Improving the Productivity of Scalable Application Development with TotalView

Chris Gottbrath *TotalView Technologies, a Rogue Wave Software Company*

May 12, 2010

Abstract

Scientists and engineers who set out to solve grand computing challenges need TotalView at their side. The TotalView debugger provides a powerful and scalable tool for analyzing, diagnosing, debugging and troubleshooting a wide variety of different problems that might come up in the process of such achievements. These teams, and teams of scientists pursuing a wide range of computationally complex problems on Cray XT systems, are frequently diverse and geographically distributed. These groups work collaboratively on complex applications in a computational environment that they access through a batch resource management system. This talk will explore the productivity challenges faced by scientists and engineers in this environment – highlighting both long standing (but perhaps unfamiliar) and recently introduced capabilities that TotalView users on Cray can take advantage of to boost their productivity. The list of capabilities will include the CLI, subset attach, Remote Display Client, TVScript, MemoryScape's reporting, and ReplayEngine.

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

1 Introduction

1.1 The Productivity Challenges of HPC Debugging

There has been an effort over the past five to ten years to shift the focus of discussion in High Performance Computing (HPC) from performance towards productivity. This is a difficult thing to do because to talk about productivity means focusing less on bits, bytes and flops and more on the human element in HPC. For a community of technologists, computer and computational scientists this is a challenging thing to do. When we talk about the productivity of HPC we are drawing our attention to all of the people who interact with HPC systems and explicitly focusing on how the system as a whole, or the component of the system that we choose to focus on, fits the needs of that group of human beings and aids them in accomplishing the larger goals.

This paper will look at how this plays out with scalable debugging. Debugging is something that all programmers need to do with all codes both in HPC and in other programming disciplines. This paper focuses on the specific challenges faced by HPC programmers writing code for the Cray XT. HPC programmers are typically physical scientists (or biological scientists, mathematicians economists or other domain specialists), computer scientists, engineers or software developers., all with different characteristics and needs.

Debugging plays a key role whenever developers write, deploy and/or maintain code since unexpected defects can turn up at any point. Software, particularly parallel software, is a complex system and when things don't go as expected in the system someone, often the developer of that software, needs to sit down and determine out why. This kind of troubleshooting can be a very time-consuming process – sometimes tracking down bugs and understanding them can take days or weeks. Debuggers that are designed with the needs of these users in mind can reduce these delays and have a very direct impact on the productivity of the system as a whole.

Three aspects of HPC software development are particularly relevant to this discussion:

- The distributed nature of development

- Dealing with user environments
- Making the best use of a developers attention

The Distributed Nature of Development

Most scalable parallel software is too complex to have been written by a single developer. Many of the most prominent applications are collaboratively developed by quite diverse teams of researchers, scientists and computer scientists. These teams are frequently geographically distributed across multiple buildings, campuses or even continents. Some or all of the team then are not co-located with the supercomputer that they are writing software for. There are well known techniques for logging in and using HPC systems across long distances, but many of them are ill-suited for the kind of interactive, and graphics heavy approach that is generally most effective for debugging. A particular challenge for these teams exists when the team member who can best troubleshoot a problem somone who can't easily access the system. There are less well known techniques for setting up graphical connections across long distances but they tend to involve multiple steps and be somewhat arcane. If we can make it equally easy for all the developers to access the system we can improve the productivity of the team as a whole.

Once a problem is located in the code other members of the team may need to be consulted. While many scalable applications regularly apply computer science best practices, such as the use of reusable components and clear layering and APIs many resources are shared by those components and a developer working in one area of code may uncover a defect in another. Making it easier and faster for teams to collaborate and resolve such issues provides a clear productivity enhancement.

Dealing with User Environments

There are a wide variety of different ways in which HPC clusters can be set up for users to access; different vendors provide different tools and many sites have unique institutional needs or approaches to managing the computing resources that the cluster provides. This means that developers who are working on a given simulation, code or project are likely to use a variety of different systems. Individual tools can reduce complexity by abstracting away some of those details when they are not particularly relevant to the troubleshooting task itself and by providing capabilities that work the same regardless of which system the developer is working with at a given time.

Large supercomputers are valuable resources and organizations that put them to use or provide them for others to use quite naturally put systems and processes in place to make sure that their capabilities are shared fairly and used in an efficient manner. In most cases this means that access to the HPC cluster is managed by the use a batch resource management and queuing system. This encourages users to break up their computation into jobs that can be run sequentially and independently by the batch queuing system itself. Users prepare these jobs and submit them into some system of queues based on the policies and practices established by the organization that is allocating users time on the compute resource.

There is a bit of a mismatch between the practices that users develop based around non-interactive use and the interactive exploration that is typical of troubleshooting sessions. Troubleshooting tools that can provide at least some capabilities for working within this batch workflow may lower the barrier for tool adoption among developers might otherwise eschew the use and benefits of advanced debugging tools. Frequently those programmers will instead use debugging techniques like adding print statements to their programs. This labor-intensive process involves a very slow cycle of recompiling and rerunning. There is a lot of potential for significant productivity improvement through the development of tools specifically for batch debugging.

Making the Best Use of a Developer's Attention

The domain scientists, computer scientists, engineers, and others who interact with these large systems have very specialized training and have many demands on their time. It is important that we regard any time and effort that they put into troubleshooting as especially valuable. Tools should do a variety of things to ensure that the developer who is troubleshooting isn't overwhelmed and can focus as clearly as possible on the line of investigation that is the essence of troubleshooting.

As more and more clusters reach into the petascale range there is a very real question about whether programmers really want to attach a debugger to hundreds of thousands of separate threads of execution. Even with scalable architectures and

advanced visualization it isn't clear that developers are well served by trying to deal with that many processes or threads. However, there will always be a class of defects that cannot easily be reproduced when the job is run at a more modest scale. Tools need to support the concept of debugging a subset of the whole scale job.

Parallel applications are complex and it is important that debugging tools go beyond simply allowing the developer to step through the program and look at data. Troubleshooting tools should allow programmers to query their application, adding assertions and sanity checks – automating the process of discovering places where the program isn't doing what the programmer expects. Once the defect is located the tool should allow for testing that the developer has really understood the problem, perhaps replacing code with a new code fragment. Abstractions such as templates store data very efficiently but may do so in a way that makes it hard for the developer to get a high level logical view of the data that is important to them. Tools can provide ways to transform that data into a more readable form.

Fundamentally troubleshooting is about working backwards from the result of an error towards the cause of that error. Tools that can provide a way to work with the application in the same way can greatly simplify many aspects of troubleshooting.

1.2 Organization of this paper

The remainder of this paper will introduce TotalView Technologies' three debugging products and then highlight specific features and functionality of these products that address the productivity challenges outlined above. Section 2 introduces TotalView, MemoryScape and ReplayEngine. Section 3 discusses features that specifically address the concerns of a distributed team of developers. Section 4 talks about features that help developers work within a resource-managed environment. Section 5 highlights features that help developers leverage the attention that they are spending on debugging most effectively. The paper concludes with some highlights of recent progress on the Cray XT platform specifically.

2 TotalView on the Cray XT

The Cray XT is a highly scalable distributed supercomputer architecture from Cray, Inc. It is a cluster of compute node based on the x86-64 architecture with a proprietary high bandwidth, low latency network interconnect. It features a Linux- based front end for compiling codes and preparing and launching jobs, and compute nodes that either run a proprietary OS called Catamount or an optimized variant of Linux referred to as the Cray Linux Environment (CLE). The full system includes various levels of disk storage and software for managing and administering the compute resource. These include the *aprun* parallel program launch command.

2.1 TotalView

TotalView provides a powerful environment for debugging parallel programs. [9] 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[5] and interoperates with the aprun launcher mechanism on the Cray XT Series.

2.2 MemoryScape

MemoryScape is the name for the memory debugging functionality within the TotalView product

family.[7] Most but not all license configurations of TotalView include a copy of MemoryScape, but it can also be purchased and used separately. MemoryScape provides vital information about the state of memory in either a highly graphical format or through a variety of reports that can be stored for later analysis.. 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.

MemoryScape is designed to be used with parallel and multi-process 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 run-time performance cost.

At CUG 2007 we presented a technical overview of our work to port MemoryScape to the Cray XT platform. [4]

2.3 ReplayEngine

ReplayEngine is an add-on to the TotalView debugger that enables users to do reverse debugging. It is available for 32- and 64-bit x86-based systems running Linux, and supports reverse debugging applications written in C, C++, Fortran 77, Fortran 90 and UPC. [8]

Reverse debugging means working directly backwards from the visible sign of a defect (the crash or incorrect result that you see that lets you know that there is a bug) through the execution history of the program to find the root cause of the bug. This is a radically simpler approach to troubleshooting because at any point in the process you are able to take advantage of hindsight: you know exactly which memory address, variable, allocation or data value is the problem and the question is just "how did it get this way".

At CUG 2008 we presented an introduction to the technique of reverse debugging and an overview of the record and replay technology used to implement it. [2] At that time we had just introduced the product and while we supported Linux-x86 and x86-64 clusters we knew (and highlighted in the paper) that there would be problems with applying the technology to the Cray XT series. However there was a lot of enthusiasm expressed and since then we've been working on the problem. We expect that when this paper is published customers will be able to use ReplayEngine with Cray XT supercomputers that are running up-to-date installations of ReplayEngine and of the Cray software environment.

3 Working with Distributed Teams

HPC development is an activity performed by distributed teams of developers with diverse skills. Here I highlight two specific ways that we help out with the challenges that come up in these collaborations.

3.1 Remote Debugging with TotalView

If scientists or developers who need to debug a problem on an HPC cluster are not co-located geographically with the 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. Most users' interactions with supercomputers are mediated by and through a batch resource management systems of some kind. This means that users 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 data sets 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.

This requires the free TotalView Remote Display Client. It is included within recent versions of TotalView (beginning with 8.6), 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 Technologies' website.

The client is a simple executable that can be run on Linux, Windows, or Apple Mac OS X. It provides 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 mech-

anism for their login password. The client takes care of the rest, setting up a secure graphical connection between the HPC server and the user's 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, 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 it 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 user's workstation.

3.2 MemoryScape Reporting

MemoryScape supports multiple modes of usage. A key feature is its capability to create reports of various characteristics such as the amount of memory used by data structures in different parts of the program, and the amount of memory that has been leaked and where that memory was allocated within the program. These reports are generated based on the analysis of a running instance of the program and are structured to present information hierarchically; developers can start with an overview of the memory usage and "drill down" on sections of the code that seem to have many or unexpectedly large leaks or allocations. MemoryScape also provides extremely flexible filtering capabilities to help developers either focus on a specific set of data based on a specified characteristic (such as only allocations that are bigger than a defined size) or to exclude allocations and leaks that are uninteresting because they are in vendor libraries that the developer can't control.

Memory is a resource that is shared by all the different parts of the program. An error in one part of the program can lead to corrupted data in another part of the program, so often the developer who discovers a memory error or comes to suspect that memory isn't being used in an efficient way may not be the same developer who is in the best position to understand how to change the code to correct the error or improve efficiency. We've therefore focused some significant effort in making sure that it is easy for developers who use MemoryScape to discover such issues to clearly communicate what they find with their colleagues.

MemoryScape can package information on a given target in two ways. First, it can generate a report of the data in one of two formats. These reports provide a way to highlight the results of analyzing a program with MemoryScape. If filtering has been selected before the report is generated, information will be filtered in or out of the report based on those settings.

The most helpful format for many users when they simply want to notify a collaborator about a leak in a section of code is the active-HTML report, a javascript-enhanced HTML page that is a simplified form of MemoryScape's source or backtrace heap allocation or leak report. It features a high-level view of the memory usage, organizing memory allocations or leaks by program location and providing aggregated statistical information at each level of the hierarchy. The report can be reviewed in any javascript-enabled HTML browser, from the summary view to drilling down to individual allocations.

MemoryScape can also generate a text report with the same information. Due to the static nature of plain text the recipient would have to navigate the hierarchical listing to find the allocations of interest (instead of being able to start with the collapsed tree and pick elements to explore). On the other hand. the text report is perfect for input to any kind of scripted or automated processing that the user might want to do. Both of these export formats allow developers with MemoryScape to effectively communicate with colleagues without requiring those colleagues to use MemoryScape.

The second way MemoryScape can package information is via an export of raw data on the target process memory state to a binary file that can be reloaded and reanalyzed later by the same or a different instance of MemoryScape. This allows a developer to set aside the state of the process for more intensive analysis for a more convenient time. One capability that this opens up is the ability to do a detailed comparison of the state of heap mem-

ory within a single program at two different points in time or between two different runs of an application. For users who are concerned about maintaining code quality this comparison technique opens the door for some very powerful validation techniques.

4 Working with User Environments

The richness and diversity of parallel cluster configurations can be an issue for HPC programmers and users. Here I highlight ways that TotalView helps to deal with that complexity and some of the issues that can come up in batch environments.

4.1 Cross Platform and Indirect Launch

One challenge for developers is that there is frequently a non-trivial amount of complexity to just getting jobs running and launched on the various different HPC resources that they may use. This complexity is the result of both the number of systems and the variety of batch management systems, usage policies, filesystem configurations, sets of appropriate commands for the architecture, various local configurations and customization used on each large system. A variety of different tools have been created to try to manage this complexity, from modules that shield users from having to be cognizant of local environmental settings for library and executable search paths, to extensive web service enablement of computational science work-flows such as ADIOS and Kepler. [6, 1]

In this context TotalView provides a very straightforward advantage for users. It supports a wide range of platforms with a very uniform set of capabilities and the same look and feel for scientists and developers who may be moving back and forth between various machines and architectures.

Until fairly recently however the startup command for TotalView on a parallel job was a point of potential confusion. One of the key features that TotalView provides is the ability to issue a single command and have the debugger attach to not just one process but all the processes (or an arbitrary subset, see below) that make up a parallel job. This requires a significant degree of integration with the mechanism that is used on that particular cluster and with that particular MPI. Because some MPI launch mechanisms are scripts and others binaries, users needed to remember different ways of starting TotalView on different systems. A few years ago we introduced indirect launch, allowing users to start up the debugger in the same way regardless of the system, with the exception of the BlueGene platform. Users simply start up TotalView on the application they wish to debug, select the MPI library they would like to use, specify the number of MPI ranks and other options, and click Continue. They have an opportunity to inspect their code and set breakpoints in their program. They launch their program with the Go button, and TotalView takes care of the details.

4.2 TVScript for Batch Debugging

The traditional use of a debugger involves a developer actively interacting with the debugging tool while the program is running. While that is a great way to explore program behavior, developers might prefer a different approach for a number of reasons. 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 provision only for small-scale runs. Third, they may want to survey the behavior of a slice of their program over time. Fourth, they may want to do a parametric study of a defect, running the program while varying an input parameter 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 segmentation faults and memory errors. For each event the developer can define an action to be taken. The action typically logs 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 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 have the debugger trigger events when it detects that a chunk of memory that has had its guard blocks violated is being freed. At that point you can print out information about the event, such as the time and location within the program, and store detailed heap memory information files for later analysis.

TVScript support for the Cray required an effort beyond that required for traditional Linux clusters. Interested readers can contact the author for a pre-release version that will work on the Cray XT. General availability is anticipated in the near future.

5 Making the Best Use of a Developer's Attention

Over the years we have developed many capabilities in TotalView that make it possible for developers to do more with the attention and time that they put into troubleshooting and debugging. I'd like to highlight a few of these here.

5.1 Subset Attach

TotalView provides a mechanism to focus on an arbitrary subset of the program. The subset attach mechanism can be engaged either when the parallel job is launched 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 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 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. The GUI makes it easy to expand the set of processes based on this relationship.

Debugger operations after launch have a run-time performance and responsiveness that scale 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 across thousands of processes.

Processes 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 run-time 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 may be to attach to all the processes.

5.2 Evaluation Points

Evaluation points provide a way to associate code fragments, typically written in C, C++ or Fortran, with breakpoints and watchpoints. These code fragments are interpreted within the context or scope of the evaluation point and can also make use of additional context and functionality provided by the debugger. This allows for a significant reduction in time in getting to the bottom of issues when troubleshooting.

Many of the codes that run on the Cray XT feature numerous loop operations, making it impractical to "step" through the code, since it may take an extremely large number of "steps" to get to a point of interest. Setting a regular breakpoint can be used to run the program into the next iteration of the loop, but when loops are going to be executed thousands or millions of times, get-

ting to the next loop may not be particularly helpful. In this case evaluation points can be used to run the program to an integration of interest. In a loop over the integer i putting the expression
if (i == 49553) { $stop; }
in a breakpoint on a line of code in the loop is all that is needed to stop the application for inspection at the 49553rd iteration of the loop. The expression can get as nuanced as the necessary to select the point at which to stop; for example, multiple variables can be compared, functions can be called that operate on the data and temporary variables can be created to store intermediate values. This means that the expression point can be used as a sort of assertion or sanity check. Many times troubleshooting is about testing a hypothesis about the state of the program, so this kind of assertion checking directly automates one of the common operations of debugging and troubleshooting.

TotalView provides a way to view array data in graphical format and an evaluation point can be added with a *$visualize(array)* directive (where array is the name of the array to be displayed). If this is placed within a loop and the program is run the debugger will display an animation of the data within that array as it is transformed by the program. This may help in understanding what is going on with the data in the program. This differs from standard scientific visualization in that the array data that is displayed can be any intermediate data within the program; it doesn't have to be the final output data from the program. It is worth noting that this functionality should not be attempted using the remote display client as the rapid graphical changes will overwhelm the connection.

Typically once developers understand the problem they will exit the debugger, change their code, recompile it and rerun the application to see if their change fixes the error. If the problem was one that required a delicate set up to examine in detail this can be very tedious. Here again evaluation points can be used to greatly simplify the debugging process. Instead of recompiling and restarting the application the user can prototype the change in the debugger using evaluation points and either rerun the application from the beginning or rerun part of the application, and verify that the new code changes the behavior in the desired way.

5.3 TTF and C++View

Modern languages such as C++ and Fortran 90 provide developers with ways to create various kinds of abstractions that allow data to be stored in a way that is compact or efficient and operate on that data in a simplified way. Generally debuggers present the user with a very direct representation of the data as it is stored within the program's memory. This direct representation can itself be a significant hurdle because it can make it hard for the developer to follow the flow of the program, and sometimes makes it extremely tedious to get at the data that the user wants to see. For example, frequently a linked list construct is used for variable collections of data with a variable of unknown length. A raw display of the data may take a series of dive operations equal to the number of items in the list just to see the values in the collection.

TotalView provides several facilities that are helpful. Many C++ programmers use a facility called the Standard Template Library which contains a generic implementation of a list (along with numerous other standard ways to work with collections of data) which is implemented very similarly to what I just described. TotalView automatically transforms these objects and displays them in a way that transparently represents the data that has been placed in the collection. This makes it easier for developers to focus on how they are using the STL collection object in their application as opposed to how the STL collection object is itself implemented.

TotalView provides two additional ways for users themselves to create transformations for objects that they define. A user can create a type transform using the Type Transform Facility (TTF). These transformations can either transform structures to other structures (omitting, reorganizing, renaming fields to bring the relevant data to the foreground), or aggregate data together into array-like types (a walking list and other data structures to create something that looks like an array object). These transformations occur entirely in the debugger and can be used on corefiles and programs that are hung. The main limitation of these transforms is that they can only display data that already exists in the objects to be transformed; they can't make use of existing or new functionality in the target program to derive new data. The TTF transforms are written in a simple stack- based addressing language and coded in the TCL scripting language.

In addition we've recently developed a technique

to allow developers to define type transformations using a C++ call-back mechanism. This feature is called C++View and is currently in experimental pre-release.[11] The first advantage that this approach provides is that developers can work in C++, which is frequently more familiar and comfortable than TCL and the TotalView addressing language. It may also be shorter; many useful transforms are only a few lines of C++ code. The second advantage is that transformations can be more expansive. Summary data can be generated and decisions about what data to represent can be made on the fly. The third advantage is that the transforms can be made a part of the user's application. This is a big benefit for large HPC teams because a single member of the team can define the transformations, make them a part of the HPC application itself and the entire collaboration can benefit from easier debugging without having to take any action. The biggest disadvantage of this technique is that it requires the target program to run in order to perform the transformation. This means that it does not provide any benefit when the user is debugging corefiles and hung jobs. Users can visit the TotalView Technologies website to obtain more information about this feature, and register interest in and download the interface files required to take advantage of the feature.

5.4 ReplayEngine

Much of the frustration and complexity of debugging with current tools comes from the fact that programmers have to work in a very indirect way to make the connection between the result of a bug and its cause. The general nature of troubleshooting is that errors in programs may not be immediately evident; the program continues to evolve forward until such time that it generates an error, either an invalid operation or a bit of output that is obviously wrong. That's the point where we get involved with a debugger.

ReplayEngine allows developers to analyze the program directly from that point where the error is visible "backwards in time" to the point where the error actually occurred. This can radically simplify the debugging process in two ways. First, it can greatly simplify the process of isolating the defect. Second, in troubleshooting, a developer no longer needs to work through a series of careful steps punctuated by restarting the job and running it to an earlier point.

In the parallel context a program may fail in such a way that the defect occurs on different MPI processes each time the program runs. This makes it very difficult (though not impossible) to use traditional forward debugging techniques because the developer has to carefully run all processes forward together (which may be easier or harder depending on how the parallel code is structured) and watch each process for signs of the one that is going to exhibit the crash. With ReplayEngine developers can run an application on the cluster and trace the behavior of each process, and when one crashes or generates output data that is clearly incorrect, focus their attention on just that process. They can examine execution history to discover where the error occurred. It is possible that the process they initially focus on as the site of the crash actually crashes because it got bad data from a second process. In that case they can switch their attention to this second process, run it back to the point of interest (where it sent the bad data) and continue debugging backwards towards the root cause.

The advantage of not having to restart the application numerous times is that developers can focus on following one clue to the next clue and so on back to the root cause. In troubleshooting using traditional forward debugging they frequently need to step aside from the problem they are chasing and focus on re-running the application to an earlier state. Even with tools like evaluation points this can be a critical distraction, taking a lot of time and introducing other activity that can distract from seeing the clue they were looking for when they restarted.

This product was discussed in greater depth in a previous presentation here at CUG 2008 and an update was given a year later at the EuroPVM/MPI 2009. [2, 3] Since these papers were presented we have matured the product to the point that it can now be used within the Cray XT environment (when the compute nodes run CLE; it does not work with Catamount). This requires ReplayEngine 1.7 (distributed along with TotalView 8.8) and the Cray TotalView Support module version 1.1.

Several other significant improvements have been made to ReplayEngine since that presentation. First, we've introduced support for long-running programs. ReplayEngine defines a limit (user-adjustable) for the recorded history to consume. When data necessary to store recorded history exceeds that amount the oldest data is discarded and execution continues. Previous versions simply stopped the program from running when the limit was reached. This means that for a short program

the developer may be able to run all the way back to the very start of the program but with a long-running program there will be some limit to how far back the developer will be able to explore.

The second major improvement is support for a Backwards Continue command. This is the analog of the go command and runs the program to the nearest breakpoint or watchpoint in the backwards direction. It is important to note that the user can freely set, modify, enable and disable breakpoints and watchpoints before hitting Backwards Continue. So the developer can let the program run to a crash, select a variable of interest (generally one that has an unexpected value), set a watchpoint on that variable and run to the point in the execution history when the variable was set. This can be repeated to follow bad data back to its source.

Finally, the performance and stability of ReplayEngine, especially when used in conjunction with multiple threads, has been steadily improved over time.

6 Conclusion

This paper has reviewed eight major features and areas of functionality within the TotalView debugger product family that directly address the kinds of concerns that sometimes limit the productivity of scientists, engineers and software developers working on HPC software. Some of these features are brand new and I've highlighted where users need to look at the latest version of TotalView to gain these benefits. Other features have been part of TotalView for a long time. TotalView is a comprehensive and rich product and I encourage users to explore its features both in the documentation and the product itself, talk to one another using our user forums, and speak to us here at TotalView Technologies about any questions, comments, and suggestions that they might have. [12]

Acknowledgements

Thanks in particular to Gayle Procopio and to my colleagues at TotalView Technologies and Rogue Wave Software.

About the Author

Chris Gottbrath is Principal Product Manager at TotalView Technologies, a Rogue Wave Software Company. He can be reached at 24 Prime Park Way, Natick, MA 01760. Email: Chris.Gottbrath@totalviewtech.com.

References

[1] National Center for Computational Sciences. Adios adaptable io system. http://www.nccs.gov/user-support/center-projects/adios/, 2010.

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

[3] Chris Gottbrath. Bringing reverse debugging to hpc. In *Proceedings, 16th European PVM/MPI Users' Group Meeting*, Espoo, Finland, September 2009.

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

[6] Kepler Project Team. Kepler project. https://kepler-project.org/, 2010.

[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.

[11] TotalView Technologies. TotalView C+View feature web page. http://www.totalviewtech.com/forms/cppview.html, 2010.

[12] TotalView Technologies. TotalView Technologies User Forum. http://forum.totalviewtech.com/, 2010.