Debugging and Optimizing Programs Accelerated with Intel® Xeon® Phi™ Coprocessors

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

***Abstract*— Intel® Xeon® Phi™ coprocessors present an exciting opportunity for Cray users to take advantage of many-core processor technology. Since the Intel Xeon Phi coprocessor shares many architectural features and much of the development tool chain with multi-core Intel Xeon processors, it is generally fairly easy to migrate a program to the Intel Xeon Phi coprocessor. However, to fully leverage the Intel Xeon Phi coprocessor, a new level of parallelism needs to be expressed which may require significantly re-thinking algorithms. Scientists need tools that support debugging and optimizing hybrid MPI/OpenMP parallel applications that may have dozens or even hundreds of threads per node.**

This paper will discuss how recent upgrades to TotalView® and ThreadSpotter™ are setting the stage for Cray users to adopt the Intel Xeon Phi coprocessor with confidence.

Keywords—Debugging, Optimization

I. INTRODUCTION

The Intel Xeon Phi coprocessor is a new environment in which Cray users may find themselves debugging and optimizing. The Intel Xeon Phi is an instance of what Intel calls the Many Integrated Core (MIC) architecture. As its name suggests the Xeon Phi is a many-core (rather than multi-core) processor and it sports >50 separate processor cores each of which is capable of executing up to 4 thread contexts. Programs can be written either to run directly on the Intel Xeon Phi coprocessor, treating it as a >50 core node, or on the host processor with which computationally intense code is offloaded to one or more Intel Xeon Phi coprocessor cards through the use of offloading extensions, such as Intel's LEO (Language Extensions for Offloading).

With Cray's adoption of the Intel Xeon Phi coprocessor as an accelerator on the Cray XC, users will be looking at a wide range of codes and considering the challenges of migrating applications to run on the Intel Xeon Phi coprocessor. This paper will review Rogue Wave's key debugging and optimization technologies in the context of the Cray XC and with respect to the Intel Xeon Phi processor.

The TotalView debugger supports source-code debugging of C, C++, and Fortran across the many nodes of Cray XT, XE, XK, and XC series supercomputers. Previous versions of TotalView are still supported on Cray X1 supercomputers. The paper will highlight how Rogue Wave has adapted TotalView to work with the Intel Xeon Phi coprocessor, which is offered on the Cray XC.

ThreadSpotter provides users with a detailed view of how their programs' execution interacts with the CPU cache, which resides between the main memory and CPU. Even relatively slow processors can perform many computational operations in the same amount of time that it takes to request one bit of data from the memory, therefore efficient use of the cache memory is vital in achieving high performance across a wide range of supercomputer architectures. This paper will briefly introduce ThreadSpotter and discuss work that is being undertaken to allow ThreadSpotter to analyze programs' cache memory on the Intel Xeon Phi coprocessor.

II. DEBUGGING ON THE CRAY XC WITH TOTALVIEW

A. TotalView for the Cray XT, XE, and XC

TotalView provides a powerful and intuitive graphical source code debugging environment for a variety of different supercomputing architectures including the Cray XT, XE, and XC. TotalView gives users control over and visibility into program execution.

TotalView provides users control over the program through a single debugger interface. Process and thread control features allow users to easily synchronize all the threads and to exert nuanced control over large parallel jobs. The debugger also provides exceptional capabilities for controlling thread execution. Breakpoints can be set with thread width so that users can more easily work with thread parallelism constructs, such as OpenMP parallel for loops.

TotalView features the ability to attach to an arbitrary subset of a parallel job and change that subset on the fly. TotalView gracefully handles MPMD parallel jobs – with automatically generated groups that span the entire job and other groups that operate only on the subsets that share executable images.

Variables and complex data structures can be examined and navigated with an intuitive variable display, data visualization, and exploration capability. This display capability makes type casting, working with pointers, and nested aggregate data types extremely easy and straightforward.

Since many scientific codes feature very important array-type data, TotalView provides a powerful array display. Arrays can be sliced and displayed using arbitrary striding using Fortran slice notation (even in C). Array data can be displayed in three ways.

1. As memory-ordered elements in list form;
2. 2D slices displayed in "spreadsheet" format; and
3. Represented graphically with line plots and surfaces.

TotalView excels in working with arrays of aggregate data types. The user interface features a "dive-in-all" capability that makes extracting numerical fields from array-of-aggregate type structures very easy.

Data abstraction with tools like C++ template libraries can be a great thing, but it can also serve to unintentionally obfuscate what is happening in a program when being debugged. Rogue Wave provides TotalView with automatic translation support for STL List, Map, Vect, and String classes. Support for Set, Multi-set, and Multi-map are in the 8.12 version, which is currently available for beta preview. These objects are transformed automatically into easy to work with array-, structure-, or array-of-structure type objects. Furthermore TotalView provides the user with the ability to transform their custom data types in the same manner.

1) *MPI debugging*

TotalView is integrated with the Cray **aprun** command. The user manual provides greater details, but for a high level overview, users can simply use **qsub** to create an interactive partition on the Cray system and then run:

TotalView aprun –a –n<num> a.out

The debugger queries **aprun** for information about all the MPI tasks that make up the mpi job and then attaches to all of them.

Rogue Wave Software is currently collaborating with specific customers on a scalability project. The project team members implemented a server tree network using the MRnet technology. The tree allows for scalable broadcast and reduction techniques to be used on communication between the debugger and debug agent processes.

This work is being done across three platforms: the IBM Blue Gene, Cray XE, and x86 based Linux + Infiniband. The MRnet infrastructure is already in place and users can receive a technical preview of TotalView that includes the MRnet capability by contacting the author. MRnet will be fully folded into the product with documentation in a future release.

Optimization of TotalView operational performance using the new infrastructure is ongoing. Rogue Wave is working with a range of different applications and tuning the debugger's performance with respect to those applications.

While a variety of different tests has been run at different scales and on different architectures, TotalView has already been able to debug more than 1 million threads on the Blue Gene/Q.

2) *Memory Debugging*

TotalView includes the MemoryScape memory debugger that gives users the ability to detect memory leaks, heap memory allocation overruns, and execute heap memory analysis and optimization. MemoryScape is integrated into TotalView and supports performing memory analysis across the many tasks of an MPI job.

3) *Reverse Debugging*

One of the most unique features of TotalView is its reverse debugging feature. Reverse debugging allows running the program backwards from the point where the failure appears to the root cause of that failure.

TotalView's reverse debugging feature is called ReplayEngine, which allows users to step backwards through the program's execution history utilizing a record and deterministic replay technique. As the program runs, the tool operates in a record mode in which program execution is recorded, with particular attention paid to non-deterministic inputs such as I/O, thread context switches, and operating system calls. If at any point the user wishes to see the previous state of the process, the tool arranges to place a synthetic unix process in that same state. It does this by creating a copy of the code and data state that was saved earlier and then re-executing the code deterministically along exactly the same execution trajectory that the program took during the record phase.

All of this is managed behind the scenes by ReplayEngine. The user interface simply shows "backwards step" and "backwards continue" commands that can be used to take the process back to earlier states. Once the process has been replayed to the desired state all the usual process and thread inspection capabilities are usable. Any variable or data (even those that the user did not previously know would be important) can be inspected during this replay process.

The deterministic nature of this replay process makes it especially helpful to track down hard to reproduce or intermittent bugs. These defects, which might otherwise take days or weeks to diagnose without TotalView, can sometimes be resolved within a single session with ReplayEngine.

4) *Scripting with TVScript*

TotalView is most often used for interactive graphical debugging, but it also is completely scriptable with a TCL based CLI. This can be used to automate repetitive tasks and drive completely non-interactive debugging sessions.

TVScript is a simple way to drive such non-interactive debugging sessions. It is a driver script, written in the TotalView TCL command line interface language, which takes a target executable and a set of instructions about where to set breakpoints and then drives the target program towards completion. TVScript has an event-action model. An event is triggered each time that a program hits a breakpoint. Other events occur when the program reaches certain other specifically defined states, such as program completion, segmentation violations, or memory errors.

TVScript driver program can take a variety of different actions in response to these events.

The most frequent action is to report information to a debugging logfile, which can be parsed after the fact to diagnose the behavior of the program. TVScript merges some of the benefits and conveniences of "print" style debugging with the power and capabilities of a powerful interactive debugger.

MemoryScape, ReplayEngine, and TVScript are fully supported for Cray XT, XE, and XC environments.

B. TotalView Support for the Intel Xeon Phi Coprocessor

Intel Xeon Phi coprocessors can be used either in an accelerator-like mode or in a mode that more closely resembles separately addressable multi-core nodes. The workflows are a bit different, as are the ways that the Intel Xeon Phi coprocessor appears in the user interface.

Each of those modes of use is briefly discussed below.

1) Treating the Intel Xeon Phi coprocessor as a hosted multi-core node

There are a couple of ways that users can utilize the Intel Xeon Phi coprocessor as a multi-core Linux node. The Intel Xeon Phi coprocessor runs a separate OS instance, usually a special version of Linux, from the OS running on the host node. It is usually possible to log directly into that Linux instance and run applications there. Code can generally be cross-compiled on the host system using the Intel compilers by specifying that the desired target architecture is MIC (the Intel compiler uses the -mmic flag for this mode). An application compiled this way can be manually run in the traditional fashion on the Intel Xeon Phi coprocessor.

Alternately, if a system is set up correctly users can utilize mpiexec to run a parallel job. Again, the job will be compiled with the -mmic flag and users may need to direct the MPI or resource management system to run the MPI tasks specifically on the Intel Xeon Phi coprocessors.

TotalView supports remote debugging and is compatible with a variety of MPI launcher programs. Rogue Wave needed to adapt TotalView to display the instructions and registers used on the MIC architecture. In addition, Rogue Wave compiled a variant of the TotalView debug server so that it would run on the Intel Xeon Phi coprocessor and to provide MPI and remote-system, cross-debugging support for MPI. TotalView's main executable runs on the host Linux-x86-64 environment when users are debugging Intel Xeon Phi coprocessors.

Users will need to ensure that their program executable is accessible from the host environment. They also need to confirm that TotalView is accessible from both the host processor and the Intel Xeon Phi coprocessor node. This software can be placed on a file-system volume that is mounted both on the host and on the Intel Xeon Phi coprocessor device. This is not an unusual configuration.

Then users need to verify that they can directly address the Intel Xeon Phi coprocessor device/s on which the processes that will be debugged are running. Users need to do this from the machine in which they are planning on running TotalView. A user's debugging session, right now, will be limited to the set of Intel Xeon Phi coprocessors that they can directly address from the host node. Rogue Wave is examining what it might take to support situations in which a cluster of host nodes exists, each of which can only address its own Intel Xeon Phi coprocessors.

The final condition is licensing. Those interested can contact Rogue Wave to find out if additional tokens are needed to debug on Intel Xeon Phi coprocessors.

If those conditions are met, users can simply direct TotalView to debug a remote process, which happens to be running on the Intel Xeon Phi coprocessor, or they can launch an MPI parallel job on the host machine

Launching TotalView to debug a native, non-MPI application uses a command line such as:

```
totalview -r <name> ./a.out
```

The -r flag simply tells the debugger that the target program is remotely running on the host named in the *name* parameter.

Launching TotalView to debug an MPI job uses a command line in the form of:

```
totalview -args mpiexec <mpiexec args> ./a.out
```

This is the same concept as with many non-Intel Xeon Phi coprocessor launch scenarios. Users are starting up mpiexec under TotalView and when the parallel job is launched an interface is used between the debugger and mpiexec so that mpiexec tells the debugger the location of the tasks that comprise the job. (Please see the TotalView documentation for more example launch strings.)

In either case the debugging experience that users receive on the Intel Xeon Phi coprocessor is very much what would be expected while debugging native code on the host processor. In the likely event that the program running on the Intel Xeon Phi coprocessor uses OpenMP or some other threading discipline, users will want to take advantage of TotalView's thread control capabilities.

2) Treating the Intel Xeon Phi coprocessor as an accelerator

The other way that users are encouraged to utilize Intel Xeon Phi coprocessors, especially when adapting large programs that already run on the host processor, is to take the already existing code and simply add directives that provide the compiler with hints on how to act if an Intel Xeon Phi coprocessor is present, then certain units of work could be offloaded to the coprocessor. These extensions are supported by the Intel compilers. Other similar models, based on language extensions such as OpenACC, OpenCL, and OpenMP, are being developed by a variety of vendors.

TotalView was extended to handle this situation by having the capability to recognize when the host program is dispatching a new routine to the coprocessor. When it recognizes that this is happening, users need to pause the application and start up a debug agent on the Intel Xeon Phi coprocessor. Behind the scenes, the debugging process is almost exactly like the native, non-mpi scenario outlined above, except that TotalView ends up attached to both a local process (the host process that dispatched the work to the Intel coprocessor) and a process running on the coprocessor (the one that resulted from the dispatch of work). The host process and offload process will show up in the debugger as separate processes running on distinct nodes.

The Intel compiler will generally create two copies of code designated to be offloaded to the Intel Xeon Phi coprocessor. One copy is compiled for the host Xeon processor and is used in the event that the code is run on a machine without an Intel Xeon Phi coprocessor. The other copy is compiled to run on the Intel Xeon Phi coprocessor. These two distinct function implementations (for two different processor architectures) are conceptually the same to the user.

TotalView recognizes this fact by ensuring that breakpoints are shared across both function instances. Therefore, users can set breakpoints on offloaded functions in the host process even before work is dispatched. When the program later runs and the function is loaded on the Intel Xeon Phi coprocessor, TotalView recognizes that these functions are the same and will automatically apply a corresponding breakpoint to the program running in the coprocessor.

In terms of features, there really is not a lot of difference between offload and remote native mode debugging with TotalView on the Intel Xeon Phi coprocessor, except that users may end up actively debugging both on the host and offload processes.

TotalView support for Intel Xeon Phi coprocessors was provided in an early access form in 8.11 and is a fully-available feature in 8.12, which is now in beta. The following describes the current level of functionality; however, this is an area where Rogue Wave is putting a fair amount of effort, so future versions are likely to have additional capability.

There are a few limitations for debugging with TotalView on the Intel Xeon Phi coprocessor. Two major features are not available: ReplayEngine and MemoryScape. MemoryScape should be delivered within the next few releases, but ReplayEngine support is yet to be determined.

Conditional watchpoints are not yet supported for the Intel Xeon Phi coprocessor. Users can utilize watchpoints and conditional evaluation points, but not yet conditional watchpoints. Finally, there are a few Intel Xeon Phi coprocessor specific instructions and registers that are not being disassembled and displayed correctly at this time. See the release notes and product documentation for details.

III. Optimizing on the Cray XC with ThreadSpotter

A. *ThreadSpotter for the Cray XE and XC*

ThreadSpotter is a CPU cache analysis tool that monitors the memory access pattern of a running program and generates a rich and detailed report that can be used to guide cache memory optimization. ThreadSpotter provides a more detailed analysis of programs' behavior than what is generally provided by hardware-counter driven techniques. A profiler will tell users where there is code that consumes wall-clock time without indicating why, similarly a hardware counter tool will tell users which lines are causing cache misses, but not why.

ThreadSpotter carries this analysis to the next level by showing users places in code where different categories of problems are occurring. It will, for example, flag sections of the code in which a heavily accessed data structure contains unused fields. It will clearly tell users if a significant fraction of the memory bandwidth is being consumed to move data into the cache simply because it lays adjacent to critical data.

1) *Cache Coherency*

Caches become involved in data movement between threads. Caches in multicore and many-core processors, such as on Xeon host CPUs and on the Intel Xeon Phi coprocessor ensure that data written to a given cache line (which represents a memory location) will be read when another core subsequently tries to read data associated with that same memory location.

This is done through a status field and a coherency protocol. Each processor has its own L1 cache and a shared L2 cache. If two threads running on different cores (cores A and B) read the same data at address X, then both A and B L1 caches will have copies of the cache line containing copies of the data at address X. If one of those processors, i.e. processor A, overwrites the value at address X, then the cache line which contains X will exist in A's L1 cache with the new value and it will have a status field that reflects that the cache now contains unique data. This in turn causes a message to be sent out to the other cache. This message tells the L1 cache belonging to core B that the cache data it has for the line containing X is now invalid. The only authoritative record of the value at X exists in L1's cache.

ThreadSpotter analyzes these situations and can highlight thread communication hotspots (in which there is a lot of coherency traffic on the cache), as well as more subtle effects, such as false sharing. False sharing can occur when two threads are not modifying the same data, but are actually modifying two distinct variables that simply happen to reside on the same cache line.

2) *ThreadSpotter and Multi-node Parallelism*

Since the CPU cache operates between the CPU and the main memory, ThreadSpotter is primarily designed to provide analysis of on-node performance issues. However, it is frequently useful for developers to use ThreadSpotter to try to understand cache behavior in HPC applications that run across multiple nodes of a cluster-style supercomputer. In order to support this, Rogue Wave has been working on improving the support for sampling and analyzing multiple processes.

The first stage of that work, which was delivered in ThreadSpotter 2012.2, was to provide scripts and documentation to support sampling of multi-node, distributed MPI applications on a variety of different architectures, including the Cray XE. Each instance of the ThreadSpotter sampler program is functionally distinct, and the challenge was to ensure that many of the samplers could execute at the same time without overwriting temporary files. Rogue Wave also made minor changes to make it easy to generate uniquely named sample files for later analysis. In addition, Rogue Wave provided documentation, with examples, of how ThreadSpotter is to be used with mpirun on a vanilla Linux cluster, apron, and either Torque PBS or Slurm.

The second stage of this work includes two components: First, Rogue Wave is implementing a distributed solution for launching and controlling the sampling mechanism using launchMON. This will give Rogue Wave a way to coordinate sample gathering across the entire parallel application. Second, Rogue Wave is developing and implementing a way to analyze thousands of distinct sample files to identify a small number of equivalence sets. These are sets of sample files that appear to share a lot of the same information. This will allow developers to effectively optimize cache - even with datasets that can only be run at large scale.

B. *Adapting ThreadSpotter for the Intel Xeon Phi coprocessor*

Rogue Wave is working to adapt ThreadSpotter for the Intel Xeon Phi coprocessor and is looking forward to demonstrating early versions of this work at ISC 2013. The work involves two distinct modifications. First, Rogue Wave adapted the sampling mechanism that ThreadSpotter uses to work natively on the Intel Xeon Phi coprocessor. Second, Rogue Wave extended the cache model that ThreadSpotter uses to be able to properly accommodate the Intel Xeon Phi coprocessor's cache architecture.

Adapting the sampler involved overcoming several different challenges. Originally designed for use on Xeon host processors, the sampler takes advantage of some vector instructions that are not included in the set of vector instructions available on the Intel Xeon Phi coprocessor. Rogue Wave is also investigating the implications of scaling up the sampler to be able to sample a large number of threads, which is expected to be common on the Intel Xeon Phi coprocessor. When a multithreaded program runs, there are data-structures in ThreadSpotter that need to be updated from more than one thread. While this is currently done via atomic operations, there are still scaling issues with data structure access that need to be overcome.

ThreadSpotter uses a statistical model of data flow through the cache architecture during program execution in order to perform analysis and issue detection. The Intel Xeon Phi coprocessor has >50 cores and each core has a private L1 cache and a shared L2 cache. The L2 caches work almost like a single shared L2 cache, in that information found in any core's L2 cache can be quickly transmitted to any core. However, unlike a more traditional shared L2 cache, this L2 cache has the property of allowing duplication. The same cache line might be duplicated across more than one of the L2 caches. ThreadSpotter needs to model this behavior to avoid over-estimating the effective capacity of the L2 cache.

IV. CONCLUSION

TotalView provides a seamless and polished Intel Xeon Phi coprocessor debugging experience on the Cray XC. ThreadSpotter is currently being enhanced for the Intel Xeon Phi coprocessor and will give users visibility into what is happening as data flows through the many-core coprocessor's unique distributed cache architecture. These tools allow Cray users to more easily and efficiently port and maintain scientific codes for the Cray XE, Cray XC, and for any machine accelerated with an Intel Xeon Phi coprocessor.