TotalView Brochure

Comprehensive tool for verifying, debugging, and optimizing complex applications

A White Paper by Rogue Wave Software.

Rogue Wave Software
5500 Flatiron Parkway,
Suite 200
Boulder, CO 80301, USA
www.rougewave.com

TotalView Brochure

Comprehensive tool for verifying, debugging, and optimizing complex applications

by Rogue Wave Software

Original Publication – April 2012

Printed in the United States of America

TABLE OF CONTENTS

TotalView

TotalView is a source code debugger for scientists and engineers troubleshooting complex, multi-threaded, or multi-process programs. It simplifies and shortens the troubleshooting process necessary to understand bugs and ultimately resolves defects in desktop applications, programs running on servers, and scientific simulations running on clusters. Offering extreme scalability, TotalView features memory debugging, programmability, powerful visualization and analytical capabilities, and support for a wide range of platforms.

TotalView can be used on x86 and x86-64 Linux laptops, workstations, or servers. It can also be used on Apple Mac laptops, workstations or servers as well as AIX, HP-UX and Oracle Solaris servers. In addition to workstations and servers it also supports a wide variety of computational clusters. These include clusters that combine x86 or x86-64 Linux systems with interconnects like Ethernet, gigabit Ethernet, or Infiniband. TotalView also works on specialized HPC systems such as the Cray X2, XT and XE lines, IBM Blue Gene/L, Blue Gene/P, and NEC SX systems. It supports computational accelerators like the Cell and the NVIDIA Tesla and Fermi.

TotalView supports debugging applications written in C, C++, Fortran 77, Fortran 90, and UPC, and is compatible with a number of different compiler families including the GCC compiler collection, Pathscale compilers, PGI C/C++ and Fortran compilers, and the Intel compiler family. It also supports applications that make use of MPI, UPC, OpenMP, and hybrid MPI/OpenMP architectures.

High Productivity

TotalView is first and foremost a graphical debugger. It presents the program, its current data and state of execution, and a series of simple-to-use buttons and menu items.

Your code is represented front and center in the TotalView process window, with the function call stack and local variables displayed in nearby areas of the window. A small set of important operations like stepping are done via prominently placed buttons; more rarely used features are available through the menus. Looking at a function is as simple as clicking on that function name in the function call stack pane. You can set a breakpoint with a single click on the relevant line number. The windows representing code and data are all highly interactive. You can use the mouse to dive on any function, variable, pointer, element, thread, process, or breakpoint for more detail. Forward and back navigation (similar to a web browser interface) is available whenever you are diving on a function or variable.

Between the basic interface and the dive concept, the learning curve for using TotalView's more sophisticated capabilities is quite gentle. Many debugging sessions require only TotalView's easiest-to-locate features. However, when troubleshooting involves a more nuanced inspection of, or control over, an application, you can call on TotalView's rich set of advanced features.

Advanced Features

A smaller number of troubleshooting situations require the analysis of large and/or complex data sets. TotalView provides a great degree of flexibility in the representation of data; you can easily navigate complex structures that contain references and types whose interpretation depends on other parts of the program. Often you can recognize correct and incorrect patterns within a program when you see its data. You can call up a graphical representation of program data (surface plots of arrays), program state (program call graph),

communication state (MPI message queues), and memory management state (heap statistics and heap graphical display) to look for these patterns. TotalView can also help you flag erroneous patterns by providing statistical analysis of program data, detecting cycles within communication patterns, and highlighting leaked blocks of memory.

Sometimes the sheer volume of data can be overwhelming; TotalView provides coping mechanisms. It can display sliced and filtered portions of very large array type data sets; allow you to subset attach to very large parallel applications, or filter data displayed on the message queue and program call graphs; and it provides a Boolean filtering mechanism on heap memory data (see below). If the problem relates to the way data changes as the program executes, TotalView allows you to set watchpoints on any memory location. These watchpoints are somewhat like breakpoints that are triggered when the memory value at the location changes. As a more advanced technique, watchpoints can be set to trigger only on specific conditions. TotalView highlights recently changed values when displaying program data.

For "racy" or other more difficult bugs you may need to take a more active role in steering the application in order to reproduce an otherwise rare situation, input data or sequence of concurrent operations. In these cases, you can leverage an unparalleled degree of process control: inject data directly by adjusting the value of variables or registers; use scripting, memory painting and hoarding; call functions and C++ methods; reset the program counter; and patch the program on the fly without recompilation. All of these let you concentrate on the code, confident of your ability to reproduce even the most obscure scenario.

TotalView provides a fully programmable and scriptable command line interface, enabling you to systematically look at data and/or control program execution. Two of the unique features of the TotalView command line interface are that it remains asynchronous, and that individual commands can be directed at arbitrary sets of threads and/or processes. This means that you can request the value of a variable as it exists in hundreds or thousands of processes with a single command. The output of such a command can easily be captured, parsed, and analyzed in place in the debugging script. The TotalView command line interface is asynchronous in that one or more processes can be running while the interface is accepting input, and commands that continue one or more process return right away.

Remote Access

Many scientists and software developers who could benefit from TotalView are using unique computational resources that are centrally located at an HPC center, logging in over a long-distance network connection. They do most of their work through the command line and may or may not even know how to set up a graphical connection. TotalView includes a Remote Display Client that greatly simplifies setting up and accelerates the display of a full interactive graphical user interface over slow long-distance connections.

Batch Debugging

Another reality of the HPC work environment is the requirement to use a batch resource management system. In this case, users are often very comfortable using a command line interface to set up a job, submit it to the batch queue, and receive and analyze results generated by the job. TotalView includes TVScript for batch queue workflows and environments.

Parallel Debugging

TotalView provides a powerful environment for debugging parallel programs in that it allows you to easily control and inspect applications composed of either a single process, or sets of thousands of processes running across the many compute nodes of a supercomputer.

TotalView's user interface is built around the idea that you typically want to focus on a single process from among the many that make up a parallel application. At any point you may wish to work with this process alone, with this process and related other processes, or with all the processes that make up the job. Alternatively, you may want to switch your focus to another process entirely.

The root window provides one way to navigate the many processes that make up a job. It provides a list of all of the processes (and optionally some or all of the threads) to which the job or jobs may be attached. The list can be customized in several ways; the default identifies all the processes in MPI_COMM_WORLD rank order. For very large jobs, it is sometimes convenient to sort the list by current state (running, stopped, at a breakpoint, etc.). These states can be rolled-up for easier viewing.

TotalView also displays processes graphically on the processes tab of each process window, making it easier to understand and use process groups. Process groups can be used to query or control a program. A group is automatically defined for any set of processes that are "in lockstep" with the current process, making it easy to work with a set of closely related processes. Groups can also be defined based on other characteristics: MPI communicator membership, process state at a given time, or any other arbitrary grouping based on your knowledge of the program structure. Data inspection, evaluation and process control commands can operate on any of these groups. Breakpoints, watchpoints, and barrier objects can operate on single threads or processes, on groups of processes, or entire parallel applications.

Graphical representations of complex data are often easier for scientists and developers to review and analyze. When working with many processes, TotalView provides two important graphical representations: the call graph provides a compact view of the current point of execution for the many processes in a large parallel job; the message queue graph provides a compact view of message communication between processes in a large parallel job.

Finally, TotalView offers subset attach, a critical feature for work at extreme scales. TotalView does not need to be attached to an entire parallel job: it can attach to any arbitrary subset. Any processes that are not attached in the debugger will run freely and participate in the parallel program. This subset of attached processes can change over time as you explore your parallel application.

Advanced Memory Debugging

TotalView implements an integrated memory debugging feature called MemoryScape, to provide vital information about the state of program heap memory. It reports some errors directly as they occur, provides graphical and interactive maps of the heap memory within individual processes, and makes information such as the set of leaked blocks easy to obtain. Designed to be used with parallel applications, it provides detailed information about individual processes as well as high-level memory usage statistics across all the processes that make up large parallel applications.

MemoryScape is lightweight and has a very low runtime performance cost. It can be used separately from TotalView and is available for purchase separately.

Memory Debugging Architecture

MemoryScape uses a technique called interposition, inserting a library, the Heap Interposition Agent (HIA), between application code and the *malloc()* subsystem. This library defines functions for each of the memory allocation API functions, which are initially called by the program whenever it allocates, reallocates, or frees a block of memory.

Interposition differs from simply replacing the malloc library with a debug malloc in that the interposition library does not actually fulfill any of the operations itself; it arranges for the program's malloc API function calls to be forwarded to the underlying heap manager that would have been called in the absence of the HIA. The effect of interposing with the HIA is that the program behaves the same way as it would without the HIA, but the HIA intercepts all the memory calls and performs bookkeeping and sanity checks before and after the underlying function is called.

Memory Debugging Features

MemoryScape provides a range of heap status reports, the most popular of which is the heap graphical display. At any point where a process has been stopped, the heap graphical display paints a picture of the heap memory in the selected process. Each current heap memory allocation is represented by a green bar extending across the range of addresses that are part of the allocation. The view is interactive: selecting a block highlights related allocations and presents detailed information about the selected block and the full set of related blocks. The display can be filtered to dim allocations based on their properties (such as their size or what shared object they were allocated in).

Overall memory usage statistics are also provided in line, bar and pie charts for one, all, or an arbitrary subset of the processes that make up the debugging session.

Leak detection can be done at any point in program execution. Most leaks occur when the program ceases using a block of memory without calling free. Although it is difficult to define "ceasing to use," the debugger is able to detect leaks by checking whether the program retains a reference to specific memory locations. Leak detection on a process identifies the specific allocations for which there are no locatable references in the program. A block of memory for which the program stores no reference is highly unlikely to subsequently be subject to a free() call and is extremely likely to be a leak.

If you suspect a "wild pointer" or array bounds violation is corrupting memory, you can use heap guard blocks to track down the problem. The HIA can place a "reserved region" with known data values before and/or after individual heap memory allocations, and a change in these data values indicates that the program is writing where it should not be. The Red Zones feature, which protects a page of memory around the blocks that you want to check, directly detects writes into these protected pages.

For memory debugging of very large scale jobs, a reduced-overhead alternative can be used more naturally with batch environments. This mode, referred to as lightweight memory core file debugging, allows you to prepare your application and run it without the debugger attached. When memory events occur, or at the end of the job run, the HIA can generate a file of information about the error or the status of the heap. The file, like a traditional corefile, can be opened in TotalView to review the information.

Reverse Debugging

TotalView on Linux includes the ReplayEngine reverse debugging feature which radically simplifies troubleshooting and debugging by letting you move backward through code execution.

Architecture

The reverse debugging capability is predicated on the ability to record and deterministically replay the execution of each individual process in the program. The debugger arranges for each target process to load an instrumentation library that manages the process of recording and replaying execution history. For the most part, the debugger operates on each process as it would otherwise do to suspend execution, read and write registers and memory, set breakpoints, and resume the process or individual threads. The debugger also has the ability to instruct the process to "go to" any previous point in execution history. The instrumentation library transparently takes any action necessary to put the process into the right state so that the debugger can then query registers and memory from that point in recorded history.

Recording Program Execution

One of the challenges tackled by the instrumentation library is recording enough of the program execution to allow the program to be re-executed along exactly the same trajectory over and over again. This includes inputs such as data from external files, data transmitted over the network, the return value of system calls, and the sequence of thread execution. TotalView does this very efficiently both in terms of the time that it takes to record the information and in terms of the volume of data recorded.

Deterministic Replay

Given its ability to record program execution, the instrumentation library can reproduce any point on the trajectory of the program by starting from a previous known state and replaying the program forward in a controlled fashion. During this replay the simulated process reads data and makes system calls that all return the values that were recorded during the execution of the real process. The instrumentation library manages a collection of known states in such a way that most backwards stepping operations can be done with little delay. However, there may also be points in execution history that will take more than a few seconds to reach, especially in very long-running processes.

Implications for Debugging

The architecture mentioned here has some implications that are worth reviewing. The instrumentation library needs to make the execution trajectory deterministic in programs that have multiple threads per process. It does this by ensuring that only one thread at a time runs, which may significantly impact the performance of some programs.

Recording process behavior incurs some significant overhead, which will vary from process to process depending on individual characteristics. This overhead comes in the form of execution time and memory usage.

Under some circumstances, the process being debugged may need to write output to a file. During the replay of recorded execution history some of the operations that TotalView users are accustomed to using (such as setting program variables, "set pc," and full asynchronous thread control) may not be available because their use would cause the simulated execution to depart from the recorded trajectory.

Large computational cluster resources are almost always run in a mode where jobs are managed and accounted for using a resource management system. Looking at troubleshooting and debugging as a process, TotalView seeks to eliminate and shorten the process cycles by allowing developers to work backwards from failure to error. However, this improvement involves a trade off between many shorter runs and a single longer run. For reverse debugging to be adopted by users of large managed clusters, the rules governing access may need to allow for longer running single jobs in the portion of the machine set aside for interactive debugging.

Usage Models

There are at least four different operations that will be important when you are working backwards through your program.

Backwards Stepping

The simplest and most frequently used mechanism for inspecting the historical state of the program is to step backwards line-by-line through the program. This is an analog of stepping in the forward direction and the same navigation rules apply. If you are stepping through a loop construct, back stepping from the top line will take you to the conditional, and then back into the bottom of the loop.

As with forward stepping, there are several slight variations on the stepping theme for backwards stepping. The "back next" operation takes the target program backwards over a single executable line of code at the same scope level, even if it contains one or more function calls. The "back step" operation takes the target program backwards over a line of code -- unless that line of code contains one or more functions, in which case it takes the target program to the previously executed line of code even if that line occurs in one of the called subroutines. In either of the previous cases, backwards stepping from the first line in a function will take the program to the point in time before the function was called. When working in a function, "back out" will take the program to the line just before the current function was called.

Together, these operations give you easy access to the recent past of the program's execution. They can be coupled with forward stepping commands so you can step back and forth over a single line many times, if that is what is needed to understand what is going on. This should make unfamiliar programs much easier to work with since any nuances of the program's behavior that are missed at first can be picked up on further examination.

Reverse “Run To”

While you will often step line-by-line through interesting sections of programs, there are other sections of the program that are simply not relevant, so you may need a way to skip back over longer stretches of the execution history. This is provided by the "run back to"

operation that lets you examine the state of the program at the most recent time the program was at any given line. The selected line might be a line in the same function, a line in a calling function, a line in any other function that happened to have been called previously in the program, or a line in main. This is a direct analog of the TotalView "run to" command, which works the same way in the forward direction.

This ability is particularly effective when the function being examined contains one or more loop constructs, and instead of stepping around the loop many times, you can simply jump to before the loop started. You can also return to an earlier program phase by selecting lines in the top level program structure that represent these phases.

Backwards Continue

When you have several possible breakpoints set, the “continue” button runs the program forward to the next breakpoint (or watchpoint). “Backwards continue” provides the reverse analog of this operation. With it, the debugger notes the current point in execution history and searches back through recorded execution history in the context of all the currently active breakpoints and watchpoints to find the one that occurs closest to the current point. It is as if the program actually ran backwards and stopped at the first breakpoint or watchpoint that was triggered.

Using ‘backwards continue’ is a very useful strategy in cases where there is a gulf between the error and failure, marked by the presence of bad or unexpected data in a variable. The failure is the result of the program attempting to compute based on this bad data, whose source is often highly mysterious. The usual suspects include buffer overruns, racy writes, or writes to invalid pointers, and tracking these down without reverse debugging is a chore. With TotalView, working across this gulf is brilliantly easy: just set a watchpoint on the memory address that contains the bogus data and use “backwards continue.” Watchpoints trigger when the debugger detects changing data at that location, so the program will stop at the instruction before the one that set the bogus value. You can step forwards and backwards and watch it go from a good value to a bad one, and more importantly look at the context of the change.

Random Access

In some cases there is no specific line defining the point in the execution history you want to examine. TotalView gives you the ability to specify a time and ask to inspect the program at that point. This is the backwards analog to halting a program after a specified time has passed.

Since TotalView provides both a Graphical User Interface (GUI) and a Command Line Interface (CLI), reverse debugging can be driven by either interface.

Graphical User Interface

Most users use the GUI. You enable reverse debugging on a specific process by selecting a checkbox in the TotalView new program dialog box. No further action is required – the debugger takes care of starting the program in such a way that the instrumentation library is loaded.

The TotalView process window contains a series of prominently placed buttons for the stepping commands. The backwards stepping commands are next to the forward stepping commands. Since they are such close analogs of the forward stepping commands, you can figure them out with just a little experimentation. The "run back to" command, like the forward debugging "run to" command requires you to select a line in the program before it is available.

How do you return to record mode after examining the historical state? You can either forward step out of history or use the "return to live" button on the process window toolbar.

Command Line Interface

The Command Line Interface (CLI) for TotalView, best used to script, extend or programmatically drive the debugger through a precise series of operations, has been extended to include reverse debugging commands. For example, the dstep command includes a –back flag that allows it to be used in the reverse direction.

The direct access to history, described above, is currently available only in the CLI. In that context a query command provides a numeric "time-like" measure of the program's location along the process's execution trajectory. The "go to time X" command allows you to randomly access that execution history.

Debugging Heterogeneous Architectures

TotalView supports debugging applications utilizing NVIDIA GP-GPU accelerated CUDA and OpenACC, as well as Intel Xeon Phi coprocessors.

NVIDIA® GP-GPU Accelerated CUDA™ and OpenACC® Debugging

TotalView on Linux includes visibility into, and control over CUDA and OpenACC-based NVidia accelerator debugging. The graphical presentation makes it easier for you as you step through either host code or CUDA kernels to diagnose problems in how they are executing.

When debugging code that contains CUDA APIs, you can move between the host and device code in the same session. CUDA specific features supported in TotalView include the following:

- Linux and GPU device thread visibility
- Full visibility to the hierarchical device, block, and thread memory
- Navigating device threads by logical and device coordinates
- CUDA function calls, host pinned memory regions and CUDA contexts
- Handling CUDA functions inline and on the stack
- Command line interface (CLI) commands for CUDA functions
- Applications that use multiple NVIDIA devices at the same time
- MPI applications on CUDA-accelerated clusters
- Unified Virtual Addressing and GPUDirect
- CUDA C++ and inline PTX
- Reporting memory errors and handling CUDA exceptions

Another unique feature for understanding CUDA applications is the device status window which combines information about the available CUDA hardware and information about how the logical tasks that are part of the program are being mapped to that hardware.

Intel® Xeon Phi™ Debugging

The support in TotalView for the Intel Xeon Phi can be used to debug applications that are either compiled to run directly on the coprocessor or to run on the host while offloading specific tasks or computations to the coprocessor.

TotalView has the following Intel Xeon Phi debugging capabilities:

- Full asynchronous thread control on both the host and Intel Xeon Phi coprocessor
- Simultaneously view what is happening in both the host and offload processes
- Certain breakpoints are shared across the host and coprocessor code
- Support for clusters and multi-device configurations
- Support for launching MPI and hybrid MPI + OpenMP applications natively into one or many Intel Xeon Phi coprocessors
- Support for debugging native Intel Xeon Phi applications launched manually on the coprocessor
- Support for debugging host side applications using the Intel Language Extensions for Offloading (LEO)

Putting It Together

TotalView provides the most comprehensive tool available for verifying and debugging complex applications. It includes a unique combination of capabilities to pinpoint and fix hard to reproduce bugs, memory leaks and race conditions.

Developing parallel, data-intensive applications is hard. We make it easier.

About Rogue Wave Software

Rogue Wave Software, Inc. is the largest independent provider of cross-platform software development tools and embedded components for the next generation of HPC applications. Rogue Wave marries High Performance Computing with High Productivity Computing to enable developers to harness the power of parallel applications and multicore computing. Rogue Wave products reduce the complexity of prototyping, developing, debugging, and optimizing multi-processor and data-intensive applications. Rogue Wave customers are industry leaders in the Global 2000, ISVs, OEMs, government laboratories and research institutions that leverage computationally-complex and data-intensive applications to enable innovation and outperform competitors. For more information, visit http://www.roguewave.com.