LA-UR-10-05488

Approved for public release; distribution is unlimited.

Title:

Towards Scalable Parallelism in Monte Carlo Particle Transport Codes Using Remote Memory Access

Author(s):

Paul K. Romano, Benoit Forget, Forrest B. Brown

Intended for:

SNA + MC - 2010 Conference
October 17-21, 2010
Tokyo, Japan

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Form 836 (7/06)

Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2010 (SNA + MC2010)
Hitotsubashi Memorial Hall, Tokyo, Japan, October 17-21, 2010

Towards Scalable Parallelism in Monte Carlo Particle Transport Codes Using Remote Memory Access

Paul K. ROMANO [1], Benoit FORGET [1], and Forrest BROWN [2]

[1] Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA
[2] Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545-1663, USA

One forthcoming challenge in the area of high-performance computing is having the ability to run large-scale problems while coping with less memory per compute node. In this work, we investigate a novel data decomposition method that would allow Monte Carlo transport calculations to be performed on systems with limited memory per compute node. In this method, each compute node remotely retrieves a small set of geometry and cross-section data as needed and remotely accumulates local tallies when crossing the boundary of the local spatial domain. Initial results demonstrate that while the method does allow large problems to be run in a memory-limited environment, achieving scalability may be difficult due to inefficiencies in the current implementation of RMA operations.

KEYWORDS: Monte Carlo, neutron transport, parallelism, MPI-2, remote memory access, one-sided communication

I. Introduction

In recent years, there has been an increasing interest in using Monte Carlo methods to solve the neutron transport equation not only for validation purposes but also as a routine design tool. This shift has necessitated higher levels of complexity and detail in the geometric models being used and, consequently, higher demand on computing resources. At the same time, the advancement of high-performance technical computing has allowed researchers to begin studying large-scale problems that would have previously been difficult, if not impossible, to solve using Monte Carlo methods.

Notwithstanding the benefits of the Monte Carlo method, the fact remains that simulations using Monte Carlo codes may take considerably longer to run than their deterministic counterparts, especially when determining local quantities such as reaction rates. As a result, Monte Carlo codes are routinely run in parallel on a workstation, cluster, or supercomputer. Monte Carlo simulations are inherently parallel since each stochastic realization of a neutron being tracked through phase space is completely independent of all other realizations. This is especially true for fixed source problems where no synchronization is necessary. For criticality problems, care must be taken to ensure that the source distribution and eigenvalue converge before tallying quantities of interest and to ensure reproducibility.

However, the ability to run a Monte Carlo code in parallel depends not only on the nature of the simulation but also on the computer architecture it is being run on. In order for most Monte Carlo codes to run in parallel, each process must have access to all the geometry and cross-section data. This means that the problem data are replicated and made available to each processor, thus imposing a constraint on the amount of system memory required. For example, if the geometry and cross-section data consume 4 GB of memory and the user desires to run on 8 processors, the total memory requirement will be at least 32 GB. For high-fidelity problems such as a reactor full-core model with hundreds or thousands of depletion zones, it may not be possible to use all the processors on a single node[1] if doing so would exceed the amount of memory available on that node.

II. Methodology

II.1. History and Motivation

Over the course of the last 40 years, microprocessor architecture and manufacturing processes have improved dramatically as evidenced by the persistent trend of increasing transistor density, larger die sizes, and increasing processor clock frequency. The clock frequency is often used as a measure of the performance of a processor. Thus, the performance of processors has also improved over time due to higher clock frequencies which allow for more operations per second.

However, in the last half decade, we have seen only modest increases in clock frequencies. Nowadays, major improvements in performance are achieved by other means such as pipelining, instruction-level parallelism, multiple functional units, and most importantly multi-core and many-core processors.

The advent of multi-core chips has forced a paradigm shift in the programming world, with increased emphasis on threaded calculations (e.g., using OpenMP or pthreads) on a

[1] By “node” here, we mean a typical symmetric multiprocessing (SMP) machine whereby multiple processors or cores are connected to a single shared memory.

single SMP node, with message-passing (e.g., OpenMPI, MPICH2) between different nodes. Having a greater number of cores per processor will only benefit true performance if programmers take advantage of the parallelism inherent in the architecture.

The above considerations lead one to conclude that in the near future, symmetric multiprocessing nodes will likely have hundreds of processor cores all sharing memory. The memory available on each node to be shared among cores will likely show only modest growth, leading to possibly less memory per core. In turn, the memory required by Monte Caro simulation will grow over time due to both the increasing fidelity of problems being simulated as well as a higher number of processors on a single shared memory node. As a result, there will be a need for new methods that cope with limited memory in Monte Carlo (and other) simulations.

II.2. Data Decomposition

In order to run a problem which would exceed the memory of a single node, the typical approach is to decompose the spatial domain and follow particles in a single domain on one compute node, moving particles between domains as needed. While this may work well for problems where the particle distribution is nearly uniform, non-uniformity may lead to poor load balancing, possibly even zero speedup due to idle processors and increasing communication. The inherent parallelism in Monte Carlo is over particles, not over spatial domains.

In this paper, we look at an alternative scheme for parallelization. In this scheme, the problem data are still spatially decomposed, but need not reside locally on the compute nodes. Rather than assigning particles to compute nodes based on their spatial coordinates, parallelism on particles is achieved by having each compute node retrieve geometry and cross-section data from other nodes only as needed. This approach is reminiscent of schemes used in the early days of computing, where much data was stored in extended memory devices (e.g., LCM, ECS, SSD) or disk storage and fetched into memory as needed. This approach ensures that each compute node follows the same number of particles and thus performs approximately the same amount of work.

The mapping of the particle and data processes onto the computing nodes is very flexible in the proposed data decomposition method. Since one-sided communication allows any one node to fetch data from another node as needed, the geometry and cross-section data can be stored anywhere in memory. The particle and data processes could thus be mapped to the same or different computing nodes. If needed, data nodes could be replicated to prevent contention between two nodes trying to access the same data.

II.3. Network Communication

The inherent limitation of the proposed method is greater communication as opposed to greater memory. Although message-passing would normally make such a scheme highly inefficient, the introduction of remote data access features in MPI-2 (one-sided remote puts and gets) could make it an effective approach for solving problems too large to fit in the memory of a single node. It is instructive at this point to review the basics of one- and two-sided communication.

II.3.a. Remote Memory Access

The Message Passing Interface (MPI) has become the de-facto standard API for programming parallel computers. The MPI-1 standard[1)] was entirely based on what's known as "two-sided communication". In this model, two processes can send data to one another, but in doing so, both processes must explicitly know that the communication is taking place. Thus, one process issues a command to send data from a buffer in its memory while the other process issues a command to receive data into a buffer in its memory.

The MPI-2 standard[2)] introduced a set of features called one-sided communication. This mode of communication allows a process to remotely access the memory of another process without explicitly issuing a call to an MPI routine, hence the alternate name remote memory access (RMA). In order to do so, the originating process must specify all the communication parameters. The target process may not even know what buffer in memory was accessed or which remote process accessed its memory.

There are three basic RMA communication calls: remote reads (MPI_GET), remote writes (MPI_PUT), and remote updates (MPI_ACCUMULATE). The operation performed during the remote update could be adding a value to a remote buffer, multiplying it by a value, or one of a number of other options.

II.3.b. Algorithm

In the data decomposition algorithm, we subdivide the domain of the problem over a number of nodes as illustrated in **Figure 1**. Each colored region corresponds to a subdomain that is stored on one node. When a particle hits the boundary of the subdomain, rather than moving the particle to another processor as in the domain decomposition method, the node remotely retrieves the geometry for the region which it is about to enter and accumulates any tallies for the previous region it was in. One also needs to accumulate tallies whenever a particle leaks out of the geometry or is absorbed by a material.

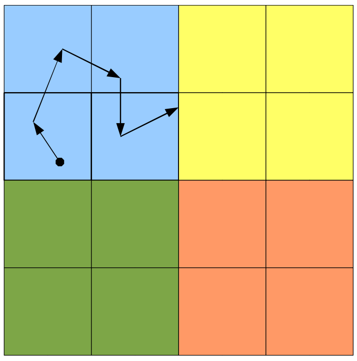
Figure 1: Decomposition of geometry into several subdomains

While the basis of the method is simple enough, the devil is in the details. There are many variables at hand that make actually implementing this method in an efficient way a difficult problem. For instance, one should consider how large the subdomains should be, how each node will know where to retrieve data from and accumulate tallies to, when should problem data be replicated, etc. Rather than deal with all the complexities at hand immediately, in this paper we implement the method in a simple Monte Carlo code to gain some basic insights into the potential performance of this algorithm.

II.4. Performance of MPI Implementations

Before we begin looking at the performance of the data decomposition method itself, it is instructive to first look at the performance of the underlying remote memory access routines in various MPI implementations since the method relies heavily on RMA functionality. While there are many MPI implementations in existence, the two major implementations with large developer communities and active development are OpenMPI and MPICH2. Both of these implementations fully conform to the MPI-2 standard.

To test the performance of one-sided communications, we looked at the average time it took to complete a remote read and a remote update for OpenMPI 1.4 and MPICH2-1.2.1. **Figure 2** shows how the average time varies with the number of nodes simultaneously making calls to MPI_GET. We note that each process calling MPI_GET was remotely getting data from a distinct node so that there should be no data contention.

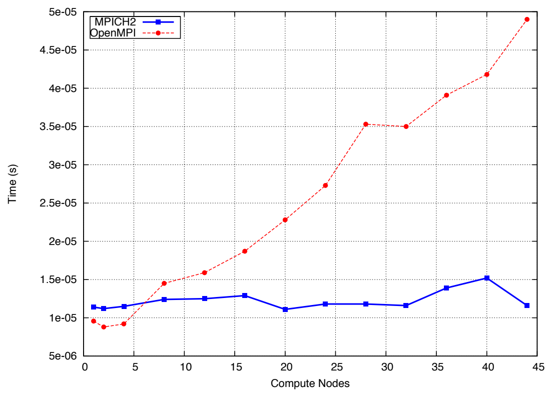

Figure 2: Average time to perform MPI_GET for MPI implementations

Figure 3 shows that remote updates perform similar to remote reads. It is clear from these figures that using OpenMPI in its current release will result in non-scalable code since the average time to perform an RMA operation increases with the number of nodes. Thus, we have elected to use MPICH2 for any testing and performance studies.

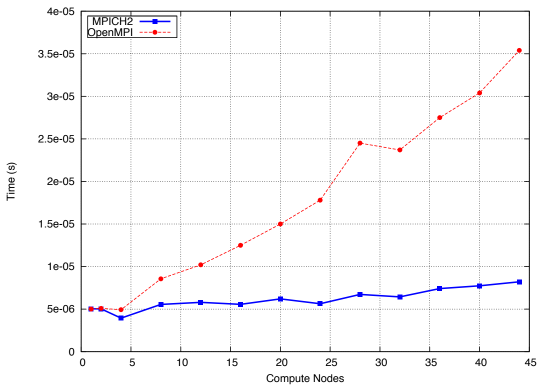

Figure 3: Average time to perform MPI_ACCUMULATE for MPI implementations

III. Performance Studies

III.1. Description of Simple Code

As a first step, a simple Monte Carlo criticality code was written in Fortran 90 and parallelized using point-to-point and collective communications. The MPICH2 implementation was used to provide capabilities for message-passing and remote memory access. To make matters simple, only one energy group is used and scattering is assumed to be isotropic in the lab coordinate system. No variance reduction techniques are employed, so a particle's weight does not change throughout its history.

The geometry is divided into a three-dimensional structured rectangular mesh, and each mesh cell is assigned a material number. The outer edges of the geometry can be assigned either vacuum or reflective boundary conditions. The method of successive generations is employed to converge on a stable fission source distribution and eigenvalue.

III.1.a. Retrieval of Problem Data

In our implementation, regions are specified with a derived data type named *region*. This data type has three attributes: the (x,y,z) coordinates of the lower-left corner of the region, the (x,y,z) coordinates of the upper-right corner of the region, and the material number assigned to the region. Thus, the entire geometry consists of one dynamically allocated array of type *region*.

The algorithm for dynamic retrieval of geometry data by the slave processes works as follows. At the beginning of a particle's life as well as each time a boundary is crossed, a call is made to a routine *updateRegion* provided the particle hasn't leaked out of the geometry or hit a reflective boundary condition. The *updateRegion* routine checks whether the particle has hit the edge of its local spatial domain, and if so, it makes a call to MPI_GET to fetch a new block of *region* data that includes the region it is about to enter. To facilitate this process, each particle is assigned two triplets (i,j,k), one indicating what global region the particle is currently in and one indicating what local region the particle is currently in.

The accumulation of local tallies onto the data nodes is similar in principle to the retrieval of data. The size of the tally arrays on the compute and data nodes is the same as the size of the geometry arrays.

III.2. Worst-Case Scenario

Let us now look at the performance of the simple scheme outlined above to see what improvements and refinements, if any, are necessary to make. We begin by looking at the worst-case scenario whereby there is no data decomposition and no data replication across nodes, i.e. all the problem data and tally structures sit on the master process.

The worst-case scenario is represented by the diagram in **Figure 4**. Here we see that the master process and the sole data node are one and the same. This will result in non-scalability for a number of reasons. Firstly, as the number of compute nodes increases, there will be increasing contention for retrieving data and accumulating tallies on the master process. Another possible source of contention will be simultaneous requests for work by one compute node and for data by another compute node. This source of contention would, in theory, be obviated if the network interconnect supports remote direct memory access (RDMA) wherein the RMA operations do not interfere at all with computation.

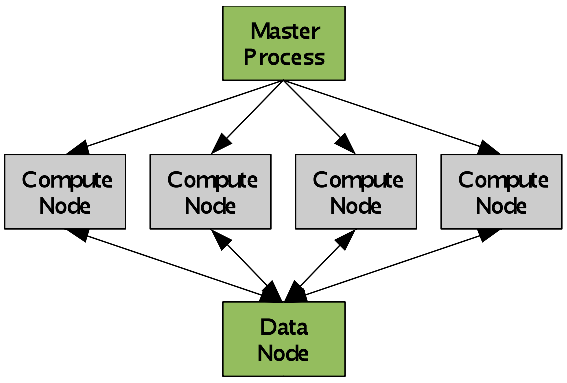

Figure 4: Physical depiction of worst-case scenario

By having all the problem data on one node, we are guaranteed to have the maximum amount of contention for data since every compute node will query the master process for data whenever it needs to get a new geometry region or accumulate tallies.

There are two manners in which we can test the performance of this scheme. The first, and more common, method is to run the problem with varying number of processors for a fixed amount of work, e.g. 100,000 histories per cycle. This is known as strong scaling. The other method is to increase the amount of work proportionally to the number of processors so that each processor has the same amount of work to perform regardless of how many processors are being run on. This ensures that the ratio between time spent in computation and time spent in communication should stay equal, and thus, the problem should scale. This is known as weak scaling.

Based on the above considerations, it should be no surprise that this scheme shows very poor performance with an increasing number of processors. **Figure 5** shows the speedup versus the number of compute nodes. Optimal performance is reached with only three compute nodes. Past this point, adding further compute nodes only slows down the overall run due to data contention at the master process. Each run consisted of 20 cycles with 10,000 histories per compute node per cycle (weak scaling). These runs were performed on the Kilkenny cluster at MIT, a commodity Linux cluster with a Gigabit Ethernet network interconnect.

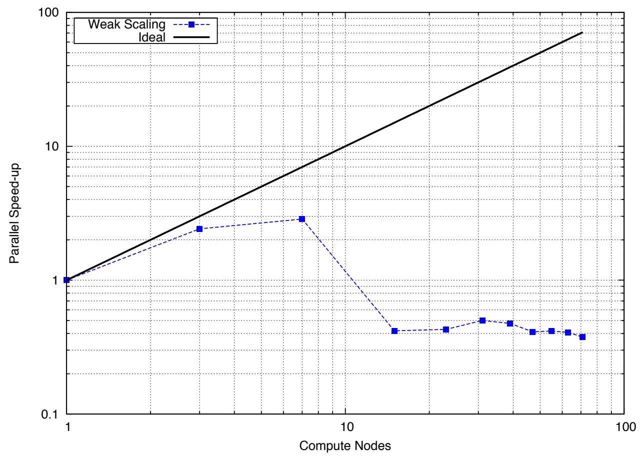

Figure 5: Speed-up for worst-case scenario

III.3. Best-Case Scenario

The next scenario we look at is one in which each compute node has a dedicated data node that has all the problem data fully replicated. In this scenario, there should be no data contention since no two compute nodes will be making requests for data on the same data node. The diagram of this scenario shown in **Figure 6** illustrates that each compute node has its associated data node on the same physical node. This will minimize the communication latency since accessing memory on the same physical node should be faster than accessing memory on a different node.

For the worst-case scenario, it was sufficient to see that even with weak scaling, the scheme was non-scalable. However, for the best-case scenario, it is instructive to look at both weak and strong scaling. Runs with up to 40 compute nodes and 40 data nodes were performed, again using the Kilkenny cluster. For the weak scaling cases, 10,000 histories per compute node were used. For the strong scaling cases, 400,000 total histories were used.

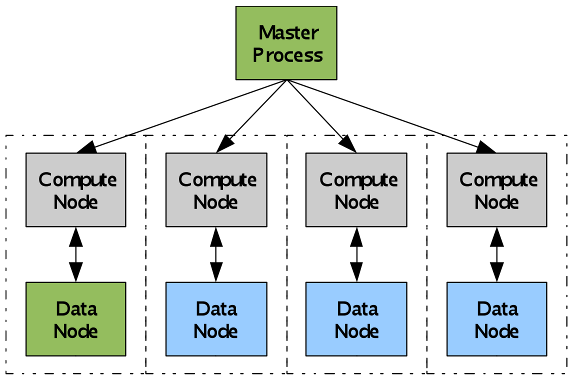

Figure 6: Physical depiction of best-case scenario

With no contention for data on each of the data nodes, the speed-up becomes nearly linear. **Figure 7** shows the speedup versus the number of compute nodes.

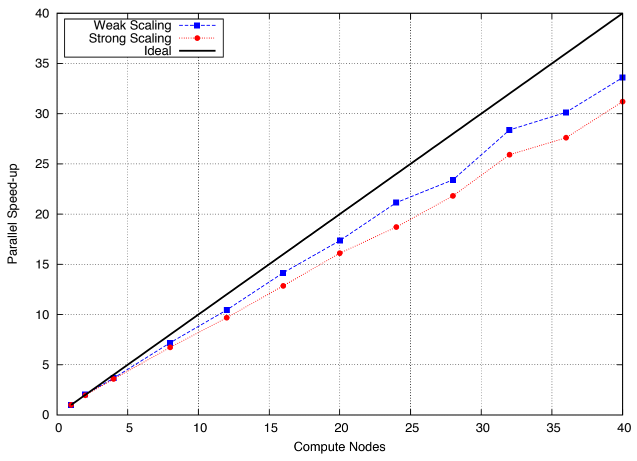

Figure 7: Speed-up for best-case scenario

We can see in Figure 7 that weak scaling performs slightly better than strong scaling as one might expect from intuition. The results here are encouraging given that the runs were performed on commodity hardware. On a supercomputer with a fast network interconnect (e.g., InfiniBand), the results should be even better than shown here.

III.4. Effect of Locality of Data on Communication Costs

In our best-case scenario above, we conjectured that by having the data stored on the same physical node as the compute node, the communication latency would be minimized. To actually quantify the effect of the locality of the data on the communication latency, two cases were run to see how performance fared when having all the data local to the compute nodes as well as the opposite. These cases are similar to our best-case scenario above in that each compute node has a dedicated data node.

In the first case, illustrated in **Figure 8**, each compute node has its corresponding data node on a different physical node. The second case, illustrated in **Figure 9**, is the same as our best-case scenario above where each compute node has its corresponding data node on the same physical node. The first case will presumably be slower due to a higher communication latency associated with performing RMA operations on separate physical nodes. Each of these cases was run on 16 processes, eight compute nodes and eight data nodes (including the master process), for 20 cycles with 100,000 histories per cycle.

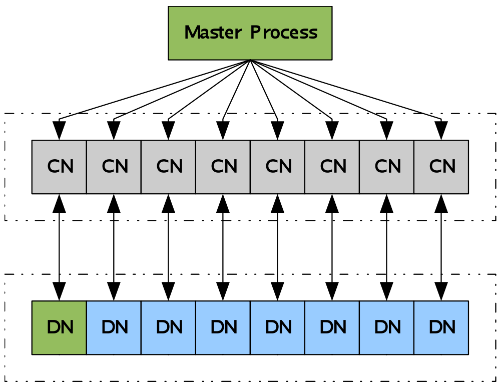

Figure 8: Physical depiction of non-local data nodes

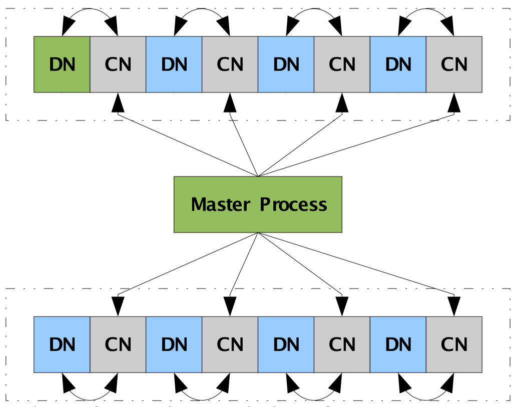

Figure 9: Physical depiction of local data nodes

The first case with non-local data ran in 87.04 seconds whereas the second case with local data ran in 19.98 seconds. Thus, we see that performing RMA operations on non-local data incurs a factor of about four penalty in execution time versus performing RMA operations on local data due to the higher communication latency. This will have important implications when considering how best to decompose data over several nodes since it is desirable to have RMA operations performed on local nodes.

III.5. Effect of Local Mesh Size on Communication Costs

The size of the mesh data being remotely retrieved or accumulated may have a drastic effect on the communication cost. On one hand, having a larger local spatial domain will result in fewer RMA operations since a particle will cross the boundary of the local spatial domain with less frequency. On the other hand, a larger local spatial domain implies that the cost of performing a single RMA operation will be higher since there is more data to transfer.

As a first step, we seek to quantify these two opposing effects by looking at how much data is transferred each cycle as the size of the local mesh. For this purpose, we looked at a 2D problem with 18x18 mesh cells. The local mesh size was varied from 1x1 up to 18x18 (all square meshes). This problem was run with 10,000 histories per cycle for 100 cycles. The number of MPI_GET calls was tallied each cycle and then averaged over the cycles. In the 18x18 case, the entire geometry is available after a single MPI_GET, so the number of MPI_GET calls will simply be equal to the number of histories per cycle. **Figure 10** shows the number of MPI_GET calls per cycle as a function of the local mesh size.

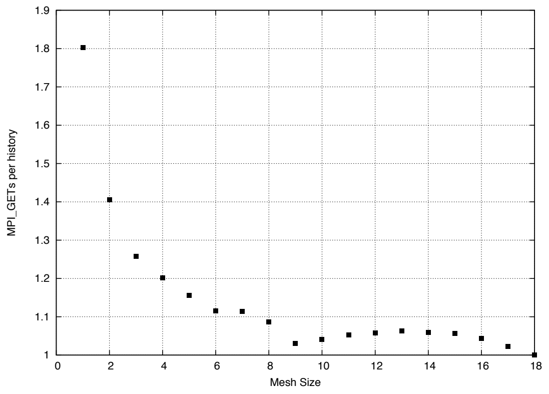

Figure 10: Remote memory accesses as a function of local mesh size

We see from Figure 10 that using a local mesh size larger than about 4x4 for this particular problem yields a negligible decrease in the number of remote memory accesses. As the mesh size increases, the amount of data being transferred with each MPI_GET goes up quadratically (since it is a 2D problem). This will result in monotonically increasing data transfer with respect to the local mesh size[2]. Thus, for this problem, a small local mesh size (2x2 or 3x3) will result in optimal run times. We note that these results will be strongly dependent on the physical nature of the problem, i.e. whether it is strongly scattering or strongly absorbing.

This simple characterization of the effect of the local mesh size on the communication cost is not sufficient to draw any major conclusions. We recommend that further studies be performed, particularly looking at how the physical nature of the problem affects data transfer requirements. In addition to this, another important line of inquiry will be to look at the time required for a single RMA operation as a function of the amount of data being transferred.

IV. Conclusions

Although this approach may end up being a suboptimal solution at the present time, given the aforementioned trends it will become increasingly viable in the near future. Proposed architectures for exaflop systems postulate millions or 100s of millions of processor cores, with reduced memory per core. Effective use of such systems for large-scale Monte Carlo calculations will require extremely large numbers of independent computational threads as well as the use of remote data access. Advances in network interconnects have significantly improved the ability to transfer large amounts of data with low latency and high bandwidth (e.g. InfiniBand, 10 Gigabit Ethernet), and these advances are likely to continue.

Our work to date has focused on basic demonstration and characterization of algorithms based on a "particle parallelism plus data decomposition" approach. We have investigated best- and worst-case bounds on performance, and are encouraged. Much future work is needed to investigate such aspects as: the optimal mapping of compute nodes and data nodes to a given computer system architecture; the ability to dynamically monitor remote memory accesses and adjust the compute/data node mappings; prefetching of remote data to minimize compute delays; performance scaling for 1000s and 100s of thousands of processor cores. In addition, some further changes in Monte Carlo algorithms may also be required, such as: the use of batch statistics, rather than history-based statistics; new iterative methods for k_{eff} eigenvalue calculations, such as concurrent calculation of multiple problems, each with particle parallelism and data decomposition; improved random number generators with longer periods and efficient skip-ahead schemes; increased use of on-the-fly computing methods rather than using precomputed data tables (e.g., for Doppler broadening of cross-section data).

V. Acknowledgment

This work was funded partly by the Los Alamos National Laboratory and an NRC Graduate Education Fellowship.

VI. References

1) Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 1.3," University of Tennessee, Knoxville, 2008.
2) Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.1," University of Tennessee, Knoxville, 2008.
3) MPICH2 – A high-performance and highly portable implementation of the Message Passing Interface standard (both MPI-1 and MPI-2), URL: http://www.mcs.anl.gov/research/projects/mpich2
4) R. Graham, T. Woodal, J. Squyres, "Open MPI: A Flexible High Performance MPI," *Proceedings of PPAM 2005*, 228-239 (2005).

[2] Note that although going from a 1x1 local mesh to a 2x2 local mesh yields a 22% decrease in the number of RMA operations, at the same time the amount of data being transferred with each RMA operation increases four-fold