# Debugging Scalable Applications on the XT

Chris Gottbrath *TotalView Technologies*

May 1, 2009

**Abstract**

Debugging at large scale on the Cray XT can involve a combination of interactive and non-interactive debugging; the paper will review subset attach and provide some recommendations for interactive debugging at large scale, and will introduce the TVScript feature of TotalView which provides for non-interactive debugging. Because many users of Cray XT systems are not physically co-located with the HPC centers on which they develop and run their applications, the paper will also cover the new TotalView Remote Display Client, which allows remote scientists and computer scientists to easily create a connection over which they can use TotalView interactively. The paper will conclude with a brief update on two topics presented at the previous two CUG meetings: memory debugging on the Cray XT and Record and Replay Debugging.

**Keywords:** Troubleshooting, Debugging, Scalable Debugging, Batch Debugging, Remote Debugging, Memory Debugging, Reverse Debugging

# 1 Introduction

## 1.1 The challenges of HPC debugging

Debugging isn't something that most scientists and developers find enjoyable. We generally feel better if, after a day's worth of work, we can say that we added some new functionality to our programs or were able to make progress towards using our codes to accomplish scientific or engineering goals. While it can occasionally be satisfying to put a particularly vexing bug to rest, it is hard not to find debugging an unpleasant though necessary stage in the development and maintenance of the software we develop and use.

Developers working in the domain of High Performance Computing (HPC) have it especially hard. They have some of the most complex applications and they run in some of the most challenging environments for debugging that one can find. Applications written for modern HPC systems such as the Cray XT are fundamentally concurrent and distributed because modern HPC systems are essentially distributed memory clusters with fast internode network connections and great parallel storage. The distributed nature of the computation means that the developers need to tackle numerous challenges beyond simply getting their application to run. Distributed data must be kept in sync without incurring unnecessary communication so applications typically have non-trivial communication back and forth between processes. Calculations on the entire "problem dataset" often proceed only as fast as the slowest node (an oversimplification, but generally true) so care has to be taken to either distribute work opportunistically or constantly monitor and manage the amount of work being performed on each node of the cluster – load balancing the system so that none of the nodes are left idle while other parts of the system catch up. As the "main loop" of the code is parallelized any remaining serial routines may come to dominate the runtime, an effect called Amdahl's law, so even seemingly simple tasks like loading and storing data may become the subject of scrutiny and require some kind of optimization. [1, 5]

The Message Passing Interface (MPI) library interface standard that is nearly ubiquitous in HPC is extremely powerful and flexible, giving developers access to multiple types of communication and a great degree of freedom to optimize. [6, 2] MPI developers have access to both blocking and non-blocking point-to-point communication, one-sided communication (though this capability is rarely used), and highly optimized collective operations. A wide range of applications using MPI have been able

to achieve levels of scalability that are quite remarkable, running efficiently across tens of thousands of processors in large terra- and peta-scale systems. However the richness of the interface means that it is possible for both straightforwad and very subtle defects to be introduced. These errors can lead to numerical errors, crashes, communication deadlocks, hangs, and race conditions. Sometimes these kinds of errors manifest at any scale, so the scientist or developer can troubleshoot the program at a small scale in a "corner" of the machine. More rarely the defect may only manifest when the program is run at a large scale across a significant portion of the machine. How does one perform effective troubleshooting and problem solving on applications that are spread across hundreds or thousands of processes?

As systems grow one of the current trends is that the amount of memory available per processor core is either not increasing or is actually decreasing. This puts additional pressure on developers and scientists who are developing new applications, maintaining and extending existing applications or porting applications to new architectures. This pressure is to make sure that as the program runs it does not get itself into a position where it makes a request for memory that exceeds the memory that is physically available on the system. This means that developers need to be very aware of any possible defects in memory management within their programs. Languages such as C, C++ and Fortran 90 place the management of certain types of memory explicitly in the hands of developers, making it possible for applications to lose track of, or leak, allocated memory. Even a slow leak can cause the program to gobble up precious memory and lead to a fault, often in an unrelated portion of the code later as the program executes. Even in programs that don't leak developers are under pressure to make sure that they understand memory usage patterns so that they know where they might look within their program to make optimizations and trade-offs that can make more efficient use of the limited memory available to them.

HPC systems are managed by the centers and organizations that run them to make sure that they aren't wasted and that they are available to their users. This typically means that they are made available to the user community via some form of batch queue-based job management framework. This is generally set up to allow a mix of different sized jobs to run in such a way that the utilization fraction of the machine is high and that users get fair access to the machine. No one who should have access gets blocked out of the system. Jobs that are submitted are generally non-interactive with a program file, some sort of configuration file, and then some set of input data files to process. One of the side effects of the fact that the system is based on a queue of work is that there is a delay between submitting a job and having resources allocated to run that job. This delay isn't that big a deal with a non-interactive job but can be frustrating when the developer needs to actually interact with the application.

HPC applications are very rarely, if ever, developed by a single individual. Instead they are the result of collaborative effort by teams, sometimes large, teams of individuals with different specializations. Many of them make extensive use of external libraries and have a modular structure that allows scientists to concentrate on contributing according to their area of expertise without having to become specialists in parallel programming. Defects in any one bit of code can potentially cause failures in other sections of the application. Troubleshooting the resulting error will take the developer into code that may be unfamiliar to them. In some cases the debugging process will get to the point of narrowing down the potential location of a defect to a specific area of code. Scientists and developers need to clearly communicate this diagnosis to their collaborators in a way that the collaborator can take action without having to repeat the entire troubleshooting process.

It is sometimes true that the entire development and use of an HPC application happens "under one roof" in the same geographic location, in other cases it is likely that some or even all of the developers collaborating together to develop and maintain an HPC code are widely scattered in terms of geography. Logging in and getting a command line prompt via the secure shell (SSH) application is straightforward for many users. Establishing a graphical connection that performs well enough to enable interactive use of a tool like a debugger is an entirely different matter. Many users report that they've tried to use the "X forwarding" capability within the SSH package across wide area networks. X traffic is highly sensitive to latency because a lot of the operations require many round trips of "small" packets of information. Most users report that regardless of the bandwidth of their connection long distance X forwarding is slow to the point of being unusable. This

makes interactive debugging over long distances a serious practical challenge for many users.

#### 1.1.1 What is TotalView?

TotalView is a source code debugger that is specifically designed to help HPC developers address the technical and organizational challenges outlined above. [9]It supports both interactive and batch debugging of distributed parallel applications on HPC systems including Cray's XT series of supercomputers. As an interactive debugger it allows users to control and examine the behavior of their parallel application, providing them with the kind of information that they need to troubleshoot and resolve a wide range of defects. As a batch debugger it can provide automated detection of certain classes of errors and support a trace-style troubleshooting model where the developer chooses points and variables of interest and generates a file that gives detailed information about the execution of the program, specifically tracing the variables and locations of interest. It works with the languages, libraries, and environments that are important to scientists and developers in HPC and provides specific capabilities to simplify troubleshooting MPI applications. It includes a mechanism for easily establishing secure and fast graphical connections that widely scattered users and collaborators can use to interactively debug on supercomputer resources that are located at major HPC centers. Users can store detailed data from within the program to enable later comparison and to share with collaborators who may be in a better position to respond to the errors that are identified.

### 1.2 This paper

The remainder of this paper is organized as follows. I'll first give a functional and architectural overview of TotalView as a debugger and as a memory debugger. Then I'll discuss three specific enhancements that have been added to TotalView specifically to address the major challenges outlined above; interactive subset debugging, batch debugging, and remote debugging. I'll close with a brief update on some upcoming developments for memory debugging and reverse debugging on the Cray XT.

## 2 TotalView on the Cray XT

### 2.1 TotalView as a Parallel Debugger

TotalView provides a powerful environment for debugging parallel programs. It allows users to easily control and inspect applications that are composed of not just a single process but sets of thousands of processes running across the many compute nodes of a supercomputer. At any time during a debugging session the user can choose to focus on any specific process – inspecting individual variables; looking at the call stack; setting breakpoints, watchpoints, and controlling that process; calling functions; and evaluating expressions within the context of that process. The user might choose instead to look at the parallel application as a whole – looking at the call tree graph which represents the function call stacks of all the processes in a compact and graphical form; looking at variables across all the processes (scalar variables are represented as arrays indexed across the set of processes, 1-d arrays as 2-d arrays, etc..); setting breakpoints, barrier points, and watchpoints across the whole application; running, synchronizing, and controlling the application as a whole; or looking at characteristics that are specific to parallel applications, such as the state of the MPI message queues. Alternately the user can choose to define, examine and control various sets of related processes through TotalView's dynamic process and thread set mechanism.

TotalView supports debugging applications written in C, C++, Fortran 77 or Fortran 90 and is compatible with a number of compilers. It supports applications that make use of MPI[6] and interoperates with the yod launcher mechanism on the Cray XT Series.

#### 2.1.1 TotalView Parallel Debugger Architecture

TotalView debugger provides for parallel debugging by itself becoming a parallel application – a single front-end process provides the user with a point of interaction with a graphical or command line interface while a set of lightweight debugging agents are created in the cluster to interact directly with the many processes that constitute the parallel program being debugged.

On an XT cluster running Linux on the compute nodes TotalView creates a set of debugging agents (called tvdsvr processes) alongside the users target

program. On Cray XT systems that run Catamount a variation of this architecture where the tvdsvr processes run on service nodes is used to provide the same debugging capabilities.

# 3 TotalView as a Memory Debugger

TotalView debugger implements an integrated memory debugging tool that provides vital information about the state of memory. It reports some errors directly as they occur, provides graphical and interactive maps of the heap memory within individual processes and makes information like the set of leaked blocks easy to obtain. TotalView's memory debugging is designed to be used with parallel and multiprocess target applications – it provides detailed information about individual processes as well as high level memory usage statistics across all the processes that make up a large parallel application. TotalView's memory debugging is lightweight and has a very low runtime performance cost.

The memory debugging capabilities are also available to users in stand alone form as a product called ReplayEngine. [7]

### 3.0.2 TotalView Memory Debugging Architecture

TotalView accomplishes memory debugging on the Cray XT through the use of a technique called interposition. [4]TotalView provides a library called the Heap Interposition Agent (HIA) that is inserted between the user's application code and the malloc() subsystem. This library defines functions for each of the memory allocation API functions and it is these functions that 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 it would without the HIA except that the HIA is able to intercept all of the memory calls and perform bookkeeping and sanity checks before and after the underlying function is called.

The bookkeeping that the HIA library does is to build up and maintain a record of all of the active allocations on the heap as the program runs. For each allocation in the heap it records not just the position and size of the block but also a full function call stack representing what the program was doing when the block was allocated. The sanity checks that the HIA performs are the kinds of things that allow the HIA to detect malloc() errors such as freeing the same block of memory twice or trying to reallocate a pointer that points to a stack address. Depending on how it has been configured, the HIA can also detect whether some bounds errors have occurred. The information that the HIA collects is used by the TotalView debugger to provide the user with an accurate picture of the state of the heap that can be inspected just like any other part of the program's state during the debugging session.

The interposition technique used by TotalView was chosen in part because it provides for lightweight memory debuggging. For most programs that users will encounter the run time performance of the program being debugged is very similar to the performance that would be encountered without the HIA being interposed. This is absolutely critical for HPC applications where a heavyweight approach that significantly slowed the target program down might well make the run time of programs exceed the patience of developers, administrators and job schedulers.

# 4 Interactive Subset Debugging

Many programs that are important to HPC have a degree of regularity. Every process may not behave exactly the same way but there are often patterns or relationships between processes that behave similarly. This presents the possibility that when troubleshooting developers may be able to make progress without attaching a debugger to every single process that makes up the job. Instead in some cases it is sufficient to attach the debugger to a representative sample of the processes that make up the job. This is a very compelling idea when the overall job size is in the thousands or even tens of thousands of processes.

TotalView provides a mechanism to focus on an arbitrary subset of the program. The subset attach mechanism can be engaged either when the user

launches the parallel job or at any point after the debugger is already attached to the parallel program. If it is used during start up the whole job will launch but those processes that are not selected will run freely without any debugger intervention. The debugger, which is typically licensed by users based on the number of processes they intend to debug, will count for licensing purposes only those processes that it is attached to. At any later point the user can reopen the subset attach dialog and select a smaller, larger or simply different subset. The debugger will attach to new processes that are selected and detach from processes that are deselected.

The basic mechanism of the subset attach GUI is a list of processes from which users can select the ones they want to attach to. Filters are also available to make it easy to specify subsets based on communication patterns observed within the program. When working with a subset of processes it is possible that one of those processes is communicating, perhaps receiving a message from, a process that is part of the job but not part of the subset. This makes it easy to expand the set of processes based on this relationship.

Debugger operations after launch have a runtime performance and responsiveness that scales with the number of attached processes, rather than the whole job size. Certain debugger operations involve coordinating all the processes, which can take more than a few seconds if the user is asking for that coordination to go on across thousands of processes.

Processess that are not attached are not under the control of the debugger, and will run without interruption. The MPI communication mechanisms don't time out so any detached process will simply wait when it gets to the point at which it needs information from an attached process that might be paused. If a detached process encounters a fatal error the debugger will not be in a position to "catch" it for analysis and the error will cause the process to exit. Generally the MPI runtime will detect the exit and terminate the session as a whole. This behavior means that if a different process is failing each time that a program is run the best strategy is to attach to all the processes.

# 5 Using TVScript in Batch

The traditional use of a debugger involves the developer actively interacting with the debugging tool while the program is running. While that is a great way to explore program behavior there are a number of reasons developers might prefer a different approach. First, they are often used to batch submissions when working with the supercomputer and they may wish to fit their debugging into that mode rather than work out how to do an interactive session. Second, at some sites there is either no provision for an interactive session or if provisions only for small-scale runs. Third, they may want to survey the behavior of a slice of their program over time. Finally, they may want to do a parametric study of the defect, running the program while varying a specific input parameter to try to understand how the program behaves differently.

TotalView supports non-interactive debugging with a feature called TVScript. TVScript allows the developer to define a set of points of interest within the program, perhaps functions or lines within a function. Each time any process or thread within the program reaches these points an event is generated. Events can also be generated in response to other program behavior such as segementation faults and memory errors. For each event the developer can define an action to be taken. The action typically logs some information of interest such as the backtrace or the value of a variable. A log file is generated with all the information from all the events. The developer can then submit all of this as a batch queue submission and examine the logfile that is generated when it is complete.

Here is a usage example which will generate a logfile with a backtrace each time a.out enters *funcA()* or line 187 of *funcB()*.

```
tvscript \
-create_actionpoint "funcA" \
-create_actionpoint "funcB#187" \
-event_action "any_event=display_backtrace" \
./a.out
```

Use the command tvscript without any arguments on any recent installation of TotalView for a detailed list of all the arguments and options that it provides.

TVScript is especially useful if you want to detect memory type errors. You can enable memory debugging functionality such as guard blocks and then have the debugger trigger events when it detects that a chunk of memory is being freed that has had its guard blocks violated. At that point you can both print out information about the event, such as the time and location within the program and also store detailed heap memory information files for later analysis.

# 6 Remote Debugging with TotalView

If scientists or developers who needs to debug a problem on an HPC cluster are not co-located geographically with computer they may face a hurdle before even being able to consider debugging. Many sites provide either direct or indirect (via some intermediate host) SSH access to their authorized users. This kind of access is great for working at the level of the Unix shell. As mentioned previously, with the batch resource management systems most users interacting with supercomputers must upload a program, compile it, upload some data, set up a batch job that specifies running the program on the data, submit it, wait for some period of time, and then download the results. But interactive troubleshooting in the debugger with its graphical display of datasets and program state, doesn't fit well into this simple command line usage model. TotalView has a feature that automates and simplifies setting up a graphical connection between the users' local workstations and any remote supercomputer site that they have SSH access to.

In order to do this users first need the free TotalView Remote Display Client. It is included within recent (beginning with 8.6) versions of TotalView, so site administrators can post it to their users, or users can simply download it from the site they plan to log into. It can also be obtained directly from TotalView Tech's website.

The client is a simple executable that can be run on Linux or Windows. It launches a GUI with intuitive fields such as "username" and "hostname" and a Connect button. If users aren't using SSH's public-key infrastructure they will be prompted by the underlying SSH mechanism for their login password. The client takes care of the rest, setting up a secure graphical connection between the HPC server and the users desktop.[10]

In some cases the user isn't permitted to log directly into the HPC machine but must instead connect to one or more intermediate hosts. The client can handle that situation as well. The user specifies the sequence and the client connects to the first host and then connects to the second host via the first. The system doesn't store or even directly handle passwords and can work with cryptographic token technologies like SecureID.

Once the user has configured a connection they can store that connection as a profile for easy reuse. Individual users may have multiple profiles if they log into different supercomputers and profiles can be shared between users or distributed by site administrators, simplifying setup even further.

The remote display feature is architected to preserve the network security of the HPC resource. In particular it does not create listening ports of any kind on the HPC system. The graphical connection is established via a single outgoing connection from the HPC center machine where TotalView is running back through SSH to a non-privileged listening port on the users workstation.

# 7 Conclusion

There are a number of practical concerns that come into play when HPC users sit down to use debugging tools. We've found that it is necessary but not sufficient for a tool to have the technical characteristics such as supporting MPI and the latest compilers. We are now looking at the debugging and troubleshooting process from end to end and are identifying the places where there may be challenges and impediments for users. Debugging and troubleshooting isn't likely to ever be fun, but perhaps it can involve more satisfaction and less frustration. This paper highlights three solutions to such impediments. It has shown how users can focus on just a few processes drawn from the very largest scale jobs using the subset attach feature. The TVScript feature makes it easier for developers to work within the constraints of the batch queue workflow. Finally, geographically distributed users can use the Remote Display Client to connect into centralized HPC centers with just a few clicks.

Presentations at previous CUG meetings have highlighted two other important technologies. In 2007 we presented an overview of our work on Memory Debugging on the Cray XT.[4] Working with memory when porting and scaling applications was then and continues to be an important challenge. At CUG in 2008 we introduced a radically new technology to enable reverse debugging through recording and deterministically replaying program execution history. [3]I'll end with a few words to update readers on recent developments in these areas.

## 7.1 Memory Debugging

TotalView Technologies is currently completing development on a major improvement to the mem-

ory debugging features discussed in the 2007 paper. TotalView memory debugging supports error detection, leak detection, and the detection of heap array bounds errors. The guard block mechanism currently provided for detecting array bounds errors is very lightweight but provides detection only after the fact. It detects that the bounds of an array have been violated only when the block is freed or if the user specifically stops the application and asks the tool to perform a guard blocks check. Users have requested support for a more immediate detection of heap allocation bounds. We understand that they want a way for the tool to tell them about an array bounds violation as it occurs, not after the fact.

The next version of the TotalView and MemoryScape products for the Cray XT Linux Environment (CLE) platform will include a new capability based on the idea of allocating a memory-protected "red zone" before or after heap memory allocations within the user's program. The tool uses the page protection mechanism to obtain notification of array bounds violations as they happen. One significant benefit of this approach is that there is very little direct runtime overhead. If the program never tries to write to or read from the Red Zone the program runs at full speed. The first time that it does read or write to the zone the program comes to a halt and the user is notified.

The Red Zone page protection mechanism complements but does not completely supersede the guard block mechanism within TotalView and MemoryScape because they have different tradeoffs. Guard blocks can be as small as a single word (a few bytes) which makes it often quite reasonable to use them across all the heap allocations in the program. Red Zones have a fixed size of an entire page (several kilobytes). If the average size of an allocation in the heap is many kilo- or mega-bytes that overhead too is likely to be acceptable. However if the program makes many small heap allocations then the overhead of using Red Zones can become prohibitive.

TotalView provides two ways for users to obtain notification of bounds errors within programs that have many small allocations. Red Zones can be selectively enabled and disabled as the program runs. Users can run to a point of interest, enable Red Zones, step through the allocation of the data structures they are interested in analyzing, and then disable Red Zones. Alternately, they can define a size range of allocations that they are interested in having instrumented with Red Zones. Other allocations outside this range will not be instrumented with Red Zones. Both of these ways allow the user to focus the Red Zone capability on only those allocations that they are interested in. This has the potential to greatly reduce memory overhead and make Red Zones widely applicable.

### 7.2 Reverse Debugging

TotalView Technologies is also actively working with the help of Cray to enable the ReplayEngine product on Cray XT machines running CLE. [8] Several major technical issues have been addressed, though a few remain. Since the 2008 paper was presented, we have added MPI support to ReplayEngine. A version of ReplayEngine will be released soon with the ability to support the use of shared memory and Direct Memory Addressing (DMA) technology and long-running applications. All of these hurdles for supporting the Cray XT were identified in that earlier paper. A few smaller technical hurdles remain but we are optimistic we will be able to overcome them.

The enthusiastic interest that was expressed in Helsinki for the idea of reverse debugging has inspired us and we look forward to offering all developers on Cray XT CLE systems a radically simplified way to approach complex troubleshooting and debugging.

## Acknowledgements

Thanks to Gayle Procopio and the TotalView Technologies staff.

## About the Author

Chris Gottbrath is Director of Product Management at TotalView Technologies. He can be reached at 24 Prime Parkway, Natick, MA 01760. Email: Chris.Gottbrath@totalviewtech.com.

# References


[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. AFIPS*, 30, 1967.

[2] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.

[3] Chris Gottbrath. Reverse debugging with the totalview debugger. *Proc. Cray Users Group*, 30, 2008.

[4] Chris Gottbrath, Ariel Burton, Robert Moench, and Luiz DeRose. Debugging memory problems on cray xt supercomputers with totalview debugger. *Proc. Cray Users Group*, 2007.

[5] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[6] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[7] TotalView Technologies. MemoryScape. `http://www.totalviewtech.com/products/memoryscape.html`, 2009.

[8] TotalView Technologies. ReplayEngine. `http://www.totalviewtech.com/products/replayengine.html`, 2009.

[9] TotalView Technologies. TotalView Debugger. `http://www.totalviewtech.com/products/totalview.html`, 2009.

[10] TotalView Technologies. Using the Remote Display Client. `http://www.totalviewtech.com/support/documentation/totalview/remote_display.pdf`, 2009.