Dhirubhai Ambani Institute of Information and Communication Technology

CS-301: High Performance Computing

Parallel Prefix-Sum Algorithm

April 25, 2020

Assigned by:
Prof. Bhaskar Chaudhury

Submitted by:

Raj Patel (201701422)
Shubh Desai (201701466)

Contents

1 Introduction

In this project we have implemented Prefix-Sum Algorithm and tried to optimize it parallelly. In computer science, Prefix-Sum of a sequence of number $x_0, x_1, x_2, ...$ is a second sequence of numbers $y_0, y_1, y_2, ...$ such that,
$y_0 = x_0$
$y_1 = y_0 + x_1$
$y_2 = y_1 + x_2$

input numbers	1	2	3	4	5	6	...
prefix sums	1	3	6	10	15	21	...

Figure 1: Prefix-Sum Algorithm

Prefix sums are trivial to compute in sequential models of computation, by using the formula $\boldsymbol{y_i = y_{i-1} + x_i}$ to compute each output value in sequence order. However, despite their ease of computation, prefix sums are a useful primitive in certain algorithms such as counting sort, and they form the basis of the scan higher-order function in functional programming languages.

In functional programming terms, the Prefix-Sum can be generalized to any binary operation(not just the addition operation); the higher order function resulting from this generalization is called a scan, and it is closely related to the fold operation. Both operations differ in that the scan returns the whole sequence of results from the binary operation, whereas the fold returns only the final result. For instance, the sequence of factorial numbers may be generated by a scan of the natural numbers using multiplication instead of addition and fold of the natural number returns the factorial of that particular number.

Prefix-Sum Algorithm has many applications, some of them are listed below.

2 Applications of Prefix-Sum Algorithm:

- To evaluate polynomials.
- To solve recurrences.
- To implement Quick Sort.
- Counting sort is an integer sorting algorithm that uses the prefix sum of a histogram of key frequencies to calculate the position of each key in the sorted output array. It runs in linear time for integer keys that are smaller than the number of items, and is frequently used as part of Radix Sort, a fast algorithm for sorting integers that are less restricted in magnitude.
- To solve tridiagonal linear systems
- To dynamically allocate processors
- To perform lexical analysis. For example, to parse a program into tokens.
- To search for regular expressions. For example, to implement the UNIX grep program.
- To implement some tree operations. For example, to find the depth of every vertex in a tree

3 Serial Algorithm and Complexity Analysis

3.1 Algorithm

1. n = Problem-Size
2. Sum = 0
3. for i = 1 to n do
4. sum = sum + X[i]
5. X[i] = Sum
6. end for

3.2 Complexity Analysis

- Implementing a sequential version of Prefix-Sum computation (that could be run in a single thread on a CPU, for example) is as shown above. We simply loop over all the elements in the input array and add the value of the previous element of the input array to the sum computed for the previous element of the output array, and write the sum to the current element of the output array.
- This code performs exactly n adds for an array of length n; this is the minimum number of adds required to produce the scanned array. When we develop our parallel version of scan, we would like it to be work-efficient. A parallel computation is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version.
- Since each element must be accessed once, the complexity of this algorithm is **O(n)**.

4 Scope of Parallelism and Parallelization Strategies

4.1 Scope of Parallelization

- Unlike many other algorithms, parallelization of Prefix-Sum algorithm is different. Many other algorithms' performance can be improved by sharing the sequential work to multiple threads.
- In Prefix-Sum algorithm to compute the nth element, the previous n-1 elements must be calculated in advance. Therefore even if we have m processors to compute the Prefix-Sum of m elements, the performance of sequential algorithm can not be improved.
- Because of this reason, to compute Prefix-Sum parallelly, we need to implement a different algorithm such that more than one threads can work simultaneously and the overall executing time for computing Prefix-Sum reduces.

4.2 Parallelization Strategies

As discussed above, to calculate Prefix-Sum parallelly, a different algorithm needs to be implemented. Some of the strategies are described below:

4.2.1 Strategy-1

We have assumed that there are n array elements and m number of processors.

- **Pseudo-Code:**

1. n = Problem-Size
2. for i = 0 to $\lceil \boldsymbol{log_2 n} \rceil - \mathbf{1}$ do
3. for j = n-1 to 0 do in parallel
4. if $\boldsymbol{j < 2^i}$ then
5. $\boldsymbol{x_j^{i+1} = x_j^i}$
6. else
7. $\boldsymbol{x_j^{i+1} = x_j^i + x_{j-2^i}^i}$
8. end for
9. end for

- **Analysis and Explanation of the Algorithm:**

1. In this algorithm, the outer loop runs for $\boldsymbol{log_2 n}$ times, and the inner loop runs for n number of times, where n is the number of elements of the array.
2. If we have n processors, then we can distribute the work of the inner loop simultaneously to each processors, and thus can compute that particular step in one time unit.
3. Now, this process has to be repeated $\boldsymbol{log_2 n}$ number of times, and since we have assumed total number of processors to be n, we can calculate the Prefix-Sum in $\boldsymbol{O(log_2 n)}$ time.
4. In this algorithm, after k steps(outer loop), first $\mathbf{2^{k+1}}$ elements are calculated correctly.
5. Instead of n processors, if we have m processors(m $<$ n), the complexity of the algorithm is: $\boldsymbol{O(\frac{nlog_2 n}{m})}$.
6. Thus the algorithm is efficient if $\boldsymbol{(\frac{nlog_2 n}{m}) < n}$

7. Another optimization can be applied to the algorithm by exploiting the fact that after k operations $\mathbf{2^{k+1}}$ elements are correct. So after k operations, instead of running the loop for all n elements we can modify it to run for only $\frac{n-2^{k+1}}{m}$ times.

- **Pictorial Representation:**

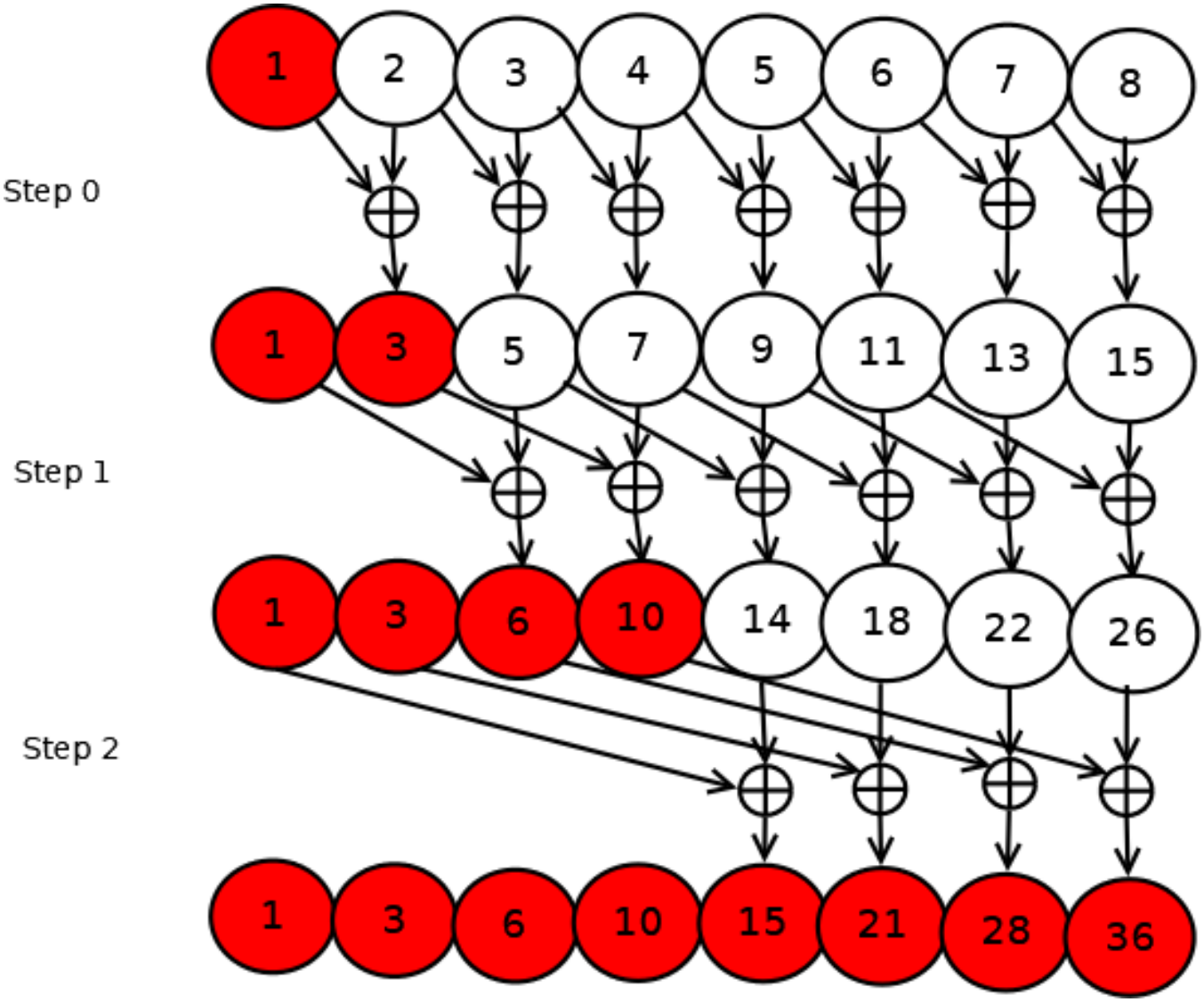

Figure 2: Algorithm - 1 Pictorial Representation

4.2.2 Strategy-2

We have assumed that there are n array elements and m number of processors.

- **Pseudo-Code:**

1. Divide the array into chunks of $\frac{n}{m}$ elements.
2. Compute the Prefix-Sum of these m sub-arrays.

3. Create a temporary array containing the last elements of all the sub-arrays except the last sub-array.
4. Now compute the Prefix-Sum of this sub-array.
5. The i-th element from this temporary array is added to all the elements of the (i+1)th sub-array.

- **Analysis and Explanation of the Algorithm:**
  1. In the second step of the algorithm, the Prefix-Sum of m sub-arrays is computed in $\boldsymbol{O(\frac{n}{m})}$ time.
  2. In the fourth step of the algorithm, as there are total m chunks, this step will be computed in O(m) time.
  3. The fifth step of the algorithm will be computed in $\boldsymbol{O(\frac{n}{m})}$ time.
  4. The overall time complexity of the algorithm is $\boldsymbol{O(\frac{2n}{m} + m)}$. In the case in which n is various order of magnitude greater than m, the time spent in computing intermediate prefix sum is negligible and in that case the complexity becomes $\boldsymbol{O(\frac{2n}{m})}$.
- **Pictorial Representation: (N = 9, M = 3)**

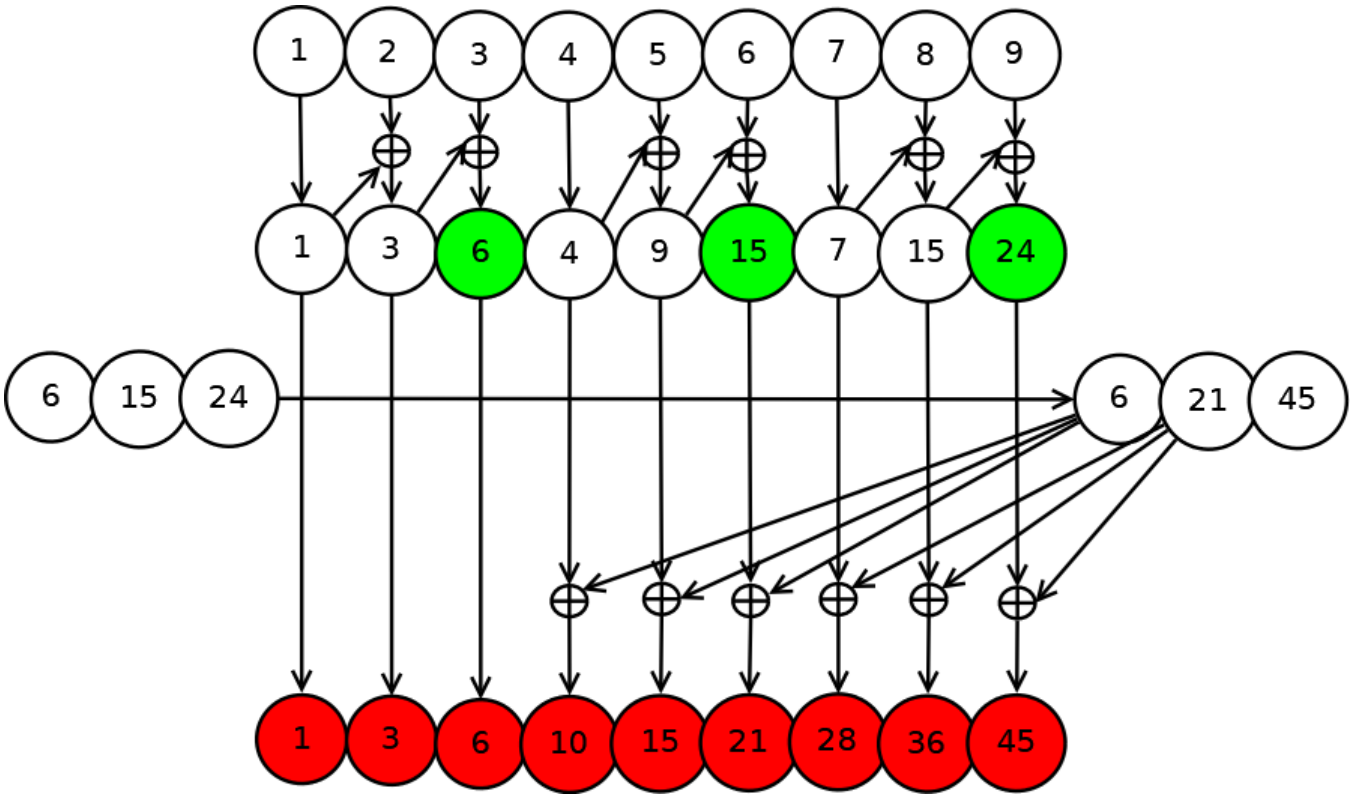

Figure 3: Algorithm - 2 Pictorial Representation

4.2.3 Strategy-3

We have assumed that there are n array elements and m number of processors.

- **Pseudo-Code:**
  1. Divide the array into 2m sub-arrays. Thus each sub-array contains $\frac{n}{2m}$ elements.
  2. Calculate the Prefix-Sum of these sub-arrays. In the first iteration, each processor k computes the Prefix-Sum of (2k-1)-th sub-array. In the second iteration, each processor k computes the Prefix-Sum of (2k)-th sub-array from the previously computed Prefix-Sum of (2k-1)th sub-array. Thus, after the whole process, Prefix-Sum of first two sub-arrays are correctly computed.
  3. Then, we compute the cumulative sum between all the elements in one sub-array and the last element of the all previous sub-arrays. (This step becomes more clear from the below pictorial iterations.)
- **Analysis and Explanation of the Algorithm:**
  1. In the second step of the pseudo-code, during the first iteration, the Prefix-Sum of half of the sub-arrays can be computed in $\boldsymbol{O(\frac{n}{2m})}$ time. This is same for the second iteration as well. Therefore overall time to execute this step is $\boldsymbol{O(\frac{n}{m})}$.
  2. In the third step of pseudo-code, the cumulative sum is calculated in $\boldsymbol{O(\frac{nlog_2m}{2m})}$ time.
  3. Therefore the overall time-complexity of the algorithm is: $\boldsymbol{O(\frac{n}{m} + \frac{nlog_2m}{2m})}$.
  4. This strategy is better than the first strategy if: $(\frac{n}{m} + \frac{nlog_2m}{2m}) < \frac{nlog_2n}{m}$.

- **Pictorial Representation: (N = 16, M = 4)**

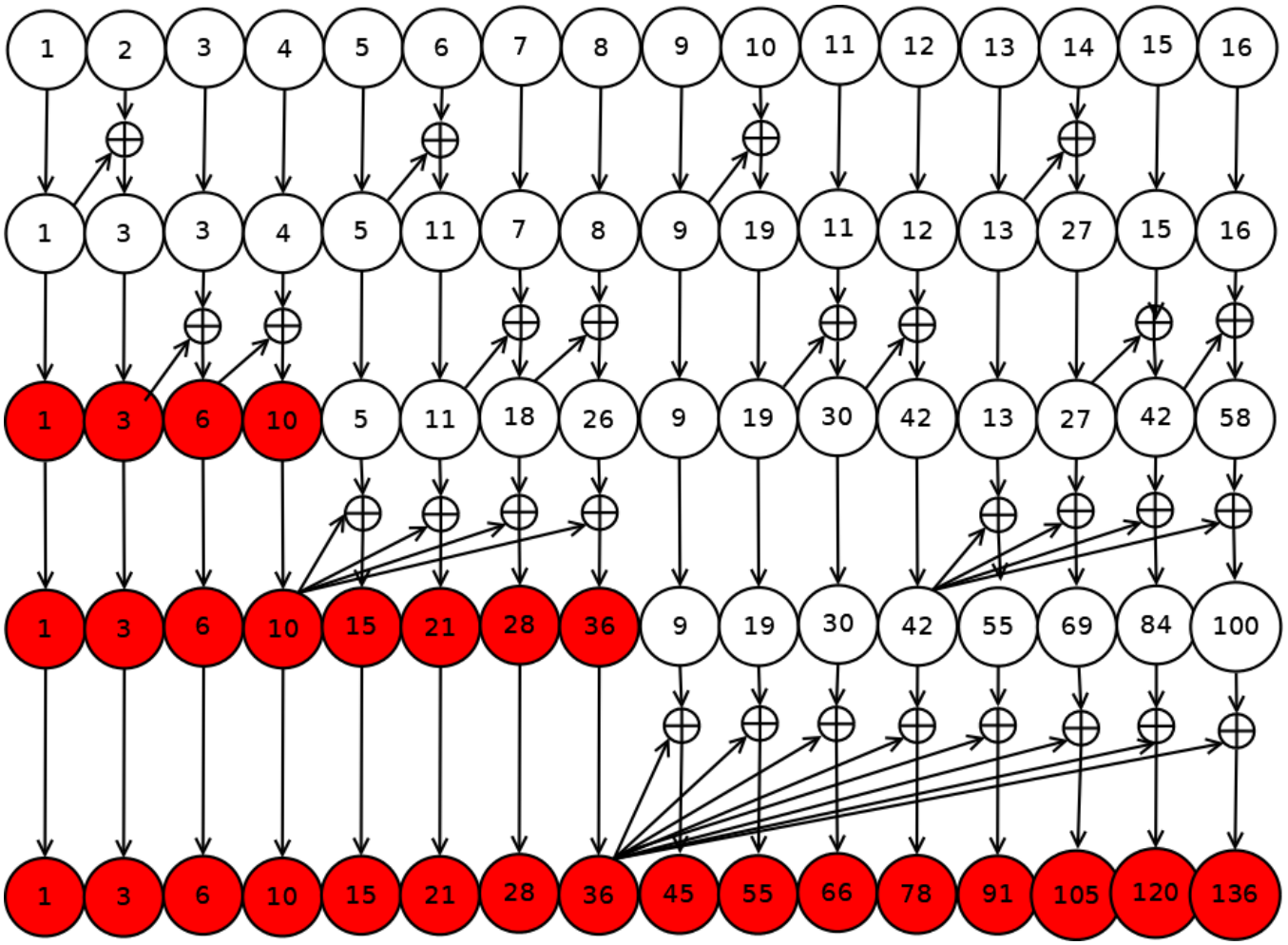

Figure 4: Algorithm - 3 Pictorial Representation

5 Graphical Analysis:

5.1 Strategy-1

- **Execution Time**
  As the problem-size increases, execution time increases. For smaller problem-size, execution time is also similar for all cases. As problem-size increases, execution with more number of threads gets quicker.

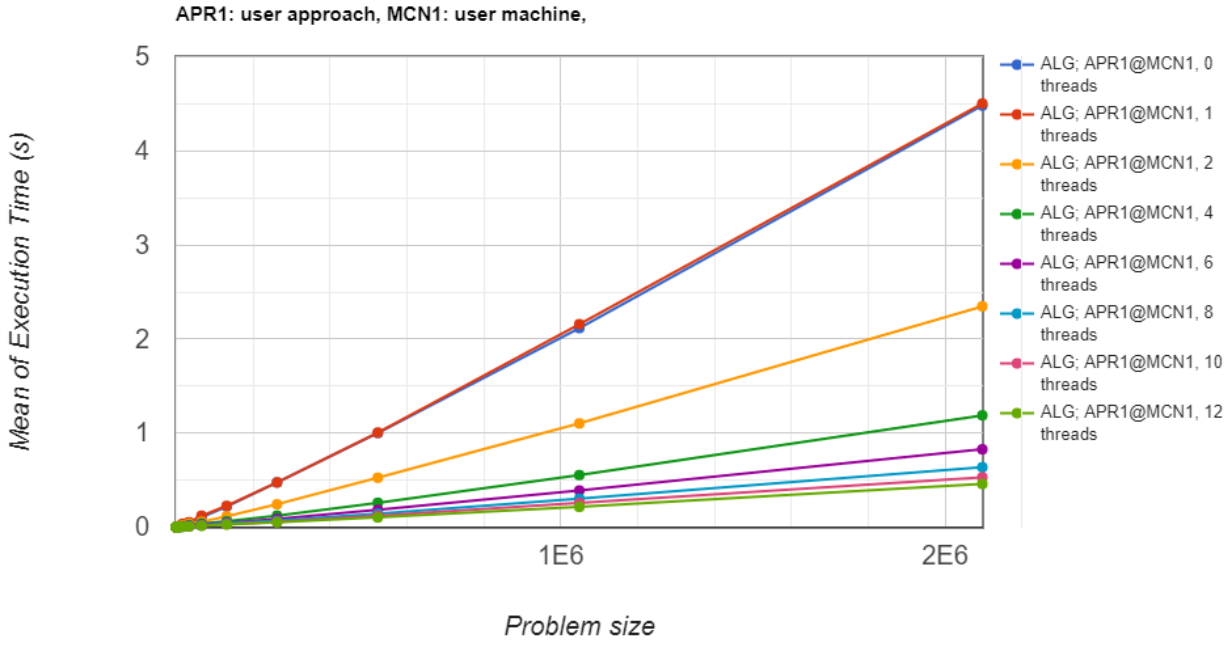

Figure 5: Execution Time-Strategy-1

- **Speedup**
  As number of cores increases, speedup increases.For smaller problem size, speedup is less than the steady value which is due to parallel overhead for smaller problem size. Achieved speedup is less than theoretical speedup, which can be Amdahl's Law.

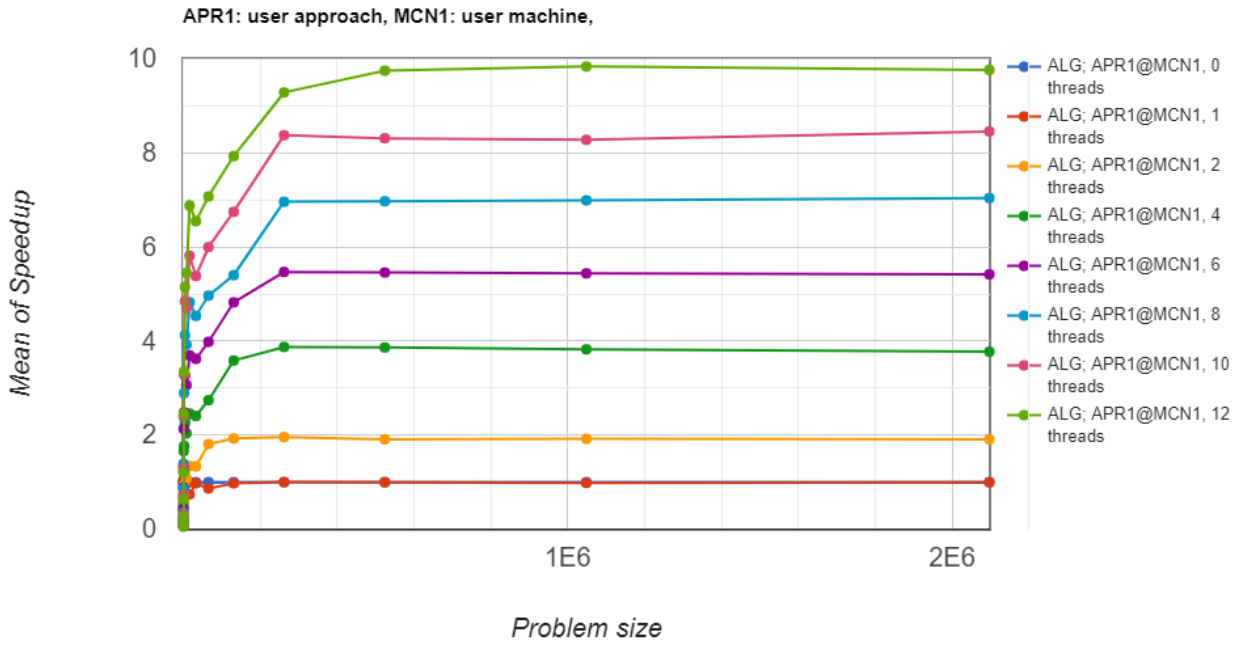

Figure 6: Speedup-Strategy-1

- **Efficiency**
  As problem size increases, efficiency increases and eventually it becomes steady. As number of cores increases, efficiency decreases.

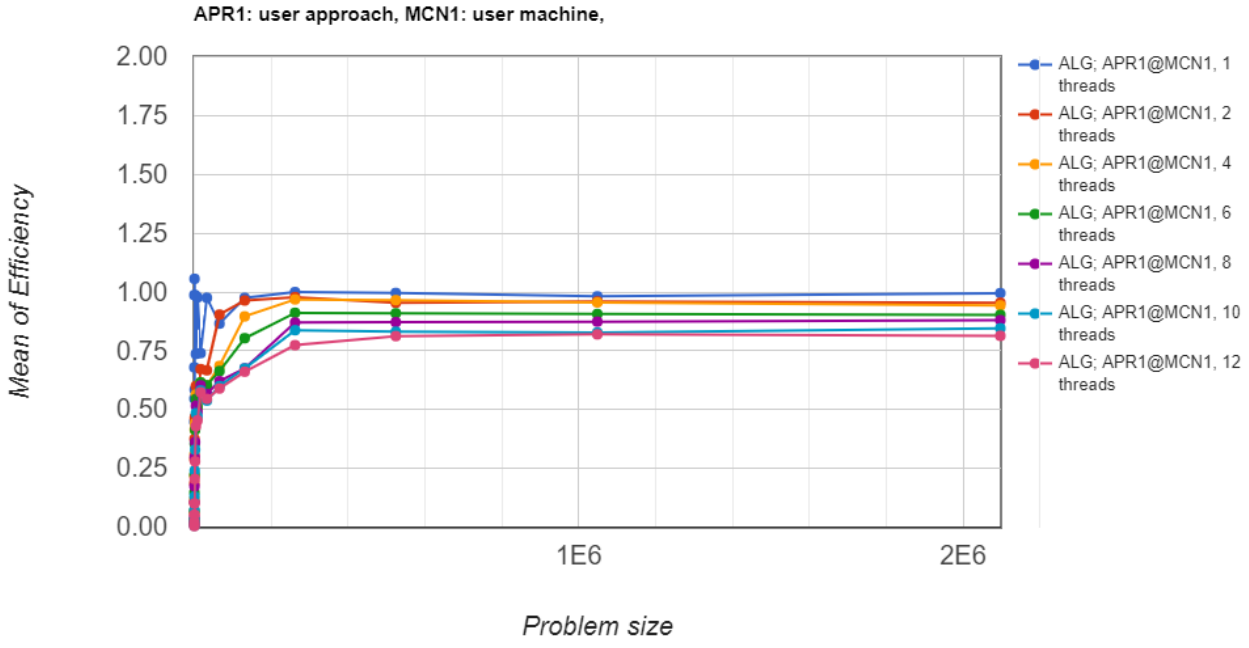

Figure 7: Efficiency-Strategy-1

5.2 Strategy-2

- **Execution Time**
  Overall this algorithm is better than the first algorithm as comparatively the execution time less than the first algorithm. For single processor, for some problem size, execution time is slightly more than serial algorithm because of parallel overhead. Apart from this, the observations for first algorithm is also applicable here.

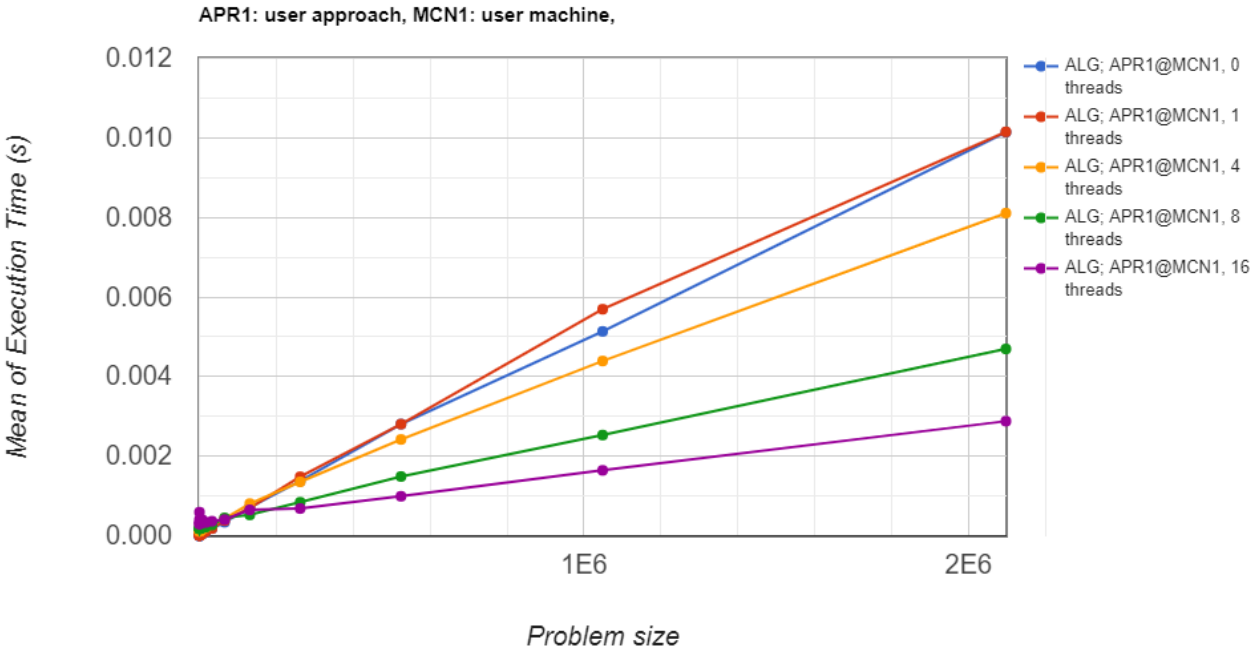

Figure 8: Execution Time-Strategy-2

- **Speedup**
  As number of cores increases, speedup increases.For smaller problem size, speedup is less than the steady value which is due to parallel overhead for smaller problem size. Achieved speedup is less than theoretical speedup, which can be Amdahl's Law. In comparison to first strategy, speedup is less, which is because of cache inefficiency and more memory access.

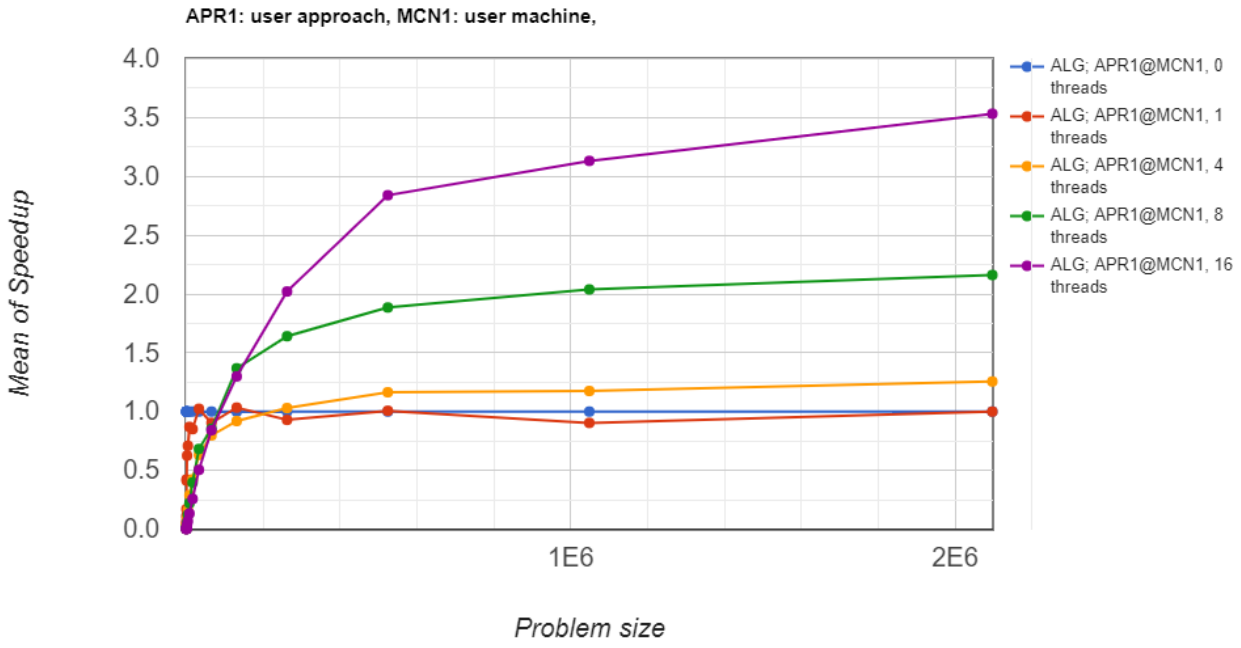

Figure 9: Speedup-Strategy-2

- **Efficiency**
  As problem size increases, efficiency increases and eventually it becomes steady. As number of cores increases, efficiency decreases. Comparatively it is less efficient than the first algorithm.

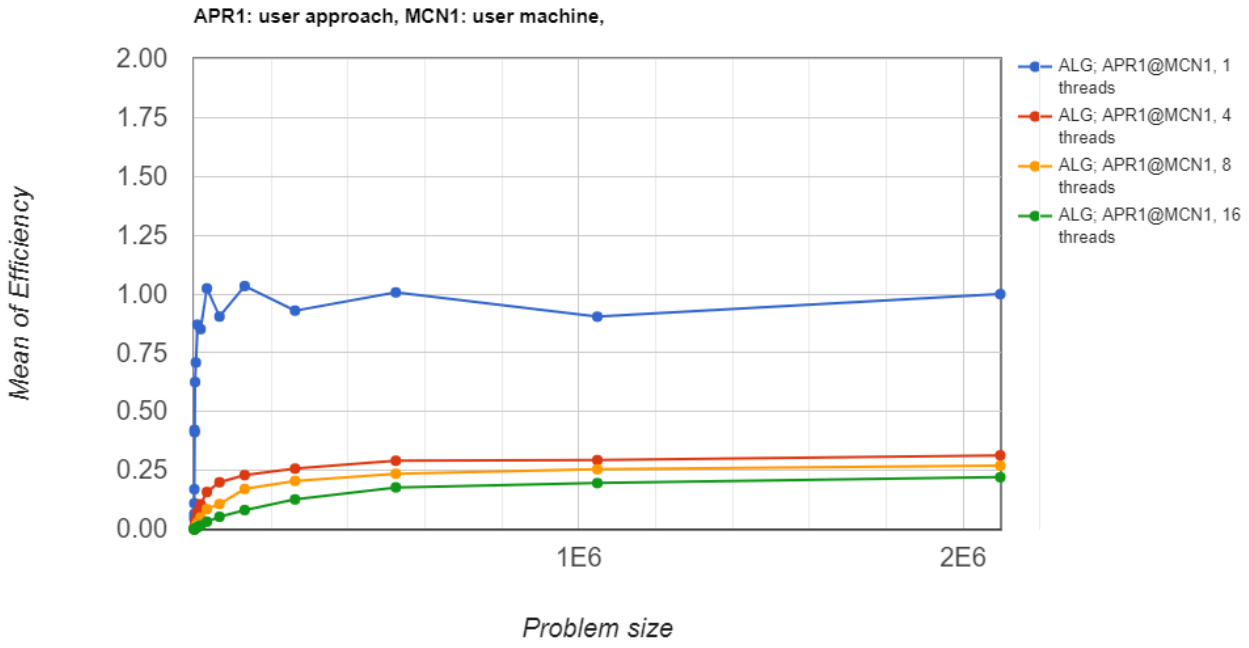

Figure 10: Efficiency-Strategy-2

6 Further Possible Analysis

- Till now, we have implemented the first parallel algorithm and included its graphical analysis. We are planning to implement the other two strategies as well because in several cases, the second and third strategies have better performance.
- As described in Section-2, there are many applications of Prefix-Sum algorithm. We can implement some of them, by implementing parallel Prefix-Sum algorithm in any of its application, we can also improve the performance of that particular problem.
- We have not taken into account cache-efficiency till now. In future, we can modify the algorithms such that it also becomes cache-efficient.

7 References

1. https://en.wikipedia.org/wiki/Prefixsum
2. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda