Debugging and Optimizing Scalable Applications on the Cray

Chris Gottbrath
Rogue Wave Software
Boulder, CO
Chris.Gottbrath@roguewave.com

Abstract— **Cray XE6 and XK6 systems can deliver record-breaking computational power, but only to applications that are error free and are optimized to take advantage of the performance that the system can deliver. The cycle of development, debugging and tuning is a constant task, especially when custom application developers implement new algorithms, simulate new physical systems, port software to leverage higher core count nodes or take advantage of accelerators, scale their code to higher and higher node, core, or thread counts. Rogue Wave Software offers a powerful set of tools to aid in these efforts. ThreadSpotter pinpoints cache inefficiencies, educates and guides scientists and developers through the cache optimization process while TotalView provides scalable, bi-directional, parallel source code and memory debugging.**

Keywords-- Debugging

I. DEBUGGING AND OPTIMIZATION CHALLENGES

Scientists and developers engaging in high performance computing have a daunting task. They are faced with breaking down a substantial computational problem into various smaller units of computation that can proceed independently. In the simplest case, they have to think about how to break the data down, how that data is going to get transmitted to the site of the computation, the computation itself and then how it is going to get collected back and reassembled. Frequently the problem resists easy decomposition -- calculations may depend sensitively on inputs that are not easily broken down into discrete units. It is always possible to duplicate, share or fetch the data required, but that frequently introduces so much overhead that it begins to cut into the performance gains that parallelism promises to provide. Solving the problem of decomposition, so that it performs well, often requires taking a calculation that is easily expressed in a few formulas on a single sheet of paper and breaking it apart into what could work out to hundreds or even thousands of lines of code.

Problem set decomposition is only part of the challenge facing developers working on the latest generation of supercomputers. When we think about parallel programing in the abstract, we typically think about one kind of parallelism at a time – breaking work out across the multiple cores of a single node or across multiple nodes of a massive cluster. However, in the day-to-day work of developers who are striving to take advantage of hardware like the Cray XE6 or XK6, there are at least two, and sometimes many, architectural "dimensions" of potential parallelism in the system that scientists or developers need to consider when breaking their problem down. With an XK6, developers have a cluster with thousands of nodes. Each node can have up to 32 cores. Each processor is capable of vector computations. Tasks can be run on those cores using either process level parallelism or thread level parallelism. In addition, the XK6 boasts many NVIDIA GPUs, each of which is a collection of streaming vector multiprocessors capable of several hundred concurrent operations.

We sometimes talk about which form of parallelism is "best" for a specific problem. Developers who want to take full advantage of the power of leading-edge machines need to grapple with several types of parallelism at the same time, within the same computation. As a result the code that manages breaking down the tasks, moving data, and ensuring consistency is much more complicated than idealized MPI-only or OpenMP-only code examples.

Bugs can be introduced in any one of those thousands of lines of code. They can be logic bugs, typos, race conditions, deadlocks, mistakes in packing and unpacking data, or subtle numerical effects, such as, those that stem from changes in the way that tiny rounding errors accumulate in the system.

This set of challenges defines the world in which the scientist who needs to do supercomputing lives; and they are the reason that HPC developers demand the most sophisticated and powerful troubleshooting tools available. Rogue Wave provides the TotalView debugger in order to provide developers a way to easily see exactly what is happening inside their applications. TotalView supports all the different kinds of parallelism that the Cray XE6 and Cray XK6 can provide and shows how they work together in the application. Rogue Wave is committed to continue extending TotalView to work with the latest network and operating system technology from Cray, the latest processors from Intel and AMD, and the latest computational accelerating co-processors from NVIDIA and Intel. With accelerators rapidly becoming mainstream, GPU and vector debugging capabilities must now become a standard part of any debugger used in this environment. Memory is another critical resource and one that is easy to lose track of. A full-featured memory debugger must also become a standard feature of an HPC debugger.

Unfortunately, an HPC developer is not done once their application generates correct results. They need to make sure that their program scales up and makes efficient use of the computational resources it consumes to generate results. Rogue Wave offers a powerful cache memory optimization tool called ThreadSpotter. ThreadSpotter pinpoints opportunities in the code to take better advantage of cache memory and multi-core parallelism.

II. Debugging on the Cray with TotalView

This section will review some of the main capabilities that TotalView provides developers and scientists working with Cray supercomputers. It will specifically highlight features and capabilities that are new in the TotalView release 8.10 which Rogue Wave is launching at the Cray Users Group (CUG) 2012.

A. *Scalable debugging*

TotalView provides users with robust and rich parallel and multithreaded debugging functionality. TotalView integrates with the batch queue systems and MPI launcher programs, such as APRUN, and makes it easy to launch a large parallel program under the debugger while providing a single debugging session where developers can view and control their entire parallel application. TotalView can scale across thousands of nodes and can easily be attached to an already running job, even a hung job without any premeditation.

Sometimes bugs will only be visible with a very specific dataset or only happen above a certain scale. If this is the case, users may want to attach the debugger to a full-scale program without necessarily starting the debugger on every single process that is part of that application. TotalView supports this mode of use with a subset attach feature.

As mentioned above, modern HPC applications usually involve many forms of parallelism. The TotalView debugger handles the kind of parallel hybrid application that couples MPI multi-process parallelism together with OpenMP or Pthreads to provide multi-threading. TotalView allows developers to focus on any thread of their parallel task, supports the definition of both process and thread groups, and offers features like barriers, stepping and breakpoints that are optimized to work with parallel applications made up of sets of multithreaded processes.

B. *Accelerators*

Cray XK6 systems have nodes that are accelerated with NVIDIA Fermi GPU co-processors. HPC developers who want to take advantage of the favorable performance per watt and raw performance of those GPU accelerators need to adapt their application while carefully considering how to work this additional dimension of parallelism into their programs and generate special computational kernels that are compiled to run on the GPU. There are two ways to build such applications: either using the CUDA language extension or the Cray Compiler Edition OpenACC compiler directives. TotalView has full support for CUDA debugging with CUDA 3.2, 4.0 and 4.1. Rogue Wave has specifically tested TotalView's CUDA functionality in Cray XK6 environments.

OpenACC was announced at SC'11 as a new open standard for providing accelerator directives by a consortium of vendors including NVIDIA and Cray. At the time of the announcement there were not any OpenACC implementations available for users. Cray moved quickly to provide OpenACC compiler support and Rogue Wave has been able to provide Cray customers with a working solution for debugging applications compiled with Cray CCE 8's OpenACC feature in TotalView 8.10. The support is officially designated: "Early Access Release – Not Supported," but users are encouraged to test it and provide feedback while Rogue Wave does more testing and productizes this exciting new capability.

C. *Reverse Debugging*

One of the most complex things about troubleshooting is that by the time the program has crashed, hung, or generated an incorrect result, the program has already made whatever logic error that caused the crash, hang or incorrect result. The logic error causing the bug might be in the same routine as the point where the program hangs, crashes or outputs a recognizably incorrect result. However, more often the root cause is somewhere else in the code and has the effect of setting the program into an invalid state or corrupting some bit of internal data. That invalid state or corrupt data may appear to be harmless in the short term but later cause the program to misbehave in a recognizable way, which I will call the "fallout".

Since the error and the fallout may be widely separated, troubleshooting can quickly evolve into a very complex process of trying to work out a hypothesis of what the root cause might be based on the state of the system when it crashes or generates invalid data. Then the developer must come up with a way to drive the application (usually after a restart) to the precise point of execution where that hypothesis can be tested and the logic error pinned down.

Troubleshooting would be radically simplified if it were possible to capture the detailed trajectory of the program's execution and work backwards from the fallout to the cause - after the fact. TotalView now makes it possible for developers to apply just that approach. The ReplayEngine feature, available on the Cray XE and XK, records program execution for later replay. It then gives the user the ability to simply step backwards and forwards through the recorded history, watching the branches the program takes and which variables are changed as the program executes. This makes it possible to easily follow clues, such as corrupted data, that may exist at the site of the fallout and work backwards to whatever logic error put that corrupt data into place.

1) *ReplayEngine*

ReplayEngine works by recording, at a low level, the execution trajectory through the program and then guiding the program through that same trajectory again when the user wants to examine in detail any part of recorded history. ReplayEngine has been heavily optimized and relies primarily on recognizing the moments where the program encounters a non-deterministic input. For example, if the program reads from a network device, then the value that it reads depends on the state of the device and will almost certainly not be the same at some other arbitrary point in time. ReplayEngine stores such data for subsequent replay. When the user wishes to replay the program through that part of the execution trajectory ReplayEngine isolates the process and places it in a controlled sandbox, such that the network read doesn't actually happen. When the program plays back to the point where it wants to read from the network device it is told that the network device contains a copy of the data that was stored during record. Similar techniques are used around thread context switches, file IO, and system calls. Other tricks are used to create snapshots of program execution that can easily be employed as the starting point for replay operations. This careful orchestration of the process during recoding and replay happens "behind the scenes." From the developers' perspective, when they are using TotalView's ReplayEngine, they simply have the ability to run their program either forwards or backwards.

2) *ReplayEngine on the Cray XE*

Rogue Wave has made several important advances to our ReplayEngine technology since the last release of TotalView. First, it has been adapted it to work with the Cray Gemini interconnect (used in Cray XE and XK series supercomputers). This involved working closely with Cray to develop a technique to track the data flowing into an individual MPI process over the network. In the simplified example above I talked about the process of reading from the network device. However these kinds of explicit interactions between the process and the network device are a source of latency that Cray and other vendors actively work to reduce. Most vendors use zero-copy direct memory transfer technologies to reduce latency. These techniques write data directly into designated regions of the program memory without interrupting the program. Working with Cray we are now able to identify the active DMA regions of the program memory. ReplayEngine is now able to monitor the times when the program reads new data from those regions and store that data in its non-deterministic input log. ReplayEngine has similar functionality for the Mellanox and Voltaire Infiniband networks used on a variety of other Intel and AMD Linux-based clusters.

3) *Replay On Demand*

Secondly, Rogue Wave added the capability to activate ReplayEngine during the middle of a debugging session. Previous versions of TotalView had a limitation in that the ReplayEngine feature was an "all or nothing" decision that needed to be made at the outset of the debugging session. With TotalView 8.10, the flexibility has been increased such that developers can be in the middle of a debugging session and decide to begin recording execution for later reverse debugging. There is a new "activate recording" button that they can use to initiate recording at will. A frequent situation that customers want to be able to handle is that they have a program that does significant work to "initialize" data. They want to be able to run past initialization to a point where their program is starting to do something "interesting," from the perspective of the bug that is in question, and start the recording there. A simple way to do that with TV 8.10 is to fire up the program under TotalView, set a breakpoint where things start to get interesting, run the program (at full speed, no recording and no slowdown from the recording) to the point of interest. At this point recording can be enabled and the user can run from there to the crash. During this second stage the execution will be recorded with some amount of overhead involved. Then the user can very simply work backwards from the crash to the root cause within the recorded history.

D. *Memory Debugging*

TotalView includes a full-featured heap memory debugging tool called MemoryScape. MemoryScape gives HPC developers a view into how their program is making use of memory buffer space allocated with the malloc() interface. In C or C++ the programmer explicitly manages this heap memory. This requires programmers to be alert in order to avoid errors such as memory leaks and array bounds violations - especially when memory is being allocated and used in more than one thread. While this memory buffer space is managed directly by the Fortran runtime, Fortran programmers can still benefit from being aware of what heap usage is occurring and where memory is being consumed within the program heap. MemoryScape can directly detect memory leaks, even at the point where only a single allocation has been leaked. This advanced memory debugging can also be used to find heap memory buffer overruns (sometimes called array bounds violations), which otherwise subject programs to instability and random crashes. Finally MemoryScape is a very capable tool for optimizing memory usage. It can show a graphical display to tell how different processes of a parallel job are using memory. Then, for any specific process, it can break down memory usage by object file (library), source code file, function, line number, and function call backtrace. This allows developers to identify places in their program where they use more memory than anticipated and where the problem can be decomposed differently or some other trade off can be made to make more efficient use of memory.

E. *Three Ways to Debug on a Cray*

Users typically interact with Cray supercomputers through the batch resource management system such as OpenPBS. There are three ways that users can use TotalView in this context.

1) *APRUN*

The first way is to launch the program directly with APRUN from the interactive node. The downside to this approach is that the user has to issue the command and then wait for the job to be scheduled and the debugger to appear, which could happen quickly or slowly depending on the availability of the supercomputer.

2) *TVScript*

The second way is to plan a sequence of desired debugging operations and then submit a batch queue request that instructs TotalView in a non-interactive fashion. When the batch queue job is done, users receive a report with the results of the debugging operations. This approach makes use of the TVScript feature. TVScript is a simplified asynchronous “driver” script that takes a program and performs a series of debug operations (like “run to this breakpoint and print X”) on it. TVScript gathers the results of those debug operations and makes them available to the user through a trace output file.

3) *Remote Display Client*

The third way is to use the Rogue Wave Remote Display Client (RDC) to log into the supercomputer from a user’s desktop. The RDC is able to interact with the batch management system and handle the submission of job requests. The Remote Display Client workflow is similar to the interactive batch session method except that the RDC provides a secure and fast graphical connection between the supercomputer and the user’s desktop. This connection is designed for long distance, high latency networks, and can be used, for example, to run an interactive debugging session across the Atlantic Ocean.

III. Optimizing on the Cray with ThreadSpotter

Debugging isn’t the only challenge that Cray users face. They are also likely looking to extend their models with more comprehensive calculations (multi-physics), encompass larger problem domains, longer problem timescales, or increase the numerical resolution of simulations. Supercomputing is about solving problems that are too large to be calculated any other way - so almost by definition, datasets and runtimes are large. Therefore the best practice for an HPC developer is to spend some amount of their effort optimizing their application.

One of the challenges that users frequently confront on supercomputers is getting the data to and from the processor in an effective way. The processor can often consume and produce data at a bandwidth that is an order of magnitude higher than the bandwidth between the main memory and the processor. In order to effectively use the processor, a series of caches are utilized and frequently used data is placed there for easy access. The data transfer bandwidth between cache and core is sufficiently high that if the program can place the data that it needs in the cache, then the processor can perform arithmetic operations on each clock cycle and have data available for the next cycle.

Rogue Wave offers the ThreadSpotter cache memory optimization tool to help users quickly identify program bottlenecks that may be preventing them from taking advantage of available processing power. Because reading and writing main memory is more costly in terms of system power, optimizing cache memory usage can also reduce the power and cooling cost for the supercomputer, while improving performance.

ThreadSpotter can identify and prioritize a number of different kinds of problems for the user. It sorts issues in terms of how much impact they will have on performance, so that the user can start with the issues that have a higher reward first. For each problem identified, ThreadSpotter provides a statistical analysis, detailed information about what lines of code and ultimately what read or write instructions are involved. For each category of issue, ThreadSpotter provides helpful advice about how the issues can be addressed. Sometimes it will recommend changes to loop order; other times it might recommend breaking up data structures, or it might advise telling the processor to fetch the data from main memory without disrupting data already resident in the cache.

A. *Three Examples of Cache Memory Issues*

1) *Cache Utilization*

When memory is placed into the cache hierarchy, it comes in the form of cache lines. These are typically 128 byte chunks of memory, and a whole cache line will be fetched if any byte that is part of it needs to be read. In the ideal scenario, when a line is placed in the cache, each byte will be read many times before the cache line is flushed. In the least favorable scenario, the program might touch only one byte out of the 128, and then perform other actions that ultimately cause the cache line to be flushed without the other data contained on that line ever being touched. Poor cache line utilization can easily lead to a pronounced memory bottleneck. ThreadSpotter can identify locations where poor cache utilization happens, and happens frequently enough to have an impact on performance.

2) *Cache Reuse*

Each cache has a limited number of cache lines it can store at a time. Frequently, programs will be written in such a way that they repeatedly access a data set of a certain size. If that size is smaller than the cache, then the cache will generally be able to hold that memory, and the processor can proceed with very few cache misses. However, if the overall dataset is larger than the cache, then the order the operations occur become critically important. Generally, there will be multiple ways to arrange the calculation, some of which will place repeated accesses together, so that the cache can be highly effective at reducing the need for reading and writing to main memory. Other mathematically equivalent ways of structuring the calculation will spread repeated accesses out, causing much more traffic to the main memory. ThreadSpotter will highlight regions with poor cache usage and provide advice for how the programmer can approach improving the behavior.

3) *Cache Coherency*

The cache architecture is usually engineered to provide all of the cores within the processor with a consistent view of memory at all times. If there is more than one cache (as is almost always the case), then this requires a cache coherency mechanism. Cache coherency works but increases the overhead of memory access. If the same data is written to repeatedly from different cores, then many of those write operations may require the extra steps to ensure coherency, which will slow down the calculation. ThreadSpotter can identify locations in the code where such accesses are occurring frequently enough to have an impact on performance. ThreadSpotter can identify places where different cores are not touching the same data, but are nonetheless incurring coherency overhead. This can occur when different processor cores are updating two different data elements that happen to reside on the same cache line.

Because discovering bottlenecks by hand is labor intensive, it is especially valuable to have a tool, like

ThreadSpotter, available to quickly check the application and discover if poor cache optimization is an issue. If not, then the developer can move on and look at other forms of optimization such as load balancing, tuning MPI communication, etc.

IV. Conclusion

This paper highlighted some of the challenges facing developers working on Cray supercomputers and described two powerful tools that Rogue Wave provides to make such development easier (TotalView debugger and ThreadSpotter cache optimizer) . Recent improvements to the TotalView debugger on the Cray XE and XK environments include support for ReplayEngine on the Cray XE, support for CUDA on the Cray XK, and support for the OpenACC pragma-based language extensions in the Cray CCE.