Debugging Memory Problems on Cray XT Supercomputers with TotalView Debugger

Chris Gottbrath, Ariel Burton
TotalView Technologies

Robert Moench, Luiz DeRose
Cray Inc.

ABSTRACT: *The TotalView Source Code Debugger gives scientists and engineers a way to debug memory problems on the Cray XT series supercomputer. Memory problems such as memory leaks, array bounds violations, and dangling pointers are difficult to track down with conventional tools on even a simple desktop architecture – they can be much more vexing when encountered on a distributed parallel architecture like that of the XT.*

KEYWORDS: Programming Environment Tools, Debuggers, Memory Debuggers, Cray XT Series

1. Introduction

The purpose of this paper is to highlight the availability of memory debugging on the Cray XT series supercomputer. Scientists and engineers taking advantage of the unique capabilities of the Cray XT can now identify problems like memory leaks and heap allocation bounds violations – problems that are extremely difficult and tedious to track down without the kinds of capabilities described here.

We begin by discussing and categorizing memory errors then provide a brief overview of both TotalView Source Code Debugger and the Cray XT Series. Finally we discuss how to perform several important memory debugging operations with TotalView Debugger.

2. Classes Of Memory Errors

Programs typically make use of several different categories of memory that are managed in different ways. These include stack memory, heap memory, shared memory, thread private memory and static or global memory. Programmers have to pay special attention, though, to memory that is allocated out of the Heap Memory. This is because the management of heap memory is done explicitly in the program rather than implicitly at compile or run time.

There are quite a number of ways that a program can fail to make proper use of dynamically allocated Heap memory. It is useful to develop a simple categorization of these mistakes for discussion. We will describe these in terms of the C **malloc()** API. However we believe that analogous errors can be made with memory that is allocated using the C++ **new** statement and the Fortran 90 **allocate** statement.

Malloc Errors

Malloc errors occur when a program passes an invalid value to one of the operations in the C Heap Manager API. This could happen if the value of a pointer (the address of a block) was copied into another pointer and then at some time later both pointers were passed to **free()**. In this case, the second **free()** is incorrect because the specified pointer does not correspond to an allocated block. The behavior of the program after such an operation is undefined.

Leaks

Leaks occur when a program finishes using a block of memory, discards all references to the block, but fails to call **free()** to release it back to the heap manager for reuse. The result is that the program is neither able to make use of the memory nor reallocate it for a new purpose. The impact of leaks depends very much on the nature of the application. In some cases the effects are very minor, in others where the rate of leakage is high enough or the runtime of the program is long enough, leaks can significantly change the memory behavior and the performance characteristics of the program. For long running applications or where memory is limited, even a small leakage rate can have a very serious cummulative adverse effect. This somewhat paradoxically makes leaks all that much more annoying – since they often linger in otherwise well understood codes.

Managing dynamic memory in complex applications to ensure that allocations are released exactly once so that **malloc** and leak errors do nor occur can be challenging.

Dangling Pointers

A pointer can be said to be dangling when it references memory that has already been deallocated. Any memory access, either a read or a write, through a dangling pointer can lead to undefined behavior. As with leaks the programs with dangling pointer bugs may sometimes appear to function without any obvious erorrs, sometimes for significant amounts of time – if the memory that the dangling pointer points to happens to not get recycled into a new allocation during the time that it is accessed.

Memory Bounds Violations

Individual memory allocations that are returned by **malloc()** represent discrete blocks of memory with defined sizes. Any access to memory immediately before the lowest address in the block or immediately after the highest address in the block results in undefined behavior.

Read-before-Write Errors

Reading memory before it has been initialized is a common error. Most languages assign default values to uninitialized global memory, and many compilers can identify when local variables are read before being initialized. What is more difficult and generally can only be done at runtime is detecting when memory accessed through a pointer is read before being initialized. Dynamic memory is particualy affected, since this is always accessed through a pointer, and in most cases, the content of memory obtained from the memory manager is undefined.

3. TotalView Overview

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 their attention on any specific process – inspecting individual variables, looking at the call statck, 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.

One advanced capability that can significantly aid users who are working at extreme scales is that TotalView does not need to be attached to the entire parallel job – it supports the idea of attaching to an arbitrary subset of the processes that make up the parallel application. Any processes that are 'detached' from the debugger will run freely and participate in the parallel program. This subset of attached processes can change over time as the user explores their parallel application.

TotalView supports debugging applications written in C, C++, Fortran 77 or Fortran 90 and is

compatible with a number of different compilers. It supports applications that make use of MPI and interoperates with the yod launcher mechanism on the Cray XT Series.

Details on these capabilities are provided in a variety of resources, such as the *TotalView Debugger Users Guide* and *Reference Guide*, Tip of the Week archive, and the TotalView Technologies Developer's Forum. These resources are linked to from http://www.totalviewtech.com/ .

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 GUI or a CLI 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 a cluster running Linux on the compute nodes TotalView creates a set of debugging agents (called tvdsvr processes) on the compute nodes alongside the user's target program and the processes can use the operating system's debugging mechanisms to debug the individual processes that make up the parallel program. As discussed below the XT architecture features compute nodes which are dedicated to running the users application with an absolute minimum of overhead. These most often run a lightweight operating system called Catamount. To support the parallel debugging of applications on the Cray XT series TotalView users a variation on the basic architecture (Figure 1).

Cray XT systems contain an additional set of nodes called 'service nodes'. The service nodes run linux kernel based operating systems. When used to debug a parallel application on the Cray XT, TotalView creates a set of tvdsvr processes on the service nodes. Each one of these server processes uses a specialized interface to remotely control some number of the user's processes running on the compute nodes. For example, in a typical configuration a user with 8192 processes to debug might be interacting with a TotalView session composed of one front end process and 128 tvdsvr processes -- each of which is controlling and debugging 64 of the processes that make up the users application, running on 64 different compute nodes. All of this happens in the background, most of the time there is no reason for the user to be aware of the number of tvdsvrs created or the identity of the service nodes on which the tvdsvrs are running.

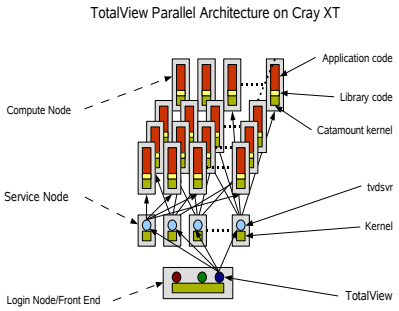

Figure 1: TotalView debugging architecture on the Cray XT3

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 both 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.

TotalView Memory Debugging Architecture

TotalView accomplishes memory debugging on the Cray XT through the modified use of a technique called interposition. 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 wheher some bounds errors have occurred. The information that the HIA collects is used by the TotalView Source Code 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. Low overheads are important if the performance of a program it not to suffer because of the presence of the HIA. In most cases, the runtime performance of a program being debugged with the HIA engaged will be similar to that where the HIA is absent. This is absolutely critical for high performance computing type applications – where a heavyweight approach that significantly slowed the target program down might well make the runtime of programs exceed the patience of developers, administrators and job schedulers.

4. Cray XT Series

Cray XT Series Overview

Cray XT series supercomputer systems are powerful, massively parallel processing (MPP) systems. Cray has combined commodity and open source components with custom designed components to create a system that can operate efficiently at immense scale.

Cray XT series systems are based on the Red Storm technology that was developed jointly by Cray Inc. and the U.S. Department of Energy's Sandia National Laboratories.

Cray XT series systems are designed to run applications that require large-scale processing, high network bandwidth, and complex communications. Typical applications are those that create detailed simulations in both time and space, with complex geometries that involve many different material components. These long running, resource-intensive applications require a system that is programmable, scalable, reliable, and manageable.

Cray XT Series Node configuration

The basic scalable component is the node. There are two types of nodes. Compute nodes run user applications. Service nodes provide support functions, such as managing the user's environment, handling I/O, and booting the system. Each compute node and service node is a logical grouping of a processor, memory, and a data routing resource.

Users log into service nodes and invoke commands and user applications from them. User applications are then propagated to compute nodes where they run using the Message Passing Interface (MPI) and SHMEM parallel programming, distributed memory models.

5. Memory Debugging

This section walks the user through a few simple memory debugging operations with TotalView on the Cray XT series supercomputer.

Compiling the Program

Prepare your application for parallel memory debugging by compiling and linking it as follows. The program should be compiled with optimization turned off and with debugging symbols included.

```
gcc -g -o target_app.o \
   -c target_app.c
```

This will generate a valid object file with debug symbols. In addition the compute node executable needs to be explicitly linked with the HIA library. The additional linker arguments " –Lpath –ltvheap_xt3 -gmalloc" will put the HIA into the parallel application being debugged.

```
gcc -L/<tvdir>/lib/ -ltvheap_xt3 \
-lgmalloc -o target_app target_app.o
```

The test app will run with the HIA enabled. Without the debugger present the HIA will simply record heap activity and pass through calls to the underlying malloc function.

Starting TotalView

On the XT series, parallel jobs are typically started using the yod launcher application. If the normal command used to launch the program was

```
yod -sz=256 target_app
```

then you start it for debugging with

```
totalview yod -a -sz=256 target_app
```

After issuing this command TotalView is at first debugging the yod executable itself. This is correct. The -a instructs TotalView to pass all remaining arguments to the target, in this case, yod. What you want to do is first instruct the debugger to run the yod executable by using the "go" command on the GUI or in the CLI. yod will run and when it has the parallel application started and ready to be debugged it will notifiy the debugger using an established interface. Typically TotalView at this point will prompt the user and attach to all of the processes that make up the application. This behavior can be customized to allow for subset attach.

If at any point in the debugging session you wish to restart the job you can do so. Because the debugger started the yod process first it will kill all the processes then reissue the same yod command – restarting the entire job.

See the *TotalView Debugger Users Guide* for details on how to control and inspect the state of the parallel application.

Activating Memory Debugging

Once the program is started the user will notice that pointers pointing into the heap memory are annotated with information about the heap status of the blocks they point to (Figure 2).

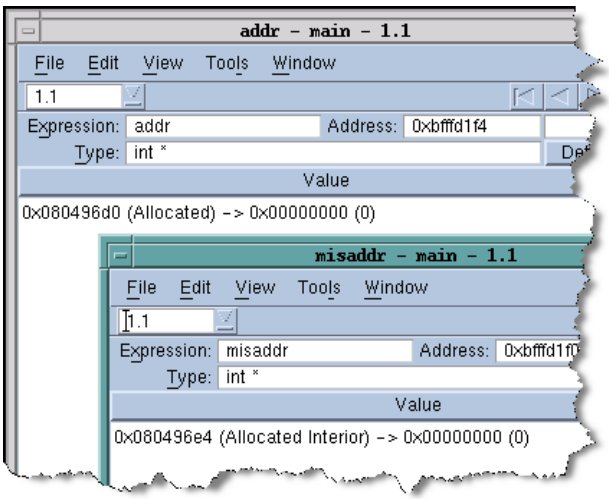

Figure 2: Annotation of pointers to heap allocations

Further memory information can be found on the memory debugging window. Open the memory debugging window by clicking on the tools menu and selecting "Memory Debugging" from about half-way down the list.

The memory debugging window presents a list of processes on the left and a tabbed area on the right. A variety of optional memory debugging features can be turned on and off on the configuration tab. For example heap guard blocks which are used to catch heap bounds violations can be enabled and disabled as can memory painting which is used to help track down dangling pointer and uninitialized memory errors. Other tabs allow the user to view memory usage statistics, heap status, memory leaks and to do detailed memory comparisons.

The remainder of this section highlights three specific tasks that the user may want to do when memory debugging a parallel application on the Cray XT Series supercomputer with TotalView. Please see *Debugging Memory Problems Using the TotalView Debugger* manual for a more systematic introduction.

Comparing Memory Statistics

Many parallel applications have known or expected behaviors in terms of memory usage. They may be structured such that all of the nodes should allocate the same amount of memory, or they may be structured such that memory usage should depend in some way on the MPI_COMM_WORLD rank of the process. If such a pattern is expected or if the user wishes to simply examine the set of processes to look for patterns the first place to look is on the Memory Statistics tab of the Memory Debugging window of TotalView. This will provide overall memory usage statistics in a number of graphical forms (line, bar and pie charts, see Figure 3) for one, all or an arbitrary subset of the processes that make up the debugging session. The user should drive the program to a specific breakpoint, barrier, or simply halt all the processes at an arbitrary point in execution. Then in the memory debugging window the user should select the set of processes they wish to see statistical information about, then select the type of view that they want to see then click 'generate view'. The generated view represent the state of the program at that point in time. The user may then use the debugger process controls to drive the program to a new point in execution and then update the view to look for changes.

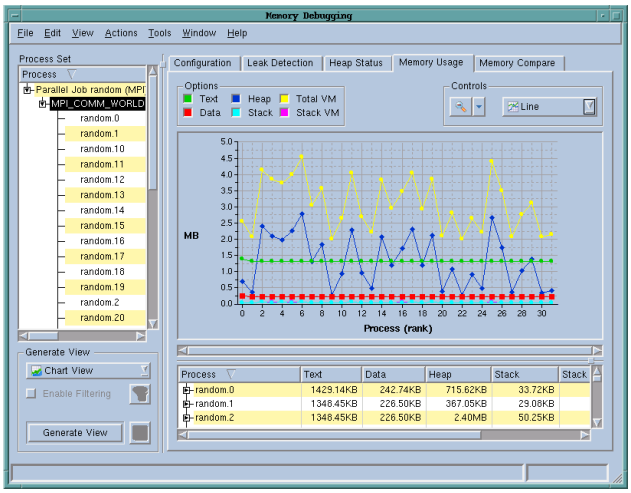

Figure 3: Comparing memory statistics among a set of processes.

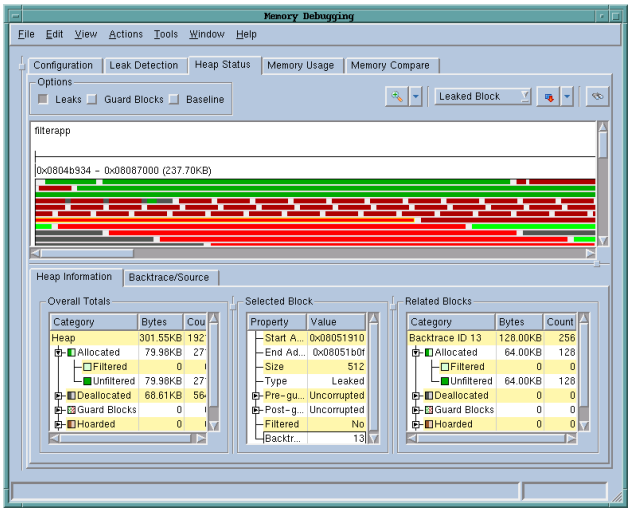

Figure 4: Graphical heap display with leaks marked in red.

If any processes look out of line the user will likely want to look more closely at the detailed status of the heap memory.

Looking at Heap Status

TotalView 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 user can get a graphical view of the heap. This is obtained by bringing up the Memory Debugging window, selecting the heap status tab, selecting one or more processes, choosing the graphical view and clicking 'generate view'. The resulting display (Figure 4) paints a picture of the heap memory in the selected process. Each current heap memory allocation is represented by a green line extending across the range of addresses that are part of the allocation. This gives the user a great way to see the composition of their heap memory at a glance. The view is interactive – selecting a block highlights related allocations and presents the user with detailed information about both 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). The display also supports setting a baseline which lets the user see which allocations and deallocations occur before and after that baseline.

Detecting Leaks

Leak detection can be done at any point in program execution. As discussed above leaks occur when the program ceases using a block of memory without calling free. It is hard to define 'ceasing to use' but the debugger is able to do a leak detection by looking to see if the program retains a reference to specific memory locations. Perform heap memory leak detection by driving the program to a known state (a breakpoint perhaps) or simply halting the processes of a running parallel application using the 'halt' command in the GUI or the CLI. Then select the leak detection tab in the memory debugging window, select one or more of the processes in the parallel job and then generate the leak report.

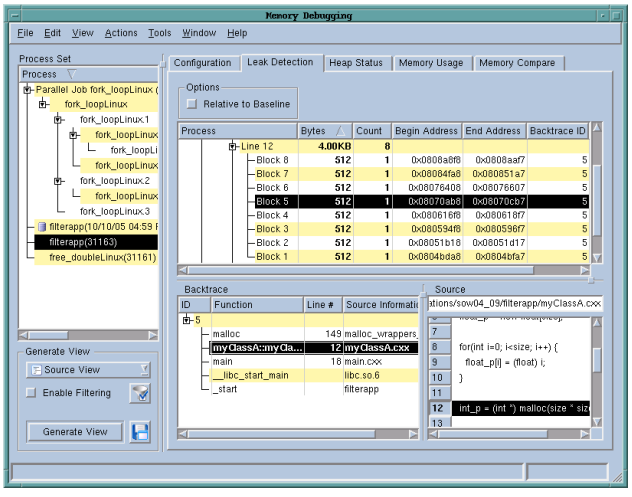

Figure 5 : Leak report

The resulting report (Figure 5) will list all of the heap allocations in the program for which there are no longer any valid references anywhere in the program's registers, or accessible memory. A block of memory that the program isn't storing a reference to anywhere is highly unlikely to subsequently be subject to a **free()** call and is extremely likely to be a leak.

Leaks can also be observed in the heap graphical display discussed above (Figure 4). Toggle the checkbox labeled 'Detect Leaks' to have leaked blocks displayed in red on the graphical display.

6. Conclusions

In this paper we have briefly reviewed some of TotalView's Heap Memory Debugging capabilities. We have shown how these capabilities can help programmers find memory leaks in their codes. We have also shown how TotalView can be used to determine when data are used before being initialized, or after having been released back to the heap manager. The Heap Debugger can also be used to help locate bounds errors, where, for eample, the program writes beyond the ends of a dynamically allocated array.

TotalView's powerful reporting tools allow the states of a program at different points of its execution to be compared and analyzed. This allows programmers to improve their understanding of their programs' behaviors

TotalView runs on a wide range of platforms, from single user workstations at one end of the spectrum to supercomputers at the other. In many cases no special steps are required to prepare a program for memory debugging. The architecture of the Cray XT3, however, is such that the usual approach must be modified. We described how all that needs to be done to enable memory debugging is to link the program with Heap Interposition Agent, and of course, start the program under TotalView.

Bibliography

Cray XT3TM System Overview

www.totalviewtech.com.

http://www.totalviewtech.com/Documentation/mpi_startup.html

About the Authors

Chris Gottbrath is Product Manager for the TotalView Debugger at TotalView Technologies LLC. He can be reached at 24 Prime Parkway, Natick, MA 01760 USA, E-mail: chris.gottbrath@totalviewtech.com

Ariel Burton is a Senior Software Engineer at TotalView Technologies LLC. He has worked extensively on the TotalView's Memory Debugging functionality. He can be reached at 24 Prime Parkway, Natick, MA 01760 USA, E-mail: ariel.burton@totalviewtech.com

Luiz DeRose is a Sr. Technical Engineer and the Programming Environment Tools Manager at Cray Inc. He can be reached at 1340 Mendota Heights Rd, Mendota Heights, MN 55120 USA, E-mail: ldr@cray..com

Bob Moench is a Software Engineer at Cray Inc. He has worked on debuggers for the last several years. He can be reached at 1340 Mendota Heights Rd, Mendota Heights, MN 55120 USA, E-mail: rwm@cray.com