TotalView

Classic TotalView Getting Started Guide

Version 2023.4
November, 2023

PERFORCE

www.perforce.com

Copyright 2007-2023 by Rogue Wave Software, Inc., a Perforce company (“Rogue Wave”). All rights reserved.
Copyright 1998–2007 by Etnus LLC. All rights reserved.
Copyright 1996–1998 by Dolphin Interconnect Solutions, Inc.
Copyright 1993–1996 by BBN Systems and Technologies, a division of BBN Corporation.
All trademarks and registered trademarks are the property of their respective owners.

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

Perforce has prepared this manual for the exclusive use of its customers, personnel, and licensees. The information in this manual is subject to change without notice, and should not be construed as a commitment by Perforce. Perforce assumes no responsibility for any errors that appear in this document.
TotalView and TotalView Technologies are registered trademarks of Rogue Wave. TVD is a trademark of Rogue Wave.
Perforce uses a modified version of the Microline widget library. Under the terms of its license, you are entitled to use these modifications. The source code is available at https://rwkbp.makekb.com/.
All other brand names are the trademarks of their respective holders.

ACKNOWLEDGMENTS

Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Perforce Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation.

ROGUE WAVE MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL PERFORCE SOFTWARE, INC. BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

TotalView by Perforce
http://totalview.io

Contents

Getting Started with TotalView Products

The Basics 2
What is TotalView 2
Starting TotalView and Creating a Debugging Session 3
Loading Programs into TotalView for Debugging 5
Managing Debugging Sessions 6
Getting Around TotalView 7
TotalView Commands 8
The Root Window 8
The Process Window 10
Variable Window and Array Viewer 11
Accessing TotalView Remotely 16
Debugging on a Remote Host 17
Setting Breakpoints and Stepping through a Program 18
Action Points (breakpoints) 18
Stepping Through a Program 19
Examining and Editing Data 21
Diving and Viewing Data 21
Editing Data 23
Evaluating Expressions 23
Working with Multi-Processes and Multi-Threads 25
Starting a Parallel Debugging Job 26
Working with and Viewing Processes and Threads 26
Debugging Using the Command Line Interface (CLI) 30
Debugging CUDA Programs 31
Memory Debugging 32
Viewing Memory Event Information 33
Finding Memory Leaks 34
Detecting Memory Corruption 35
Analyzing Memory 36
Finding Dangling Pointers 36
Setting and Using Baselines 37

Reverse Debugging with ReplayEngine 39

Getting Started with TotalView Products

This guide provides an introduction to TotalView®'s basic features to help you get quickly started. Each topic here is introduced briefly, with links to more detailed discussions for more information.

NOTE: This guide does not discuss all of TotalView's features, but rather focuses on those primary tools that all programmers need when debugging programs of any complexity and scope. For a more complete introduction to TotalView, see "About TotalView" in the *Classic TotalView User Guide*.

This guide includes:

- The Basics on page 2
- Accessing TotalView Remotely on page 16
- Debugging on a Remote Host on page 17
- Setting Breakpoints and Stepping through a Program on page 18
- Examining and Editing Data on page 21
- Working with Multi-Processes and Multi-Threads on page 25
- Debugging Using the Command Line Interface (CLI) on page 30
- Debugging CUDA Programs on page 31
- Memory Debugging on page 32
- Reverse Debugging with ReplayEngine on page 39

The Basics

What is TotalView

TotalView is a source- and machine-level debugger for multi-process, multi-threaded programs. Its wide range of tools provides ways to analyze, organize, and test programs, making it easy to isolate and identify problems in individual threads and processes in programs of great complexity.

It includes two primary interfaces: the Graphical User Interface (GUI) and the Command Line Interface (CLI) running within an xterm-like window for typing commands. Generally, the GUI provides tools and displays data in a way that is easy to work with and understand, and is the recommended way to use TotalView. However, the two interfaces complement one another and can be used simultaneously, providing the most complete approach to debugging as well as access to the power of all TotalView's tools.

Shipped Examples

TotalView ships with several examples that illustrate many of its features. These examples are located in your TotalView installation directory at *installdir*`/toolworks/totalview.`*version*`/`*platform*`/examples/`. You can load these examples into TotalView at startup and explore how it works.

Documentation

TotalView provides TotalView user and reference guides, as well as MemoryScape and ReplayEngine user guides. In addition, context-sensitive help launches when you click the **Help** button while using TotalView (and is also available directly from the shipped html documentation under "In-Product Help.")

More Information

For more information on the documentation set, conventions used in the documentation, and contact information, see the Resources section of the *User Guide*.

Starting TotalView and Creating a Debugging Session

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.

NOTE: The most common way to start TotalView is by entering:

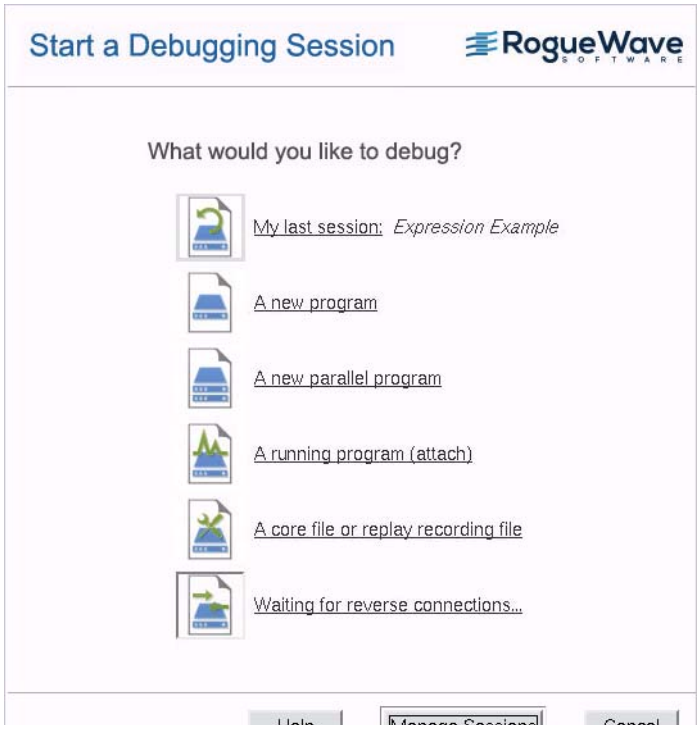
totalview *program_name*

where ***program_name*** is the executable of the program you are debugging.

NOTE: To run a program in TotalView, compile it for debugging, usually with the **-g** command-line option, depending on your compiler.

Starting TotalView with no arguments (i.e. just **totalview**) launches the Start a Debugging Session dialog.

Figure 1 TotalView Sessions Manager: Start a Debugging Session dialog



This dialog is part of the Sessions Manager and is the easiest way to load a program into TotalView. Once you configure a debugging session using this dialog, the settings are saved so you can access them later.

From here, you can:

1. **Open a debugging session**

   Select a type of session:

   A new program to launch the Program Session dialog and selecting a program to debug (equivalent to starting TotalView with a program name argument).

   A new parallel program to launch the Parallel Program Session dialog and entering parallel system settings.

   A running program (attach) to launch the Attach a running program(s) dialog and selecting an already-running process.

   A core file or replay recording file to launch the Core or Replay Recording Session File dialog and selecting an executable and associated core file or replay recording session file.

My last session
to launch a debugging session using your last session.

2. **Manage your debugging sessions**

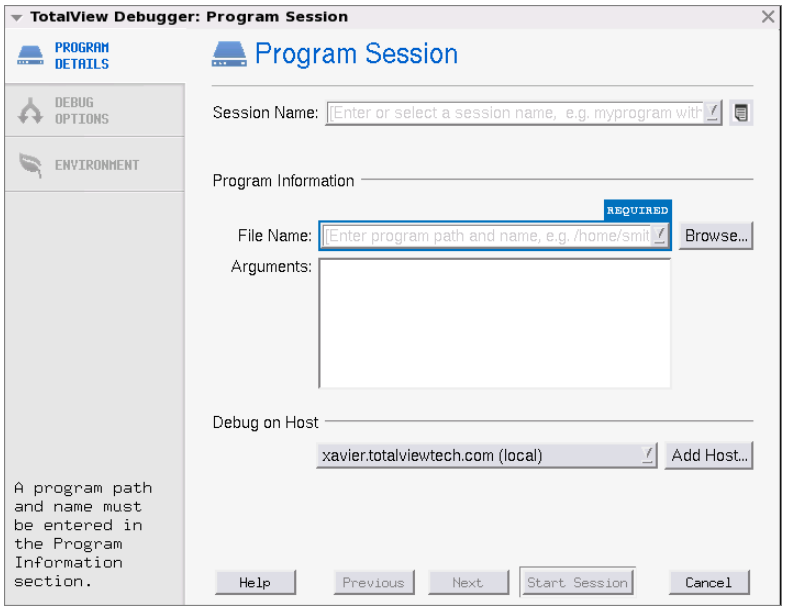
   Select **Manage Sessions** to edit, delete, or view the details of any saved sessions.

RELATED TOPICS	
Compiling your program for debugging	"Compiling Programs" in the *Classic TotalView User Guide*

Loading Programs into TotalView for Debugging

When you select the type of debugging session you want from the Start a Debugging Session dialog, the relevant screen launches to help you configure your session. For instance, selecting **A new program** launches the Program Session dialog.

Figure 2 Program Session dialog



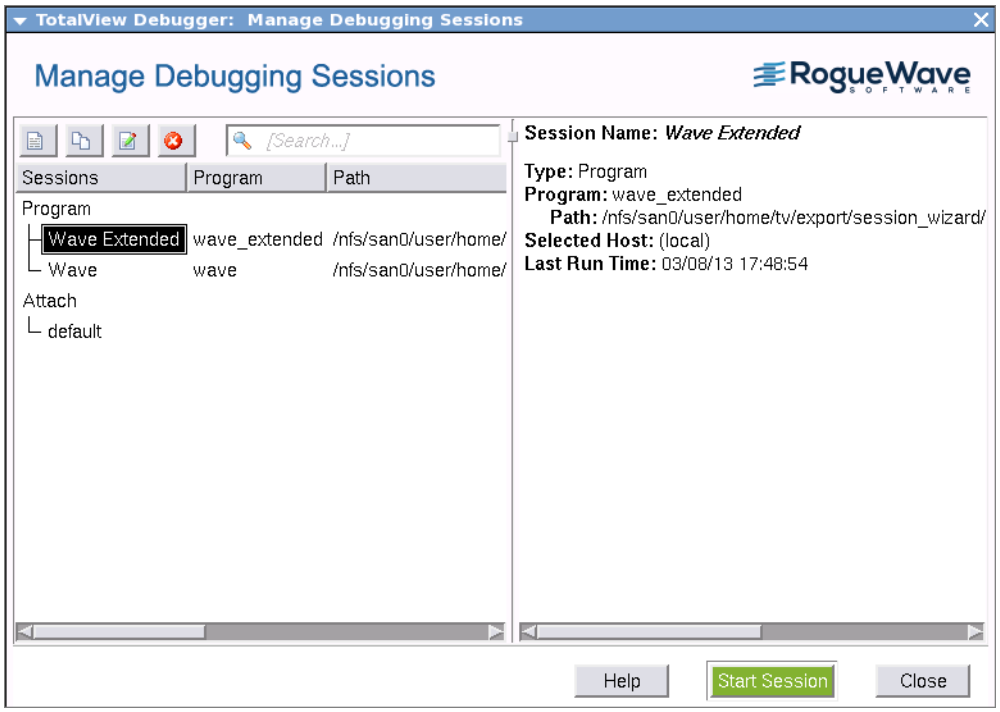
Here, you enter a name for your session, the file to debug and any arguments. Use the **Debug Options** and **Environment** tabs (at left) to further configure your session.

RELATED TOPICS	
Loading programs into TotalView using the Sessions Manager	"Loading Programs from the Session Manager" in the *Classic TotalView User Guide*
Loading programs into TotalView using the CLI	"Loading Programs Using the CLI" in the *Classic TotalView User Guide*

Managing Debugging Sessions

In the Sessions Manager, select the **Manage Sessions** button to launch the Manage Debugging Sessions dialog.

Figure 3 Manage Debugging Sessions



This dialog displays previously configured debugging sessions. From here, you can edit, duplicate, or delete a session, as well as view its configuration. In addition, you can start a debugging session using the green **Start Session** button.

You can also access the Sessions Manager via **File > Manage Sessions** on both the Root and Process windows.

Figure 4 Root Window, File > Manage Sessions

RELATED TOPICS	
The Sessions Manager	"Managing Sessions" in the *Classic TotalView User Guide*
Additional ways to start TotalView	"Starting TotalView" in the *Classic TotalView User Guide*
Command line syntax for the **totalview** command	"TotalView Command Syntax" in the *Classic TotalView Reference Guide*
Compiling your program for debugging	"Compiling Programs" in the *Classic TotalView User Guide*
Loading a program into TotalView using either the GUI or the CLI	"Loading Programs from the Sessions Manager" in the *Classic TotalView User Guide*
Attaching an existing process	"Attaching to a Running Program" in the *Classic TotalView User Guide*
Debugging a core file	"Debugging a Core File" in the *Classic TotalView User Guide*
Starting a parallel debugging job	"Starting MPI Programs Using File > Debug New Parallel Program" in the *Classic TotalView User Guide*
Debugging a replay recording session file	"Debugging a Replay Recording Session" in the *Classic TotalView User Guide*

Getting Around TotalView

Once you've started TotalView and loaded a program to debug, its two primary windows launch, the Root Window and the Process Window. These windows are the heart of TotalView and provide access to all its other windows and features, such as the Variable and Array windows, among many others. Each window contains a set of menus that provide a wide range of commands specific to that window.

This section does not introduce all of TotalView's windows or commands, but you will find them as you explore the product and debug your programs.

TotalView Commands

Each window has a set of menus, such as **File**, **View**, or **Tools**, with different commands depending on the window. For instance, the Process Window contains a **Tools > Call Graph** command to view the call graph, while the Variable Window's Tools menu provides a **Tools > Visualizer** command to view array data graphically.

These GUI commands are often equivalent to commands also available through the CLI. For instance, to save a set of breakpoints to a file, use the Process Window's **Action Point > Save As** GUI command, or the CLI command **dactions** -*save filename*.

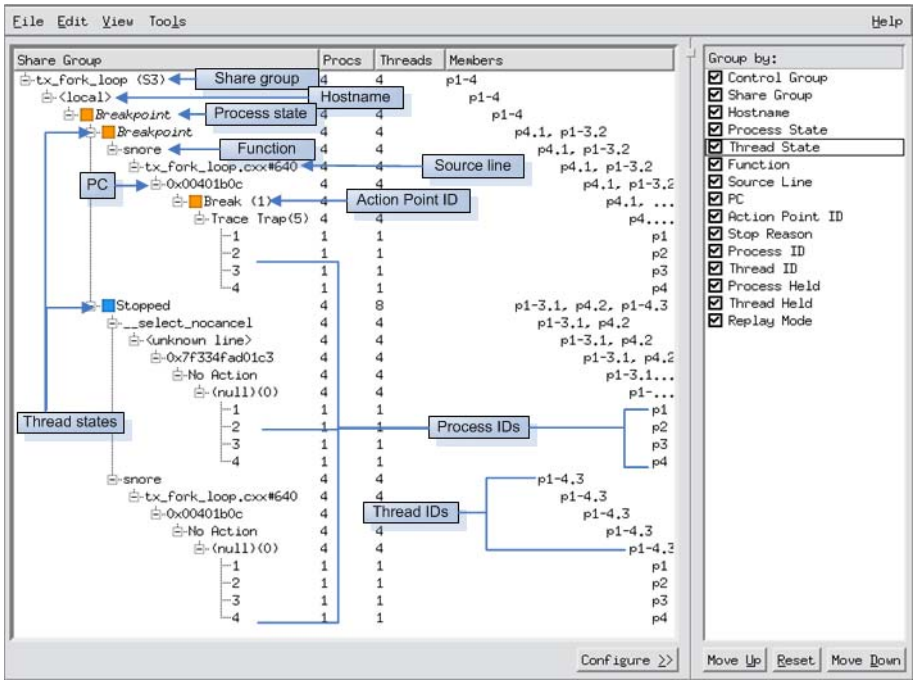
RELATED TOPICS	
GUI commands	From within TotalView, click the **Help** button in any window to launch context-sensitive help on any command (also viewable as "In Product Help" in the html version of the documentation).
CLI commands	"CLI Commands" in the *Classic TotalView Reference Guide*

The Root Window

The Root Window launches at TotalView startup, and displays a list of all the programs, processes and threads under TotalView control.

Figure 5 shows the Root Window for an executing multi-threaded multi-process program.

Figure 5 Root Window



This window displays status information and can be used to navigate to any of your programs, processes, or threads by diving on them. Diving simply means clicking on an object to launch a window with detailed information.

NOTE: Diving is central in TotalView and provides instant access to a wide range of detailed data by drilling down into the object, routine, process, thread, etc.

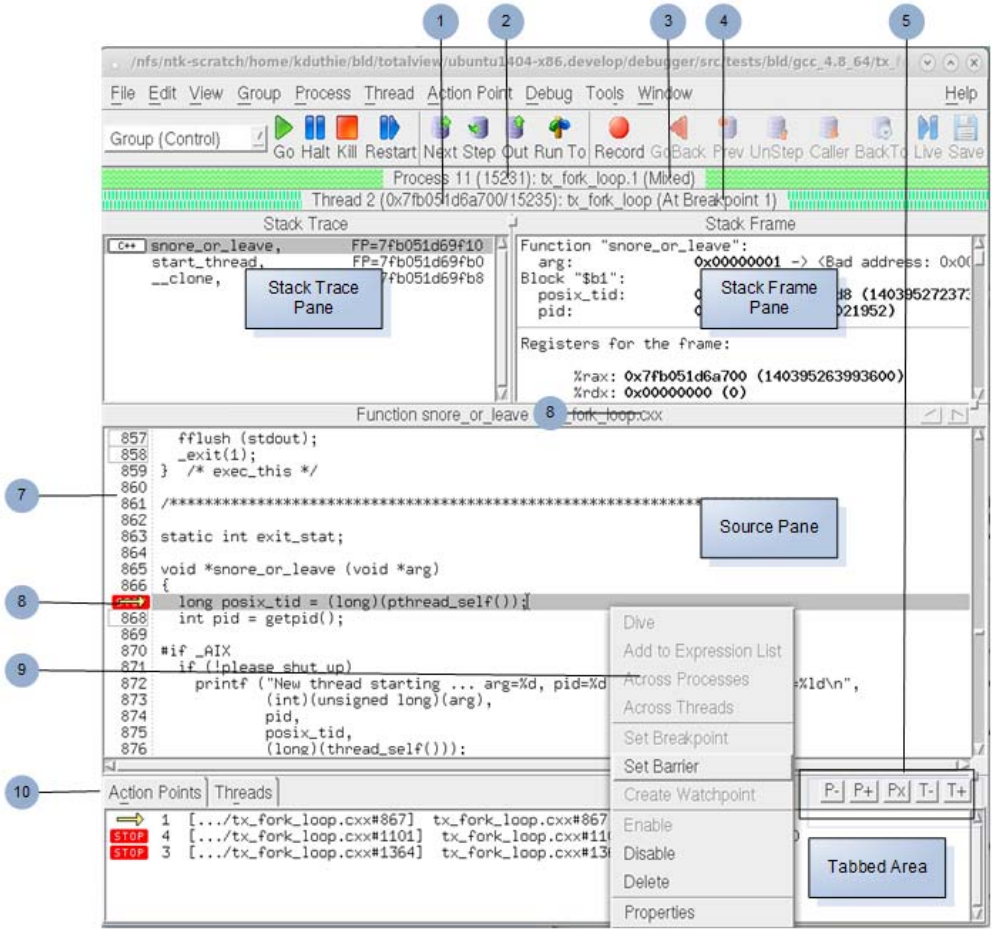
RELATED TOPICS	
General GUI features available on most windows	"Using and Customizing the TotalView GUI" in the *Classic TotalView User Guide*
The Root Window	"Using the Root Window" in the *Classic TotalView User Guide*

The Process Window

When you load any program or process into TotalView, the Process Window launches, displaying state about the current program or process and its threads. It includes some of the typical menu items of any GUI application (such as File, Edit, and View) and provides access to most of TotalView's features.

It is here that you set breakpoints, step through your program, and manage its threads.

Figure 6 The 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

The Process Window is divided into four areas:

- **Stack Trace Pane**, displaying the call stack.

- **Stack Frame Pane**, displaying the current thread's variables.
- **Source Pane**, displaying your program's source code or assembly instructions. Note the context menu that becomes available when you select a line of code in your program.
- In the "tabbed area":
  - **Action Points Tab**, which displays a list of the thread's current action points (TotalView nomenclature for its powerful set of breakpoints).
  - **Processes/Ranks Tab**, displaying a grid of the processes or ranks within the current control group.
  - **Threads Tab**, with a list of all active threads in the process.

RELATED TOPICS	
The Process Window in general	"Using the Process Window" in the *Classic TotalView User Guide*

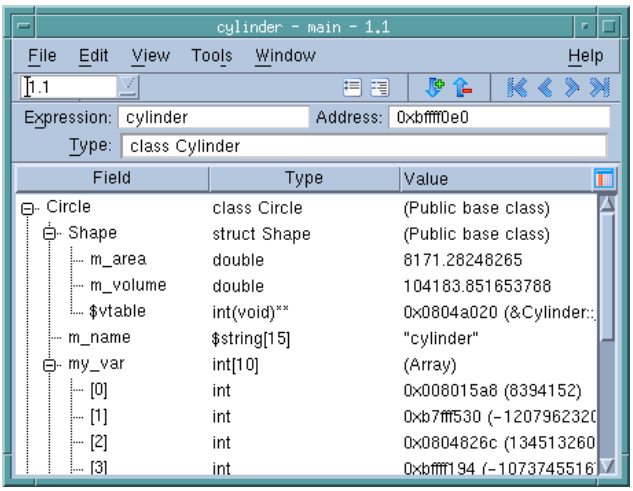
Variable Window and Array Viewer

TotalView provides multiple ways to see and edit your data, The primary window for working with data is the Variable Window. If your variable is an array, you can use the Array Viewer.

The Variable Window

The Variable Window displays details about your variables. To launch the Variable Window, Figure 7, just dive (by double-clicking) on a local variable (displayed in the Process Window's Stack Frame Pane), or on a global variable (in the Source Pane).

Figure 7 Variable Window

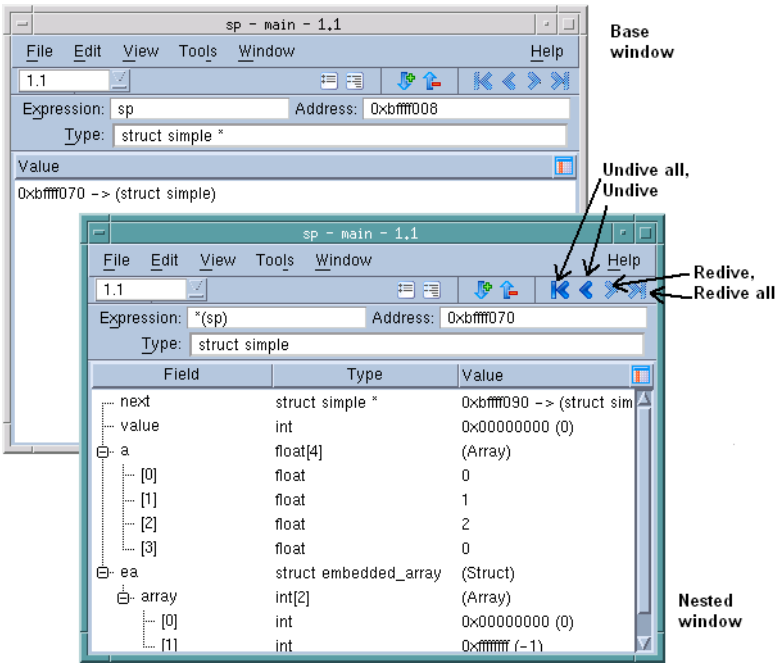


The Variable Window is a powerful tool for analyzing your program's data. You can control the display and cast your data in the Expression field, change the variable's address in the Address field, or change the data type in the Type field. For parallel programs, you can view or update the value of a variable in all of the processes or threads at once or individually. For variables that contain substructures, use the "**+**" or "**–**" icons to view them.

If the displayed variable is a pointer, structure, or array, you can dive on the value. This new dive, called a nested dive, replaces the information in the Variable Window with information about the selected variable.

Figure 8 shows a Variable Window before and after diving into a pointer variable **sp** with a type of **simple***. The *base window* displays the value of **sp** while the nested dive window shows the structure referenced by the **simple*** pointer.

Figure 8 Nested Dives



Use the undive/redive buttons to move between nested windows and the base window:

- To undive from a nested dive, click the undive arrow button so that the base window's contents appear. To undive from all dive operations, click the **Undive All** button.
- To redive after undiving, click the redive arrow button. To redive from all your undive operations, click on the **Redive All** arrow button.
- To retain access to a nested or base window so that both are visible, select the **Window > Duplicate** command to duplicate the current Variable Window.

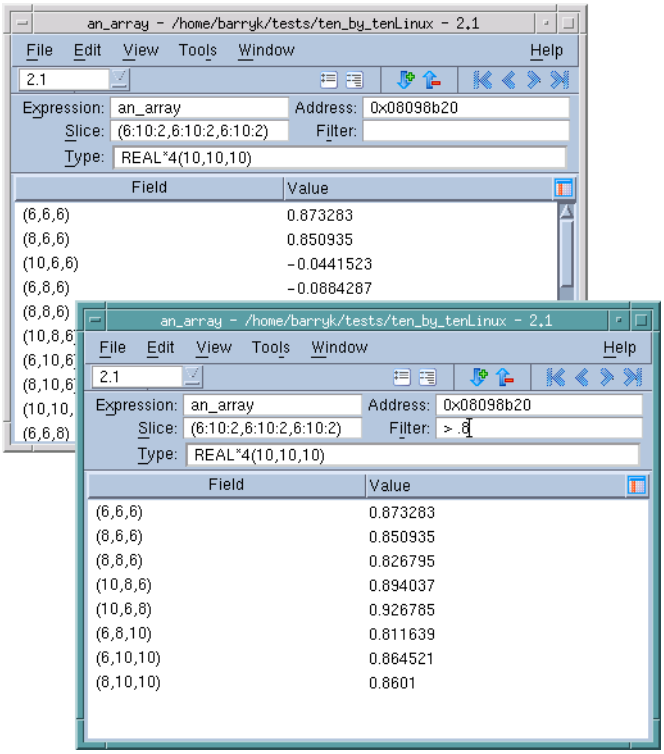
Array Variables:

For an array, the Variable Window includes a **Slice** field that shows each of the array's dimensions as a colon. You can display a section by editing the array specifier. Using the **Slice** field lets you focus on a subset of the data. For example, to display items 101 through 125 of a one-dimensional Fortran array, change the **Slice** field to **(101:125)**.

You can also enter an expression in the **Filter** field to limit the display. For example, if you're looking for values greater than 300, type "**> 300**".

In Figure 9, the top window uses a slice to limit the amount of information displayed in a three-dimensional array, while the bottom window combines a filter with a slice.

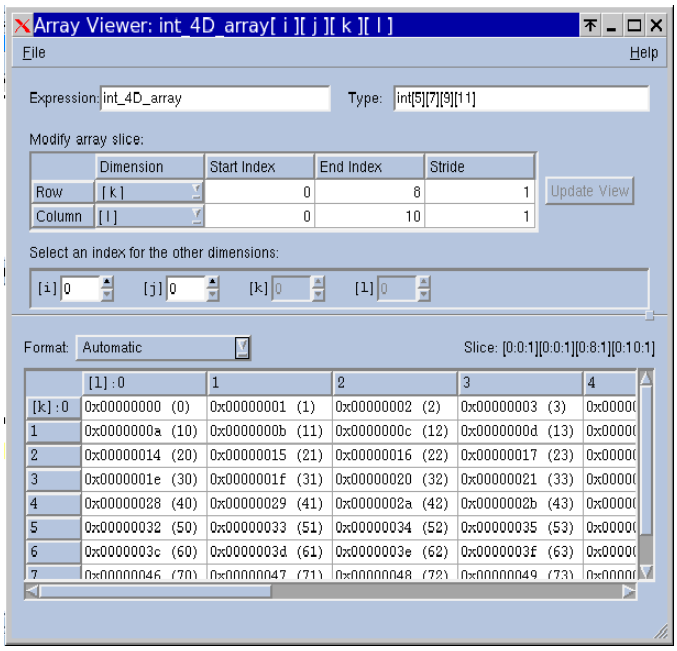
Figure 9 Sliced and Filtered Arrays



The Array Viewer

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 10.

Figure 10 Array Viewer



When the Array Viewer opens, the initial slice of displayed data depends on the values entered into the Variable Window. You can change the displayed data by modifying the Expression, Type, or Slice controls. For example, you can cast the array to another array expression, modify the type to cast the array to a different array type, or control how the slice is viewed.

RELATED TOPICS	
The Variable Window	"Displaying Variables" in the *Classic TotalView User Guide*
	"Diving in Variable Windows" in the *Classic TotalView User Guide*
	"Changing How Data is Displayed" in the *Classic TotalView User Guide*
Arrays	"Examining Arrays" in the *Classic TotalView User Guide*

Accessing TotalView Remotely

Using the Remote Display Client, you can start and run both TotalView and MemoryScape on a remote machine so you do not need to have them installed on your own machine. A licensed copy of TotalView must be installed on the remote machine, but you do not need an additional license to run the Client.

Platforms on which you can run a Client include:

- Linux x86 (32-bit) and Linux x86-64
- Microsoft Windows 7, Vista, and XP
- Apple Mac OS X Intel

Clients for all supported systems are available for download on the TotalView web site's Remote Display Client page.

RELATED TOPICS	
Remote Display Client	"Accessing TotalView Remotely" in the *Classic TotalView User Guide*

Debugging on a Remote Host

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

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

Setting Breakpoints and Stepping through a Program

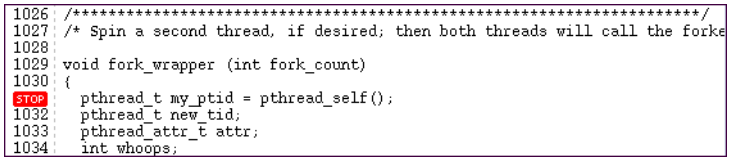
Action Points (breakpoints)

An action point is TotalView's much more powerful version of a breakpoint. Here are the four types:

- Breakpoint - stops execution of the processes or threads that reach it.
- 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.
- Evaluation Point - executes a code fragment when it is reached. Enables you to set "conditional breakpoints" and perform conditional execution.
- Watchpoint - monitors a location in memory and either stops execution or evaluates an expression when the value stored in memory is modified.

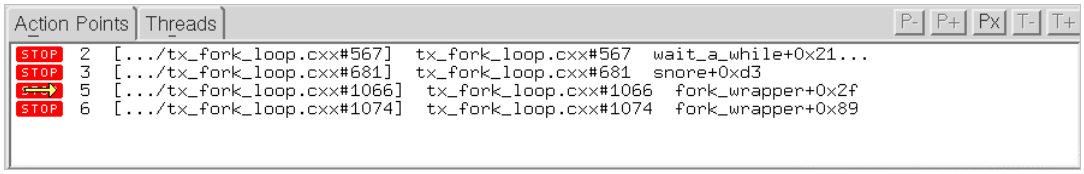
Set action points in the Process Window with a single left-click on the line number. TotalView displays a STOP sign.

Figure 11 Breakpoint Set At a Line



View all action points in the Process Window's Action Points tab.

Figure 12 Action Points Tab



When your program halts on an action point, TotalView reports this status in various ways, including in the Root Window, the Process Window's Source Pane, and through a yellow arrow (see above figure) on the Action Points tab.

Once you have created an action point, you can save, reload, suppress, and redefine its characteristics in a number of ways. You can set action points on all functions within a class or on a virtual function, and finely control how action points work in multi-threaded multi process programs.

RELATED TOPICS	
How action points work	"About Action Points" in the *Classic TotalView User Guide*
Setting action points	"Setting Action Points" in the *Classic TotalView User Guide*
The CLI command **dactions** to display, save, and reload action points	**dactions** in the *Classic TotalView Reference Guide*
The role of barrier points in multi-threaded processes	"Using Barrier Points" in the *Classic TotalView User Guide*

Stepping Through a Program

To start and step through your program, the easiest way is to use the buttons on the Process Window's toolbar:

Figure 13 Process Window Toolbar

- To start and stop your program:
  - Set a breakpoint, then select **Go** in the toolbar. Your program starts executing. Execution stops just before the line that contains a breakpoint or when you click **Halt**.
  - Select **Next**. TotalView starts your program, and then stops it immediately before the first statement in your **main()** function.
- To stop a running program, select the toolbar's **Halt** button. To restart a program, select the toolbar's **Restart** button.
- To step through your program, use the **Step** and **Next** buttons. Both tell your program to execute the current line, but when a line has a function call
  - **Step** goes *into* the function
  - **Next** completely *executes* the function

If you want to get to a line without individually stepping each line in between, select the line (not the line number) to highlight it, then click the **Run To** button. Alternatively you can use the **dskip** command to define rules to skip over or through specific functions or files. You can add rules that match a function, all functions in a source file, or a specific function in a specific source file.

- To step out of a function:

  If you stepped into a function and want to pop out to the statement that called it, click the **Out** button.

RELATED TOPICS	
Basic stepping commands	"Using Stepping Commands" in the *Classic TotalView User Guide*
Stepping in multi-process or multi-threaded programs	"Stepping Part I" and "Part II" in the *Classic TotalView User Guide*
Using CLI commands to step	"Execution Control Commands" in the *TotalView Reference Guide*

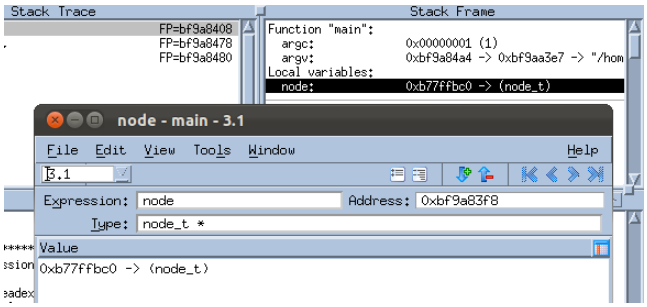
Examining and Editing Data

Diving and Viewing Data

Diving is integral to the TotalView GUI and provides a quick, intuitive, and effective way to get more information about various program elements. Diving is usually performed by just double-clicking on an element and generally launches a window with more information. You can dive on variables of course, but also on processes and threads, the call stack, functions, and source code.

To dive on a variable, just double-click on it or highlight it and select **View > Dive** to launch a Variable Window, Figure 14.

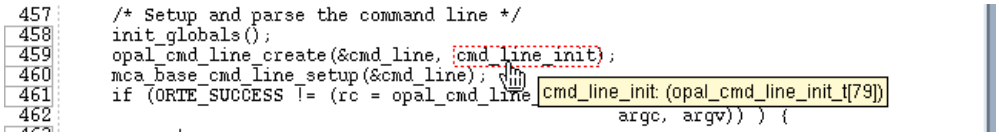
Figure 14 Diving on a variable in the Stack Frame



Local variables are visible in the Stack Frame as in Figure 14, while global variables are available in the Source Pane.

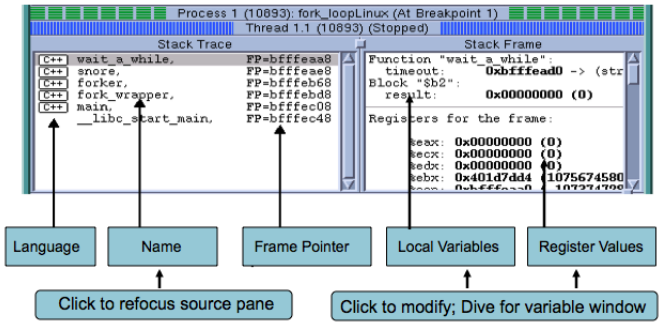
In the Source Pane, if a global variable or function can be dived on, a red dotted box appears when your cursor hovers over it, Figure 15.

Figure 15 Diving on an object in the Source Pane



For example, the Stack Frame Pane in the Process Window contains the current call stack. When you dive on a routine, TotalView shows the routine in the Source Pane and its variables in the Stack Frame Pane.

Figure 16 Diving on a routine



TotalView provides several other ways to see more detail about any aspect of your program:

- Displaying the call graph:

  Use **Tools > Call Graph** to launch a dynamic diagram that shows all the currently active routines. Click Update to recreate this display in a running program

- Viewing the state of every process and thread:

  Use **Tools > Parallel Backtrace View** to view the status of thousands of processes from a single window.

- Viewing your array data graphically:

  Use the Variable Window's **Tools > Visualize** to view array data as a graph or in the Visualizer, a versatile, stand-alone program that can be launched directly from within TotalView or separately via the command line.

RELATED TOPICS	
All objects you can dive on	"Diving into Objects" in the *Classic TotalView User Guide*
Diving in a Variable Window	"Diving in Variable Windows" in the *Classic TotalView User Guide*
The **View > Dive In All** command	"Displaying an Array of Structure's Elements" in the *Classic TotalView User Guide*
Displaying your call graph	"Displaying Call Graphs" in the *Classic TotalView User Guide*
Displaying STL variables	"Displaying STL Variables" in the *Classic TotalView User Guide*
Displaying assembler code	"Viewing the Assembler Version of Your Code" in the *Classic TotalView User Guide*
Viewing processes and threads	"Displaying a Variable in all Processes or Threads" in the *Classic TotalView User Guide*

Editing Data

You can edit a wide range of data while debugging your programs, such as variable type, value, and address, as well as source code. For instance, use the **File > Edit Source** command to examine the current routine in a text editor.

NOTE: If you edit source code while testing, be aware that these changes are within TotalView only and are not persisted to your actual files.

RELATED TOPICS	
General editing capabilities in dialog boxes	"Editing Text" in the *Classic TotalView User Guide*
Changing a variable's value or data type	"Changing a Variable's Data Type" in the *Classic TotalView User Guide*

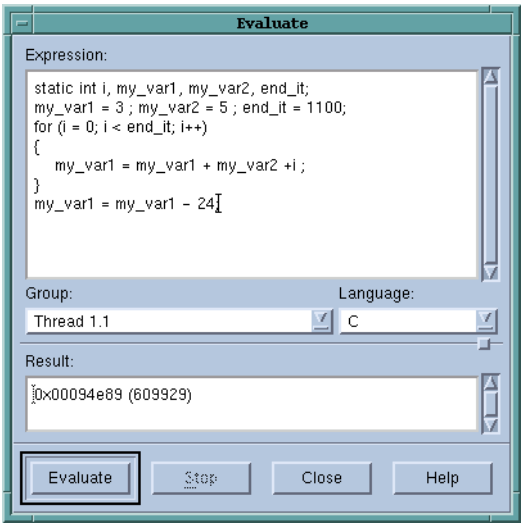
Evaluating Expressions

Expressions are used throughout TotalView. For instance. when you add code to an evaluation point (an action point that executes a code fragment), you enter it into the Expression Window. Your program's expressions are listed in the Expression List Window where you can add, edit, and control expressions.

Use the Evaluate Window (**Tools > Evaluate**) to evaluate expressions in the context of a particular process, in C, Fortran, or assembler.

Figure 17 shows a sample expression in an Evaluate Window. Note that C has been selected for the language, and the expression simply assigns the value of **my_var1-3** back to **my_var1**.

Figure 17 Tools > Evaluate Dialog Box



RELATED TOPICS

Interpreted and compiled expressions	"About Interpreted and Compiled Expressions" in the *Classic TotalView User Guide*
Evaluating expressions	"Evaluating Expressions" in the *Classic TotalView User Guide*
The Expression List Window	"Entering Expressions into the Expression Column" in the *Classic TotalView User Guide*
Operations you can perform in the Expression List Window	"Sorting, Reordering, and Editing" in the *TotalView User Guide*

Working with Multi-Processes and Multi-Threads

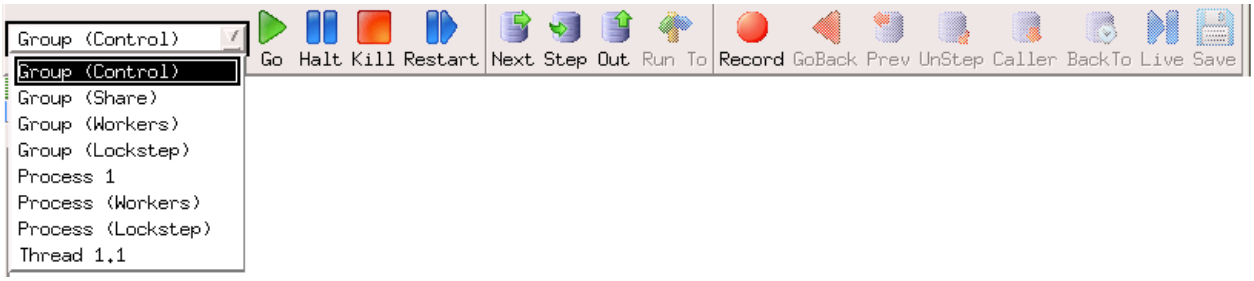
TotalView's real strength is in debugging multi-process, multi-threaded programs, many of which are tremendously complex.

Here's a brief rundown of TotalView's primary features that support this kind of complicated parallel computing:

- Organizing your processes and threads into groups, making it possible to debug programs running thousands of processes and threads across hundreds of computers.
- Placing 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.
- Automatically bringing any threads or processes spawned by your program under TotalView's control, avoiding the need to run multiple debuggers.
- Allowing you to focus on, run, set breakpoints on, and display individual processes, threads, or groups.

For example, to act on a particular process, select it from the toolbar's target pulldown menu, Figure 18. This defines the focus, so when you select the command **Go** or **Step**, TotalView knows what to act on.

Figure 18 Selecting a Target from the Toolbar Pulldown



RELATED TOPICS

Threading and multi-process applications in general	"Debugging Multi-process and Multi-threaded Programs" in the *Classic TotalView User Guide*

RELATED TOPICS	
How TotalView organizes processes and threads into groups	"About Threads, Processes, and Groups" in the *Classic TotalView User Guide*
Tips on parallel debugging	"Debugging Strategies for Parallel Applications" in the *Classic TotalView User Guide*

Starting a Parallel Debugging Job

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

You can start a parallel debugging job either from the GUI or directly from a shell, depending on your environment, parallel program, and preferences.

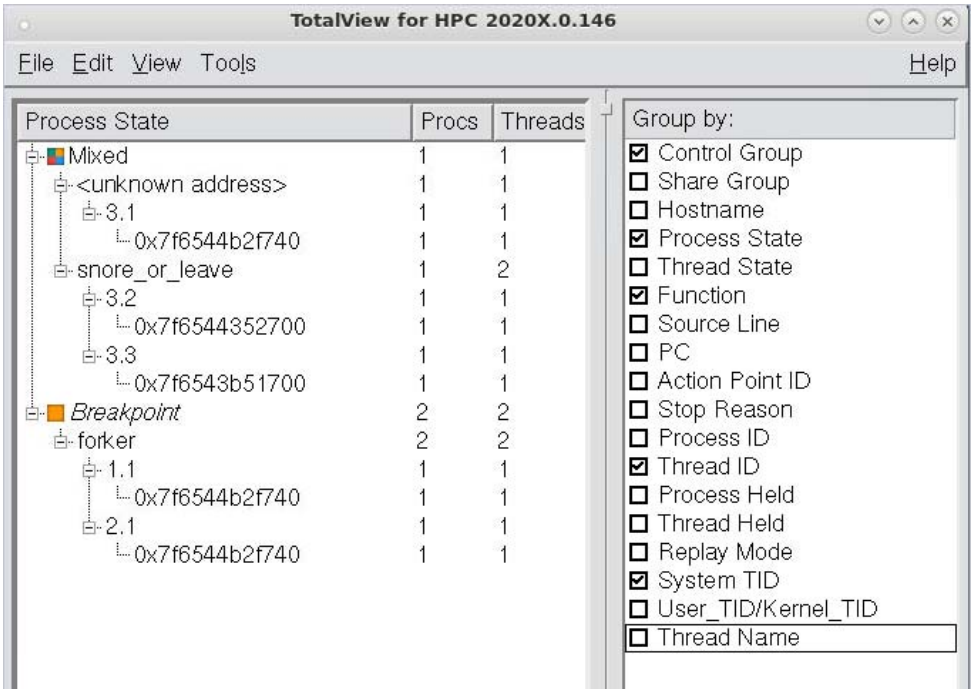
RELATED TOPICS	
Starting a parallel job in the GUI	"Starting MPI Programs Using File > Debug New Parallel Program" in the *Classic TotalView User Guide*
Starting a parallel job from the command line	One example is "Starting TotalView on an MPICH Job" in the *Classic TotalView User Guide*
Supported MPIs	*The TotalView Platforms Guide*

Working with and Viewing Processes and Threads

You can view the status of any or all your processes and threads in a variety of ways.

The Root Window contains an overview of all processes and threads being debugged. Just dive on a process or a thread for detailed information.

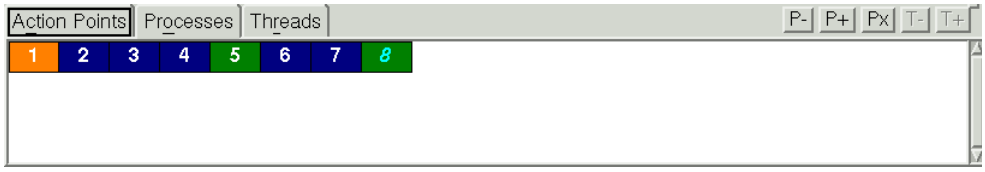
Figure 19 Processes and Threads in the Root Window



The Process Window's Processes tab (when the process grid is enabled) and Threads tab display information about all threads and processes, color-coded to define state.

Figure 20 shows a tab with processes in three different states:

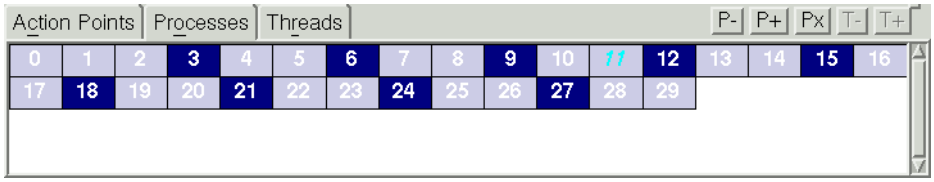
Figure 20 The Processes Tab



The orange cells represent processes that are at a breakpoint, blue is a stopped process (usually due to another process or thread hitting a breakpoint), and green denotes that all threads in the process are running or can run.

If you select a group using the Process Window's group selector pulldown menu, TotalView dims the blocks for processes not in the group, Figure 21.

Figure 21 The Processes Tab: Showing Group Selection



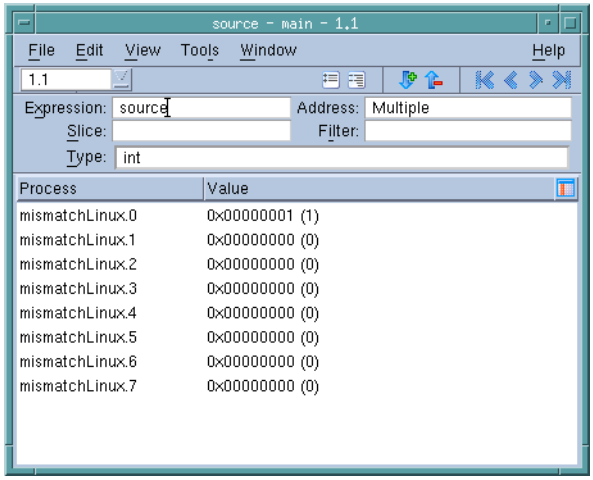
To switch between running processes, click on a box representing a process to switch to that context.

Similarly, clicking on a thread in the Threads tab changes the context to that thread.

Viewing the value of a variable in each process or thread:

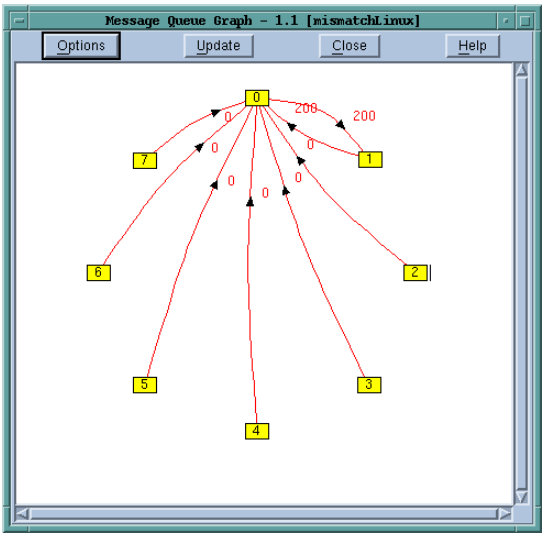
It is often useful to simultaneously see the value of a variable in each process or thread. Use **View > Show Across > Thread** or **View > Show Across > Process** to display the variable either across processes or threads, Figure 22.

Figure 22 Viewing Across Processes



If you are debugging an MPI program, use the **Tools > Message Queue Graph** Window graphically to display the program's message queues.

Figure 23 A Message Queue Graph



You can click on the boxed numbers to place the associated process into a Process Window, or click on a red number next to an arrow to display more information about that message queue.

RELATED TOPICS

Manipulating processes and threads in various ways	"Manipulating Processes and Threads" in the *Classic TotalView User Guide*

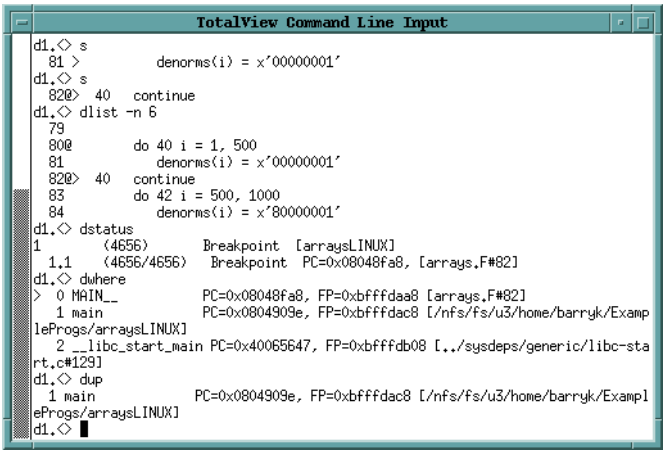
Debugging Using the Command Line Interface (CLI)

The Command Line Interface (CLI) is a command-line debugger integrated with TotalView. You can use it and never use the TotalView GUI, or you can use it and the GUI simultaneously, which is the assumed approach in much of the documentation.

The CLI is embedded in a Tcl interpreter, so you can also create debugging functions that exactly meet your needs. You can then use these functions in the same way you use TotalView's built-in CLI commands. 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.

Start the CLI from the GUI using **Tools > Command Line** in the Root or Process Windows, or directly from a shell prompt by typing **totalviewcli**. Figure 24 shows the CLI window debugging part of a program.

Figure 24 CLI xterm Window



In the *Classic TotalView User Guide*, CLI commands are frequently provided alongside GUI procedures, always within a gray box to be easily recognizable, for example:.

CLI: dactions -save *filename*

The command above saves your action points to a file, and is the equivalent of using the **Action Point > Save All** command. The *Classic TotalView Reference Guide* details all the CLI commands.

RELATED TOPICS	
Using the CLI	Part III, "Using the CLI" in the *Classic TotalView User Guide*
Details of CLI commands	"CLI Commands" in the *Classic TotalView Reference Guide*

Debugging CUDA Programs

The TotalView CUDA debugger is an integrated debugging tool capable of simultaneously debugging CUDA code that is running on the hosts 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
- All host debugging features are supported, except for ReplayEngine

RELATED TOPICS	
Using the CUDA debugger	"About the CUDA Debugger" in the *Classic TotalView User Guide*
The CLI **dcuda** command	**dcuda** in the *Classic TotalView Reference Guide*

Memory Debugging

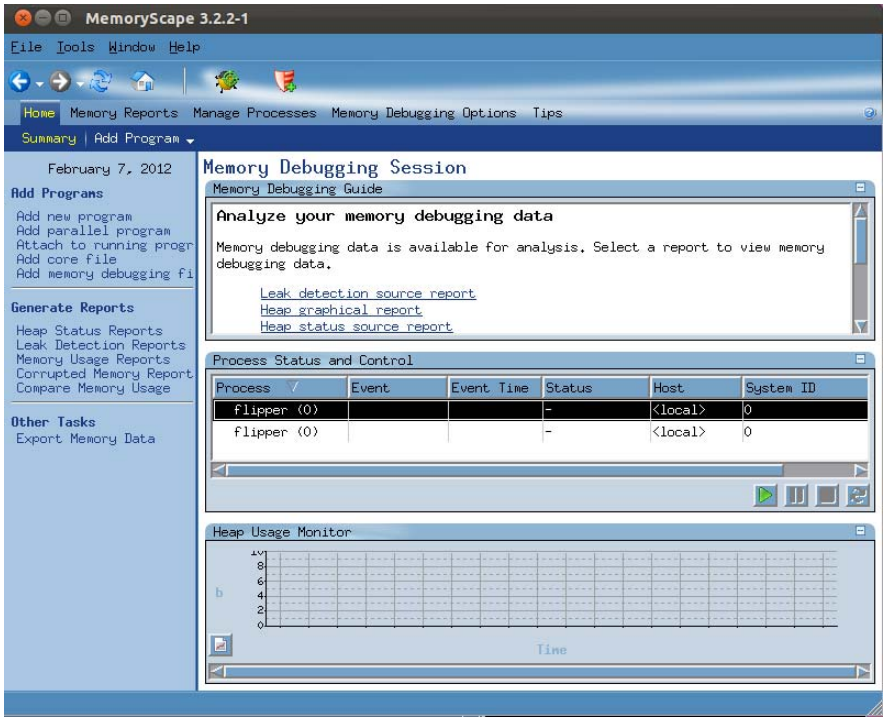
TotalView has a fully integrated version of the MemoryScape product for debugging memory issues. MemoryScape is also available as a standalone product.

MemoryScape can monitor how your program uses **malloc()** and **free()** and related functions such as **calloc()** and **realloc()**. You must enable memory debugging before you start running your program. Here are three ways to enable memory debugging:

- From the New Program Window, select **Enable Memory Debugging.**
- From the Process Window, select **Debug > Enable Memory Debugging.**
- On the command line, type **memscape** (which launches MemoryScape without TotalView)

Once you have loaded a program to debug in TotalView, select **Debug > Open MemoryScape** to launch the primary MemoryScape window.

Figure 25 MemoryScape home window



Because MemoryScape monitors calls to the malloc API, you can even debug programs that use their own memory management libraries. The only requirement is that these libraries eventually use the API. In most cases, you don't need to recompile or relink your program to use MemoryScape.

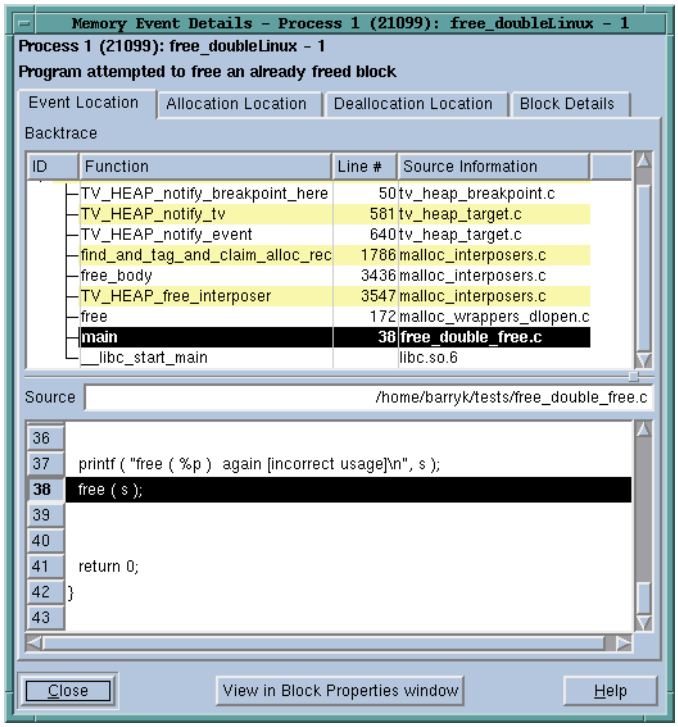
RELATED TOPICS	
Using MemoryScape	Use of MemoryScape is explained in a separate document, *Debugging Memory Problems with MemoryScape*, however this section introduces a few key features.

Viewing Memory Event Information

After you enable memory debugging, MemoryScape stops your program and raises an event flag if a memory problem occurs. If you are working within TotalView, TotalView also displays an event window, Figure 26. You can see the detailed information about the event either in the TotalView event window or by clicking on the MemoryScape event flag.

The details include the backtrace — that is, a list of stack frames — that existed when your program caused the memory error. Clicking on a stack frame shows the corresponding source code. The other tabs let you further explore where the memory block was allocated and deallocated. You can also see the contents of the block in the Block Details tab.

Figure 26 Memory Event Details



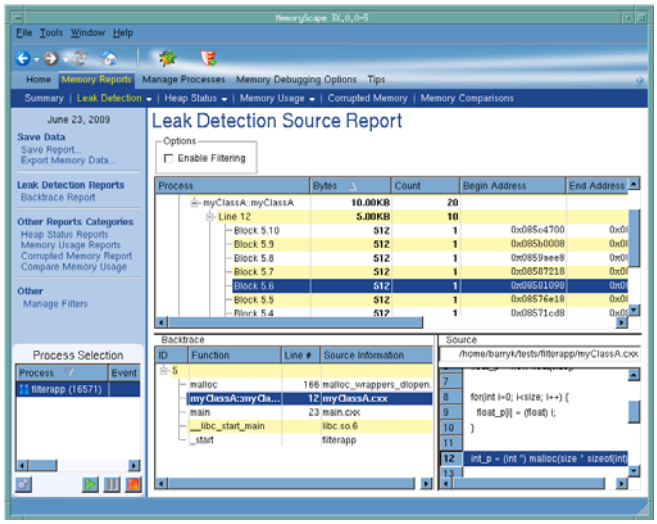
RELATED TOPICS	
MemoryScape error notification	"Event and Error Notification" in *Debugging Memory Problems with MemoryScape*
Halting execution at a memory error	"Halt Execution on Memory Event or Error" in *Debugging Memory Problems with MemoryScape*

Finding Memory Leaks

After you enable memory debugging, start your program. If you are working within TotalView, be sure to select **Debug > Open MemoryScape** to access MemoryScape's features.

Whenever you stop execution, you can ask for a report of your program's leaks.

Figure 27 Leak Detection Source Reports



When you click on a leak in the top part of the window, MemoryScape places the backtrace associated with the leak in the bottom part. When you click on a stack frame in this backtrace, MemoryScape displays the line within your program that allocated the memory.

RELATED TOPICS	
Finding memory leaks	"Finding Memory Leaks" in *Debugging Memory Problems with MemoryScape*
Memory leak reports	"MemoryScape Information" in *Debugging Memory Problems with MemoryScape*

Detecting Memory Corruption

You can detect memory block overrun and underrun errors with either guard blocks or Red Zones.

Guard Blocks:

Use guard blocks to detect writing beyond the limits of a memory block. To turn them on, either

- Select **Medium** from Basic Memory Debugging Options, or
- Select **Guard allocated memory** from Advanced Memory Debugging Options.

With guards on, MemoryScape adds a small segment of memory before and after each block that you allocate. You can find corrupted memory blocks in two ways:

- When the program frees the memory, the guards are checked for corruption. If a corrupted guard is found, MemoryScape stops program execution and raises an event flag. Click on the event flag to see the event details.
- Select Corrupted Memory Report from the Memory Reports page.

Red Zones:

Use Red Zones to find both read and write memory access violations, notifying you immediately if your program oversteps the bounds of your allocated block.

To turn them on, either

- Select **High** from Basic Memory Debugging Options, or
- Select **Use Red Zones to find memory access violations** from Advanced Memory Debugging Options.

With Red Zones on, a page of memory is placed either before or after your allocated block, and if your program tries to read or write in this zone, MemoryScape stops program execution and raises an event flag. Click on the event flag to see the event details.

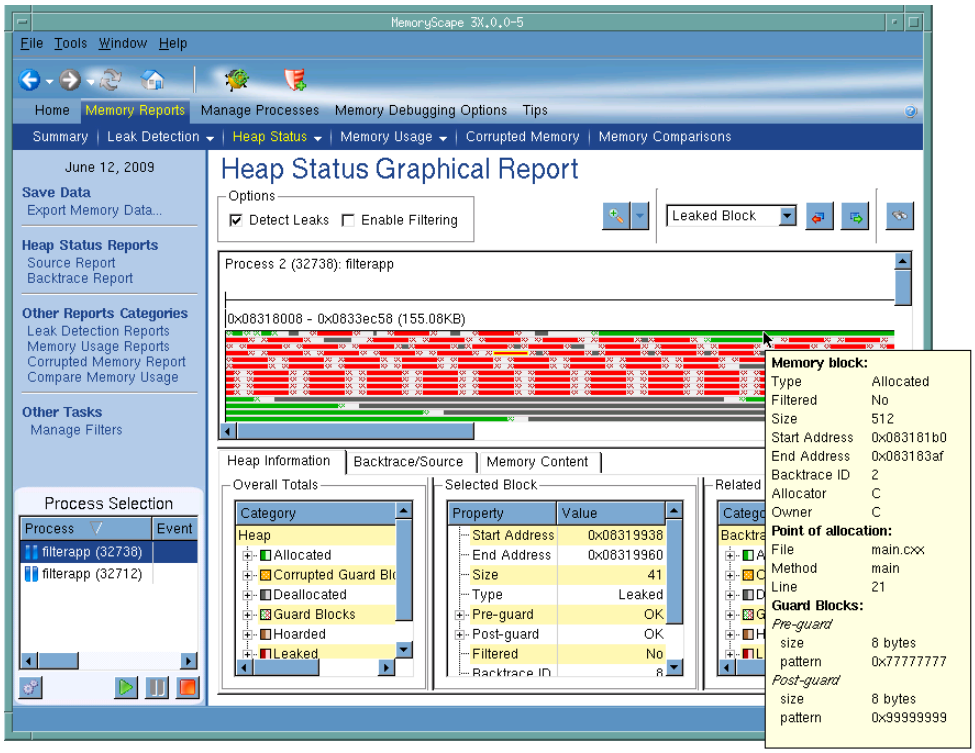
The default is to check for overruns, but you can check for underruns using Advanced Options controls.

RELATED TOPICS	
Locating corrupted memory	"Viewing Corrupted Memory" in *Debugging Memory Problems with MemoryScape*
Guard blocks and Red Zones	"Using Guard Blocks and Red Zones" in *Debugging Memory Problems with MemoryScape*

Analyzing Memory

To analyze how your program is using memory, select the Heap Graphical Report on the Memory Reports Page to see the memory your program is using, Figure 28.

Figure 28 Heap Status Graphical Report



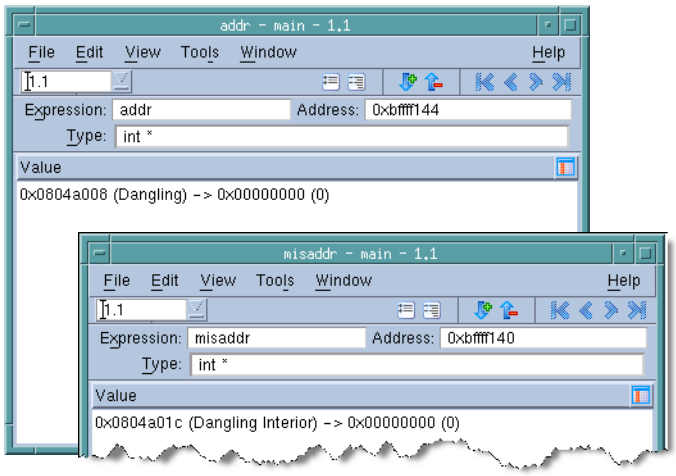
When you select a block in the top area, MemoryScape displays information about the selected block in the lower area. In addition, and perhaps more importantly, it displays how many other allocations are associated with the same backtrace and the amount of memory allocated from the same place. Other reports within the Heap Status Reports Page let you display the backtrace and source line associated with an allocation.

RELATED TOPICS	
The heap graphical report	"Graphically Viewing the Heap" in *Debugging Memory Problems with MemoryScape*

Finding Dangling Pointers

With memory debugging turned on in TotalView, you can identify a dangling pointer (points into deallocated memory) through additional information in the Variable Windows and the Stack Frame Pane, Figure 29.

Figure 29 Dangling Pointers

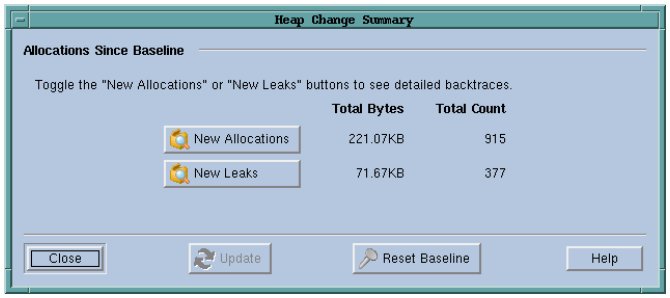


RELATED TOPICS	
Fixing dangling pointers	"Fixing Dangling Pointer Problems" in *Debugging Memory Problems with MemoryScape*
Using the command **dheap -is_dangling**	"Checking for Dangling Pointers: "**dheap -is_dangling**" in *Debugging Memory Problems with MemoryScape*

Setting and Using Baselines

Use the **Debug > Heap Baseline > Set Heap Baseline** command in the Process Window to have MemoryScape mark the current memory state. After your program has been executing, use the **Debug > Heap Baseline > Heap Change Summary** command to see what has happened to memory since you created the baseline.

Figure 30 Heap Change Summary Window



Pressing the **New Allocations** or **New Leaks** button displays more information.

Some reports within MemoryScape also have **Relative to baseline** buttons that allow you to limit the display to allocations and leaks occurring only since you set the baseline.

RELATED TOPICS	
Setting a heap baseline	"Debug > Heap Baseline >Set Heap Baseline," available directly from the **Help** button within the TotalView interface, and also provided in the shipped documentation "In-Product Help."

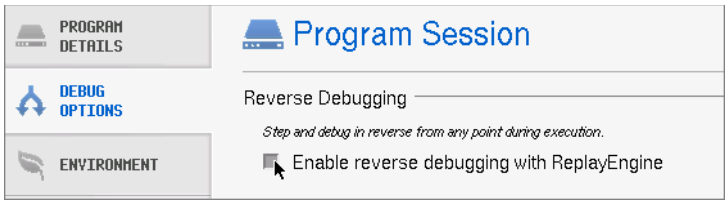
Reverse Debugging with ReplayEngine

ReplayEngine is a separately licensed product for Linux-x86 (32-bit) and Linux-x86-64 that records all your program's activities as it executes within TotalView. After recording information, you can move forward and backward within these previously executed instructions.

To enable ReplayEngine, select **Enable reverse debugging with Replay Engine** on the **Debug Options** tab after choosing either:

- **File > Debug New Program** to launch the Program Session dialog
- **File > Debug New Parallel Program** to launch the Parallel Program Session dialog
- **File > Attach to Running Program** to launch the Attach to a Running Program dialog

Figure 31 Enabling using File > Debug New Program



For a new program, ReplayEngine begins recording instructions as soon as you start program execution. For a running process you have attached to, ReplayEngine starts recording the next time you restart the process.

You can also enable ReplayEngine by selecting the **Record** button in the Process window's toolbar or by using the TotalView **-replay** command-line option:

```
dload -replay program-path
dattach -replay program-path
```

The ReplayEngine commands are on the toolbar, Figure 32

Figure 32 Tool Bar with ReplayEngine Buttons

When replaying instructions, your program's state is displayed as it was when that instruction was executed. The displayed information is read-only. For example, you cannot change the value of variables.

Existing execution commands work when replaying instructions. For example, you can use the **Step** or **Out** commands to move forward in the program's history.

Only when you reach the statement that would have executed outside of "replay mode" is the program put back into "record mode." For example, suppose you are at line 100 and you select line 25 and press the **BackTo** button. If you use commands that move forward in replay mode such as **Step**, you will switch from replay mode to record mode when get you back to line 100.

Because you can see previously executed instructions, you can quickly locate where a problem began to occur.

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