Parallel Performance Results for the OpenMOC Neutron Transport Code on Multi-Core Platforms

William Boyd[a], Andrew Siegel[b], Shuo He[b], Benoit Forget[a], Kord Smith[a]

[a]*Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139*
[b]*Argonne National Laboratory, 9700 S. Cass Ave., Lemont, IL 60439*

Abstract

The shift towards multi-core architectures has ushered in a new era of shared memory parallelism for scientific applications. This transition has introduced challenges for the nuclear engineering community as it seeks to design high-fidelity full-core reactor physics simulation tools. This paper describes the parallel transport sweep algorithm in the OpenMOC method of characteristics (MOC) neutron transport code for multi-core platforms using OpenMP. Strong and weak scaling studies are performed for both Intel Xeon and IBM Blue Gene/Q multi-core processors. The results demonstrate 100% parallel efficiency for 12 threads on 12 cores on Intel Xeon platforms, and over 90% parallel efficiency with 64 threads on 16 cores on the IBM Blue Gene/Q. These results illustrate the potential for hardware acceleration for MOC neutron transport on modern multi-core and future many-core architectures. In addition, this work highlights the pitfalls of programming for multi-core architectures, with a focal point on false sharing.

Keywords: Multi-core processors, OpenMP, neutron transport, method of characteristics

1. Introduction

It is well known that the end of clock speed scaling has forced high-performance computing (HPC) applications to leverage increasing degrees

Email addresses: wboyd@mit.edu (William Boyd), siegela@mcs.anl.gov (Andrew Siegel), shuohe@anl.gov (Shuo He), bforget@mit.edu (Benoit Forget), kord@mit.edu (Kord Smith)

of on-node parallelism to efficiently utilize modern processors. This on-node parallelism can take many forms, including a relatively small number of sophisticated MIMD-capable cores, nodes with larger core counts tailored for SIMD-style execution, or heterogeneous systems which combine complementary strategies. The programming models used to express algorithms on these architectures are similarly varied, including loops with explicit or compiler-generated vector instructions, directive-based languages which encourage coarse-grained parallelism (*e.g.*, OpenMP (1)), and accelerator-based models with explicit control of host/device memory which encourage fine-grained parallelism (*e.g.*, CUDA (2)).

Although many of these concepts are rooted in familiar technologies from previous generations, the HPC community's understanding of the "best practices" for optimizing parallel application performance is still much less mature than it is for distributed memory systems with a message passing programming model. Some of the major challenges that have been introduced with on-node parallelism include:

- Insufficent hardware and programming model controls to navigate memory contention on shared memory systems
- Insufficient fine-grained parallelism inherent to some algorithmic formulations
- Limitations based on Amdahl's Law for accelerator-based systems
- Smaller memory footprints per floating point unit
- Ineffective use of shared and distributed caches on cache-coherent systems

A deep understanding of the limitations of on-node parallel performance potentially goes beyond enabling more efficient implementations of existing methods. In some cases, inevitable trends in node architectures may favor alternative mathematical formulations or physical models to describe relevant physical systems.

One such example is the field of nuclear reactor physics, where neutron transport simulations are used to **solve the 7-dimensional neutron transport equation to find the neutron distribution in space, angle, energy and time (3)**. The neutron distribution can be used to predict criticality, power distributions, and temperature profiles in a specified reactor core. Numerical approaches to solving the neutron transport equation have a long and rich history dating back to the 1950's (4). At the highest level, two broad classes of approaches can be identified – *stochastic* (Monte Carlo) and *deterministic* methods.

Parallel algorithms for Monte Carlo methods on distributed memory systems has been a vibrant area of research for many decades, including recent advances in distributed fission banks (5) and spatial domain decomposition (6). However, studies in on-node parallelism have pointed to some key issues – scaling limitations due to memory contention (7; 8), and the difficulty of formulating Monte Carlo approaches with SIMD parallelism (9).

The majority of efforts to parallelize deterministic methods have focused on the discrete ordinates or S_N algorithm (10; 11). Less well understood, however, is parallel performance for the method of characteristics (MOC), one of the most common methods with real world applications in production 2D lattice physics tools used today (12; 13). **Although it is unclear whether MOC is a viable competitor to continuous energy Monte Carlo for whole core neutron transport, there have been a number of recent efforts to implement 3D MOC-based simulation tools, including the MPACT (14) and nTracer (15) codes.** In order to better understand MOC's viability for future **whole core** nuclear reactor analysis, MOC's ability to efficiently utilize systems with large degrees of on-node parallelism must be demonstrated. Although MOC has been parallelized with OpenMP (16; 17) and CUDA (18), to the authors' knowledge, no published work presents a comprehensive evaluation of the parallel performance of MOC on multi-core platforms.

This paper is a case study on the parallelization of the method of characteristics for multi-core processors using OpenMP. Section 2 presents the method of characteristics and its application to nuclear reactor physics calculations in the OpenMOC code (19). Section 3 discusses the methodology taken to parallelize the transport sweep in OpenMOC using OpenMP. Section 4 defines the parallel performance metrics used to evaluate the MOC implementation. Section 5 presents parallel performance results for both weak and strong scaling studies on Intel Xeon and IBM Blue Gene/Q multi-core processors. Finally, Section 6 illustrates the subtle nature of multi-core programming with a focus on how false sharing led to poor performance for one of OpenMOC's solvers.

2. Method of Characteristics

2.1. General Overview

The method of characteristics is a widely used technique for solving partial differential equations, including the Boltzmann form of the neutron transport equation. Though not a stochastic formulation, MOC is a *ray-based* algorithm akin to Monte Carlo particle tracking-based methods. In contrast to Monte Carlo, MOC uses a fixed angular quadrature which is determined *a priori*. This quadrature is used to specify 1D characteristics which cross the spatial domain. Prior to the physics computation, ray tracing must be performed to sub-divide each characteristic into *segments* within different regions in the spatial mesh. **Figure 1 illustrates the spatial mesh and cyclic characteristic laydown used in many 2D MOC neutron transport codes for light water reactor analysis.**

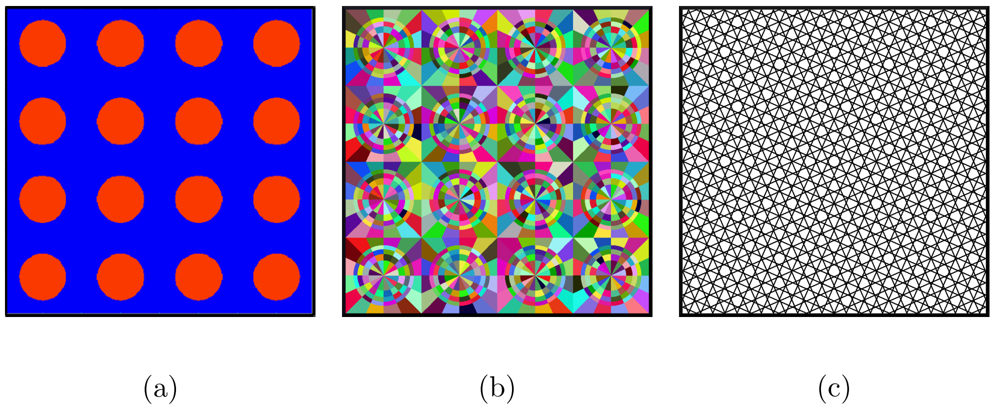


Figure 1: The coolant and fuel materials (a), method of characteristics flat source region spatial mesh (b), and cyclic characteristic laydown (c) for a 4 × 4 fuel pin lattice.

MOC propagates the angular neutron flux along each characteristic through each spatial zone. For each segment, the angular flux is attenuated due to neutron absorption and enhanced due to neutron fission or scattering in the corresponding spatial zone. MOC uses the multi-group energy approximation such that this computation is performed for neutrons within discretized *energy groups*. Finally, an angular quadrature is applied to combine the average angular flux contribution from each characteristic to compute the average scalar flux in each zone and energy group.

2.2. Theory and Derivation

The detailed theory and derivation of the method of characteristics applied to neutron transport is widely available in the literature (20). This section provides a brief overview of the MOC equuations, highlighting those points which are most relevant for the multi-core implementation.

MOC is used to solve the transport equation by discretizing polar and azimuthal angles and integrating the multi-group characteristic form of the equation for a particular azimuthal and polar angle quadrature. The multi-group form of the steady-state Boltzmann neutron transport equation is presented below,

$$\mathbf{\Omega} \cdot \nabla \Psi_g(\mathbf{r}, \mathbf{\Omega}) + \Sigma_g^T(\mathbf{r}) \Psi_g(\mathbf{r}, \mathbf{\Omega}) = Q_g(\mathbf{r}, \mathbf{\Omega}) \tag{1}$$

where $\mathbf{r}$ is the spatial position vector, $\mathbf{\Omega}$ is the angular direction vector, g is the energy group index, $\Psi_g(\mathbf{r}, \Omega)$ is the angular neutron flux, $\Sigma_g^T(\mathbf{r})$ is the macroscopic total nuclear cross-section and $Q_g(\mathbf{r})$ is the neutron source term. In MOC, this equation is discretized with respect to azimuthal and polar angles m and p along 1D characteristics k parametrized by s.

In addition, most MOC formulations spatially discretize the simulation domain into *Flat Source Regions* (or *FSRs*). **In a typical light water reactor simulation, each fuel pin may be discretized in 3-5 cylindrical rings and 8-16 angular sectors (as illustrated in Figure 1) in order to capture the spatial variation of the neutron flux to accurately predict the rate of nuclear fission within each fuel pin. In the formulation presented here,** each FSR i has a constant neutron source which results in the system of ODEs solved by MOC:

$$\frac{d\Psi_{k,g}(s)}{ds} + \Sigma_{i,g}^T \Psi_{k,g}(s) = Q_{i,g} \tag{2}$$

The neutron source term $Q_{i,g}$ is defined in terms of the fission and isotropic scattering from the average neutron scalar flux $\Phi_{i,g}$ in each FSR:

$$Q_{i,g} = \frac{1}{4\pi} \left(\sum_{g'=1}^{G} \Sigma_{i,g' \to g}^S \Phi_{i,g'} + \frac{\chi_{i,g}}{k_{eff}} \sum_{g'=1}^{G} \nu \Sigma_{i,g'}^F \Phi_{i,g'} \right) \tag{3}$$

where $\Sigma_{i,g' \to g}^S$ is the scattering cross-section for group g' to group g, $\Sigma_{i,g}^F$ is the fission cross-section, ν is the average number of neutrons produced from fission, $\chi_{i,g}$ is the fraction of neutrons produced in group g from fission and k_{eff} is the neutron multiplication factor.

Next, an integrating factor is defined in terms of the optical length $\tau_{k,i,g} = \Sigma_{i,g}^T(s'' - s')$ for characteristic k across FSR i from its entry point at s' to exit point at s''. Now the change in the angular flux along the characteristic segment in FSR i may be expressed as follows:

$$\Delta\Psi_{k,i,g} = \Psi_{k,g}(s') - \Psi_{k,g}(s'') = \left(\Psi_{k,g}(s') - \frac{Q_{i,g}}{\Sigma^T_{i,g}}\right)(1 - e^{-\tau_{k,i,g}}) \quad (4)$$

The key quantity remaining to be determined is the area-averaged scalar flux $\Phi_{i,g}$ in each FSR i and energy group g. The scalar flux is needed to compute nuclear reaction rates - most notably, fission rates - in various regions of a nuclear reactor core. The discrete ordinates approximation is applied to approximate the scalar flux from the angular flux contributions of each characteristic segment crossing **FSR i with 2D cross-sectional area A_i (or volume if in 3D)** as follows:

$$\Phi_{i,g} = \frac{4\pi}{\Sigma_{i,g}}\left[Q_{i,g} + \frac{1}{A_i}\sum_{k\in A_i}\omega_k\Delta\Psi_{k,i,g}\right] \quad (5)$$

This is the form of the transport equation solved by the MOC formulation used in work. It should be noted that the quadrature weight w_k for characteristic k is a composition of weights w_m and w_p for the azimuthal and polar angle quadratures, respectively.

2.3. OpenMOC

The OpenMOC method of characteristics neutron transport code (19) was used in this work. OpenMOC is an open source, object-oriented 2D method of characteristics code written in C/C++ with a Python API. OpenMOC uses general constructive solid geometry in a similar fashion to many other widely used neutron transport codes, and is capable of solving 2D full-core Pressurized Water Reactor (PWR) problems. In addition, OpenMOC includes routines for nonlinear Coarse Mesh Finite Difference (CMFD) acceleration. Most relevant to this work, however, is that OpenMOC incorporates parallel solvers for shared memory multi-core CPUs.

2.4. Exponential Evaluation Methods

The most expensive operation in the MOC equations is the exponential evaluation needed to compute $e^{-\tau_{k,i,g}}$ in Equation 4. All mainstream compilers provide a library with an intrinsic `exp(...)` routine. One method of avoiding the computational cost of explicitly evaluating exponentials is through the use of a linear interpolation table (21). The linear interpolation table may be constructed to compute exponentials to the desired level of accuracy on a fixed input interval (*e.g.*, 1E-3 relative error for arguments in

$[-10, 0]$). In addition to reducing the flop count for an exponential evaluation, the table is typically small enough (*e.g.*, 20 KB) to fit completely in L1 cache and can improve the memory performance of the MOC transport sweep algorithm. OpenMOC incorporates an option to evaluate exponentials using either the compiler's exponential intrinsic function or a linear interpolation table (22). The performance for each method on multi-core platforms is presented in this paper.

3. OpenMOC Transport Sweep

OpenMOC includes a solver implementation which integrates the angular flux across the geometry for each characteristic as described in Section 2.2. In particular, OpenMOC uses a nested power iteration scheme to solve for the sources $Q_{i,g}$ and scalar fluxes $\Phi_{i,g}$. The inner iteration solves for an approximate scalar flux in each FSR assuming a fixed source **according to Equation 5**. The outer iteration computes an updated source based on the inner iteration's approximation to the flux that results from the fixed source **according to Equation 3**. The inner iteration to compute $\Phi_{i,g}$ for all FSRs and energy groups will henceforth be referred to as the *transport sweep*. The transport sweep consumes over 95% of OpenMOC's simulation runtime and was the focus of this work.

3.1. Sequential Version

A description of the sequential formulation of OpenMOC's transport sweep is given by Algorithm 1. The transport sweep integrates the angular flux across the geometry for each characteristic according to Equation 4 and Equation 5. A single transport sweep involves nested loops over azimuthal **and polar** angles, characteristics, segments in different FSRs and energy groups.

It should be noted that a number of different variations of Algorithm 1 have been presented in the literature for MOC. In particular, some codes make the outermost loop over energy groups as a Gauss-Seidel approach to update to the flux in energy (17). However, data structures for the flux, source and cross-sections are typically stored with stride one in energy, which are best cached when energy is the innermost loop in the algorithm as presented here. In addition, most MOC codes, including OpenMOC, use nonlinear diffusion acceleration (such as Coarse Mesh Finite Difference Acceleration (23)), which dramatically reduces

the number of iterations beyond what is achievable with a Gauss-Seidel-like approach. It is for these reasons that Algorithm 1 is used as the basis for the present analysis.

Algorithm 1 Transport Sweep Algorithm

```
1: Zero FSR scalar fluxes
2: for all characteristics do
3:     for all segments do
4:         for all energy groups do
5:             Compute change in angular flux            ▷ Equation 4
6:             Increment FSR scalar flux                 ▷ Equation 5
7:         end for
8:     end for
9: end for
```

3.2. Parallel Version with Atomics

The parallel implementation of the transport sweep algorithm applied the `#pragma omp for` declarator to the outermost loop over characteristics in OpenMOC's transport sweep algorithm. Work was load-balanced among threads using OpenMP's `guided` scheduling directive. Two versions of the transport sweep were implemented with and without mutual exclusion locks for the scalar flux update.

The version with mutual exclusion made use of locks during the FSR scalar flux update on the innermost loop (see Algorithm 2). This version contained an array of OpenMP `omp_lock_t` variables for each FSR. When a thread computed the scalar flux contribution from a particular segment s to FSR i, the i^{th} lock was applied such that no other thread could increment the scalar flux for that FSR until the lock was released.

Although fine-grained mutual exclusion within an innermost loop typically might be a cause for concern, it was not expected to lead to significant performance degradation in this case. The atomic region was applied to each FSR, and the number of FSRs (10^5 – 10^7) is much greater than the number of threads ($\sim 10^2$) in a typical MOC calculation. Hence, from a purely statistical standpoint, the number of characteristic segments which result in two threads attempting to increment the same FSR scalar flux at the same time should be negligible. This issue was later addressed with numerical experiments presented in Section 6.

Algorithm 2 Parallel Transport Sweep Algorithm with Atomics

```
1: Zero FSR scalar fluxes
2: for all characteristics do in parallel
3:     for all segments do
4:         for all energy groups do
5:             Compute change in angular flux                ▷ Equation 4
6:             begin atomic
7:                 Increment FSR scalar flux                 ▷ Equation 5
8:             end atomic
9:         end for
10:    end for
11: end parallel for
```

3.3. Parallel Version without Atomics

A second parallel version of OpenMOC's transport sweep was implemented to avoid mutual exclusion. This version made use of private scalar flux arrays for each thread (see Algorithm 3). In this version, the scalar flux contribution from segment s to FSR i was tallied to each thread's private array of scalar fluxes rather than a shared array. At the conclusion of the transport sweep, the private FSR scalar fluxes were *reduced* across threads into the global scalar flux array.

Algorithm 3 Parallel Transport Sweep Algorithm without Atomics

```
1: Zero FSR scalar fluxes
2: Zero threadprivate FSR scalar fluxes
3: for all characteristics do in parallel
4:     for all segments do
5:         for all energy groups do
6:             Compute change in angular flux                ▷ Equation 4
7:             Increment threadprivate FSR scalar flux       ▷ Equation 5
8:         end for
9:     end for
10: end parallel for
11: Reduce threadprivate scalar fluxes
```

The disadvantage of using private scalar flux arrays is that this approach resulted in a larger memory footprint than the version with mutual exclusion. For all cases studied in this analysis, however, the extra memory requirements did not present a practical obstacle as the storage requirements for characteristic segments dominated those for the FSR scalar fluxes. For future shared memory many-core systems with 100s of cores, however, redundant storage of the scalar fluxes will likely be infeasible.

4. Measuring Parallel Performance

The parallel performance of OpenMOC's multi-threaded solver was evaluated on several different multi-core processors. Section 4.1 introduces weak and strong scaling and their application to the transport sweep algorithm. Section 4.2 defines some useful metrics for assessing MOC performance.

4.1. Scaling Studies

In *weak scaling*, the amount of work per thread is kept constant while the total work is directly proportional to the number of parallel threads. **Ideally, the solution time will remain constant such that the weak scaling "speedup" is equal to unity.** A simple way to perform a weak scaling study in MOC is to set the number of azimuthal angles proportional to the number of threads. The number of characteristics is roughly proportional to the number of azimuthal angles, scaling the amount of work available to all threads via the outermost loop in the transport sweep algorithm.

In *strong scaling*, the total work is kept constant while the work per thread is inversely proportional to the number of parallel threads. Ideally, up to the number of threads available on the system, the runtime will decrease inversely with increasing thread count. Strong scaling is straightforward to run in OpenMOC by simply selecting enough azimuthal angles to provide enough work to hide thread launch latency.

4.2. Performance Metrics

OpenMOC uses the POSIX timer interface for recording sub-millisecond resolution measurements of the total time T to converge the source. When computing parallel performance metrics for OpenMOC, the total time to solution T is normalized to the number of characteristic segments n_s, source iterations n_i and threads n_t as follows:

$$T_n(n_t) = \frac{T}{n_s \times n_i \times n_t} \tag{6}$$

The number of segments n_s varied for weak scaling but was fixed for strong scaling. The power iteration scheme converged to the same solution in the same number of iterations $n_{i,conv}$ irregardless of the thread count. However, for practical reasons the number of iterations n_i was fixed such that $n_i < n_{i,conv}$ to permit timely execution of the parallel performance studies.

The first common metric for evaluating parallel performance and scaling is the *parallel speedup* $S(n_t)$. The speedup is used to compare the ratio of the runtime to solve a problem with n_t threads, $T_n(n_t)$, with respect to the time $T_n(1)$ to solve the same problem with a single thread. The parallel speedup for n_t threads takes the following form:

$$S(n_t) = \frac{T_n(1)}{T_n(n_t)} \tag{7}$$

A second useful metric is the *integration rate* $R(n_t)$. The integration rate is the number of inner iterations of the transport sweep algorithm performed per second. Alternatively, the integration rate is the number of characteristic segments processed per second. This is simply the inverse of the normalized time to solution in Equation 6:

$$R(n_t) = \frac{1}{T_n(n_t)} \tag{8}$$

5. OpenMOC Multi-Core Performance Results

A series of weak and strong scaling studies were performed using OpenMOC's multi-threaded solvers. The parallel speedups $S(n_t)$ and integration rates $R(n_t)$ from these studies are presented for Intel Xeon and IBM Blue Gene/Q (BG/Q) platforms in Section 5.2. Some of the key performance trends are noted in Section 5.3.

5.1. Benchmark Problem

All of the studies used the 2D C5G7 benchmark problem (24), described in Section 5.1. The C5G7 problem was developed by the OECD's Nuclear Energy Agency as a modern benchmark for deterministic neutron transport methods without spatial homogenization. Figure 2 illustrates the materials in the problem and the flat source regions used in the spatial model for OpenMOC. The problem contains four 17×17 pin cell assemblies next to a water reflector. The bundles on the top left and bottom right contain UO_2

fuel while the ones on the opposite two corners are MO_X assemblies. The C5G7 problem includes seven different materials, each with seven energy group nuclear cross-section data. A model of the C5G7 problem was built with three radial and eight angular subdivisions per fuel pin and 1.26 mm - 1.26 cm Cartesian mesh in the reflector for a total of 142,964 FSRs.

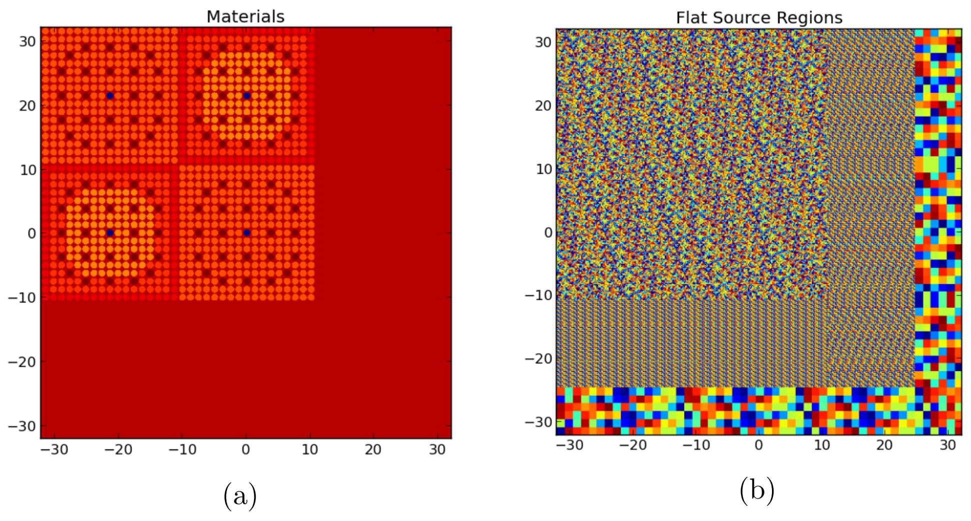


Figure 2: Materials composition (a) and flat source region spatial discretization (b) for the C5G7 benchmark problem.

5.2. Performance Results

A series of weak and strong scaling studies were performed across a sequence of parameters on Intel Xeon and IBM BG/Q multi-core processors as described in Section 5.2.1 and Section 5.2.2, respectively. In particular, experiments were performed both with and without atomics in the transport sweep, using both linear interpolation and exponential intrinsic methods for evaluating the exponential. In addition, the GNU and Intel compilers were evaluated on the Xeon, while GNU and IBM compilers were examined on the BG/Q.

Each weak scaling study varied the number of threads in integral steps from 1 to $n_{t,max}$ while increasing the number of azimuthal angles accordingly in steps of 4 (*i.e.*, 4 to $4n_{t,max}$). Each strong scaling study varied the number of threads in integral steps from 1 to $n_{t,max}$ while holding the number of azimuthal angles constant at $4n_{t,max}$. A total of 10 transport sweep iterations were performed for each data point. **On both architectures,**

the threads were distributed using OpenMP's default `SCATTER` protocol without explicit pinning of threads to particular cores.

5.2.1. Intel Xeon

The following results were recorded on two Intel i7/E5-2620 (Ivy Bridge) processors with six cores clocked at 2.5 GHz along with 24 GB of memory clocked at 1066 MHz. Intel Hyper-threading™ was enabled to provide a total of 24 hardware threads. The processor cache hierarchy included private L1 (348 KB) and L2 (1536 KB) caches with a shared L3 (15 MB) cache.

OpenMOC was compiled with version 4.4.6 of GNU's `g++` compiler with `-O3` for optimizations and `-ffast-math` for fast math routines. A second version of OpenMOC was compiled with version 13.1.0 of Intel's `icpc` compiler, with the flags `-O3` for optimizations and `-fast` for fast math routines.

The speedups $S(n_t)$ for weak and strong scaling are presented in Figure 3 and Figure 4, respectively. Likewise, the integration rate $R(n_t)$ (Equation 8) for each combination of parameters for 1, 6, 12 and 24 threads is presented in Table 1 and Table 2 for weak and strong scaling, respectively.

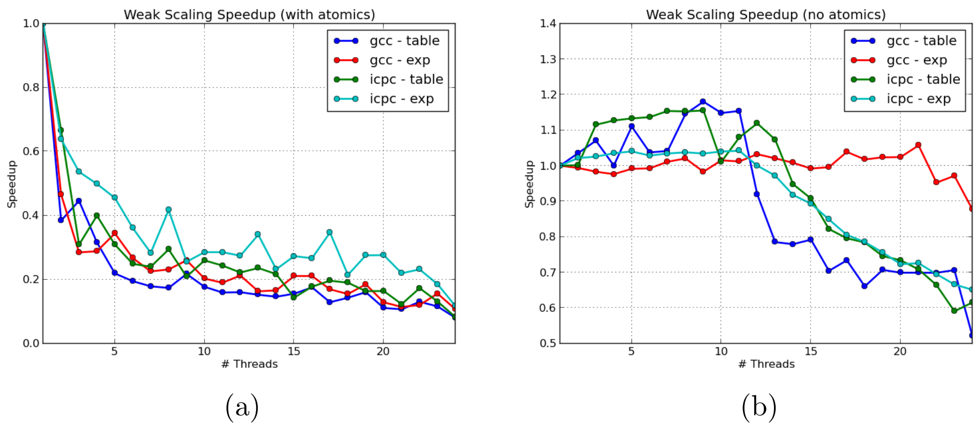
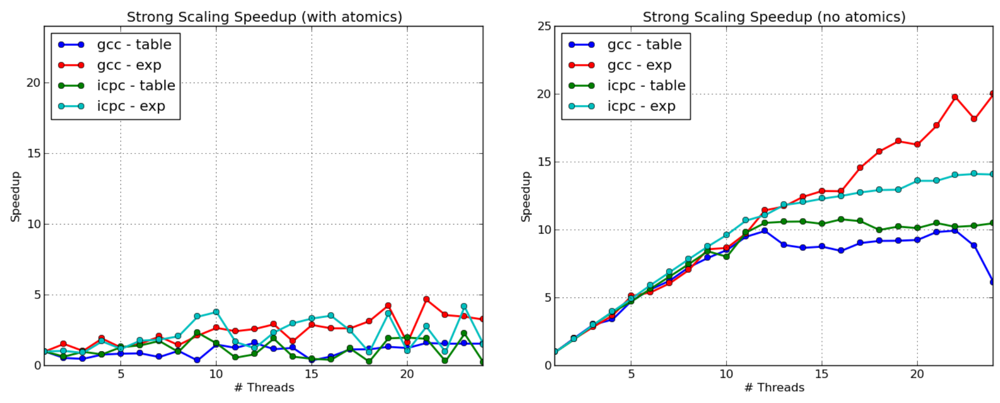


Figure 3: Weak scaling speedup with (a) and without (b) atomics on Intel Xeon.

A number of key observations can be made based upon these results. First, a dramatic difference was observed in the parallel scalability for the two solver implementations as illustrated by Figure 3 and Figure 4. In particular, the solver implementation with threadprivate variables scaled well to large thread counts, achieving $10 - 12\times$ strong scaling speedup and $0.9 - 1.1\times$ weak scaling speedup with 12 threads. However, the version with mutual exclusion saw little benefit from parallelism, with only a $1 - 3\times$ speedup from 1 to 12 threads for both weak and strong scaling.



(a) (b)

Figure 4: Strong scaling speedup with (a) and without (b) atomics on Intel Xeon.

		# Threads			
Atomics	**Compiler**	**1**	**6**	**12**	**24**
		Linear Interpolation			
Yes	GNU	35.1	40.9	67.3	67.4
Yes	Intel	35.4	52.8	94.2	70.1
No	GNU	40.4	251	446	505
No	Intel	45.6	311	612	673
		Exponential Intrinsic			
Yes	GNU	12.6	20.2	32.2	32.4
Yes	Intel	12.8	27.7	42.1	35.6
No	GNU	8.01	47.7	99.2	169
No	Intel	14.3	88.3	172	224

Table 1: Weak scaling integration rates $R(n_t)$ on Intel Xeon (in millions).

Second, as can be noted in Table 1 and Table 2, the Intel compiler non-trivially outperformed the GNU compiler in terms of the integration rate $R(n_t)$ for nearly every configuration. This observation was particularly relevant for the solver implementation without mutual exclusion, which consistently outperformed GNU by a factor of $1.3 - 1.7$ with $12 - 24$ threads. In addition, the Intel-compiled version scaled more "smoothly" with increasing threads – especially with Hyper-threading – while GNU's version displayed more erratic behavior with increasing thread count.

A final observation is that the linear interpolation method was $3 - 6\times$ faster than the exponential intrinsic for evaluating exponentials. This was

		# Threads			
Atomics	**Compiler**	**1**	**6**	**12**	**24**
		Linear Interpolation			
Yes	GNU	54.9	47.7	88.3	85.4
Yes	Intel	29.7	68.1	38.2	9.68
No	GNU	57.6	324	572	353
No	Intel	65.4	371	688	687
		Exponential Intrinsic			
Yes	GNU	9.67	15.0	25.0	31.5
Yes	Intel	14.7	25.9	18.1	21.7
No	GNU	8.57	46.2	98.0	171
No	Intel	15.9	94.3	176	224

Table 2: Strong scaling integration rates $R(n_t)$ on Intel Xeon (in millions).

expected due to the intrinsic's larger flop count and the interpolation table's cache efficiency as described in Section 2.4. Interestingly, the transport sweep without atomics but with the exponential intrinsic scaled well with Hyper-threading - achieving 20× and 14× strong scaling speedup with the GNU and Intel compilers, respectively.

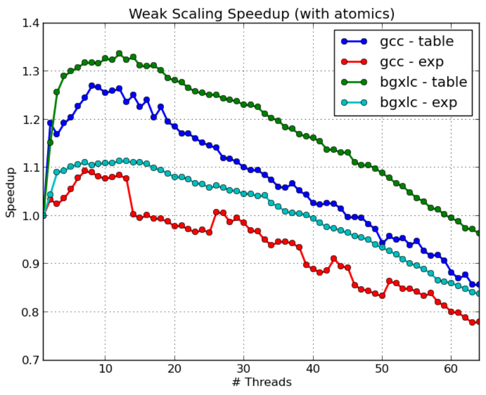
5.2.2. IBM Blue Gene/Q

The following results were recorded from runs performed on Vesta, a test/development rack of the IBM Blue Gene/Q (BG/Q) owned and managed by the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory. Each node included a single PowerPC A2 processor with 16 cores each clocked at 1.6 GHz and 16 GB of memory clocked at 1.33 GHz. All cores included 4-way hardware multi-threading. The processor cache hierarchy included local L1 (32 KB) and shared L2 (32 MB) caches.

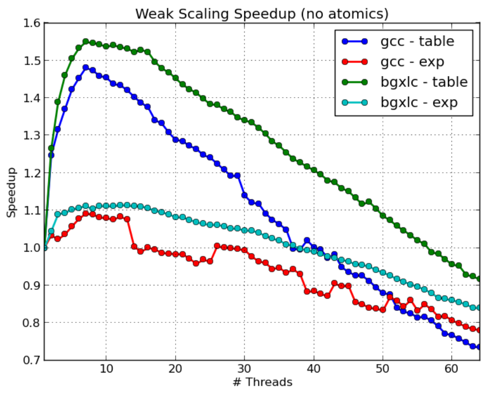
OpenMOC was compiled with version 4.4.6 of GNU's `g++` compiler with `-O3` for optimizations and `-ffast-math` for fast and approximate math routines. A second version of OpenMOC was compiled with version 12.1 of IBM's `bgxlc++_r` compiler, with the flags `-O2`, `-qtune=qp` and `-qunroll=auto` for optimizations.

The speedups $S(n_t)$ for weak and strong scaling are presented in Figure 5 and Figure 6, respectively. Likewise, the integration rate $R(n_t)$ (Equation 8) for each combination of parameters for 1, 4, 16, and 64 threads is presented in Table 3 and Table 4 for weak and strong scaling, respectively.

A first observation is that the solver implementation with mutual exclu-

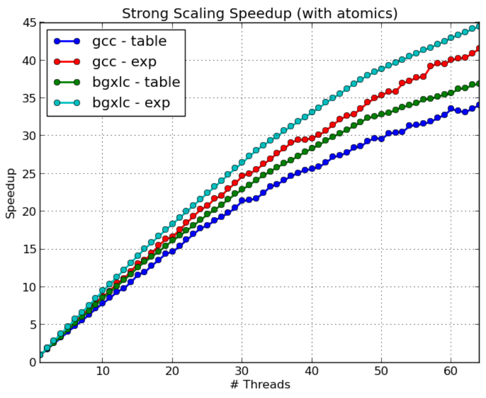


(a)

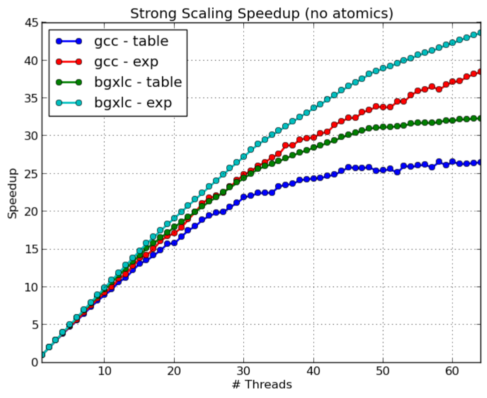


(b)

Figure 5: Weak scaling speedup with (a) and without (b) atomics on IBM BG/Q.



(a)



(b)

Figure 6: Strong scaling speedup with (a) and without (b) atomics on IBM BG/Q.

sion scaled nearly the same as the version with threadprivate variables as illustrated by Figure 5 and Figure 6. In particular, the version with mutual exclusion achieved $35 - 45\times$ strong scaling speedup and $0.8 - 0.9\times$ weak scaling speedup with 64 threads on 16 cores. Although mutual exclusion did not impact parallel scaling as it did on the Xeon, it did appear to dramatically degrade the performance as shown in Table 3 and Table 4. Indeed, the integration rates were $10\times$ faster for the version without mutual exclusion.

Second, as noted in Table 3 and Table 4, the IBM compiler consistently outperformed the GNU compiler in terms of the integration rate. However, the disparity between the two compilers was less remarkable than that

		# Threads			
Atomics	**Compiler**	**1**	**4**	**16**	**64**
Linear Interpolation					
Yes	GNU	0.351	1.67	6.96	19.2
Yes	IBM	0.354	1.82	7.41	21.8
No	GNU	3.99	21.9	87.8	188
No	IBM	3.91	22.8	95.3	229
Exponential Intrinsic					
Yes	GNU	0.164	0.679	2.62	8.16
Yes	IBM	0.159	0.695	2.82	8.54
No	GNU	1.64	6.79	26.2	81.7
No	IBM	1.59	6.95	28.2	85.4

Table 3: Weak scaling integration rates $R(n_t)$ on IBM BG/Q (in millions).

		# Threads			
Atomics	**Compiler**	**1**	**4**	**16**	**64**
Linear Interpolation					
Yes	GNU	0.564	1.86	6.75	19.2
Yes	IBM	0.591	2.05	7.89	21.8
No	GNU	7.07	27.1	95.5	188
No	IBM	7.11	28.2	107	230
Exponential Intrinsic					
Yes	GNU	0.197	0.723	2.66	8.18
Yes	IBM	0.192	0.734	2.89	8.52
No	GNU	2.12	8.22	30.1	81.5
No	IBM	2.02	8.08	31.9	88.4

Table 4: Strong scaling integration rates $R(n_t)$ on IBM BG/Q (in millions).

observed on the Xeon. In particular, with 64 threads the IBM compiled-versions of the solver achieved $1.1 - 1.2\times$ greater integration rates than the version compiled with GNU. In addition, the IBM-compiled versions scaled more "smoothly" than the GNU-compiled versions, which followed some step changes in performance for certain thread counts.

Third, the use of hardware multi-threading enabled substantial performance enhancements for all sets of parameters and solvers on the BG/Q. Indeed, hardware multi-threading provided nearly $3\times$ strong scaling speedup for 64 threads with respect to 16 threads on 16 cores. Likewise, the weak scaling speedup ranged from 0.7 – 0.95× for 64 threads. This stands in contrast to the scaling observed on the Xeon, where Hyper-threading provided little benefit for most configurations (likely due to the inclusion of out-of-order execution on the Xeon but not the BG/Q).

A final note is that the linear interpolation method was 2-3× faster than the exponential intrinsic for evaluating exponentials. Furthermore, the difference in scalability between the two exponential evaluation methods was markedly different on the BG/Q. The linear interpolation method experienced a dramatic improvement in performance from 1 to 8 threads, achieving weak scaling parallel speedups of 1.3 – 1.5×. As the thread count increased, the weak scaling speedup for the linear interpolation method continued to outperform the exponential intrinsic, though its performance degrades more rapidly than the exponential intrinsic. In contrast, the transport sweep with the exponential intrinsic scaled slightly better than the one with linear interpolation for strong scaling.

5.3. Analysis

This section analyzes the parallel performance reults of the preceding section in an attempt to better understand their implications. In particular, several general trends are highlighted and some questions are identified which remain the subject of further research.

5.3.1. Mutual Exclusion

Mutual exclusion appeared to severely degrade the parallel performance for the transport sweep with atomics. The parallel scalability was very poor for the mutual exclusion version of the solver on the Xeon. However, on the BG/Q the integration rates were significantly reduced while the scalability did not appear to suffer.

Initially, these results were interpreted as an indication that it is important to avoid mutual exclusion – particularly in computationally intensive regions of the code – for maximal compute performance. However, the marked difference in behavior on the two hardware platforms, along with some probabilistic analysis cast some doubt on this conclusion. As explained in Section 6, the probability for any two threads to "collide" when attempting to write to an FSR's scalar flux array is exceedingly small. The apparent slowdown due to mutual exclusion was investigated with a performance model and addressed with only a slight modification to the implementation as described in Section 6.

5.3.2. Super-linear Weak Scaling

The performance results for both the Xeon and BG/Q exhibited better than ideal weak scaling for several configurations. On the Xeon, the Intel-compiled solver implementation without mutual exclusion with the linear interpolation method recorded 110% parallel efficiency with 12 threads on 12 cores. The weak scaling on BG/Q was super-linear to nearly 60 threads on 16 cores for the linear interpolation method without atomics. The IBM compiler achieved slightly super-linear scaling with the exponential intrinsic method up to roughly 40 threads on 16 cores.

The reason for super-linear weak scaling is not yet understood. One explanation is that concurrency enables more efficient caching on the chip. In particular, it may be possible cores can assist each other with pre-fetching commonly referenced cache lines in memory. In the C5G7 benchmark problem, the materials data (∼2 KB) and linear interpolation table (∼20 KB) are small enough to fit entirely in L1 cache. If more efficient caching is the reason for super-linear scaling, the performance should degrade for larger problems with more materials and/or energy groups. This is one of many questions which need to be addressed in the future.

5.3.3. Hardware Multi-Threading

The use of hardware multi-threading enabled significant gains in performance **for certain runtime configurations** on both the Xeon and BG/Q platforms. The speedups achieved with multiple threads per physical core were most substantial when using the (slower) exponential intrinsic method. The exponential intrinsic achieved 14× and 20× speedups for strong scaling with 24 threads on 12 Xeon cores with Intel and GNU compilers, respectively. This compares to 11× and 6× speedups for the linear interpolation method. With 64 threads on 16 cores on the BG/Q, the strong scaling speedups of 44× and 39× for the intrinsic compare to 32× and 26× for interpolation with the IBM and GNU compilers, respectively. In addition, the slope of the speedup curves with threads is nearly flat at 64 threads for the interpolation method, while it continues to steadily increase for the intrinsic.

In analyzing these trends, it is important to note that the exponential intrinsic is more flop-intensive than the linear interpolation table (21). In addition, the linear interpolation method is specifically designed for maximal cache efficiency since the table typically only consumes 20 KB which can easily fit in private cache on most multi-core machines. Both exponential evaluation methods must load the same data from main memory for each iteration of the transport sweep – namely, the length $l_{k,i} = s'' - s'$ of each characteristic segment, the total cross-section $\Sigma^T_{i,g}$ and source $Q_{i,g}$ of each FSR i. With multiple hardware threads, the increased flops necessary to evaluate the exponential may be used to "hide" the latency needed to load this common data from memory.

Although the increased flop intensity of the intrinsic improved scalability, it did not lead to greater overall performance. This is important to note since it illustrates that poorly performing code may exhibit the best parallel scalability. As many-core machines with tens to hundreds of cores per node are deployed, the scalability of more flop-intensive code – such as the exponential intrinsic – may enable it to outperform methods best suited for today's processor architectures.

5.3.4. Vendor-Specific Compilers

The Intel and IBM compilers each resulted in code which outperformed GNU's compiler on the Xeon and BG/Q platforms, respectively. The Intel compiler's superior performance was likely due to high level optimizations and well-suited for the Xeon and BG/Q architectures. Although the vendor-specific compilers resulted in code with consistently faster integration rates, the speedups achieved by GNU were comparable to the vendor-specific compilers.

As noted earlier, the Intel- and IBM-compiled code resulted in more smoothly varying speedup curves than the versions compiled by GNU. Although the overall speedups were largely similar across compilers, GNU experienced more irregular deviations from the general trend lines. It is reasonable to speculate that NUMA effects were more significant for the GNU compiler, which manifested themselves as step changes in performance for certain thread counts, but this hypothesis was not verified in the present analysis.

5.3.5. Clock Frequency Scaling

Finally, it should be noted that the peak integration rate on the BG/Q was less than that on the Xeon. Even with 64 threads, both linear interpolation and exponential intrinsic methods for the transport sweep without atomics were faster by $\sim 3\times$ on the Xeon. Most codes experience a slowdown on the BG/Q with respect to conventional platforms for a number of reasons. First and foremost, the clock speed is significantly slower to increase power efficiency. Second, the cores on the BG/Q do not support out-of-order execution which eliminates instruction pipelining. It is likely that each of these differences played a role in the observed performance on BG/Q.

Hardware vendors should note that while reducing the clock frequency and increasing concurrency (in this case, via hardware multi-threading) may improve power efficiency, it is often not beneficial for overall application performance. As a result, users may continue to deploy some codes on platforms with fewer parallel resources but higher clock frequencies to minimize the runtime of their applications for the forseeable future.

6. Mutual Exclusion Collisions

The preceding section revealed a contrast in the performance for the transport sweep algorithm with atomics on Intel Xeon and IBM BG/Q machines. In particular, the transport sweep with mutual exclusion observed poor scalability on the Xeon, while it achieved good scaling but poor overall performance on the BG/Q. To better understand this behavior, a predictive model was derived and compared to empirical measurements as presented in Section 6.1. This work showed that mutual exclusion was likely not the culprit behind these trends. Further investigation led to the discovery that the poor performance was the result of false sharing to maintain cache coherency. Section 6.2 describes a modified implementation of the transport sweep with atomics which enabled it to achieve the same scalability as that observed for the version with threadprivate arrays **as is demonstrated with performance data in Section 6.3.**

6.1. Performance Model

To investigate these trends, a predictive model for *mutual exclusion collisions* was derived and compared to empirical observations. For the purpose of this paper, a mutual exclusion collision is defined as an instance where two (or more) threads attempt to update the scalar flux for an FSR at the same time. In the case of the transport sweep with atomics, one thread was granted exclusivity to increment an FSR's scalar flux through the `omp_set_lock(...)` routine while other threads were idled until the first thread completed the update.

A predictive model for the number of collisions was deduced using simple combinatorics based on a couple of assumptions. First, it is assumed that characteristic segments are uniformly distributed across FSRs in the geometry. Second, the problem is assumed to be perfectly load balanced such that each thread performs the same number of segment integrations.

Third, it is assumed that threads perform segment integrations in lockstep fashion in perfect synchronization with every other thread. Finally, higher order collisions of three or more threads are neglected. Each of these assumptions except for the final one are highly conservative and should lead to an upper bound estimate on the number of collisions.

There are three important terms in the model equation for the number of collisions. First, the number of collisions should scale proportionally to the number of thread pairings p_t. The number of distinct pairs of threads p_t is given by Gauss' Trick:

$$p_t \quad = \quad (n_t - 1) + (n_t - 2) + ... + 1 \quad = \quad \frac{n_t(n_t - 1)}{2} \tag{9}$$

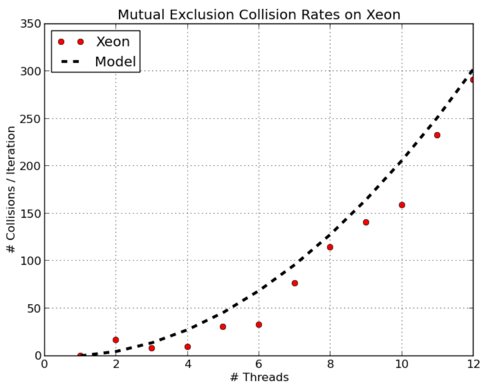
For a particular thread pairing, the probability that both threads attempt to update the scalar flux for the same FSR at the same time is given by the inverse of the number of FSRs, $^1/_{n_i}$. The number of opportunities for any two threads to collide scales linearly with the number of characteristic segments designated to each thread. For perfect load balancing this is simply $^{n_s}/_{n_t}$. The three factors combine multiplicatively to predict the total number of collisions during a transport sweep:

$$\#\text{ collisions} \quad \approx \quad \frac{n_t(n_t - 1)}{2}\frac{1}{n_i}\frac{n_s}{n_t} \tag{10}$$

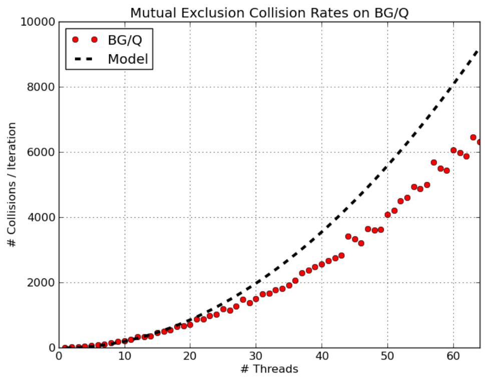
In order to verify this performance model, the number of mutual exclusion collisions was empirically measured. OpenMP permits one to query whether a mutual exclusion lock is taken with the `omp_test_lock(...)` routine. Two weak scaling studies were performed on the Intel Xeon and IBM BG/Q platforms and the number of collisions was measured using this routine.

On the Xeon, the number of threads was scaled from 1 – 12 while scaling the number of azimuthal angles from 4 – 48. On the BG/Q, the number of threads was scaled from 1 – 64 while scaling the number of azimuthal angles from 4 – 256. Ten source iterations were performed for each data point. The transport sweep with atomics was compiled with GNU for both platforms. The predicted and observed mutual exclusion collision rates for Xeon and BG/Q are presented in Figure 7.

As illustrated in the figures, the collision rate is surprisingly well predicted by the model, particularly when using fewer than or as many threads as the number of cores. This is likely due to the fact that each core can only handle a single thread context at a time. In addition, the assumption that each thread executes segment integrations in lockstep is almost certainly broken when using multiple hardware threads per core.



(a)



(b)

Figure 7: Predicted and observed collision rates on (a) Intel Xeon and (b) IBM BG/Q.

The empirical results show that for 12 threads on the Xeon, there are ∼300 collisions per transport sweep. Given that there are 7,863,448 characteristic segments in the problem with 48 azimuthal angles, mutual exclusion collisions occur for only ∼0.0038% of the segments. On the BG/Q, the rate increases to ∼0.015% of the 41,912,514 segments for 256 azimuthal angles with 64 threads. The collision rate was similar on both Xeon and BG/Q, yet the parallel scaling was strikingly different for both platforms. This indicated that mutual exclusion was not the root cause of the performance degradation. Further investigation led to a suitable explanation and solution to this issue as described in the following section.

6.2. False Sharing Resolution

The performance model for mutual exclusion collisions led to an effort to understand the poor performance observed for the transport sweep algorithm with atomics. Several permutations of the original implementation were explored before discovering that the poor scaling was not due to mutual exclusion but was instead the result of a subtle implementation issue which led to *false sharing*. False sharing occurs when a processor attempts to unnecessarily maintain cache coherency. False sharing may degrade some applications' parallel performance since the hardware is constantly synchronizing dirty cache lines across multiple cores' local cache(s).

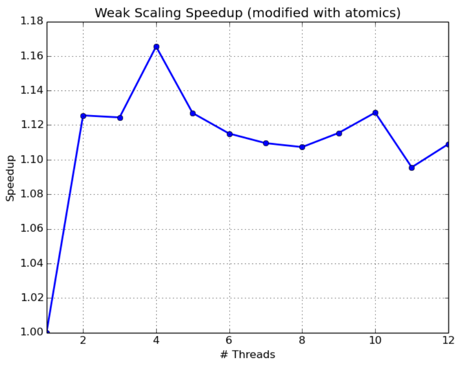
In the original version of algorithm, a contiguous array was allocated as a "scratchpad" for each thread to accumulate the contribution to an FSR's scalar flux from a particular segment during the transport sweep. This array was indexed by thread ID and energy group. The portion of the "scratchpad" array designated to each thread was not memory aligned. As more threads were introduced to the computation, those cache lines of the "scratchpad" array which were shared between threads became dirty, requiring the processor to maintain coherency across local caches.

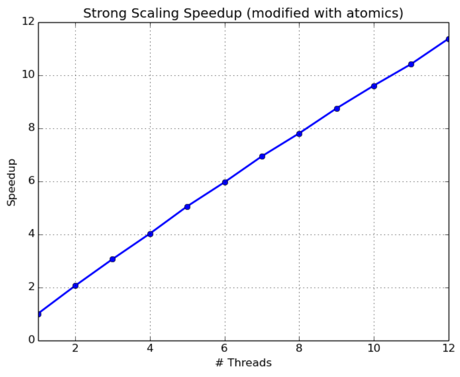
Upon further examination, it was discovered that by allowing each thread to allocate its own individual "scratchpad" array, the version of the transport sweep with atomics achieved the same performance and parallel scalability as the one without mutual exclusion. In this implementation, no cache lines were shared across cores/threads, relieving the processor of the need to maintain cache coherency. This new implementation mitigates the need for redundant storage of FSR scalar fluxes, and is the one currently available in OpenMOC's online source code.

6.3. Performance Results for the Modified Transport Sweep Algorithm

A series of weak and strong scaling studies were performed with the modified transport sweep algorithm to demonstrate good scalability with mutual exclusion. Figure 8 illustrates the weak and strong scaling speedups on Intel Xeon when compiled with Intel's compiler (note that Intel Hyper-threading was not used for technical reasons), while Figure 9 highlights the speedup on BG/Q when compiled with the IBM compiler. In both cases, the exponential linear interpolation table was used since it was the fastest runtime configuration on both platforms.

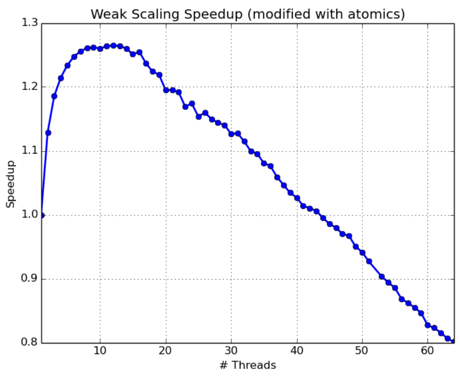
The modified transport sweep demonstrated nearly ideal strong scaling and super-linear weak scaling on the Xeon as was the case for the implementation without atomics in Section 5.2.1. The weak scaling performance on BG/Q dropped from a peak of 1.5$\times$ to 1.25$\times$, while the strong scaling speedup actually improved from 32$\times$ to 38$\times$. It should be noted that the integration rates for the modified transport sweep dropped by approximately 20% on the Xeon and 5% on the BG/Q with respect to those rates for the transport sweep without atomics presented in Section 5. However, this minor decline in performance is outweighed by the reduction in the memory footprint for the modified transport sweep over the version with threadprivate arrays (by a factor roughly equal

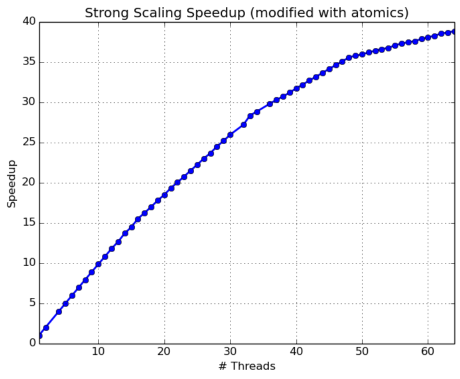




(a) (b)

Figure 8: Weak (a) and strong (b) scaling on Intel Xeon with the modified transport sweep algorithm with atomics modified to eliminate false sharing.





(a) (b)

Figure 9: Weak (a) and strong (b) scaling on IBM BG/Q with the modified transport sweep algorithm with atomics modified to eliminate false sharing.

to the number of threads). It is for this reason that the modified version of the parallel transport sweep shows promise as a good candidate algorithm for future many-core machines.

7. Conclusion

This paper presented the development and implementation of parallel algorithms for the OpenMOC neutron transport code on multi-core processors. The performance results reveal the dramatic potential for parallelization of MOC on multi-core platforms. OpenMOC achieved super-linear weak scaling and nearly perfect strong scaling with 12 threads on 12 Intel Xeon cores. The benefits of hardware multi-threading were demonstrated on the IBM Blue Gene/Q architecture, where OpenMOC achieved super-linear weak scaling up to 60 threads on 16 cores. These results reveal MOC as a promising candidate as a potential sucessor to diffusion-based methods for full-core reactor analysis on next-generation computing systems.

In addition, this paper identified some subtle challenges of multi-threaded programming for multi-core hardware platforms. Initially, it was believed that mutual exclusion led to poor parallel performance for one version of the transport sweep algorithm. However, a predictive model was developed and compared to empirical measurements to show that the performance degradation with threads could not be the result of mutual exclusion collisions. Further investigation found that the performance was sharply limited due to false sharing rather than mutual exclusion.

This work illustrated the harsh reality that small variations in code implementation can substantially affect within-node parallel performance. In the case of OpenMOC's transport sweep, only two lines of code were modified in order to gain significant speedups. A key take-home point from case studies such as this one is that developers should be cautious about making premature conclusions without fully characterizing their application's parallel performance.

The interpretation of the performance results presented in this paper may be improved with further analysis. First, analyzing performance counters (25) and/or other profiling mechanisms may be used to determine the reason for super-linear weak scaling. If our conjecture that multiple cores improves the cache efficiency of the MOC algorithm is accurate, future analyses may show that larger problem sizes with more materials and/or more energy groups has a significant impact on the parallel scaling.

Future work will aim to further improve the floating point throughput achieved on multi-core processors. Perhaps the most fruitful area for on-node hardware optimizations would be the formulation of a transport sweep algorithm which makes use of the vector floating point unit on modern multi-core processors. **In addition, a degree of distributed memory parallelism will be required for future MOC-based whole core methods. In particular, spatial domain decomposition will be needed to segment the memory footprint across multiple nodes as is presently done by the MPACT code (17). Future work should couple spatial domain decomposition for distributed memory parallelism with**

shared memory parallel transport sweeps executing on each node for efficient and scalable whole core transport.

Acknowledgements

Correspondence with Rodolfo Ferrer of Studsvik Scandpower, Inc., was helpful in achieving good parallel scaling for MOC with OpenMP. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This work was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374 and by the Center for Exascale Simulation of Advanced Reactors (CESAR), a co-design center under the U.S. Department of Energy's Contract No. DE-AC02-06CH11357.

References

[1] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 4.0, `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, [Online; accessed 12/31/2014] (2013).

[2] NVIDIA, NVIDIA CUDA C Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, [Online; accessed 12/31/2014] (2013).

[3] U.S. DOE, DOE Fundamentals Handbook: Nuclear Physics and Reactor Theory, Tech. rep., U.S. Dept. of Energy, Washington, D.C., USA (1993).

[4] E. E. Lewis, W. F. Miller, Computational Methods of Neutron Transport, Wiley, New York, 1984.

[5] P. K. Romano, B. Forget, Parallel Fission Bank Algorithms in Monte Carlo Criticality Calculations, Nuclear Science and Engineering 170 (2013) 125 – 135.

[6] N. Horelik, A. Siegel, B. Forget, K. Smith, Monte Carlo Domain Decomposition for Robust Nuclear Reactor Analysis, Parallel Computing 40 (10) (2014) 646–660.

[7] A. R. Siegel, K. Smith, P. K. Romano, B. Forget, K. G. Felker, Multi-Core Performance Studies of a Monte Carlo Neutron Transport Code, Int'l Journ. of High Perf. Computing Appl. 28 (1) (2014) 87–96.

[8] J. Tramm, A. R. Siegel, Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes, in: Joint Int'l Conf. on Supercomp. in Nucl. Appl. and Monte Carlo, Paris, France, 2013.

[9] A. G. Nelson, Monte Carlo Methods for Neutron Transport on Graphics Processing Units Using CUDA, M.S. Thesis, Pennsylvania State University (2009).

[10] C. Gong, J. Liu, L. Chi, H. Huang, J. Fang, Z. Gong, GPU Accelerated Simulations of 3D Deterministic Particle Transport Using Discrete Ordinate Method, Journ. of Comp. Phys. 230 (15) (2011) 6010–6022.

[11] C. Baker, G. Davidson, T. M. Evans, S. Hamilton, J. Jarrell, W. Joubert, High Performance Radiation Transport Simulations: Preparing for TITAN, in: Supercomputing 2012, Salt Lake City, Utah, USA, 2012.

[12] K. S. Smith, J. D. Rhodes, CASMO-4 Characteristics Methods for Two-Dimensional PWR and BWR Core Calculations, Trans. of the Amer. Nucl. Soc. 83 (2000) 294.

[13] K. S. Smith, J. D. Rhodes, Full-Core, 2-D, LWR Core Calculations with CASMO-4E, in: Proc. of PHYSOR, Seoul, South Korea, 2002.

[14] B. Kochunas, B. Collins, D. Jabaay, T. J., Downar, W. R. Martin, Overview of Development and Design of MPACT: Michigan Parallel Characteristics Transport Code, Tech. rep., American Nuclear Society, La Grange Park, IL, USA (2013).

[15] Y. S. Jung, H. G. Joo, Decoupled Planar MOC Solution for Dynamic Group Constant Generation in Direct Three-Dimensional Core Calculations, in: Proc. of the Int'l Conf. on Math. and Comp. Methods Applied to Nucl. Science and Eng., Saratoga Spring, NY, USA, 2009.

[16] X. Yang, N. Satvat, MOCUM: A Two-Dimensional Method of Characteristics Code Based on Constructive Solid Geometry and Unstructured Meshing for General Geometries, Annals of Nuclear Energy 46 (2012) 20–28.

[17] B. M. Kochunas, A Hybrid Parallel Algorithm for the 3-D Method of Characteristics Solution of the Boltzmann Transport Equation on High Performance Compute Clusters, Ph.D. Dissertation, University of Michigan (2013).

[18] W. Boyd, K. Smith, B. Forget, A Massively Parallel Method of Characteristic Neutral Particle Transport Code for GPUs, in: Proc. of the Int'l Conf. on Math. and Comp. Methods Applied to Nucl. Science and Eng., Sun Valley, ID, USA, 2013.

[19] W. Boyd, S. Shaner, L. Li, B. Forget, K. Smith, The OpenMOC Method of Characteristics Neutral Particle Transport Code, Annals of Nuclear Energy 68 (2014) 43–52.

[20] J. R. Askew, A Characteristics Formulation of the Neutron Transport Equation in Complicated Geometries, Tech. Rep. AAEW-M 1108, UK Atomic Energy Establishment (1972).

[21] A. Yamamoto, Y. Kitamura and Y. Yamane, Computational Efficiencies of Approximated Exponential Functions for Transport Calculations of the Characteristics Method, Annals of Nuclear Energy 2 (2004) 1027–1037.

[22] W. R. D. Boyd III, Massively Parallel Algorithms for Method of Characteristics Neutral Particle Transport on Shared Memory Computer Architectures, M.S. Thesis, Massachusetts Institute of Technology (2014).

[23] K. S. Smith, Nodal Method Storage Reduction by Non-linear Iteration, in: Trans. of the Amer. Nucl. Soc., Vol. 44, 1983.

[24] E. E. Lewis, G. Palmiotti, T. A. Taiwo, R. N. Blomquist, M. A. Smith and N. Tsoulfanidis, Benchmark Specifications for Deterministic MOX Fuel Assembly Transport Calculations without Spatial Homogenization, Tech. rep., Organisation for Economic Co-operation and Development's Nuclear Energy Agency (2003).

[25] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci, A Portable Programming Interface for Performance Evaluation on Modern Processors, Int'l Journ. of High Perf. Comp. Appl. 14 (3) (2000) 189–204.