Prefix-Sum computation using OpenMP and MPI

Arun Krishnakumar Rajagopalan
Texas A&M University
arunxls@gmail.com

Abstract

Prefix sum calculation is among the most fundamental operations performed in parallel applications. In this paper, we look at two separate implementations of the Prefix sum algorithm and evaluate their performance against the most optimal sequential algorithm.

1 Introduction

Prefix sum is a useful primitive in several algorithms such as sorting, scheduling, graph algorithms etc. The prefix sum (also called a scan) of a sequence of numbers $x_0, x_1, x_2, \ldots$ is defined as another sequence $y_0, y_1, y_2, \ldots$ where

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_0 + x_1 \\ y_2 &= x_0 + x_1 + x_2 \\ &\ldots \end{aligned}$$

It is trivial to compute prefix sums in a sequential fashion. The rest of the paper discusses strategies to perform Prefix sums in parallel and report and analyze the results obtained. The paper continues as follows. Section 2 describes the theoretical analysis of the algorithms to compute prefix sums and their analysis. Section 3 describes the experimental setup. Section 4 contains evaluation of the parallel algorithm in comparison to its sequential counterpart. Section 5 contains concluding remarks and avenues for future improvement.

2 Theoretical analysis

In this paper, we discuss 3 separate implementations of Prefix sum - Sequential, OpenMP and MPI.

Algorithm 1 Sequential Prefix Sum

```
1: $X \leftarrow [x_0, x_1, \ldots, x_n]$
2: $Y \leftarrow \emptyset$
3: $Y[0] \leftarrow X[0]$
4: for i from 1 to N do
5:     $Y[i] \leftarrow X[i] + Y[i-1]$
6: end for
7: return Y
```

2.1 Sequential

The sequential algorithm is described by Algorithm 1. X contains the array of input elements $[x_0, x_1, x_2, \ldots, x_N]$. Y stores the computes prefix sums. Line 3 initializes the first element of Y to the first input element. Line 4 is the only loop in the algorithm. We iterate $N-1$ times, where N refers to the size of the input array. At every loop iteration, we set $Y[i]$ to be the sum of the current input element and the previously computed sum $Y[i-1]$. Thus, at the end of each loop iteration, we end up with the correct prefix sum up to the iteration variable.

2.1.1 Complexity Analysis

The loop at Line 4 determines the complexity of the algorithm since all other steps are constant $O(1)$ operations. The loop runs $N-1$ times and each operation inside the loop runs in $O(1)$ time. Thus, the total time complexity $T(n)$ is,

$$T(n) = O(n)$$

2.2 MPI

Message Passing Interface (MPI) is a language independent communications protocol used to program parallel computers in a distributed memory system. It supports both point-to-point and collective communication.

The algorithm to compute prefix sums using MPI is described in Algorithm 2. Let X hold the

Algorithm 2 MPI Prefix Sum

```
1:  $X \leftarrow [x_0, x_1, \dots, x_n]$
2:  for N in parallel do
3:      $id \leftarrow getId()$
4:      for h from 1 to log N do
5:          if $id > 2^{h-1}$ then
6:              $x \leftarrow X[id - 2^{h-1}]$
7:              $y \leftarrow X[id]$
8:              $X[i] \leftarrow x + y$
9:          else
10:             Exit
11:         end if
12:     end for
13: end for
14: return X
```

input array. The algorithm computes the prefix sum in-place. N refers to the number of processing elements. For simplicity, let us assume that the total number of processing element equals the size of input. Section 2.4 will introduce a simple pre-processing and post-processing step that will help us lift this restriction.

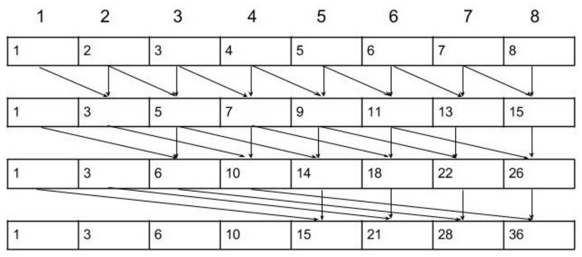


Figure 1: An example of Algorithm 2

Line 2 runs the loop in parallel across N processing elements. Line 3 returns the processor ID, which runs from 1 to N. Each processor then executes the loop on Line 4 $\log N$ times. If the condition on Line 5 is not satisfied, that particular processor will exit early and stop any further computation. Lines 6-9 are where the main computation of prefix sums takes place.

Figure 1 shows an application of this algorithm for an 8 element input size. The input elements run from 1 to 8. In each step of the algorithm, we notice that we have computed the prefix sums correctly up to $\log N$ elements.

This algorithm does not require that the number of processing nodes be a power of two. Thus, in cases where number of processors $\neq$ a power of two, we simply use the available processors and bound check appropriately so that we don't access an illegal memory location.

2.2.1 Complexity Analysis

The running time of the algorithm is governed by the loop on Line 4. Each operation inside the loop takes constant time, hence the total running time is $T(n) = O(\log n)$.

The total work done can be calculated as (total number of active processors at each iteration) $\times$ (work done per processor at each iteration). However, the work done at each iteration is O(1). Thus,

$$\begin{aligned} W(n) &= \sum_{i=1}^{\log n} n + 1 - 2^{i-1} \\ &= n\log n - \sum_{i=1}^{\log n} 2^{i-1} \\ &= n\log n - n \\ &= O(n \log n) \end{aligned}$$

2.3 OpenMP

OpenMP is a framework to program parallel applications in a shared memory model. The program starts off with a master thread which then forks off slave threads who each divide the work among themselves. The threads then run concurrently on different processors allocated by the runtime environment.

The algorithm to compute Prefix Sums using OpenMP is described in Algorithm 3. The main idea of this algorithm is to visualize the input array as a balanced binary tree and work on it two phases - a 'sweepUp' where we go from the bottom leaves to the root and a 'sweepDown', where we do the reverse.

The sweepUp phase is similar to parallel sum algorithm. At the end of this phase, the last element hold the overall sum of the input sequence.

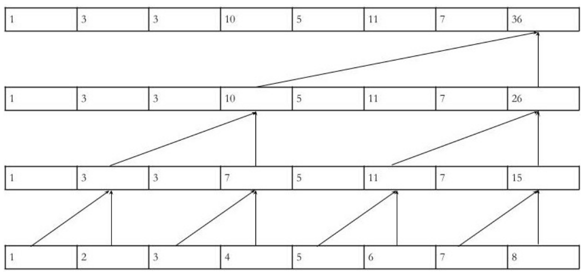
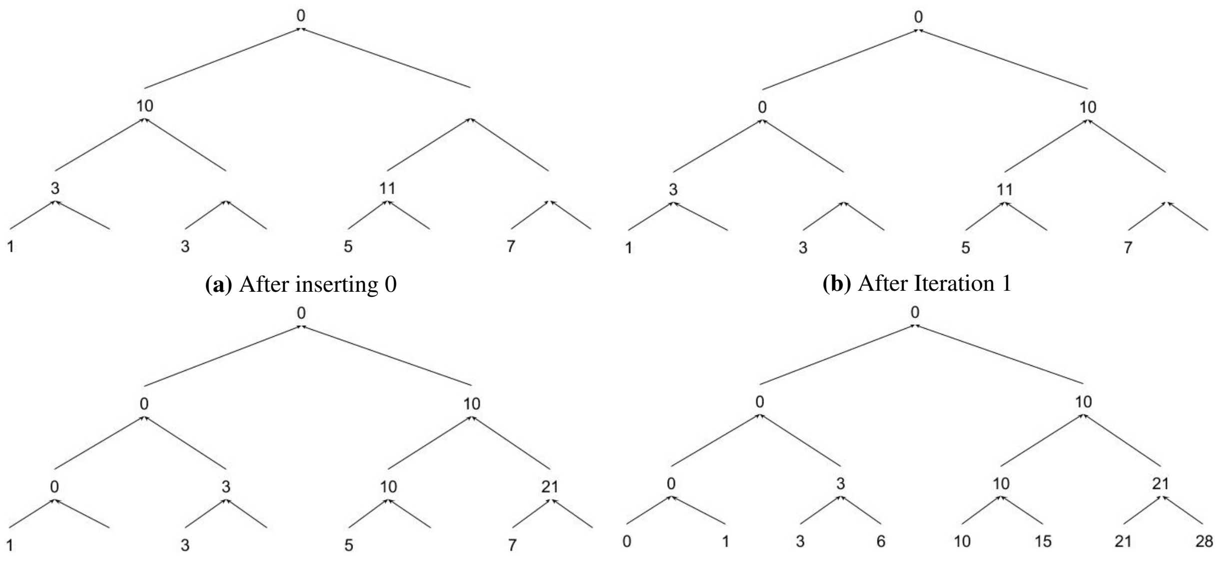


Figure 3: An example after the sweepUp Phase



(a) After inserting 0

(b) After Iteration 1

(c) After Iteration 2

(d) After Iteration 3

Figure 2: These 4 figures represent how the tree would look at the end of each iteration

Algorithm 3 OpenMP Prefix Sum

```
 1: $X \leftarrow [x_0, x_1, \ldots, x_n]$
 2: // SweepUp
 3: for h from 0 to log N - 1 do
 4:     for i from 0 to $N/2^{h+1}$ in parallel do
 5:         $a \leftarrow 2^{h+1} \times (i + 1) - 1$
 6:         $b \leftarrow a - 2^h$
 7:         $X[a] \leftarrow X[a] + X[b]$
 8:     end for
 9: end for
10:
11: // SweepDown
12: $X[N - 1] \leftarrow 0$
13: for h from log N - 1 to 0 do
14:     for i from 0 to $N/2^{h+1}$ in parallel do
15:         $a \leftarrow 2^{h+1} \times (i + 1) - 1$
16:         $b \leftarrow a - 2^h$
17:         $temp \leftarrow X[a]$
18:         $X[a] \leftarrow X[a] + X[b]$
19:         $X[b] \leftarrow temp$
20:     end for
21: end for
```

After completion of the sweepUp phase, we start the sweepDown phase. We treat the array as a balanced binary tree. The tree is constructed in such a way that the left children are all filled with values and the right children are empty. They will be filled with values as the algorithm proceeds. We start with the root and work our way down to the leaves. The root element is given the value 0 to begin with. At this stage, the tree would look like Figure 2a. Subsequent stages of the algorithm are shown in Figure 2b, 2c, 2d. At the end of this phase, we have the correct prefix sums calculated for each element.

In case the number of processors is not a multiple of two, we perform a clever manipulation taking advantage of shared memory. First, pad the input array with the enough zeros to get it to the next power of two. Then, we perform prefix sum as we would normally, but this time only using p processors. Since the extra elements were initialized to 0, the correct value of prefix sum is calculated at the end.

2.3.1 Complexity Analysis

The running time of the algorithm is bounded by the running time of the sweepUp phase + the running time of the sweepDown phase. From Algorithm 3, we can see that both the phases run $\log n$ number of times, and each iteration takes constant time. Thus, the total running time is

$$\begin{aligned} T(n) &= O(\log n) + O(\log n) \\ &= O(\log n) \end{aligned}$$

The total work done can be calculated as the sum of the work done by the sweepUp phase and the work done by the sweepDown phase. The work done by the sweepUp phase can be calculated as (total number of active processors at each iteration) × (work done per processor at each iteration). However, the work done at each iteration is O(1). Thus,

$$\begin{aligned} W(n) &= \sum_{k=1}^{\log n} \frac{n}{2^k} \\ &= n \times \left(\frac{1}{2^1} + \frac{1}{2^2} + \ldots + \frac{1}{2^{\log n}} \right) \\ &= O(\log n) \end{aligned}$$

The work complexity for the sweepDown phase is identical to the sweepUp phase. Thus, the total work is $W(n) = O(n)$. This is equal to the complexity of the sequential algorithm, making the algorithm work-optimal.

2.4 Local Prefix Sum

Algorithms 2 and Algorithm 3 work well for when the number of input elements equals the number of processing cores available. However, unless the input is trivial, the number of input elements (n) $\gg$ number of processing elements (p). In such cases, the input is divided up into chunks of $\frac{n}{p}$ size. Each processing element is assignment one of these chunks. The modified algorithm now computes the 'local Prefix Sum' of each of these chunks. Once computed, the last prefix sum calculated within each chunk is then sent to be processed according to either Algorithm 2 or Algorithm 3. Once we return from the parallel prefix-sum call, we then need to apply the appropriate offsets to each of the $\frac{n}{p}$ sections. The final result would be the correct computation of prefix sum for n input elements and p processing nodes.

2.4.1 Complexity

The algorithm consists of three phases - the first phase where were compute the local prefix sum, the second phase where we calculate the parallel prefix sums across processors and the last step where wer calculate the actual prefix sum from the offsets of the previous step. Thus, the overall time complexity $T(n)$ is,

Algorithm 4 Local Prefix Sum

```
1:  $X \leftarrow [x_0, x_1, \ldots, x_n]$
2:  $P \leftarrow [p_0, p_1, \ldots, p_n]$
3:
4:  for N in parallel do
5:      $id \leftarrow getId()$
6:      for i from $(id \times \frac{n}{p}) + 1$ to $(id + 1) \times \frac{n}{p}$ do
7:          $X[i] \leftarrow X[i] + X[i-1]$
8:      end for
9:      $Y[i] \leftarrow X[(id + 1) \times \frac{n}{p}]$
10: end for
11:
12: $ParallelPrefixSum(Y[])$
13:
14: for N in parallel do
15:     $id \leftarrow getId()$
16:     for i from $(id \times \frac{n}{p}) + 1$ to $(id + 1) \times \frac{n}{p}$ do
17:         $X[i] \leftarrow X[i] + Y[id]$
18:     end for
19: end for
```

$$T(n) = T_1(n) + T_2(n) + T_3(n)$$

Both $T_1(n)$ and $T_3(n)$ are similar to the sequential prefix sum (Algorithm 1) and share the same time complexity $O(N)$. In this case, $N = \frac{n}{p}$. $T_2(n)$ represents the time complexity of the parallel prefix sum step, which as we have seen in Algorithm 2 and Algorithm 3 is $O(\log N)$. In this case, $N = p$. The overall complexity is

$$\begin{aligned} T(n) &= O\left(\frac{n}{p}\right) + O(\log p) + O\left(\frac{n}{p}\right) \\ &= O\left(\frac{n}{p}\right) + O(\log p) \end{aligned}$$

For very large inputs, $O(\frac{n}{p}) \gg O(\log p)$.

3 Experimental Setup

This section contains various details regarding the experimental setup and the test environment.

3.1 Machine specification

All experiments were run on the **EOS** super-computer, located at Texas A&M. EOS is an IBM

Number of Processors	**Input Size** ($\times 10^9$)				
	1	**2**	**4**	**8**	**16**
1	0.297	0.595	1.193	2.407	4.907
2	0.281	0.567	1.132	2.262	4.514
4	0.164	0.331	0.665	1.337	2.732
8	0.119	0.238	0.511	1.072	2.212
12	0.130	0.256	0.502	1.026	2.087

Table 1: OpenMP implementation of Prefix Sum. All values of time are in seconds.

Number of Processors	**Input Size** ($\times 10^9$)				
	1	**2**	**4**	**8**	**16**
1	0.298	0.549	1.096	1.329	3.001
2	0.140	0.280	0.560	1.115	2.221
4	0.106	0.209	0.426	0.852	1.702
8	0.098	0.198	0.397	0.796	1.601
16	0.048	0.097	0.211	0.424	0.793
32	0.024	0.048	0.096	0.198	0.390
64	0.012	0.025	0.050	0.099	0.226
128	0.006	0.013	0.026	0.052	0.100

Table 2: MPI implementation of Prefix Sum. All values of time are in seconds.

iDataPlex commodity cluster with nodes based on Intel's 64-bit Nehalem and Westmere processor. The cluster is composed of 372 compute nodes (324 8-way Nehalem- and 48 12-way Westmere-based). The compute nodes each have 24 GB of DDR3 1333 MHz memory.

3.2 Test cases

Various experiments were run against the three algorithms explained in Section 2. For each experimental run, the input size, run time and number of processors (where applicable) were noted.

3.2.1 Input Data

The input data is generated by a call to $rand()$ after seeding it with the current processor's ID and current time-stamp. This ensures we get a random data set for each run. Each program was run on input sizes of 1, 2, 4, 8 and 16 billion integers.

3.2.2 Run time

The execution time for each run is measured using $gettimeofday()$ method. The overhead of using this function is in the order of $10 - 100ns$. Since the granularity of data needed is orders of magnitude larger, this function works well for these experiments. To get a higher expected confidence in the measured data, each experiment was run 32 times and the average runtime was taken.

3.2.3 MPI

All the MPI experiments were run on 1, 2, 4, 8, 12, 32 and 64 nodes.

3.2.4 OpenMP

All the OpenMP experiments were run on 1, 2, 4, 8 and 12 core machines. The experiments were run such that there was 1 thread per core.

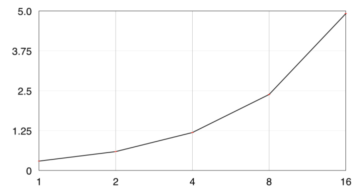
4 Evaluation

Various analysis on the observed data were carried out. In general, it was observed that the execution time for a given model reduced as we increased the number of processors in both models.

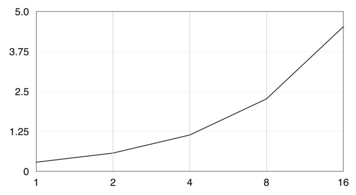
4.1 Speed Up

In this analysis, we measure the speedup achieved by the algorithm with respect to the sequential program, given by $\frac{T_s}{T_p}$, where T_s represents the time of sequential program and T_p represents the parallel implementation.Figure 7 and Figure 8 graph the speedup characteristics of both versions of prefix sum algorithm against 16 billions integers.

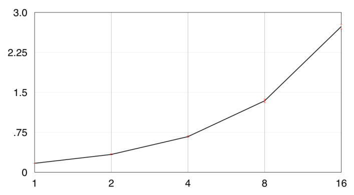
We observe that the MPI implementation scales much better as compared to the OpenMP version. For both implementations, we notice that as the processing nodes increases, the difference between theoretical and observed also increases.



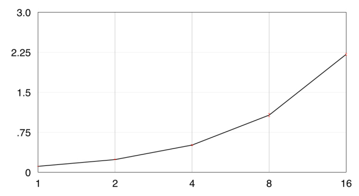
(a) OpenMP 1-thread



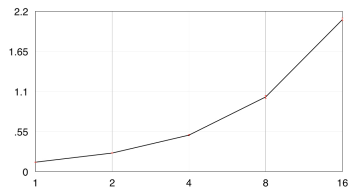
(b) OpenMP 2-thread



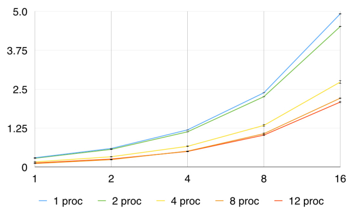
(c) OpenMP 4-thread



(d) OpenMP 8-thread



(e) OpenMP 12-thread



(f) OpenMP Thread Comparison

Figure 4: Performance of OpenMP Prefix Sum. The X axis for all figures represents 10^9 integers and the Y axis represents seconds.

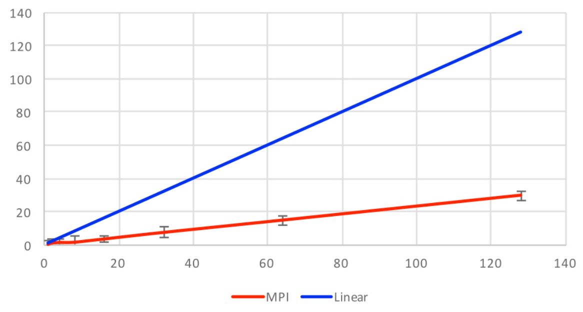


Figure 7: SpeedUp for MPI version

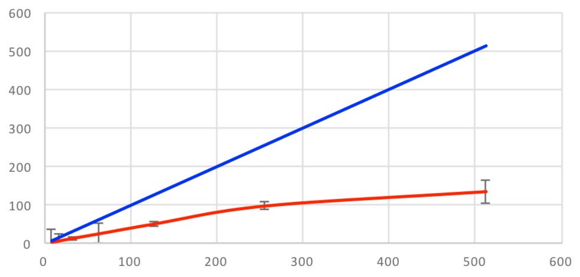


Figure 9: Strong Scaling for MPI version. Measured against 16×10^9 inputs.

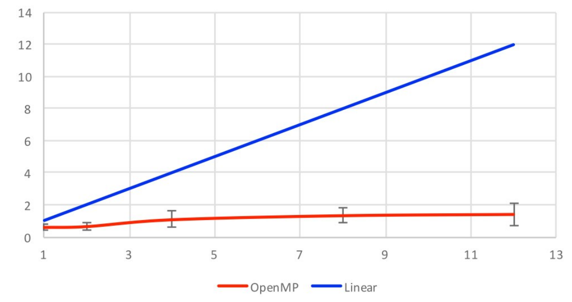


Figure 8: SpeedUp for OpenMP version

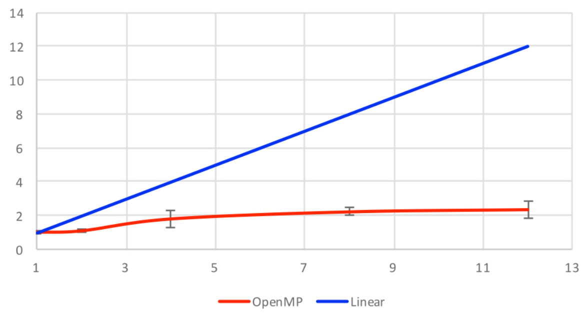


Figure 10: Strong Scaling for OpenMP version. Measured against 16×10^9 inputs.

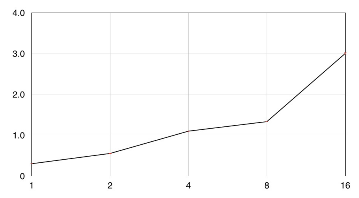
4.2 Strong Scaling

In this analysis, we measure how the solution time varies with the number of processors for a fixed problem size. Strong scaling is measured as $\frac{T_1}{T_n}$ for a given input size, where T_1 represents the time of the parallel program with 1 node and T_n represents the parallel implementation for n inputs.

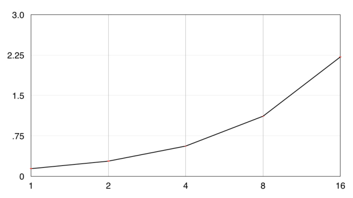
Figure 9 and Figure 10 represent the strong scaling characteristics of the MPI and OpenMP versions respectively. The MPI version has better efficiency since it is closer to the constant scale. For both graphs, we observe that the scaling between constant and the respective model deteriorates as the number of processors increases since the communication effects become more pronounced.

4.3 Weak Scaling

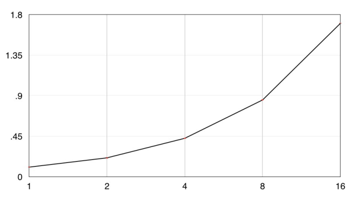
In this analysis, we measure how the solution time varies with the number of processors for a fixed problem size per processor. Weak scaling is calculated as the percentage of $\frac{T_1}{T_n}$ for a given input size



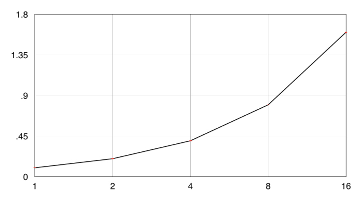
(a) MPI 1-node



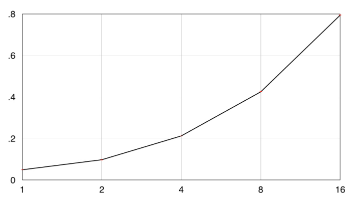
(b) MPI 2-node



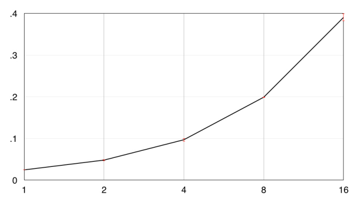
(c) MPI 4-node



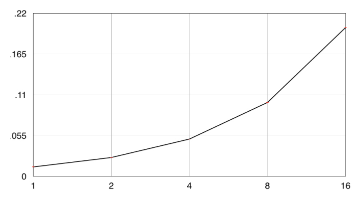
(d) MPI 8-node



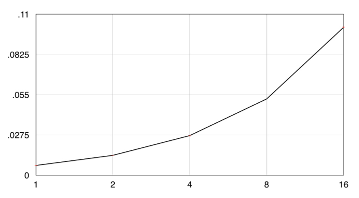
(e) MPI 16-node



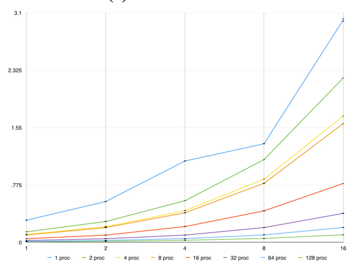
(f) MPI 32-node



(g) MPI 64-node



(h) MPI 128-node



(i) MPI Node Comparison

Figure 5: Performance of MPI Prefix Sum. The X axis for all figures represents 10^9 integers and the Y axis represents seconds.

per processor, where T_1 represents the time for 1 node with a fixed input size and T_p represents the parallel implementation for the same fixed input size across n nodes.

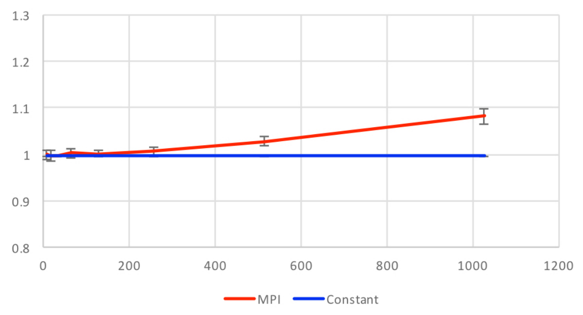


Figure 11: Weak Scaling for MPI version

Figure 11 and Figure 12 represent the weak scaling characteristics of the MPI and OpenMP versions respectively. We observe that the MPI version scales better since it is closer to the constant scale. For both graphs, we observe that the scaling deteriarates as the number of processors increases since the communication effects become more pronounced.

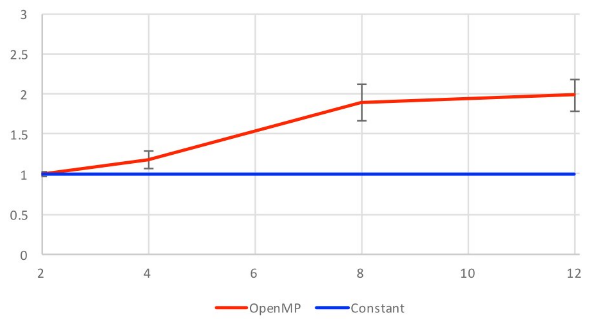


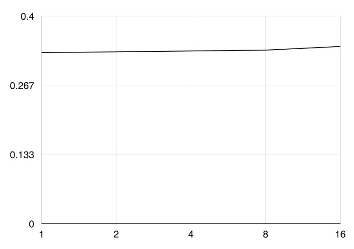
Figure 12: Weak Scaling for OpenMP version

4.4 Big O Constants

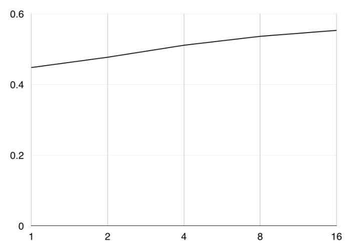
In this analysis, we determine the big-O constants for the three implementations presented above. We plot the ratio between the observed running time of the program versus its theoretical time against a range of input values. From such a graph, we calculate n_0 and c, where n_0 represents the X-axis region where the plot appears horizontal and c is the Y-axis value at this point.

Figure 13 represents such a graph for the sequential implementation of Prefix Sum. We notice that $n_0 \approx 10^4$ and $c \approx 0.147$.

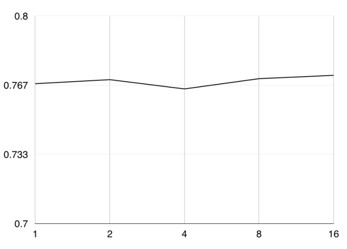
Figure 6a to Figure 6c represents such a graph for the OpenMP implementation. Similar to the



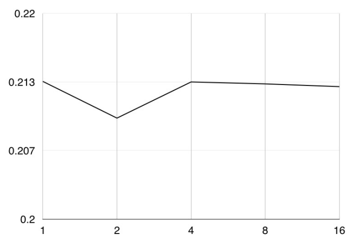
(a) OpenMP 4-threads



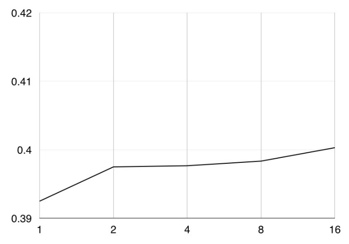
(b) OpenMP 8-threads



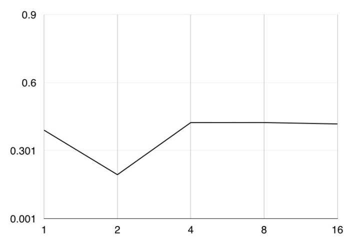
(c) OpenMP 12-threads



(d) MPI 4-nodes



(e) MPI 8-nodes



(f) MPI 16-nodes

Figure 6: Algorithm constant determination. The X axis for all figures represents 10^9 integers and the Y axis represents seconds.

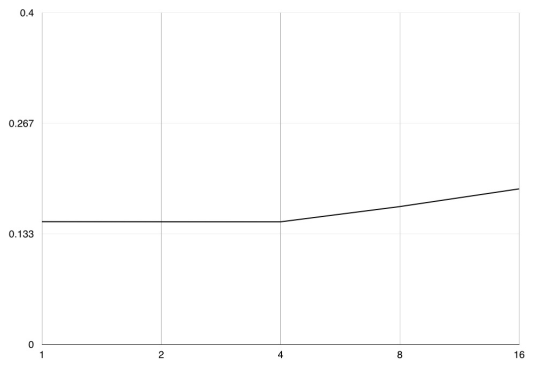


Figure 13: Algorithm constants determination for sequential Prefix Sum. The X axis for all figures represents 10^9 integers and the Y axis represents seconds.

sequential program graph, we observe that $n_0 \approx 4 \times 10^9$. However, unlike the sequential program graph, we observe that the c varies. This is explained by the communication overhead imposed as we increase the number of threads, as well as the thread-fork and join time.

As seen in Section 2.4, the complexity of the parallel prefix sum algorithm is

$$O\left(\frac{n}{p}\right) + O(\log p)$$

However in practice, we see that an additional cost $F(p)$ (where p is the number of processing nodes) is imposed on the program runtime. As the number of processing nodes increases, so does the cost associated with this function. The analysis of this extra overhead is complicated and varies significantly from system to system. $F(p)$ was observed to vary as $0.2\times$p. For the particular implementation in OpenMP, we see that $c \approx 0.4$.

Figure 6d to Figure 6f represents such a graph for the MPI implementation. We observe that $n_0 \approx 4 \times 10^9$ and similar to our reasoning for OpenMP, $c \approx 0.2$. $F(p)$ was observed to vary as $0.05\times$p. We notice that $F(p)$ has a smaller constant factor than the OpenMP implementation since process startup time was ignored.

5 Future Work

We observe that the MPI implementation of Prefix Sum is not the most work-optimal solution amongst the two implementations. The algorithm was chosen mainly because of the difficulty in implementing the Balanced scan approach for processing nodes $\neq$ powers of two. The OpenMP implementation avoids this problem through clever use of shared memory. In future work, the MPI implementation can be improved as well.

6 Conclusion

Three Prefix sum algorithms were implemented - Sequential, Parallel OpenMP and Parallel MPI. Each of these three implementation used a different algorithm and the performance across them were compared. The source code for all implementation as well as experimental results are available at github.com/arunxls/PrefixSum.

6.1 Sequential

In terms of program complexity, the sequential algorithm was the simplest. However, the running time is proportional to input size and quickly degrades in performance in comparison to both the parallel implementations.

6.1.1 MPI

It was observed that the MPI version scaled best and had the most optimal solution. However, the MPI solution did not include process start up time in this analysis. This could prove to be a significant overhead and might influence a user's preference in choosing a particular implementation. MPI also works well across a distributed compute cluster, something which neither of the other two implementation can do. However, one major disadvantage of using MPI is the need to re-write a significant portion of the program since we don't have the luxury of shared memory. This could prove to a big deciding factor when choosing a particular implementation.

6.1.2 OpenMP

The OpenMP version also gave significant improvements in performance as compared to the sequential version. However, in comparison to the MPI version, we note that the performance does not scale as well. This can be explained by the fact that the OpenMP implementation included in its timing analysis the time taken to start and stop threads, which turns out to be a significant overhead. When this overhead was ignored, it was noticed that the performance scaled much better. Another disadvantage of OpenMP is that it is limited to a single processing node since it relies on shared memory. For large inputs and complex programs, OpenMP might not be the best implementation to choose. One advantage, however, of OpenMP over MPI is in its simplicity. A given sequential program can be much more easily converted to a parallel version using OpenMP.