Multi-core Performance Studies of a Monte Carlo Neutron Transport Code

A. R. Siegel[a], K. Smith[c], P. K. Romano[c], B. Forget[c], K. Felker[b]

[a] *Argonne National Laboratory, Theory and Computing Sciences and Nuclear Engineering Division*
[b] *Argonne National Laboratory, Theory and Computing Sciences*
[c] *Massachusetts Institute of Technology, Department of Nuclear Science and Engineering*

Abstract

Performance results are presented for a multi-threaded version of the OpenMC Monte Carlo neutronics code using OpenMP in the context of nuclear reactor criticality calculations. Our main interest is production computing, and thus we limit our approach to threading strategies that both require reasonable levels of development effort and preserve the code features necessary for robust application to real-world reactor problems. Several approaches are developed and the results compared on several multi-core platforms using a popular reactor physics benchmark. Our main focus is distilling a broad range of performance studies into a simple, consistent picture of the performance characteristics of reactor Monte Carlo algorithms on current multi-core architectures. Additionally, we speculate on the source of the observed scaling bottlenecks in terms of the exhaustion of shared hardware resources, and suggest programming approaches and strategies to help overcome them.

Keywords: OpenMC OpenMP reactor analysis multi-core shared memory Monte Carlo

1. Introduction

Monte Carlo (MC) neutral particle transport methods are critical for a broad range of scientific and engineering domains. Some important exam-

Email addresses: `siegela@mcs.anl.gov` (A. R. Siegel), `kord@mit.edu` (K. Smith), `romano7@mit.edu` (P. K. Romano), `bforget@mit.edu` (B. Forget), `felker@mcs.anl.gov` (K. Felker)

ples include the design, certification, and operation of nuclear reactors [1], nuclear fusion [2], radiation shielding, weapons design, medical dosimetry [3], and cloud radiation [4]. MC methods have a long history of successfully adapting to leadership class computing architectures, including excellent scalability on distributed memory platforms [5], innovative approaches for efficient execution on vector machines [6], and more recently proof of principle calculations for stripped down codes on GPGPUs [7].

For prototypical message-passing-based, distributed memory parallel machines built on scalar architectures, MC algorithms are typically formulated using the classical history method, where particles are followed one by one from birth to death. Since particles do not mutually interact and load balancing penalties are small, for many classes of applications this approach has shown excellent performance, with current benchmarks achieving near ideal scalability on up to one hundred thousand processing elements [5].[1]

Nonetheless, time to solution is still a critical bottleneck in applying Monte Carlo robustly to many real world problems, and thus for the foreseeable future research will need to focus on techniques and programming strategies to further reduce run time for a desired level of convergence. By the same token, multi-threaded methods will need to be developed and improved to achieve good performance even for smaller problems on commodity computing platforms. This is because both current and near-future desktop and supercomputing systems will increasingly require applications to expose greater levels of fine-grained parallelism to achieve good performance. Indeed, it is expected that in the near future hundreds of cores per node will be commonplace, even for commercial off-the-shelf technologies. Making use of multi- and many-core hardware will involve identifying new avenues for parallelism and scaling to far greater overall levels of concurrency than current practice.

However, even where algorithmic parallelism can be identified and exposed, it is far from guaranteed that speedups will approach ideal levels [9]. Multi-core memory hierarchies are far more complex and less scalable than typical distributed memory models. This often includes cache coherency software for distributed L1-cache, including significant penalties for false sharing on cache lines, shared higher levels of cache, and a shared

[1]Though not the focus of the current analysis, we do mention that in passing that this success depends to a large degree on the node-by-node replication of data structures that for reactor applications are too large to fit in node memory [8], thus significant work remains on efficient data decomposition strategies for realistic reactor benchmarks.

bus to main memory. Non-uniform memory architectures (NUMA) can even further complicate performance, especially when application-friendly programming models (e.g. OpenMP) have no mechanism to express data locality. Thus, even where a high degree of algorithmic parallelism can be formulated, it is often necessary to carefully construct data structures and manipulate data layouts so as to circumvent potential bottlenecks and maximize the likelihood of achieving good performance in practice (see e.g. [9]).

While some of the most popular community MC codes (e.g. [10, 11, 12]) have experimented with on-node threading capability, and some anecdotal knowledge on performance is shared spontaneously within the community, we are not aware of any published work that attempts to systematically elucidate the key issues and test the performance of MC methods on multi-core architectures. In this work we carry out code modifications and an associated set of numerical experiments designed to take a first step in this direction.

Several approaches can be taken to carrying out such a study, on the one extreme using more sophisticated coding strategies and deeper analyses on highly stripped down "kernel" MC applications, and on the other hand migrating a full-featured production code in the context of real-world benchmark calculations. Each approach has its merits and will contribute in part to the complex overall picture of multi-core performance of MC methods. In the present work we choose to follow the latter approach, adopting the production OpenMC [13] code together with the OpenMP library to thread the critical areas of the application and test on a modified version of the popular Hoogenboom-Martin [8] reactor benchmark. We emphasize that the analysis done here is thus of greatest relevance to nuclear reactor analysis – specifically targeting classical calculations for the design and optimization of reactor cores. While many of the conclusions are relevant to a broader class of problems, reactor core analysis has unique requirements that result in performance profiles in some ways distinct from other application domains. Details are described in the following section.

As a programming model the directive-based OpenMP threading framework has limited semantics for parallelism and its performance can be highly sensitive to compiler implementations, but it allows easy incremental parallelism that greatly simplifies the migration of large production codes. Furthermore, we argue that the parallelism expressed is extremely simple and should be easily analyzable by any reasonably efficient OpenMP compiler, yielding code not too different from what is possible with high level, intrusive threading libraries.

2. Monte Carlo algorithm

2.1. Algorithms

At a high level and ignoring details of the complex treatment of physics and geometry, the MC transport algorithm can be described very simply. Let $P = \mathbb{R}^3 \times \mathbb{R}^+ \times \mathbb{S}^2 \times \mathbb{Z}$ denote the set of particles (neutrons) uniquely defined by a physical-space position $x \in \mathbb{R}^3$, energy $E \in \mathbb{R}^+$, direction $\Omega \in \mathbb{S}^2$ (where $\mathbb{S}^n$ denotes the n-sphere), and particle id $I \in \mathbb{Z}^+$. Furthermore, let $B \subseteq P$ denote a countable subset of particles referred to as a *neutron batch* of size $|B| = n$. The steady-state fission source algorithm then iterates over batches of particles (*batch loop*) and tracks them individually (*particle loop*) through a sequence of collisions from birth to death (absorption). Some absorption events will result in nuclear fission and the subsequent release of additional particles, which then populate the new batch at the next stage of the algorithm. Our primary focus in this analysis is steady state calculations, in which case the number of particles is re-scaled so that no particles are created or destroyed at each iteration of the batch loop. This is the common strategy for handling eigenvalue problems in reactor analysis, where the ratio of particles between batch iterations gives an estimate of the growth rate (eigenvalue), and the problem is solved for the steady state solution (the true eigenvalue is scaled to unity). This process is continued until a reasonable convergence criterion is met (either on the eigenvalue or spatial distribution, a detail which is not important for the present analysis). A simple pseudo-code description of this algorithm is given in Algorithm 1 below where ν represents the number of new particles generated after a given fission event, and the *rescale* operation denotes the resampling of particles so that none are created or destroyed between batch iterations (to simulate steady state behavior).

As shown in Algorithm 1, arguably the most natural approach to threading is to divide particle histories among threads – that is, each thread is responsible for carrying out the tracking of a subset of particles in a batch. We refer to this strategy in the present context as *coarse-grained* threading. Coarse-grained threading mimics the typical strategy for carrying out distributed memory parallelism, where the particles in a batch are distributed evenly among MPI processes (nodes) and key data structures, such as geometry, tallies, material and cross section data, are replicated across nodes.

Algorithm 1 Coarse-grained threading approach

```
initialize $B$
while *not converged* do
    *#pragma omp parallel for*
    for $p \in B$ do
        repeat
            $move(p)$
        until $absorbed(p)$
        if fissioned(p) then
            *create* $\nu$ *new particles* $\{p_1, p_2, \cdots p_\nu\}$
            $B_{next} \leftarrow B_{next} \cup \{p_1, p_2, \cdots p_\nu\}$
        end if
    end for
    $B \leftarrow rescale(B_{next})$
    test $B$, $B_{next}$ for convergence
end while
```

On a hybrid shared/distributed memory system (i.e. a cluster of multi- or many-core nodes) this approach would simply subdivide the particles by node in the regular manner, and then within each node further subdivide by thread. In the former case the key data structures either need to be replicated, the typical approach, or decomposed and accessed via explicit message passing. The latter approach is not typical due to lack of locality in access pattern. In the shared memory case decomposition is not necessary, but various forms of contention in the shared memory hierarchy may potentially erode scalability.

In Algorithm 1 the *move(p)* method advances a particle probabilistically through a series of collisions until absorption and possible fission. In a reactor core this includes potentially millions of material regions (e.g. when doing depletion analysis) with hundreds of nuclides. Let $J \in \mathbb{Z}^+$ denote the set of all nuclides and $M \in \mathbb{Z}^+$ denote the set of all material *regions* in the reactor core (identified by some integer tag). Let the *atomic density function* $f : M \times J \to \mathbb{R}$ denote the atomic density of a given nuclide in a given material region, and let $g : \mathbb{R}^3 \to \mathbb{Z}^+$ denote the material lookup function – i.e. g selects the material region associated with a given particle position $x_p \in R^3$. Finally, define a *microscopic cross section table* for nuclide $j \in J$ as an element of $(\mathbb{R}^+)^{NE(j)}$, where $NE(j)$ denotes the number of tabulated cross section energy levels for nuclide j. Then, Algorithm 2 represents the calculation of the macroscopic cross section, $X(E)$, used to advance the particle in the *move()* routine.

Algorithm 2 Fine-grained threading approach

```
$m \leftarrow g(x_p)$
$\#pragma\ omp\ parallel\ for$
for $j \in J$ do
  $X(E) \leftarrow X(E) \cup f(m, j)x(E)$
end for
```

In Algorithm 2, the nuclide loop at each stage in the tracking of a particle actually involves not one but multiple reaction types (depending on specifics of the application). For typical reactor applications with hundreds of nuclides and several reaction types, we find that this nuclide loop typically consumes $80 - 85\%$ of the total simulation execution time. Thus, one alternative approach, what we refer to as *fine-grained* threading in the current context, involves threading the nuclide search as shown in Algorithm 2. Since the maximum number of nuclides in a region is typically several hundred, this strategy is ultimately limited to relatively modest core counts. When considering many-core architectures, however, we may choose to implement a hybrid on-node approach that combines both the coarse and fine-grained strategies. Thus we still consider this a worthwhile approach to pursue for both the near as well as e.g. exascale computing platforms.

3. Approach to multi-threading OpenMC

OpenMC is an open source Monte Carlo neutron transport code recently developed at MIT and capable of performing calculations on arbitrary 3D geometries with continuous-energy cross-sections. It was written with a focus on scalable algorithms for leadership-class supercomputers and has demonstrated weak scaling up to hundreds of thousands of processors on ALCF's Intrepid and OLCF's Jaguar supercomputers [5]. The codebase is written in Fortran 2008 with parallelism provided via MPI. For the purposes of this study, the OpenMC code was modified using OpenMP directives to implement the coarse-grained, fine-grained, and hybrid coarse-fine on-node threading strategies described in the previous section. Since OpenMC is a relatively mature code with a relatively high degree of complexity aimed at doing real reactor benchmark problems (e.g. the implementations of physics interactions, geometry, and tally filters), we chose to use a directive-based threading approach as a first step to minimize code modifications. An overview of the key code changes to implement coarse grained, fine grained, and hybrid threading is given below. When completed the modified version of OpenMC was run through a comprehensive test suite to verify correctness.

3.1. coarse-grained threading

The key aspects of coarse-grained threading include: 1) threading the main particle loop using the *omp parallel for* construct – note that the schedule setting (*dynamic*, *static*, or *guided*) should have some effect on load imbalances among threads, a topic which is discussed in the following section; 2) marking key global mutable data structures as *threadprivate* – specifically the microscopic cross section cache, the macroscopic cross section cache[2], and the fission bank, and 3) marking all tally increments as *atomic* operations during the tracking of a particle. The most common operation, cross-section data table lookups, is a read-only operation and thus inherently thread safe. The cross section arrays are therefore kept in shared memory. In addition, there were a moderate number of code changes required to overcome shortcomings in the interaction of OpenMP with advanced Fortran constructs, particularly Fortran pointers.

Regarding the threadprivate variables, the microscopic and macroscopic cross section arrays, which store respectively the per nuclide and total cross section value for a given collision, are updated per particle per interaction and occupy very small amounts of memory; choosing to make them threadprivate is a straightforward decision as it eliminates the possibility of cache line conflicts (real or false) at negligible additional storage cost. The fission bank is more subtle. It is updated continuously during the tracking of a particle and records all necessary information each time a fission event occurs. Since it is updated sequentially each time a thread samples a fission event, keeping it in global memory requires synchronizing access. However, as the scheduling of threads in non-deterministic, such an approach will in general yield different orderings for different executions (even with identical random number seed). In order to maintain strict (bitwise) reproducibility of results (a common requirement with reactor licensing authorities), it was

[2]Here, *cache* refers to a temporary copy associated with a single interaction of an individual particle. This is a convenient coding strategy since cross section values need to be interpolated from large lookup tables on a per interaction basis

necessary instead to implement threadprivate versions of the fission bank and explicitly synchronize each thread's local bank into a global fission bank at the end of each batch. This obviously increases the memory footprint but was observed to have little impact on performance.

The specific choice of tally events and filters depends to a large extent on the particular calculation. For depletion analysis, which is one of the most critical applications to the reactor designer, we have estimated elsewhere an aggregate 1 TB is necessary for robust reactor analyses [14]. For other reactor (and non-reactor) applications the requirements may be much more modest. In all cases, though, the tallies require simply incrementing counters for the range of events of interest. One then has the choice of creating local counters and aggregating at the end of each batch, or keeping global counters and synchronizing with *atomic* annotations of the counter increments. After experimenting with both and seeing negligible impact on performance (tally increments are a tiny fraction of overall performance time), we have adopted the latter approach for simplicity of code structure.

3.2. fine-grained threading

The fine grained threading approach as described in Algorithm 2 is implemented in a straightforward manner by using a *parallel for* construct with a reduction operation on the nuclide loop. Since this loop is called with extremely high frequency (once per collision per particle), one early observation is that the overhead in creating the parallel region and carrying out the reduction nullifies any performance gain when the particle undergoes an interaction in the non-fuel regions of the reactor (i.e. which contain relatively few isotopes and thus the number of loop iterations is small). Thus, using the OpenMP *if* clause, the threaded region was limited to cases where the interaction took place within the fuel. We point out this still occupies a significant fraction of the total computational time for a broad class of applications. Further details are discussed below.

3.3. hybrid threading

Hybrid threading was considerably more challenging to implement within the OpenMP framework. While OpenMP version 3 contains support for nested threaded regions, the semantics are extremely limited and make it awkward to express the required relationships between the variables, particularly with advanced Fortran constructs.

The key for OpenMC threading was creating threadprivate variables at nesting level 1 (coarse grained threading across particles) that behaved as global variables at nesting level 2 – variables that were private to each particle but global for all nuclides at each particle interaction. When global variables are used and thus marked as threadprivate, they are considered threadprivate at all nesting levels and cannot be marked with a *shared* construct in the nested region. To overcome this shortcoming required non-trivial internal code changes.

One further issue involves the lack of flexibility within the nested threaded region. Ideally one would like to allow threads to be assigned to the region dynamically to accelerate particle tracking when particles were interacting with fuel regions and, rather than remain idle, carry out particle tracking when they were otherwise free. Such a dynamic threadpool model was not possible to express in OpenMP and thus limited the possible available performance benefit of this approach.

4. Numerical experiments

4.1. Overview

A broad set of numerical experiments were conducted. Of these, we report on a small subset that aim to give a consistent picture of the key scaling issues. Since our interest is production computing for reactor applications, we do not focus on the details of architecture-specific optimizations. While a number of tuning strategies were explored on particular platforms, for the purposes of the present analysis we take a general view of modern multi-core architectures and aim to identify potential scalability and the source of any common bottlenecks that might erode performance.

All of our numerical experiments involve the Hoogenboom-Martin (H-M) [8] reactor criticality benchmark. We run H-M in two different configurations generally representative of early and late phases of a depletion cycle –what we refer to as *small H-M*, with 60 nuclides in the fuel region, and *large H-M*, with 360 nuclides in the fuel region. For all experiments the relevant unit of measure is the tracking rate, expressed in number of particles tracked per unit computational time. For each experiment we use 10 batches with $50,000$ total particles per batch. These figures were selected by trial and error to ensure that the batch sizes are large and that the results are not influenced by initialization time – adding additional particles or batches does not change the computation rate or any of the other conclusions of this analysis. While we ran a large range of tally configurations, the multi-core scaling impact of additional tallies were negligible. Thus, with no effect on our main conclusions the results reported here use so called *inactive batches*,

Computer name	Challenger	Breadboard	Knight	Chimera
Institution	ANL	ANL	ANL	UDel
Processor	IBM PowerPC 450	Intel Xeon X5550	Intel Xeon X5680	AMD Opteron 6164HE
Clock speed	850MHz	2.66GHz	3.3 GHz	1.7GHz
Cores/CPU	4	4	6	12
CPUs/node	1	2	4	4
Cores/node	4	8	24	48
Memory/node	2GB	16GB	20GB	64GB
L1 Cache	32KB private	256KB private	32KB private	128KB private
L2 Cache	2KB private	1MB private	256KB private	512KB private
L3 Cache	8MB shared	8MB shared	12MB shared	12 MB shared

Table 1: Summary of computing platforms used in the study

where minimal tally information is computed and the goal is to converge the source distribution. Again, the relatively low cost of tallies is a consequence of the dominance of the macroscopic cross section loop, a characteristic of steady state reactor physics calculations. For other classes of applications tally rates may represent a non-trivial fraction of overall performance, and the conclusions drawn may differ slightly.

4.2. Platforms

We tested the benchmarks on four platforms: the University of Delaware's Chimera cluster, Argonne National Laboratory's Blue Gene/P supercomputer, ANL's Knight cluster, and ANL's heterogeneous platform Breadboard. In each case, OpenMC was deployed on a single node and used a variable number of cores with one OpenMP thread per core. Each node of the Chimera cluster consists of 4 AMD Opteron 12-core processors which share 64GB of RAM (4 GB DIMMS). A single compute card of the Blue Gene/P Challenger system contains 4 PowerPC 850MHz cores and 2GB of memory. The login node of Knight, on which the performance tests were completed, is supported by 4 Intel 6-core Xeon X5680 processors. For the Breadboard cluster, a node consists of 2 Intel Xeon 4-core 2.66GHz processors which share 16 GB of RAM. The technical specifications of each platform are summarized in Table 1.

4.3. Preliminary tests

Before studying scalability and relative execution times, we carried out a preliminary set of studies aimed at baselining our performance expectations.

The main goal was to *a priori* identify any scalability bottlenecks so that we could have a basis for determining what constitutes "good" performance. In the following section each potential scalability bottleneck is identified and discussed in the context of these preliminary results.

1. *Amdahl's Law*
   A strong scaling upper bound is set by the fraction of time spent in the threaded region. The well known Amdahl's law points out that an algorithm is limited to a speedup proportional to $\frac{1}{1-P}$, where P is the percentage of time in the execution of the parallel portion of the algorithm. Traditional supercomputers circumvent this problem using the memory added with each processing element to increase the problem size in proportion to the degree of parallelization[15], but for the multi-core shared memory nodes in this study aggregate on-node memory does not increase with the number of threads. Thus, we must evaluate the fraction of time spent in the coarse-grained loop described in Section 1.

   Over a range of simulation and parameter values, we find that the particle tracking loop accounts for between $98 - 99\%$ of the total execution time. Thus, at least for the thread counts typical on modern multi-core architectures, coarse grained threading performance should not be limited by Amdahl's law. We do note however that in the near-future many-core platforms are expected to change this scenario and require further parallel treatment of the outer-loop region.

   For fine-grained threading the situation is less ideal. We find that for the large H-M about $80 - 85\%$ of the total execution time is spent in the threaded region, but only about $50 - 60\%$ for the small H-M benchmark. Thus, we expect diminishing returns beyond a relatively small number of threads. We nonetheless study this approach given the extreme simplicity of implementing it, its potential for quick payoff on small core counts, and its potential usefulness in a hybrid approach.

2. *Thread overhead*
   In a SIMD model the overhead cost of entering and exiting parallel regions can potentially compete with the speedup gained from parallelization. We tested this in depth for the coarse-grained approach and found that, even for much smaller benchmark problems than the cur-

rent ones, the price of thread creation is completely negligible. This is not a surprise given that we leave/enter the coarse region only after 50,000 particles are tracked, which at minimum takes several seconds of computation time. Thread overheads are typically reported in tens of nanoseconds and are thus expected to have no detrimental impact on performance in the present case.

For the fine-grained approach, the threaded region is entered once per particle per interaction (or change in material region). On average this amounts to approximately forty times the frequency of the coarse-grained approach but is still found to be less than one percent penalty for the thread counts of interest. Thus, we discount thread overhead as a potential obstacle to achieving multi-core scalability.

3. *Load balancing*
   Uneven work distributions among threads in the absence of load rebalancing is another potential obstacle to achieving good on-node scaling. Intuitively, given an initial equal distribution of thousands of particles per thread, we might expect statistical fluctuations to be smoothed out resulting in roughly equal total tracking time per thread. This, in fact, is the major advantage of particle-based over physical-space domain decomposition approaches[16], where load imbalances can significantly erode performance on fine spatial grids.

   We tested this hypothesis for both the coarse and fine-grained approaches. In the former case, we found that the maximum load imbalance for all of the tested configurations were $5 - 10\%$ of the total tracking time. We were able to remove this penalty almost entirely by using *schedule(dynamic)* construct in OpenMP, with an empirically determined optimal value of 5. All of the tests reported here are based on this form of dynamic scheduling.

   For fine-grained threading, it is not surprising that load imbalance penalties were observed in all cases to be less than two percent. This follows from the fact that identical operations are being performed for each sub-batch of nuclides, the only imbalance occurring when the nuclide count is not divisible by the number of threads.

4. *Synchronized code in threaded region*
   Atomic operations in the coarse-grain threaded region are required to increment tally counters for a broad range of events. These increment operations overall represent only a very small fraction ($< 1\%$) of the total execution time, but their impact on overall performance still needs to be measured directly. To do so we directly compared timing results both without synchronization and with all tally operations removed. Doing so allowed us to verify that the total performance impact at worse was in the range of $1 - 2\%$. Thus we discount this as a major source of performance loss. In the fine-grained case, all write operations are to thread local variables and thus synchronization is not required.

5. *Scalability of memory subsystem*
   Though aspects of our testbed multi-core architectures vary significantly in their details, they are all characterized by bottlenecks in their memory subsystems that are not present on typical distributed memory platforms. Details are discussed in the following section, but our main areas of concern are threefold: all systems have distributed L1 caches, thus both real and false sharing of cache lines can potentially cause significant bottlenecks to scalability; since a significant amount of time is spent in random data lookup of large cross-section tables, contention in higher-level caches, which are shared at some level on all of our test architectures, becomes a potential scalability bottleneck; finally, the main memory bus is also shared (in different ways), and thus we must explore the possibility of exhausting bandwidth as we increase the number of threads. None of these issues is simple to diagnose robustly, but we cannot rule them out as possible sources of performance degradation compared to, e.g. distributed memory applications, whose extreme scalability has been demonstrated on a range of applications [17]. This is discussed further in the next section.

4.4. observed timings

4.4.1. coarse-grained threading

Using the above preliminary analysis as a basis of interpretation, we measure the performance of the two benchmark problems on our target architectures. We first present a basic birds-eye overview of results on the target platforms before focusing on the Xeon platform in more depth in the following section. The results presented are pared down from a very broad range numerical experiments covering a wide range of parameters, including different compilers and compiler versions, different compiler optimization options, different cross section lookup techniques (creating a unionized energy grid vs. binary searches for each interaction), and using a range of techniques to ensure minimal chance of real and false sharing of L1 cache lines. The presented results are not necessarily chosen to portray best-case scalability numbers but rather are typical across our wide range of tests. Indeed, while performance does fluctuate across machines, compilers, and other test parameters, roughly similar results are surprisingly persistent (and consistent with qualitative comparisons in the community). Departures from these "typical" results and a deeper discussion of underlying causes are discussed in the following section.

Figures 1 and 2 each show the tracking rates per thread respectively for the large and small H-M benchmark. The tracking rate measures number of particles tracked per unit processor time and is the most natural application-level measure of performance for MC codes. When the tracking rates are presented per thread count as shown, a horizontal line indicates ideal scaling, and deviations from ideal scaling are thus more readily visible compared to other approaches.

For the small H-M benchmark in Figure 1, single core tracking rates ranged from approximately 250 on a Blue Gene core to almost 4500 particles/sec on the a single Intel Xeon X5680 core. This discrepancy is due in part to the factor of four disparity in clock speeds, but we emphasize that our main focus in this analysis is multi-core scalability, and thus we did not analyze the additional source of absolute single-core performance degradation (though we did verify this tracking rate on a wide range of parameter optimization levels and code optimizations). While the PowerPC performance was poor in an absolute sense, performance on all four cores on a BG/P node achieved 97% of ideal scaling, indicating negligible impact of the shared aspects of the memory subsystem on the performance of each core. A general trend that was observed across all of our studies is an inverse relationship between single core performance and scalability. For example, turning off optimization for the Intel compiler yielded much poorer wall clock times but scalability of 85-90% across all available cores,while the optimized results presented show significant deviations from ideal scaling even for several cores. We currently have no definitive explanation for this behavior, though some of the key issues are addressed in the following section.

Figure 1 has several other noteworthy characteristics. The 8-core Xeon X5550 node achieves approximately 80% of ideal scaling when all 8 cores are used, but the performance of the 24-core Xeon X5680 node erodes more rapidly with core count, with approximately 61% efficiency on 8 cores and only 33% on all 24 cores. The Opteron 48-core node shows more complex behavior but surprisingly good performance of 65% scalability using the 48 cores. Large H-M (Figure 2) shows qualitatively identical behavior with perhaps one exception – the 24-core Intel scales non-trivially better – achieving close to 50% efficiency on the full machine.

4.5. hybrid threading

We present our sample timings for the fine-grained/hybrid threading cases on the 8-core Xeon node. Hybrid tests were run on the full range of platforms, but little extra insight is gained beyond what is evident from the Xeon results, thus we limit our discussion to only that platform. Also, the large H-M benchmark is the most natural candidate for speedup with hybrid threading. Thus, we limit our analysis only to this benchmark problem.

The raw timing results are shown in Table 2, which gives results for all possible combinations of coarse/fine threads (i.e. those whose product is less than eight) . Thus, the upper right hand entry is identically the fine-grained case, and the bottom left entry is the coarse-grained timing. Our main interest is to ascertain whether fine grained threading, or any combination of coarse-fine threading, can produce better results than the same number of threads dedicated entirely to the coarse-grained approach. Our results indicate that this is not the case – though the hybrid results are competitive and generally lie within 20% of the corresponding coarse grained values, in all cases the greatest efficiency is achieved by dedicating all threads to the coarse-grained loop. The fine-grained limit was the worse performing, with the coarse-grained threads exhibiting a tracking speed twice that of the fine-grained threads for the full eight threads.

The hybrid threading results demonstrate the subtleties that Monte Carlo developers must contend with when programming shared memory models. The level of granularity at which threads operate can have a significant impact on performance returns. While this study shows that thread resources are best devoted to the coarsest level of parallelism to avoid these performance limitations in this instance, the hybrid threading model gives us two insights. First, it clearly shows the evolution of the performance of the code as threads move higher in the looping constructs. Second, it proves that nested threads can still provide speedup, if other application considerations force a division of thread allocation.

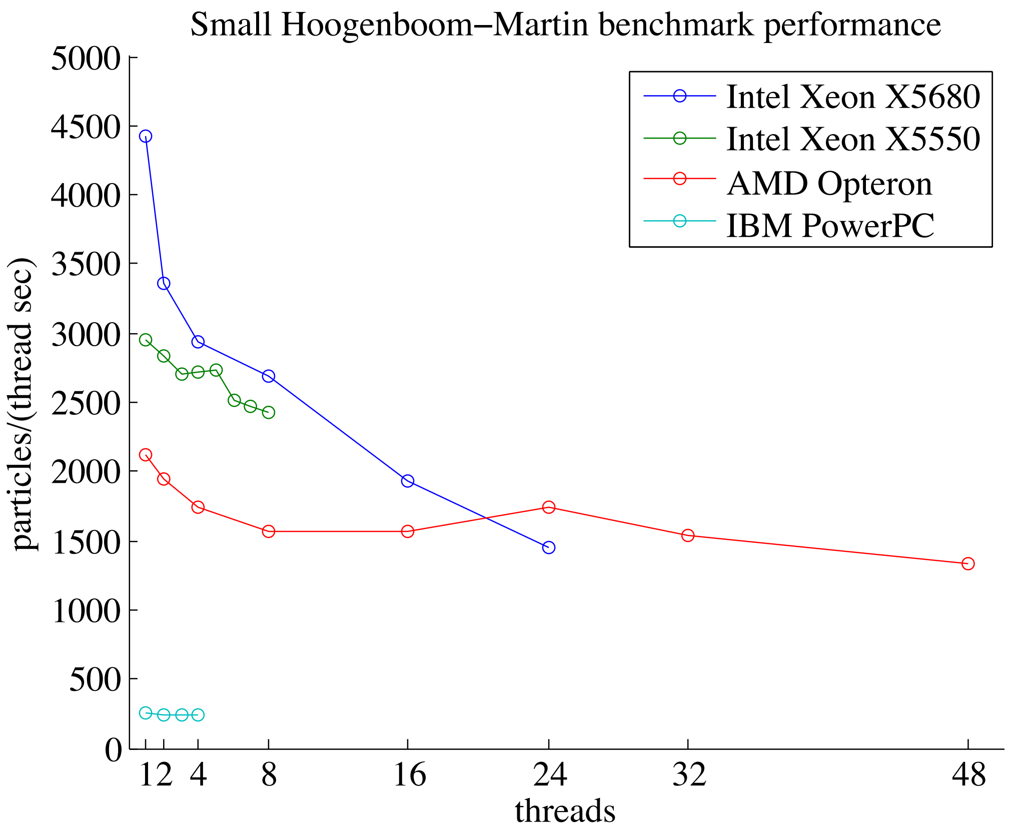

Figure 1: Coarse grained threading performance of OpenMC on small H-M benchmark.

4.6. possible sources of performance degradation

The results presented in the previous section include diverse multi-core architectures with a broad range of maximum thread counts, core interconnect technologies, and cache characteristics. The approach here was to take an abstract view of each node as providing a collection of independent cores capable of independently carrying out the instructions required for particle tracking. As was discussed in the previous section in some depth, the particle tracking algorithm itself is nearly perfectly scalable algorithmically – any significant departures from ideal scalability must come from

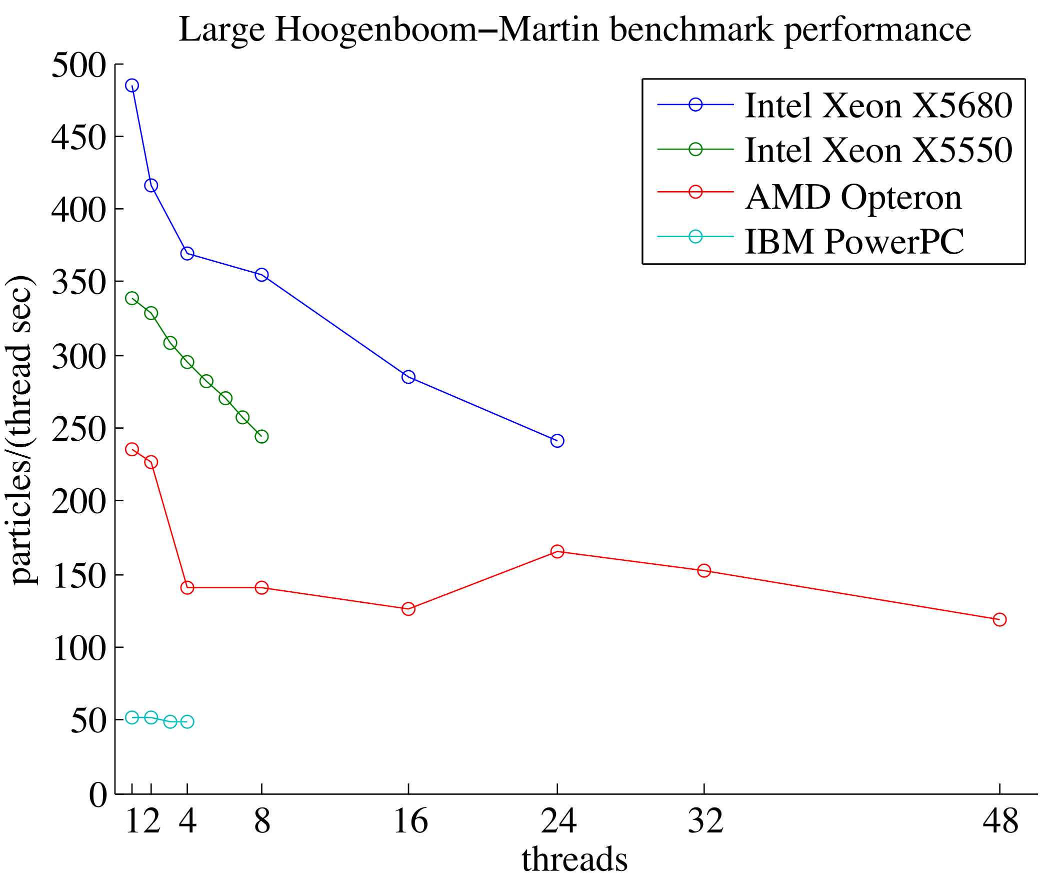

Figure 2: Coarse grained threading performance of OpenMC on large H-M benchmark.

the inability of the memory sub-system to deliver the data to the cores in a scalable manner. We have viewed OpenMP as an adequate programming model for this very high level of abstraction, where it is required merely to describe how the algorithmic work is distributed across cores, and there is no attempt to control the flow of data to the cores.

The results in the previous section are mixed and depend to some extent on perspective – on the one hand, they indicate very good on-node speedup for either modest levels of work in real-world application codes – speedups of thirty times on 48-cores, for example, were observed on the AMD platform. On the other hand, it is clear the scalability is limited compared to what is feasible in principle (e.g. on a machine where bandwidth increases

$nt_c \backslash nt_f$	1	2	3	4	5	6	7	8
1	371	561	812	1060	1193	1210	1218	1262
2	672	1118	1496	1808	.	.	.	.
3	956	1597	.	.	.	.	.	.
4	1262	2122	.	.	.	.	.	.
5	1568	.	.	.	.	.	.	.
6	1858	.	.	.	.	.	.	.
7	2163	.	.	.	.	.	.	.
8	2275	.	.	.	.	.	.	.

Table 2: Performance comparison of fine (nuclide-loop) and coarse (particle-loop) threading in OpenMC. Entries are in particles/sec. As the column number increases, more threads are devoted to the inner loop over nuclides. As the row number increases, more threads are used to divide the work of the outer loop over particles. Using exactly 1 OMP thread/core, the only possible combinations of threads are those that satisfy [(# coarse threads)$\times$ (# fine threads) ≤ 8].

proportionally to processing elements), and furthermore that the trend is to continue to erode as cores are added. This is not surprising given the complexity of the shared resources in the underlying memory subsystem. However, it is instructive to carry out some preliminary deeper analyses to try to speculate on exactly where the performance is lost. Such an analysis is not easy [18, 19] and we present here only the key issues as a basis for further study.

Scaling bottlenecks for multi-core applications generally include: 1) real and false sharing on L1 cache lines (as L1 caches are distributed for all of the target architectures); 2) contention for higher levels of shared cache, 3) contention on the shared bus to main memory, and 4) NUMA effects for multi-socket architectures. As each of these aspects of the memory subsystem is opaque to OpenMP, additional analysis tools are required to gain information on which effects may be leading to performance degradations, and furthermore what algorithmic or implementation modifications may be made to mitigate their impact.

To this end, as part of this process we carefully instrumented Performance Application Programming Interface (PAPI) [20] hardware counters on the Intel Xeon platforms in OpenMC. The performance data enabled us to make careful code modifications to guarantee the negligible impact of L1 cache write conflicts as we increased core counts, resulting in noticeable performance improvements for the tests presented here. Declaring the macroscopic cross section variable as *threadprivate* and ensuring their allocation in sufficiently disjoint memory locations (to avoid false sharing conflicts) is one such example. However, further attempts to disambiguate between the remaining potential causes of slowdown of the algorithm have not yet produced definitive results. The conclusions of our analysis have varied non-trivially with choice analysis tool, compiler technology, and computing architecture. It is also likely that a lower level programming model will be required to implement more scalable algorithms once the performance culprits have been identified. A follow-up study using a simplified MC application kernel will explore the topic in depth.

5. Conclusion

We modified the Monte Carlo neutron transport code OpenMC to use OpenMP-enabled shared-memory parallelization within each MPI process in three different configurations. The scaling performance of the configurations was compared with the application-relevant reactor calculation parameters of the Hoogenboom-Martin benchmark.

Our results accomplish two primary objectives. First, they show the practical benefits available to Monte Carlo methods as the field of high performance computing moves to many-core architectures. Significant speedup of the neutron tracking rate is easily achieved with OpenMP on 4 core to 48 core modern processing nodes. Second, the degradation of scaling at higher core counts elucidates the complex limitations imposed by the multitude of hardware and software considerations which are imposed by the many-core model. This study demonstrates that a variety of performance factors unique to shared-memory programming, including NUMA memory hierarchies, cache bottlenecks, and thread overhead need to be considered by Monte Carlo developers. More precise tools are needed to diagnose the exact influence of these factors.

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 Chimera computer used in this research was funded by NSF Award CNS-0958512.

References

[1] Yousry Azmy, Enrico Sartori, and Jerome Spanier. Monte Carlo methods. In *Nuclear Computational Science*, pages 117–165. Springer Netherlands, 2010.

[2] D. Heifetz, D. Post, M. Petravic, J. Weisheit, and G. Bateman. A Monte-Carlo model of neutral-particle transport in diverted plasmas. *J. Comput. Phys.*, 46(2):309 – 327, 1982.

[3] D. W. O. Rogers. Fifty years of Monte Carlo simulations for medical physics. *Phys. Med. Biol.*, 51(13):R287, 2006.

[4] William O'Hirok and Catherine Gautier. A three-dimensional radiative transfer model to investigate the solar radiation within a cloudy atmosphere, Part I: spatial effects. *J. Atmos. Sci.*, 55(12):2162–2179, 1998.

[5] Paul K. Romano, Benoit Forget, and Forrest Brown. Towards scalable parallelism in monte carlo particle transport codes using remote memory access. *Prog. Nucl. Sci. Technol.*, 2:670–675, 2011.

[6] Forrest B. Brown and William R. Martin. Monte Carlo methods for radiation transport analysis on vector computers. *Progress in Nuclear Energy*, 14(3):269–299, 1984.

[7] Francois A. van Heerden. A coarse grained particle transport solver designed specifically for graphics processing units. *Transport Theory and Statistical Physics*, 41(1-2):80–100, 2012.

[8] J. Eduard Hoogenboom, William R. Martin, and Bojan Petrovic. The Monte Carlo performance benchmark test - aims, specifications and first results. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*, Rio de Janeiro, Brazil, May 2011.

[9] Seon Wook and Kim Rudolf Eigenmann. Where does the speedup go: quantitative modeling of performance losses in shared-memory programs. *Parallel Processing Letters*, 10:227–238, 2000.

[10] T. M. Sutton, T. J. Donovan, T. H. Trumbull, P. S. Dobreff, E. Caro, D. P. Griesheimer, L. J. Tyburski, D. C. Carpenter, and H. Joo. The MC21 Monte Carlo transport code. In *Proc. ANS Mathematics & Computation Division Topical Meeting, M&C2007*, Monterey, 2007.

[11] X-5 Monte Carlo team. Mcnp – a general n-particle transport code. Technical Report LA-UR-03-1987, Los Alamos National Laboratory.

[12] Jaakko Leppänen. *Development of new Monte Carlo reactor physics code.* PhD thesis, Helsinki University of Technology, 2007.

[13] Paul K. Romano and Benoit Forget. The OpenMC Monte Carlo particle transport code. *Ann. Nucl. Energy*, 51:274–281, 2013.

[14] Kord Smith. Reactor core methods. Invited lecture at the M&C 2003 International Workshop, April 2003.

[15] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[16] A.R. Siegel, K. Smith, P.K. Romano, B. Forget, and K. Felker. The effect of load imbalances on the performance of Monte Carlo algorithms in LWR analysis. *Journal of Computational Physics*, (0):–, 2012.

[17] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a million processors.

In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Jeffrey R. Diamond, Martin Burtscher, John D. McCalpin, Byoung-Do Kim, Stephen W. Keckler, and James C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *ISPASS*, pages 32–43, 2011.

[19] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *ISPASS*, pages 2–11, 2011.

[20] Browne Dongarra Garner, S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14:189–204, 2000.