Project Report: A Comparison of Parallel Domain Copy and Decomposition for a 1D Monte Carlo Transport Code

Simon Bolding
CSCE 626

May 5, 2015

References

1. People in class - Daniel Holladay

2. wikipedia.org/wiki/Prefix_sum,computing.llnl.gov, mpitutorial.com

3. EOS website: sc.tamu.edu/systems/eos

4. Exploring Monte Carlo Methods, by Shultis and Faw, 2011

1 Introduction

1.1 Summary

In this work experimental analysis was performed to evaluate the performance of two different approaches to parallelizing a Monte Carlo particle transport code: domain copy and domain decomposition. In a domain copy algorithm, each processor has a copy of the entire physical domain of the problem. This is opposed to domain decomposition in which each processor only contains a portion of the domain. Each has their own benefits and difficulties. An illustration of the two strategies is given in Fig. 1 The algorithms were implemented in a simplified version of a research code that simulates pure-absorber, one-dimensional particle transport using the Monte Carlo method [Shultis and Faw, 2011]. Speed up in the full code is not of particular interest to this project. The goal is to explore the algorithms and have a working simplified model that can be used to test future algorithms.

In general, domain copy is a much simpler and efficient parallelization strategy. However, for methods that require additional information to be stored and communicated more often, it can become unfeasible. In particular, the full solution method used by this research code will require too much information to be communicated efficiently in higher spatial dimensions. Thus, it is of interest explore the basics of the domain decomposition strategy; it does not require tallied information to be communicated at the end of simulations, but it does require more communication during the simulation.

In this work, the two algorithms used are described in detail. Expected theoretical complexities are discussed. Strong and weak scaling studies were performed to compare the performance of each of the algorithms. The tests were performed for two problems which are representative of the range of potential problem types. The algorithms are benchmarked against the original sequential algorithm. As expected, the domain copy algorithm showed good scalability for this simple 1D problem, but the domain decomposition algorithm showed good scaling in optically thick problems.

1.2 Monte Carlo Transport Basics

Monte Carlos solves particle transport problem by tracking random particle histories throughout a domain and estimating the particle density by tallying information. The average of all histories produces the expected value of the tally of interest. The original code was designed to simulate time-dependent, thermal radiative transfer problems using Residual Monte Carlo and a deterministic acceleration method. The simplified version of the code only models the

Monte Carlo portion of the code for a single steady state solve. It uses Monte Carlo to perform a single batch of histories to simulate a transport problem for a fixed distribution of source particles. This simplification was necessary because the codes data structures were not built to be easily decomposed or communicated. By simplifying the code, the algorithms could still be realistically tested, without the large unnecessary overhead of parallelizing the entire code.

The physical domain of the problem is represented with a uniform space-angle finite-element mesh: one dimension representing location x in 1D space and one dimension representing the angle, or direction, of particles μ. The problem is broken up into space-angle elements (or cells) over which information is tallied. The only material property of interest is the removal cross section σ which represents the average probability a particle interacts per differential unit length. Because of the formulation of the Residual Monte Carlo code, no scattering occurs in the Monte Carlo solver (this is representative of the real code). Thus, particles that undergo interaction are simply absorbed, and thus removed from the simulation. For all problems herein there is a single material cross section in the domain to simplify analysis.

The solution of interest is the steady-state distribution of particles in the system. This distribution is represented by a cell-wise linear representation, in space and angle, of the particle density referred to as the **angular flux** $\psi(x, \mu)$. Each cell i has a linear representation denoted as $\psi_i(x, \mu)$. The union of all elements, with a proper spatial closure, results in a piece-wise continuous representation of the solution. The final result that is typically of interest is an angular integrated particle density, referred to as the **scalar flux** $\phi(x)$. Tallies are used to estimate the representation of $\psi_i(x, \mu)$ in each cell. These tallies estimate the representation of ψ based on the average density of particle pathlengths that traverse that cell over the entire simulation.

To solve a problem, the scalar flux is determined based on an input source distribution of particles. The source is represented over the mesh with a cell-wise linear distribution that can be easily sampled. To determine the scalar flux, the basic process defined in Alg. 1 is performed.

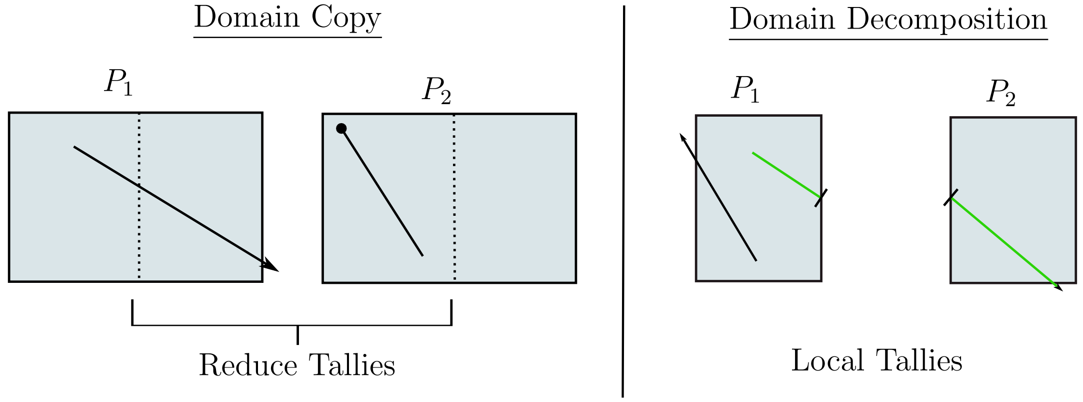


Figure 1: Illustration of domain copy (left) and domain decomposition (right) strategies on two processors.

Algorithm 1 Serial algorithm for solving a transport problem with Monte Carlo.

```
Input: $N$ histories, mesh, source distribution
  1. Compute source strength in each $x$-$\mu$ cell $i$
  2. For $N$ histories do:
       (a) Source random $x$ and $\mu$ from the specified source distribution
       (b) Sample how far the particle travels $x_0$ from the distribution $p(x_0) = \sigma e^{-\sigma x_0}$.
       (c) Track particle to location of interaction:
             • Tally the contribution to $\psi_i(x,\mu)$, the linear angular flux representation with
               each space-angle cell $i$ that the history traverses
       (d) Terminate particle history
  3. For each cell: average contribution of $N$ histories to all tallies
  4. Compute $\phi(x)$ by integrating $\psi(x,\mu)$ over $\mu$
return $\phi(x)$
```

2 Algorithms and Theoretical Analysis

2.1 Sequential Algorithm

The sequential algorithm is defined by Alg. 1. Tracking particle histories from birth to death can be an expensive process and is typically the dominating term in run times. Thus, the complexity for running time is expected to be dominated by NT_k, where T_k is the average time to simulate a single history k and N is the number of histories performed. T_k is a function of the problem size and parameters, but it is, on average, a constant for a particular problem. Thus, $T(N) = O(N)$.

2.2 Domain Copy Algorithm

There are two standard approaches to parallelizing Alg. 1: domain copy and decomposition. A diagram of the two approaches can be seen in Fig. 1. The first approach is domain copy, in which each processor gets a copy of the entire physical mesh, source, and tallies. Each processor simulates (N/p) histories, where p is the number of processors and N is the total number of histories to be simulated. Random numbers are generated using the counter-based Easy123 pseudo random number generator (RNG). Each processor uses its MPI rank as the key; this provides each processor a unique, independent set of random numbers, and thus independent histories. After all processors have completed their history tracking, the scalar flux representation is computed based on its tallies before communication is performed. On average, each processor's N/p histories take the same time because the particle histories are samples of the same underlying distributions. This limits the amount of asynchronization of processors before communication.

The representation of $\phi(x)$ is stored as a vector of spatial degrees of freedom per element.

An MPI_Allreduce is performed to compute the linear sum of all results from each processor and then copy the result to each of the processors. The sum must be divided by p to produce the correct average. Since each processor's N/p histories were independent, this is equivalent to running N histories with a single processor. The broadcasting of the result to all processors is done because in a real simulation each processor would need to know the scalar flux for the next portion of the algorithm. The domain copy algorithm is given in Alg. 2.

As discussed for the serial algorithm, the dominating term in the simulation run time is performing the histories. The MPI_Allreduce uses a parallel summing strategy which will be $O(\log n_x)$, where n_x is the number of elements in the problem. This communication term is relatively small and a constant for a particular problem size. Thus $T(N) = O(N/p + \log n_x) = O(N/p)$. At very large p, or a low number of N, the $O(\log n_x)$ term could become dominating. All processors are working for the entire simulation. Thus, neglecting the typically small communication cost, $W(N) = O(N/p * p) = O(N)$, which is work optimal. It is noted that because T_j is a function of problem size, there is no way to easily measure or study the size of $\log n_x$ with scaling studies.

Algorithm 2 MPI domain copy algorithm.

```
Input: $N$ histories, $p$ processors, mesh, source distribution
  1. For each processor $j$ pardo
      (a) Initialize my copy of the mesh
      (b) Compute source strength in each $x$-$\mu$ cell $i$
      (c) For $N/p$ histories do:
            i. Source random $x$ and $\mu$ from the specified source distribution
           ii. Sample how far the particle travels $x_0$ from the distribution $p(x_0) = \sigma e^{-\sigma x_0}$.
          iii. Track particle to location of interaction:
                 • Tally the contribution to $\psi_i(x, \mu)$, the linear angular flux representation
                   with each space-angle cell $i$ that the history traverses
           iv. Terminate particle history
      (d) For each cell: average contribution of $N$ histories to all tallies
      (e) Compute $\phi(x)$ by integrating $\psi(x, \mu)$ over $\mu$
  2. MPI_Allreduce with sum operation on $\phi(x)$ vector.
/* Each processor now has a copy of $\phi(x)$ */
```

2.3 Domain Decomposition Algorithm

2.3.1 Algorithm

In the decomposed case, each processor owns a portion of the space-angle mesh. A uniform decomposition is performed based on the spatial coordinate x. Thus each processor has a domain $\mathcal{D}_j$ consisting of all of the space-angle elements for some a particular region of space (the collection of all angle elements corresponding to a particular set of spatial elements). The domain is defined as $\mathcal{D}_j = \{x : x_l < x < x_r\}$, where x_l is the left edge of the region and x_r is the right edge of the region for processor j. Because the spatial mesh is one-dimensional, the decomposition is ordered in a linear processor array. The mesh is divided so that boundaries of processor domains correspond with spatial element boundaries. To simplify implementation, each processor has a copy of the entire spatial mesh but is only allowed to track over a certain set of spatial elements.

During tracking of the history, if a particle leaves the domain of that processor, it is stored in the appropriate "bank" (there is one each for particles leaving left and right). The bank is implemented as a growable vector of objects. These banks are then communicated to the appropriate adjacent processor as necessary. Communication is performed using explicit MPI send and receive commands. The size of the bank is communicated before the banks because the size of the array is variable. Bank contains particle structures that are transfered using `MPI_Type_struct` types. Each banked structure contains all of the necessary information to continue tracking the particle as if it had been on one processor (with the exception of a different set of random numbers).

For simplicity, the most naive approach to communicating throughout the simulation is performed. Each processor runs their initial batch of source particles, banking particles that have left the domain as necessary. Then, all processors send their banked particles to their left and right neighbors as appropriate. Each processor then tracks all of the received particles from their neighbors, including banking as necessary. The process of communicating banks and simulating the received particles is repeated until all particles have been absorbed or leaked from the system. The processors must synchronize at each step to ensure they receive the bank of particles before new ones are added. This communication approach can become very expensive and inefficient in problems where particles are likely to stream across the entire problem; some processors will be idle for long periods of time in intermediate steps, even though there is work they could be doing.

Once all particles have terminated, $\phi(x)$ is computed locally on each processor. There is no need to reduce the solution as all histories that have contributed to that portion of the domain were simulated by that particular processor. In implementation, a communication step is performed to verify correct output, but this is not included in timing results. The overall algorithm is outlined in Alg. 3.

2.3.2 Complexity

The complexity for this algorithm is less straightforward. To simplify the analysis, the following assumptions are made: uniform material properties, a uniform processor decomposition, and a uniform number of particles N/p are born in each processor (this would be the case for a uniform source distribution and processor decomposition). With these assumptions,

Algorithm 3 MPI domain decomposition algorithm.

```
Input: $N$ histories, $p$ processors, mesh, source distribution
  1. For each processor $j$ pardo
       (a) Initialize portion of the mesh for $P_j$
       (b) Compute number of histories $N_j$ in all $x$-$\mu$ cell belonging to $P_j$
       (c) For $N_j$ histories do:
             i. Source random $x$ and $\mu$ from the specified source distribution
            ii. Sample how far the particle travels $x_0$ from the distribution $p(x_0) = \sigma e^{-\sigma x_0}$.
           iii. Track particle to location of interaction or processor boundary:
                  • Tally the contribution to $\psi_i(x, \mu)$, the linear angular flux representation
                    with each space-angle cell $i$ that the history traverses
            iv. IF particle exited $P_j$'s domain:
                  • Bank particle in right or left bank
                  • continue
             v. Else: terminate history /* Leakage or absorption */
  2. local var: alldone := false, done := false, mylists
  3. While not alldone
       (a) IF size(rightbank) > 0:
             • Inform $P_{j+1}$ bank size to receive
             • Send(rightbank, destination=$P_{j+1}$)
       (b) IF size(leftbank) > 0:
             • Inform $P_{j-1}$ bank size to receive
             • Send(leftbank, destination=$P_{j-1}$)
       (c) IF recieving banks:
             • Recieve banks into one list of particles → mylist
       (d) Synchronization among processors /* Ensure all banks have been received */
       (e) For each particle in mylist
             • Execute steps 1(c)ii to 1(c)v on particle
       (f) IF (size(rightbank) = 0 & size(leftbank) = 0) done := true
       (g) Compute logical OR using MPI_Reduce on done's, stored in alldone
     end While /* If all banks are empty, then we are done */
  4. For each of $P_j$'s elements: average contribution of $N$ histories to all tallies
  5. Compute $\phi(x)$ for all elements in $P_j$'s domain
/* Each processor now knows $\phi(x)$ for its domain */
```

the time complexity can be defined as

$$T(N) = (N/p)T_j + \sum_{t=1}^{n_c} (T_j N_t + T_{c,t}) \,. \tag{1}$$

Here, n_c is the number of communication steps, $T_{c,t}$ is the time to communicate the largest bank at communication step t, T_j is the average time to track a history across the processor domain $\mathcal{D}_j$, and N_t is the size of the largest received bank of particles after communication step t. Because synchronization is enforced, the processors with the largest bank to simulate or communicate are the limiting factor. Because the same number of total histories N is simulated, the total amount of work to transport particles is the same as in the serial case. Thus, the work is $W(N) = O(N) + W_c$, where W_c is the total work spent on the communication steps.

It is difficult to reduce the time expression more, except for in the limiting cases. In the limiting worst case of an optically thin problem (near void), all particles can cross the domain yielding $n_c \simeq p$ and $N_t = O(N/p)$. The communication time will likely dominate this simulation. For the case of a very thick (particles are absorbed very quickly), then $n_c \simeq 1$, $N_t \simeq 0$, and $T(N) = O(N/P)$ and $W(N) = O(N)$, which is work and time optimal. Thus, it is expected in optically thick problems that this algorithm will scale as well as domain copy.

3 Experimental Setup

3.1 Machine Information

The parallel programs were tested on *eos*, a machine at Texas A&M. For all the results in this work, the available Intel "Nehalem" nodes were used. These processors use Intel 64-bit architecture. Each node contains two sockets, each with a chip containing 4 processing units, resulting in 8 processing units per core. There is some potential difference in memory access times on the chip when going from 4 to 8 cores, where memory must be accessed off chip. The interconnection of nodes is done using a "Fat Tree" topology. This results in a constant communication time to access any off board node from any other.

For memory, each core has 32 kB L1 cache and 256 kB of unified L2 cache. Each Nehalem chip (containing four cores) has an 8 MB shared L3 cache. There is $\sim$ 22 GB of shared RAM available to each node (i.e., 2 chips, or 8 cores). The RAM has non-uniform access time, with longer access times when a core accesses the DRAM that is located near the other chip on that node. More details about the architecture of *eos* can be found at `http://sc.tamu.edu/systems/eos/`

3.2 Problem Descriptions

To investigate the effect of the problem type, all experiments are performed for two different types of problems. These problems are chosen to show the limiting cases of performance for the domain decomposition algorithm. Both problems have a source (including scattering source) defined to produce a uniform distribution of particles throughout the domain. The problems will be referred to herein as the optically "thick" and optically "thin" problem.

The *more* optically thick the problem, the *less* likely a particle will traverse the entire problem. In general, thicker problems require less tracking across the mesh, resulting in shorter run times. For the case of domain copy, this has minimal effect on timing results. However, for domain decomposition this has a significant effect, as discussed in the theoretical analysis section. The very thin problem is chosen so that the majority of histories will traverse the entire problem. This means that approximately p communication steps, of very large banks, must be performed. It is expected that this will demonstrate the worst case scenario for the domain decomposition algorithm, relative to the domain copy algorithm. The optically thick problem is chosen so that, on average, only 1 communication step will occur, of a relatively small size. More communication steps may happen, but they will involve very few particles. There will always be at least 1 step required due to the fact that some particles will be born right next to a processor boundary. The specifics of the two problems are defined below:

- **Optically thin:** The total domain has width $T = 4.51$ cm and total removal cross section $\sigma = 0.02$. The probability a history traverses the entire domain before interacting (crossing all processors) is $e^{-\sigma T} = 90\%$.

- **Optically thick:** The total domain has width $T = 100cm$ and total removal cross section $\sigma = 6$. In the worst case of a 128 processor decomposition, the probability a particle crosses more than 1 processor is $e^{-\sigma T/128} < 1\%$

3.3 Description of Experiments

Batch files to run the various jobs were created using a Python script. Output files were processed with a Python script as well. The portions of code run times that is included for results of each algorithm are justfied and detailed in the sections describing each algorithm; they exclude any extra input, output, or initialization timing costs and only measure the cost of the Monte Carlo solver. Timing information was performed using built in C functions, such as `gettimeofday`.

3.3.1 Strong Scaling Study

The purpose of the strong scaling study is to see how much faster a problem of a fixed size can be solved by using more processors. Speed up was used as a performance measure for the strong scaling study. Speed up is defined as the ratio T_{ser}/T_p, where T_{ser} is the time to solve the problem using the most efficient serial algorithm and T_p is the time to solve the program using p processors. A program which scales perfectly would show a linear, one-to-one speed up. In general this is not the case due to communication and memory overhead.

3.3.2 Weak Scaling Study

A weak scaling study determines the efficiency of the algorithm as you increase the number of processors, while keeping the problem size *per core* fixed. The goal of a weak scaling study is to determine the increased cost of an algorithm as more processors are used to solve increasingly larger problems. This indicates how well an algorithm can be used to solve problems that may be too large to solve with a serial algorithm, or even at lower core counts. The metric for the weak scaling studies used was efficiency, defined as Efficiency $= T_p(n)/T_1(n) \times 100\%$, noting that T_1 is the time for the parallel algorithm with one processor, not the time for the serial algorithm. The ideal efficiency would be 100%, resulting in a flat line for Efficiency as a function of p. For the Monte Carlo algorithms described, decreases in efficiency likely indicate cost increase from communication times, relative to transport time.

3.4 Statistics

The experiments must be repeated to measure various forms of variability in the system, e.g., variable communication time, memory access times, inaccuracy of the timer, etc. Unless noted otherwise, the experiments were repeated 32 times for each plotted data point. The entire program is rerun for each iteration to ensure the effect of variability in memory initialization times on total execution time is represented accurately.

From all 32 repeated simulations, the reported run times are simply the average of the particular result from 32 simulations. The standard error in the average of a quantity is $\sigma/\sqrt{N}$, where σ is the sample standard deviation of the quantity from all 32 simulations. Since speed up and scalability are calculated quantities with a statistical variance in both terms, it is necessary to approximate the error in the quantity. Based on the standard error propagation formula (`http://en.wikipedia.org/wiki/Propagation_of_uncertainty`), the error for the ratio of two timing results T_i and T_j is

$$\sigma\left(\frac{T_i}{T_j}\right) = \frac{T_i}{T_j}\sqrt{\left(\frac{\sigma_{T_i}{}^2}{T_i}\right) + \left(\frac{\sigma_{T_j}{}^2}{T_j}\right)}. \tag{2}$$

The above equation is used to determine the standard error for all plotted speed ups and scalability. The plotted values are the 95% confidence interval. This confidence interval, assuming a Gaussian distribution of the error, is plotted as 1.96σ.

4 Experimental Results and Analysis.

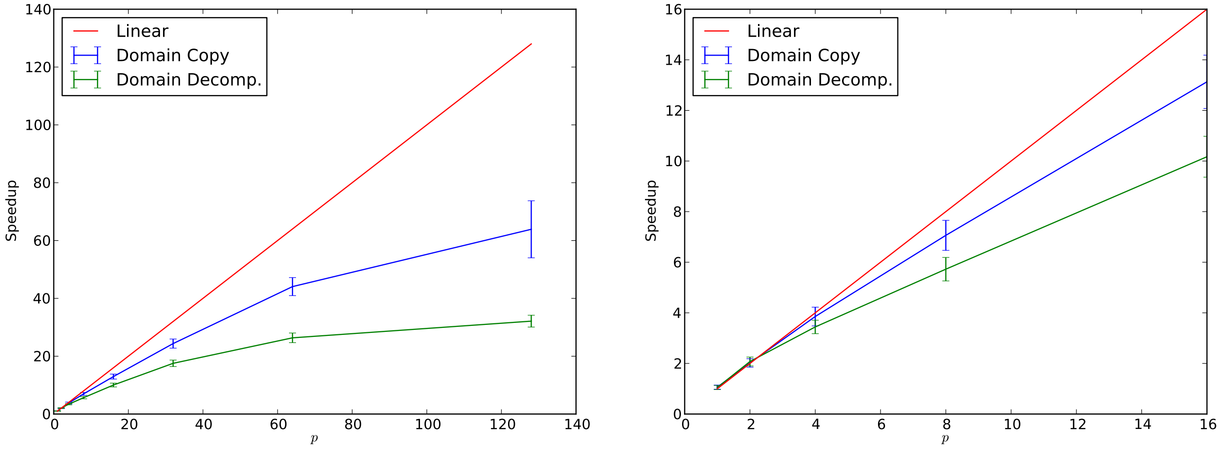
Below are given results and discussion for the two experiments. Only strong and weak scaling studies are performed for brevity. For the strong and weak scaling studies, the simulations use a mesh of 248 x cells × 6 μ cells. This mesh is chosen to have the tracking time of particles to be representative of a realistic problem, while still having simulation times that a single core can reasonably achieve. Also, this allows for the 128 processor simulation to have at least one spatial element per processor. For the optically thick simulations, more histories per processor are needed to increase simulation time because particle tracking takes much less time, per history, as discussed in the experimental setup.

4.1 Strong Scaling Study

The results for the speed up study for the optically thin and thick problem are given in Fig. 2 and Fig. 3. For the optically thin and thick problems $N = 5 \times 10^6$ and $N = 20 \times 10^6$ histories were performed in each simulation, respectively. These were chosen to produce reasonable run times for the sequential and largest parallel simulations.

For the optically thin problem, the domain copy algorithm demonstrates much better scaling than the decomposition algorithm, as expected. After starting out at near linear scaling, the speedup begins to decline for the domain copy algorithm. This is a result of the all-reduce communication step becoming a more dominant contribution to the time, relative to the $O(N/p)$ history simulation time. As a rough measure of this cost, for the case of $p = 64$, the communication time T_c is $\sim 15\%$ of the total run time. Thus, the complexity is more correctly represented as $O(N/p + T_c)$. The domain decomposition algorithm scales poorly for the thin problem due to the large number of communication steps and the size of the particle banks communicated at each step.

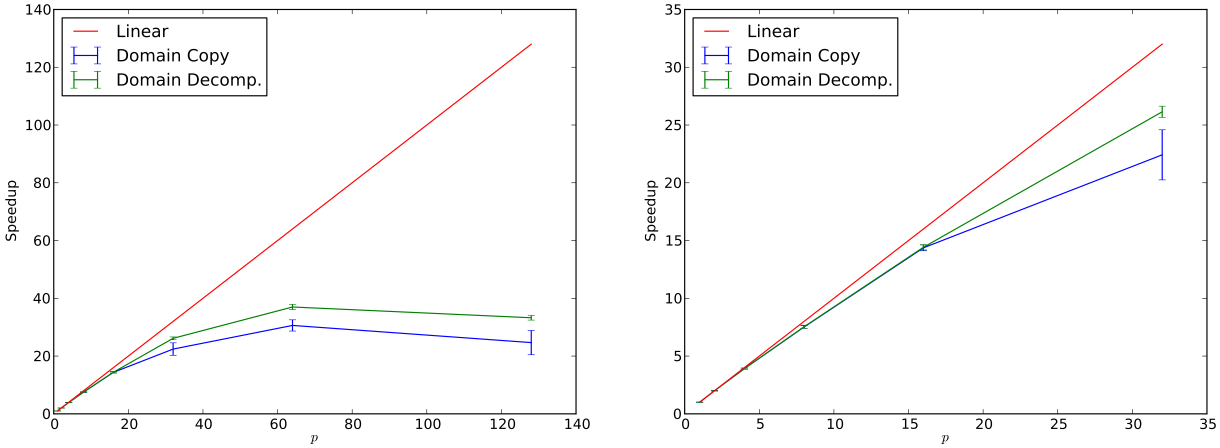
For the thick problem, at lower processor counts both algorithms show very good scaling. Because there is minimal particles to be communicated at boundaries, the domain decomposition algorithm becomes essentially embarrassingly parallel. Both methods do not scale well above 32 processors. This poor scaling has to do with the fact that in optically thick problems the tracking of histories is relatively fast (even though there are $20 \times 10^6/p$ histories to simulate). Above 32 processors the dominant term in the simulations is the communication and MPI synchronization overhead as before. For the case of 64 processors and the domain copy algorithm, the cost of communication is $\sim 20\%$ of the total simulation cost.



(a) Strong scaling to 128 processors.

(b) Zoomed view

Figure 2: For the optically thin problem, a plot of speedup versus number of processors p for $N = 5 \times 10^6$ histories.



(a) Strong scaling to 128 processors.

(b) Zoomed view

Figure 3: For the optically *thick* problem, a plot of speedup versus number of processors p for $N = 20 \times 10^6$ histories.

4.2 Weak Scaling Study

A plot of the weak scaling efficiency for various thread counts for the OpenMP and MPI algorithms are given below. In general, the weak scaling does not perform well for either algorithm. From 2-8 cores, the weak scaling efficiency are very similar for both the MPI and OpenMP algorithm. As noticed, the efficiency drops significantly from 2 to 8 cores. The large initial drops in the efficiency at low core counts are likely due to memory access times, as there is minimal communication cost at these low core counts. If the cause of the initial drop in efficiency at low core counts can be mitigated, it is likely the case that the total efficiency would scale much better because this issue likely is effecting the processors on all nodes locally.

For the MPI algorithm, the efficiency begins to level off at around 20% above 16 cores. It is noted however, that above ∼32 cores, increasing the problem size, per core, does not result in any loss in efficiency. Although the MPI algorithm is not exceptionally efficient, very large problems can be ran without much loss in computational time due to additional parallel communication costs.

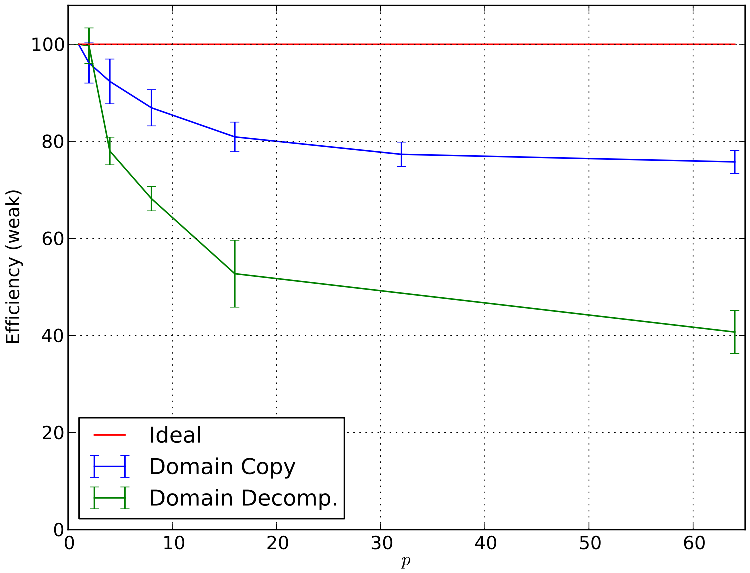


Figure 4: Weak scaling study for optically thin problem with $N = 250,000$ histories per processor

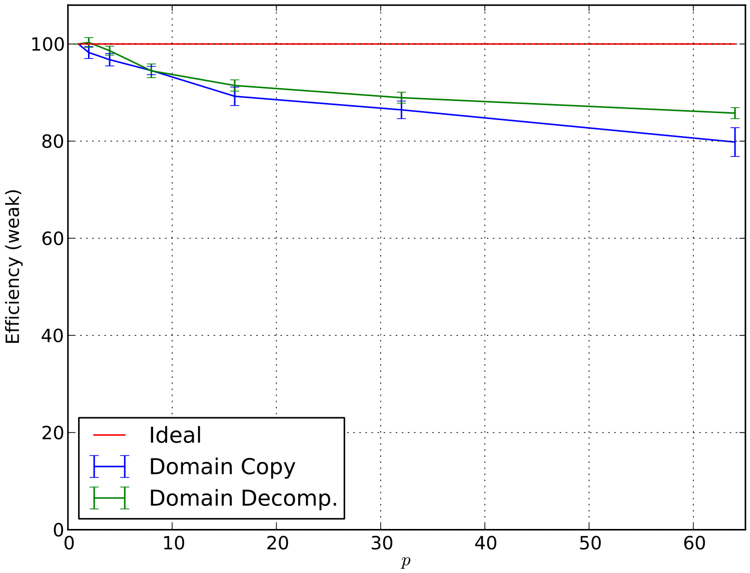


Figure 5: Weak scaling study for optically thick problem with $N = 2 \times 10^6$ histories per processor

4.3 Computational time versus problem size

Plots of computational time versus problem size, for various *fixed* p, are given below for each algorithm. It is expected that the dominant term in the time complexity for both algorithms is $O(n/p)$, so the computational times should scale linearly with n. This behavior was observed, demonstrated by the linear shape of the plotted run times. For the case of $p = 4$, for MPI, there is a slight increase at $2{\times}10^9$. This increase is the result of the problem size being at the edge of the limits of available RAM, likely resulting in an increased time due to many off chip memory accesses.

As p is increased, the slope of the lines decreases because the time to solve the same size of problems with more processors should be less, as expected. The sequential times scales as $O(n)$, and the parallel algorithms should scale with the dominate term of $O(n/p)$, where p is fixed. Thus, the ratio of the slopes of the serial to the parallel line should be roughly equal to $1/p$. By visual examination, this was found to be the case (at least generally), indicating that in fact the $O(n/p)$ term is mostly dominant through the regime of problems tested.

(a) $p = 4$

(b) $p = 16$

Figure 6: Plot of time T_{exp} vs problem size n for MPI algorithm.

4.4 Determining Asymptotic Time Coefficients

The experiment to determine the asymptotic coefficients as discussed in the experimental set up was performed for various processor counts, for both algorithms. Assuming the $O(n/p)$ term dominates, as demonstrated in the previous section, the plotted coefficients represent an approximate experimental time of the form

$$T_{exp}(n,p) \simeq C_0 T_{pred}(n,p) = C_0 \left(\frac{n}{p}\right), \quad n > n_0 \tag{3}$$

A table summarizing the visually estimated coefficients, for both algorithms and various p, is given in Table 1. Plots of the ratio T_{exp}/T_{pred} versus n, for various p, are given below. These plots were used to visually estimate the value of the coefficents. Enlarged graphs are included as necessary to better discern the values of data points.

As demonstrated in the table, the model in Eq. (3) is fairly accurate as the coefficients generally agree, but there is a clear trend with p that is not being accounted for. The fact that C_0 is increasing with p indicates that there is extra *positive* terms not included in the predicted time model T_{pred}. This is not expected as T_{pred} does not attempt to include a term for the $O(p)$ or $O(\log p)$ parallel steps.

The variablity in the threshold coefficient n_0 in the MPI algorithm is also a result of the lack of including weighted $O(p)$ and $O(\log p)$ terms. In particular, at lower values of n the ratio T_{exp}/T_{pred} is above C_0. This indicates that the experimental times are much larger than $C_O(n/p)$ predicts. Thus, at lower values of n, the $O(p)$ terms are more significant (relative to the total run time) so the $O(n/p)$ is no longer the dominant term in the time complexity. As p is increased, the $O(n/p)$ term becomes less dominant, leading to an increased value of n_0. This can be seen in particular in the $p = 64$ case. The OpenMP algorithm does not demonstrate as much variablity in n_0 because n is large relative to the low values of p, and the shared memory allows for the simple $O(p)$ serial prefix summation in the algorithm to be performed with relatively low performance cost.

Table 1: Tabulated results for estimating asymptotic coefficients

p	C_0	n_0
OpenMP		
2	0.0034	0.1×10^9
4	0.0044	0.1×10^9
8	0.0075	0.1×10^9
MPI		
4	0.0044	0.4×10^9
16	0.0095	0.4×10^9
64	0.0150	1.0×10^9

5 Conclusions

The various experiments performed were able to provide insight into the performance of the two algorithms. Both algorithms were able to demonstrate speedup over the serial algorithm. Although the weak scaling studies did not show very good efficiency, they did demonstrate that the algorithms can be used to solve large problems that a serial algorithm could not necessarily handle. The time complexity experiment demonstrated that in general for parallel algorithms the dominant parallel $O(n/p)$ terms can provide a good estimate of scaling with n, but may not be sufficiently accurate across a large range of p. Also, the weak scaling experiment demonstrated that there is extra costs that we are clearly not accounting for in our model. In general, because the prefix sum involves such primitive operations, it exposed any memory or other overhead in computations, even at relatively large input sizes of $\mathcal{O}(10^9)$ integers. This caused the need for very efficient code to demonstrate speed up.

Overall, the MPI algorithm is a better choice over OpenMP, at least for this machine and sufficiently large values of n. The MPI algorithm showed similar, or slightly better, performance at low thread counts than OpenMP, for all experiments performed. In addition, it can easily be extended to large core counts without the need for shared memory. On a machine that did not have the issue of non-uniform RAM access times across multiple chips, the OpenMP algorithm may show better performance at low core counts. The OpenMP algorithm may also perform better at lower values of n that were not thoroughly tested in the experiments. The results of the asymptotic complexity experiments indicate that the value of n_0 is generally larger for the MPI algorithm. Once n is large enough, this fixed cost of the MPI communication, per p, is negligible. Although the OpenMP algorithm has a similar overhead due to the cost of managing threads, it is not quite as large.

A couple strategies could be used.

ASYNCHRONOUSE WOULD BE BETTER. A simple setup to play around with 1D parallelized code. It can be used in the future to test other algorithmic technqiues without the overhead of a full MOnte Carlo code. T