 Breakpoint  PC=0x1000acd0,
                    [./omp_prog.f#35]
```

Stringing multiple focuses together might not produce the most readable result. In this case, it shows how one **dfocus** command can modify what another sees and acts on. The ultimate result is an arena that a command acts on. In these examples, the **dfocus** command tells the **dstatus** command what to display.

Naming Incomplete Arenas

In general, you do not need to completely specify an arena. TotalView provides values for any missing elements. TotalView either uses built-in default values or obtains them from the current focus. The following explains how TotalView fills in missing pieces:

- If you don't use a width, TotalView uses the width from the current focus.
- If you don't use a PID, TotalView uses the PID from the current focus.
- If you set the focus to a list, there is no longer a default arena. This means that you must explicitly name a width and a PID. You can, however, omit the TID. (If you omit the TID, TotalView defaults to the less-than symbol <.)

  You can type a PID without typing a TID. If you omit the TID, TotalView uses the default **<**, where **<** indicates the lowest numbered worker thread in the process. If, however, the arena explicitly names a thread group, **<** means the lowest numbered member of the thread group.

  TotalView does not use the TID from the current focus, since the TID is a process-relative value.
- A dot before or after the number specifies a process or a thread. For example, **1.** is clearly a PID, while **.7** is clearly a TID.

  If you type a number without typing a dot, the CLI most often interprets the number as being a PID.
- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and don't specify a PID or TID, TotalView uses the PID and TID from the current focus.

  If you use a letter as a group specifier, TotalView obtains the rest of the arena specifier from the default focus.
- You can use a group ID or tag followed by a **/**. TotalView obtains the rest of the arena from the default focus.

Focus merging can also influence how TotalView fills in missing specifiers. For more information, see Merging Focuses on page 595.

Naming Lists with Inconsistent Widths

TotalView lets you create lists that contain more than one width specifier. This can be very useful, but it can be confusing. Consider the following:

```
{p2 t7 g3.4}
```

This list is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should TotalView use this set of processes, groups, and threads?

In most cases, TotalView does what you would expect it to do: it iterates over the list and acts on each arena. If TotalView cannot interpret an inconsistent focus, it prints an error message.

Some commands work differently. Some use each arena's width to determine the number of threads on which it acts. This is exactly what the **dgo** command does. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the TOI. TotalView may wait for threads in multiple processes for group-level arenas. The command description in the *Classic TotalView Reference Guide* points out anything that you need to watch out for.

Stepping (Part II): Examples

The following are examples that use the CLI stepping commands:

- **Step a single thread**

  While the thread runs, no other threads run (except kernel manager threads).

  Example: `dfocus t dstep`

- **Step a single thread while the process runs**

  A single thread runs into or through a critical region.

  Example: `dfocus p dstep`

- **Step one thread in each process in the group**

  While one thread in each process in the share group runs to a goal, the rest of the threads run freely.

  Example: `dfocus g dstep`

- **Step all worker threads in the process while nonworker threads run**

  Worker threads run through a parallel region in lockstep.

  Example: `dfocus pW dstep`

- **Step all workers in the share group**

  All processes in the share group participate. The nonworker threads run.

  Example: `dfocus gW dstep`

- **Step all threads that are at the same PC as the TOI**

  TotalView selects threads from one process or the entire share group. This differs from the previous two items in that TotalView uses the set of threads 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 in one statement. This command is the built-in alias for **dfocus gL dstep**.

sl Finds all threads in the current process that are at the same PC as the TOI, and steps them all in one statement. This command is the built-in alias for **dfocus L dstep**.

RELATED TOPICS

Stepping through your program	Using Stepping Commands on page 178
Stepping (Part 1)	Stepping (Part I) on page 575

Using P/T Set Operators

At times, you do not want all of one type of group or process to be in the focus set. TotalView lets you use the following three operators to manage your P/T sets:

|	Creates a union; that is, all members of two sets.
-	Creates a difference; that is, all members of the first set that are not also members of the second set.
&	Creates an intersection; that is, all members of the first set that are also members of the second set.

For example, the following creates a union of two P/T sets:

```
p3 | L2
```

You can, apply these operations repeatedly; for example:

```
p2 | p3 & L2
```

This statement creates an intersection between p3 and L2, and then creates a union between **p2** and the results of the intersection operation. You can directly specify the order by using parentheses; for example:

```
p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint(*ptset*)	Returns a list of all threads that are stopped at a breakpoint.
comm(***process***, "***comm_name***")	Returns a list containing the first thread in each process associated within a communicator within the named process. While ***process*** is a P/T set it is not expanded into a list of threads.
error(*ptset*)	Returns a list of all threads stopped due to an error.
existent(*ptset*)	Returns a list of all threads.
held(*ptset*)	Returns a list of all threads that are held.
nonexistent(*ptset*)	Returns a list of all processes that have exited or which, while loaded, have not yet been created.
running(*ptset*)	Returns a list of all running threads.
stopped(*ptset*)	Returns a list of all stopped threads.
unheld(*ptset*)	Returns a list of all threads that are not held.

watchpoint(*ptset*) Returns a list of all threads that are stopped at a watchpoint.

The way in which you specify the P/T set argument is the same as the way that you specify a P/T set for the **dfocus** command. For example, **watchpoint(L)** returns all threads in the current lockstep group. The only operator that differs is **comm**, whose argument is a process.

The dot operator (**.**), which indicates the current set, can be helpful when you are editing an existing set.

The following examples clarify how you use these operators and functions. The P/T set **a** (all) is the argument to these operators.

f {breakpoint(a) | watchpoint(a)} dstatus
Shows information about all threads that are stopped at breakpoints and watchpoints. The a argument is the standard P/T set indicator for **all**.

f {stopped(a) - breakpoint(a)} dstatus
Shows information about all stopped threads that are not stopped at breakpoints.

f {. | breakpoint(a)} dstatus
Shows information about all threads in the current set, as well as all threads stopped at a breakpoint.

f {g.3 - p6} duntil 577
Runs thread 3 along with all other processes in the group to line 577. However, it does not run anything in process 6.

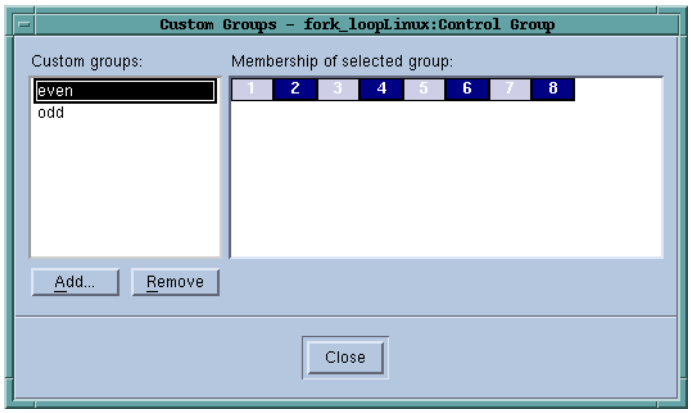
f {($PTSET) & p123}
Uses just process 123 in the current P/T set.

Creating Custom Groups

Debugging a multi-process or multi-threaded program most often focuses on running the program in one of two ways: either you run everything or run one or two things. Figuring out what you should be running, however, is a substantial part of the art of debugging. You can make things easier on yourself if you divide your program into groups, and then control these groups separately. When you need to do this, use the **Groups > Custom Groups** Dialog Box. (See Figure 257.) This dialog box also lets you alter a group's contents as well as delete the group.

NOTE: You can manage only process groups with this dialog box. Thread groups can only be managed using the CLI. In addition, the groups you create must reside within one control group.

Figure 257, Group > Custom Groups Dialog Box



When you first display this dialog box, TotalView also displays a second, used to enter the group's name.

The dialog's right side contains a set of boxes. Each represents one of your processes. The initial color represents the process's state. (This just helps you coordinate within the display in the Process Window's Processes/Ranks Tab.) You can now create a group using your mouse by clicking on blocks as follows:

- **Left-click on a box**: Selects a box. No other box is selected. If other boxes are selected, they are deselected.
- **Shift-left-click and drag**: select a group of contiguous boxes.
- **Control-left-click on a box**: Adds a box to the current selection.

Edit an existing group in the same way. After making the group active by clicking on its name on the left, click within the right to make changes. (In most cases, you'll be using a control-left-click.)

If you've changed a group and then select **Add** or **Close**, TotalView asks if you want to save the changed group.

If you click **Add** when a group is selected, TotalView creates a group with the same members as that group.

Finally, you can delete a group by selecting its name and clicking **Remove**.

Scalability in HPC Computing Environments

TotalView provides features and performance enhancements for scalable debugging in today's HPC computing environments, and no special configuration or action is necessary on your part to take advantage of TotalView's scalability abilities.

This chapter details TotalView's features and configurations related to scalability, as follows:

- **Root Window.** The Root Window aggregates program state so that it can display quickly and is easy to understand.
- **Scalability Configuration Settings**. Depending on your needs, you might want to set specific configuration variables that enable scalable debugging operations.
- **MRNet Configuration Settings**. TotalView uses MRNet, a tree-based overlay network, for scalable communication. TotalView is preconfigured for scalability, but in some situations you may want to change MRNet's configuration.
- **dstatus and dwhere command options**. These options provide aggregated views of various process and thread properties. (See the **-group_by** option in the Reference Guide entries for these commands.)
- **Compressed process/thread list**. The **ptlist** compactly displays the set of processes and threads that have been aggregated together.

RELATED TOPICS	
Compressed List Syntax (**ptlist**)	"Compressed List Syntax (ptlist)" in the **dstatus** entry of the *Reference Guide*
the **Px** "Jump to Process/Thread" button	Using the Processes/Ranks and Threads Tabs on page 418
dstatus -group_by and **dwhere-group_by** options	**dstatus** and **dwhere** in the *Reference Guide*

Configuring TotalView for Scalability

To take advantage of TotalView's features that support better scalability, disable user-thread debugging. User thread debugging is an area of the debugger that has not yet been parallelized, and can therefore slow down job launch and attach time. However, disabling user thread debugging also disables support for displaying thread local storage (e.g., via the `__thread` compiler keyword. This limitation will be fixed in a future release.

To configure TotalView with these settings, create a TotalView startup file in **<***totalviewInstallDir***>/<***PLATFORM***>/lib/.tvdrc** and add the following lines:

```
# If TLS is not required, disable user threads for faster launch
# and attach times
dset -set_as_default TV::user_threads false
```

Process Window's Process Tab

By default, TotalView 8.15.0 and later suppress the display of the process grid in the Processes tab, because it can have a negative impact when scaling to a large number of processes. If debugger scalability is not a concern and you prefer to display the Processes tab in the Process Window, specify the **-process_grid** option or set the **TV::GUI:process_grid_wanted** state variable to **true** in the startup file.

To display the Process Window's Processes tab when you start TotalView, pass TotalView the **-process_grid** command option:

totalview -process_grid

To always display the Processes tab in the Process Window by default, set the state variable **TV::GUI:process_grid_wanted** to **true** for use when initializing TotalView:

```
dset TV::GUI:process_grid_wanted true
```

dlopen Options

When a target process calls `dlopen()`, a **dlopen** event is generated and must be handled by TotalView. Because **dlopen** event handling can affect debugger performance for a variety of reasons, especially if the application loads many shared libraries or the debugger is controlling many processes, TotalView provides ways to configure **dlopen** for better performance and scalability in HPC computing environments:

- Filtering **dlopen** events to avoid stopping a process for each event.
- Handling **dlopen** events in parallel, reducing client/server communication overhead to fetch library information. **Note**: Both this option and MRNet must be enabled for TotalView to fetch libraries in parallel.

dlopen Event Filtering

You can filter **dlopen** events to plant breakpoints in the *dlopened* libraries only when the process stops for some other reason. Deferring **dlopen** event processing allows the debugger to handle all dynamically loaded shared libraries at the same time, which is much more efficient than handling them serially.

A robust combination of settings support a range of options, so that you can finely control which **dlopen** events are reported immediately, and which are deferred.

Filtering **dlopen** events may be particularly beneficial when using Open MPI or other highly dynamic runtime libraries.

For detail, see "Filtering dlopen Events" in the *Classic TotalView Reference Guide.*

Handling dlopen Events in Parallel

TotalView's default behavior is to handle *dlopened* libraries serially, creating multiple, single-cast client-server communications. This can degrade performance, depending on the number of libraries a process *dlopens*, and the number of processes in the job.

A state variable and a command line option support handling events in parallel: **TV::dlopen_read_libraries_in_parallel** and **dlopen_read_libraries_in_parallel**, discussed in the *Classic TotalView Reference Guide*

To handle *dlopened* libraries in parallel using MRNet, enter the following in your **tvdrc** file so that all future invocations of TotalView will have this set:

```
dset TV::dlopen_read_libraries_in_parallel true
```

Or for a single invocation of TotalView, simply launch TotalView using the command parameter:

```
totalview -dlopen_read_libraries_in_parallel
```

NOTE: Enabling this option does not guarantee that **dlopen** performance will improve on all systems in all scenarios. Be sure to test the impact of this setting on your system and debugging environments.

Remember that MRNet must also be enabled for this to work.

MRNet

MRNet stands for "Multicast/Reduction Network." MRNet uses a tree-based front-end to back-end communication model to significantly improve the efficiency of data multicast and aggregation for front-end tools running on massively parallel systems.

The following description is from the MRNet web site (http://www.paradyn.org/mrnet/):

> MRNet is a software overlay network that provides efficient multicast and reduction communications for parallel and distributed tools and systems. MRNet uses a tree of processes between the tool's front-end and back-ends to improve group communication performance. These internal processes are also used to distribute many important tool activities, reducing data analysis time and keeping tool front-end loads manageable.
>
> MRNet-based tool components communicate across logical channels called streams. At MRNet internal processes, filters are bound to these streams to synchronize and aggregate dataflows. Using filters, MRNet can efficiently compute averages, sums, and other more complex aggregations and analyses on tool data. MRNet also supports facilities that allow tool developers to dynamically load new tool-specific filters into the system.

TotalView's use of MRNet is part of a larger strategy to improve the scalability of TotalView as high-end computers grow into very high process and thread counts.

TotalView supports MRNet on Linux x86_64, Linux ARM64, Linux PowerLE clusters, and Cray XT/XE/XK/XC.

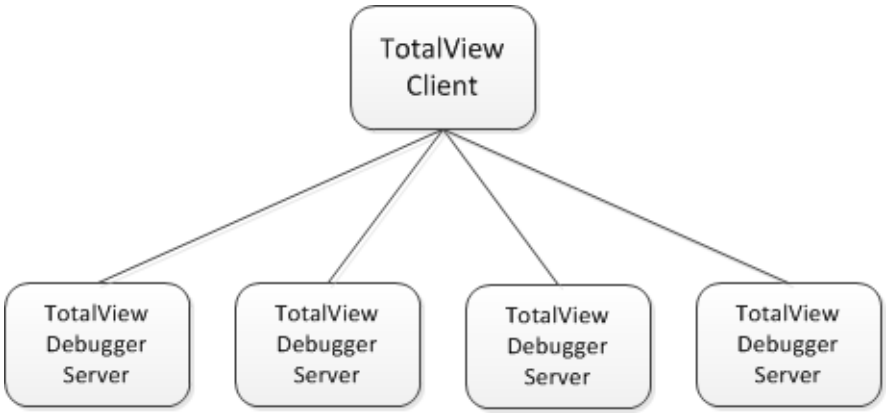
TotalView Infrastructure Models

Starting with TotalView 8.11.0, the TotalView debugger supported two infrastructure models that control the way the debugger organizes its TotalView debugger server processes when debugging a parallel job involving multiple compute nodes. Starting with TotalView 8.15, TotalView uses the tree-based infrastructure described below by default.

The first model uses a "flat vector" of TotalView debugger server processes. The TotalView debugger has always supported this model, and still does. Under the flat vector model, the debugger server processes have a direct (usually socket) connection to the TotalView front-end client. This model works well at low process scales, but begins to degrade as the target application scales beyond a few thousand nodes or processes. This is the default infrastructure model.

Figure 258 shows the TotalView client connected to four TotalView debugger servers (`tvdsvr`). In this example, four separate socket channels directly connect the client to the debugger servers.

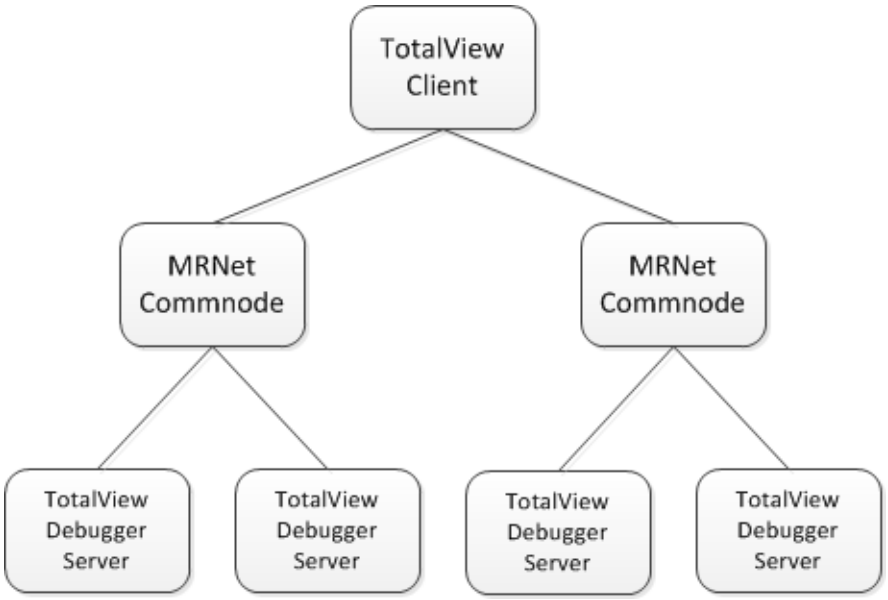
Figure 258, Flat Vector of Servers Infrastructure Model



The second model uses MRNet to form a tree of debugger server and MRNet communication processes connected to the TotalView front-end client, which forms the root of the tree. MRNet supports building many different shapes of trees, but note that the shape of the tree (for example, depth and fan-out) can greatly affect the performance of the debugger. The following sections describe how to control the shape of the MRNet tree in TotalView.

Figure 259 shows an MRNet tree in which the TotalView client is connected to four TotalView debugger servers through two MRNet commnode processes using a tree fan-out value of 2.

Figure 259, MRNet Infrastructure Model



Using MRNet with TotalView

TotalView is already preconfigured for maximum scalability, so no further customization is necessary. This section is for advanced users and describes TotalView options and state variables related to the use of MRNet with TotalView, as follows:blue

- General Use
- Using MRNet on Cray Computers

Please refer to the TotalView documentation for a general description of how options and state variables can be used with TotalView.

General Use

This section discusses basic configuration options of MRNet with TotalView. If you are working on a Cray computer, you will need to look at the section specific to that system as well.

Disabling MRNet Before Startup

By default, TotalView uses the MRNet infrastructure on the platforms where it is supported (see the *TotalView Platforms Guide* for specifics). On platforms where MRNet is not supported, TotalView uses its standard vector-of-servers infrastructure.

If for some reason you do not want to use the MRNet infrastructure to debug an MPI job, you must first disable MRNet in TotalView before launching the MPI job. MRNet can be disabled by:

- Starting TotalView with the **-nomrnet** option:

  ```
  prompt> totalview -nomrnet
  ```

- With TotalView running, use the command line interface (CLI) to set the **TV::mrnet_enabled** state variable:

  ```
  prompt> dset TV::mrnet_enabled false
  ```

MRNet Server Launch String

Option: -mrnet_server_launch_string *string*

State variable: TV::mrnet_server_launch_string *string*

Default string: `%B/tvdsvr%K -working_directory %D -set_pw %P -verbosity %V %F`

The server launch string defines configuration options when launching a debugging server. TotalView has a default string it uses when launching a server using the vector-of-servers architecture, and an option and state variable that allow you to modify the default string. The MRNet usage of TotalView also has a default launch string and corresponding option and state variables.

The MRNet launch string differs from the standard launch string in two ways: it does not contain a remote shell command expansion (e.g., `rsh` or `ssh`), and it has no `-callback` option.

TotalView always appends the following string to the expanded MRNet launch string:

```
-mrnet_launch node_id
```

where `node_id` is an integer that specifies the server's TotalView node ID within the job. If `node_id` is `0`, the server assigns itself a node ID equal to its MRNet rank plus 1.

Controlling the Shape of an MRNet Tree

The shape of the MRNet tree calculated by TotalView can be controlled through a collection of options and state variables. Given the list of hosts, which is typically extracted from the MPIR proctable, TotalView calculates an MRNet topology string to create various shapes of trees.

These are the basic controls:

- **Tree fan-out:** specifies the maximum number of children a node can have. If the number of leaves in the tree is not a power of the fan-out, some of the tree nodes will have fewer children.
- **Tree depth:** specifies the maximum depth of the tree (that is, the number of levels below the root). If the number of leaves is not greater than the square of the tree depth value, a shallower tree is built.
- **Extra root node**: Whether to allocate an extra communications node below the root.
- **Create a "super bushy" tree**: Create one debugger server process per MPI process rather than the default of creating one debugger server process per node, to overcome a CUDA limitation.

MRNet Tree Fan-Out

Option: –mrnet_fanout *integer*

State variable: TV::mrnet_fanout *integer*

Default value: 32

If you change the default value, the new value must be greater than or equal to **2** and less than or equal to **32768**.

MRNet Tree Depth

Option: –mrnet_levels *integer*

State variable: TV::mrnet_levels *integer*

Default value: 2

The MRNet tree depth can be specified in terms of the number of levels below the root. If you change the default value, the new value must be greater than or equal to **-2** and less than or equal to **32**.

- If the tree depth is **0**, the MRNet tree fan-out value is used, and TotalView attempts to honor the fan-out value near the bottom of the tree (the leaves).
- If the tree depth is set to a value that is **greater than 0** (which includes the default value of 2), the fan-out value is ignored and a balanced tree is built with at most the specified number of levels.
- If the tree depth is **-1**, the fan-out value is used, and TotalView attempts to honor the fan-out value near the top (the root) of the tree, rather than near the bottom of the tree (the leaves).
- If the tree depth is **-2**, TotalView builds a tree similar to the one created when the tree depth is **-1**, except that the tree is unbalanced from side-to-side.

As an example, consider a tree with a root node and eight leaf nodes. If the fan-out value is **4** and the tree depth value is **0**, a tree that is "bushy" near the leaves is built because TotalView honors fan-out at the leaf end of the tree.

```
root:1 => n1:2 n5:2 ;
  n1:2 => n1:1 n2:1 n3:1 n4:1 ;
  n5:2 => n5:1 n6:1 n7:1 n8:1 ;
```

However, for the same tree when the tree depth setting is **-1**, a tree that is "bushy" near the root is built because TotalView honors fan-out at the root end of the tree.

```
root:1 => n1:2 n3:2 n5:2 n7:2 ;
  n1:2 => n1:1 n2:1 ;
  n3:2 => n3:1 n4:1 ;
  n5:2 => n5:1 n6:1 ;
  n7:2 => n7:1 n8:1 ;
```

Allocate an Extra Root Node

Option: –mrnet_extra_root *boolean*

State variable: TV::mrnet_extra_root *boolean*

Default value: `false`

For example, for a tree with a root and eight leaf nodes, using a fan-out value of **4**, a tree depth value of **0**, and requesting an extra root node, the following topology string will be calculated:

```
root:3 => root:1 ;
  root:1 => n1:2 n5:2 ;
    n1:2 => n1:1 n2:1 n3:1 n4:1 ;
    n5:2 => n5:1 n6:1 n7:1 n8:1 ;
```

Create a "Super Bushy" Tree

Option: -mrnet_super_bushy

State variable: TV::mrnet_super_bushy

Default value: `false`

Set this option to **true** if you are debugging an MPI job in which more than one CUDA process is running on a node. This option addresses the CUDA debug API limitation that allows a debugger process (such as the **tvdsvr**) to debug at most one target process using a GPU.

Path to MRNet Components

Option: -mrnet_commnode_path *path-to-mrnet_commnode*

State variable: TV::mrnet_commnode_path *path-to-mrnet_commnode*

Default value: *tv-installation-root*`/`*platform*`/bin/mrnet_commnode`

In a TotalView distribution, this is a path to a shell script that sets environment variables and execs the proper executable for the platform.

Path to the MRNet shared library directory

Option: -mrnet_filterlib_dir *path-to-mrnet-shlib-directory*

State variable: TV::mrnet_filterlib_dir *path-to-mrnet-shlib-directory*

Default value: *tv-installation-root*`/`*platform*`/shlib/mrnet/obj`

The TotalView server tree filters library `libservertree_filters.so.1` and the MRNet `libxplat.so` and `libmrnet.so` libraries are stored in this directory.

Performance Notes

Rogue Wave has conducted performance tests on some specific systems, and based on this testing we here provide a couple of tips. These tips should be considered as guidelines. The only way to know how performance is affected by different tree configurations on your system is by trying out alternatives with your own jobs.

- In general, higher fan-outs seem to perform better than deeper trees. Specifically, trees deeper than two levels consistently performed worse than a two-level tree.
- In our testing, a one-level tree failed due to resource shortages at around 512 nodes, so this is not a viable option at higher scales.

MRNet and ssh/rsh

Controlling MRNet's Use of rsh vs ssh

When MRNet is used as the infrastructure in a Linux cluster, MRNet's built-in support is used to instantiate the tree of debugger servers and communications processes. Tree instantiation is based on a remote shell startup mechanism. By default, MRNet uses `ssh` as the remote shell program, but some environments require that `rsh` be used instead. TotalView controls the remote shell used by MRNet using the `TV::xplat_rsh` state variable or the `-xplat_rsh` TotalView command option to set this state variable. If this variable isn't explicitly set and the `XPLAT_RSH` environment variable is not set or is empty, TotalView uses the value of **TV::launch_command** when instantiating an MRNet tree.

On Cray XT, XE, and XK systems, MRNet uses the ALPS Tool Helper library to instantiate the tree, which does not require the use of a separate remote shell program.

Tips on Using ssh/rsh with MRNet

The use of `rsh` / `ssh` differs in every system environment, therefore you should consult your system's documentation to know whether `rsh` or `ssh` should be used for your system. The `rsh` and `ssh` man pages are also a useful resource. Regardless, we offer the following tips as a guideline for how to configure `rsh` and `ssh`:

- Configure `rsh` or `ssh` to allow accessing the remote nodes without a password. `rsh` typically uses a file named `$HOME/.rhosts` (see `man 5 rhost` on a Linux system). `ssh` typically uses a pair of private/public keys stored in files under your `$HOME/.ssh` directory (see `man 1 ssh` on a Linux system).
- Disable X11 forwarding in `ssh` in your `$HOME/.ssh/config` file (see `man 5 ssh_config` on a Linux system).
- Set `StrictHostKeyChecking` to `no` in `ssh` in your `$HOME/.ssh/config` file (see `man 5 ssh_config` on a Linux system). If the `ssh` host keys change for a remote host, you may need to delete the lines for the host from the `$HOME/.ssh/known_hosts` file, or remove the file.

Using MRNet on Cray Computers

The following sections describe the options and state variables that control the configuration and use of MRNet on Cray. Please refer to the *TotalView Reference Guide* for a general description of how options and state variables can be used with TotalView.

For more information on Cray, see Debugging Cray XT/XE/XK/XC Applications on page 554.

Is Cray XT Flag

State variable: TV::is_cray_xt *boolean*

Default value: Set to `true` if TotalView is running on Linux-x86_64 or Linux-ARM64 (aarch64) and `/proc/cray_xt/nid` exists; otherwise, set to `false`.

Note that some Cray front-end (elogin) nodes do not have a `/proc/cray_xt/nid` file, in which case a job must be submitted to start TotalView on a Cray XT/XE/XK/XC node, or **tvconnect** must be used in your batch job. (For detail on **tvconnect**, see Reverse Connections on page 505.)

Is Cray CTI Flag

State variable: **TV::is_cray_cti** *boolean*

Default value: Set to `true` if TotalView is running on Linux-x86_64 or Linux-ARM64 (aarch64) and `/opt/cray/pe/cti/` exists; otherwise, set to `false`. TotalView uses the CTI (Cray Tools Interface) library to deploy debugger processes on the node where your application is running.

Cray XT MRNet Server Launch String

Option: –cray_xt_mrnet_server_launch_string *string*

State variable: TV::cray_xt_mrnet_server_launch_string *string*

Default value: `/var/spool/alps/%A/toolhelper%A/tvdsvr%K \`

`-working_directory %D -set_pw %P -verbosity %V %F`

Analogous to the standard MRNet server launch string, the Cray XT MRNet server launch string is used when MRNet launches the TotalView debugger servers on Cray when using the ATH (ALPS Tool Helper) library. TotalView expands the launch string using the normal launch string expansion rules.

Cray XT MRNet Transfer File List

Option: –cray_xt_mrnet_xfer_file_list *stringlist*

State variable: TV::cray_xt_mrnet_xfer_file_list *stringlist*

Default value:

The default value is calculated at TotalView startup time, as follows. The following is used as a "base" list of files needed by TotalView on the Cray compute nodes when MRNet and the Cray ATH libraries are in use.

```
TVROOT/bin/mrnet_commnode_main_cray_xt
TVROOT/bin/tvdsvr_mrnet
TVROOT/bin/tvdsvrmain_mrnet
TVROOT/shlib/mpa/obj_cray_xt/libmpattr.so.1
TVROOT/shlib/unwind/obj/libunwind-*.so.8
TVROOT/shlib/mrnet/obj_cray_xt/libmrnet.so
TVROOT/shlib/mrnet/obj_cray_xt/libxplat.so
TVROOT/shlib/mrnet/obj_cray_xt/libservertree_filters.so.1
TVROOT/shlib/mrnet/obj_cray_xt/libtvwrapalps.so.1
/lib64/libthread_db.so.1
```

Note that the name of the "libunwind-*.so.8" library depends on the platform, and will be either "libunwind-x86_64.so.8" for x86_64 or "libunwind-aarch64.so.8" for ARM64.

On the x86_64 platform, TotalView also stages the libraries required to support ReplayEngine, which include:

```
/usr/bin/ld
/usr/bin/objcopy
TVROOT/lib/libundodb_debugger_x64.so
TVROOT/lib/undodb_a_x64.o
TVROOT/lib/undodb_infiniband_preload_x64.so
TVROOT/lib/undodb_a_x32.o
TVROOT/lib/undodb_infiniband_preload_x32.so
```

The above list is then passed to the shell script named "cray_sysdso_deps.sh" to calculate the system shared libraries needed by the executables and shared libraries on the base list. The actual list of system libraries can vary from system to system, but typically consists of the following files:

```
/lib64/libgcc_s.so.1
/usr/lib64/libbfd-<version>.so
```

```
/usr/lib64/libstdc++.so.6
```

The version of `libbfd`, which is needed by `ld` and `objcopy`, varies from system to system.

The default value is a space-separated string-list of file names that are transferred (staged) to the compute nodes. These files are the shell script, executable and shared library files required to run the MRNet commnode and TotalView debugger server processes on the compute nodes. When instantiating the MRNet tree on Cray, the ALPS Tool Helper library is used to broadcast these files into the compute nodes' ramdisk under the `/var/spool/alps/apid` directory. *TVROOT* is the path to the platform-specific files in the TotalView installation.

Note that most up-to-date Cray systems support the debugger using the Cray Tools Interface (CTI) library, however TotalView attempts to support older legacy Cray systems that do not have CTI available by using the ALPS Tool Helper (ATH) library.

Cray CTI MRNet Transfer File List

Option: **–cray_cti_mrnet_xfer_file_list** *stringlist*

State variable: **TV::cray_cti_mrnet_xfer_file_list** *stringlist*

Default value:

The default value is calculated at TotalView startup time, as follows. The following is used the “base” list of files needed by TotalView on the Cray compute nodes when MRNet and the Cray CTI libraries are in use.

```
TVROOT/bin/mrnet_commnode_main_cray_cti
TVROOT/bin/tvdsvrmain_mrnet
TVROOT/shlib/mpa/obj_cray_xt/libmpattr.so.1
TVROOT/shlib/unwind/obj/libunwind-*.so.8
TVROOT/shlib/mrnet/obj_cray_cti/libmrnet.so
TVROOT/shlib/mrnet/obj_cray_cti/libxplat.so
TVROOT/shlib/mrnet/obj_cray_cti/libservertree_filters.so.1
TVROOT/shlib/mrnet/obj_cray_cti/libtvwrapcti.so.1
/lib64/libthread_db.so.1
```

Note that the name of the "libunwind-*.so.8" library depends on the platform, and will be either "libunwind-x86_64.so.8" for x86_64 or "libunwind-aarch64.so.8" for ARM64.

On the x86_64 platform, TotalView also stages the libraries required to support ReplayEngine, which include:

```
/usr/bin/ld
/usr/bin/objcopy
TVROOT/lib/libundodb_debugger_x64.so
TVROOT/lib/undodb_a_x64.o
TVROOT/lib/undodb_infiniband_preload_x64.so
```

Note that CTI does not support staging 32-bit ELF files, therefore they are not included in the above list. Shared library dependencies are calculated by CTI itself, therefore CTI takes care of staging any additional required shared library dependencies.

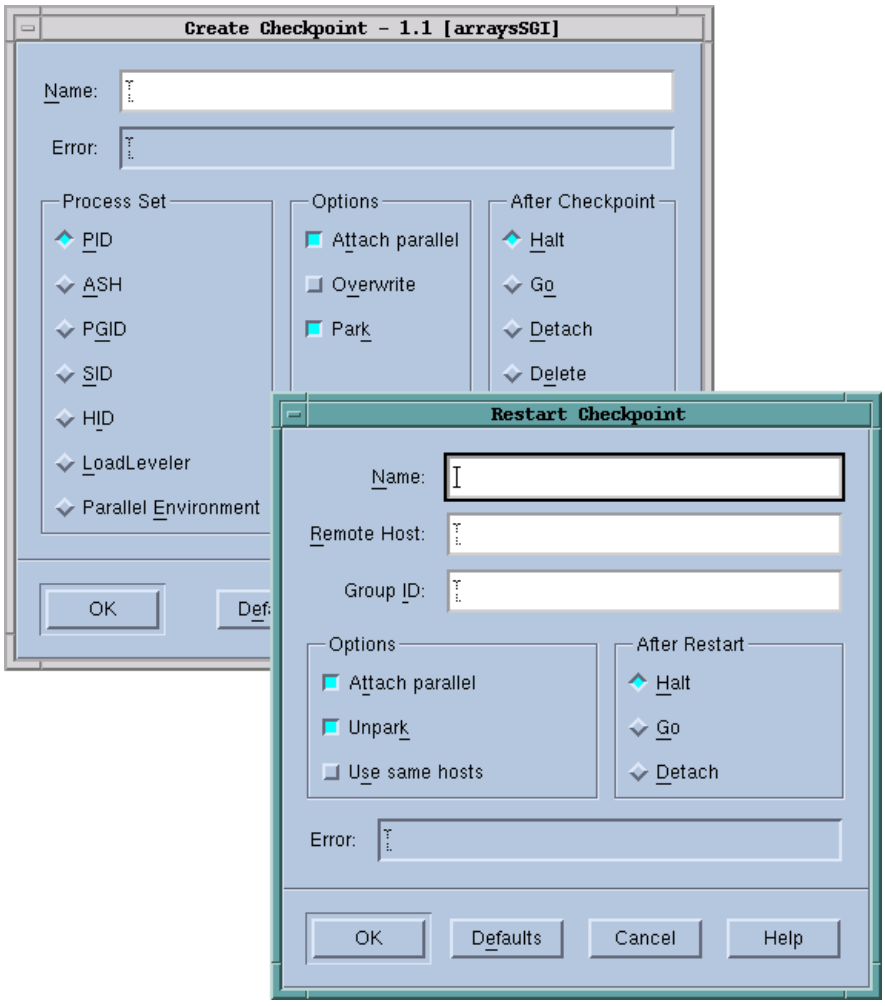
Checkpointing

You can save the state of selected processes and then use this saved information to restart the processes from the position where they were saved. For more information, see the Process Window **Tools > Create Checkpoint** and **Tools > Restart Checkpoint** commands in the online Help, Figure 260.

This feature is currently available only on IBM RS/6000 platforms.

CLI: **dcheckpoint**
drestart

Figure 260, Create Checkpoint and Restart Checkpoint Dialog Boxes



Fine-Tuning Shared Library Use

When TotalView encounters a reference to a shared library, it normally reads all of that library's symbols. In some cases, you might need to explicitly read in this library's information before TotalView automatically reads it.

On the other hand, you may not want TotalView to read and process a library's loader and debugging symbols. In most cases, reading these symbols occurs quickly. However, if your program uses large libraries, you can increase performance by telling TotalView not to read these symbols.

Preloading Shared Libraries

As your program executes, it can call the **dlopen()** function to access code contained in shared libraries. In some cases, you might need to do something from within TotalView that requires you to preload library information. For example, you might need to refer to one of a library's functions in an eval point or in a **Tools > Evaluate** command. If you use the function's name before TotalView reads the dynamic library, TotalView displays an error message.

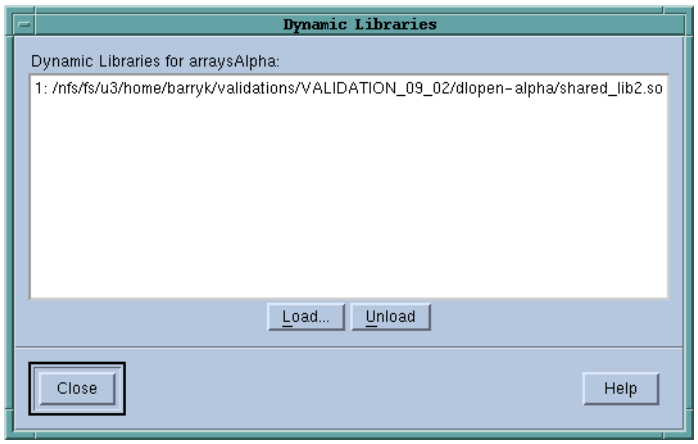
Use the **Tools > Debugger Loaded Libraries** command to tell the debugger to open a library.

CLI: ddlopen

This CLI command gives you additional ways to control how a library's symbols are used.

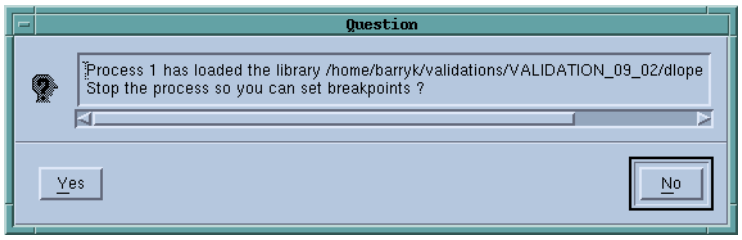
After selecting this command, TotalView displays the following dialog box:

Figure 261, Tools > Debugger Loaded Libraries Dialog Box



Selecting the **Load** button tells TotalView to display a file explorer dialog box that lets you navigate through your computer's file system to locate the library. After selecting a library, TotalView reads it and displays a question box that lets you stop execution to set a breakpoint:

Figure 262, Stopping to Set a Breakpoint Question Box



NOTE: TotalView might not read in symbol and debugging information when you use this command. See Controlling Which Symbols TotalView Reads on page 623 for more information.

RELATED TOPICS

TV:dll	**TV::dll** in the *Classic TotalView Reference Guide*
The **ddlopen** command	**ddlopen** in the *Classic TotalView Reference Guide*

Controlling Which Symbols TotalView Reads

When debugging large programs with large libraries, reading and parsing symbols can impact performance. This section describes how you can minimize the impact that reading this information has on your debugging session.

NOTE: Using the preference settings and variables described in this section, you can control the time it takes to read in the symbol table. For most programs, even large ones, changing the settings is often inconsequential, but if you are debugging a very large program with large libraries, you can achieve significant performance improvements.

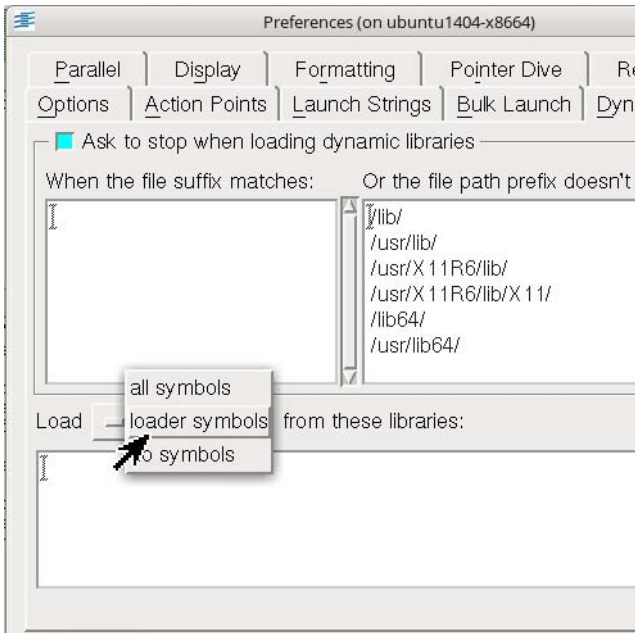
A shared library contains, among other things, loader and debugging symbols. Typically, loader symbols are read quite quickly. Debugging symbols can require considerable processing. The default behavior is to read all symbols. You can change this behavior by telling TotalView to only read in loader symbols or even that it should not read in any symbols.

NOTE: Saying "TotalView reads all symbols" isn't quite true as TotalView often just reads in loader symbols for some libraries. For example, it only reads in loader symbols if the library resides in the **/usr/lib** directory. (These libraries are typically those provided with the operating system.) You can override this behavior by adding a library name to the **All Symbols** list that is described in the next section.

Specifying Which Libraries are Read

After invoking the **File > Preferences** command, select the Dynamic Libraries Page.

Figure 263, 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. Placing entries into these areas does the following:

all symbols	This is the default operation. You only need to enter a library name here if it would be excluded by a wildcard in the **loader symbols** and **no symbols** areas.
loader symbols	TotalView reads loader symbols from these libraries. If your program uses a number of large shared libraries that you will not be debugging, you might set this to asterisk (*). You then enter the names of DLLs that you need to debug in the **all symbols** area.
no symbols	Normally, you wouldn't put anything on this list since TotalView might not be able to create a backtrace through a library if it doesn't have these symbols. However, you can increase performance if you place the names of your largest libraries here.

When reading a library, TotalView looks at these lists in the following order:

1. **all symbols**
2. **loader symbols**

3. **no symbols**

If a library is found in more than one area, TotalView does the first thing it is told to do and ignores any other requests. For example, after TotalView reads a library's symbols, it cannot honor a request to not load in symbols, so it ignores a request to not read them.

CLI: dset TV::dll_read_all_symbols
dset TV::dll_read_loader_symbols_only
dset TV::dll_read_no_symbols

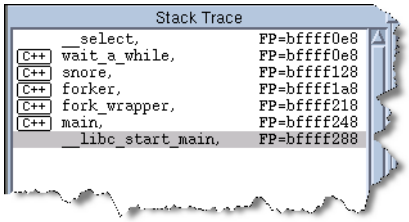
See the online Help for additional information.

If your program stops in a library that has not already had its symbols read, TotalView reads the library's symbols. For example, if your program SEGVs in a library, TotalView reads the symbols from that library before it reports the error. In all cases, however, TotalView always reads the loader symbols for shared system libraries.

Reading Excluded Information

While you are debugging your program, you might find that you do need the symbol information that you told TotalView it shouldn't read. Tell TotalView to read them by right-clicking your mouse in the Stack Trace Pane and then selecting the **Load All Symbols in Stack** command from the context menu.

Figure 264, Load All Symbols in Stack Context menu



After selecting this command, TotalView examines all active stack frames and, if it finds unread libraries in any frame, reads them.

CLI: TV::read_symbols
This CLI command also gives you finer control over how TotalView reads in library information.

PART V Using the CUDA Debugger

This part introduces the TotalView CUDA debugger and includes the following chapters:

- About the TotalView CUDA Debugger
  Introduces the CUDA debugger, including features, requirements, installation and drivers.
- CUDA Debugging Model and Unified Display
  Explores setting and viewing action points in CUDA code.
- CUDA Debugging Tutorial
  Discusses how to build and debug a simple CUDA program, including compiling, controlling execution, and analyzing data.
- CUDA Problems and Limitations
  Issues related to limitations in the NVIDIA environment.
- Sample CUDA Program
  Compilable sample CUDA program.

About the TotalView CUDA Debugger

The TotalView CUDA debugger is an integrated debugging tool capable of simultaneously debugging CUDA code that is running on the host system and the NVIDIA® GPU. CUDA support is an extension to the standard version TotalView, and is capable of debugging 64-bit CUDA programs. Debugging 32-bit CUDA programs is currently not supported.

Supported major features:

- Debug CUDA application running directly on GPU hardware
- Set breakpoints, pause execution, and single step in GPU code
- View GPU variables in PTX registers, local, parameter, global, or shared memory
- Access runtime variables, such as threadIdx, blockIdx, blockDim, etc.
- Debug multiple GPU devices per process
- Support for the CUDA MemoryChecker
- Debug remote, distributed and clustered systems
- Support for directive-based programming languages
- Support for host debugging features

Requirements:

The CUDA SDK and a host distribution supported by NVIDIA. For SDK versions and supported NVIDIA GPUs, please see the *TotalView Supported Platforms* guide.

Installing the CUDA SDK Tool Chain

Before you can debug a CUDA program, you must download and install the CUDA SDK software from NVIDIA using the following steps:

- Visit the NVIDIA CUDA Zone download page:

  https://developer.nvidia.com/cuda-downloads
- Select Linux as your operating system
- Download and install the CUDA SDK Toolkit for your Linux distribution (64-bit)

By default, the CUDA SDK Toolkit is installed under `/usr/local/cuda/`. The nvcc compiler driver is installed in `/usr/local/cuda/bin`, and the CUDA 64-bit runtime libraries are installed in `/usr/local/cuda/lib64`.

You may wish to:

- Add `/usr/local/cuda/bin` to your `PATH` environment variable.
- Add `/usr/local/cuda/lib64` to your `LD_LIBRARY_PATH` environment variable.

Directive-Based Accelerator Programming Languages

Converting C or Fortran code into CUDA code can take some time and effort. To simplify this process, a number of directive-based accelerator programming languages have emerged. These languages work by placing compiler directives in the user's code. Instead of writing CUDA code, the user can write standard C or Fortran code, and the compiler converts it to CUDA at compile time.

TotalView currently supports Cray's OpenMP Accelerator Directives and Cray's OpenACC Directives. TotalView uses the normal CUDA Debugging Model when debugging programs that have been compiled using these directives.

CUDA Debugging Model and Unified Display

Debugging CUDA programs presents some challenges when it comes to setting action points. When the host process starts, the CUDA threads don't yet exist and so are not visible to the debugger for setting breakpoints. (This is also true of any libraries that are dynamically loaded using **dlopen** and against which the code was not originally linked.)

To address this issue, TotalView allows setting a breakpoint *on any line* in the Source View, whether or not it can identify executable code for that line. The breakpoint becomes either a *pending* breakpoint or a *sliding* breakpoint until the CUDA code is loaded at runtime.

The Source Pane provides a unified display that includes line number symbols and breakpoints that span the host executable, host shared libraries, and the CUDA ELF images loaded into the CUDA threads. This design allows you to easily set breakpoints and view line number information for the host and GPU code at the same time. This is made possible by the way CUDA threads are grouped, discussed in the section Unified Source Pane and Breakpoint Display on page 634The TotalView CUDA Debugging Model on page 631.

- Unified Source Pane and Breakpoint Display on page 634The TotalView CUDA Debugging Model on page 631
- Pending and Sliding Breakpoints on page 633

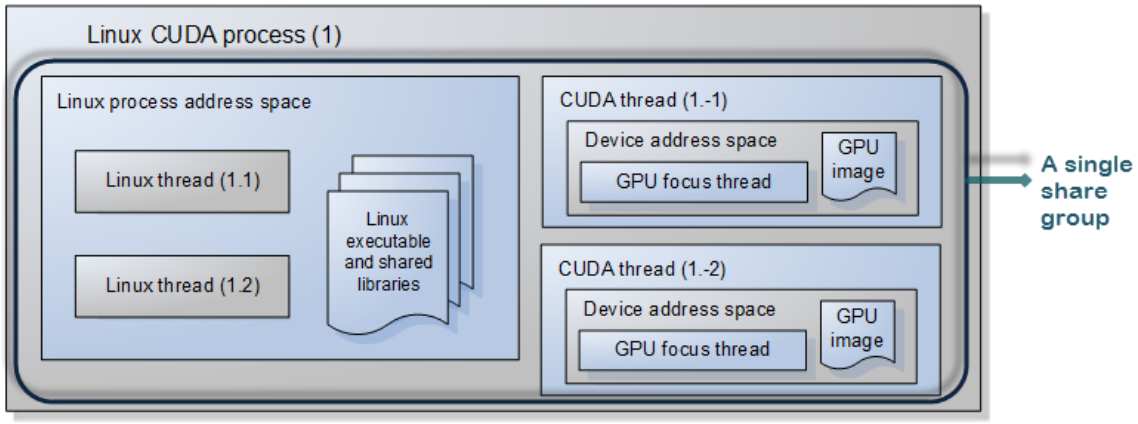
Unified Source Pane and Breakpoint Display on page 634The TotalView CUDA Debugging Model

The address space of the Linux CPU process and the address spaces of the CUDA threads are placed into the same share group. Breakpoints are created and evaluated within the share group, and apply to all of the image files (executable, shared libraries, and CUDA ELF images) in the share group.

That means that a breakpoint can apply to both the CPU and GPU code. This allows setting breakpoints on source lines in the host code that are then planted in the CUDA images at the same location once the CUDA kernel starts.

Consider a Linux process consisting of two Linux pthreads and two CUDA threads. (A CUDA thread is a CUDA context loaded onto a GPU device.) Figure 265 illustrates how TotalView would group the Linux and CUDA threads.

Figure 265, TotalView CUDA debugging model



The Linux host CUDA process

A Linux host CUDA process consists of:

- A Linux process address space, containing a Linux executable and a list of Linux shared libraries.
- A collection of Linux threads, where a Linux thread:
  - Is assigned a positive debugger thread ID.
  - Shares the Linux process address space with other Linux threads.
- A collection of CUDA threads, where a CUDA thread:

- Is assigned a negative debugger thread ID.
- Has its own address space, separate from the Linux process address space, and separate from the address spaces of other CUDA threads.
- Has a "GPU focus thread", which is focused on a specific hardware thread (also known as a core or "lane" in CUDA lingo).

The above TotalView CUDA debugging model is reflected in the TotalView user interface and command line interface. In addition, CUDA-specific CLI commands allow you to inspect CUDA threads, change the focus, and display their status. See the **dcuda** entry in the *Classic TotalView Reference Guide* for more information.

Pending and Sliding Breakpoints

Because CUDA threads and the host process are all in the same share group, you can create pending or sliding breakpoints on source lines and functions in the GPU code before the code is loaded onto the GPU. If TotalView can't locate code associated with a particular line in the source view, you can still plant a breakpoint there, if you know that there will be code there once the CUDA kernel loads.

Pending and sliding breakpoints are not specific to CUDA and are discussed in more detail in Setting Action Points on page 188.

RELATED TOPICS	
Sliding breakpoints	Sliding Breakpoints on page 198
Pending breakpoints	Pending Breakpoints on page 201
Pending eval points	Creating a Pending Eval Point on page 223
How the unified Source Pane displays breakpoints in dynamically-loaded code	Unified Source Pane and Breakpoint Display on page 634
Using **dactions** to display pending and mixed breakpoint detail before and after CUDA code has loaded.	"Examples of Actions Points in Both Host and Dynamically Loaded Code" in the **dactions** entry in the *TotalView Reference Guide*

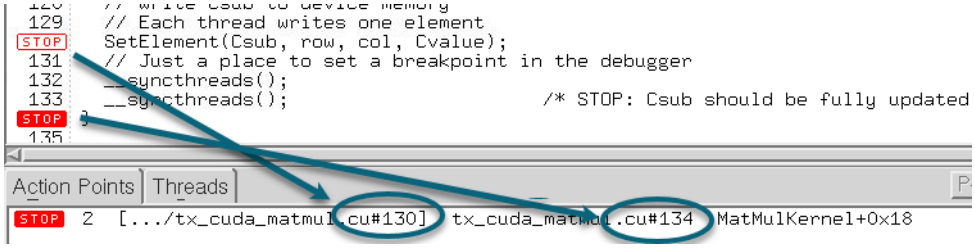
Unified Source Pane and Breakpoint Display

Because CUDA threads are in the same share group as are their host Linux processes, the Source Pane can visibly display a unified view of lines and breakpoints set in both the host code and the CUDA code. TotalView determines the equivalence of host and CUDA source files by comparing the base name and directory path of each file in the share group; if they are equal, the line number information is unified in the Source Pane.

NOTE: A unified display is not specific to CUDA but is particularly suited to debugging CUDA programs. It is discussed in more detail in The Source Pane on page 161.

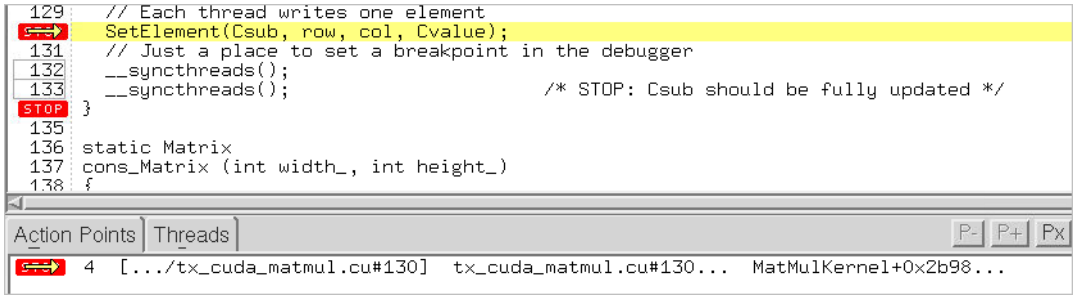
This is particularly visible when breakpoints are set. For example, Figure 266 shows source code before the CUDA thread has launched. A breakpoint has been set at line 130 which slid to line 134 in the host code.

Figure 266, Source Pane before CUDA kernel launch



After CUDA kernel launch, Figure 267 shows that TotalView has read the line number information for the CUDA image and the slid breakpoint now displays according to the full breakpoint expression in the Action Points tab.

Figure 267, Source Pane after CUDA kernel launch



Notice also that the source-line breakpoint boxes for the CUDA code have been unified with the CPU code. For example, lines 132 and 133 appeared with no breakpoint boxes before runtime, but after the CUDA threads have launched, TotalView is able to identify line symbol information there, so they now appear with gray boxes.

RELATED TOPICS	
More on the unified Source Pane display	Unified Source Pane Display on page 161
The CUDA share group model	Unified Source Pane and Breakpoint Display on page 634The TotalView CUDA Debugging Model on page 631
Using **dactions** to display pending and mixed breakpoint detail before and after CUDA code has loaded.	"Examples of Actions Points in Both Host and Dynamically Loaded Code" in the **dactions** entry in the *TotalView Reference Guide*

CUDA Debugging Tutorial

This chapter discusses how to build and debug a simple CUDA program using TotalView.

Compiling for Debugging

When compiling an NVIDIA CUDA program for debugging, it is necessary to pass the **-g -G** options to the **nvcc** compiler driver. These options disable most compiler optimization and include symbolic debugging information in the driver executable file, making it possible to debug the application. For example, to compile the sample CUDA program named **tx_cuda_matmul.cu** for debugging, use the following commands to compile and execute the application:

```
% /usr/local/bin/nvcc -g -G -c tx_cuda_matmul.cu -o tx_cuda_matmul.o
% /usr/local/bin/nvcc -g -G -Xlinker=-R/usr/local/cuda/lib64 \
  tx_cuda_matmul.o -o tx_cuda_matmul
% ./tx_cuda_matmul
A:
[    0][    0] 0.000000
...output deleted for brevity...
[    1][    1] 131.000000
%
```

Access the source code for this CUDA program tx_cuda_matmul.cu program at Sample CUDA Program on page 663.

Compiling for Fermi

To compile for Fermi, use the following compiler option:

```
-gencode arch=compute_20,code=sm_20
```

Compiling for Fermi and Tesla

To compile for both Fermi and Tesla GPUs, use the following compiler options:

```
-gencode arch=compute_20,code=sm_20 -gencode arch=compute_10,code=sm_10
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Kepler

To compile for Kepler GPUs, use the following compiler options:

```
-gencode arch=compute_35,code=sm_35
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Pascal

To compile for Pascal GPUs, use the following compiler options:

```
-gencode arch=compute_60,code=sm_60
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Compiling for Volta

To compile for Volta GPUs, use the following compiler options:

```
-gencode arch=compute_70,code=sm_70
```

See the NVIDIA documentation for complete instructions on compiling your CUDA code.

Starting a TotalView CUDA Session

A standard TotalView installation supports debugging CUDA applications running on both the host and GPU processors. TotalView dynamically detects a CUDA install on your system. To start the TotalView GUI or CLI, provide the name of your CUDA host executable to the **totalview** or **totalviewcli** command. For example, to start the TotalView GUI on the sample program, use the following command:

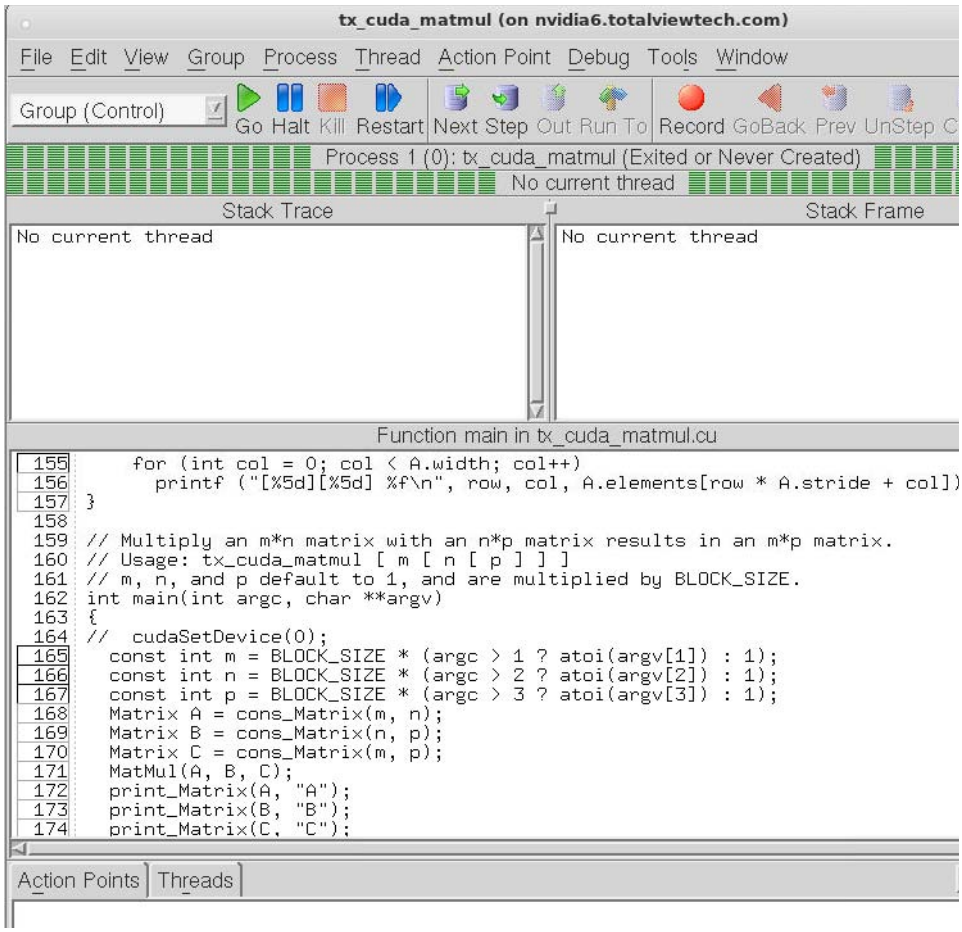
```
% totalview tx_cuda_matmul
```

If TotalView successfully loads the CUDA debugging library, it prints the current CUDA debugger API version and the NVIDIA driver version:

```
CUDA library loaded: Current DLL API version is “8.0.128”; NVIDIA driver version
384.125
...
```

After reading the symbol table information for the CUDA host executable, TotalView opens the initial process window focused on main in the host code, as shown in Figure 268.

Figure 268, Initial process window opened on CUDA host code



You can debug the CUDA host code using the normal TotalView commands and procedures.

Controlling Execution

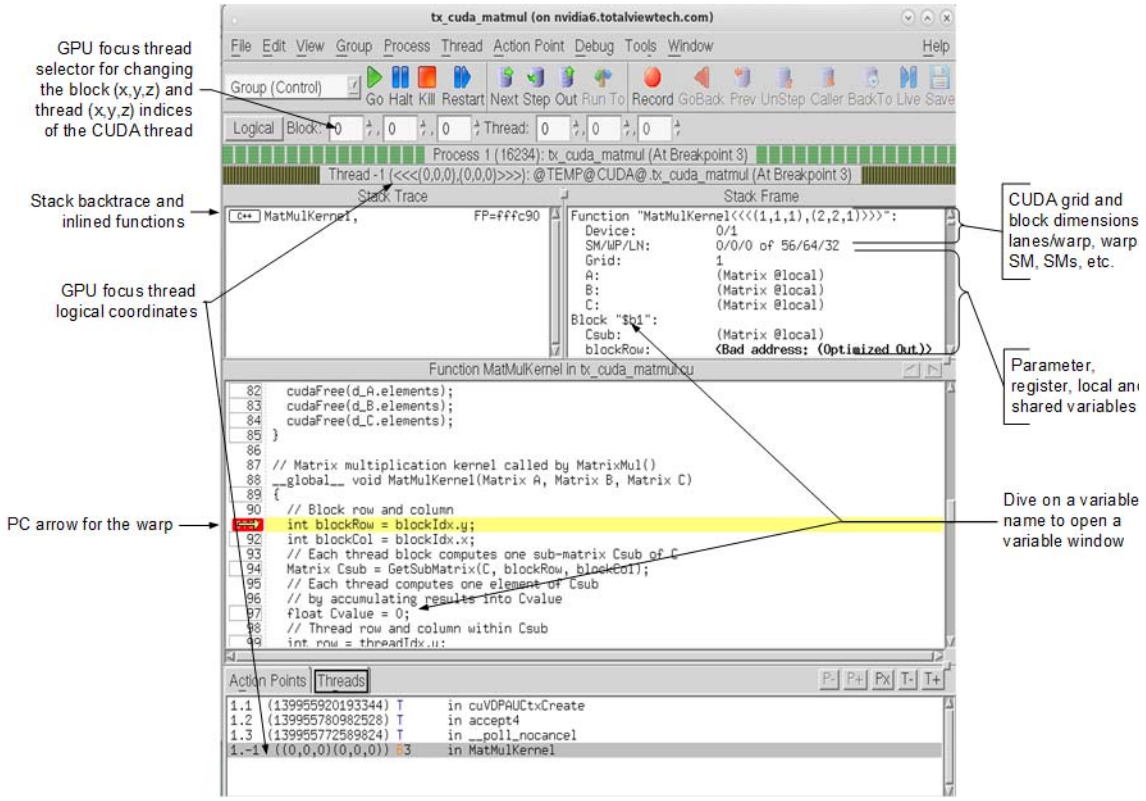
NOTE: **Set breakpoints in CUDA code before you start the process.** If you start the process without setting any breakpoints, there are no prompts to set them afterward.

Note that breakpoints set in CUDA code will *slide* to the next host (CPU) line in the source file, but once the program is running and the CUDA code is loaded, TotalView recalculates the breakpoint expression and plants a breakpoint at the proper location in the CUDA code. (See Sliding Breakpoints on page 198.)

Viewing GPU Threads

Once the CUDA kernel starts executing, it will hit the breakpoint planted in the GPU code, as shown in Figure 269.

Figure 269, CUDA thread stopped at a breakpoint, focused on GPU thread <<<(0,0,0),(0,0,0)>>>



The logical coordinates of the GPU focus threads are shown in the thread status title bar and the Threads pane. You can use the GPU focus thread selector to change the GPU focus thread. When you change the GPU focus thread, the logical coordinates displayed also change, and the stack trace, stack frame, and source panes are updated to reflect the state of the new GPU focus thread.

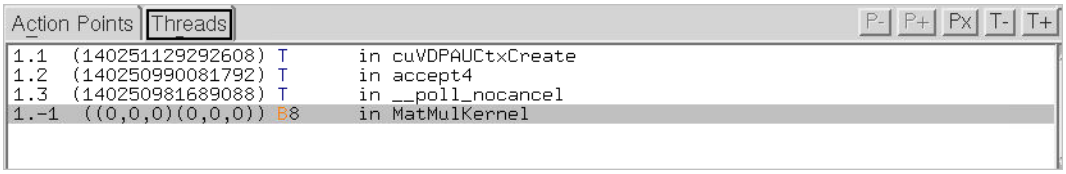
The yellow PC arrow in the source pane shows the execution location of the GPU focus thread. The GPU hardware threads, also known as "lanes," execute in parallel so multiple lanes may have the same PC value. The lanes may be part of the same warp (up to 32 maximum threads that are scheduled concurrently), or in different warps.

The stack trace pane shows the stack backtrace and inlined functions. Each stack frame in the stack backtrace represents either the PC location of GPU kernel code, or the expansion of an inlined function. Inlined functions can be nested. The "return PC" of an inlined function is the address of the first instruction following the inline expansion, which is normally within the function containing the inlined-function expansion.

The stack frame pane shows the parameter, register and local variables for the function in the selected stack frame. The variables for the selected GPU kernel code or inlined function expansion are shown.

CUDA Thread IDs and Coordinate Spaces

TotalView gives host threads a positive debugger thread ID and CUDA threads a negative thread ID. In this example, the initial host thread in process "1" is labeled "1.1" and the CUDA thread is labeled "1.-1".

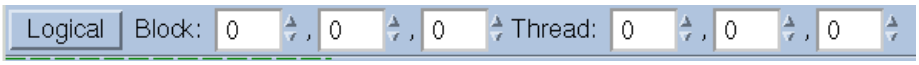


In TotalView, a "CUDA thread" is a CUDA kernel invocation consisting of registers and memory, as well as a "GPU focus thread". Use the "GPU focus selector" to change the physical coordinates of the GPU focus thread.

There are two coordinate spaces. One is the logical coordinate space that is in CUDA terms grid and block indices: <<<(Bx,By,Bz),(Tx,Ty,Tz)>>>. The other is the physical coordinate space that is in hardware terms the device number, streaming multiprocessor (SM) number on the device, warp (WP) number on the SM, and lane (LN) number on the warp.

Any given thread has both a thread index in this 4D physical coordinate space, and a different thread index in the 6D logical coordinate space. These indices are shown in a series of spin boxes in the process window. If the button says “Physical,” the physical thread number is displayed; if “Logical” (Figure 269), the logical number. Pressing this button switches between the two numbering systems, but does not change the actual thread.

Figure 270, Logical / physical toggle in the process window



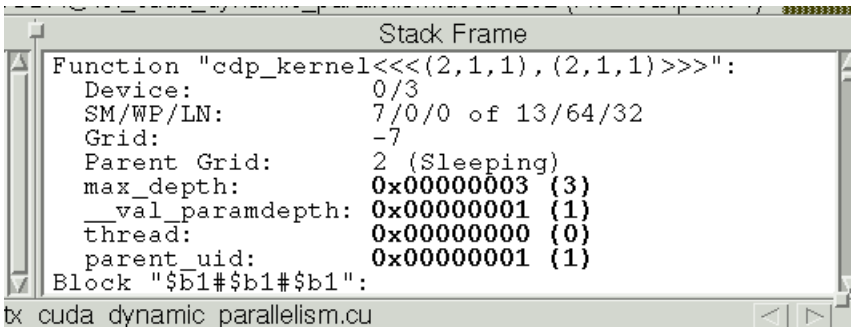
To view a CUDA host thread, select a thread with a positive thread ID in the Threads tab of the process window. To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU thread selector to focus on a specific GPU thread. There is one GPU focus thread per CUDA thread, and changing the GPU focus thread affects all windows displaying information for a CUDA thread and all command line interface commands targeting a CUDA thread. In other words, changing the GPU focus thread can change data displayed for a CUDA thread and affect other commands, such as single-stepping.

Note that in all cases, when you select a thread, TotalView automatically switches the stack trace, stack frame and source panes, and Action Points tab to match the selected thread.

Viewing the Kernel's Grid Identifier

TotalView supports showing the grid identification in the stack frame information when a CUDA thread stops, Figure 271.

Figure 271, Viewing the Grid and Parent Grid Identifiers



The grid is a unique identifier for a kernel running on a device. CUDA supports kernels launching parallel kernels on the same device. The parent grid is the identifier of the grid that launched the kernel currently in focus.

Single-Stepping GPU Code

TotalView allows you to single-step GPU code just like normal host code, but note that a single-step operation steps the entire warp associated with the GPU focus thread. So, when focused on a CUDA thread, a single-step operation advances all of the GPU hardware threads in the same warp as the GPU focus thread.

To advance the execution of more than one warp, you may either:

- set a breakpoint and continue the process
- select a line number in the source pane and select "Run To".

Execution of more than one warp also happens when single-stepping a **__syncthreads()** thread barrier call. Any source-level single-stepping operation runs all of the GPU hardware threads to the location following the thread barrier call.

Single-stepping an inlined function (nested or not) in GPU code behaves the same as single-stepping a non-inlined function. You can:

- step into an inlined function,
- step over an inlined function,
- run to a location inside an inlined function,
- single-step within an inlined function, and
- return out of an inlined function.

Halting a Running Application

You can temporarily halt a running application at any time by selecting "Halt", which halts the host and CUDA threads. This can be useful if you suspect the kernel might be hung or stuck in an infinite loop. You can resume execution at any time by selecting "Go" or by selecting one of the single-stepping buttons.

Displaying CUDA Program Elements

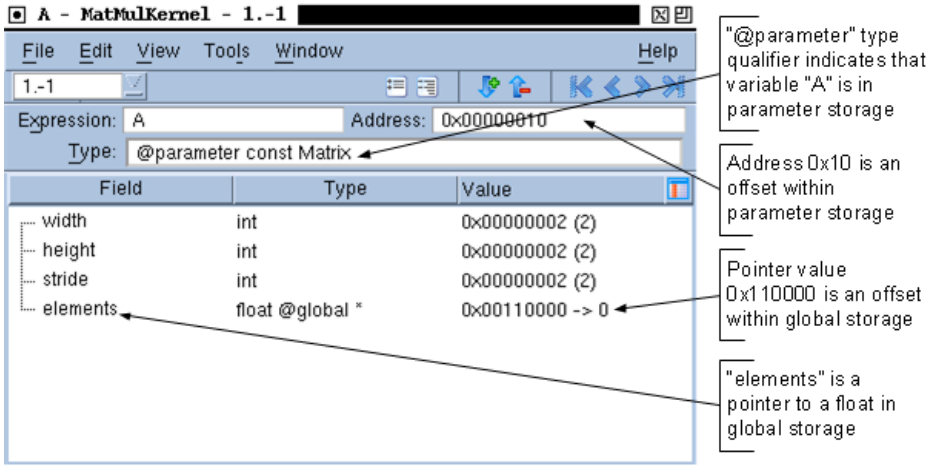
GPU Assembler Display

Due to limitations imposed by NVIDIA, assembler display is not supported. All GPU instructions are currently displayed as 32-bit hexadecimal words.

GPU Variable and Data Display

TotalView can display variables and data from a CUDA thread. The stack frame pane of the process window contains parameter, register, local, and shared variables, as shown in Figure 272. The variables are contained within the lexical blocks in which they are defined. The type of the variable determines its storage kind (register, or local, shared, constant or global memory). The address is a PTX register name or an offset within the storage kind.

Figure 272, A variable window displaying a parameter



Dive on a variable in the stack frame pane or source pane in the process window to open a variable window. Figure 272 shows a parameter named `A` with type `@parameter const Matrix`.

The identifier `@parameter` is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is parameter storage. The debugger uses the storage qualifier to determine how to locate `A` in device memory. The supported type storage qualifiers are shown in Table 3.

Table 3: Supported Type Storage Qualifiers

Storage Qualifier	Meaning
`@code`	An offset within executable code storage
`@constant`	An offset within constant storage
`@generic`	An offset within generic storage
`@frame`	An offset within frame storage
`@global`	An offset within global storage
`@local`	An offset within local storage
`@parameter`	An offset within parameter storage
`@iparam`	Input parameter
`@oparam`	Output parameter
`@shared`	An offset within shared storage
`@surface`	An offset within surface storage
`@texsampler`	An offset within texture sampler storage
`@texture`	An offset within texture storage
`@rtvar`	Built-in runtime variables (see CUDA Built-In Runtime Variables)
`@register`	A PTX register name (see PTX Registers)
`@sregister`	A PTX special register name (see PTX Registers)
@managed_global	Statically allocated managed variable. See Managed Memory Variables.

The type storage qualifier is a necessary part of the type for correct addressing in the debugger. When you edit a type or a type cast, make sure that you specify the correct type storage qualifier for the address offset.

Managed Memory Variables

About Managed Memory

The CUDA Unified Memory component defines a managed memory space that allows all GPUs and hosts to "see a single coherent memory image with a common address space," as described in the NVIDIA documentation "Unified Memory Programming."

Allocating a variable in managed memory avoids explicit memory transfers between host and GPUs, as any allocation created in the managed memory space is automatically migrated between the host and GPU.

A managed memory variable is marked with a "**__managed__**" memory space specifier.

How TotalView Displays Managed Variables

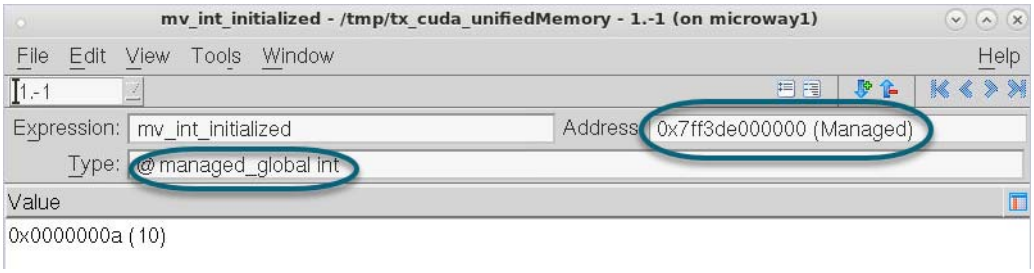
To make it easier to recognize and work with managed variables, TotalView annotates their address with the term "Managed", and, for statically allocated variables, adds the **@managed_global** type qualifier.

Statically Allocated Managed Variables

For example, consider this statically allocated managed variable, declared with the **__managed__** keyword:

```
__device__ __managed__ int mv_int_initialized=10;
```

TotalView adds "Managed" in the Address field and decorates the type with **@managed_global**:

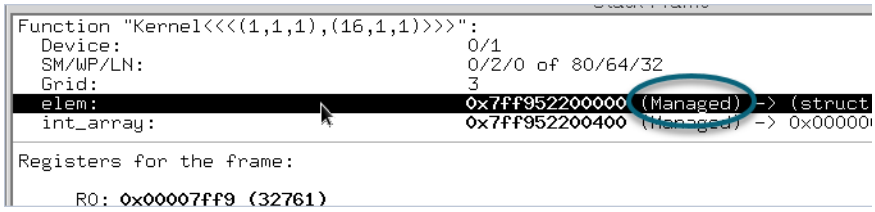


Dynamically Allocated Managed Variables

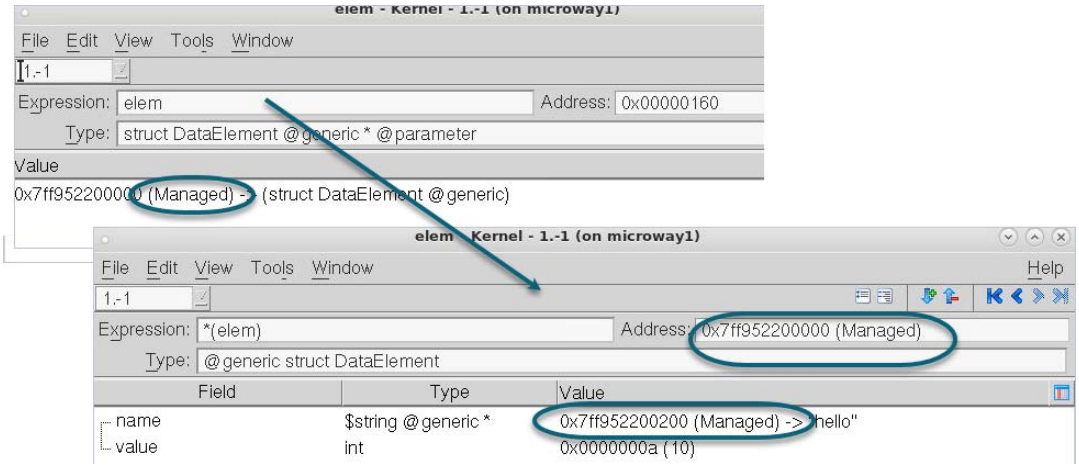
Managed memory can be dynamically allocated using the **cudaMallocManaged()** function, for example:

```
cudaMallocManaged((void**)&(elm->name), sizeof(char) * (strlen("hello") + 1) );
```

Here, the Stack Frame shows that the variable **elem** points into managed memory. That is, **elem** is a pointer and its value points into managed memory; the pointer's value is annotated with "(Managed)".



Diving on it shows that the pointer's value points into managed memory. Diving on the pointer itself annotates the Address value with "Managed". Note that one of its members, **name**, also points into managed memory.



CUDA Built-In Runtime Variables

TotalView allows access to the CUDA built-in runtime variables, which are handled by TotalView like any other variables, except that you cannot change their values.

The supported CUDA built-in runtime variables are as follows:

- `struct dim3_16 threadIdx;`
- `struct dim3_16 blockIdx;`
- `struct dim3_16 blockDim;`
- `struct dim3_16 gridDim;`
- `int warpSize;`

The types of the built-in variables are defined as follows:

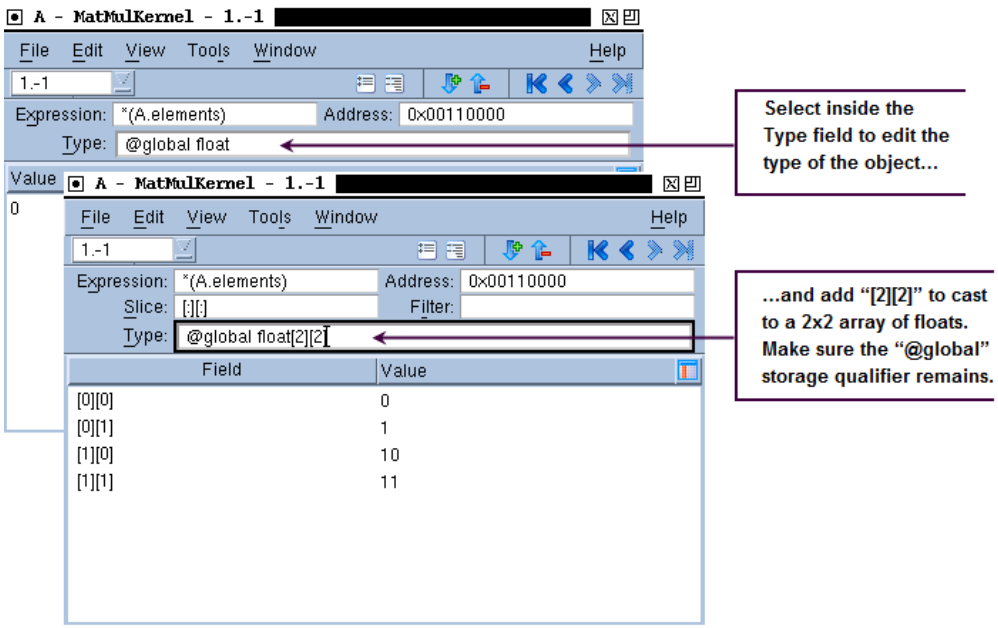
- `struct dim3_16 { unsigned short x, y, z; };`
- `struct dim2_16 { unsigned short x, y; };`

You can dive on the name of a runtime variable in the source pane of the process window, or input its name into the **View > Lookup Variable... (v)** menu command dialog box. Built-in variables can also be used in the TotalView expression system.

Type Casting

The variable window allows you to edit the types of variables. This is useful for viewing an address as a different type. For example, Figure 273 shows the result of casting a float in global storage to a 2x2 array of floats in global storage.

Figure 273, Casting to a 2x2 array of float in global storage



You can determine the storage kind of a variable by diving on the variable to open a variable window in the graphical user interface (GUI), or by using the **dwhat** command in the command line interface (CLI).

Here are some examples of using the CLI to determine variable types and to perform type casts. **Use Tools > Command Line** from the process window menu to open a CLI window from the GUI.

The following examples use the CLI for ease of illustration, but you can instead use the GUI by entering the cast expression (**dprint** *argument*) in the Expression field of the variable window.

When you are using the CLI and want to operate on a CUDA thread, you must first focus on the CUDA thread. The GPU focus thread in the CLI is the same as in the GUI:

```
d1.<> dfocus .-1
d1.-1
d1.-1>
```

The **dwhat** command prints the type and address offset or PTX register name of a variable. The **dwhat** command prints additional lines that have been omitted here for clarity:

```
d1.-1> dwhat A
```

```
In thread 1.-1:
Name: A; Type: @parameter const Matrix; Size: 24 bytes; Addr: 0x00000010
...
d1.-1> dwhat blockRow
In thread 1.-1:
Name: blockRow; Type: @register int; Size: 4 bytes; Addr: %r2
...
d1.-1> dwhat Csub
In thread 1.-1:
Name: Csub; Type: @local Matrix; Size: 24 bytes; Addr: 0x00000060
...
d1.-1>
```

You can use **dprint** in the CLI to cast and print an address offset as a particular type. Note that the CLI is a Tcl interpreter, so we wrap the expression argument to **dprint** in curly braces {} for Tcl to treat it as a literal string to pass into the debugger. For example, below we take the address of "A", which is at 0x10 in parameter storage. Then, we can cast 0x10 to a "pointer to a Matrix in parameter storage", as follows:

```
d1.-1> dprint {&A}
 &A = 0x00000010 -> (Matrix const @parameter)
d1.-1> dprint {*(@parameter Matrix*)0x10}
 *(@parameter Matrix*)0x10 = {
   width = 0x00000002 (2)
   height = 0x00000002 (2)
   stride = 0x00000002 (2)
   elements = 0x00110000 -> 0
 }
d1.-1>
```

The above "@parameter" type qualifier is an important part of the cast, because without it the debugger cannot determine the storage kind of the address offset. Casting without the proper type storage qualifier usually results in "Bad address" being displayed, as follows:

```
d1.-1> dprint {*(Matrix*)0x10}
 *(Matrix*)0x10 = <Bad address: 0x00000010> (struct Matrix)
d1.-1>
```

You can perform similar casts for global storage addresses. We know that "A.elements" is a pointer to a 2x2 array in global storage. The value of the pointer is 0x110000 in global storage. You can use C/C++ cast syntax:

```
d1.-1> dprint {A.elements}
 A.elements = 0x00110000 -> 0
d1.-1> dprint {*(@global float(*)[2][2])0x00110000}
 *(@global float(*)[2][2])0x00110000 = {
   [0][0] = 0
   [0][1] = 1
   [1][0] = 10
   [1][1] = 11
```

```
 }
d1.-1>
```

Or you can use TotalView cast syntax, which is an extension to C/C++ cast syntax that allows you to simply read the type from right to left to understand what it is:

```
d1.-1> dprint {*(@global float[2][2]*)0x00110000}
 *(@global float[2][2]*)0x00110000 = {
   [0][0] = 0
   [0][1] = 1
   [1][0] = 10
   [1][1] = 11
 }
d1.-1>
```

If you know the address of a pointer and you want to print out the target of the pointer, you must specify a storage qualifier on both the pointer itself and the target type of the pointer. For example, if we take the address of "A.elements", we see that it is at address offset 0x20 in parameter storage, and we know that the pointer points into global storage. Consider this example:

```
d1.-1> dprint {*(@global float[2][2]*@parameter*)0x20}
 *(@global float[2][2]*@parameter*)0x20 = 0x00110000 -> (@global float[2][2])
d1.-1> dprint {**(@global float[2][2]*@parameter*)0x20}
 **(@global float[2][2]*@parameter*)0x20 = {
   [0][0] = 0
   [0][1] = 1
   [1][0] = 10
   [1][1] = 11
 }
d1.-1>
```

Above, using the TotalView cast syntax and reading right to left, we cast 0x20 to a pointer in parameter storage to a pointer to a 2x2 array of floats in global storage. Dereferencing it once gives the value of the pointer to global storage. Dereferencing it twice gives the array in global storage. The following is the same as above, but this time in C/C++ cast syntax:

```
d1.-1> dprint {*(@global float(*@parameter*)[2][2])0x20}
 *(@global float(*@parameter*)[2][2])0x20 = 0x00110000 -> (@global float[2][2])
d1.-1> dprint {**(@global float(*@parameter*)[2][2])0x20}
 **(@global float(*@parameter*)[2][2])0x20 = {
   [0][0] = 0
   [0][1] = 1
   [1][0] = 10
   [1][1] = 11
 }
d1.-1>
```

PTX Registers

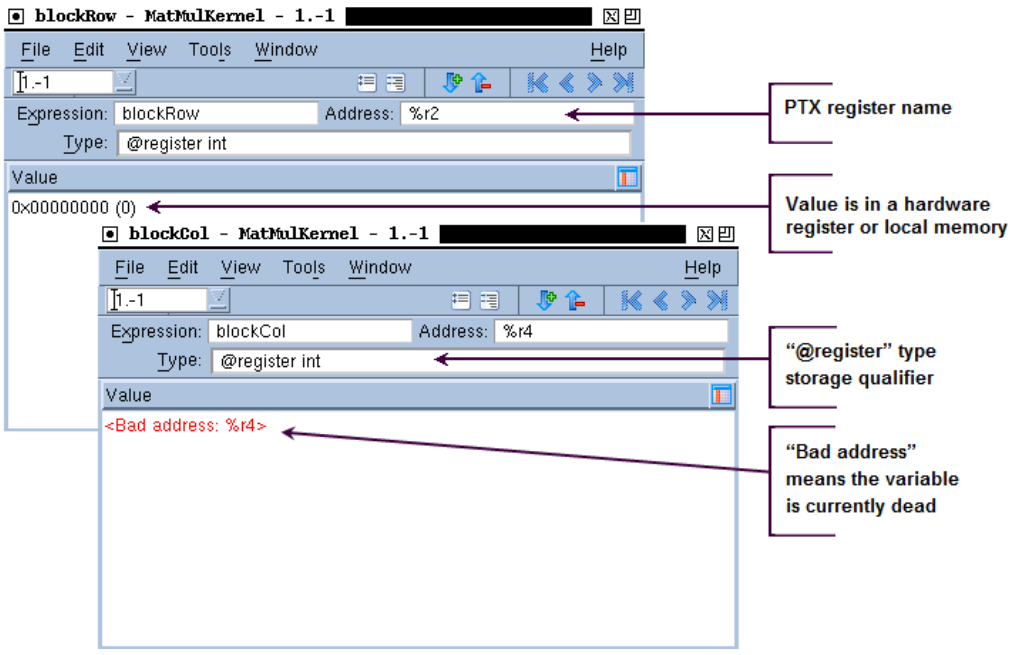
In CUDA, PTX registers are more like symbolic virtual locations than hardware registers in the classic sense. At any given point during the execution of CUDA device code, a variable that has been assigned to a PTX register may live in one of three places:

- A hardware (SAS) register
- Local storage
- Nowhere (its value is dead)

Variables that are assigned to PTX registers are qualified with the "@register" type storage qualifier, and their locations are PTX register names. The name of a PTX register can be anything, but the compiler usually assigns a name in one of the following formats: `%rN`, `%rdN`, or `%fN`, where `N` is a decimal number.

Using compiler-generated location information, TotalView maps a PTX register name to the SASS hardware register or local memory address where the PTX register is currently allocated. If the PTX register value is "live", then TotalView shows you the SASS hardware register name or local memory address. If the PTX register value is "dead", then TotalView displays `Bad address` and the PTX register name as show in Figure 274.

Figure 274, PTX register variables: one live, one dead



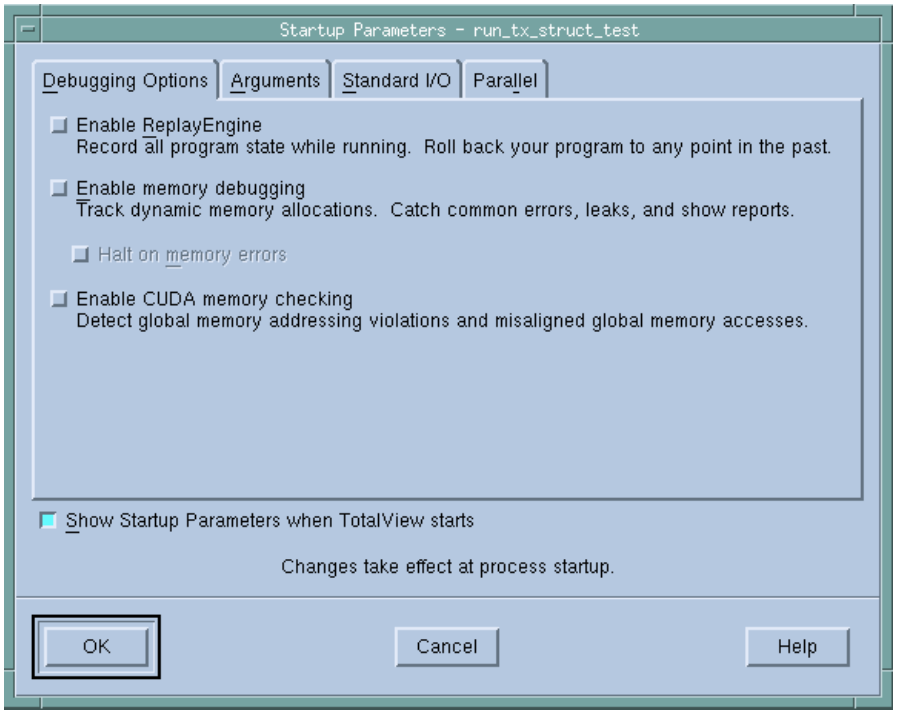
Enabling CUDA MemoryChecker Feature

You can detect global memory addressing violations and misaligned global memory accesses by enabling the CUDA MemoryChecker feature.

To enable the feature, use one of the following:

- Select "Enable CUDA memory checking" from the Startup Parameters dialog box in TotalView, as shown in Figure 275.

Figure 275, Enabling CUDA memory checking from TotalView Startup Parameters



- Pass the `-cuda_memcheck` option to the `totalview` command.
- Set the `TV::cuda_memcheck` CLI state variable to `true`. For example:
  ```
  dset TV::cuda_memcheck true
  ```

Note that global memory violations and misaligned global memory accesses will be detected only while the CUDA thread is running. Detection will not happen when single-stepping the CUDA thread.

GPU Core Dump Support

CUDA GPU core dumps can be debugged just as you debug any other core dump. To obtain a GPU core dump, you must first set the `CUDA_ENABLE_COREDUMP_ON_EXCEPTION` environment variable to `1` to enable generation of a GPU core dump when a GPU exception is encountered. This option is disabled by default.

To change the default core dump file name, set the `CUDA_COREDUMP_FILE` environment variable to a specific file name. The default core dump file name is in the following format: `core.cuda.`*<hostname>*`.`*<pid>* where *<hostname>* is the host name of machine running the CUDA application and *<pid>* is the process identifier of the CUDA application.

To debug a GPU core dump, TotalView must be running on a machine with the CUDA SDK installed.

As with any core dump, you must also supply the name of the executable that produced the core dump:

```
totalview <executable> <core-dump-file>
```

GPU Error Reporting

By default, TotalView reports GPU exception errors as "signals." Continuing the application after these errors can lead to application termination or unpredictable results.

Table 4 lists reported errors, according to these platforms and settings:

- Exception codes `Lane Illegal Address` and `Lane Misaligned Address` are detected using all supported SDK versions when CUDA memcheck is enabled, on supported Tesla and Fermi hardware.
- All other CUDA errors are detected only for GPUs with sm_20 or higher (for example Fermi) running SDK 3.1 or higher. It is not necessary to enable CUDA memcheck to detect these errors.

Table 4: CUDA Exception Codes

Exception code	Error Precision	Error Scope	Description
CUDA_EXCEPTION_0: `"Device Unknown Exception"`	Not precise	Global error on the GPU	An application-caused global GPU error that does not match any of the listed error codes below.
CUDA_EXCEPTION_1: `"Lane Illegal Address"`	Precise (Requires memcheck on)	Per lane/thread error	A thread has accessed an illegal (out of bounds) global address.
CUDA_EXCEPTION_2: `"Lane User Stack Overflow"`	Precise	Per lane/thread error	A thread has exceeded its stack memory limit.
CUDA_EXCEPTION_3: `"Device Hardware Stack Overflow"`	Not precise	Global error on the GPU	The application has triggered a global hardware stack overflow, usually caused by large amounts of divergence in the presence of function calls.
CUDA_EXCEPTION_4: `"Warp Illegal Instruction"`	Not precise	Warp error	A thread within a warp has executed an illegal instruction.
CUDA_EXCEPTION_5: `"Warp Out-of-range Address"`	Not precise	Warp error	A thread within a warp has accessed an address that is outside the valid range of local or shared memory regions.
CUDA_EXCEPTION_6: `"Warp Misaligned Address"`	Not precise	Warp error	A thread within a warp has accessed an incorrectly aligned address in the local or shared memory segments.

Table 4: CUDA Exception Codes

Exception code	Error Precision	Error Scope	Description
CUDA_EXCEPTION_7: `"Warp Invalid Address Space"`	Not precise	Warp error	A thread within a warp has executed an instruction that attempts to access a memory space not permitted for that instruction.
CUDA_EXCEPTION_8: `"Warp Invalid PC"`	Not precise	Warp error	A thread within a warp has advanced its PC beyond the 40-bit address space.
CUDA_EXCEPTION_9: `"Warp Hardware Stack Overflow"`	Not precise	Warp error	A thread within a warp has triggered a hardware stack overflow.
CUDA_EXCEPTION_10: `"Device Illegal Address"`	Not precise	Global error	A thread has accessed an illegal (out of bounds) global address. For increased precision, enable memcheck.
CUDA_EXCEPTION_11: `"Lane Misaligned Address"`	Precise (Requires memcheck on)	Per lane/thread error	A thread has accessed an incorrectly aligned global address.
CUDA_EXCEPTION_12 : `"Warp Assert"`	Precise	Per warp	Any thread in the warp has hit a device side assertion.
CUDA_EXCEPTION_13: `"Lane Syscall Error"`	Precise (Requires memcheck on)	Per lane/thread error	A thread has corrupted the heap by invoking free with an invalid address (for example, trying to free the same memory region twice)
CUDA_EXCEPTION_14 : `"Warp Illegal Address"`	Not precise	Per warp	A thread has accessed an illegal (out of bounds) global/local/shared address. For increased precision, enable the CUDA `memcheck` option. See Enabling CUDA MemoryChecker Feature on page 653.
CUDA_EXCEPTION_15 : `"Invalid Managed Memory Access"`	Precise	Per host thread	A host thread has attempted to access managed memory currently used by the GPU.

Displaying Device Information

TotalView can display each device installed on the system, along with the properties of each SM, warp, and lane on that device. Together, these four attributes form the physical coordinates of a CUDA thread. To view the window, select **Tools > CUDA Devices**.

Figure 276, CUDA Devices when no CUDA threads are present

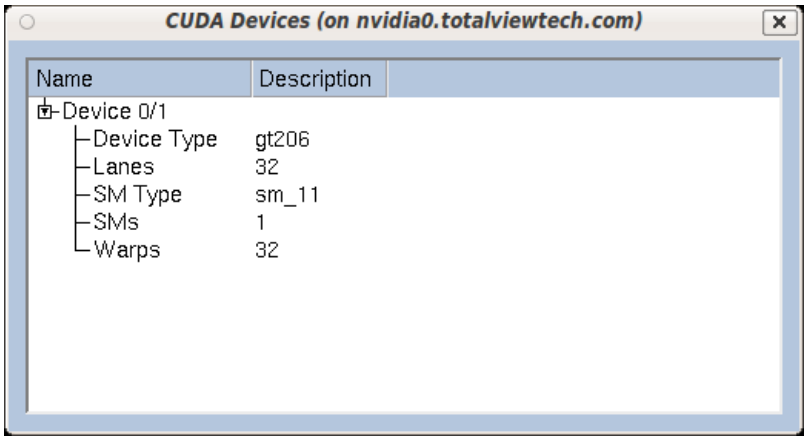
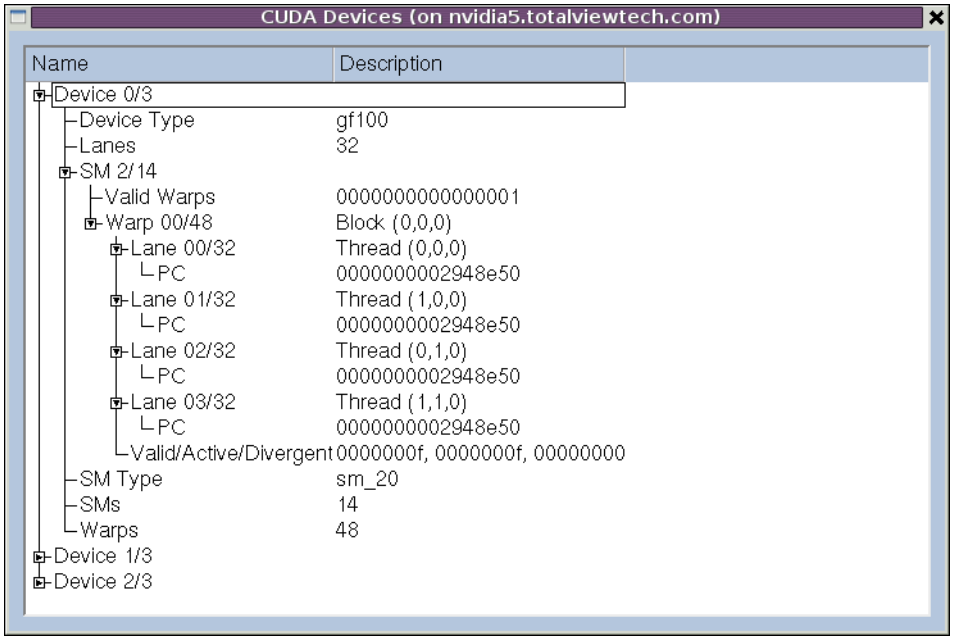


Figure 277, CUDA Devices when CUDA threads are present



CUDA Problems and Limitations

CUDA TotalView sits directly on top of the CUDA debugging environment provided by NVIDIA, which is still evolving and maturing. This environment contains certain problems and limitations, discussed in this chapter.

Hangs or Initialization Failures

When starting a CUDA debugging session, you may encounter hangs in the debugger or target application, initialization failures, or failure to launch a kernel. Use the following checklist to diagnose the problem:

Serialized Access
: There may be at most one CUDA debugging session active per node at a time. A node cannot be shared for debugging CUDA code simultaneously by multiple user sessions, or multiple sessions by the same user. Use ps or other system utilities to determine if your session is conflicting with another debugging session.

Leaky Pipes
: The CUDA debugging environment uses FIFOs (named pipes) located in "/tmp" and named matching the pattern "cudagdb_pipe.N.N", where N is a decimal number. Occasionally, a debugging session might accidentally leave a set of pipes lying around. You may need to manually delete these pipes in order to start your CUDA debugging session:

```
rm /tmp/cudagdb_pipe.*
```

If the pipes were leaked by another user, that user will own the pipes and you may not be able to delete them. In this case, ask the user or system administrator to remove them for you.

Orphaned Processes
: Occasionally, a debugging session might accidentally orphan a process. Orphaned processes might go compute bound or prevent you or other users from starting a debugging session. You may need to manually kill orphaned CUDA processes in order to start your CUDA debugging session or stop a compute-bound process. Use system tools such as ps or top to find the processes and kill them using the shell kill command. If the process were orphaned by another user, that user will own the processes and you may not be able to kill them. In this case, ask the user or system administrator to kill them for you.

Multi-threaded Programs on Fermi
: We have seen problems debugging some multi-threaded CUDA programs on Fermi, where the CUDA debugging environment kills the debugger with an internal error (SIGSEGV). We are working with NVIDIA to resolve this problem.

CUDA and ReplayEngine

You can enable ReplayEngine while debugging CUDA code; that is, ReplayEngine record mode will work. However, ReplayEngine does not support replay operations when focused on a CUDA thread. If you attempt this, you will receive a Not Supported error.

CUDA and MRNet

The CUDA API has a limitation that allows a debugger process (such as the **tvdsvr**) to debug just one target process using a GPU. If you are using the MRNet (Multicast/Reduction Network) infrastructure model and are running multiple CUDA processes on a node, you'll need to set the **TV::mrnet_super_bushy** to **true**.

This setting creates a "super bushy" MRNet tree by launching one MRNet **tvdsvr** process per target MPI process, instead of the default in which TotalView launches one **tvdsvr** process per node.

See **TV::mrnet_super_bushy** in the *Classic TotalView Reference Guide*.

Sample CUDA Program

```
/*
 * NVIDIA CUDA matrix multiply example straight out of the CUDA
 * programming manual, more or less.
 */

#include <cuda.h>
#include <stdio.h>

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
  int width;/* number of columns */
  int height;/* number of rows */
  int stride;
  float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
  return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value)
{
  A.elements[row * A.stride + col] = value;
}

// Thread block size
#define BLOCK_SIZE 2

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
  Matrix Asub;
  Asub.width = BLOCK_SIZE;
  Asub.height = BLOCK_SIZE;
  Asub.stride = A.stride;
  Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
        + BLOCK_SIZE * col];
  return Asub;
}

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
  // Load A and B to device memory
  Matrix d_A;
  d_A.width = d_A.stride = A.width; d_A.height = A.height;
  size_t size = A.width * A.height * sizeof(float);
  cudaMalloc((void**)&d_A.elements, size);
  cudaMemcpy(d_A.elements, A.elements, size,
       cudaMemcpyHostToDevice);
  Matrix d_B;
  d_B.width = d_B.stride = B.width; d_B.height = B.height;
  size = B.width * B.height * sizeof(float);
  cudaMalloc((void**)&d_B.elements, size);
  cudaMemcpy(d_B.elements, B.elements, size,
       cudaMemcpyHostToDevice);
  // Allocate C in device memory
  Matrix d_C;
  d_C.width = d_C.stride = C.width; d_C.height = C.height;
  size = C.width * C.height * sizeof(float);
  cudaMalloc((void**)&d_C.elements, size);
  // Invoke kernel
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
  MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
  // Read C from device memory
  cudaMemcpy(C.elements, d_C.elements, size,
       cudaMemcpyDeviceToHost);
  // Free device memory
  cudaFree(d_A.elements);
  cudaFree(d_B.elements);
  cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;
  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
```

```
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }
  // Write Csub to device memory
  // Each thread writes one element
  SetElement(Csub, row, col, Cvalue);
  // Just a place to set a breakpoint in the debugger
  __syncthreads();
  __syncthreads();/* STOP: Csub should be fully updated */
}

static Matrix
cons_Matrix (int height_, int width_)
{
  Matrix A;
  A.height = height_;
  A.width = width_;
  A.stride = width_;
  A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
  for (int row = 0; row < height_; row++)
    for (int col = 0; col < width_; col++)
      A.elements[row * width_ + col] = row * 10.0 + col;
  return A;
}

static void
print_Matrix (Matrix A, char *name)
{
  printf("%s:\n", name);
  for (int row = 0; row < A.height; row++)
    for (int col = 0; col < A.width; col++)
      printf ("[%5d][%5d] %f\n", row, col, A.elements[row * A.stride + col]);
}

// Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
// Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
// m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
int main(int argc, char **argv)
{
//  cudaSetDevice(0);
  const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
  const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
  const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
  Matrix A = cons_Matrix(m, n);
  Matrix B = cons_Matrix(n, p);
  Matrix C = cons_Matrix(m, p);
  MatMul(A, B, C);
  print_Matrix(A, "A");
  print_Matrix(B, "B");
  print_Matrix(C, "C");
  return 0;
}
```

PART VI Appendices

This part contains appendices that are essentially reference material:

- Appendix A, Glossary, on page 667
  Definitions for technical terms used in this documentation.
- Appendix B, Open Source Software Notice, on page 685
  Licenses for 3rd-party software.
- Appendix C, Resources, on page 686
  General information on resources available to you.

Appendix A
Glossary

ACTION POINT: A debugger feature that lets a user request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, eval points, and barriers.

ACTION POINT IDENTIFIER:

A unique integer ID associated with an action point.

ACTIVATION RECORD:

See stack frame on page 681.

ADDRESS SPACE:

A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

ADDRESSING EXPRESSION:

A set of instructions that tell TotalView where to find information. These expressions are only used within the type transformation facility.

AFFECTED P/T SET:

The set of process and threads that are affected by the command. For most commands, this is identical to the target P/T set, but in some cases it might include additional threads. (See p/t (process/thread) set on page 678 for more information.)

AGGREGATE DATA:

A collection of data elements. For example, a structure or an array is an aggregate.

AGGREGATED OUTPUT:

The CLI compresses output from multiple threads when they would be identical except for the P/T identifier.

API:

Application Program Interface. The formal interface by which programs communicate with libraries.

ARENA:

A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

ARRAY SECTION:

In Fortran, a portion of an array that is also an array. The elements of this array is a new unnamed array object with its own indices. Compare this with a TotalView array slice on page 668.

ARRAY SLICE:

A subsection of an array, which is expressed in terms of a lower bound on page 675, upper bound on page 684, and stride on page 682. Displaying a slice of an array can be useful when you are working with very large arrays. Compare this with a TotalView array section on page 668.

ASYNCHRONOUS:

When processes communicate with one another, they send messages. If a process decides that it doesn't want to wait for an answer, it is said to run "asynchronously." For example, in most client/server programs, one program sends an RPC request to a second program and then waits to receive a response from the second program. This is the normal *synchronous* mode of operation. If, however, the first program sends a message and then continues executing, not waiting for a reply, the first mode of operation is said to be *asynchronous*.

ATTACH:

The ability for TotalView to gain control of an already running process on the same machine or a remote machine.

AUTOLAUNCHING:

When a process begins executing on a remote computer, TotalView can also launch a **tvdsvr** (TotalView Debugger Server) process on the computer that will send debugging information back to the TotalView process that you are interacting with.

AUTOMATIC PROCESS ACQUISITION:

TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you don't have to attach to them manually. If the process is on a remote computer, automatic process acquisition automatically starts the TotalView Debugger Server (**tvdsvr**).

BARRIER POINT:

An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

BASE WINDOW:

The original Process Window or Variable Window before you dive into routines or variables. After diving, you can use a **Reset** or **Undive** command to restore this original window.

BLOCKED:

A thread state in which the thread is no longer executing because it is waiting for an event to occur. In most cases, the thread is blocked because it is waiting for a mutex or condition state.

BREAKPOINT:

A point in a program where execution can be suspended to permit examination and manipulation of data.

BUG:

A programming error. Finding them is why you're using TotalView.

BULK LAUNCH:

A TotalView procedure that launches multiple tvdsvr processes simultaneously.

CAF:

See CoArray Fortran.

CALL FRAME:

The memory area that contains the variables belonging to a function, subroutine, or other scope division, such as a block.

CALL STACK:

A higher-level view of stack memory, interpreted in terms of source program variables and locations. This is where your program places stack frames.

CALLBACK:

A function reference stored as a pointer. By using the function reference, this function can be invoked. For example, a program can hand off the function reference to an event processor. When the event occurs, the function can be called.

CHILD PROCESS:

A process created by another process (*see* parent process on page 677) when that other process calls the **fork()** function.

CLOSED LOOP:

See closed loop on page 669.

CLUSTER DEBUGGING:

The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are of the same type and have the same operating system version.

COARRAY FORTRAN (CAF):

A variation of Fortran with array syntax augmented to depict arrays distributed across processes.

COMMAND HISTORY LIST:

A debugger-maintained list that stores copies of the most recent commands issued by the user.

CONDITION SYNCHRONIZATION:

A process that delays thread execution until a condition is satisfied.

CONDITIONAL BREAKPOINT:

A breakpoint containing an expression. If the expression evaluates to true, program stops. TotalView does not have conditional breakpoints. Instead, you must explicitly tell TotalView to end execution by using the $stop directive.

CONTEXT SWITCHING:

In a multitasking operating system, the ability of the CPU to move from one task to another. As a switch is made, the operating system must save and restore task states.

CONTEXTUALLY QUALIFIED (SYMBOL):

A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

CONTROL GROUP:

All the processes that a program creates. These processes can be local or remote. If your program uses processes that it did not create, TotalView places them in separate control groups. For example, a client/server program has two distinct executables that run independently of one another. Each would be in a separate control group. In contrast, processes created by the **fork()** function are in the same control group.

CORE FILE:

A file that contains the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

CORE-FILE DEBUGGING:

A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

CPU:

Central Processing Unit. The component within the computer that most people think of as "the computer". This is where computation and activities related to computing occur.

CROSS-DEBUGGING:

A special case of remote debugging where the host platform and the target platform are different types of machines.

CURRENT FRAME:

The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

CURRENT LANGUAGE:

The source code language used by the file that contains the current source location.

CURRENT LIST LOCATION:

The location governing what source code appears in response to a list command.

DATASET:

A set of array elements generated by TotalView and sent to the Visualizer. (See visualizer process on page 684.)

DBELOG LIBRARY:

A library of routines for creating event points and generating event logs from TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

DBFORK LIBRARY:

A library of special versions of the **fork()** and **execve()** calls used by TotalView to debug multi-process programs. If you link your program with the TotalView **dbfork** library, TotalView can automatically attach to newly spawned processes.

DEADLOCK:

A condition where two or more processes are simultaneously waiting for a resource such that none of the waiting processes can execute.

DEBUGGING INFORMATION:

Information relating an executable to the source code from which it was generated.

DEBUGGER PROMPT:

A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER:

See tvdsvr process on page 683.

DEBUGGER STATE:

Information that TotalView or the CLI maintains to interpret and respond to user commands. This includes debugger modes, user-defined commands, and debugger variables.

DEPRECATED:

A feature that is still available but might be eliminated in a future release.

DISASSEMBLED CODE:

A symbolic translation of binary code into assembler language.

DISTRIBUTED DEBUGGING:

The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Macros (PARMACS), run on more than one host.

DIVING:

The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

DLL:

Dynamic Link Library. A shared library whose functions can be dynamically added to a process when a function with the library is needed. In contrast, a statically linked library is brought into the program when it is created.

DOPE VECTOR:

This is a run time descriptor that contains all information about an object that requires more information than is available as a single pointer or value. For example, you might declare a Fortran 90 pointer variable that is a pointer to some other object, but which has its own upper bound, as follows:

```
integer, pointer, dimension (:) :: iptr
```

Suppose that you initialize it as follows:

```
iptr => iarray (20:1:-2)
```

iptr is a synonym for every other element in the first twenty elements of **iarray**, and this pointer array is in reverse order. For example, **iptr(1)** maps to **iarray(20)**, **iptr(2)** maps to **iarray(18)**, and so on.

A compiler represents an **iptr** object using a run time descriptor that contains (at least) elements such as a pointer to the first element of the actual data, a stride value, and a count of the number of elements (or equivalently, an upper bound).

DPID:

Debugger ID. This is the ID TotalView uses for processes.

DYNAMIC LIBRARY:

A library that uses dynamic loading to load information in an external file at runtime. Dynamic loading implies dynamic linking, which is a process that does not copy a program and its data into the executable at compile time.

EDITING CURSOR:

A black line that appears when you select a TotalView GUI field for editing. You use field editor commands to move the editing cursor.

EVAL POINT:

A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG:

A file that contains a record of events for each process in a program.

EVENT POINT:

A point in the program where TotalView writes an event to the event log for later analysis with TimeScan.

EXCEPTION:

A condition generated at runtime that indicates that a non-standard event has occurred. The program usually creates a method to handle the event. If the event is not handled, either the program's result will be inaccurate or the program will stop executing.

EXECUTABLE:

A compiled and linked version of source files

EXPRESSION SYSTEM:

A part of TotalView that evaluates C, C++, and Fortran expressions. An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of a source language. Not all Fortran 90, C, and C++ operators are supported.

EXTENT:

The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

FIELD EDITOR:

A basic text editor that is part of TotalView. The field editor supports a subset of GNU Emacs commands.

FOCUS:

The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you're using the default prompt).

FRAME:

An area in stack memory that contains the information corresponding to a single invocation of a subprocedure. See stack frame on page 681.

FRAME POINTER:

See stack pointer on page 681.

FULLY QUALIFIED (SYMBOL):

A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

GARBAGE COLLECTION:

Examining memory to determine if it is still be referenced. If it is not, it sent back to the program's memory manager so that it can be reused.

GID:

The TotalView group ID.

GLOBAL ARRAYS:

(from a definition on the Global Arrays web site) The Global Arrays (GA) toolkit provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without need for explicit cooperation by other processes. For more information, see http://hpc.pnl.gov/globalarrays/.

GRID:

A collection of distributed computing resources available over a local or wide area network that appears as if it were one large virtual computing system.

GOI:

The group of interest. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

GROUP:

When TotalView starts processes, it places related processes in families. These families are called "groups."

GROUP OF INTEREST:

The primary group that is affected by a command. This is the group that TotalView uses when it is trying to determine what to step, stop, and so on.

HEAP:

An area of memory that your program uses when it dynamically allocates blocks of memory. It is also how people describe my car.

HOST COMPUTER:

The computer on which TotalView is running.

IMAGE:

All of the programs, libraries, and other components that make up your executable.

INFINITE LOOP:

See loop, infinite on page 675.

INSTRUCTION POINTER:

See program counter.

INITIAL PROCESS:

The process created as part of a load operation, or that already existed in the runtime environment and was attached by TotalView or the CLI.

INITIALIZATION FILE:

An optional file that establishes initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

INTERPRETER:

A program that reads programming language statements and translates the statements into machine code, then executes this code.

LAMINATE:

A process that combines variables contained in separate processes or threads into a unified array for display purposes.

LHS EXPRESSION:

This is a synonym for **lvalue**.

LINKER:

A program that takes all the object files creates by the compiler and combines them and libraries required by the program into the executable program.

LOCKSTEP GROUP:

All threads that are at the same PC (program counter). This group is a subset of a workers group. A lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group. By definition, all members of a lockstep group are in the same workers group. That is, a lockstep group cannot have members in more than one workers group or more than one control group.

LOOP, INFINITE:

See infinite loop on page 674.

LOWER BOUND:

The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

LVALUE:

A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

MACHINE STATE:

Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

MANAGER THREAD:

A thread created by the operating system. In most cases, you do not want to manage or examine manager threads.

MESSAGE QUEUE:

A list of messages sent and received by message-passing programs.

MIMD:

An acronym for Multiple Instruction, Multiple Data, which describes a type of parallel computing.

MISD:

An acronym for Multiple Instruction, Single Data, which describes a type of parallel computing.

MPI:

An acronym for “Message Passing Interface.”

MPICH:

MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see https://www.mcs.anl.gov/research/projects/mpi/.

MPMD PROGRAMS:

An acronym for Multiple Program, Multiple Data, which describes programs that involve multiple executables, executed by multiple threads and processes.

MULTITASK:

In the context of high performance computing, this is the ability to divide a program into smaller pieces or tasks that execute separately.

MULTI-PROCESS:

The ability of a program to spawn off separate programs, each having its own context and memory. multi-process programs can (and most often do) run processes on more than one computer. They can also run multiple processes an one computer. In this case, memory can be shared

MULTI-THREADED:

The ability of a program to spawn off separate tasks that use the same memory. Switching from task to task is controlled by the operating system.

MUTEX (MUTUAL EXCLUSION):

Techniques for sharing resources so that different users do not conflict and cause unwanted interactions.

NATIVE DEBUGGING:

The action of debugging a program that is running on the same machine as TotalView.

NESTED DIVE:

TotalView lets you dive into pointers, structures, or arrays in a variable. When you dive into one of these elements, TotalView updates the display so that the new element appears. A nested dive is a *dive* within a dive. You can return to the previous display by selecting the left arrow in the top-right corner of the window.

NODE:

A machine on a network. Each machine has a unique network name and address.

NULLIFIED:

A breakpoint expression that, when reevaluated, points to an invalid address block.

OFF-BY-ONE:

An error usually caused by forgetting that arrays begin with element 0 in C and C++.

OPENMP:

(from a definition on the OpenMP web site) OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. The MP in OpenMP stands for Multi Processing. We provide Open specifications for Multi Processing via collaborative work with interested parties from the hardware and software industry, government and academia. For more information, see https://openmp.org/.

OUT-OF-SCOPE:

When symbol lookup is performed for a particular symbol name and it isn't found in the current scope or any that contains scopes, the symbol is said to be out-of-scope.

PAGE PROTECTION:

The ability to segregate memory pages so that one process cannot access pages owned by another process. It can also be used to generate an exception when a process tries to access the page.

PARALLEL PROGRAM:

A program whose execution involves multiple threads and processes.

PARALLEL TASKS:

Tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results.

PARALLELIZABLE PROBLEM:

A problem that can be divided into parallel tasks. This type of program might require changes in the code and/or the underlying algorithm.

PARCEL: The number of bytes required to hold the shortest instruction for the target architecture.

PARENT PROCESS:

A process that calls the **fork()** function to spawn other processes (usually called child processes).

PARMACS LIBRARY:

A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

PARTIALLY QUALIFIED (SYMBOL):

A symbol name that includes only some of the levels of source code organization (for example, file name and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PATCHING:

Inserting code in a breakpoint that is executed immediately preceding the breakpoint's line. The patch can contain a GOTO command to branch around incorrect code.

PC:

An abbreviation for *Program Counter*.

PID:

Depending on the context, this is either the process ID or the program ID. In most cases, this is the process ID.

POI:

The process of interest. This is the process that TotalView uses when it is trying to determine what to step, stop, and so on.

/PROC:

An interface that allows debuggers and other programs to control or obtain information from running processes. ptrace also does this, but /proc is more general.

PROCESS:

An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP:

A group of processes associated with a multi-process program. A process group includes program control groups and share groups.

PROCESS/THREAD IDENTIFIER:

A unique integer ID associated with a particular process and thread.

PROCESS OF INTEREST:

The primary process that TotalView uses when it is trying to determine what to step, stop, and so on.

PROGRAM CONTROL GROUP:

A group of processes that includes the parent process and all related processes. A program control group includes children that were forked (processes that share the same source code as the parent), and children that were forked with a subsequent call to the **execve()** function (processes that don't share the same source code as the parent). Contrast this with share group on page 679.

PROGRAM EVENT:

A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

PROGRAM STATE:

A higher-level view of the machine state, where addresses, instructions, registers, and such are interpreted in terms of source program variables and statements.

P/T (PROCESS/THREAD) SET:

The set of threads drawn from all threads in all processes of the target program.

PTHREAD ID:

This is the ID assigned by the Posix pthreads package. If this differs from the system TID, it is a pointer value that points to the pthread ID.

QUEUE:

A data structure whose data is accessed in the order in which it was entered. This is like a line at a tollbooth where the first in is the first out.

RACE CONDITION:

A problem that occurs when threads try to simultaneously access a resource. The result can be a deadlock, data corruption, or a program fault.

REMOTE DEBUGGING:

The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

RESUME COMMANDS:

Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

RHS EXPRESSION:

This is a synonym for **rvalue**.

RVALUE:

An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

SATISFACTION SET:

The set of processes and threads that must be held before a barrier can be satisfied.

SATISFIED:

A condition that indicates that all processes or threads in a group have reached a barrier. Prior to this event, all executing processes and threads are either running because they have not yet hit the barrier, or are being held at the barrier because not all of the processes or threads have reached it. After the barrier is *satisfied*, the held processes or threads are released, which means they can be run. Prior to this event, they could not run.

SCOPE:

The region in your program in which a variable or a function exists or is defined. This region begins with its declaration and extends to the end of the current block.

SEARCH PATH:

A list that contains places that software looks to locate files contained within the file system. In TotalView, the search path contains locations containing your program's source code.

SERIAL EXECUTION:

Execution of a program sequentially, one statement at a time.

SERIAL LINE DEBUGGING:

A form of remote debugging where TotalView and the **tvdsvr** communicate over a serial line.

SERVICE THREAD:

A thread whose purpose is to *service* or manage other threads. For example, queue managers and print spoolers are service threads. There are two kinds of service threads: those created by the operating system or runtime system and those created by your program.

SHARE GROUP:

All the processes in a control group that share the same code. In most cases, your program has more than one share group. Share groups, like control groups, can be local or remote.

SHARED LIBRARY:
A compiled and linked set of source files that are dynamically loaded by other executables.

SIGNALS:
Messages informing processes of asynchronous events, such as serious errors. The action that the process takes in response to the signal depends on the type of signal and whether the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

SIMD:
An acronym for Single Instruction, Multiple Data, which describes a type of parallel computing.

SINGLE PROCESS SERVER LAUNCH:
A TotalView procedure that individually launches tvdsvr processes.

SINGLE STEP:
The action of executing a single statement and stopping (as if at a breakpoint).

SISD:
An acronym for Single Instruction, Single Data, which describes a type of parallel computing.

SLICE:
A subsection of an array, which is expressed in terms of a lower bound on page 675, upper bound on page 684, and stride on page 682. Displaying a slice of an array can be useful when you are working with very large arrays. Compare this with a TotalView array section on page 668.

SOID:
An acronym for symbol object ID. A SOID uniquely identifies all TotalView information. It also represents a handle by which you can access this information.

SOURCE FILE:
Program file that contains source language statements. TotalView lets you debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler files.

SOURCE LOCATION: For each thread, the source code line it executes next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

SPAWNED PROCESS:
The process created by a user process executing under debugger control.

SPMD PROGRAMS:
An acronym for Single Program, Multiple Data, which describe a type of parallel computing that involves just one executable, executed by multiple threads and processes.

STACK:

A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

STACK FRAME:

Whenever your program calls a function, it creates a set of information that includes the local variables, arguments, contents of the registers used by an individual routine, a frame pointer pointing to the previous stack frame, and the value of the program counter (PC) at the time the routine was called. The information for one function is called a "stack frame" as it is placed on your program's stack.

When your program begins executing, it has only one frame: the one allocated for function **main()**. As your program calls functions, new frames are allocated. When a function returns to the function from which it is called, the frame is deallocated.

STACK POINTER:

A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored. This is also called a frame pointer.

STACK TRACE:

A sequential list of each currently active routine called by a program, and the frame pointer that points to its stack frame.

STATIC (SYMBOL) SCOPE:

A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

STEPPING:

Advancing program execution by fixed increments, such as by source code statements.

STL:

An acronym for Standard Template Library.

STOP SET:

A set of threads that TotalView stops after an action point triggers.

STOPPED/HELD STATE:

The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) is required before it is capable of continuing execution.

STOPPED/RUNNABLE STATE:

The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

STOPPED STATE:

The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

STRIDE:

The interval between array elements in a slice and the order in which TotalView displays these elements. If the stride is 1, TotalView displays every element between the lower bound and upper bound of the slice. If the stride is 2, TotalView displays every other element. If the stride is -1, TotalView displays every element between the upper bound and lower bound (reverse order).

SYMBOL:

Entities within program state, machine state, or debugger state.

SYMBOL LOOKUP:

Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively so that contains scopes are searched in an outward progression.

SYMBOL NAME:

The name associated with a symbol known to TotalView (for example, function, variable, data type, and so on).

SYMBOL TABLE:

A table of symbolic names used in a program (such as variables or functions) and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **-g** option) and is used by debuggers to analyze the program.

SYNCHRONIZATION:

A mechanism that prevents problems caused by concurrent threads manipulating shared resources. The two most common mechanisms for synchronizing threads are *mutual exclusion* and *condition synchronization*.

TARGET COMPUTER:

The computer on which the process to be debugged is running.

TARGET PROCESS SET:

The target set for those occasions when operations can only be applied to entire processes, not to individual threads in a process.

TARGET PROGRAM:

The executing program that is the target of debugger operations.

TARGET P/T SET:

The set of processes and threads that a CLI command acts on.

TASK:

A logically discrete section of computational work. (This is an informal definition.)

THREAD:

An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

THREAD EXECUTION STATE:

The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of predefined states.

THREAD OF INTEREST:

The primary thread affected by a command. This is abbreviated as TOI.

TID:

The thread ID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

TLA:

An acronym for Three-Letter Acronym. So many things from computer hardware and software vendors are referred to by a three-letter acronym that yet another acronym was created to describe these terms.

TOI:

The thread of interest. This is the primary thread affected by a command.

TRIGGER SET:

The set of threads that can trigger an action point (that is, the threads upon which the action point was defined).

TRIGGERS:

The effect during execution when program operations cause an event to occur (such as arriving at a breakpoint).

TTF:

See type transformation facility on page 683.

TRAP:

An instruction that stops program execution and which allows a debugger to gain control over your program.

TVDSVR PROCESS:

The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

TYPE TRANSFORMATION FACILITY:

This is abbreviated as TTF. A TotalView subsystem that allows you to change the way information appears. For example, an STL vector can appear as an array.

UNDISCOVERED SYMBOL:

A symbol that is referred to by another symbol. For example, a **typedef** is a reference to the aliased type.

UNDIVING:

The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you click the **undive** icon in the upper-right corner of the window.

UPC:

(from a definition on the UPC web site) The Unified Parallel C language, which is an extension to the C programming language that is designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. See http://upc.lbl.gov/ for more information.

UPPER BOUND:

The last element in the dimension of an array or the slice of an array.

USER THREAD:

A thread created by your program.

USER INTERRUPT KEY:

A keystroke used to interrupt commands, most commonly defined as Ctrl+C.

VARIABLE WINDOW:

A TotalView window that displays the name, address, data type, and value of a particular variable.

VISUALIZATION:

In TotalView, visualization means graphically displaying an array's values.

VISUALIZER PROCESS:

A process that works with TotalView in a separate window, allowing you to see a graphic representation of program array data.

WATCHPOINT:

An action point that tells TotalView to stop execution when the value of a memory location changes.

WORKER THREAD:

A thread in a workers group. These are threads created by your program that performs the task for which you've written the program.

WORKERS GROUP:

All the worker threads in a control group. Worker threads can reside in more than one share group.

Appendix B

Open Source Software Notice

TotalView publishes the open source software products it uses in the ATTRIBUTION.HTML file located in the **doc** directory where you installed TotalView.

Appendix C

Resources

TotalView Family Differences

This section describes the TotalView Enterprise, TotalView Team, and TotalView Developer debuggers. Each of these supports the use of the CLI debugger as well.

NOTE: The most fundamental differences between TotalView Team and TotalView Enterprise are the way resources are shared and used. When you purchase TotalView Team, you are purchasing "tokens" that represent the number of job processes that can be run at a single time. For example, if you have 64 tokens available, 64 users could each debug a one-process job; or two users could each debug a 32-process job. In contrast, a TotalView Enterprise license is based on the number of users and the number of licensed processors. You'll find more precise information on our web site.

The basic differences are:

Topic	TotalView Team	TotalView Enterprise	TotalView Developer
Architecture limitations	Execute on any licensed computer of the same architecture.	Execute on any licensed computer of the same architecture.	Execute on any licensed computer of the same architecture but only by a single, named user.
Processor limitations	Unlimited	Determined by license.	Can only run on machines with 32 or less processors.
Process/thread limitations	Processes limited by the number of tokens, one token equals one process. No limit on threads.	Unlimited processes, but only on number of processors specified.	No more than 32 processes and threads.
User limits	User limit is determined by the number of tokens.	User limit is determined by the license.	Single named user allowed to use it on multiple machines.
Remote access	Processes can execute on any computers in the same network.	Processes can execute on any computers in the same network.	Processes must execute on the installed computer.

TotalView Documentation

The following table describes all available TotalView documentation:

Product	Title	Description	HTML	PDF	Print
General TotalView Documentation					
	Getting Started with TotalView Products	Introduces the basic features of TotalView, MemoryScape, and ReplayEngine, with links for more detailed information			
	TotalView Platforms Guide	Specifies supported platforms for TotalView, MemoryScape, and ReplayEngine			
	TotalView Evaluation Guide	Brochure that introduces basic TotalView features			
User Guides					
	TotalView User Guide	Primary resource for information on using the TotalView GUI and the CLI			
	Debugging Memory Problems with MemoryScape	How to debug memory issues, relevant to both TotalView and the MemoryScape standalone product			
	Reverse Debugging with Replay Engine	How to perform reverse debugging using the embedded add-on ReplayEngine			
Reference Guides					
	TotalView Reference Guide	A reference of CLI commands, how to run TotalView, and platform-specific detail			
New Features		New features in the current release			
	What's new in this release	On the landing page of the HTML documentation, lists new features for the documented release			
	TotalView Change Log	Details the changes to the product from release to release			
Installation Guides					

Product	Title	Description	HTML	PDF	Print
	TotalView Install Guide	Installing TotalView and the FLEX*lm* license manager			
	MemoryScape Install Guide	Installing MemoryScape as a standalone product			
In-Product Help		Help screens launched from within the product's GUI			
	TotalView Help	Context-sensitive help launched from TotalView			
	MemoryScape Help	Context-sensitive help launched from MemoryScape			

Conventions

This section describes the formatting conventions used throughout the documentation.

Table 5: Formatting conventions in commands and their descriptions

Convention	Meaning
Bold	Bold formats literal text, i.e., commands, keywords, or options that must be entered exactly as displayed. For example: **dactions -save** [*filename*]
italics	Italics format parameter values entered by the user, for example: **dactions -save** *myFile*
[ ]	Brackets describe optional parts of a command. For example, in this command, both **-g** and **-r** are optional parameters: **dattach** [**-g** *gid*] [**-r** *hname*]
{ }	Curly braces wrap a group of entries where exactly one choice is required. For example, in this command, if you provide the **-c** parameter, you must provide a parameter value: **dattach** [**-c** { *core-file* | *recording-file* }]
|	A vertical bar identifies a choice, for example, provide either a *core-file* or a *recording-file*: **dattach** [**-c** { *core-file* | *recording-file* }]
`Monospace`	Example code or a response to a shell or CLI prompt

Table 6: Formatting conventions in regular text

Convention	Meaning
Bold	Words that are used in a programmatic way rather than their normal way, including UI elements and menus such as "**File |Preferences**."
italics	Emphasis
`Monospace`	Example code or a response to a shell or CLI prompt

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.

- **By email**: techsupport@roguewave.com
- **By phone**: See TotalView's Contact Support page (https://totalview.io/support#contact-support) for support numbers.

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.

Index

Symbols

A

B

C

D

E

F

G

H

I

M

S

T

U

V

W

X

Y

Z