Programming Assignment 2

Simon Bolding
CSCE 626

March 30, 2015

References

1. People in class - Hans Hammer, Daniel Holladay, Josh Hansel
2. wikipedia.org/wiki/Prefix_sum,computing.llnl.gov, mpitutorial.com
3. Class notes
4. EOS website: sc.tamu.edu/systems/eos

1 Introduction

In this work experimental analysis was performed to evaluate the performance of two different methods for computing prefix sums in parallel: an algorithm implemented with MPI and an algorithm implemented using OpenMP. Given a sequence of numbers $x_1, x_2, ..., x_n$, the prefix sums are the partial sums

$$\begin{aligned} s_1 &= x_1 \\ s_2 &= x_1 + x_2 \\ &\vdots \\ s_n &= x_1 + x_2 + \cdots + x_n. \end{aligned}$$

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, for both algorithms. Coefficients for the theoretical asymptotic complexity were determined. Both strong and weak scaling studies were performed as well, and speed up studied as a function of the number of elements. The algorithms are benchmarked against the best sequential algorithm.

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

Algorithm 1 Serial prefix sum algorithm

```
procedure PrefixSumSerial(A, B)                    ▷ Assumes len(A)=len(B)=n
    variables: i, n
    n = 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
```

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 $m \approx n/p$ 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

Algorithm 2 MPI prefix sum algorithm

procedure PREFIXSUMMPI(A) ▷ Assumes len(A)=n
▷ Assume A and B distributed evenly on processors, $\approx n/p$ 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 $0, 1, \ldots, n-1$
$m := \lfloor n/p \rfloor$ ▷ 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 $0, 1, \ldots, n-1$
Local variable: $mybeg$, $myend$ ▷ Determine my portion of the array
$m := \lfloor n/p \rfloor$ ▷ 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

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 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.

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 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

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

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. 1. 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. 2. A more legible plot for the cases of $p \leq 16$ is given in Fig. 3. 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.

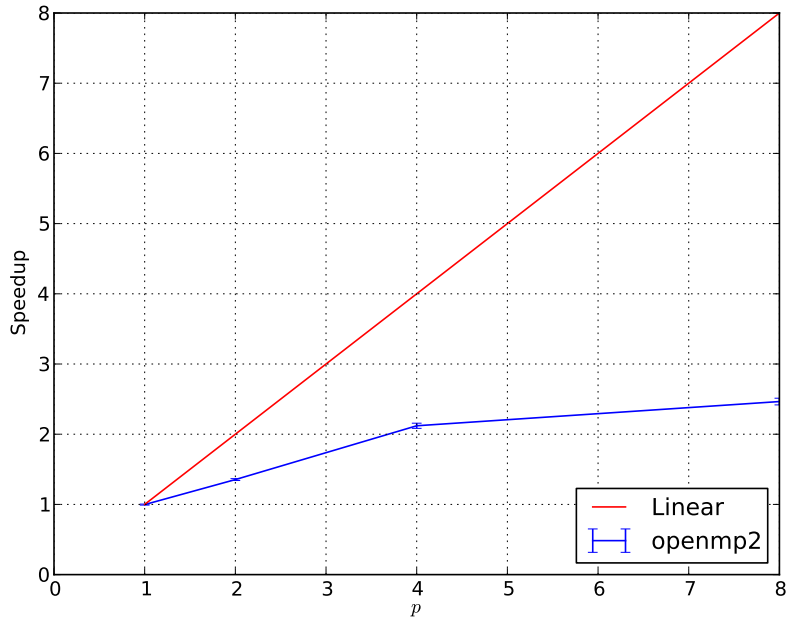


Figure 1: Plot of speedup versus p for OpenMP algorithm, for prefix sum of 10^9 integers.

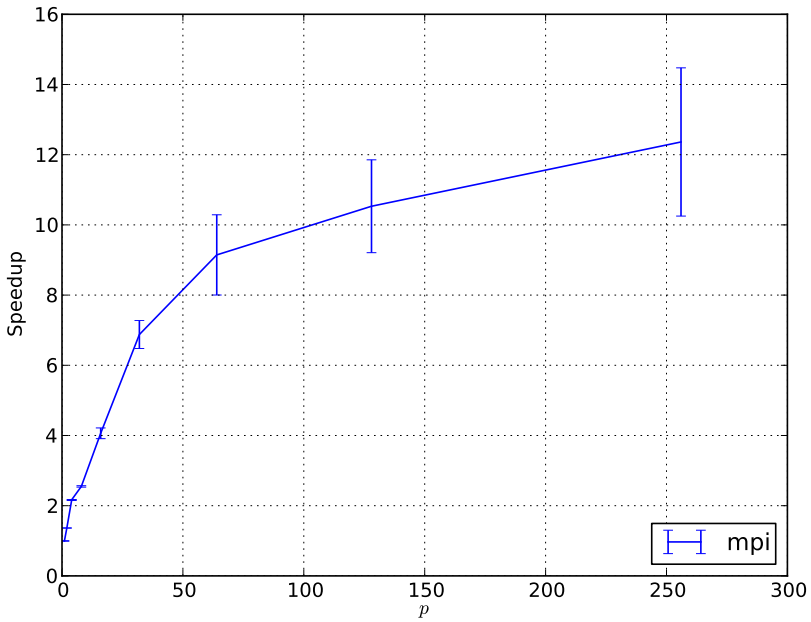


Figure 2: Plot of speedup versus p for MPI algorithm, for prefix sum of 10^9 integers.

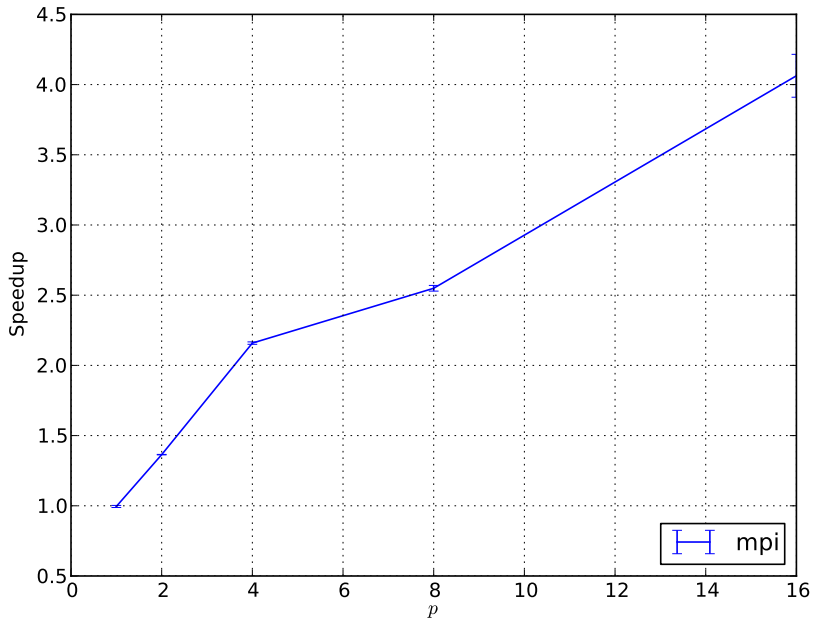


Figure 3: Zoomed in plot of speedup versus p for MPI algorithm, for prefix sum of 10^9 integers.

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.

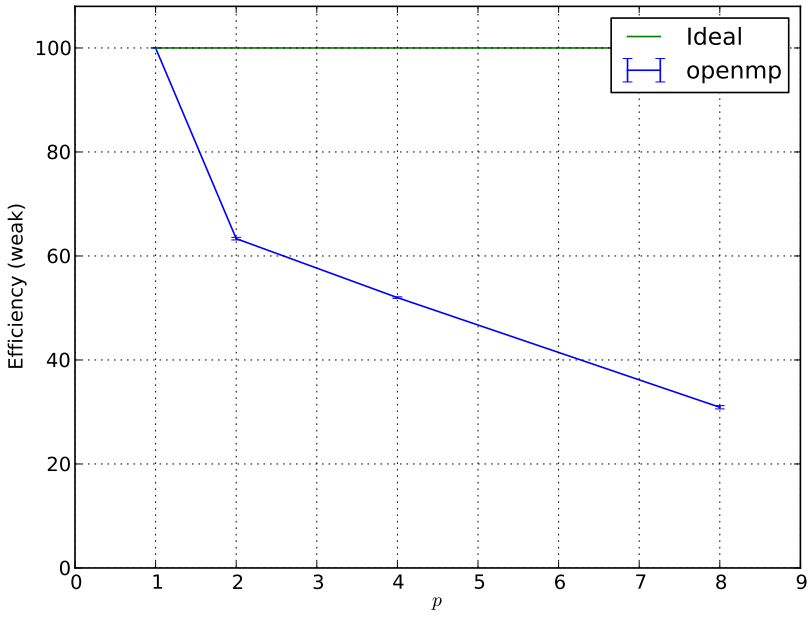


Figure 4: Plot of weak scaling efficiency versus p for OpenMP algorithm, for 10^8 integers per processor.

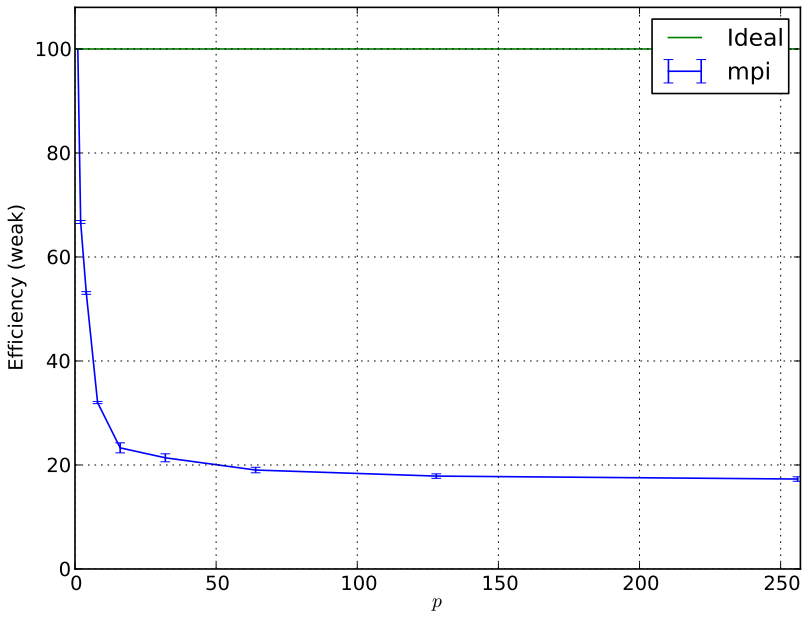


Figure 5: Plot of weak scaling efficiency versus p for MPI algorithm, for 10^8 integers per processor.

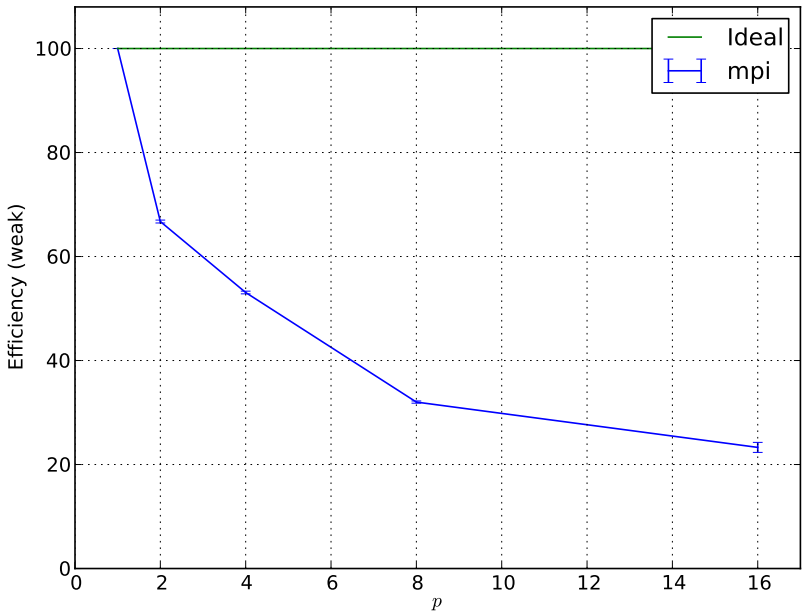
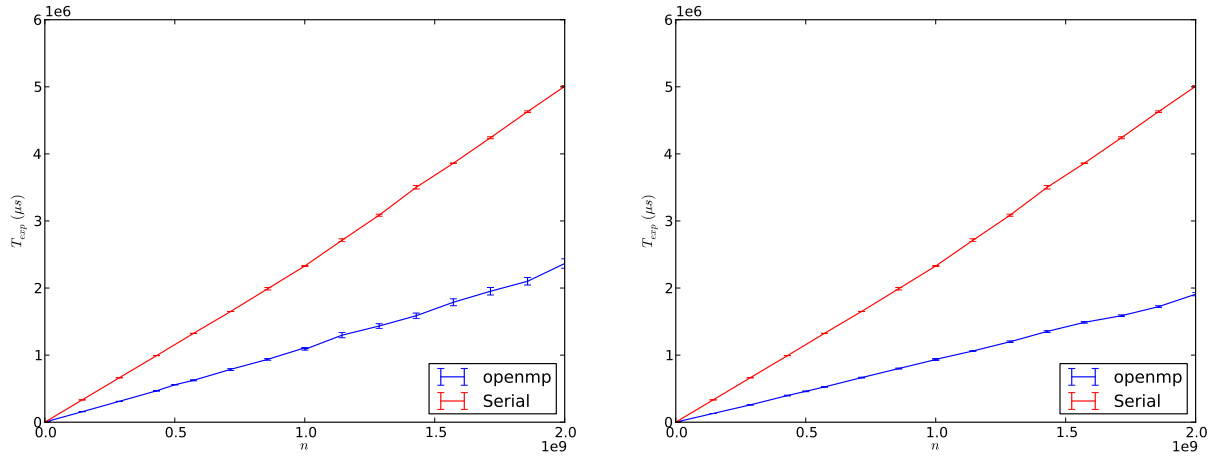


Figure 6: Zoomed in plot of weak scaling for MPI algorithm, for 10^8 integers per processor.

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 complexities 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 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 as 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 7: Plot of time T_{exp} vs problem size n for OpenMP algorithm.

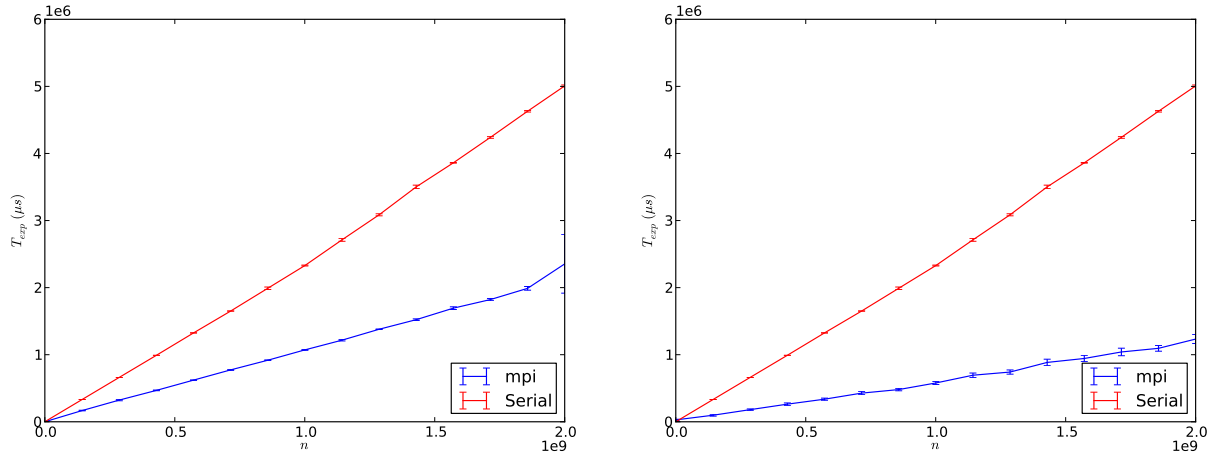
4.4 Determining Asymptotic Coefficients

4.4.1 OpenMP

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

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

A table summarizing the visually estimated coefficients, for both algorithms and various p, is given in Table **??**. Figures are given below, with zoomed in photos given with the removed



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

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

data points to resolve the data points. The fact that C_0 is increasing with p, indicates that there is extra terms the model in Eq. (2) is not accounting for. This is clearly true as we have not attempted to determine the coefficients in front of the parallel step $O(p)$ or $O(\log p)$. The variablity in the asymptotic coefficient is partially due to the fact that the communication time is more significant, relative tot he total run time, as compared to the total run time. Since more processors are used by the case of $p = 64$, the n/p term becomes not the dominant term in the model.

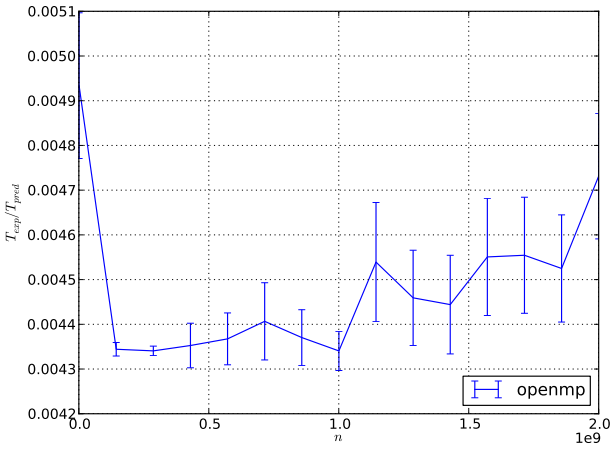
Table 1: Tabulated results for estimating asymptotic coefficients

p	C_0	n_0
OpenMP		
4	0.004	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

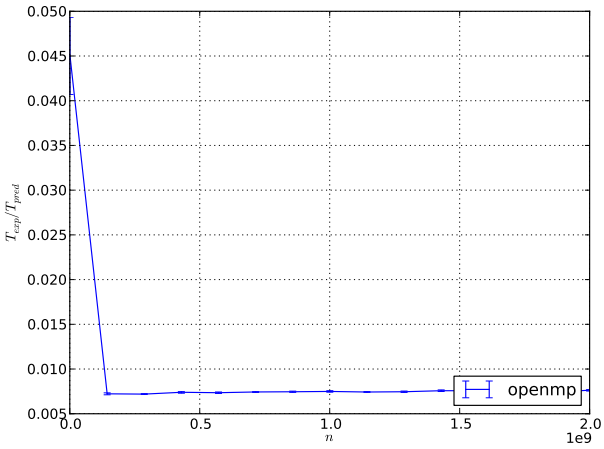
4.4.2 MPI

5 Conclusions

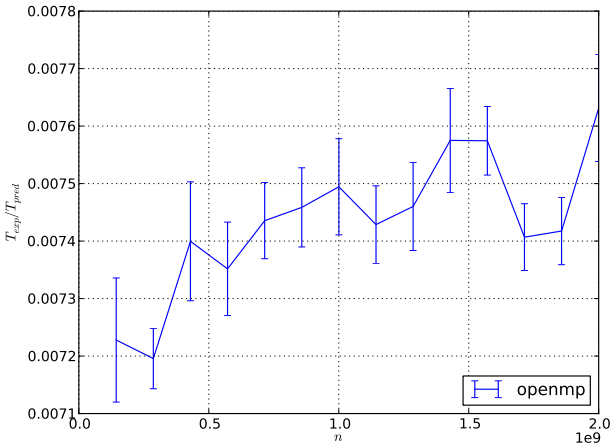
Since the prefix sum involves such primitive operations, it exposed any memory or other overhead in computations. It took very efficient code to demonstrate much speed up. The



(a) $p = 4$

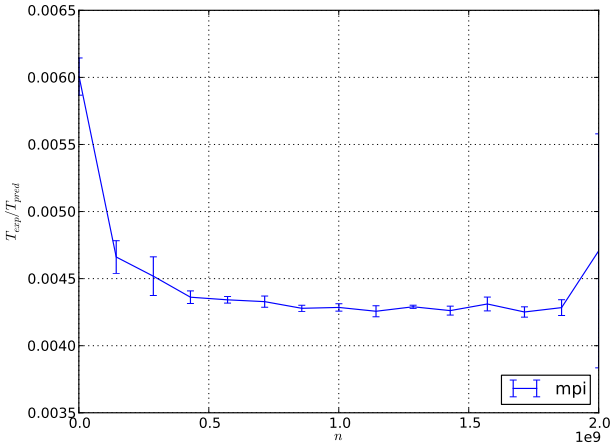


(b) $p = 8$

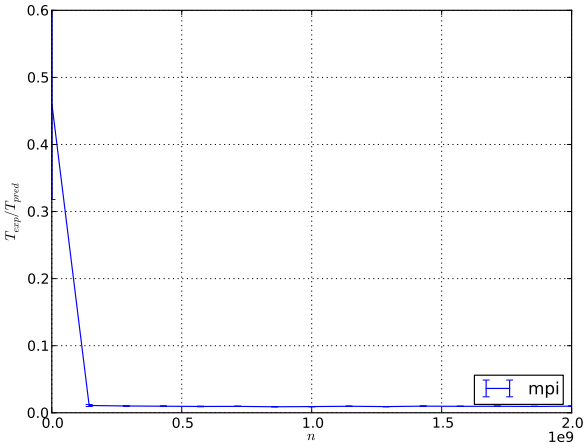


(c) $p = 8$, enlarged

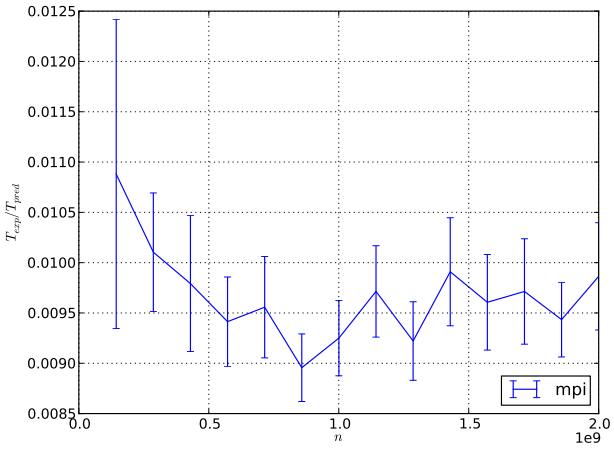
Figure 9: Plot of T_{exp}/T_{pred} vs problem size n for OpenMP algorithm.



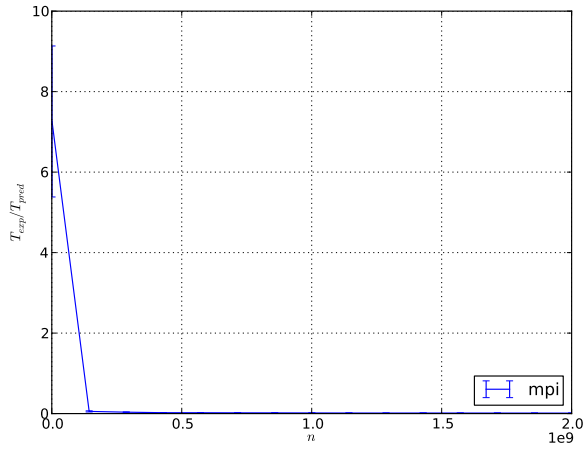
(a) $p = 4$



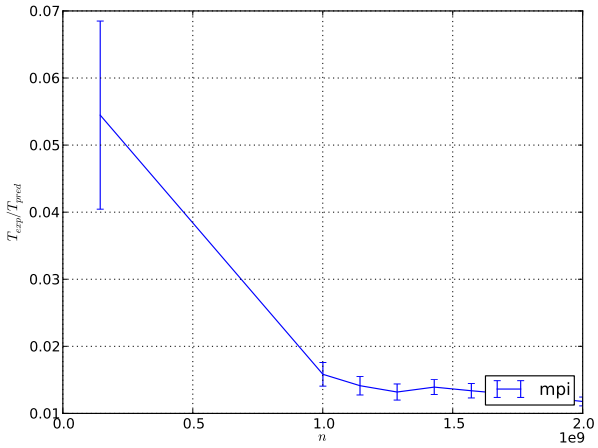
(b) $p = 16$



(c) $p = 16$, enlarged



(d) $p = 64$



(e) $p = 64$, enlarged

Figure 10: Plot of T_{exp}/T_{pred} vs problem size n for MPI algorithm.

weak scaling experiment demonstrated that there is extra costs that we are not expecting.