Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)
La Cité des Sciences et de l'Industrie, Paris, France, October 27–31, 2013

Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes

John R. Tramm[1] and Andrew R. Siegel[1]

[1]*Argonne National Laboratory, Center for Exascale Simulation of Advanced Reactors, 9700 S Cass Ave, Argonne, IL 60439*

We have extracted a kernel that executes only the most computationally expensive steps of the Monte Carlo particle transport algorithm - the calculation of macroscopic cross sections - in an effort to expose bottlenecks within multi-core, shared memory architectures.

KEYWORDS: Monte Carlo, Neutron Transport, Reactor Simulation, Cross Section, Benchmarks, High Performance Computing

I. Introduction

Current and next generation processor designs require exploiting on-chip, fine-grained parallelism to achieve a significant fraction of theoretical peak CPU speed. The success or failure of these designs will have a tremendous impact on the performance and scaling of a number of key reactor physics algorithms run on next-generation computer architectures. One key example is the Monte Carlo (MC) method for neutron transport. MC methods are characterized by complex memory access patterns that heavily tax shared resources of multi-core memory hierarchies. In this analysis we study in depth the on-node scaling properties and memory contention issues of MC particle transport specifically for reactor physics calculations.

There has been significant research into the performance and scaling of MC particle transport algorithms on distributed memory, High Performance Computing (HPC) systems.[(1)(2)] One such effort, the *OpenMC* transport code,[(1)] has investigated scaling on many-node distributed memory architectures, such as Blue Gene/P. However, there is little reported on the performance of such applications on multi-core, shared memory architectures. At least one recent study does provide a comprehensive view of on-node scaling behavior at the algorithmic level but does not go into great depth on the underlying architectural causes of scaling degradation.[(3)] This is notable as there are significant memory contention issues in multi-core scaling that are not present on distributed memory architectures. Typical[(3)] multi-core scaling for the MC particle transport algorithm is shown in Figure 1.

To investigate scaling and performance issues of robust, quasi-static nuclide depletion calculations (i.e., where hundreds of nuclides are present in the fuel region and performance is dominated by macroscopic cross section calculations), such as are performed by *OpenMC*, we abstract a key computational kernel that is responsible for the majority of the algorithm's runtime and implement it in the form of the "proxy application" *XSBench*. The end result is that the essential computational conditions and tasks of fully featured MC transport codes are retained in the kernel, without the additional complexity of the full application. This provides a much simpler and more transparent platform for isolating where both hardware and software bottlenecks inhibit scaling of the algorithm. We then use and modify our extracted kernel to identify low-level hardware and software bottlenecks on an Intel Xeon system, so that we can make an intelligent prediction as to how the MC transport algorithm will scale on next generation, many-core systems.

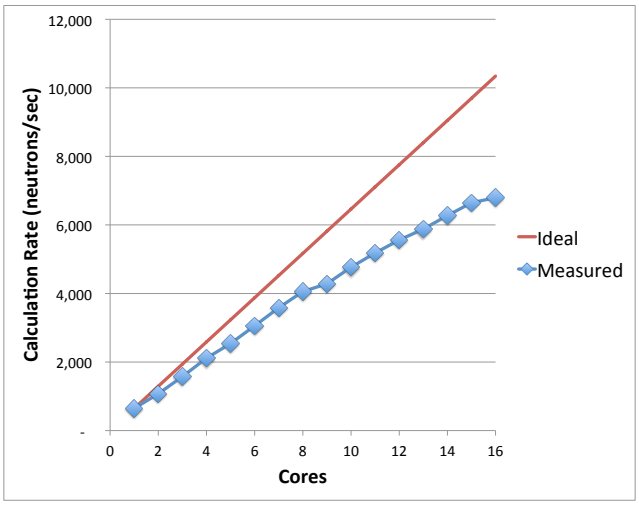

Figure 1: *OpenMC* Performance Scaling on a 16-Core Xeon Node

1. The Reactor Simulation Problem

Computer-based simulation of nuclear reactors is a well established field, with origins dating back to the early years of digital computing. Traditional reactor simulation techniques aim to solve the diffusion equation for a given material geometry and starting (source term) neutron distribution within the reactor. This is done in a deterministic fashion using well developed numerical methods. Deterministic codes are capable of running quickly and providing precise solutions, however, there are

other approaches to the problem that offer potential advantages.

An alternative method, Monte Carlo (MC) simulation, simulates the path of a particle neutron as it travels through the reactor core. As many particle histories are simulated, a picture of the full distribution of neutrons within the reactor core is developed. Such codes are inherently simple, easy to understand, and potentially easy to rethink when moving to new, novel architectures. Furthermore, the methodologies utilized by MC simulation require very few assumptions, resulting in highly accurate results assuming adequate statistical convergence. The downside to this method, however, is that a huge number of neutron histories must be run in order to achieve an acceptably low variance in the results. For many problems this means an impractically long time-to-solution, though such limitations may be overcome given the increased computational power of next-generation, exascale supercomputers.

2. OpenMC

OpenMC is a Monte Carlo particle transport simulation code focused on neutron criticality calculations.[1] It is capable of simulating 3D models based on constructive solid geometry with second-order surfaces. The particle interaction data is based on ACE format cross sections, also used in the MCNP and Serpent Monte Carlo codes. *OpenMC* has been used to investigate scaling concerns on distributed memory architectures, such as the IBM Blue Gene/P and Blue Gene/Q.

OpenMC was originally developed by members of the Computational Reactor Physics Group at the Massachusetts Institute of Technology starting in 2011. Various universities, laboratories, and other organizations (including CESAR) now contribute to the development of *OpenMC*.

3. XSBench

The *XSBench* proxy application models the most computationally intensive part of a typical MC transport algorithm – the calculation of macroscopic neutron cross sections, a kernel which accounts for around 85% of the total runtime of *OpenMC*.[3] *XSBench* retains the essential performance-related computational conditions and tasks of fully featured reactor core MC neutron transport codes, yet at a fraction of the programming complexity of the full application. Particle tracking and other features of the full MC transport algorithm were left out of *XSBench* as they take up only a small portion of runtime. This provides a much simpler and far more transparent platform for testing the algorithm on different architectures, making alterations to the code, and collecting hardware runtime performance data.

XSBench is in active development by members of the Center for Exascale Simulation of Advanced Reactors (CESAR) at Argonne National Laboratory. The application is written in C, with multi-core parallelism support provided by OpenMP. *XSBench* is an open source software project. All source code is publicly available online.

II. Algorithm

1. Reactor Model

When carrying out reactor core analysis, the geometry and material properties of a postulated nuclear reactor must be specified in order to define the variables and scope of the simulation model. For the purposes of *XSBench*, we use a well known community reactor benchmark known as the Hoogenboom-Martin model.[4] This model is a simplified analog to a more complete, "real-world" reactor problem, and provides a standardized basis for discussions on performance within the reactor simulation community. *XSBench* recreates the computational conditions present when fully featured MC neutron transport codes (such as *OpenMC*) simulate the Hoogenboom-Martin reactor model, preserving a similar data structure, a similar level of randomness of access, and a similar distribution of FLOPs and memory loads.

2. Neutron Cross Sections

The purpose of an MC particle transport reactor simulation is to calculate the distribution and generation rates of neutrons within a nuclear reactor. In order to achieve this goal, a large number of neutron lifetimes are simulated by tracking the path and interactions of a neutron through the reactor from its birth in a fission event to its escape or absorption, the latter possibly resulting in subsequent fission events.

Each neutron in the simulation is described by three primary factors: its spatial location within a reactor's geometry, its speed, and its direction. At each stage of the transport calculation, a determination must be made as to what the particle will do next. Possible outcomes include uninterrupted continuation of free flight, collision, or absorption (possibly resulting in fission). The determination of which event occurs is based on a random sampling of a statistical distribution that is described by empirical material data stored in main memory. This data, called *neutron cross section data*, represents the probability that a neutron of a particular speed (energy) will undergo some particular interaction when it is inside a given type of material. To account for neutrons across a wide energy spectrum and materials of many different types, the data structure that holds this cross section data must be very large. In the case of the simplified Hoogenboom-Martin benchmark roughly 5.6 GB[1] of data is required.

3. Data Structure

A material in the reactor model is composed of a mixture of nuclides. For instance, the "reactor fuel" material might consist of several hundred different nuclides, while the "pressure vessel side wall" material might only contain a dozen or so. In total, there are 12 different materials and 355 different nuclides present in the modeled reactor. The data usage requirements to store this model are significant, totaling 5.6 GB, as summarized in Table 1.

[1] We estimate that for a robust depletion calculation, in excess of 100 GB of cross section data would be required.[5]

For each nuclide, an array of nuclide grid points are stored as data in main memory. Each nuclide grid point has an energy level, as well as five cross section values (corresponding to five different particle interaction types) for that energy level. The arrays are ordered from lowest to highest energy levels. The number, distribution, and granularity of energy levels varies between nuclides. One nuclide may have hundreds of thousands of grid points clustered around lower energy levels, while another nuclide may only have a few hundred grid points distributed across the full energy spectrum. This obviates straightforward approaches to uniformly organizing and accessing the data.

In order to increase efficiency of access, the algorithm utilizes another data structure, called the *unionized energy grid*, as described by Leppänen[6] and Romano.[1] The unionized grid facilitates fast lookups of cross section data from the nuclide grids. This structure is an array of grid points, consisting of an energy level and a set of pointers to the closest corresponding energy level on each of the different nuclide grids.

Nuclides Tracked	355
Total # of Energy Gridpoints	4,012,565
Cross Section Interaction Types	5
Total Size of Cross Section Data Structures	5.6 GB

Table 1: *XSBench* Data Structure Summary

4. Access Patterns

In a full MC neutron transport application, the data structure is accessed each time a macroscopic cross section needs to be calculated. This happens anytime a particle changes energy (via a collision) or crosses a material boundary within the reactor. These macroscopic cross section calculations occur with very high frequency in the MC transport algorithm, and the inputs to them are effectively random. For the sake of simplicity, *XSBench* was written ignoring the particle tracking aspect of the MC neutron transport algorithm and instead isolates the macroscopic cross section lookup kernel. This provides a large reduction in program complexity while retaining similarly random input conditions for the macroscopic cross section lookups via the use of a random number generator.

In *XSBench*, each macroscopic cross section lookup consists of two randomly sampled inputs: the neutron energy E_p, and the material m_p. Given these two inputs, a binary (log n) search is done on the unionized energy grid for the given energy. Once the correct entry is found on the unionized energy grid, the material input is used to perform lookups from the nuclide grids present in the material. Use of the unionized energy grid means that binary searches are not required on each individual nuclide grid. For each nuclide present in the material, the two bounding nuclide grid points are found using the pointers from the unionized energy grid and interpolated to give the exact microscopic cross section at that point.

All calculated microscopic cross sections are then accumulated (weighted by their atomic density in the given material), which results in the macroscopic cross section for the material. Algorithm 1 is an abbreviated summary of this calculation.

In theory, one could "pre-compute" all macroscopic cross sections on the unionized energy grid for each material. This would allow the algorithm to run much faster, requiring far fewer memory loads and far fewer floating point operations per macroscopic cross section lookup. However, this would assume a static distribution of nuclides within a material. In practice, MC transport nuclide-depletion calculations are quasi-static; they will need to track the burn-up of fuels and account for heterogeneous temperature distributions within the reactor itself. This means that concentrations are dynamic, rather than static, therefore necessitating the use of the more versatile data model deployed in *OpenMC* and *XSBench*.

Algorithm 1 Macroscopic Cross Section Lookup

1: $R(m_p, E_p)$ ▷ randomly sample inputs
2: Locate E_p on Unionized Grid ▷ binary search
3: **for** $n \in m_p$ **do** ▷ for each nuclide in input material
4: $\sigma_a \leftarrow n, E_p$ ▷ lookup bounding micro xs's
5: $\sigma_b \leftarrow n, E_p + 1$
6: $\sigma \leftarrow \sigma_a, \sigma_b$ ▷ interpolate
7: $\Sigma \leftarrow \Sigma + \rho_n \cdot \sigma$ ▷ accumulate macro xs
8: **end for**

We have verified that *XSBench* faithfully mimics the data access patterns of the full MC application under a broad range of conditions. The runtime of full-scale MC transport applications, such as *OpenMC*, is 85% composed of macroscopic cross section lookups.[3] Within this process, *XSBench* is virtually indistinguishable from *OpenMC*, as the same type and size of data structure is used, with a similarly random access pattern and a similar number of floating point operations occurring between memory loads. Thus, performance analysis done with *XSBench* provides results applicable to the full MC neutron transport algorithm, while being far easier to implement, run, and interpret.

III. Application

To investigate the performance bottlenecks of MC transport methods on existing systems, we carried out a series of tests using *XSBench* on single node, multi-core, shared memory systems. The systems used were as follows:

- PC node consisting of two Intel Xeon E5-2650 octo-core CPUs for a total of 16 physical CPUs. All tests, unless otherwise noted, were run at 2.8 GHz using Intel Turbo Boost.
- Single node of the IBM Blue Gene/Q (BG/Q) supercomputer *Mira* at Argonne National Laboratory consisting of 16 physical CPUs, which were run at 1.6 GHz.

We performed a scaling study to determine performance improvements as additional cores were added. For both the Xeon system and the BG/Q node, we ran *XSBench* with only a single thread to determine a baseline performance against which scaling can be measured. Then, further runs were done to test each number of threads between 1 and 16. Scaling is defined in Equation 1, where n is the number of cores, R_n is the experimental calculation rate for n cores, and R_1 is the experimental calculation rate for 1 core.

$$\text{Scaling}_n = \frac{R_n}{R_1 \times n} \quad (1)$$

The tests reveal that even for this idealized representation of the key MC transport algorithm, perfect scaling was not achievable. Figures 2 and 3 show that scaling degraded gradually as more cores were used on the nodes. For the Xeon system, scaling at 16 cores degraded to 69%. For the BG/Q system, scaling at 16 cores degraded to 96%.

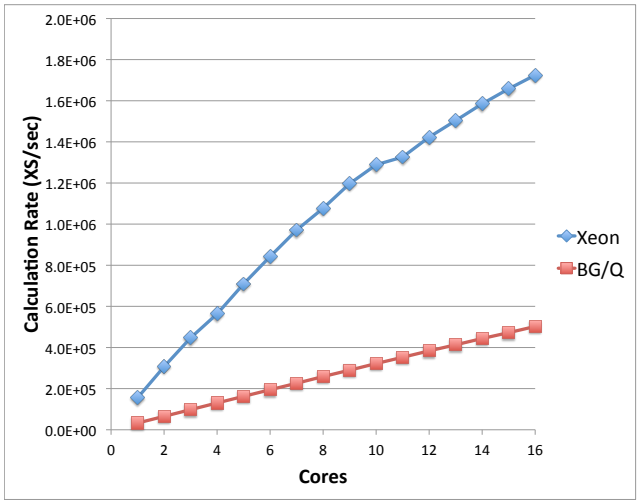

Figure 2: *XSBench* Performance Scaling

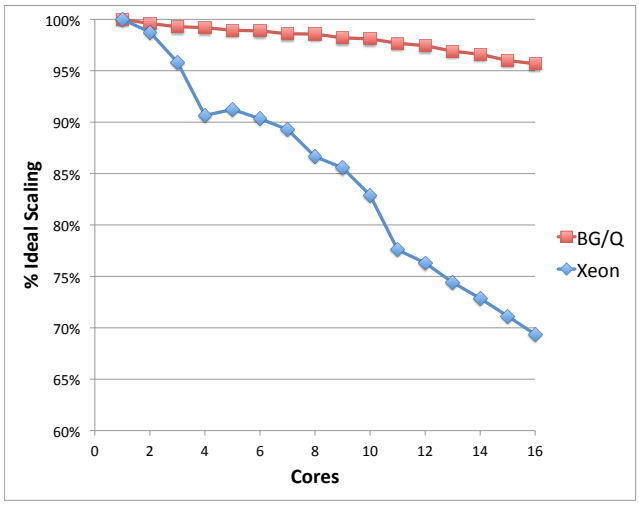

Figure 3: Percent of Ideal Scaling

One might reasonably conclude that 69% or 96% scaling out to 16 cores is adequate speedup. However, next-generation node architectures are likely to require up to thousand-way on-node shared memory parallelism,[7][8][9][10] and thus it is crucial to ascertain the cause of the observed degradation and the implications for greater levels of scalability. Considering nodes with 32, 64, 128, or 1024 shared memory cores and beyond, it cannot be taken for granted that performance will continue to improve. We thus seek to identify to the greatest extent possible which particular system resources are being exhausted, and how quickly, so that designers of future hardware systems as well as developers of future MC particle transport applications can avoid bottlenecks.

High performance computing (HPC) applications generally have several possible reasons for performance loss due to scaling:

1. FLOP bound – A CPU can only perform so many floating point operations per second.
2. Memory Bandwidth Bound – A finite amount of data can be sent between DRAM and the CPU.
3. Memory Latency Bound – An operation on the CPU that requires data be sent from the DRAM can take a long time to arrive.
4. Inter-Node Communication Bound – Nodes working together on a problem may need to wait for data from other nodes, incurring large latency and bandwidth costs. This is not an issue for this application since we are focusing only on single node, shared-memory parallelism.

Given these potential candidates for bottlenecks, we aim to determine which exact subsystems are responsible for performance degradation by performing a series of studies to identify which specific resources our kernel exhausts first.

IV. Experiment & Results

To determine the cause of multi-core scaling degradation, we performed a series of experiments. Each experiment involves varying a system parameter, monitoring hardware usage using performance counters, and/or altering a portion of the *XSBench* code. The following section presents descriptions, results, and preliminary conclusions for each experiment. For the purposes of simplicity, we concentrate our analysis on the Intel Xeon system described in section III. This allows us to get highly in-depth results as we are able to run experiments dealing with architecture-specific features and hardware counters.

1. Effect of Processor Speed

It is postulated that processors with faster clock speeds should in general be more susceptible to memory latency and bandwidth related bottlenecks. Faster processors are capable of generating more read requests per second, resulting in a higher potential for bandwidth usage. Furthermore, when a CPU makes a read request from main memory, there is a large component of latency that is independent of processor speed (e.g. speed of light). If the processor is incapable of masking that latency, or is only capable of pipelining a fixed number of read requests, then the CPU will be forced to sit idle while the outstanding read request is filled. The faster the processor, the more CPU cycles will be wasted while waiting for outstanding reads. Thus, simply slowing down the clock speed of a processor may reduce bandwidth requirements, improve scalability, and potentially improve power efficiency.

From Figure 3, we can see that BG/Q scales much better than Xeon. Given that BG/Q runs at 1.6 GHz, whereas the Xeon node runs at 2.8 GHz, it would appear that processor

speed may indeed have a correlation to scaling. To further test this hypothesis, we disabled Turbo Boost on the Xeon node, reducing the clock speed to 2.0 GHz and ran another set of scaling tests. As shown in Figure 4 and Figure 5, scaling improved from 69% at 2.8 GHz to 79% at 2.0 GHz.

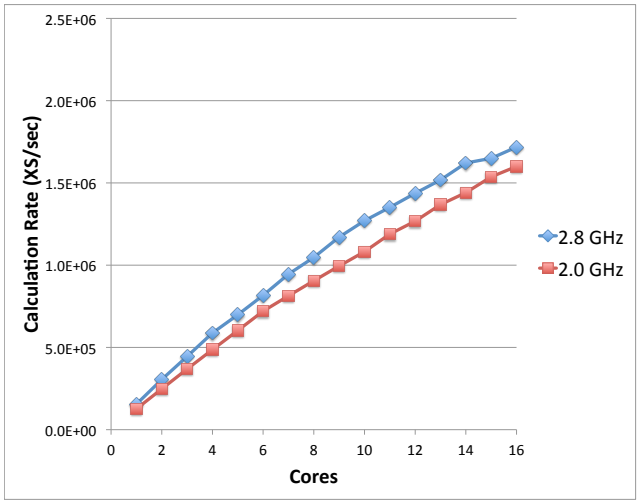

Figure 4: Xeon Performance Scaling for Various Clock Speeds

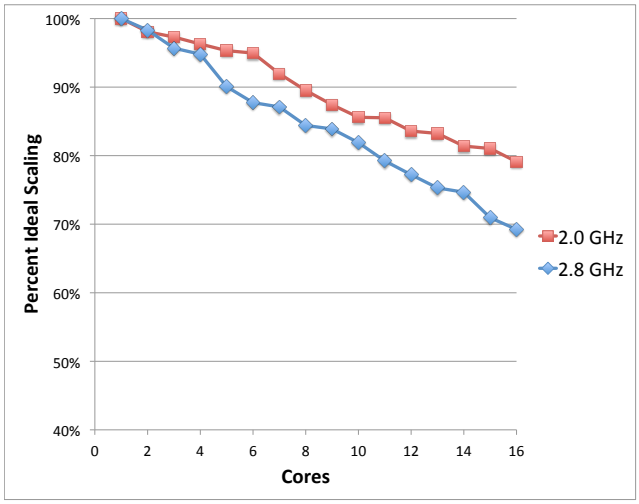

Figure 5: Xeon % of Ideal Scaling for Various Clock Speeds

Figure 4 also shows that the calculation rate for 16 cores only slowed down by 7% while the processor speed was reduced by over 28%. A reduction in clock speed can result in a large savings to the power budget for the node, as the CPU can be run at a lower voltage, reducing power. Thus, a small sacrifice in calculation time-to-completion results in both a) a higher cycle efficiency and b) a significantly lower clock speed. These effects combine to potentially reduce the total power cost of the calculation.

2. Resource Usage

To better understand scaling degradation in our kernel, we implemented performance counting features into the source code of *XSBench* using the Performance Application Programming Interface (PAPI).[11] This allowed us to select from a large variety of performance counters (both preset and native to our particular Xeon chips). We collected data for many counters, including:

- `ix86arch::LLC_REFERENCES` - Last Level (L3) Cache References
- `ix86arch::LLC_MISSES` - Last Level (L3) Cache Misses
- `PAPI_TOT_CYC` - Total CPU Cycles
- `RESOURCE_STALLS:ANY` - CPU cycles stalled for resources for any reason
- `RESOURCE_STALLS:RS` - CPU cycles stalled because there is no eligible entry in the Reservation Station (RS)
- `PAPI_FP_INS` - Floating point instructions

These raw performance counters allowed us to calculate a number of composite metrics, including bandwidth usage, FLOP utilization, cache miss rate, and identification of cycle stall causation. Each of the metrics are discussed in the following subsections.

2.1. Bandwidth & FLOPs

Two very important resources for any computer are the available memory bandwidth and floating point capacity of the node. Consumption of these resources is calculated for *XSBench* using Equation 2 and Equation 3.

$$\text{Bandwidth} = \frac{\text{LLC\_MISSES} \times \text{Linesize}}{\text{PAPI\_TOT\_CYC}} \times \text{Clock (Hz)} \quad (2)$$

$$\text{FLOPs} = \frac{\text{PAPI\_FP\_INS}}{\text{PAPI\_TOT\_CYC}} \times \text{Clock (Hz)} \quad (3)$$

Using Equation 2, we collected the bandwidth usage for *XSBench* as run on varying numbers of cores, as shown in Figure 6. Note that the maximum theoretically available bandwidth for the Xeon node is 51.2 GB/s.[12] Figure 6 shows that less than half the available bandwidth is ever used by *XSBench*, even when running at 16 cores per node.

There is, however, the question as to how much bandwidth is realistically usable on any given system. Even a perfectly constructed application that floods the memory system with easy, predictable loads is unlikely to be able to use the full system bandwidth. In order to determine what is actually usable on our Xeon system, we ran the *STREAM* benchmark, which measures "real world" bandwidth sustainable from ordinary user programs.[13] Results from this benchmark are shown in Figure 7, and compared to *XSBench* in Figure 8. At 16 cores, *STREAM* only achieves 27.5 GB/s compared to *XSBench's* 19.9 GB/s. Thus, *XSBench* is using a much higher percentage (72%) of the realistically available bandwidth than was previously assumed when comparing to the theoretical maximum of 51.2 GB/s.[12]

The other major system resource to consider is FLOP utilization. Using Equation 3, we were able to determine that

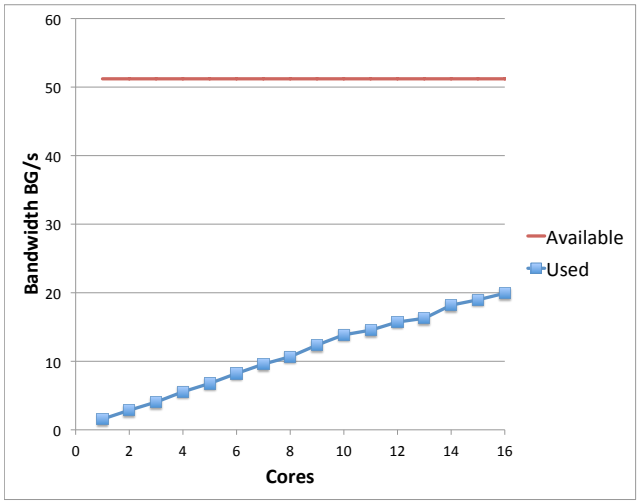

Figure 6: *XSBench* Bandwidth Usage Scaling

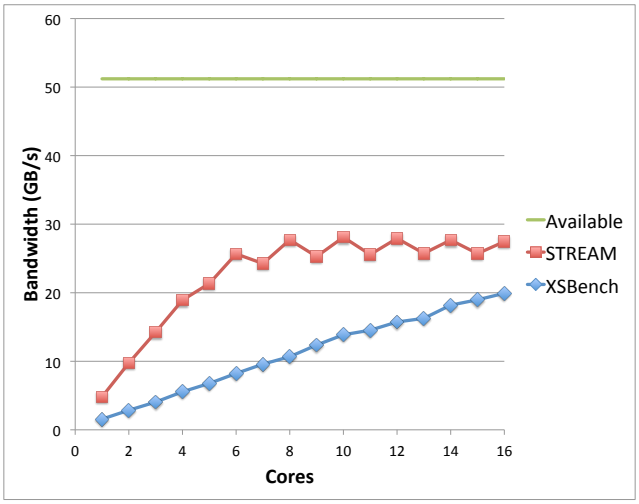

Figure 8: Bandwidth Usage: *STREAM* vs *XSBench*

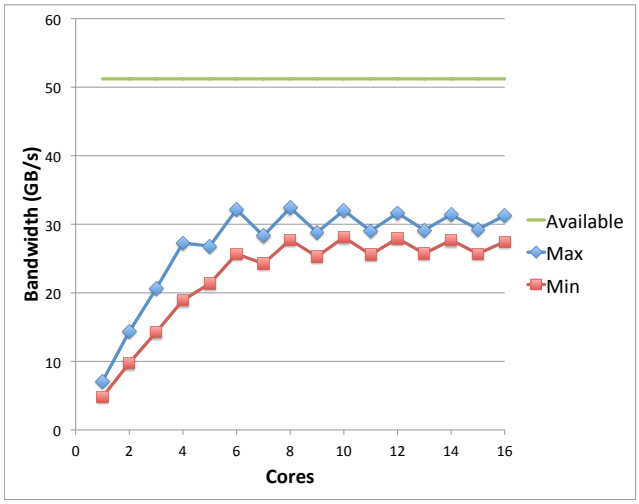

Figure 7: Bandwidth Usage of the *STREAM* Benchmark

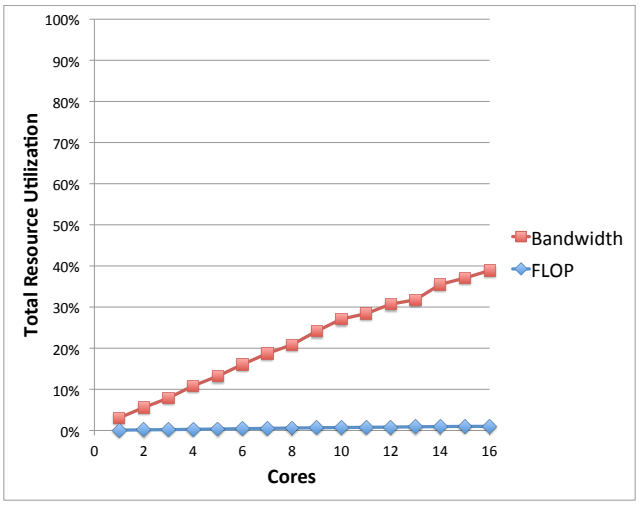

Figure 9: Bandwidth and FLOP Usage

XSBench achieved at most 3.59 GFLOPs. Given that the Xeon node is theoretically capable of 358 GFLOPs, *XSBench* only uses about 1% of the system's floating point capability.

A comprehensive plot of system resource utilization (bandwidth and FLOP) is shown in Figure 9. As can be seen, *XSBench* does not exhaust the system's memory bandwidth or floating point capability. Floating point provisions are not stressed, and memory bandwidth only ever reaches roughly 40% of the theoretical maximum (72% of the experimental maximum).

2.2. Cache Misses

Another important metric for application performance is the last level cache (LLC) miss percentage. Modern CPUs, as used in our Xeon node, rely heavily on caching to minimize the cost of data loads and stores. Caches are split up into several levels, each growing larger but also with significantly higher access times. When the CPU makes a data read request, all levels of cache are checked. If the data is not found in cache, then it must be retrieved from main memory (DRAM) on the node. Accessing main memory takes orders of magnitude longer than accessing cache, meaning that applications with exceptionally high cache miss rates are often slow, as CPUs are left sitting idle as data arrives from main memory. Using PAPI performance counters, we were able to calculate the last level (L3) cache miss rates using Equation 4. We found that across all cores, for any number of threads 1 through 16, the LLC miss rate was 65%.

$$\text{Cache Miss \%} = \frac{\text{LLC\_MISSES}}{\text{LLC\_REFERENCES}} \times 100 \qquad (4)$$

2.3. Resource Stalls

Figure 9 clearly shows that only a small fraction of the CPUs floating point capabilities are being used. From this, it is obvious that CPUs spend a significant fraction of their time sitting idle. PAPI provides counters that specify the precise reason for such wasted cycles. We tested a number of available counters to determine the primary cause of cycle stall. It turned out that 90-94% of cycles were spent stalled waiting for resources, and

that this number was not dependent on the number of cores being run on the chip. Using Equation 5, we found that 98% of resource stalls were attributed to there being no eligible reservation station (RS) entry available.

$$\%\ \text{Stalls in RS} = \frac{\text{RESOURCE\_STALLS:RS}}{\text{RESOURCE\_STALLS:ANY}} \times 100 \quad (5)$$

The RS counter of the Xeon CPU reveals situations where instructions are held while waiting for operands. When the CPU is idle, it continuously checks the RS to determine if there are any instructions that have the necessary operands to be completed. There are a finite number of RS entries available (36 in some Xeon chips).[14] From the Intel processor events online manual:

> This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire). In these situations new instructions can not enter the reservation station and start execution.[15]

Thus, given the virtually random nature of *XSBench*'s access patterns and accompanying high cache miss rate (65%), increasing the number of reservation stations would likely increase single core calculation rates as more read requests could be in-flight at any point in time.

3. Impact of Hardware Threads on Bandwidth Usage

Looking at Figure 6 and considering that the bandwidth usage trend continues upwards towards the maximum, it is possible to conclude that perhaps Xeon hardware threads (also known as Hyper-Threads) may extend this scaling out to reach the full capacity of the chip.

Hardware threading provides infrastructure to store data and instructions from one thread while the CPU waits for outstanding read requests to return from main memory while one or more additional threads concurrently execute operations on the CPU. The primary benefit of hardware threading is its ability to mask the latency between the CPU and main memory. As a result, applications that suffer from a high cycle stall rate due to cache misses may benefit from use of hardware threading. The additional performance benefit of hardware threading comes at the cost of only a small increase in die space without significantly increasing the amount of power consumed.

Because *XSBench* experiences a high rate of cache misses (65%), we tested *XSBench* on the Xeon Hyper-Thread cores by running scaling studies to 32 threads. The usual performance counter data was also collected. These results are summarized in Figure 10, which shows marginal performance gains from the use of hardware threads.

STREAM was also run on our Xeon node out to 32 threads to test the Hyper-Thread cores. As shown in Figure 11, *XSBench*

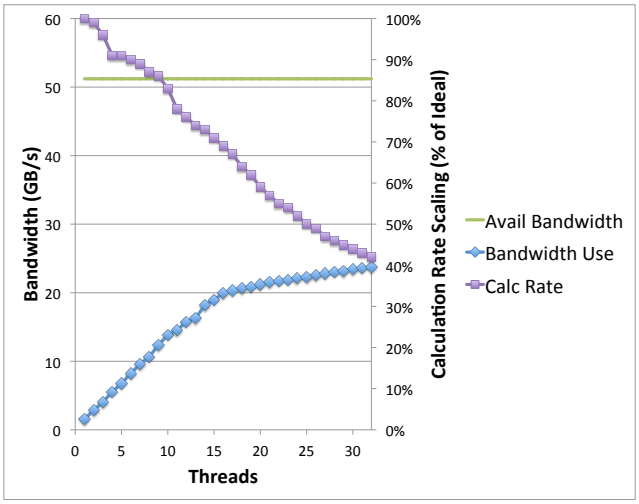

Figure 10: Impact of Hyper-Threading on Performance and Bandwidth

uses 23.7 GB/s of bandwidth, which is extremely close to the *STREAM*'s use of 25.8 GB/s. We find that we were able to effectively reach the maximum practical bandwidth limit on the system, as no real-world application can realistically expect to attain a higher bandwidth than *STREAM*. Thus, we have shown that when Hyper-Threading is enabled on our Xeon node, *XSBench* becomes a bandwidth bottlenecked application.

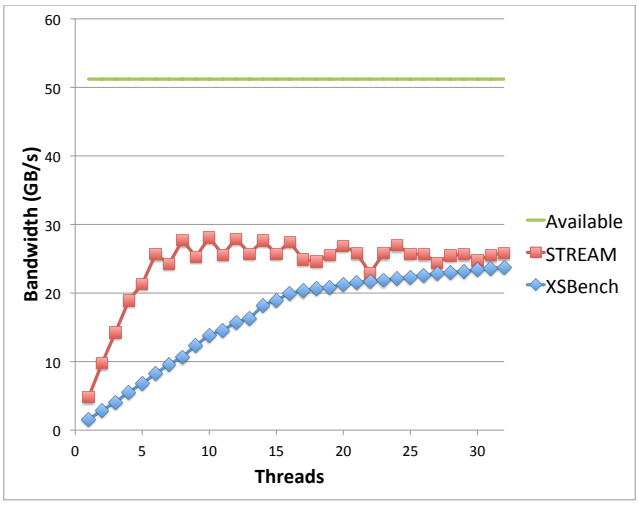

Figure 11: Impact of Hyper-Threading on *XSBench* and *STREAM* Bandwidth

4. Effects of Additional FLOPs

We also studied the effects of increasing the FLOP to load ratio by injecting "dummy" FLOPs every time a microscopic XS was calculated in *XSBench*. These results are presented in Figure 12 and Figure 13. By comparing Figure 3 and Figure 13, we can see that the scaling at 16 cores improves from 69% to 84% after only 50 FLOPs have been added per load. This change suggests that increasing the number of FLOPs per load gives the CPUs more work to perform, thus spacing out the memory loads in time, and therefore easing away from the memory bandwidth

bottleneck.

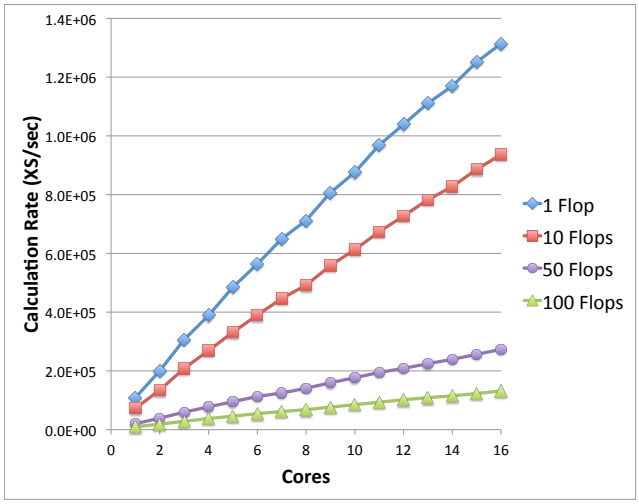

Figure 12: Scaling with Varying Number of Dummy FLOPs added per Micro-XS Calculation

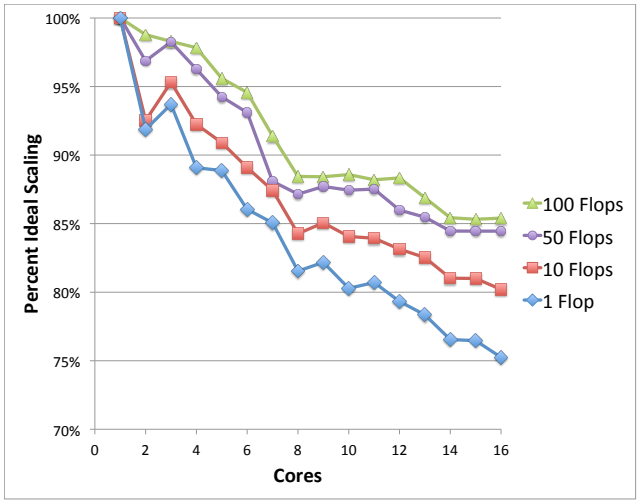

Figure 13: % of Ideal Scaling with Varying Number of Dummy FLOPs added per Micro-XS Calculation

Since the MC particle transport algorithm benefits from additional FLOPS, changes to the algorithm or its data structures that accomplish useful work via extra FLOPs are highly beneficial. There are several potential features that can accomplish this task:

- On-the-fly Doppler broadening[16][17] - This method accounts for the material temperature dependence of cross sections. In a full-scale reactor simulation, cross section data needs to be stored for both varying neutron energy levels and varying material temperatures. Doppler broadening removes the need to store temperature dependent cross section data, instead only storing the 0 K data and computing data for other temperatures on-the-fly whenever data requests are made. Addition of such Doppler broadening techniques could allow for a significant amount of useful FLOP work to be done between memory loads, while also reducing the memory footprint and memory bandwidth requirements, potentially improving scaling of the calculation.
- Cross section data compression techniques - Methods can be employed to reduce neutron energy dependent cross section data into resonance regions. These methods also greatly reduce the memory footprint and bandwidth requirements of the algorithm, in exchange for a number of FLOPs to be completed to "unpack" the resonance data back into the required cross sections.

V. Conclusions

Using an extracted kernel to simplify the MC particle transport algorithm, we have developed methodologies to identify and possibly minimize performance bottlenecks. We have characterized the scaling behavior of our kernel, and found that for the chosen problem configuration only 69% of ideal scaling is achievable on the Xeon platform. To determine the specific reasons for this scaling degradation we performed the following experiments:

1. Role of Processor Speed
2. Resource Usage
   (a) Bandwidth & FLOP Usage
   (b) Cache Miss Rate
   (c) Resource Stalls
3. Impact of Hyper-Threading on Bandwidth Usage
4. Effects of Additional FLOPs

Through these experiments we were able to identify, at a low level, why our kernel was not running faster or scaling better.

On a single core, the performance bottleneck can be isolated to the reservation station (RS) queue of the CPU. Our data showed that 98% of resource stalls were due to the RS being filled – an impressively high number, given that resource stalls account for over 90% of the CPU runtime. As RS stalls are caused by high latency operations, such as cache misses, the data suggests that increasing the size of the RS could allow for more instructions to be queued at once, and therefore more data to be in flight at any given time, thus increasing performance.

As more cores are added to the system, up to the 16 on our node, performance scaling degrades to 69%. FLOP usage is at less than 1% of capacity, and bandwidth usage is only at 72% of the experimental maximum (as measured by the *STREAM* benchmark). *XSBench* has not exhausted bandwidth or FLOP resources, but rather latency hiding resources – namely the number of entries in the RS. This bottleneck can be partially reduced by decreasing the clock speed of the processor, which improves scaling as well as the cycle efficiency of the calculation. This has the added result of potentially improving the power efficiency of the calculation. This bottleneck can also be mitigated by finding useful FLOP work to be done (e.g., on-the-fly Doppler broadening) in order to reduce the frequency of loads from main memory.

Running *XSBench* with 32 threads on 16 cores, using Hyper-Threading, we find that outstanding read request capacity has been increased. Performance can continue to rise, all the way up to 32 threads where memory bandwidth is exhausted. Here, we find that *XSBench* uses 23.7 GB/s of bandwidth, which is extremely close to the practicle maximum of 25.8 GB/s. At this point, the MC particle transport algorithm becomes limited by the available system bandwidth. Adding cores, hardware threads, or improving other latency masking techniques will not result in faster calculation rates; bandwidth must be increased for performance to increase.

VI. Future Work

There are additional capabilities that do not yet exist in full-scale MC neutron transport algorithms, such as on-the-fly Doppler broadening to account for the material temperature dependence of cross sections, that we plan on adding to *XSBench* for experimentation with various hardware architectures and features. Addition of such Doppler broadening techniques could allow for a significant amount of useful FLOP work to be done between memory loads, potentially improving scaling of the calculation. Furthermore, we want to investigate cross section data compression techniques, specifically by reducing the data into resonance regions, in the hopes of quantifying the trade offs (in terms of decreased DRAM memory footprint and bandwidth vs. the increase in CPU FLOP requirements) of such compression techniques.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract DE-AC02-06CH11357 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

1) P. K. Romano and B. Forget, "The OpenMC Monte Carlo particle transport code," *Annals of Nuclear Energy*, **51**, *C*, 274–281 (2013).
2) P. K. Romano, B. Forget, and F. Brown, "Towards Scalable Parallelism in Monte Carlo Particle Transport Codes Using Remote Memory Access (Selected Papers of the Joint International Conference of Supercomputing in Nuclear Applications and Monte Carlo : SNA + MC 2010)," *Progress in nuclear science and technology*, **2**, 670-675 (2011).
3) A. R. Siegel, K. Smith, P. K. Romano, B. Forget, and K. G. Felker, "Multi-core performance studies of a Monte Carlo neutron transport code," *International Journal of High Performance Computing Applications* (2013).
4) J. E. Hoogenboom, W. R. Martin, and B. Petrovic, "Monte Carlo Performance Benchmark for Detailed Power Density Calculation in a Full Size Reactor Core Benchmark Specifications," *Ann Arbor*, **1001**, 48109–42104 (2010).
5) P. K. Romano, A. R. Siegel, B. Forget, and K. Smith, "Data decomposition of Monte Carlo particle transport simulations via tally servers," *Journal of Computational Physics*, **252**, *0*, 20 - 36 (2013).
6) J. Leppänen, "Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation," *Annals of Nuclear Energy*, **36**, *7*, 878–885 (2009).
7) S. Dosanjh et al., "Exascale design space exploration and co-design," *Future Generation Computer Systems* (2013).
8) N. Attig, P. Gibbon, and T. Lippert, "Trends in supercomputing: The European path to exascale," *Computer Physics Communications*, **182**, *9*, 2041 - 2046 (2011).
9) N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez, "The low power architecture approach towards exascale computing," *Journal of Computational Science* (2013).
10) C. Engelmann, "Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale," *Future Generation Computer Systems* (2013).
11) "PAPI - Performance Application Programming Interface," http://icl.cs.utk.edu/papi/index.html, 2013.
12) Intel®, "Xeon® Processor E5-2650 CPU Specifications," http://ark.intel.com/products/64590/, 2013.
13) J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (1995).
14) D. Levinthal, "Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors," http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.
15) Intel®, "Intel® Processor Events Reference," http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/amplifierxe/win/win_reference/index.htm#pmn/events/resource_stalls.html, 2013.
16) W. R. Martin, S. Wilderman, F. B. Brown, and G. Yesilyurt, "Implementation of On-the-Fly Doppler Broadening in MCNP," *Proc. International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Sun Valley, Idaho, 2013.
17) T. Viitanen and J. Leppänen, "Explicit Treatment of Thermal Motion in Continuous-Energy Monte Carlo Tracking Routines," *Nucl. Sci. Eng.*, **171**, 165–173 (2012).