Performance Study of Monte Carlo Codes on Xeon Phi Coprocessors — Testing MCNP 6.1 and Profiling ARCHER Geometry Module on the FS7ONNi Problem

Tianyu Liu[1], *Noah* Wolfe[1], *Hui* Lin[1], *Kris* Zieb[1], *Wei* Ji[1], *Peter* Caracappa[1], *Christopher* Carothers[1], and *X. George* Xu[1,⋆]

[1] *Nuclear Engineering, Rensselaer Polytechnic Institute (RPI), Troy, New York, USA, 12180*

Abstract. This paper contains two parts revolving around Monte Carlo transport simulation on Intel Many Integrated Core coprocessors (MIC, also known as Xeon Phi). (1) MCNP 6.1 was recompiled into multithreading (OpenMP) and multiprocessing (MPI) forms respectively without modification to the source code. The new codes were tested on a 60-core 5110P MIC. The test case was FS7ONNi, a radiation shielding problem used in MCNP's verification and validation suite. It was observed that both codes became slower on the MIC than on a 6-core X5650 CPU, by a factor of ∼4 for the MPI code and, abnormally, ∼20 for the OpenMP code, and both exhibited limited capability of strong scaling. (2) We have recently added a Constructive Solid Geometry (CSG) module to our ARCHER code to provide better support for geometry modelling in radiation shielding simulation. The functions of this module are frequently called in the particle random walk process. To identify the performance bottleneck we developed a CSG proxy application and profiled the code using the geometry data from FS7ONNi. The profiling data showed that the code was primarily memory latency bound on the MIC. This study suggests that despite low initial porting effort, Monte Carlo codes do not naturally lend themselves to the MIC platform — just like to the GPUs, and that the memory latency problem needs to be addressed in order to achieve decent performance gain.

1 Introduction

In recent years hardware acceleration using Many Integrated Core coprocessors (MICs) made by Intel or Graphics Processing Units (GPUs) by Nvidia has become increasingly common in scientific computing. As of June 2016, these two types of accelerators have been employed in 89 supercomputers on the Top-500 list [1] and, most impressively, in 19 out of the top 20 supercomputers on the Green-500 list [2] which ranks the systems in energy efficiency (floating-point operations per Joule). Several national laboratories in the U.S. have been building their next-generation supercomputers based on accelerators, notably "Trinity" at LANL, "Cori" at NERSC, "Aurora" at ANL, "Summit" at ORNL and "Sierra" at LLNL.

With the advent of these new platforms, of our particular interest in nuclear engineering and science community is how to effectively speed up some routinely used but extremely slow applications such as Monte Carlo simulation of radiation transport. Two specific questions from developers are: ① *how hard is it to port existing codes to accelerators, how good is the performance and what is the bottleneck*, ② *how hard is it to perform accelerator-specific optimization?*.

1.1 Easiness of porting

Currently the MICs (Knights Corner generation) and GPUs (Kepler and Maxwell generations) are not binary compatible with the CPUs, which means existing programs cannot directly run on accelerators.

For GPUs, the codes need to be rewritten in Nvidia's GPU-specific Application Programming Interface (API) called CUDA [3]. Users' programming responsibility typically includes determining parts of codes to be run on the GPUs in parallel, devising appropriate multithreading strategy, modifying those parts of codes as GPU kernel or device functions, managing GPU memory whose address space is separate from the host system memory, and controlling data transfer between the host and GPUs. At present, porting large-scale production or legacy codes completely to CUDA could be a onerous task despite not being entirely impossible. None the less, over years we have observed a steady evolution in CUDA with respect to programmability. Examples include the development of "CUDA runtime API" built on the original low-level driver API, which significantly reduces the amount of boilerplate codes and improves readability, "unified memory" which carries the burden of memory management to some extent by eliminating the need for explicit data copy. The cost of porting will likely continue to drop in the years to come. In addition, alternative languages do exist, such as

[⋆] e-mail: xug2@rpi.edu

the compiler directive based OpenACC [4] and new version of OpenMP (> 4.0) [5], to facilitate code porting at the cost of less functionality and lower performance than CUDA.

For MICs, code porting is usually easier. The fastest solution is a straightforward "recompile and run". Using special compiler flags MIC-compatible programs can be directly generated, and the MIC coprocessor is simply used as a separate compute node. Production codes can be ported this way effortlessly to enable a quick test drive. The drawback, however, is that the entire program has to run on the MICs and the serial parts can be very slow due to low single-core performance. Besides, the entire data to be used by the serial and parallel parts need to fit the memory on the MIC node and be copied to it beforehand. Solutions without this limitation are available but require code changes. Examples are (1) Intel MPI [6]. It allows communications between MPI processes on the host node and MIC node, thus making CUDA-like, heterogeneous computing possible. (2) Compiler directive based languages such as OpenACC, OpenMP and Intel offload pragma [7]; (3) Intel APIs such as "Cilk Plus" [8] which uses a shared memory model analogous to CUDA's unified memory, "Coprocessor Offload Infrastructure (COI)" [9] which is a low-level, versatile API that permits fine control of memory allocation and copy.

1.2 Potential problems of directly ported code

While MIC's "recompile and run" significantly reduces the initial porting cost, we have some *theoretical* concern over the performance of Monte Carlo codes generated this way. (1) The MICs are most suited to vectorized computations for compute-bound programs. The unique Single-Instruction Multiple Data (SIMD) 512-bit wide registers on the MICs allow simultaneous operations on 8 double-precision data at a single instruction. However, the history-based Monte Carlo codes cannot directly benefit from this feature: Each compute thread still tracks a single particle at a time instead of 8; the prevalent `do-while` loops that require indefinite steps to exit, and the data dependency across different iterations within a loop, essentially inhibit compiler's capability to perform automatic vectorization. (2) The Knights Corner (KNC) generation of MICs use the legacy in-order execution processors. Such processors will stall when waiting for the data to be loaded from the memory to the cache then to the registers. Hardware threading on the MICs mitigates this problem but to a limited degree. Monte Carlo codes usually require intensive memory access in a random, scattered pattern, and there is a fair chance that data are not available in the cache when needed for computation and must be loaded from the memory, leading to processor stall and long latency.

This paper uses *experimental* approach to answer question ① by examining two Monte Carlo codes on the MIC platform. In the first part of the paper, MCNP 6.1 was recompiled into MIC-compatible parallel codes with multithreading and multiprocessing capacity, respectively. The performance was studied using a strong scaling test. In the second part, the performance of a Constructive Solid Geometry (CSG) proxy application in ARCHER was characterized. Both experiments used the radiation shielding benchmark problem "FS7ONNi" in MCNP's verification and validation suite.

2 Part 1: Testing of MCNP 6.1

2.1 Method

The MCNP 6.1 source code and build system (using the GNU make tool set) support build configurations for both OpenMP (shared memory model) and MPI (distributed memory model) parallel computing approaches. Cross compilation of the large MCNP 6.1 code base was performed on the host x86/x86_64 machine in order to generate MIC compatible binary executables to run on the MIC device. The Intel fortran ("ifort") compiler was used to compile the fortran code, and the Intel C ("icc") and MPI ("mpiicc") compilers were used to compile the C code for OpenMP and MPI executables respectively. The version of these tools is 16.0.2. During cross compilation, the `-mmic` flag was added to both the compiler and linker to reference the MIC compatible libraries so that the emitted binary was able to execute the Initial Many Core Instructions (IMCI) on the MIC.

The test case FS7ONNi is a neutron transport problem, where a disc source is placed in a room ($9.0\times6.9\times6.8\ m^3$) with multiple shielding objects in it. There are 178 cells and 61 surfaces in total. The flux at a certain point in the room (point detector tally) is calculated.

The MIC coprocessor is 5110P with KNC architecture, which has 60 cores at 1.052 GHz and 7.75 GB memory (ECC enabled). Two CPUs were used to compare against the MIC. They are X5650 CPU with Westmere architecture and 6 cores at 2.66 GHz, and E5-2697 v3 CPU with Haswell architecture and 14 cores at 2.6 GHz. Each core of the MIC supports four hardware threads (also commonly known as hyperthreads), whereas that of the CPUs support two.

2.2 Result

2.2.1 OpenMP/MPI on a single MIC

Both the OpenMP and MPI versions of MCNP were run in the "native" execution mode. In this mode, the simulation was entirely performed on the MIC alone. The executable files, cross-section data, user-input, and dynamic libraries were uploaded to the MIC in advance.

In the first test one MIC coprocessor was used. The threads or processes were evenly distributed among 60 physical cores. Simulation with only 1 particle was performed at first to determine the time of initialization and finalization. This process was found to be ~22 seconds on the KNC MICs (due to low single-core performance) and ~2 seconds on the Westmere and Haswell CPUs. This value was subtracted from the time of complete simulation to derive the time of parallel particle transport.

The scalability of OpenMP and MPI on the MIC is compared in Figure 1. The sweet spot was found to be 60 processes for the MPI version with 1 process pinned to each physical core, and 40 threads for the OpenMP version with one thread on each physical core where some cores were unused. At 60 or less MPI processes or OpenMP threads, one process or thread possesses full utilization of its corresponding core's cache and memory access system. The particle transport process consists of a large amount of irregular memory accesses. Therefore, memory latency is vital in order to supply the execution pipelines of each core. Moving past 60 MPI processes, or OpenMP threads, increases the demand on each core's cache and memory access system. As a result, performance is diminished as execution pipelines are starved waiting for memory. Furthermore, it was observed that over subscription of MIC resulted in serious performance degradation. For example, when 480 threads were launched, the computation time almost quadrupled that of 240 threads. Such cases always have remarkably high "system CPU time", which is presumably caused by a disproportionate increase in the parallel overhead. It is worth mentioning that despite better performance, MPI with distributed memory model is not intended for use on a single node such as one MIC card, as the memory is soon to be consumed by the processes with separate address space. In fact, for our test case one MIC can only run 120 processes at a time.

According to [10], the scalability of multithreaded OpenMP was largely reduced when the compiler flag `-heap-arrays` was applied to an earlier version of compiler. We compiled the program with `-heap-arrays` and `-no-heap-arrays` (this is also compiler's default flag) flags respectively and compared their performance in Figure 1. In general, `-no-heap-arrays` resulted in faster code, since this flag places statically allocated arrays on the stack which has faster access than the heap memory. However, it did not improve scalability appreciably.

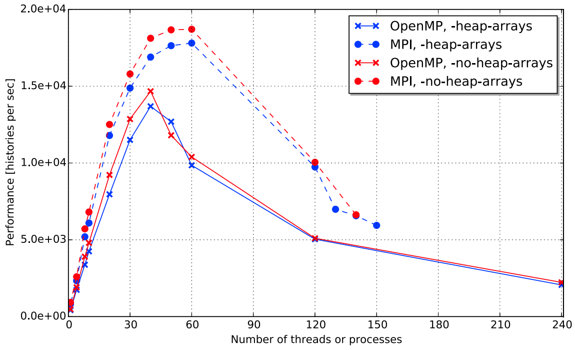

Figure 1: Comparison of MCNP 6.1 OpenMP and MPI performance when the program is compiled with `-heap-arrays` and `-no-heap-arrays` flags respectively. Only the time of parallel particle transport is considered. The performance is normalized to particle histories simulated per second.

The OpenMP scalability of MCNP 6.1 on the MIC and CPUs are compared in Figure 2. On both CPUs the scalability are much better, as linearity holds until all the physical cores are used, beyond which hardware threads help to hide memory access latency and improve the performance to some extent. Compared to Westmere and Haswell CPUs, the peak performance on the MIC was found to be lower by a factor of 4.2× and 14×, respectively, shown in Table 1. This is believed to be a result of the underlying interconnection network of cores and memory controllers within the MIC architecture. The current generation MIC based on KNC architecture uses a bi-directional multi-layered ring interconnect for data transfer between cores and memory controllers and can quickly get saturated with the irregular memory access of MCNP. The future Knights Landing (KNL) and Knights Hill (KNH) architectures with their proposed 2D mesh interconnect may benefit memory performance.

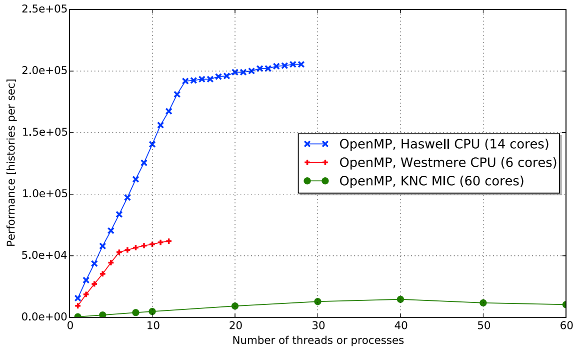

Figure 2: Strong scaling of MCNP 6.1 on the MIC and CPUs. Only the time of parallel particle transport is considered. The performance is normalized to particle histories simulated per second. The Haswell CPU outperforms KNC MIC by a wide margin.

Table 1: Peak performance comparison of MCNP 6.1 on the CPUs and MIC. The compute time of the parallel particle transport is normalized to per 1 million particles.

Processor	Number of execution unit	Compute time[sec]
6-core Westmere CPU	12 threads	16
14-core Haswell CPU	28 thread	4.9
60-core KNC MIC	40 threads	85

3 Part 2: Development and Profiling of CSG Module in ARCHER

3.1 Method

3.1.1 CSG Module and Data Structure

The CSG module in ARCHER is based on OpenMC [11], an open source neutronics Monte Carlo transport code in Fortran. Several basic features have been inherited from it, such as the capability to handle universe-based geometries

DOI: 10.1051/epjconf/201715306022

and lattice structures. Our modifications include porting it to ARCHER's CPU-GPU-MIC framework, rewriting in C++11, adding the functionality of reading and parsing MCNP input files.

The CSG data consists of cell and surface arrays, shown in Figure 3. Both arrays are a list of pointers, with each element pointing to an object. The objects are not contiguous in memory since they are created at different points when the MCNP input file is parsed. The cell objects are instantiated from a single class, whose data members include cell ID, universe ID, material ID and a vector of indices of the bounding surfaces. The surface objects vary in size, as they are instantiated from different classes (planes, cylinders, spheres, etc). These surface classes are derived from a common base class whose data members include surface ID and vectors of indices of the cells in the positive or negative sense of the surface. The difference of these surface classes is the number of geometry parameters. For instance, a plane perpendicular to x axis contains 1 parameter ($x0$), while a sphere contains 4 ($x0, y0, z0, r$). In order to obtain all data in a cell or surface object from the memory, the program needs to undergo a few levels of indirection — the pointer of the object, the object itself, and the index vectors. This data layout indicates that memory access can be subject to long latency in case of cache misses on the MIC, whereby the data are not readily available in the cache. Cache misses are simulated in our proxy application described in the next section.

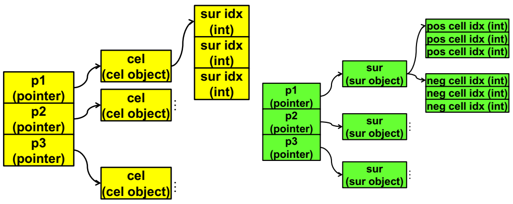

Figure 3: Memory layout of CSG cell (yellow) and surface (green) data structure. Separate boxes indicate that the data are not contiguous in memory. Accessing all data in a cell or surface requires several levels of indirection.

3.1.2 CSG Proxy Application

The functions in the CSG module are intensively used in the particle tracking process. In order to study the properties of this module we developed a proxy application for it. A proxy application [12] is *a catchall term for the simplification of characteristics of real applications that are of interest to DOE*. Our CSG proxy application falls into the "mini app" category where the CSG module combined with an artificial, simplified interaction model forms a stand-alone application. The interaction model assumes one-group transport and only simulates absorption ($\Sigma_a = 0.001 cm^{-1}$) and isotropic scattering ($\Sigma_a = 0.099 cm^{-1}$). The particle's path-length in the entire spatial region is scored as the program output.

The workflow of the CSG proxy application is shown in algorithm 1. In a full Monte Carlo transport code, some events such as constructing macroscopic cross-sections, interpolating pretabulated data, and accumulating tallies, entail memory read or write operations to other memory and can cause the CSG data to be evicted from the cache. This effect has been taken into account in the proxy application. Specifically, all the cell and surface objects were manually evicted from both L1 and L2 cache on the MIC. The eviction was implemented using compiler intrinsic function `_mm_clevict(p, hint)`, where `p` is the memory address and `hint` specifies eviction mode. Allowed options include eviction from L1 only and from both L1 and L2.

Table 2 lists three most time-consuming functions in the CSG proxy application. The instrumented function `EvictCSGCache` appears on the top mainly because the overhead of artificial L2 cache eviction is very large. Our current focus is to study the 2nd and 3rd hotspots.

Table 2: Top 3 hotspots in the CSG proxy application.

Function	Time percentage [%]
EvictCSGCache	56%
FindCell (FC)	13%
FindDistanceToBoundary (FD)	12%

To characterize the performance of CSG proxy application, we derived the following 5 performance metrics from the profiling results. The exact definition of these metrics can be found in Intel's reference [13]. The profiler "Intel VTune Amplifier XE 2016" was used to collect the metrics.

- *Vectorization Intensity (VI).*
  A measure of how effectively the 512-bit wide SIMD register is utilized overall. Higher is better. On MICs both scalar and vector operations use vector registers via masks. For pure scalar operations, one bit of the mask is set so that only one 64-bit double-precision data element is applied (VI=1), while for full vector operations, all bits of the mask are set, allowing all 8 data elements (VI=8).
- *L1 compute to data access ratio (L1CD).*
  A measure of how many computations are performed per L1 cache access. Higher is better. When L1CD is smaller than VI, the program is considered not computationally dense.
- *L2 compute to data access ratio (L2CD).*
  A measure of how many computations are performed per L2 cache access. Higher is better. When L2CD is smaller than 100 × L1CD, the program is considered to have too many L1 cache misses [13].
- *L1 hit rate (L1H).*
  A measure of the probability that the data hits L1 cache per memory access. Higher is better. This metric is a conservative estimate of L1 hits, as it takes into account a special type of L1 cache misses, where the data are not in L1 cache but are being prefetched into L1. When L1H is smaller than 95%, the program is considered to have too many L1 cache misses [13].

- *Estimated Latency Impact (ELI)*.
  An approximation of the clock cycles spent on L2 cache access and memory access per L1 cache miss. Lower is better. On MICs, a conservative estimate of L2 hit rate is unavailable due to the limitation that the special type of L2 cache misses, where the data are not in L2 cache but are being prefetched into L2, cannot be properly determined. Intel recommend using ELI as a workaround. If all data not in L1 are found in L2, then ELI=21 (lower bound), which is the clock cycles of a L2 cache access after a L1 cache miss. If there are L2 cache misses, it will take extra hundreds of clock cycles to load the data from the system memory. When ELI is larger than 145 clock cycles, the program is considered to suffer from long access latency due to L2 cache misses [13].

3.2 Results

3.2.1 Performance Characterization

The profiling results for the two time-consuming functions `FindCell` and `FindDistanceToBoundary` are listed in Table 3. The VI values of both functions are 2, which means the SIMD feature of MIC is almost completely untapped. In fact, by checking compiler's vectorization report, no automatic vectorization was made for these functions at all, and VI values should be 1. It is speculated that the mask operations in scalar calculations also count as vector operations and hence contribute to VI. The fact that L1CD values are lower than VI is an indicator that the code is not computationally dense but instead memory access demanding. Specifically, the cache utilization is inefficient, confirmed by the borderline low L1H, the significantly low L2CD, and the high ELI value. These values suggest that due to L1 cache (only costing 1 clock cycle) misses, access to L2 cache (costing 21 clock cycles) is frequent. To exacerbate the situation, those data that miss L1 cache mostly do not hit L2 cache, and thus can only be loaded from the main memory. The resulting performance penalty is quantified by the ELI values, which exceed the 145 clock cycle threshold. These data indicate that the two functions are memory latency bound.

Table 3: Profiling result of ARCHER's CSG proxy application on the MIC. Refer to subsubsection 3.1.2 for the metrics' full name. FC: FindCell, FD: FindDistanceToBoundary.

Metrics	FC	FD	Investigate if
VI	2	2	< 8
L1CD	1.3	1.4	< VI
L1H	96%	92%	< 95%
L2CD	34	18	< 100 × L1CD
ELI	1135	972	> 145

It should be pointed out that that the geometry data in the FS7ONNi test case are not sufficiently large in size. Therefore two important metrics — L1 and L2 translation lookaside buffer (TLB) miss ratios [13] were found trivial in this performance study. They are, however, useful when data significantly exceed the memory page size, i.e. 2 MB on the MIC.

The result of the strong scaling test is shown in Figure 4. Again, only the parallel particle transport was timed. The OpenMP thread affinity was set to "balanced" whereby threads are evenly spread across 60 cores. Scalability observed was not satisfactory as the performance increase started to become sublinear before 60 physical cores are used. However, what is better than part 1 is that the increase continued and reached its peak when all 240 hardware threads were used. Here the compute performance of MIC and CPU codes was not compared. This is because on the CPU, there is no intrinsic function equivalent to `_mm_clevict(p, hint)` that simply performs cache eviction. The most similar, CPU-specific intrinsic function `_mm_clflush(p)` invalidates the cache line but also writes data back to the memory. Therefore it has large overhead and the performance result would be heavily biased against the CPU.

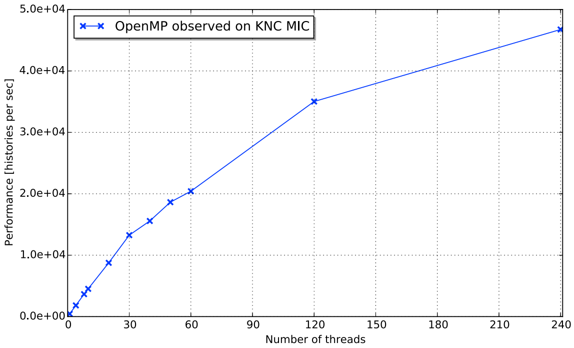

Figure 4: Strong scaling of ARCHER's CSG proxy application on the MIC. Only the time of parallel particle transport is considered. The performance is normalized to particle histories simulated per second.

4 Discussion

The bottleneck of Monte Carlo codes on the MICs are the lack of vectorization and excess of memory access latency. Solutions to these problems exist, in spite of being only effective to some specific parts of Monte Carlo codes. For example, the macroscopic cross-section construction subroutine is usually one of the hotspots in neutronics transport simulation. This subroutine has a relatively simple structure and offer good opportunity for optimization. In our recent study [14], we optimized the XSBench code [15] to the accelerators. XSBench abstracts the macroscopic cross-section construction process from a full Monte Carlo code OpenMC [11]. Compared to the original XSBench run on a 6-core CPU, our directly ported codes were found to have a speedup factor of only 2.7× on a MIC (60 cores) and 1.4× on a GPU (15 streaming processors), whereas these numbers rose to 6.0× and 8.1×, respectively, as a result of several optimization techniques that focuses on vectorizing the computation and hiding memory latency. Another example is the study by

[16], where the OpenMC code was tested on the MICs and OpenMP-specific optimizations were made to reduce tally overhead, such as replacing OpenMP's `critical` sections with `atomic` operations.

5 Conclusions

In this study we tested two Monte Carlo codes on the Knights Corner generation of MIC coprocessor: the production code MCNP 6.1, and the CSG module of our ARCHER code developed specifically for the heterogeneous architecture. The experiments have confirmed our concern that the Monte Carlo codes do not naturally lend themselves to the KNC MIC coprocessors, because the 512-bit wide SIMD registers are underutilized, and the random, scattered memory access pattern is not cache-friendly and causes long memory latency. Such problems also occur to the GPUs which implement Single instruction, multiple thread (SIMT) model and have even less amount of cache per thread. Software optimizations and algorithm improvements must be conducted in order for Monte Carlo codes to achieve decent performance on accelerators.

Acknowledgments

We thank OpenMC community for the high quality code and Intel Corp. for the hardware donation.

References

[1] *Top 500 list* (2016), `https://www.top500.org`

[2] *Green 500 list* (2016), `http://www.green500.org/`

[3] Nvidia, *CUDA C Programming Guide* (2016)

[4] OpenACC committee, *The OpenACC application programming interface, version 2.5* (2015)

[5] OpenMP committee, *The OpenMP application programming interface, version 4.5* (2015)

[6] Intel, *Intel MPI library reference manual for Linux OS* (2016)

[7] Intel, *Intel C++ compiler 16.0 user and reference guide* (2016)

[8] Intel, *Intel Cilk Plus language extension specification, version 1.2* (2013)

[9] C.J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, R. McGuire, *Offload compiler runtime for the Intel Xeon Phi coprocessor*, in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (IEEE, 2013), pp. 1213–1225

[10] O. Mikhail, *Openmp and -heap-arrays not compatible since ifort 13?* (2016)

[11] P.K. Romano, B. Forget, Annals of Nuclear Energy **51**, 274 (2013)

[12] M. Heroux, R. Neely, S. Swaminarayan, Tech. Rep. LA-UR-13-20460 and LLNL-TR-592878, Sandia, LLNL, LANL (2013)

[13] S. Cepeda, *Optimization and Performance Tuning for Intel Xeon Phi Co-processors, Part 2: Understanding and Using Hardware Events* (2012)

[14] T. Liu, N. Wolfe, C.D. Carothers, W. Ji, X.G. Xu, *Optimizing the Monte Carlo neutron cross-section construction code, XSBench, for MIC and GPU platforms (in press)* (2016)

[15] J.R. Tramm, A.R. Siegel, T. Islam, M. Schulz, *XS-Bench - The development and verification of a performance abstraction for Monte Carlo reactor analysis*, in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future* (Kyoto, 2014)

[16] D. Ozog, A.D. Malony, A. Siegel, *Full-core PWR transport simulations on Xeon Phi clusters*, in *M&C + SNA + MC 2015 — Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method* (American Nuclear Society, 2015)

Appendix

```
input  : Geometry data
input  : n: number of histories
output: Particle path-length in all cells

for $i \leftarrow 0$ to $n - 1$ do
    InitializeParticle()
    while true do
        if cell not found then
            FindCell()
        end
        d = FindDistanceToBoundary()
        s = SampleDistanceToCollisionSite()
        UpdatePosition()
        ScorePathLength()
        EvictCSGCache() // manually evict data
        if $d < s$ then
            BoundaryCrossing() // contains calls of
            FindCell()
        else
            Collision()
        end
        if particle is killed then
            break
        end
    end
end
```

Algorithm 1: Pseudocode of the CSG proxy application