Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations

John R. Tramm and Andrew R. Siegel

Argonne National Laboratory, USA
jtramm@mcs.anl.gov

Abstract. Current Monte Carlo neutron transport applications use continuous energy cross section data to provide the statistical foundation for particle trajectories. This "classical" algorithm requires storage and random access of very large data structures. Recently, Forget et al.[1] reported on a fundamentally new approach, based on multipole expansions, that distills cross section data down to a more abstract mathematical format. Their formulation greatly reduces memory storage and improves data locality at the cost of also increasing floating point computation. In the present study, we abstract the multipole representation into a "proxy application", which we then use to determine the hardware performance parameters of the algorithm relative to the classical continuous energy algorithm. This study is done to determine the viability of both algorithms on current and next-generation high performance computing platforms.

Keywords: Monte Carlo, multi-core, neutron transport, reactor simulation, multipole, cross section

1 Introduction

Monte Carlo (MC) transport algorithms are considered the "gold standard" of accuracy for a broad range of applications – e.g., nuclear reactor physics, shielding, detection, medical dosimetry, and weapons design to name just a few examples. In the design and analysis of nuclear reactor cores, the key application driver of the present analysis, MC methods for neutron transport offer significant potential advantages compared to deterministic methods given their simplicity, avoidance of ad hoc approximations in energy treatment, and lack of need for complex computational meshing of reactor geometries.

On the other hand it is well known that robust analysis of a full reactor core is still beyond the reach of MC methods. Tremendous advances have been made in recent years, but the computing requirements for full quasi-static depletion analysis of commercial reactor cores is a performance-bound problem, even on existing leadership class computers. It is also clear that many of the issues related to scalability on distributed memory machines have been adequately addressed in recent studies[2][3], and that the path to future speedups involves taking better

advantage of a broad range of multi-core systems. For MC methods this is most naturally done, as a first step, in a MIMD context, which allows us to most easily exploit the natural parallelism over particle tracks, each with complex, nested branching logic. Siegel et al.[4] carried out an in-depth study of on-node scalability of the *OpenMC*[2] transport code, showing encouraging results as well as limitations due to memory contention. Tramm et al.[5] carried out an in-depth study based on the *XSBench* mini-application, further elucidating the underlying performance bottlenecks that inhibit scalability. Indeed, with less memory bandwidth per core as nodes become more complex, developing new approaches that minimize memory contention and maximize use of each core's floating point units becomes increasingly important.

Recently, Forget et al.[1] proposed a new algorithm for representing neutron cross section data in a more memory efficient manner. This algorithm, based on multipole expansions, compresses data into a more abstract mathematical format. This greatly reduces the memory footprint of the cross section data and improves data locality at the expense of an increase in the number of computations required to reconstruct it when it is needed. As next-generation leadership class computers are likely to favor floating point operations over data movement[6][7][8][9], the multipole algorithm may provide significant performance improvement compared to the classical approach.

In this analysis we study in-depth two different implementations of the MC neutron transport algorithm – the "classical" continuous energy cross section format and the multipole representation format. Then, we assess the on-node scaling properties and memory contention issues of these algorithms in the context of a reactor physics calculation.

1.1 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 deterministic equations (typically a variant of the the diffusion equation) for a given material geometry and initial neutron distribution (source) within the reactor. This is done using mature and well understood numerical methods. Deterministic codes are capable of running quickly and providing relatively accurate gross power distributions, but are still limited when accurate localized effects are required, such as e.g. at sharp material interfaces.

An alternative formulation, the Monte Carlo (MC) method, simulates the path of individual neutrons as they travel through the reactor core. As many particle histories are simulated and tallied, a picture of the full distribution of neutrons within the domain emerges. Such codes are inherently simple, easy to understand, and potentially easy to restructure when porting to new systems. Furthermore, the methodologies utilized by MC simulation require very few assumptions, resulting in highly accurate results given adequate statistical convergence. The downside to this method, however, is that a huge number of neutron histories are required 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.

1.2 OpenMC

OpenMC is a Monte Carlo particle transport simulation code focused on neutron criticality calculations[2]. 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 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 now contribute to its development. The application is written in FORTRAN, with parallelism supported by a hybrid OpenMP/MPI model. *OpenMC* is an open source software project available online[10].

1.3 XSBench

The *XSBench* proxy application models the most computationally intensive part of a typical MC reactor core transport algorithm – the calculation of macroscopic neutron cross sections, a kernel which accounts for around 85% of the total runtime of *OpenMC*[4]. *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 not included in *XSBench* as they take up only a small portion of runtime in robust reactor computations. 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 was developed 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[11].

1.4 RSBench

The *RSBench* proxy application is similar in purpose to *XSBench*, but models an alternative method for calculating neutron cross sections – the multipole method. This method organizes the data into a significantly more compact form, saving several orders of magnitude in memory space. However, this method also requires "unpacking" of this data by way of a significant number of additional computations (FLOPs).

RSBench is in active development by members of the CESAR group at Argonne National Laboratory. The application is written in C, with multi-core

parallelism support provided by OpenMP. *RSBench* is an open source software project. All source code is publicly available online[12].

2 Algorithm

2.1 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* and *RSBench*, we use a well known community reactor benchmark known as the Hoogenboom-Martin model[13]. 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* and *RSBench* recreate 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 , and a similar distribution of FLOPs and memory loads.

2.2 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 classical algorithm, as represented by *XSBench*, requires use of a very large data structure that holds cross section data points for many discrete energy levels. In the case of the simplified Hoogenboom-Martin benchmark, roughly 5.6 GB[1] of data is required. The multipole method greatly reduces these requirements down the the order of approximately 100 MB for all data.

[1] We estimate that for a robust depletion calculation, in excess of 100 GB of cross section data would be required.[14]

2.3 Classical Continuous Energy Cross Section Representation

The classical continuous energy cross section representation, as used by real world applications like *OpenMC*, is abstracted in the proxy-application *XSBench*. This section describes the data structure used by this algorithm along with the access patterns of the algorithm.

Data Structure A material in the Hoogenboom-Martin 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.

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 , the algorithm utilizes another data structure, called the *unionized energy grid*, as described by Leppänen [15] and Romano [2]. 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

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.

Algorithm 1 Classical Continuous Energy 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**

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*. Even if static concentrations were assumed, pre-computation of the full spectrum of macroscopic cross sections would need to be done for all geometric regions (of which there are many millions) in the reactor model, leading to even higher memory requirements.

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[4]. 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.

2.4 Multipole Cross Section Representation

The multipole representation cross section algorithm is abstracted in the proxy-application *RSBench*. This section summarizes the data structure used by this algorithm along with the access patterns and computations performed by the algorithm. The multipole representation stores cross section data in the form of resonances. Each resonance can be characterized by several variables, defining the parameters of the resonance (i.e., width, height, reaction width, etc.) that can be used to compute the actual microscopic cross section at any energy within the resonance. A more in-depth explanation of the mathematics behind the multipole representation is offered by Forget et al.[1] For the purposes of this analysis, several factors that play a minimal role in performance, such as the handling of energy levels in the unresolved resonance region, were ignored.

Data Structure The primary data structure employed by *RSBench* is a 2-D jagged array of resonance data structures. The first dimension correlates to each nuclide present in the reactor. The second dimension correlates to the number of resonances present in that nuclide (each nuclide has a different number of resonances, varying from 100 to 6,000)[1]. Each element of this array is a resonance data structure that holds several pieces of information including the width of the resonances and several mathematical values that define the shape, magnitude, and number of poles in the resonance. Compared to the classical method, such as used by *XSBench*, this 2-D array is in total much smaller as no unionized energy grid is necessary and far fewer data points are needed, as shown in Table 2.

Nuclides Tracked	355
Average Resonances per Nuclide	3,000
Total # of Resonances	1,065,000
Cross Section Interaction Types	4
Total Size of Cross Section Data Structures	41 MB

Table 2. *RSBench* Data Structure Summary

Access Patterns The access patterns of the multipole algorithm are simpler than the classical methods due to the lack of a unionized energy grid. This means

that for a given neutron energy level and material, generating the macroscopic cross sections is a relatively simple process. For each nuclide in the material, a modulus operation is done to determine the index of the resonance that covers the neutron's energy. The presence of multiple resonances covering the same energy level is not considered, as this is a relatively infrequent occurrence and in our experience does not significantly impact performance. With the resonance data retrieved, several somewhat lengthy computations are done to determine the various microscopic cross sections. Macroscopic cross sections are then accumulated. This process is summarized in Algorithm 2.

Algorithm 2 Multipole Macroscopic Cross Section Lookup

1: $R(m_p, E_p)$ ▷ randomly sample inputs
2: **for** $n \in m_p$ **do** ▷ for each nuclide in input material
3: Fetch Resonance R Covering Energy E_p
4: $\sigma_g \leftarrow$FLOPs$\leftarrow R$ ▷ calculate n(n,γ) XS
5: $\sigma_f \leftarrow$FLOPs$\leftarrow R$ ▷ calculate n(n,f) XS
6: $\sigma_a \leftarrow$FLOPs$\leftarrow R$ ▷ calculate absorption XS
7: $\sigma_t = \sigma_g + \sigma_f + sigma_a$ ▷ accumulate total XS
8: $\Sigma \leftarrow \Sigma + \rho_n \cdot \sigma$ ▷ accumulate macro xs
9: **end for**

The equations used to assemble microscopic cross sections out of the resonance data are described in detail by Forget et al[1]. Simplified forms of the multipole equations used by *RSBench*, the single level Breit-Wigner equations, are given in Equations 1, 2, and 3. Note that single level Breit-Wigner resonances are not a full implementation of the multipole methodology, as would be done in a full scale MC transport application, but do provide a similar FLOP to load ratio and utilize nearly the same data structure as the full multipole representation. Furthermore, the effects of neutron spin are neglected under the assumption that all neutrons are spin zero. These simplifications are made to reduce the programming complexity of the *RSBench* application, making it easier to instrument and port to new languages and systems, while still retaining a similar performance profile to the full multipole algorithm.

$$\sigma_x(E) = \sigma_o \frac{\Gamma_x}{\Gamma} \sqrt{\frac{E_o}{E}} \frac{\Gamma^2}{4(E - E_o) + \Gamma^2} \tag{1}$$

$$\sigma_s(E) = \sigma_o \sqrt{\frac{E_o}{E}} \frac{\Gamma^2}{4(E - E_o) + \Gamma^2} \left[\frac{\Gamma_n}{\Gamma} + \frac{4(E - E_o)}{\Gamma} \frac{R}{\lambda} \right] + \sigma_{pot} \tag{2}$$

$$\sigma_t(E) = \sum \sigma_x + \sigma_a \tag{3}$$

where

$$\sigma_o = 4\pi\lambda_o^2 \frac{\Gamma_n E_o}{\Gamma} \tag{4}$$

$$\sigma_{pot} = 4\pi R^2 \tag{5}$$

$$\lambda = \Gamma\sigma_o\sqrt{\frac{E_o}{E}} \tag{6}$$

σ_x are the microscopic reaction cross sections (radiative capture and fission), σ_s is the microscopic scattering cross section, σ_t is the total microscopic cross section, σ_o is the peak value of the total resonance cross section, Γ_n, Γ_x, and Γ are, respectively, the width for neutron emission, the width for the reaction x, and the total width of the resonance, E is the energy of the neutron, E_o is the center point energy of the resonance, R is the nuclear radius of the nuclide, λ is the reduced de Broglie wavelength of the neutron, λ_o is the value of the reduced wavelength of the neutron at the resonance peak, and σ_{pot} is the potential scattering cross section for $l = 0$ (s-wave) neutrons.

3 Application

To investigate the performance profiles of our two MC transport cross section algorithms on existing systems, we carried out a series of tests using *RSBench* and *XSBench* on single node, multi-core, shared memory system. The system used was a single 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.

We performed a scaling study to determine performance improvements as additional cores were added. We ran both proxy applications with only a single thread to determine a baseline performance against which efficiency can be measured. Then, further runs were done to test each number of threads between 1 and 32. Efficiency is defined as

$$\text{Efficiency}_n = \frac{R_n}{R_1 \times n} \tag{7}$$

, 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 one core.

The tests reveal that even for these proxy-applications of the MC transport algorithm, perfect scaling was not achievable. Figure 1 and Figure 2 show that efficiency degraded gradually as more cores were used on the nodes. For the Xeon system, efficiency at 16 cores degraded to 69% for *XSBench* and 81% for *RSBench*.

One might reasonably conclude that 69% or 81% efficiency 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[6][7][8][9], 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

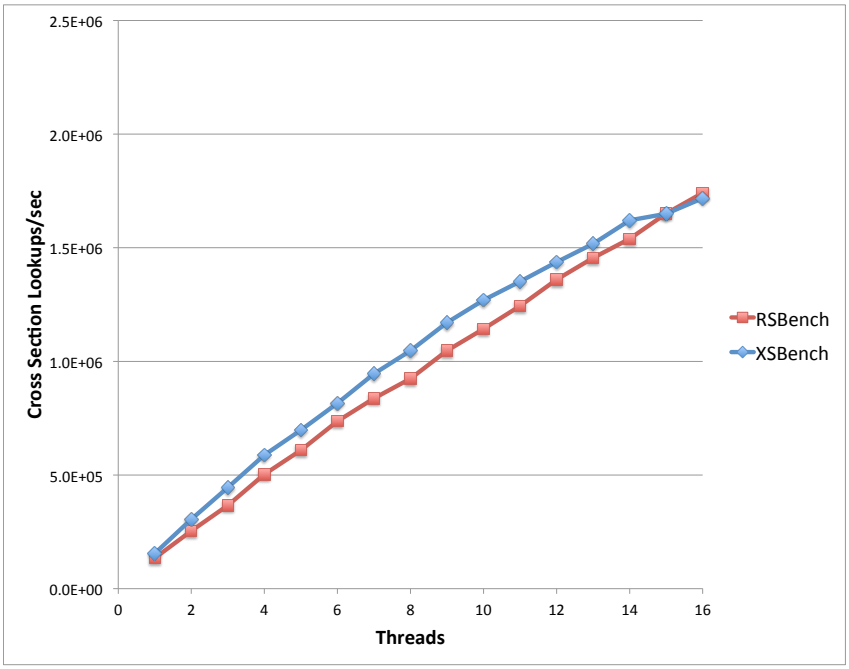

Fig. 1. Performance Scaling

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 two kernels exhaust first.

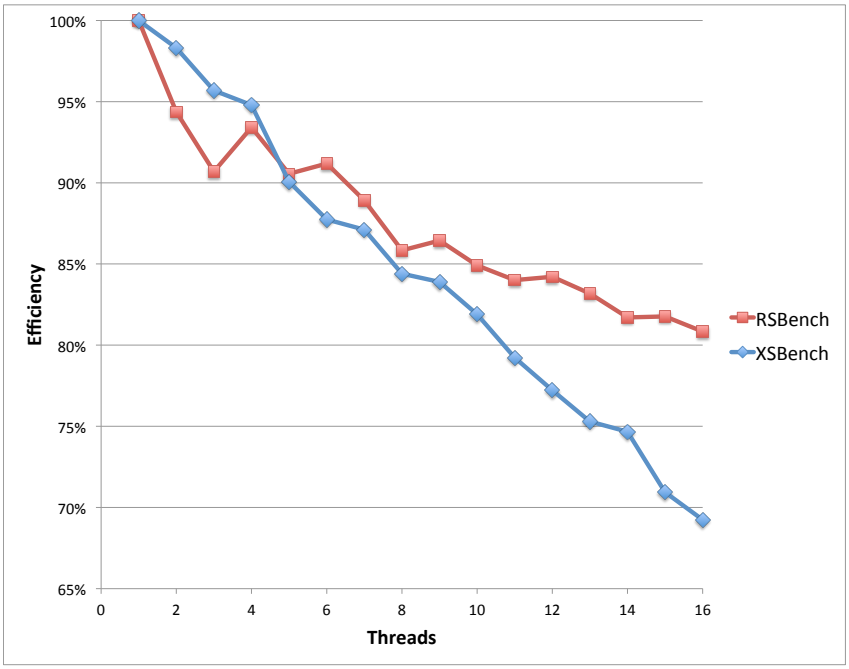

Fig. 2. Efficiency Scaling

4 Experiment & Results

To investigate the performance and resource utilization profiles of both proxy applications, and 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* and *RSBench* codes. 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 3. This allows us to get highly in-depth results as we are able to run experiments dealing with architecture-specific features and hardware counters.

4.1 Resource Usage

To better understand scaling degradation in our kernels, we implemented performance counting features into the source code of *XSBench* and *RSBench* using the Performance Application Programming Interface (PAPI)[16]. 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
- `PAPI_FP_INS` - Floating point instructions

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

Bandwidth Consumption of available system bandwidth resources used by *XSBench* and *RSBench* is calculated using Equation 8.

$$\text{Bandwidth} = \frac{\text{LLC\_MISSES} \times \text{Linesize}}{\text{PAPI\_TOT\_CYC}} \times \text{Clock (Hz)} \tag{8}$$

Using Equation 8, we collected the bandwidth usage for our proxy applications as run on varying numbers of cores, as shown in Figure 3. Note that the maximum theoretically available bandwidth for the Xeon node is 51.2 GB/s[17]. Figure 3 shows that less than half the available bandwidth is ever used by either of our proxy applications, even when running at 32 threads per node[2].

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.[18] Results from this benchmark are shown in Figure 3, and compared to *XSBench* and *RSBench*. As can be seen, *XSBench* converges with *STREAM*, leading us to believe that the classical cross section algorithm is bottlenecked by system bandwidth. In contrast, we find that the bandwidth usage of *RSBench* is much more conservative – using less than half of what *XSBench* does.

FLOPs Consumption of available system floating point resources used by *XSBench* and *RSBench* is calculated using Equation 9.

$$\text{FLOPs} = \frac{\text{PAPI\_FP\_INS}}{\text{PAPI\_TOT\_CYC}} \times \text{Clock (Hz)} \tag{9}$$

Using Equation 9, we were able to determine the FLOP performance of our proxy applications, as shown in Figure 4. We found that *XSBench* achieved at most 3.6 GFLOPs, while *RSBench* achieved over twice the FLOP performance, at 8.4 GFLOPs.

[2] The 16-core Xeon node used in our testing features hardware threading, supporting up to 32 threads per node.

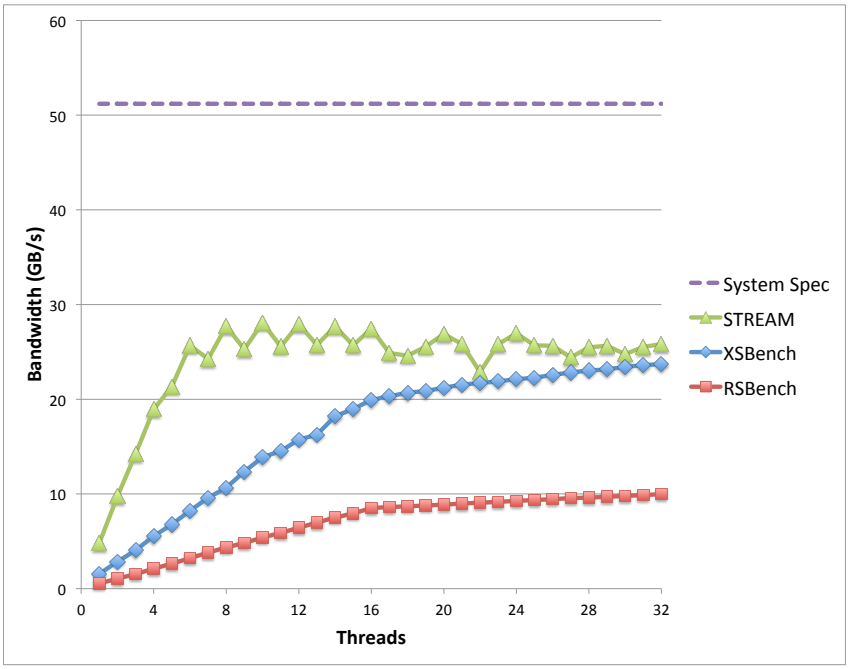

Fig. 3. Bandwidth Usage Scaling

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 10. We found that across all cores, for any number of threads 1 through 16, the LLC miss rate of *XSBench* was 65%, while the LLC miss rate of *RSBench* was only 45%. While those may seem like trivial differences, this factor alone can account for a nearly 50% savings in bandwidth to main memory.

$$\text{Cache Miss } \% = \frac{\text{LLC\_MISSES}}{\text{LLC\_REFERENCES}} \times 100 \tag{10}$$

The significant difference in number of cache misses between the two algorithms can be explained by the differences in Table 1 and Table 2. The data

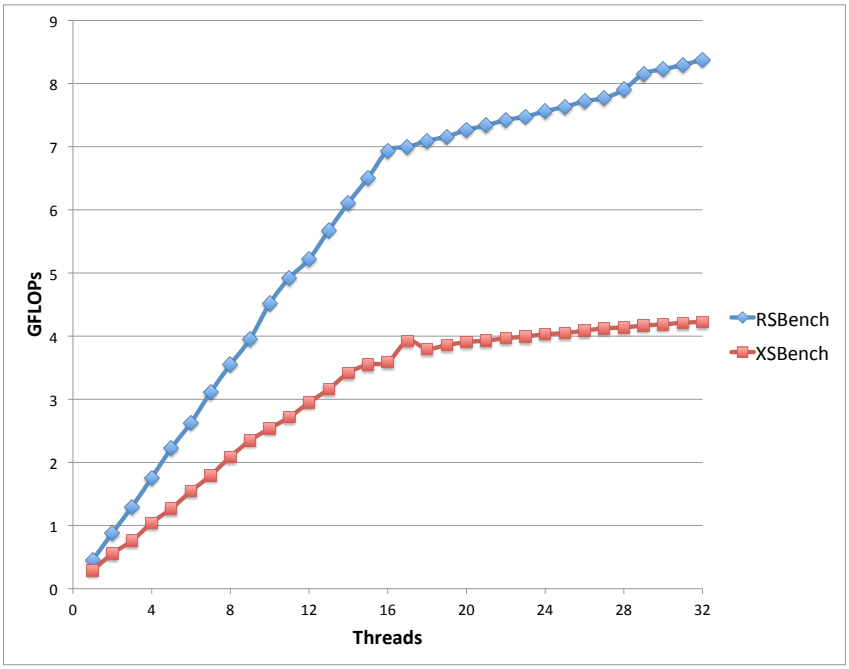

Fig. 4. FLOP Usage

structure used in the multipole algorithm is small enough, at 41 MB, to have a large portion of it fit into cache.

5 Conclusions

We have performed an in-depth analysis of two different implementations of the MC neutron transport algorithm – the "classical" continuous energy cross section format (i.e., *XSBench*) and the multipole representation format (i.e., *RSBench*). We have also assessed the on-node scaling properties and memory contention issues of these algorithms in the context of a robust reactor physics calculation.

Through our investigations of the classical MC neutron cross section lookup algorithm, via *XSBench*, we found that it achieves bandwidth usage extremely close to the practical maximum of 25.8 GB/s when running 32 threads per node. 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 for this algorithm.

We also found that the multipole algorithm (i.e., *RSBench*) uses less than half as much bandwidth and achieves over twice the FLOP performance when

compared to the classical algorithm. This is a more desirable performance profile for scaling on next-generation systems, as processor cores per node and computational capacity is expected to greatly outpace increases in bandwidth to main memory. Furthermore, the multipole algorithm requires less latency masking hardware as memory loads and cache misses are less frequent, meaning that threads can be kept busier than in the classical algorithm. Finally, the multipole algorithm has been shown to be capable of better scaling (81% efficiency at 16 cores per node vs. 69% for the classical algorithm), which also suggests better potential for on-node scaling in next-generation, many core systems.

6 Future Work

There are additional capabilities that do not yet commonly 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 to implement in *XSBench* and *RSBench* for experimentation with various hardware architectures and features. This addition is predicted to enhance the advantages of the multipole algorithm as Doppler broadening is an inherently easier task when cross section data is already stored in the multipole format.

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. Forget, B., Xu, S., smith, k.: Annals of Nuclear Energy. Annals of Nuclear Energy **64**(C) (February 2014) 78–85
2. Romano, P.K., Forget, B.: The OpenMC Monte Carlo particle transport code. Annals of Nuclear Energy **51**(C) (January 2013) 274–281
3. Romano, P.K., Forget, B., Brown, F.B.: Towards scalable parallelism in monte carlo particle transport codes using remote memory access. (2010) 17–21
4. Siegel, A.R., Smith, K., Romano, P.K., Forget, B., Felker, K.G.: Multi-core performance studies of a Monte Carlo neutron transport code. International Journal of High Performance Computing Applications (March 2013) 1–25

5. Tramm, J., Siegel, A.R.: Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes. In: Joint International Conference on Supercomputing in Nuclear Applications + Monte Carlo, Paris, Argonne National Laboratory (October 2013)
6. Dosanjh, S., Barrett, R., Doerfler, D., Hammond, S., Hemmert, K., Heroux, M., Lin, P., Pedretti, K., Rodrigues, A., Trucano, T., Luitjens, J.: Exascale design space exploration and co-design. Future Generation Computer Systems (0) (2013) –
7. Attig, N., Gibbon, P., Lippert, T.: Trends in supercomputing: The european path to exascale. Computer Physics Communications **182**(9) (2011) 2041 – 2046
8. Rajovic, N., Vilanova, L., Villavieja, C., Puzovic, N., Ramirez, A.: The low power architecture approach towards exascale computing. Journal of Computational Science (0) (2013) –
9. Engelmann, C.: 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 (0) (2013) –
10. : Openmc monte carlo code. https://github.com/mit-crpg/openmc (January 2014)
11. : Xsbench: The monte carlo macroscopic cross section lookup benchmark. https://github.com/jtramm/XSBench (January 2014)
12. : Rsbench: A mini-app to represent the multipole resonance representation lookup cross section algorithm. https://github.com/jtramm/RSBench (January 2014)
13. Hoogenboom, J.E., Martin, W.R., Petrovic, B.: MONTE CARLO PERFORMANCE BENCHMARK FOR DETAILED POWER DENSITY CALCULATION IN A FULL SIZE REACTOR CORE Benchmark specifications. Ann Arbor **1001** (2010) 48109–42104
14. Romano, P.K., Siegel, A.R., Forget, B., smith, k.: Data decomposition of Monte Carlo particle transport simulations via tally servers. JOURNAL OF COMPUTATIONAL PHYSICS **252**(C) (November 2013) 20–36
15. Leppänen, J.: Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation. Annals of Nuclear Energy **36**(7) (July 2009) 878–885
16. : Papi - performance application programming interface. http://icl.cs.utk.edu/papi/index.html (September 2013)
17. Intel®: Xeon® processor e5-2650 cpu specifications. http://ark.intel.com/products/64590/ (September 2013)
18. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (December 1995) 19–25