LLNL-TR-665868

LAWRENCE LIVERMORE NATIONAL LABORATORY

Characterization of UMT2013 Performance on Advanced Architectures

L. Howell

January 6, 2015

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Characterization of UMT2013 Performance on Advanced Architectures

Louis Howell

howell4@llnl.gov, phone (925) 422-6105
Lawrence Livermore National Laboratory, Livermore, CA

Abstract

This paper presents part of a larger effort to make detailed assessments of several proxy applications on various advanced architectures, with the eventual goal of extending these assessments to codes of programmatic interest running more realistic simulations. The focus here is on UMT2013, a proxy implementation of deterministic transport for unstructured meshes. I present weak and strong MPI scaling results and studies of OpenMP efficiency on the Sequoia BG/Q system at LLNL, with comparison against similar tests on an Intel Sandy Bridge TLCC2 system. The hardware counters on BG/Q provide detailed information on many aspects of on-node performance, while information from the mpiP tool gives insight into the reasons for the differing scaling behavior on these two different architectures. Preliminary tests that exploit NVRAM as extended memory on an Ivy Bridge machine designed for "Big Data" applications are also included.

Introduction

Modern computer architectures pose major challenges for simulation code development, not only requiring large-scale parallel scalability but also increasing attention to on-node parallel efficiency and memory management. Efforts are under way at many sites to better understand how codes of programmatic interest behave on advanced hardware both in order to make the implementations of key algorithms more efficient and to guide procurements of new machines. A team at LLNL has studied three proxy applications that include simplified implementations of important algorithms. This paper focuses on the results for UMT2013 [1], a proxy app for deterministic radiation transport on an unstructured mesh. (The other two proxy apps in the study, not presented here, are the Monte Carlo tracking code MCB and the algebraic multigrid code AMG2013.)

Most of the results presented here are for the IBM BG/Q architecture. The runs were done on Vulcan, part of the LLNL Sequoia procurement. The hardware counters on BG/Q provide detailed information on memory bandwidth, cache utilization, instruction mix, and other aspects of on-node performance [2]. For comparison, runs were also done on the Cab machine, a Linux cluster using

8-core Intel Xeon E5-2670 Sandy Bridge processors with InfiniBand QDR interconnect. Cab is part of the TLCC2 procurement. The mpiP [3] and memP [4] tools were used on both machines to give detailed information about MPI efficiency and heap memory use. One additional test used Catalyst, a Ivy Bridge based machine with unusually large memory (128GB per node) and an additional 800GB of SSD NVRAM per node. Software support for using the SSD memory without major code modifications has recently become available, but test results using the SSD memory show disappointing performance.

None of the performance tools were perfect. I will mention some difficulties with the tools in the context of the individual tests, and will summarize some lessons learned in a short section at the end of this report.

Weak Scaling

A weak scaling test measures parallel performance in a series of runs where the size of the problem increases in proportion to the number of processors used, so the problem size per processor is held fixed. On Vulcan UMT2013 showed fairly good weak scaling to at least 131072 cores. At the largest scale the speed per iteration was 67% of that at small scale. (There was some additional slowdown due to the fact that the code required more iterations to converge at large scale. There were also issues with the scaling of the measurement tools themselves.) Results on Cab were similar but were only run out to 4096 cores. The parallel efficiency for the largest runs on Cab was comparable to that of the largest runs on Vulcan, though the latter used 32 times as many cores (and 128 times as many threads)!

In UMT2013 large-scale MPI parallelism is over 3D spatial domains. This discrete ordinates (S_n) code for multigroup deterministic transport [5, 6] also allows user control over the number of energy groups and the number of angles used in the discretization, and OpenMP threading is used over the angles. (The angles represent the directions in which photons are moving. In any spatial zone there can be radiation moving in many different directions with different intensities, and the set of angles can be considered a discretization of these directions over the 2D surface of the unit sphere. Combined with the 1D spectrum of photon energies and the 3D spatial discretization the complete system is then 6 dimensional, plus one more for time. Since intensities must be stored for every combination of zone, angle, and energy, it is no surprise that memory management is a major consideration for this code.)

On Vulcan I did two series of weak scaling tests, one using $10 \times 10 \times 10$ and one using $12 \times 12 \times 12$ spatial grids per MPI task. Both problems used 16 energy groups and 256 angles. The runs were done with 8 MPI tasks per BG/Q node and 8 threads per task. Vulcan has 16 cores per node with 4 hardware threads per core, so this configuration made full use of the hardware threads. On Cab I ran the series with $12 \times 12 \times 12$ grids, using 16 MPI tasks per node and 1 thread per task. Cab has 16 cores per node and no hardware threads. The reason for running half as many MPI tasks per node on Vulcan was that this machine has only 16GB of memory on each node, compared to 32GB for Cab; running 16 tasks per node on Vulcan would require smaller grids per task. The runs on Vulcan used roughly 7.5GB per node for $10 \times 10 \times 10$ grids and 12.5GB for $12 \times 12 \times 12$, as measured by memP. The Cab runs used 26.4GB per node. (Comparisons exploring the performance effects of OpenMP threading in detail appear in later sections below.)

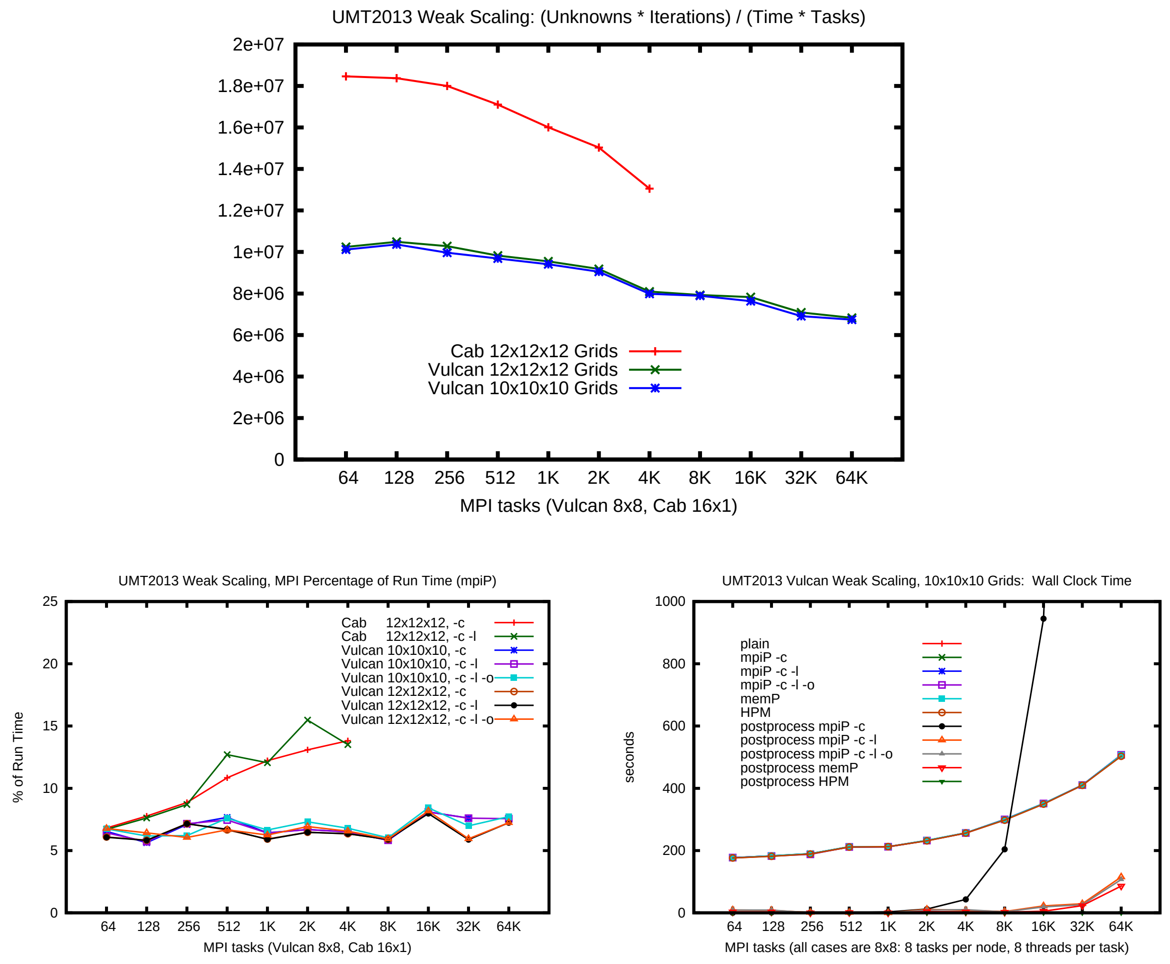

Figure 1: Weak scaling study. Top: Figure of merit. Bottom Left: Percentage of run time tied up in MPI communication. Bottom Right: Wall clock times showing overheads caused by the performance instrumentation.

Runs were instrumented using the mpiP tool [3] for MPI profiling and the memP tool [4] to measure heap memory use. Figure 1 shows the performance results in three different ways. First we look at the figure of merit, a metric that focuses on the rate at which basic components are executed and factors out the numbers of iterations required for convergence. (The iteration count is also an important component of overall runtime, but our main concern here is with the performance of this *implementation* on particular architectures, as opposed to the mathematical properties of the algorithm.) Perfect weak scaling would yield flat lines. The Vulcan (BG/Q) results are not perfect but are still fairly good. The Cab (TLCC2) results show a steeper drop in performance at scale, even on the much smaller number of available processors, though individual processors are faster than those on BG/Q. The difference between the $10 \times 10 \times 10$ and $12 \times 12 \times 12$ tests is negligible.

The second plot (bottom left) shows the percentage of time spent in MPI communication as measured by mpiP. Various mpiP options were tried. The `-c` option combines results from all MPI tasks into a single report, `-l` switches to a different algorithm for combining information from different tasks, and `-o` turns off profiling from the beginning of execution, instead relying on explicit calls I put in to activate the instrumentation only during the main timestep loop. The `-o` test did not work on Cab because the activation calls failed to turn on the MPI profiling. (On the TLCC2 machines UMT2013 builds using dynamic libraries, and apparently picks up the wrong version of the activation routines when linking with mpiP.)

The MPI profiles on Vulcan show a small and roughly constant percentage of time going to MPI communication, independent of grid size and the mpiP options. Cab, on the other hand, shows an increasing amount of time going to MPI as the number of tasks grows. Apparent differences between the two curves generated for Cab are not repeatable and therefore not really caused by the different mpiP options tested for this plot. All timing results on Cab vary more from one run to the next than those measured on Vulcan. I will demonstrate this in more detail in the strong scaling section below.

The third figure (bottom right) looks closely at the costs of running the instrumentation itself. All runs are for the $10\times10\times10$ series of tests on Vulcan, with separate curves for the code without instrumentation, with the various mpiP options, with the memP memory profiling tool, and with the HPM (mpitrace) library for gathering information from the BG/Q hardware performance counters. (I will show results from the HPM library in a later section.) These are wall clock measurements, and a major component of the upward trend for the six main curves is the increase in the number of iterations required for convergence at larger problem sizes. All six curves for the main code lie exactly on top of one another, which shows that the tools are not significantly perturbing these timing measurements. Where the tools do show a significant cost, though, is in the post-processing phase after the main part of UMT has finished. Post-processing costs for mpiP skyrocket without the `-l` option. (Memory costs for this phase also rise dramatically. Tests for the series with $12\times12\times12$ grids were closer to filling memory and often crashed without `-l`.) The post-processing costs for mpiP with `-l` are smaller but are also starting to rise for the largest runs; the same is true for memP. The overheads added by HPM library were exceptionally low compared to those from the other tools, both during the collection phase and during postprocessing.

Strong Scaling

A strong scaling test measures parallel performance with the total size of the problem held constant. In UMT2013 MPI parallelism is over 3D spatial domains, so as the number of processors increased I reduced the spatial grid size on each processor to keep the total number of unknowns unchanged. The tests shown in Fig. 2 used grids varying from $12\times12\times12$ down to $3\times3\times3$. Users also have control over the number of angles (transport directions) and energy groups, but for this test I held these constant. Note that OpenMP threading is over the angles in each octant and so is independent of the spatial grid size.

Both Vulcan and Cab have 16 cores per node, but on Vulcan there are also 4 hardware threads per core. OpenMP threading allows access to these hardware threads and can also be used for running multiple cores per MPI task. All of the Cab runs shown here used 16 MPI tasks per node and 1 thread per task (16×1), but the Vulcan runs were done in three groups: a (16×1) series was similar to the Cab runs but did not use all the hardware threads, a (16×4) series was faster because it did use all the hardware threads, and an (8×8) series was faster still because it used twice as many cores as the other two. The large spread between the different Vulcan curves is not a surprise

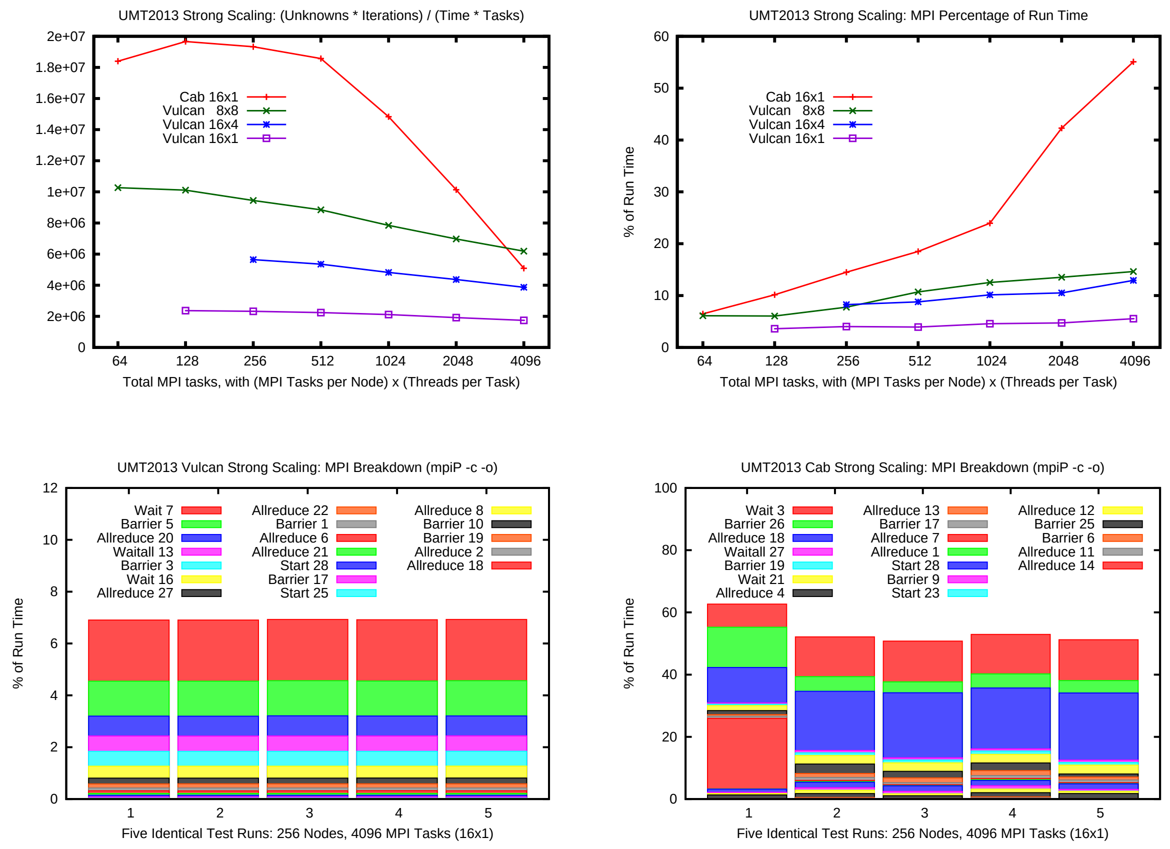

Figure 2: Strong scaling runs on Vulcan (BG/Q) and Cab (TLCC2), with the Vulcan results also demonstrating reasonably efficient use of OpenMP threads. Top Left: Figure of merit. Top Right: Percentage of run time tied up in MPI communication. Bottom: Comparison showing five identical runs of the largest 16 x 1 problem on each machine, with MPI time during the timestep loop broken down by callsite. The callsites are arranged in the same order for both runs even though mpiP identifies them with different labels. Note the different scale and greater variability for the tests on Cab. Each series of five tests was run with a single script and therefore used the same processors under similar runtime conditions.

since these tests used different machine resources: different numbers of cores and hardware threads. For a comparison showing the effects of allocating the same resources in different ways see the next section.

The top left plot in Fig. 2 shows the figure of merit for the central timestep loop: with perfect strong scaling the lines would be flat. Cab is faster for small runs but shows much worse parallel scaling than Vulcan. The top right plot shows how much time in each run was spent doing MPI communication, as reported by mpiP. (These mpiP measurements are for the whole code, not just the central timestep loop, but that doesn't make a major difference here.) The obvious conclusion is that Vulcan has slower processors but a better interconnect. The bottom two plots, though, show that there is another factor involved that may be even more critical at scale. Times spent in each

MPI communication step are much more variable on Cab than on Vulcan from one run to the next. Operating system noise is a likely explanation: when interrupts occur during an MPI collective the processors that are delayed keep all the others waiting. Contention for network resources with other jobs running on the machine is another contributing factor. The BG/Q network is better at isolating jobs from each other.

These MPI breakdown plots use the `-o` option to mpiP in order to filter out some variability that happens during initialization and focus on the main timestep loop. To get `-o` to work on Cab I changed the build to use static libraries. Even though mpiP uses different numbers to identify MPI callsites on the two different machines, I have arranged the figures so that corresponding callsites appear in the same order and can be compared directly.

Threading and Hardware Performance Counters

The examples in the previous section showed that OpenMP threading improved performance on Vulcan for a fixed number of MPI tasks by using additional cores and hardware threads per task. It is a more complicated question whether MPI or OpenMP threads provide the more efficient use of a fixed allocation of hardware resources. This section explores the tradeoff between MPI and OpenMP parallelism, and also highlights some of the detailed information available from BG/Q hardware performance counters that helps interpret the differences between tests.

In the top row of Fig. 3 I show the performance for 64 nodes of Vulcan and of Cab as the tradeoff varies from all-MPI to all-OpenMP on each node. Though there are 16 cores per node on each machine, the 64 hardware threads on each BG/Q node can be accessed in any MPI/OpenMP combination from (64×1) to (1×64). On Cab the combinations vary from (16×1) to (1×16). The figure of merit used for these plots is slightly different from the one used in the scaling studies: since the amount of machine hardware is held constant I do not divide by the number of MPI tasks. Also note that because each test keeps the total number of mesh zones constant, but varies the number of MPI tasks, the size of the spatial domain handled by each MPI task is changing as we move from left to right across each plot.

Several runs are shown with variations in spatial grid size and numbers of angles and energy groups. (The grid sizes shown in each legend are the ones corresponding to 8 MPI tasks per node. So on Vulcan a line labeled "$12\times12\times12$" actually uses grids that vary from $3\times3\times3$ to $24\times24\times24$ per MPI task as the number of tasks changes.) High figures of merit in these plots tend to result from using large numbers of energy groups. All energy groups share the same geometric quantities associated with a given combination of zone and angle, so in UMT2013 the loop over energy groups is the innermost loop. With more groups the geometric calculations are amortized over more unknowns. The lowest curve is the one using only three energy groups. This is also the flattest curve, though. It runs more efficiently with large numbers of OpenMP threads because it has a large number of angles to thread over. The other curves show more of a drop in performance for large numbers of threads because they don't have enough angles per octant to keep all the threads busy.

The results for Cab show a sharper drop in performance for the (1×16) mode, even in the cases with large numbers of angles that perform well with up to 64 threads on Vulcan. There are two sockets on each node of Cab. Threading works well across the 8 cores on each Xeon chip, but

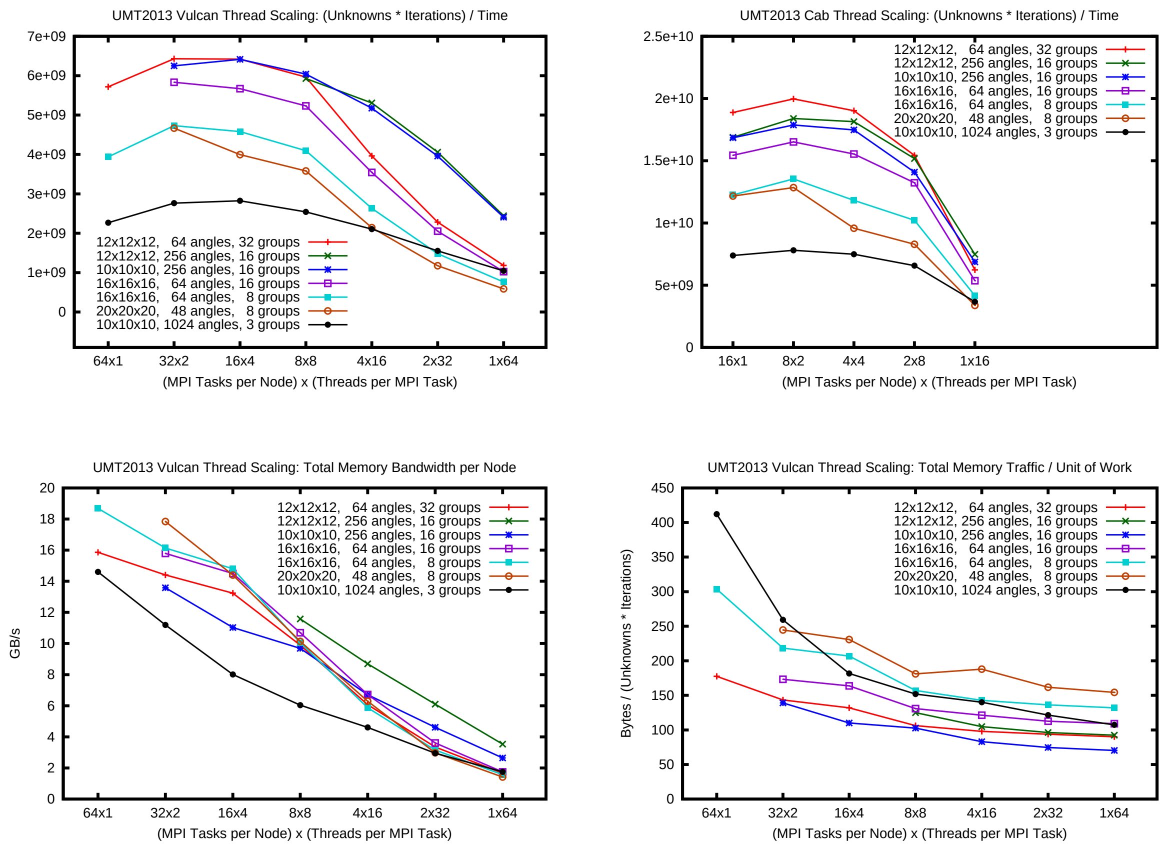

Figure 3: Performance for a fixed number of processors, showing changes as the code shifts from pure MPI parallelism to heavy use of OpenMP threading. Top: Figures of merit on Vulcan and on Cab. Bottom Left: Memory bandwidth per node on Vulcan. Bottom Right: Memory traffic per work unit.

threading across the entire node is much slower due to low bandwidth between the two sockets.

The second row of Fig. 3 shows the use of memory bandwidth for UMT2013 on BG/Q, as reported by the HPM (mpitrace) library [2]. The total bandwidth available on a BG/Q node in practice is about 28 GB/s. A previous version of UMT was bandwidth-limited, but this was changed by a revision in the data structure layout in UMT2013. None of the test cases here uses more than 19 GB/s. There is a strong and somewhat puzzling dependence on the number of threads per MPI task, with pure-MPI cases using the most bandwidth. To attempt to explain why bandwidth use declines so consistently with increasing thread count, the plot at lower right looks at memory bandwidth divided by the figure of merit (with a constant scaling factor). The result is the number of bytes of memory traffic required to execute each iteration of the algorithm, per unknown. The downward trend in these curves for small numbers of threads can be interpreted as different threads sharing data fetched into cache. For larger numbers of threads, though, the curves flatten out, showing that the continued decrease in memory bandwidth is tied to the decrease in the figure of merit: the code draws less memory simply because it is doing less useful work. The apparently steady decline in memory bandwidth is thus seen to be a combination of two different effects for

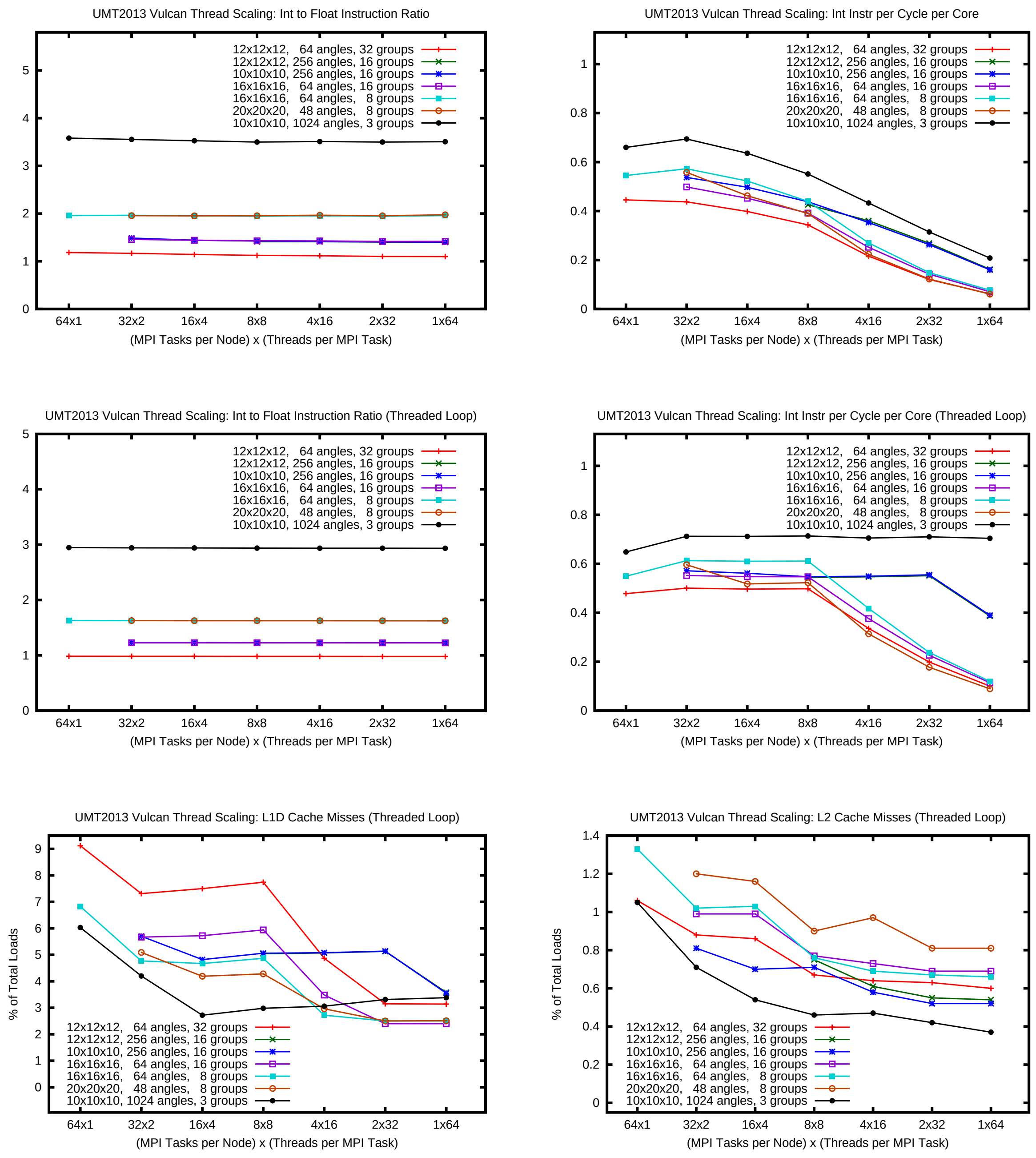

Figure 4: BG/Q hardware performance counter information showing effects of shifting from pure MPI to OpenMP for a fixed number of processors. Top: ratio between int and float instructions, and int instructions per cycle, measured over the full timestep loop which includes some unthreaded code. Middle: same quantities measured only for the threaded loop over angles. Bottom: L1 and L2 cache miss rates for the threaded angle loop, which help explain why some test cases achieved higher instruction rates than others.

different parts of the curves.

As a separate issue, the memory footprint was also higher for cases with more MPI tasks, even though all tests on each curve have the same number of unknowns. Missing points on some of the curves were runs that failed due to insufficient memory. All of these missing points are on the left sides of the various plots. I'm not including plots of memory usage because the results are somewhat irregular. The memP tool does not appear to be thread-safe, and becomes more and more likely to return corrupt results as the number of threads increases. There is also a heap memory estimate returned by mpitrace, but this is consistently higher than the memP estimate even in cases where memP appears to be functioning correctly. It appears that that mpitrace reports the heap memory allocated to the process by the operating system, while memP reports the amount actually used by the application.

The "overloaded" modes (64×1) and (32×2) on BG/Q use multiple MPI tasks per core. It is surprising that these perform even as well as they do. The heavy use of both memory and memory bandwidth in these modes, though, indicates that they do not make effective use of machine resources and will not be practical for most production calculations.

The six plots in Fig. 4 show more information about these same runs obtained using the BG/Q hardware performance counters. The top two plots were derived from measurements over the full timestep loop, which includes both threaded and unthreaded code. The ratio between integer and floating point instructions does not depend at all on the number of threads per task but only on the number of energy groups. Integer instructions here are mostly indexing operations: striding through the mesh. All of the test cases use at least as many int as float instructions. The highest floating point usage occurs with large numbers of energy groups because these cases have more work per mesh zone (recall that the loop over groups is the innermost loop). A BG/Q core can perform at most one int and one float instruction per cycle, and it is unusual to see rates much above 0.7 in practice. The plot on the right shows that the int instruction rate can reach a large fraction of the maximum rate when the number of threads is small. The integer instruction rate appears to be a primary limiting factor for UMT2013.

The second row of plots shows the same quantities measured only for the threaded loop over angles. The integer instruction rate is instructive. Each downturn in each of the curves happens when the number of threads exceeds the number of angles per octant. Without this effect the curves would be essentially flat. The plot above this one, that measures the same quantity over the full timestep loop, shows a more steady decline in the instruction rate. This decline is therefore associated with the unthreaded code outside the angle loop, which can dominate the run time when the number of threads is large.

The final two plots in Fig. 4 give L1 and L2 cache miss rates as a fraction of loads. These help explain the remaining differences between integer instruction rates for the various test cases. Some features of the instruction rate curves are due to idle threads, as discussed above. The remaining differences correlate with cache miss rates, and tests with higher instruction rates generally showed lower cache miss rates. The lowest cache miss rates were for the test problem with the fewest energy groups. This is the case that did the most integer operations per floating point operation, reached the highest integer instruction rates, and drew the least bandwidth from main memory.

There are many other counters available in addition to the ones used for the figures. Almost no

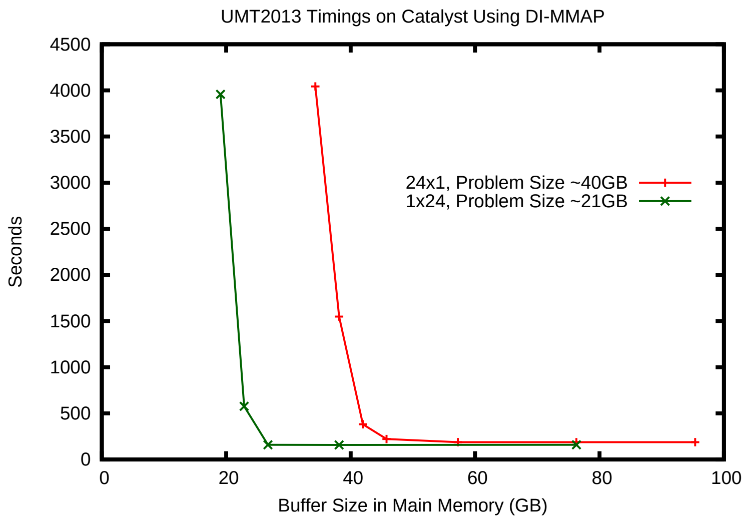

Figure 5: Performance on a single node of Catalyst as a function of the DI-MMAP buffer size in main memory. When the buffer size is too small, allocated pages migrate to NVRAM and execution time goes up. The large increases in run time observed for small buffers with both MPI and OpenMP parallelism show that the current implementation is not effectively managing memory for UMT.

SIMD instructions are generated by UMT2013, and none at all in the threaded loop over angles where most of the work is done. If the code were modified to make better use of the SIMD units—perhaps through explicit directives—then the SIMD counter information would give valuable information about the effects of the change. The instruction cache miss rate turned out to be an issue for one of the other apps (MCB). UMT2013 has a very low rate of instruction cache misses, though, so this is not a performance issue worth exploring here.

Using NVRAM to Supplement Main Memory

The Catalyst machine at LLNL is similar to the Sandy Bridge TLCC2 machine Cab but uses 12-core Xeon E5-2695 v2 Ivy Bridge processors. Its most distinguishing feature is large memory (128GB per node) and an additional 800GB of SSD NVRAM per node. Given that UMT with its six-dimensional discretization requires storage of large amounts of data, and that future architectures may provide less memory per processor and may require use of complex hierarchical memory systems, it was interesting to look at whether UMT could use the SSDs on Catalyst as a supplement to main memory. (Other Lab codes have begun to do so [7].)

The software package DI-MMAP, currently under development at LLNL [8], allows pages on the heap to migrate between NVRAM and main memory as needed. A buffer in main memory holds the currently resident pages. If this buffer is large enough to contain the entire heap, then the application runs nearly as fast as it would normally (measurements with UMT2013 showed a slowdown of around 15% compared to the same tests run without DI-MMAP). If the heap is too large for the buffer then some pages will only exist in NVRAM at any given time and may be swapped into the buffer as needed.

The obvious approach was to allocate a buffer that nearly fills main memory on a node and then experiment with UMT2013 test problems larger than the buffer. This turned out to be more difficult than expected. UMT2013 is an idealized proxy app, and though it scales well to large numbers of MPI tasks it is not set up to handle unusually large memory sizes per task. The grid generator scales very poorly with increasing zone count per spatial domain. Though this initialization time is not counted as run time for the application itself, it quickly becomes prohibitive—hours to initialize a test that would take minutes to run. There are also built-in limits on the numbers of angles and energy groups. Rather than try to modify UMT to enable larger problem sizes, I decided to frame the test a different way. After all, dealing with a large main memory was not the point of the exercise.

Figure 5 shows timings for two series of tests, each one holding the problem size fixed while varying the size of the buffer allocated in main memory. This not only avoids the need to construct very large test problems, it also allows us to estimate the costs of swapping pages to NVRAM without the complication of changing other computational costs at the same time. Both test problems ran on a single node of 24 cores. One used 24 MPI tasks, the other used 24 OpenMP threads (though the code was built with support for MPI+OpenMP in both cases).

The run times approach a constant value for all buffer sizes large enough to contain the entire allocated heap. As the buffer becomes smaller than the heap size, though, run times increase dramatically. Even small amounts of memory moving to and from NVRAM can use more time than the rest of the algorithm combined.

This is a disappointing result and has not yet been explained. On the bright side is the fact that DI-MMAP worked at all for this code. This its first use with MPI, and DI-MMAP developer Brian Van Essen had to make significant modifications in order to get it running. He believes there may be an interaction with MPI that causes pages to migrate more than necessary, so improvement in future versions of DI-MMAP is possible. On the other hand, the access patterns used by UMT2013, such as the fact that the code accesses almost all of its memory every timestep, may render this sort of automatic approach impractical. It may be necessary to modify the code in order to explicitly stage access to memory before it is needed, much as MPI codes can be structured to overlap communication and computation.

Performance Tool Lessons Learned

The performance tools gave useful information but did not always work well—some of the problems were outright bugs. Some issues were described in more detail in earlier sections. Here is a quick summary of difficulties encountered:

- Building UMT with dynamically-linked libraries caused trouble for both mpiP and memP, though many features could be made to work.
- mpiP post-processing does not scale well without the `-l` option.
- memP can give incorrect heap sizes with run with threads or when run on Catalyst even without threads.

- memP shows the heap size used by the program, which may be significantly smaller than the total size of the memory pool allocated to the process by the operating system.
- memP never worked for MCB.
- At high instruction cache miss rates some derived quantities returned by the HPM library on BG/Q became unreliable with the default counter group 0. Using `HPM_GROUP=8` avoided this problem by explicitly including instruction cache misses.

The last two items were issues for MCB, not UMT, but are included for completeness. (UMT has a very low rate of instruction cache misses.)

Conclusions

This is very much a work in progress, and the conclusions we can draw from study of UMT2013 alone can only be tentative. Further work will require comparison with other proxy apps and with production codes. We must also understand how codes behave on additional architectures with accelerators and more complex memory hierarchies. Nevertheless a few interesting results are apparent from these UMT experiments alone.

MPI scaling was generally good. This is not surprising. Codes and algorithms have been designed, studied, written, and rewritten for distributed-memory parallel computers for quite a few years now, not just in academic research but in the lab environment that produced UMT. The parallel machines themselves have also gone through several generations of evolution to better support programmatic applications. The significant differences between the machines examined here highlight the importance of consistency in node and interconnect performance for large-scale bulk-synchronous applications. A conclusion is that either that consistency must be maintained in future architectures, or the largest-scale applications will have to move away from bulk-synchronous programming models in order to adapt better to a changing runtime environment.

The results for on-node performance with OpenMP show that the current UMT threading model is effective only for small numbers of threads. UMT2013 may make more efficient use of memory bandwidth than earlier UMT versions, but it is still near the bandwidth limit BG/Q. Cache performance is reasonable but differences in cache miss rates may explain some of the performance differences among various test cases. Both threading and memory management are likely to drive future developments in code and compiler design; these issues appear to be less settled than MPI parallelism.

The high ratio of int to float instructions shows how integer instructions cannot be considered an afterthought for complex codes, even for a mesh-based code like UMT2013. Integer instruction rate was the most important factor limiting performance. Future architecture choices should take this into account, and not focus too much on features such as floating point vectorization that may only benefit a limited set of codes. (Other proxy apps used more complex data structures and had even more lopsided int to float ratios than UMT.)

Finally, on the issue of how to use NVRAM and other new additions to the memory hierarchy, we have only scratched the surface. Preliminary experiments shown here were not promising, and suggest that major shifts in algorithm design may be needed.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] `https://asc.llnl.gov/CORAL-benchmarks/`.

[2] Walkup, R. E., "MPI Wrappers for BGQ," IBM Advanced Computing Technology Center, Yorktown Heights, NY, unpublished document available on BG/Q systems (2013).

[3] Vetter, J., and Chambreau, C., "mpiP: Lightweight, Scalable MPI Profiling," `mpip.sourceforge.net` (2014).

[4] Chambreau, C., "memP: Parallel Heap Profiling," `memp.sourceforge.net` (2010).

[5] Nowak, P. F., and Nemanic, M. K., "Radiation Transport Calculations on Unstructured Grids Using a Spatially Decomposed and Threaded Algorithm," in *Proceedings of the International Conference on Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Madrid, Spain, September 27–30, 1999, Vol. 1, p. 379.

[6] Nowak, P., "Deterministic Methods for Radiation Transport: Lessons Learned and Future Directions," *ASC Workshop on Methods for Computational Physics and Modern Software Practices*, March 2004, Lawrence Livermore National Laboratory report UCRL-CONF-202912.

[7] Van Essen, B., Pearce, R., Ames, S., and Gokhale, M., "On the Role of NVRAM in Data Intensive HPC Architectures: an evaluation," in *IEEE International Parallel & Distributed Processing Symposium* (2012).

[8] Van Essen, B., Hsieh, H., Ames, S., Pearce R., and Gokhale, M., "DI-MMAP–a Scalable Memory-map Runtime for Out-of-core Data-intensive Applications," *Cluster Computing* (2013).