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

Simon Bolding
CSCE 626

May 2, 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

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 domain copy each processor has a copy of the entire physical domain of the problem, as opposed to domain decomposition in which each processor only contains a portion of the domain. Each has their own benefits and difficulties. The algorithms were implemented in a simplified version of a research code that solves a pure-absorber, one-dimensional Monte Carlo particle transport code. The goal is to explore the algorithms and have a working simplified model that can be used to test parallelization strategies in the future.

In general, domain copy is a much simpler and efficient parallelization strategy. However, for very large problems, or in methods that require additional information to be stored or communicated more often, it can become unfeasible due to memory and communication restraints. In particular, the full solution method the research code the algorithms were implemented in fits into this category.y research code uses fits in to this case. That is why it is of interest for me to explore and understand the basics of the domain decomposition strategy, which does not require tallied information to be communicated at the end of simulations, but does require more communication during the simulation.

Multiple experiments were performed, and the results were analyzed to compare the performance of the algorithms as a function of the number of parallel processors used. Coefficients for the theoretical asymptotic time complexity were determined were applicable. Both strong and weak scaling studies were performed as well, and speed up studied as a function of the number of histories performed. The algorithms are benchmarked against the original sequential algorithm. As expected, the domain copy algorithm was significantly more efficient for this simple 1D problem, but the domain decomposition algorithm was able to be demonstrated.

1.2 Monte Carlo Transport Basics

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 mesh: one dimension representing location x in 1D space and one dimension representing the angle, or direction, of particles μ). The mesh is broken up into elements (or cells). The only material property of interest is the removal cross section σ which represents the average probability a particle interacts per differential unit length. For all problems herein there is a single material cross section in the domain to simplify analysis. A mean free path $\lambda = 1/\sigma$ represents the average distance a particle travels before interacting.

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** $\psi(x, \mu)$. The final result that is typically of interest is an angular integrated particle density, referred to as the **scalar flux** $\phi(x)$. This scalar flux also has a linear shape within a cell that must be computed. The scalar flux is determined based on an input source distribution of particles that is represented over the mesh by a linear distribution (in space and angle) over each element. To determine the scalar flux, the following basic process is performed:

1. For N histories:
   (a) Source a random particle history from the specified source distribution: a location x and angle μ.
   (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
2. For each cell, average contribution of all histories to tallies
3. Compute $\phi(x)$ by integrating $\psi(x, \mu)$ over μ

It is noted that the reason particles do not have to scatter is that a deterministic solver provides a representation of scattering events as a term included in the source distribution. This determinstic solve is performed in simulations, but is not included in the timing of algorithms because it is performed before the Monte Carlo solve ever begins.

1.3 Domain Copy and Decomposition

There are two basic approaches to simulating the above problem in parallel: 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

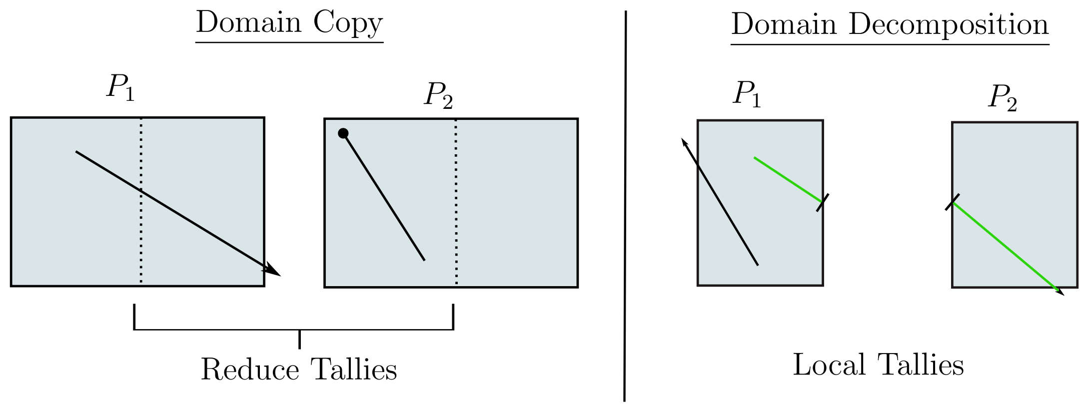

Figure 1: Illustration of domain copy (left) and domain decomposition (right) strategies on two processors.

and N is the total number of histories to be simulated. Random numbers are gerated 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 results of the scalar flux The linear representation for each cell results in two points for each spatial cell to be communicated. for each processor are combined at the end of the simulation using an all-reduce strategy. On average, each processors (N/p) simulations take the same time as the particle histories are samples of the same underlying distributions, limiting asynchronization of processors.

2 Theoretical Analysis

2.1 Sequential Algorithm

An efficient, time optimal sequential algorithm was used as a reference to compare to parallel algorithms. The algorithm is given in Alg. 1. This algorithm has asymptotic time complexity $T(n) = O(n)$, where n is the number of input integers. Because $n - 1$ additions must be performed to calculate the prefix sum, this is clearly the optimal sequential algorithm. It is noted as an implementation detail that the call to `PrefixSumSerial` uses the same function call in the parallel versions of the code. This ensures there are no optimization differences for this function call.

2.2 MPI Algorithm

The algorithm for the MPI implementation is given in Alg. 2. This algorithm uses the standard approach from the previous homework of computing local prefix sums, followed by a global prefix sum on the last element in each local array, before adding the sum of previous processors to each element.

The algorithm assumes that the input elements and output prefix elements are distributed across p processors. Thus, for p processors and n integers, each processor contains $m \approx n/p$

Algorithm 1 Serial prefix sum algorithm

procedure PrefixSumSerial(A, B) ▷ Assumes len(A)=len(B)=n
  variables: i, n
  $n = \text{len}(A)$
  $B(0) := A(0)$ ▷ Initialize prefix sum
  for $i = 1$ to $n - 1$ **do**
    B(i) := A(i)+B(i-1)
  end for ▷ B now contains the prefix sum of A
end procedure

elements. If n/p is not an integer, then the remaining elements are distributed, one per processor, amongst the first $n \mod p$ processors. Each processor computes the prefix sum of its m elements, an order $O(n/p)$ operation. Using `MPI_Allgather`, a copy of the last element in each processor's prefix sum array (representing the sum of all elements on each processor) is copied to *all* processors. Thus, each processor now contains an array C of size p, with the i-th element containing the sum of the elements on the i-th processor. The prefix sum of C is computed by each processor, an $O(p)$ operation. Finally, each processor adds the sum of integers from all previous processors to its local prefix sum array as necessary.

For time and work complexity, the initial prefix sums and final value additions require $O(n/p)$ time and $O(n)$ work. To perform the global prefix sum, the `MPI_Allgather` command was utilized to copy the processor sums to each processor. Although p numbers must be communicated, by combining messages in a tree-reduction, this command only requires the large communication overhead of $O(\log p)$ messages. Thus, as far as asymptotic complexity, this process represents an $O(\log p)$ communication time, with $p \log p$ work. The serial prefix sum of each processors array is computed only up to that processor, but this is bounded from above by $O(p)$ time and $O(p^2)$ work. Thus, in total, $T(n) = O(n/p + p + log(p)) \simeq O(n/p + p)$ and $W(n) = O(n + p^2 + p \log p) \simeq O(n + p^2)$. It is noted that for the case of the experiments performed in this work, $n \gg p$, and the $O(p)$ terms become negligible. We have verified this algorithm works in previous homework.

2.3 OpenMP Algorithm

The OpenMP algorithm is given in Alg. 3. The OpenMP algorithm works similarly to the MPI algorithm above. Each thread performs the local prefix sum on its portion of the input array, before a serial prefix sum is performed on the p processor sums, stored in an additional array. This step is performed with the simple serial algorithm by one processor. This is done because for the shared memory experiments performed, $p \ll n$, and the memory overhead of using a more complicated algorithm would ultimately result in a less efficient algorithm and longer run times. Finally, the sum of previous processor's values are added to each portion of the array in parallel, as necessary.

For time complexity, the initial prefix sum and addition of values at the end are $O(n/p)$, with work complexity $O(n)$. The prefix sum of the processor sums is $O(p)$ time, but, because there is shared memory access, this step can be performed by a single processor. This results

Algorithm 2 MPI prefix sum algorithm

```
procedure PrefixSumMPI(A)                                  ▷ Assumes len(A)=n
        ▷ Assume A and B distributed evenly on processors, ≈ n/p to each processor
    Local variables: id, p, m            ▷ MPI Rank, number of processors, my size
    Local variables: C, D                ▷ Arrays of size p for parallel prefix sum
    id := get_my_id()                    ▷ Procs. numbered 0, 1, . . . , n − 1
    m := ⌊n/p⌋                           ▷ Determine size on each processor
    if id < n  mod p then
        m := m + 1
    end if
    Local variables: myints              ▷ this procs' m input values of A
    Local variables: mypsums             ▷ this procs' m prefix sum values of B
    if  len(A) = 1 then
        B(0) := A(0)
        return
    else
        PrefixSumSerial(myints,mypsums)
    end if
    MPI_Allgather(mypsums(m − 1),C)      ▷ C now contains copy of sum of elements for
each of p processors
    if id > 0 then D(0) := C(0)
        for i = 0 to id − 1 do   ▷ All Procs perform prefix sum on their local copy of C
            D(i) := D(i − 1)+C(i)
        end for
        for i = 0 to m do
            mypsums(i) := mypsums(i)+D(id − 1)
        end for
    end if                 ▷ Collectively the p arrays mypsums contain prefix sum of A
end procedure
```

Algorithm 3 OpenMP prefix sum algorithm

```
procedure PrefixSumOpenMP(A)                              ▷ Assumes len(A)=n
                              ▷ Assume A and B are in shared memory among n processors
    Local variables: id, p, m              ▷ MPI Rank, number of processors, my size
    Local variables: C, D                   ▷ Arrays of size p for parallel prefix sum
    id := get_my_id()                                 ▷ Threads numbered 0, 1, ..., n − 1
    Local variable: mybeg, myend                 ▷ Determine my portion of the array
    m := ⌊n/p⌋                                          ▷ Initial size for each processor
    r := n mod p
    if id < r then                                       ▷ Account for n/p not an integer
        mybeg := (m + 1) ∗ id
        myend := mybeg + m
    else
        mybeg := r ∗ (m + 1) + (id − r) ∗ m
        myend := mybeg + (m − 1)
    end if
    if len(A) = 1 then
        B(0) := A(0)
        return
    else
        PrefixSumSerial(A(mybeg : myend), B(mybeg : myend))        ▷ myend is inclusive
    end if
    C(id) := B(myend)                                   ▷ Store sum of each thread's ints
    if id = 0 then
        PrefixSumSerial(C,D)                      ▷ D contains prefix sums of each thread
    else
        for i = mybegin to myend do
            B(i) := B(i)+D(id − 1)
        end for
    end if                                           ▷ B now contains the prefix sum of A
end procedure
```

in $O(p)$ work. In total, for this algorithm, $T(n) = O(n/p + p)$, where $p \ll n$, and Work is $O(n + p)$. We have verified this algorithm works in previous homework.

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 Description of Experiments

Several experiments were performed to gauge the performance and scalability of Alg. 2 and 3. The experiments are discussed individually below. Batch files to run the various jobs were created using a Python script. Output files were processed with a Python script as well. Example scripts can be found in the submitted tar ball. It is noted that the measured simulation times for the experiments times only account for the prefix sum computations; they exclude any extra input, output, or initialization timing costs. Timing information was performed using built in C functions, such as `gettimeofday`. We only sample two different problems, but really there is much more of the phase space could be sampeled, but this is representative of the two approaches.

3.2.1 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. This is different than scalability, which is the ratio T_1/T_p, which can also be used as a performance measure for strong scaling. 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.

For both the MPI and OpenMP algorithm, 10^9 integers was chosen as the input problem size. The choice of this number was to ensure that the size of each portion of the input and prefix sum arrays stored by each processor was larger than the $L3$ cache, for all simulations. The size of the input array for 10^9 integers is 4 bytes/int $\times 10^9$ integers = 4 GB. The size of the output arrays, which use 8 byte longs, is 8×10^9 GB. In total, around 12 GB of memory is needed, which is well under the limit of 22 GB per node. The largest run of 256 cores still requires around 10 MB per core, which is greater than the size of the L3 cache.

3.2.2 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. Decreases in efficiency likely indicate cost increase from overhead due to memory access or communication times.

For the weak scaling studies, 10^8 integers, per core, were used for both the MPI and OpenMP. This number was chosen for the same reasons as in the strong scaling study: the value is larger than the cache size and the total problem size will fit in the available RAM per node.

3.2.3 Speedup Versus Problem Size

A study similar to strong scaling was performed, in which the problem size is varied, but the number of processors is fixed. Thus, the interest is in whether the algorithm, for a particular p, behaves as expected when n is increased. This helps to validate our asymptotic analysis, and will determine if the $O(n/p)$ term is truly dominant as anticipated. This study provides information similar to the strong and weak scaling studies, but should expose any scaling issues that are not necessarily related to communication, e.g., memory issues. It also provides a good overall indication of the algorithms' ability to solve problems of varying size more efficiently than the serial algorithm.

For a couple different choices of p, the algorithms were tested on various problem sizes n. The run times of these simulations is then compared to the run times for the same test problems by the serial program.

3.2.4 Determining Asymptotic Coefficients

To determine the validity of our theoretical analysis of the algorithms, simulations were performed for a fixed number of processors with variable input sizes. This helps to determine if our theoretical model for run time, as a function of input size and number of processors, is sufficiently accurate to model the run time. Our model does not account for communication latencies or memory access times. This study also helps to determine in what range of problem sizes our asymptotic scalings are accurate.

The theoretical output time of the model is predicted as T_{pred}. In general, the model T_{pred} is a function of the problem size n and number of processors p. The model can be represented as $T_{pred} = C_0 g(n,p)$ for some $n > n_0$, where $g(n,p)$ is the expected time complexity determined by algorithmic analysis, for a given compute system. By performing experiments for various n, given a fixed p, the coefficients of the model C_0 and n_0 can be determined by plotting the ratio of the actual experimental time T_{exp} to predicted run times. The point at which the plot levels off represents n_0, and the value of the ratio where it has leveled off approximates C_0.

For both the OpenMP and MPI algorithm, the dominant term is expected to be $O(n/p)$. Rather than trying to fit a function to determine the other coefficients in the model (e.g., $C_0 * n/p + C_1 * p + C_2 * \log p$), which would be difficult due to statistical noise in the results and the small contribution from the $O(p)$ terms, only the coefficients for the dominant $O(n/p)$ term are determined.

3.3 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

$$\sigma\left(\frac{T_i}{T_j}\right) = \frac{T_i}{T_j}\sqrt{\left(\frac{\sigma_{T_i}}{T_i}^2\right) + \left(\frac{\sigma_{T_j}}{T_j}^2\right)}. \tag{1}$$

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 each of the experiments.

4.1 Strong Scaling Study (speed up)

4.1.1 OpenMP

The results for the speed up study for the OpenMP algorithm is given in Fig. 2. As demonstrated, the algorithm does not demonstrate ideal speedup (as expected), but is able to solve the problem increasingly faster as the number of threads used is increased. Some rough timing estimates indicate that the $O(p)$ calculation in the OpenMP algorithm is < 0.1 % of the total runtime (for 10^9 integers), which was on the order of 5 seconds. Thus the limiting factor is likely the overhead of managing threads and memory access times.

The algorithm described by Alg. 3 had to be modified slightly to achieve the shown results (and all later OpenMP results). It was modified to limit the cost of memory access times. In the corrected algorithm, each thread was allowed to create its own memory within the `omp parallel` section (essentially emulating distributed memory), for storing the input and prefix sums. This was necessary to allow the program to correctly assign memory near the location of the cores. This significantly improved the speedup, in particular going from 4 to 8 threads, which previously showed a drop in speedup. This drop was the result of processors having to go off chip to access their portion of the array in RAM.

4.1.2 MPI

The speedup results for the MPI algorithm are given in Fig. 3. A more legible plot for the cases of $p \leq 16$ is given in Fig. 4. The MPI algorithm demonstrated slightly better speed up at equivalent core counts than the OpenMP algorithm. It is noted there is an irregular drop from 4 to 8 cores in the MPI speed up plot. This is likely due to the fact that 4 cores will be on 1 chip, whereas for 8 cores some of the RAM will be off chip, as discussed in the previous section and machine specifications, leading to a decrease in expected efficiency gain.

The program continued to scale out to 128 and 256 cores, although at a reduced rate of increase at higher core counts. The rate of increase drops off at higher core counts due to the increased cost of the communication in the $O(\log p)$ communication step, relative to the $O(n/p)$ prefix sum steps by each processor. There was much greater variance at 128 and 256 cores, due to much more off node communication time, leading to some difficulty in determining how well the speedup is increased at the high core counts.

It is noted that although the MPI algorithm has the same theoretical time complexity as a linear array approach, there is less communication steps ($O(\log p)$ versus $O(p)$ communications), which leads to overall a significantly more efficient algorithm as the cost of the communication steps becomes the limiting factor at higher MPI rank counts. A linear array approach was also tested and found to not scale past 64 cores due to the increased cost of communication. This demonstrates an inaccuracy in our model, which is expected because it doesn't account for communication overhead. Alg. 2 has similar communication cost to a tree-traversal based algorithm.

Figure 2: Plot of speedup versus p for OpenMP algorithm, for prefix sum of 10^9 integers.

Figure 3: Plot of speedup versus p for MPI algorithm, for prefix sum of 10^9 integers.

Figure 4: Zoomed in plot of speedup versus p for MPI algorithm, for prefix sum of 10^9 integers.

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 5: Plot of weak scaling efficiency versus p for OpenMP algorithm, for 10^8 integers per processor.

Figure 6: Plot of weak scaling efficiency versus p for MPI algorithm, for 10^8 integers per processor.

Figure 7: Zoomed in plot of weak scaling for MPI algorithm, for 10^8 integers 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 2×10^9. 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) $p = 8$

Figure 8: Plot of time T_{exp} vs problem size n for OpenMP algorithm.

(a) $p = 4$ (b) $p = 16$

Figure 9: 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

$$T_{exp}(n,p) \simeq C_0 T_{pred}(n,p) = C_0 \left(\frac{n}{p}\right), \quad n > n_0 \tag{2}$$

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. (2) 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.