LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-665607

Advanced Computing Architecture Challenges for the Mercury Monte Carlo Particle Transport Project

P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O'Brien, D. E. Stevens, B. R. Beck, E. D. Brooks III, R. C. Bleile

December 22, 2014

ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method
Nashville, TN, United States
April 19, 2015 through April 23, 2015

Disclaimer

ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method · Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015)

ADVANCED COMPUTING ARCHITECTURE CHALLENGES FOR THE MERCURY MONTE CARLO PARTICLE TRANSPORT PROJECT

Patrick S. Brantley, Shawn A. Dawson, Michael Scott McKinley, Matthew J. O'Brien, David E. Stevens, Bret R. Beck, Eugene D. Brooks III
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94551
brantley1@llnl.gov; dawson6@llnl.gov; quath@llnl.gov; obrien20@llnl.gov;
stevens9@llnl.gov; beck6@llnl.gov; brooks3@llnl.gov

Ryan C. Bleile
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403
rbleile@cs.uoregon.edu

ABSTRACT

We describe the challenges posed to the Mercury Monte Carlo particle transport code development team from emerging and future advanced computing architectures. We review recent work to scale Mercury to large numbers of MPI processes as well as to improve compute node parallelism via OpenMP threading and demonstrate these capabilities using a reactor eigenvalue calculation. We then describe initial progress for enabling Mercury for the Intel Xeon Phi-based MIC architecture. We present preliminary results of research investigations into the use of event-based algorithms in a Monte Carlo test code for application to GPU architectures. We then briefly describe work to enable storage of nuclear data in shared memory and to enable the use of the Generalized Nuclear Data format in Mercury via the General Interaction Data Interface.

Key Words: Monte Carlo transport, parallel algorithms, MIC, GPU

1 INTRODUCTION

Mercury is a Monte Carlo particle transport code under development at Lawrence Livermore National Laboratory (LLNL) [1, 2]. Mercury can transport neutrons, photons, and light element (hydrogen and helium) charged particles. Both fixed source and criticality problems are treated. Mercury runs efficiently on current generation massively parallel computing platforms, and a major current focus is to enable its use on emerging advanced computing architectures.

Mercury uses domain decomposition and domain replication [3] along with load balancing [4] to enable efficient and scalable calculations with MPI parallelism. We have recently performed a significant amount of algorithmic work to enable Mercury to scale to large numbers of MPI processes [5, 6] and to improve compute node parallelism via OpenMP threading [1].

Power consumption considerations are driving future high performance computing platforms toward advanced computing architectures. Mercury is expected to run efficiently on the advanced architecture machines being procured through the National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) Program. The current Sequoia machine [7] at Lawrence Livermore National Laboratory is an IBM 20 petaFLOP/s machine with 16 PowerPC A2 1.6 GHz cores per node (with four hardware threads per core) and 16 GB memory per node – lower memory per core than typical current generation platforms. The Trinity machine to become available at Los Alamos National Laboratory (LANL) beginning in the late 2015 time frame will use both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing many integrated core (MIC) architecture coprocessors. The Sierra machine to be available at LLNL beginning in the 2018 time frame will use an IBM PowerPC architecture along with Nvidia graphics processing unit (GPU) architecture accelerators. As a result of these different advanced architectures, the computing landscape for the upcoming years is complex. A significant challenge for the Mercury project is to simultaneously support efficient versions of the code for both the current generation and advanced computing architectures within a single source code base.

In this paper, we briefly review the work of the Mercury code development team to prepare for the emerging diversity of computing architectures. In Section 2, we describe recent efforts to assess and improve the scalability of the code to large numbers of MPI processes. In Section 3, we briefly describe the OpenMP threading capability in the Mercury code. Section 4 describes initial work to enable Mercury to run on Intel MIC architectures and presents initial numerical results. Section 5 presents preliminary research investigations of an event-based Monte Carlo algorithm implemented on a GPU for a Monte Carlo test code. We conclude with a discussion of a new shared memory implementation of nuclear data that reduces memory usage in Section 6 as well as initial capabilities to use the Generalized Nuclear Data format in Mercury in Section 7.

2 MPI PARALLEL SCALING

We have devoted a substantial amount of effort in recent years to improving the parallel scalability of Mercury to large (of order millions) numbers of MPI processes. Here we define *scalability* as the ability of a code to perform efficiently as the number of processes increases. An example of a code that is *non-scalable* is one that exhibits a run time proportional to the number of processes. An example of a code that is *scalable* is one that exhibits a run time proportional to the logarithm of the number of processes. Enabling a code to be scalable to millions of processes requires attention to the details of algorithms and memory usage that can be safely ignored when considering scaling to only a few thousand processes.

The specific Mercury algorithms improved for scalability include the algorithms for sourcing particles, for globally resolving particle locations onto the correct process, for load balancing the particle workload across processes, and for determining at what point particle streaming communication has completed for domain-decomposed problems [6]. Sourcing only process-local particles coupled with a scalable global particle find algorithm leads to scalable particle sourcing. The scalable load balancing algorithm considers only process-wise pairs instead of using a non-scalable global view of the workload. In addition to addressing parallel scalability of the algorithms

themselves, the memory usage of the various algorithms has also been carefully scrutinized to enable scalability. Further details of this work are described in Refs. [5] and [6].

We previously presented [1] weak scaling results for a Godiva critical sphere calculation up to over two million MPI processes on the LLNL IBM Sequoia computer [7]. Mercury exhibited excellent weak scaling in the particle tracking time for this geometrically simple eigenvalue calculation, with the particle tracking time varying by less than 7% as the number of MPI processes increased from 64 to 2,097,152.

To further examine the efficiency of the Mercury MPI parallel implementation for a more complicated calculation, we used a Mercury model of the Annular Core Research Reactor (ACRR) [8] located at Sandia National Laboratory. The ACRR is a water-moderated pool research reactor that can be operated in steady-state or pulsed mode. The reactor core is composed of a 236 element array of uranium dioxide/beryllium oxide (enriched to 35 weight percent) with stainless steel cladding. The ACRR has a dry irradiation cavity in the center of the core that can be used to expose experiments to neutron and gamma fluxes. The Mercury ACRR model is composed of 2,846 geometric cells and is shown (without the water moderator) in Fig. 1. We present results for the "Up DC" control rod position for delayed neutron criticality in the ACRR with the transient rods in the "full up" configuration and the 32 inch pedestal in the central cavity [8].

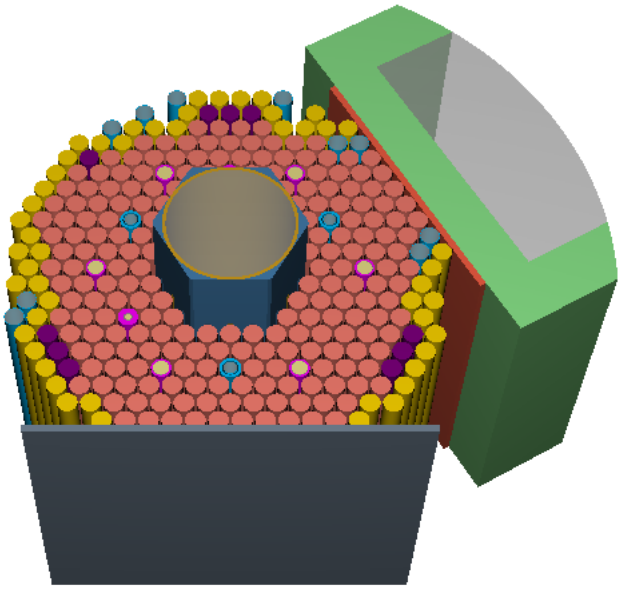

Figure 1. Mercury ACRR model

We performed a weak scaling study in which the amount of computational work per MPI process is maintained approximately constant as the number of MPI processes is increased. We performed a k-eigenvalue calculation using continuous energy LLNL TART2004 nuclear data, 25 initial generations discarded prior to averaging the eigenvalue, and 25 active generations. The 64 MPI process run used 10^6 Monte Carlo particles per generation, and we doubled the number

of particles each time we doubled the number of MPI processes. We performed the simulations on the LLNL IBM Sequoia machine [7], an IBM 20 petaFLOP/s machine with 16 PowerPC A2 1.6 GHz cores per node (with four hardware threads per core) and 16 GB memory per node. We investigated weak scaling for both the total calculation time and the particle tracking time by computing a parallel efficiency defined as the ratio of the wallclock for 64 MPI processes divided by the wallclock for the given number of MPI processes. The Mercury calculation with 64 MPI processes produced $k_{eff} = 1.000854 \pm 0.000254$ which statistically agrees with the reported MCNP value of $k_{eff} = 1.000146 \pm 0.001313$ [8] obtained by averaging eleven calculated k_{eff} values used to compute the delayed critical rod position.

The weak scaling results are shown in Table I. The particle tracking exhibits nearly perfect weak scaling as the number of MPI processes is increased from 64 to 131,072. The total calculation also demonstrates excellent weak scaling up to 131,072 MPI processes, with a parallel efficiency of 0.95 and higher at all MPI process counts.

Table I. ACRR Eigenvalue Calculation Sequoia Weak Scaling

Number	Total Calculation		Particle Tracking	
MPI Processes	Time [s]	Parallel Efficiency	Time [s]	Parallel Efficiency
64	3.77e+04	—	3.48e+04	—
128	3.78e+04	1.00	3.46e+04	1.00
256	3.80e+04	0.99	3.47e+04	1.00
512	3.81e+04	0.99	3.46e+04	1.00
1,024	3.86e+04	0.98	3.49e+04	1.00
2,048	3.90e+04	0.97	3.50e+04	0.99
4,096	3.95e+04	0.96	3.52e+04	0.99
8,192	3.91e+04	0.97	3.48e+04	1.00
16,384	3.94e+04	0.96	3.50e+04	0.99
32,768	3.95e+04	0.95	3.50e+04	0.99
65,536	3.93E+04	0.96	3.46E+04	1.01
131,072	3.96E+04	0.95	3.46E+04	1.01

We note that the calculation times shown in Table I for the Sequoia machine and a given number of MPI processes are significantly longer than on a current generation capacity platform (which is one driver to use larger numbers of processes on Sequoia). For example, the ACRR eigenvalue calculation using 64 MPI processes on the LLNL RZMerl Linux cluster with 16 Intel Xeon Sandy Bridge 2.6 GHz cores per node required 3.36e+03 seconds, over a factor of eleven less than on Sequoia. For the 64 MPI process case, utilizing the four hardware threads with OpenMP (more details in the next section) available on the Sequoia cores produces a calculation time of 1.68e+04 seconds, or a factor of five times longer than RZMerl.

3 OPENMP THREADING

Mercury originally exclusively used pure MPI to achieve parallelism on massively parallel computers. In this approach, MPI is used across compute nodes of the computational platform as well as on the cores of individual nodes. We have added the ability for Mercury to use threads via OpenMP to enable parallelism across the cores of each node. The user can now use a combination of both MPI and OpenMP to distribute the Monte Carlo work across the cores of a node. Each node may have one or more MPI processes, and each MPI process may use one or more threads to access compute cores on the node (and any hardware threads available for the cores). OpenMP pragmas are used to thread over particles, cells, and other sections of the code that perform significant work. OpenMP may be used with both spatial decomposition and spatial replication [3]. We note that MCNP has a similar combined MPI/OpenMP parallelism for the replication-only case [9]. The threading capability can also potentially result in significant memory savings, because nuclear data can be stored once per MPI process instead of once per core. This memory savings is expected to be important as future computer platforms move toward a larger numbers of nodes and cores per node coupled with lower memory available per node.

Coarse grain threading is achieved by creating a particle *vault* (list of particles to be tracked) for each thread and distributing particles evenly across the vaults. At a high level, each thread works on the particles in its vault. Implementing this capability requires an additional thread dimension in tally data structures to enable multiple threads to operate on particles independently without requiring thread critical sections that can degrade efficiency. This additional thread dimension in tally data structures does increase memory as compared to the use of critical sections; however, this memory would have also been used for per-process tallies in the MPI-only case. We also plan to assess the use of atomic memory updates as an alternative to duplication of memory. At the end of the particle transport, non-threaded code sums tallies over the threads to thread zero. Mercury also uses fine grained OpenMP parallelism at lower loop levels outside of the particle processing loop.

To examine the efficiency of the Mercury threading implementation, we performed a scaling study using the Mercury model of the Annular Core Research Reactor (ACRR) [8] described above. We performed a k-eigenvalue calculation using continuous energy LLNL TART2004 nuclear data, 10^7 Monte Carlo particles per generation, 25 initial generations discarded prior to averaging the eigenvalue, and 25 active generations. We performed a strong scaling study in which we fixed the problem size (geometry and number of Monte Carlo particles) and varied the number of processes used to simulate the problem from 256 to 8,192. We used a larger number of particles per generation for this strong scaling study so that the number of particles per core was not too small at higher core counts. We performed these strong scaling simulations on the LLNL IBM Sequoia computer [7]. We investigated the impact on the particle tracking time of the number of threads per node used (varied from one to sixteen) for each of the total number of core values. We also investigated the performance gains obtained by using the four hardware threads available on each IBM PowerPC A2 core of the node.

The wallclock time spent tracking particles is shown in Table II. The strong scaling of Mercury for this problem is excellent – doubling the number of processors consistently produces nearly a

factor of two decrease in particle tracking time. For a given number of cores, the particle tracking time is only weakly sensitive to the number of threads used (except when using sixteen MPI processes and one thread). These results demonstrate that the threading implementation for particle tracking is essentially as efficient as using MPI processes within a node. For this problem, the least efficient approach on Sequoia appears to be the use of sixteen MPI processes and one thread per MPI process. The wallclock times shown in Table II exclude the initialization and finalization times; including those times introduces up to approximately a 40% loss of efficiency at 8,192 cores as the number of particles to be tracked per core becomes small. This degradation in total calculation time efficiency implies that additional opportunities for threading exist in the initialization and finalization portions of the code.

Table II. ACRR Particle Tracking* Wallclock Time [s] Sequoia Strong Scaling

	Number MPI Processes × Number Threads Per MPI Process					
Number Cores	16 × 1	8 × 2	4 × 4	2 × 8	1 × 16	16 × 4
256	8.63e+04	8.18e+04	8.06e+04	8.09e+04	8.14e+04	3.60e+04
512	4.32e+04	4.12e+04	4.05e+04	4.08e+04	4.10e+04	1.85e+04
1,024	2.20e+04	2.08e+04	2.06e+04	2.07e+04	2.10e+04	9.53e+03
2,048	1.11e+04	1.06e+04	1.05e+04	1.06e+04	1.08e+04	4.94e+03
4,096	5.63e+03	5.42e+03	5.42e+03	5.50e+03	5.61e+03	2.63e+03
8,192	2.90e+03	2.81e+03	2.81e+03	2.89e+03	2.97e+03	1.39e+03

* Excludes initialization and finalization time

Finally, we examined the performance gains produced through the use of the four hardware threads available on each Sequoia PowerPC A2 core. The final column in Table II shows the particle tracking time for the case of sixteen MPI processes per node and four threads per MPI process (each core has four hardware threads). For all cases, the use of four hardware threads gives greater than a factor of two decrease in the particle tracking time.

4 INITIAL INTEL XEON PHI MIC ARCHITECTURE INVESTIGATIONS

The Trinity machine to become available at Los Alamos National Laboratory (LANL) beginning in the late 2015 time frame will use both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing many integrated core (MIC) architecture coprocessors. Our initial research approach for enabling Mercury to run efficiently on the Trinity machine is to leverage the recent OpenMP threading investment. We have ported Mercury to the Sandia National Laboratory (SNL) Morgan machine that has nodes with ten Intel Xeon Ivy Bridge 2.5 GHz cores and two Intel Xeon Phi Knights Corner MIC coprocessors. Each Knights Corner coprocessor has 57 1.1 GHz cores with four hardware threads per core, resulting in a maximum number of 228 threads per MIC coprocessor. Efficient use of Intel Xeon Phi MIC coprocessors requires a code that can continue to achieve speedups at this large number of threads.

To examine the initial efficiency of Mercury on the Intel Xeon Phi MIC coprocessors, we performed a strong scaling study on the SNL Morgan machine using the Mercury model of the Annular Core Research Reactor (ACRR) [8] described above. We performed a strong scaling study in which we fixed the problem size (geometry and number of Monte Carlo particles), used one MPI process on a single MIC coprocessor, and varied the number of threads from 2 to 224. (We attempted to leave one core on the MIC coprocessor unassigned to handle system operations.) We performed a k-eigenvalue calculation using continuous energy LLNL TART2004 nuclear data, 10^5 Monte Carlo particles per generation, 5 initial generations discarded prior to averaging the eigenvalue, and 5 active generations. We used fewer particles per generation and fewer generations on this platform to reduce the run time for the calculations with a small number of threads. We investigated strong scaling for the total calculation time, the total physics (particle tracking plus physics initialization/finalization) time, and the non-physics time by computing a parallel efficiency defined as the ratio of the observed speedup divided by the ideal speedup.

The strong scaling results for the ACRR problem are shown in Table III. The total physics time exhibits excellent scaling with a parallel efficiency above 0.94 up to 56 threads (using approximately all of the cores) and a parallel efficiency above 0.79 up to 112 threads (using approximately all of the cores and two hardware threads per core). The total physics efficiency degrades to 0.53 by 224 threads (using approximately all of the cores and four hardware threads per core). At 224 threads, each thread has less than 500 particles to track, however, so the degradation in efficiency is not completely unexpected. The time associated with the non-physics initialization and finalization parts of the code is also shown in Table III for each of the calculations. The time for the non-physics part of the code is approximately constant but increases slightly with the number of threads. The total calculation parallel efficiency is reduced as a result of the non-physics parts of the code. Additional investigation is required to understand the cause of the time spent in the the non-physics parts of the code.

Table III. ACRR Eigenvalue Calculation Morgan Strong Scaling

Number	Total Calculation		Total Physics		Non-Physics	
Threads	Time [s]	Parallel Eff	Time [s]	Parallel Eff	Time [s]	Parallel Eff
2	3.51E+04	1.00	3.45E+04	1.00	5.67E+02	1.00
4	1.74E+04	1.01	1.68E+04	1.03	5.68E+02	0.50
7	1.09E+04	0.92	1.03E+04	0.96	5.68E+02	0.29
14	5.55E+03	0.90	4.99E+03	0.99	5.69E+02	0.14
28	3.05E+03	0.82	2.48E+03	1.00	5.69E+02	0.07
56	1.89E+03	0.66	1.32E+03	0.94	5.71E+02	0.04
84	1.56E+03	0.53	9.90E+02	0.83	5.72E+02	0.02
112	1.36E+03	0.46	7.83E+02	0.79	5.74E+02	0.02
140	1.30E+03	0.38	7.28E+02	0.68	5.75E+02	0.01
168	1.24E+03	0.34	6.65E+02	0.62	5.77E+02	0.01
224	1.16E+03	0.27	5.81E+02	0.53	5.80E+02	0.01

5 PRELIMINARY NVIDIA GPU ARCHITECTURE INVESTIGATIONS

The Sierra machine to be available at LLNL beginning in the 2018 time frame will use an IBM PowerPC architecture along with Nvidia graphics processing unit (GPU) architecture accelerators. In this section, we describe the preliminary research investigations we have carried out related to GPU architectures.

Previous researchers [10–12] have noted that the use of an event-based Monte Carlo particle transport algorithm [13] may be beneficial for GPU or vector-based architectures. We investigated the implementation of an event-based Monte Carlo algorithm in the ALPSMC Monte Carlo test code [14] that models particle transport in a one-dimensional planar geometry binary stochastic medium. The ALPSMC code was originally implemented using a standard history-based Monte Carlo algorithm as shown in Alg. 1.

Algorithm 1: History-based Monte Carlo algorithm

```
1  foreach particle history do
2      generate particle from boundary condition or source
3      while particle not escaped or absorbed do
4          sample distance to collision in material
5          sample distance to material interface
6          compute distance to cell boundary
7          select minimum distance, move particle, and perform event
8          if particle escaped spatial domain then
9              update leakage tally
10             end particle history
11         if particle absorbed then
12             update absorption tally
13             end particle history
```

We investigated the event-based algorithm shown in Alg. 2 as a way to potentially optimize performance on GPU or vector-type architectures. In event-based particle tracking, the individual events can be treated by a series of data parallel operations. The data parallel model matches the GPU hardware with an emphasis on performing the same operations on many pieces of data at one time.

We implemented the event-based version of ALPSMC using both the Nvidia CUDA programming model [15] and the Nvidia C++ Thrust library [16] (based on C++ STL algorithms and syntax). Thrust is a useful and portable library that compiles into multiple language backends (serial C++, CUDA, OpenMP, etc.) and provides data containers such as a host and device vector. In addition, the Thrust library provides data parallel operations/algorithms such as `map`, `gather`, `scatter`, `scan`, `reduce`, and `sort`. Our Thrust implementation of ALPSMC utilizes many data parallel operations and utilizes Thrust data types for managing memory. However, using CUDA directly

Algorithm 2: Event-based Monte Carlo algorithm

```
1  foreach batch of particle histories (fits in GPU memory) do
2      generate all particles in batch from boundary condition or source
3      determine next event for all particles (collision, material interface crossing, cell
       boundary crossing)
4      while particles remaining in batch do
5          foreach event E in (collision, material interface crossing, cell boundary crossing) do
6              identify all particles whose next event is E
7              perform event E for identified particles and determine next event for these
               particles
8          if particle escaped spatial domain then
9              update leakage tally
10         if particle absorbed then
11             update absorption tally
12         delete particles absorbed or leaked
```

enables more fine grained control at the kernel level and enables important access to different memory spaces such as GPU shared memory. For example, Monte Carlo particles were initially allocated in GPU global memory and then copied to shared memory for all further operations within a kernel. All problem constants such as cross sections and mean chord length values were placed in GPU constant memory. In the native CUDA implementation of ALPSMC, we found it to be useful to continue to use Thrust algorithms in building various maps. ALPSMC is implemented using double precision floating point numbers throughout. The Thrust and CUDA implementations of ALPSMC give identical physics results to the original history-based implementation.

We performed scaling studies in which we varied the number of Monte Carlo particle histories (problem size) and the implementation methodology (Thrust or CUDA). (The results presented are for Case 1a [14] with a spatial domain of 10 cm.) We also examined the differences in performance on two different computer platforms. The RZGPU platform has Intel Xeon Westmere-EP 2.8 GHz host cores with Nvidia Tesla M2070 GPU device accelerators. The Max platform has Intel Sandy Bridge 2.6 GHz host cores with Nvidia Tesla K20X GPU device accelerators. The Tesla K20X GPU has improved double precision performance over the Tesla M2070. We computed speedups over a serial calculation by dividing the wallclock time of a serial run of the history-based version of ALPSMC on the host core of the given machine by the wallclock time of the event-based version of ALPSMC running on both the host and GPU device. The speedups obtained are shown in Table IV.

The speedups obtained using the CUDA implementation of the event-based algorithm are significantly larger than those obtained using the Thrust implementation by up to over a factor of four. We attribute this improved performance to the fact that CUDA offers more control over the memory spaces available on the GPU (e.g. shared memory). Thrust does not offer such flexibility and manages the memory allocation internally. We conclude based on these preliminary investigations that a direct CUDA implementation is more efficient than a Thrust implementation

for event-based Monte Carlo. Also, the speedups on the Tesla K20X GPU are larger than on the Tesla M2070 GPU by up to a factor of approximately two, presumably a result of the improved double precision performance of the K20X.

Table IV. ALPSMC Event-Based Monte Carlo GPU Speedups

	Number Particle Histories		
	10^6	10^7	10^8
CUDA (K20X)	5.90	11.88	11.91
CUDA (M2070)	3.96	6.05	6.05
Thrust (K20X)	2.11	2.60	2.60
Thrust (M2070)	1.42	1.64	1.63

6 SHARED MEMORY MPI FOR NUCLEAR DATA

The general expectation is that the memory available per core on future advanced architectures will continue to decrease. For example, current generation massively parallel capacity computers at LLNL typically have 2 GB per core, whereas the Sequoia machine has only 1 GB per core. Nuclear data can use a significant amount of memory in a Monte Carlo particle transport calculation depending on the number of isotopes in the calculation, and that nuclear data memory is duplicated across MPI processes on a node when the cores available on a node are all utilized as MPI processes.

One approach to reducing memory usage is to store a single copy of the nuclear data in shared memory on the node, with each of the MPI processes accessing as necessary the single copy of the nuclear data from shared memory. The MPI Shared Memory (MPISM) library [17] has recently been developed at LLNL to enable the use of shared memory on a node that can be accessed by any MPI process running on the node. Mercury uses the Monte Carlo All Particle Method (MCAPM) library [18] to access nuclear data and to perform collision physics. The MCAPM library has been modified to use the MPISM library such that one copy of the nuclear data can be stored per node instead of one copy per MPI process.

For example, a single copy of the nuclear data for the ACRR eigenvalue calculation described previously requires approximately 38 MB of memory. For a calculation on a machine with 16 cores per node in which all cores are treated as MPI processes, the total nuclear data storage is 38 MB $\times$ 16 MPI processes = 608 MB per node. Using the shared memory capability, a single copy of the nuclear data (38 MB) is stored per node for a memory savings of 570 MB per node. Identical physics answers are obtained with and without the use of shared memory, and the wallclock times for the two calculations are the same to within approximately 1%.

7 INITIAL GND/GIDI INTEGRATION

The Generic Nuclear Data (GND) format [19] is currently being developed at LLNL as a replacement for the legacy ENDL format used to store nuclear data at LLNL. The General

Interaction Data Interface (GIDI) library [19] is also being developed as the eventual replacement for the MCAPM library. The MCAPM library uses a monolithic data structure to store the nuclear data and related information for all isotopes in the problem. GIDI provides more flexibility to store and use the data for each isotope individually. The Mercury project views GIDI as an essential component of our strategy to enable the code to run efficiently on future computer architectures due to its more flexible data structures. GIDI is being actively integrated into Mercury, and current results will be presented in a companion paper [20].

8 CONCLUSIONS

We have reviewed the ongoing efforts of the Mercury Monte Carlo particle transport code development team to enable efficient use of emerging and future advanced computing architectures. We presented numerical results for an eigenvalue calculation of the ACRR reactor demonstrating excellent MPI scaling and threading performance. We also presented initial investigations of Mercury on an Intel Xeon Phi MIC-based architecture machine. Our results demonstrate that the transport physics part of the code exhibits excellent scaling, but the non-physics part of the code requires additional improvements to achieve acceptable overall performance. We also described preliminary investigations of event-based Monte Carlo algorithms for GPU architectures using a research Monte Carlo test code. We found that a CUDA implementation of an event-based Monte Carlo algorithm performed significantly better than a Thrust implementation, most likely a result of additional flexibility in access to different memory spaces on the GPU. We briefly described work to reduce memory usage by storing nuclear data in shared memory and quantified the memory savings for an ACRR eigenvalue calculation. Finally, we reviewed efforts to access and use nuclear data based on the Generic Nuclear Data format in Mercury.

9 ACKNOWLEDGMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. We acknowledge Richard "Spike" Procassini for providing the Mercury ACRR model.

10 REFERENCES

[1] P. S. Brantley et al., "Recent Advances in the Mercury Monte Carlo Particle Transport Code," *Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, Sun Valley, Idaho, May 5-9, 2013, (2013).

[2] "Mercury Web Site," (2014), http://mercury.llnl.gov.

[3] G. Greenman, M. J. O'Brien, R. J. Procassini, and K. I. Joy, "Enhancements to the Combinatorial Geometry Particle Tracker in the Mercury Monte Carlo Transport Code: Embedded Meshes and Domain Decomposition," *Proceedings of International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)*, Saratoga Springs, New York, May 3-7, 2009, (2009).

[4] R. J. Procassini, M. J. O'Brien, and J. M. Taylor, "Load Balancing of Parallel Monte Carlo Transport Applications," *Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*, Avignon, France, September 12-15, 2005, (2005).

[5] M. J. O'Brien, P. S. Brantley, and K. I. Joy, "Scalable Load Balancing for Massively Parallel Distributed Monte Carlo Particle Transport," *Proceedings of International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, Sun Valley, Idaho, May 5-9, 2013, (2013).

[6] M. J. O'Brien, *Scalable Domain Decomposed Monte Carlo Particle Transport*, Ph.D. Dissertation, University of California, Davis (2014).

[7] "Advanced Simulation and Computing: Sequoia," (2014), https://asc.llnl.gov/computing_resources/sequoia/.

[8] K. R. DePriest, P. J. Cooper, and E. J. Parma, "MCNP/MCNPX Model of the Annular Core Research Reactor," *Sandia National Laboratories Report SAND2006-3067* (2006).

[9] D. B. Pelowitz, "MCNP6™ User's Manual," *Los Alamos National Laboratory Report LA-CP-11-01708* (2011).

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

[11] T. Liu, X. Du, W. Ji, X. G. Xu, and F. B. Brown, "A Comparative Study of History-Based Versus Vectorized Monte Carlo Methods in the GPU/CUDA Environment for a Simple Neutron Eigenvalue Problem," *Proceedings of Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC)*, Paris France, October 27-31, 2013, (2013).

[12] R. M. Bergmann and J. L. Vujic, "Algorithmic Choices in WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs," *Annals of Nuclear Energy*, **77**, pp. 176–193 (2015).

[13] F. B. Brown and W. R. Martin, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, **14**, pp. 269–299 (1984).

[14] P. S. Brantley, "A Benchmark Comparison of Monte Carlo Particle Transport Algorithms for Binary Stochastic Mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, **112**, pp. 599–618 (2011).

[15] "CUDA Web Site," (2014), http://www.nvidia.com/object/cuda_home_new.html.

[16] "Thrust Web Site," (2014), https://developer.nvidia.com/Thrust.

[17] E. D. Brooks, "The MPISM Shared Memory Library For Applications That Use MPI," *unpublished Lawrence Livermore National Laboratory Report* (2014).

[18] B. Beck et al., "MCAPM-C Generator and Collision Routine (Gen2000Bang2000) Documentation," *unpublished Lawrence Livermore National Laboratory Report* (2014).

[19] C. M. Mattoon et al., "Generalized Nuclear Data: A New Structure (with Supporting Infrastructure) for Handling Nuclear Data," *Nuclear Data Sheets*, **113**, pp. 3145–3171 (2012).

[20] M. S. McKinley and B. R. Beck, "Implementation of the Generalized Interaction Data Interface (GIDI) in the Mercury Monte Carlo Code," *Proceedings of ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, April 19-23, 2015, (2015).