M&C 2017 - International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea, April 16-20, 2017, on USB (2017)

LLNL Monte Carlo Transport Research Efforts for Advanced Computing Architectures

Patrick S. Brantley, Ryan C. Bleile, Shawn A. Dawson, N. A. Gentile, M. Scott McKinley, Matthew J. O'Brien, Michael M. Pozulp, David F. Richards, David E. Stevens, Jonathan A. Walsh, Hank Childs*

Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551
**Department of Computer and Information Science, University of Oregon, Eugene, OR 97403*
{brantley1, bleile1, dawson6, gentile1, mckinley9, obrien20, pozulp1, richards12, stevens9, walsh23}@llnl.gov, hank@oregon.edu

Abstract - *This paper presents ongoing research efforts at Lawrence Livermore National Laboratory to enable the Mercury Monte Carlo particle transport code to run efficiently on current and upcoming advanced computing architectures. We briefly describe the Quicksilver proxy application that we have developed to enable more rapid prototyping of new algorithms and to engage the external computer vendor communities. We present research and development efforts with Quicksilver and Mercury focused toward the Trinity machine at Los Alamos National Laboratory that uses both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing many integrated core processors. Finally, we describe research into Monte Carlo event-based and history-based algorithms for the Lawrence Livermore National Laboratory Sierra machine that will use IBM Power processors along with Nvidia Volta graphics processing unit architecture accelerators.*

I. INTRODUCTION

Power consumption considerations are driving future high performance computing platforms toward advanced computing architectures. The computing landscape for the advanced architecture machines being procured through the National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) Program is depicted in Fig. 1. The current Sequoia machine at Lawrence Livermore National Laboratory (LLNL) is an IBM machine with 16 PowerPC A2 1.6 GHz cores per node (with four hardware threads per core). The Trinity machine that arrived at Los Alamos National Laboratory (LANL) starting in 2015 uses both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing (KNL) many integrated core (MIC) architecture processors. The Sierra machine started arriving at LLNL in late 2016 and will use IBM Power processors along with Nvidia Volta graphics processing unit (GPU) architecture accelerators. As a result of these different advanced architectures, the computing landscape for the upcoming years is complex and includes heterogeneity both within and across computing platforms. Simultaneously supporting efficient versions of codes for both the current generation and advanced computing architectures within a single source code base is a significant challenge.

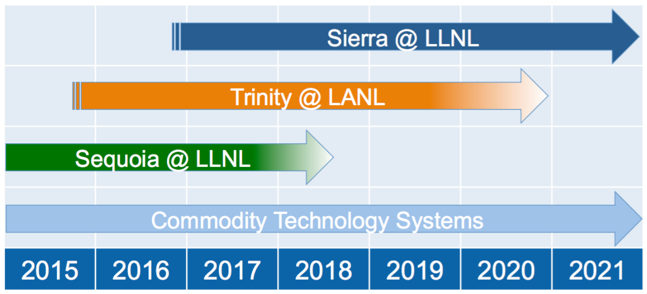


Fig. 1. Computing platform time line

Mercury is a Monte Carlo particle transport code under development at LLNL [1]. Mercury can transport neutrons, photons, and light element (hydrogen and helium) charged particles. Both fixed source and criticality problems are treated. Mercury is parallelized via domain decomposition and domain replication [2] with load balancing [3, 4, 5] and uses MPI parallelism across compute nodes and (optionally) OpenMP threading on-node. Mercury is written in C++ with a python user interface and runs efficiently on the current generation of massively parallel computing platforms.

Mercury is required to run efficiently on the advanced architecture machines being procured through the NNSA ASC Program. We have several Monte Carlo transport research efforts underway to help prepare for this diversity of computing architectures. We previously reported on parallel scaling efforts targeted at the Sequoia machine [6]. More recently, we have developed an open source proxy application for the Mercury Monte Carlo particle transport code [1] called Quicksilver [7] that we are using in engagements with computer vendors through the Trinity and Sierra Centers of Excellence. To prepare for Trinity, we are working to characterize and improve the efficiency of the MPI + OpenMP parallel implementation in Quicksilver and Mercury for the Intel Knights Landing architecture. We have also integrated the Scalable Checkpoint/Restart Library [8] into Mercury to access the DataWarp burst buffer technology available on Trinity. To prepare for Sierra, we are researching event-based Monte Carlo transport algorithms for use on GPU architectures and are also investigating the use of a big-kernel history-based Monte Carlo transport approach for GPU architectures.

The remainder of this paper is organized as follows. We briefly describe the Quicksilver proxy application in Section II. We then describe research focused toward the Trinity machine in Section III. We next describe research focused toward the Sierra machine in Section IV. We provide conclusions and ideas for future work in Section V.

II. QUICKSILVER PROXY APPLICATION

A proxy application [9] is a simplification of characteristics of a real application (algorithms, data structures, memory layout, parallelism, etc.) into a smaller, distributable code base that can serve as proxy for the larger application. We have developed an initial version of an open source proxy application for the Mercury Monte Carlo particle transport code [1] called Quicksilver [7] to enable more rapid prototyping of new algorithms and to engage the external computer vendor communities through the Trinity and Sierra Centers of Excellence. Quicksilver solves a time-dependent fixed source particle transport problem with significantly simplified and approximate physics, artificial multigroup cross sections, and a simple mesh-based geometric representation. Quicksilver uses both MPI and OpenMP parallelism, is written in C++, and is approximately 7,000 lines of source code. Because Quicksilver serves as a proxy for Mercury, similarity in computational performance, parallel scaling, etc. is desirable. We have performed initial studies investigating the correspondence of the performance of Quicksilver to Mercury, although more work is required to fully assess how accurately Quicksilver represents the performance of Mercury as a proxy application. We are currently using Quicksilver in collaborations through the Trinity and Sierra Centers of Excellence and in early investigations of the high-level big-kernel history-based GPU approach.

III. TRINITY-RELATED RESEARCH

The Trinity machine at LANL uses both Intel Xeon Haswell E5-2670 2.6 GHz processors and Intel Xeon Phi Knights Landing 7250 many integrated core architecture processors. Each Knights Landing processor has 68 1.4 GHz cores with four hyperthreads per core. Optimal use of the Intel Xeon Phi MIC processors requires the ability to use threads and the ability to use vector units. Vectorization is challenging for history-based Monte Carlo transport. Mercury has a hybrid MPI+OpenMP parallel model in which cores on a node can be tasked using either MPI processes, OpenMP threads, or a combination of the two. Our initial research approach for enabling Mercury to run on the Trinity machine is to focus on leveraging and improving the existing OpenMP threading capability. As resources and priorities dictate, we may investigate improved vectorization approaches in the future.

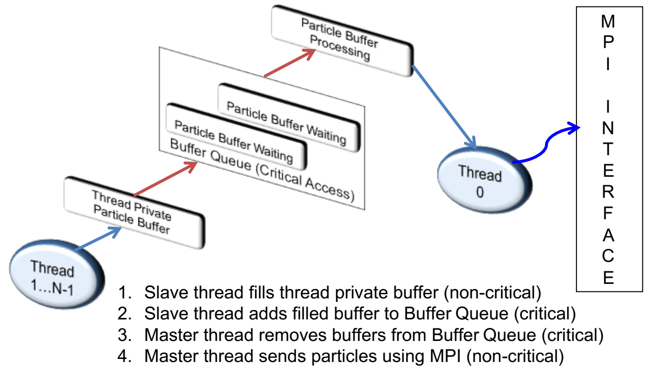
1. MPI + OpenMP Threading Research

Mercury can use a combination of both MPI and OpenMP to distribute the Monte Carlo work across the cores of a compute 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 [2]. 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. 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. We previously reported [6] on the threading algorithm and implementation used in Mercury and provided performance results for a reactor eigenvalue calculation.

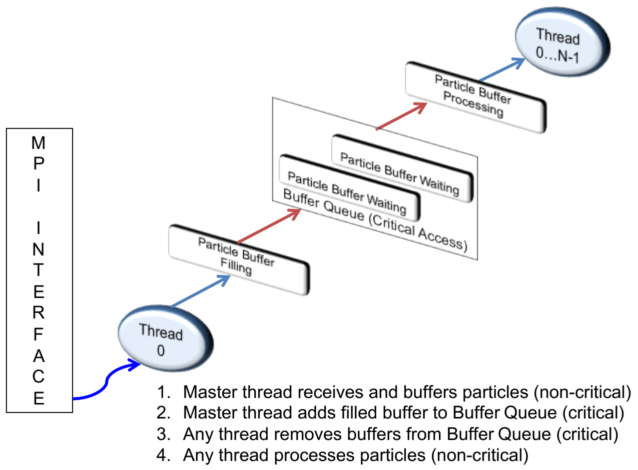
We are currently researching improved MPI + OpenMP domain-decomposition approaches for the KNL architecture using the Quicksilver proxy application. In the current MPI "thread funneled" (TF) [10] particle sending and receiving algorithms shown in Figs. 2(a) and 2(b), respectively, only thread zero performs MPI communication, requiring that threads add particles from a per-thread queue to a thread-shared work buffer queue for MPI communication to another domain. This TF approach requires 1) a work buffer queue for each MPI process that is shared by all threads of the process, requiring a critical section when particles are removed by thread zero or added by other threads, 2) a choice regarding the frequency with which thread zero checks for particles from other threads that need to be communicated, and 3) thread zero interruptions to communicate the particles.

The new MPI "thread multiple" (TM) [10] approach enables direct thread-to-thread MPI communicators as shown in Fig. 3 and is much simpler than the thread funneled approach. As a result, each thread communicates directly with its corresponding thread on other MPI processes. This TM approach bypasses critical sections in work queues and thread zero interruptions.

We studied the performance of the TF and TM approaches using a Quicksilver homogeneous medium test problem with 80 million Monte Carlo particles per time step and ten time steps. This study was performed on Intel KNL B0 (early delivery) Xeon Phi 7210 hardware with 64 1.30 GHz cores and four hyperthreads per core. The simulations used 4 MPI processes and either 16 (one thread per core), 32 (two threads per core), or 64 (four threads per core) OpenMP threads per MPI process. The wall time performance results are shown in Fig. 4. The TF results exhibit non-monotonic performance: the cycle time decreases slightly going from 16 threads (one thread per core) to 32 threads (two threads per core) but then significantly *increases* when the number of threads increases to 64 (four threads per core). We attribute this degradation in efficiency to the thread zero critical sections and interruptions at these large thread counts. The TM approach exhibits monotonically decreasing cycle time with increasing numbers of OpenMP threads. The MPI thread multiple approach with 64 threads (four hardware threads per core) provides nearly a factor of two speedup over the best MPI thread funneled performance at higher thread counts. Based on these results, we are currently implementing and testing the thread multiple approach in Mercury.


Particle Buffer Processing
Particle Buffer Waiting
Particle Buffer Waiting
Buffer Queue (Critical Access)
Thread Private Particle Buffer
Thread 1…N-1
Thread 0
MPI INTERFACE
1. Slave thread fills thread private buffer (non-critical)
2. Slave thread adds filled buffer to Buffer Queue (critical)
3. Master thread removes buffers from Buffer Queue (critical)
4. Master thread sends particles using MPI (non-critical)

(a) Particle sending algorithm


Thread 0…N-1
Particle Buffer Processing
Particle Buffer Waiting
Particle Buffer Waiting
Buffer Queue (Critical Access)
Particle Buffer Filling
Thread 0
MPI INTERFACE
1. Master thread receives and buffers particles (non-critical)
2. Master thread adds filled buffer to Buffer Queue (critical)
3. Any thread removes buffers from Buffer Queue (critical)
4. Any thread processes particles (non-critical)

(b) Particle receiving algorithm

Fig. 2. MPI thread funneled algorithm

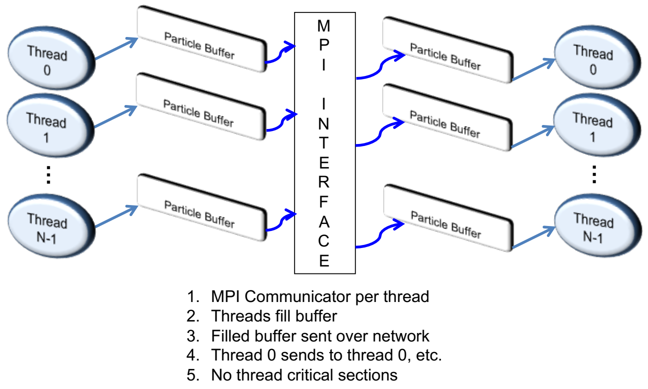

Thread 0
Thread 1
Thread N-1
Particle Buffer
MPI INTERFACE
1. MPI Communicator per thread
2. Threads fill buffer
3. Filled buffer sent over network
4. Thread 0 sends to thread 0, etc.
5. No thread critical sections

Fig. 3. MPI thread multiple particle sending and receiving algorithm

We also studied Quicksilver run time using the MPI thread multiple approach as a function of the number of MPI processes and OpenMP threads on the Intel KNL B0 hardware. For this hardware, 64 cores with 4 hyperthreads per core results in 256 potential MPI processes and/or OpenMP threads.

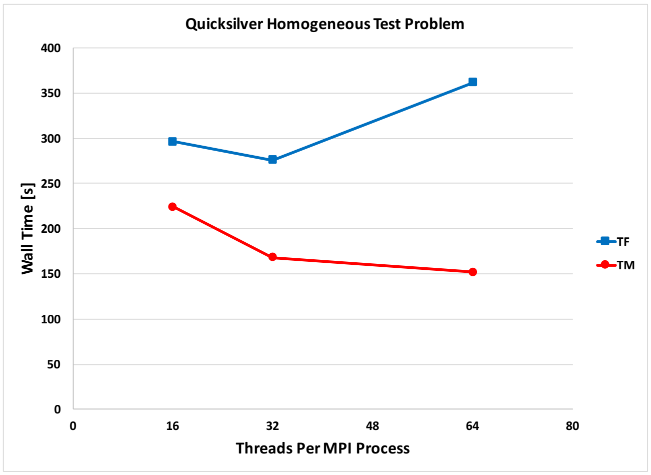


Fig. 4. Quicksilver KNL performance results for thread funneled and thread multiple algorithms

We used the Quicksilver homogeneous medium test problem with 100 million Monte Carlo particles per time step and ten time steps. We varied the number of MPI processes, N_{mpi}, and the number of OpenMP threads per MPI process, $N_{threads}$, keeping $N_{mpi} \times N_{threads} = 256$ constant. Fig. 5 shows the run times obtained in the study. The run time for all variations is the same to within 7%, demonstrating similar performance for all combinations of numbers of MPI processes and OpenMP threads. The 16 MPI × 16 threads and 32 MPI × 8 threads configurations are the most efficient. We had anticipated needing larger numbers of threads on the KNL architecture. However, the amount of memory available on the KNL architecture may limit the number of MPI processes if memory is replicated across processes.

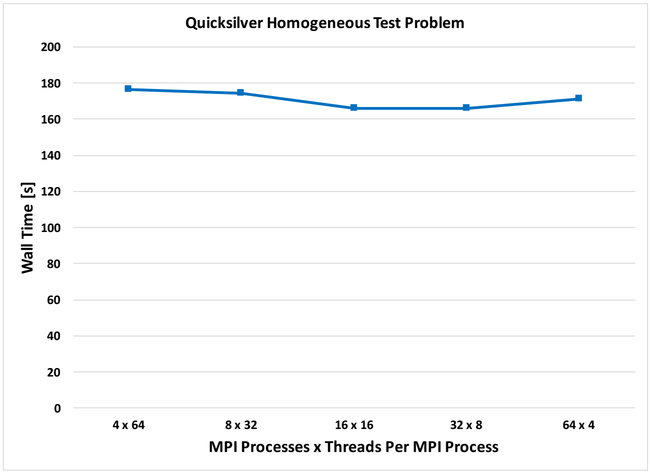


Fig. 5. Quicksilver KNL performance results varying number of MPI processes and number of threads

2. Scalable I/O Research

The Mercury code is using the Scalable Checkpoint/Restart (SCR) library [8] to leverage the storage hierarchy on Trinity for I/O speedup during checkpoint and restart. To date, we have only had access to the Trinity on-node RAM disks (not yet the DataWarp burst buffer technology) to obtain I/O performance results [11]. While the use of the DataWarp burst buffers is a major motivation for using the SCR library in Mercury, the current use of the RAM disks allows us to continue the development, integration, and testing of the SCR library regardless of the burst buffer availability. For a weak scaling test problem, we compared the time for writing checkpoints (shown in Fig. 6) and reading restarts (shown in Fig. 7) when using either the RAM disk or the Lustre parallel file system, scaling up from 1 node (32 processors) to 4,096 nodes (131,072 processors) in powers of two. Checkpointing to RAM disk instead of Lustre produced speedups at 16 nodes (512 processors) and above, including a 30X maximum output speedup. Reading restarts from RAM disk instead of Lustre produced speedups for all node counts but one, including a 9X maximum input speedup.

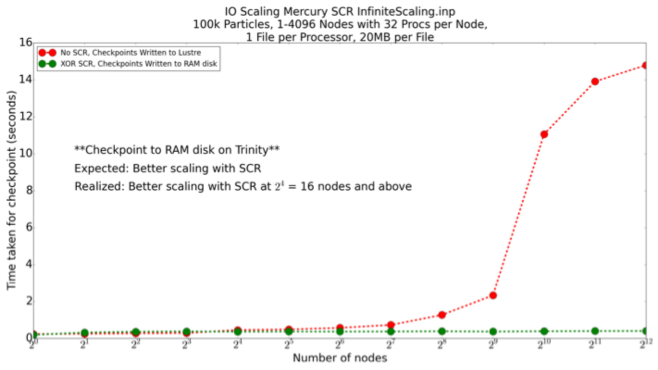


Fig. 6. Mercury SCR checkpoint time results

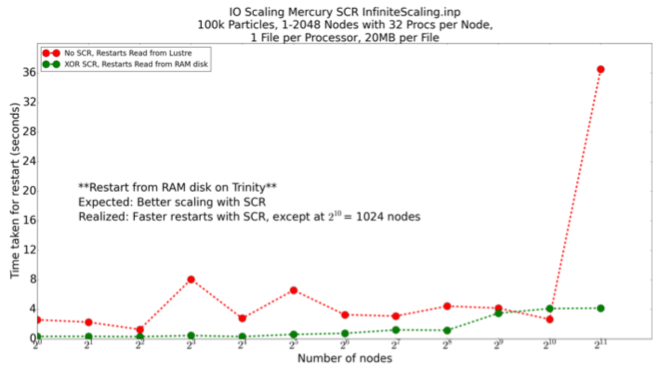


Fig. 7. Mercury SCR restart time results

IV. SIERRA-RELATED RESEARCH

We are researching both history- and event-based Monte Carlo particle transport algorithms for GPU architectures in preparation for the LLNL Sierra machine. Previous researchers [12, 13, 14] have noted that the use of an event-based Monte Carlo particle transport algorithm [15] may be beneficial for GPU architectures. The motivation for that idea is that processing particles undergoing the same event avoids branch statements that may introduce divergence in the GPU calculation. Recent work performed by Scudiero [16, 17] suggests that it may be possible to achieve performance on GPUs using a history-based Monte Carlo transport algorithm. Additionally, since Monte Carlo transport is a memory latency bound problem, using a less compute-optimized approach may be acceptable. Transforming a large production Monte Carlo transport code to use an event-based algorithm is also a significant undertaking. As a result, we are also investigating a "big-kernel" history-based approach in which the entire particle tracking function is treated as a single big GPU kernel. This big-kernel approach is our preferred alternative to event-based algorithms assuming it produces acceptable performance on GPU architectures. We have identified performant and thread safe particle work queues, thread safe tally incrementing, and inter-node communication as three main research issues for Sierra.

We have also researched the Nvidia C++ Thrust library [18] as a possible portable programming abstraction. Thrust is a portable library that compiles into multiple language backends (serial C++, CUDA, OpenMP, etc.) and provides data containers such as a host/device vector.

1. ALPS Test Code Research

Our initial research efforts for the Sierra machine involved investigations with the ALPS Monte Carlo test code [19] that models particle transport in a one-dimensional planar geometry binary stochastic medium. The ALPS code was originally implemented using a standard history-based Monte Carlo algorithm, as shown in Alg. 1. We implemented an event-based version of the ALPS Monte Carlo algorithm, shown in Alg. 2, for GPUs using a data parallel approach and the Nvidia Thrust library [18] as a portability abstraction [20, 21]. We also implemented an event-based version of ALPS explicitly with CUDA [22] as a comparison with the Thrust version. The use of CUDA enables more fine grained control at the kernel level and enables access to different memory spaces such as GPU shared memory.

A. Thrust

Thrust is a C++ header library using an STL-like template interface [23]. Thrust provides a number of parallel algorithms and data structures designed to provide access to GPU computing without needing to write explicit CUDA code [22]. Additionally, Thrust provides backend capabilities allowing these algorithms and data structures to target different devices, including CPUs with OpenMP threads. This approach was used for studying portable performance techniques with Thrust, providing a method of maintaining a single source code base that can be used on multiple platforms.

Thrust algorithms are used for implementing data parallel procedures across all particles in a batch. These algorithms perform operations such as the data parallel *map*, *reduce*, *gather*, *scatter*, or *scan* operations defined in [24]. Each of these

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
```

Algorithm 2: Event-based Monte Carlo algorithm

```
1  foreach batch of particle histories (fits in memory
   constraint) 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         remove particles absorbed or leaked
```

operations can be performed in a data parallel way.

Thrust also provides data types that can be used to manage memory for GPU devices. The thrust::device_vector and thrust::host_vector data structures operate similarly to a C++ std::vector but with automatic memory copying between host and devices whenever necessary. These data types allow for simple memory management schemes that work on both GPU and CPU based architectures.

B. Data Parallel Event-Based Algorithm Detail

An event-based algorithm focuses on performing data parallel operations across all particles undergoing the same event. Additional overhead is needed to find the grouping of particles that will be operated on and to determine an access pattern for the particles. This reorganization stage can be costly and is not directly related to solving the transport problem.

Thrust provides permutation iterators that allow for the unaligned access of data elements according to an index map. Using this iterator scheme, data elements do not need to be copied into new locations for each operation. This approach comes at the cost of performing non-contiguous memory accesses for reading and writing the information.

In order to perform an event operation on particles using this scheme, a series of data parallel operations is used to establish the correct index mapping for the permutation iterator. This scheme is defined as follows and describes in detail lines six and seven of Alg. 2:

Step 1: thrust::transform — Fill out a stencil map of 1's and 0's of all particles doing event E (where each particle whose next event is E will get a 1 in the stencil map at its index location)

Step 2: thrust::reduce — Count the number of elements labeled 1 in the stencil (determines the number of particles that will perform event E)

Step 3: Check if the number of elements is greater than 0 (check if any particles are performing event E)

Step 4: thrust::exclusive_scan — generate indices for index mapping from stencil map (indices for each particle performing event E)

Step 5: Allocate a new map of appropriate size (map to hold indices for all particles performing event E)

Step 6: Scatter indexes from scan into new index map (reduces the exclusive_scan generated indices into the map that holds only enough for particles performing event E)

Step 7: Use new index map in permutation_iterator loops over all particles (combining the index map with the permutation iterator allows loops over all particles to operate only on the particles selected in the index map)

We implemented the event-based version of ALPS using both the Nvidia CUDA programming model [22] explicitly and the Nvidia C++ Thrust library [23]. The Thrust implementation of ALPS utilizes data parallel operations and Thrust data types for managing memory. The same Thrust event-based implementation can be compiled with either CUDA for use on GPUs or OpenMP for use on CPUs, enabling portability to different platforms. In the explicit CUDA implementation of ALPS, we found it useful to continue to use Thrust algorithms in building various maps. ALPS is implemented using double precision floating point numbers throughout. The Thrust and CUDA implementations of ALPS give physics results identical to the original history-based implementation.

The CUDA implementations for this study matched the algorithm in the Thrust implementations. The differences in performance come from the capabilities that native CUDA programming provide that cannot be accomplished with Thrust. Using CUDA directly enables more fine-grained control at the kernel level and enables important access to different memory spaces such as GPU shared memory. The CUDA implementation includes a scheduling algorithm to optimize the number of active threads on the GPU for each kernel call. Additionally, the CUDA implementation includes the use of the different available memory spaces, such as constant and shared memory. For example, Monte Carlo particles were initially allocated in GPU global memory and then copied to shared memory for all operations within a kernel. All problem constants such as cross sections and mean chord length values were placed in GPU constant memory. These optimizations under certain conditions can have a significant impact on the performance of a GPU kernel.

Finally, we previously found [21] that a struct-of-arrays (SOA) data structure for storing and accessing Monte Carlo particle data was more efficient than an array-of-structs (AOS) data structure on the GPU. We also observed [21] that an optimized scheme for removing inactive particles (line twelve of Alg. 2) achieved a significant performance improvement. We chose to perform the particle remove operation if the number of inactive particles to be removed is at least half the size of the particles in the list. As a result, the maximum number of times we perform the expensive removal operation becomes log(n), where n is the size of the list. The numerical results below incorporate both of these optimizations.

C. Big Kernel History-Based GPU Implementation

Recent work performed by Scudiero [16, 17] suggests that it may be possible to achieve acceptable performance for Monte Carlo particle transport on GPUs using a history-based algorithm if the correct transformations are made. Since Monte Carlo transport is a memory latency bound problem, using a less compute-optimized approach may be acceptable from a performance standpoint. We refer to this history-based approach as a "big-kernel" approach in which the entire particle tracking function is treated as a single big GPU kernel. We have implemented a "big-kernel" history-based GPU version of ALPS [21] using CUDA by making the following transformations. First, we moved those calculations that only needed to be performed once for all particles out of the single large kernel. Second, we utilized shared memory for storing the particle data structure and read-only constant memory for storing the material data (e.g., cross section values). Finally, we removed all atomic tally updates and replaced them with a shared per particle tally that is reduced to single values after the kernel is complete.

D. Numerical Comparisons

We investigated the performance of these approaches on the LLNL Rzhasgpu computer that has Intel Xeon Haswell 3.2 GHz host cores with Nvidia Tesla K80 GPU device accelerators. Speedups relative to a history-based *serial* CPU calculation with the ALPS Monte Carlo test code using native CUDA implementations as well as using the Thrust implementation on both a GPU and with sixteen OpenMP threads on the CPU are shown in Table I [21].

The ALPS explicit CUDA event-based implementation produces a speedup of approximately 31 over the serial CPU code. The explicit CUDA "big-kernel" implementation of the Monte Carlo history-based algorithm demonstrates a larger speedup than the event-based implementation. The event-based algorithm requires overhead to identify particles that are undergoing the same event, and this overhead reduces the efficiency of the calculation. The Thrust event-based version running on the GPU is slightly more efficient than the explicit CUDA event-based implementation. We expect that this result is due to two possible factors. First, the Thrust scheduler may be launching kernels more effectively than the kernel launching scheme we implemented. Second, the memory locations of the read-only tallies and written tallies are stored with the Thrust functor which may allow Thrust to optimize what memory exists in registers or caches when the kernel launches.

TABLE I. Maximum Speedups Compared to the Original History-Based Serial Algorithm

Algorithm	Speedup
CUDA Event-based	31.3
CUDA History-based (big-kernel)	52.8
Thrust CUDA Event-based	54.6
Thrust OpenMP Event-based	5.5

We would ideally expect a speedup of sixteen for the OpenMP version when using sixteen threads. The Thrust OpenMP event-based results are significantly less than optimal with a speedup of less than six. So while Thrust provides a portable abstraction, the OpenMP results for the event-based implementation exhibit a significant reduction from optimal performance. While the event-based Thrust OpenMP results are significantly less than optimal, they do demonstrate portability and some performance gain.

2. Quicksilver-Lite Research

We have recently tested the big-kernel history-based approach in Quicksilver-Lite, a proxy application of intermediate complexity between ALPS and Quicksilver. The goal for Quicksilver-Lite is that it would be easy to port to the GPU but would maintain enough memory access and function call complexity to challenge the hardware. Quicksilver-Lite consists of simplified code (no MPI, etc.) along with simplified particles and mesh. Quicksilver-Lite is, essentially, a thin layer on top of nuclear data lookups. We implemented an explicit CUDA version of the big-kernel history-based approach in Quicksilver-Lite. We compared the performance of the big-kernel version running on half of an Nvidia Tesla K80 GPU device accelerator to an OpenMP threaded version running on one node with sixteen IBM BGQ 1.6 GHz cores. The big-kernel GPU version achieved a speedup of a factor of 1.64 compared to the threaded version running on sixteen cores. Increased performance of the big-kernel version should be possible using both GPUs on the K80. Based on these encouraging results, we are currently extending the big-kernel history-based approach to Quicksilver.

V. CONCLUSIONS

The LLNL Monte Carlo Transport Project team is working to enable efficient use of the upcoming advanced computing architectures. The Quicksilver proxy application is proving useful for algorithmic investigations and computer vendor engagements. For the Trinity machine, we have found that enabling thread-to-thread MPI communication (MPI thread multiple) improves efficiency for domain-decomposed problems on the KNL processor. In addition, higher numbers of MPI processes appears efficient for Monte Carlo transport on the KNL architecture. Finally, the SCR library is a promising approach for leveraging the storage hierarchy on Trinity for I/O speedup. For the Sierra machine, the big-kernel history-based approach implemented in both the ALPS Monte Carlo test code and the Quicksilver-Lite proxy application appears promising for GPU architectures and is our preferred path forward.

Future work will include further testing of the MPI + OpenMP thread multiple implementation in Mercury. In addition, we are implementing the big-kernel GPU approach in the Quicksilver proxy application and are performing additional high-level architectural investigations. We are investigating both OpenMP4.5 and explicit CUDA big-kernel implementations.

VI. ACKNOWLEDGMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Funding for R.C.B. was provided by the LLNL Livermore Graduate Scholar Program.

REFERENCES

1. P. S. BRANTLEY, R. C. BLEILE, S. A. DAWSON, M. S. MCKINLEY, M. J. O'BRIEN, M. POZULP, R. J. PROCASSINI, D. RICHARDS, S. M. SEPKE, and D. E. STEVENS, "Mercury User Guide: Version 5.2," *Lawrence Livermore National Laboratory Report LLNL-SM-560687 (Modification #10)* (2016).
2. 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," in "Proceedings of International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)," Saratoga Springs, New York (May 3-7, 2009 (2009)).
3. R. J. PROCASSINI, M. J. O'BRIEN, and J. M. TAYLOR, "Load Balancing of Parallel Monte Carlo Transport Applications," in "Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications," Avignon, France (September 12-15, 2005 (2005)).
4. M. J. O'BRIEN, P. S. BRANTLEY, and K. I. JOY, "Scalable Load Balancing for Massively Parallel Distributed Monte Carlo Particle Transport," in "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)).
5. M. J. O'BRIEN, *Scalable Domain Decomposed Monte Carlo Particle Transport*, Ph.D. Dissertation, University of California, Davis (2014).
6. P. S. BRANTLEY, S. A. DAWSON, M. S. MCKINLEY, M. J. O'BRIEN, D. E. STEVENS, B. R. BECK, and E. D. BROOKS III, "Advanced Computing Architecture Challenges for the Mercury Monte Carlo Particle Transport Project," in "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).
7. D. RICHARDS, P. BRANTLEY, S. MCKINLEY, and M. O'BRIEN, "Quicksilver: A Mini-App for Mercury," *Lawrence Livermore National Laboratory Report LLNL-SM-668536* (2016).
8. A. MOODY, G. BRONEVETSKY, K. MOHROR, and B. R. DE SUPINSKI, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," *Supercomputing 2010, New Orleans, LA* (2010).
9. M. HEROUX, R. NEELY, and S. SWAMINARAYAN, "ASC Co-design Proxy App Strategy," *LA-UR-13-20460/LLNL-TR-592878* (2013).
10. *MPI: A Message-Passing Interface Standard Version 3.0*, Message Passing Interface Forum (2012), available at http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.
11. M. M. POZULP, G. B. BECKER, P. S. BRANTLEY, S. A. DAWSON, K. MOHROR, A. T. MOODY, and M. J. O'BRIEN, "Optimizing Application I/O by Leveraging the Storage Hierarchy Using the Scalable Checkpoint Restart Library with a Monte Carlo Particle Transport Application on the Trinity Advanced Computing System," *Supercomputing 2016, November 13-18, 2016, Salt Lake City, Utah* (2016).
12. A. G. NELSON, *Monte Carlo Methods for Neutron Transport on Graphics Processing Units Using CUDA*, M.S. Thesis, The Pennsylvania State University (2009).
13. 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," in "Proceedings of Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC)," Paris France (October 27-31, 2013 (2013)).
14. 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**, 176–193 (2015).
15. F. B. BROWN and W. R. MARTIN, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, **14**, 269–299 (1984).
16. A. SCUDIERO, "Monte Carlo Neutron Transport: Simulating Nuclear Reactions One Neutron at a Time," in "GPU Technology Conference," San Jose, California (May

2014).
17. A. SCUDIERO, "personal communication," (2016).
18. "Thrust Web Site, https://developer.nvidia.com/Thrust," (2016).
19. P. S. BRANTLEY, "A Benchmark Comparison of Monte Carlo Particle Transport Algorithms for Binary Stochastic Mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, **112**, 599–618 (2011).
20. R. C. BLEILE, P. S. BRANTLEY, S. A. DAWSON, M. J. O'BRIEN, and H. CHILDS, "Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library," *Trans. Am. Nucl. Soc.*, **114**, 369–372 (2016), on USB.
21. C. BLEILE, P. S. BRANTLEY, M. J. O'BRIEN, and H. CHILDS, "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library," *Trans. Am. Nucl. Soc.*, **115**, 535–538 (2016), on USB.
22. "CUDA Web Site," (2014), http://www.nvidia.com/object/cuda_home_new.html.
23. "Thrust Web Site," (2014), https://developer.nvidia.com/Thrust.
24. G. E. BLELLOCH, *Vector Models for Data-Parallel Computing*, vol. 356, MIT press Cambridge (1990).