

# COMS3008A Assignment – Report

Claudio Da Mata - 2128358

23-10-2022

## 1 Problem 1: Parallel Scan

- Given a set of elements,  $[a_0, a_1, \dots, a_{n-1}]$ , the scan operation associated with addition operator for this input is the output set  $[a_0, (a_0 + a_1), \dots, (a_0 + a_1 + \dots + a_{n-1})]$ .
- For example, the input set is  $[2, 1, 4, 0, 3, 7, 6, 3]$ , then the scan with addition operator of this input is  $[2, 3, 7, 7, 10, 17, 23, 26]$ .

### 1.1 Serial Implementation:

Firstly I started off with the baseline implementation of serial scan operation given in Listing 1

```
1 void scan(int out[], int in[], int N){
2     out[0] = in[0];
3
4     for(int i=1; i<N; i++) {
5         out[i] = in[i] + out[i-1];
6     }
7 }
```

Listing 1: Sequential algorithm for computing scan operation with '+' operator<sup>1</sup>.

This got me a baseline average time to measure and compare parallel implementations. I ran the operation (10 times, taking average) on arrays ranging in sizes from 1000 to 50,000 with intervals of 1000 given in Figure 1

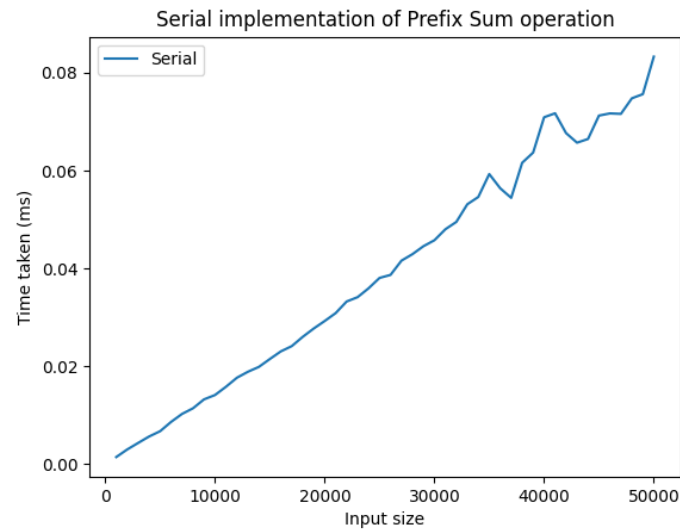


Figure 1: Serial scan operation average times

## 1.2 OpenMP Parallel Implementation:

Next I implented the scan operation within OpenMP's interface. The algorithm I used is based off the Blelloch<sup>2</sup> work-effecient algorithm given in Listing 2

```

1  void scan(int out[], int in[], int N){
2      int nthr, *z, *x = out;
3      #pragma omp parallel num_threads(4)
4      {
5          int i;
6          #pragma omp single
7          {
8              nthr = omp_get_num_threads();
9              z = malloc(sizeof(int)*nthr+1);
10             z[0] = 0;
11         }

12
13         int tid = omp_get_thread_num();
14         int sum = 0;
15         #pragma omp for schedule(static)
16         for(i=0; i<N; i++) {
17             sum += in[i];
18             x[i] = sum;
19         }

20
21         z[tid+1] = sum;
22         #pragma omp barrier

```

```

24     int offset = 0;
25     for(i=0; i<(tid+1); i++) {
26         offset += z[i];
27     }

29     #pragma omp for schedule(static)
30     for(i=0; i<N; i++){
31         x[i] += offset;
32     }
33 }
34 free(z);
35 }

```

Listing 2: OpenMP Parallel algorithm for computing scan operation<sup>3</sup>.

The algorithm works as follows:

- Start omp parallel
- Initiate a 'single' construct which allows only one thread to run the code
- Inside single should declare the size of the temp array and set  $element[0] = 0$
- After the single construct, get current threadID and initialize  $sum = 0$
- Begin an omp for construct with schedule(static)
- Inside the for loop (from 0 to  $N - 1$ ),  $sum +=$  the input array at  $i$ , and set the output array at  $i$  equal to the new  $sum$
- After the for loop, set temp array at  $[threadID + 1]$  equal to  $sum$
- Declare a Barrier construct which forces all threads to wait until other threads finish computation
- Set  $offset = 0$
- Begin a regular for loop that sums all elements of the temp array to  $offset$ , only adding elements that each specific thread has computed itself
- Begin an omp for construct with schedule(static) again, from  $i = 0$  to  $N - 1$
- sum output array element  $i$  with offset
- Finish off pragma omp and then free the temp array from memory

I experimented with different number of threads running and managed to narrow it down to thread counts of 2, 4, 6 given in Figure 2

With 2 threads running there is no noticeable speedup compared to serial implementation. With 4 and 6 threads running there is significant speedup compared to serial implementation. While input size  $n$  increases, 4 threads performs slightly better than 6 threads. Thus I choose 4 threads as my optimal implementation of OpenMP parallelization.

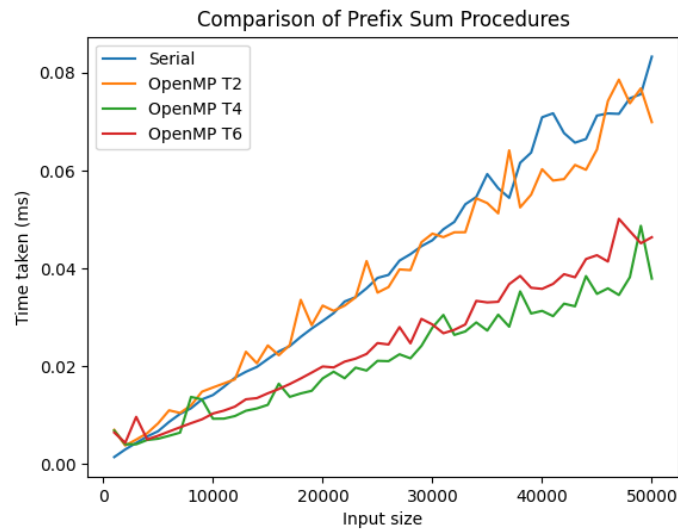


Figure 2: OpenMP Parallel scan operation average times for different thread counts

### 1.3 MPI Parallel Implementation:

Next I implemented the scan operation within MPI's interface. The algorithm I used is based off the Blelloch<sup>2</sup> prescan algorithm given in Listing 3

```

1  MPI_Init(&argc, &argv); //Initialize MPI
3  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
4  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

6  MPI_Barrier(MPI_COMM_WORLD);

8  size_t num_per_proc = n / comm_size;
9  int sum;

11 start_find_sum(my_rank, comm_size, data, num_per_proc,
12 &sum);
13 start_find_psum(my_rank, comm_size, data, num_per_proc,
14 sum);

16 MPI_Barrier(MPI_COMM_WORLD);

18 MPI_Finalize(); //Close MPI

```

Listing 3: MPI Parallel algorithm for computing scan operation<sup>4</sup>.

The essence of the algorithm comes from the `start_find_sum()` and `start_find_psum()` functions.

The algorithm works as follows:

- Start MPI
- Get comm rank and size
- Barrier that forces all threads to wait
- Set num per proc equal to array size  $n/commsize$ . This dictates how many numbers each thread will compute the sums of
- Call start\_find\_sum() function
  - The algorithm works like a binary tree with levels
  - Gets MPI\_Status
  - Sums input array elements from 0 to *numPerProc*
  - Begin for loop from  $level = 0$  to  $\log_2(commsize)$
  - Inside the loop set  $position = rank/level^2$
  - If *position* is even, receives the *sendersSum* (with  $senderRank = rank + level^2$ ) and adds to total *sum*
  - If *position* is odd, sends total *sum* to receiver with  $senderRank = rank - level^2$ , then kills current thread
  - Before for loop ends, calls Barrier so all threads wait
- Call start\_find\_psum() function
  - The algorithm works like a binary tree with levels
  - Gets MPI\_Status
  - If  $rank == 0$ , sets  $psum = sum$
  - Begin for loop from  $level = \log_2(commsize) - 1$  down to 0
  - If process is on current level:
  - Set  $position = rank/level^2$
  - If *position* is even this means this process was the parent of *sendingRank*, and so first sets  $senderRank = rank + level^2$ , then sends *psum*, then receives the *senderSum* and finally subtracts *psum* by *senderSum*
  - If *position* is odd, sets  $receivingRank = rank - level^2$ , then receives *psum* from parent, and then sends *sum* with *receivingRank*
  - Before for loop ends, calls Barrier so all threads wait
  - After the loop ends, put the *prefixSums* associated with this process in the input array
- Calls Barrier so all threads wait
- Ends MPI

With start\_find\_sum() and start\_find\_psum() functions given respectively by Listing 4 and Listing 5

```

1 void start_find_sum(int rank, int mysize, int* in,
2                     size_t num_per_proc, int* overall_sum){
3     MPI_Status status;

4
5     int sum = find_sum(in, num_per_proc);

6
7     int still_alive = 1;
8     int level;

9
10    for (level = 0; level < (int)log2(mysize); level++) {
11        if (still_alive) {
12            int position = rank / (int)pow(2, level);

13
14            if (position % 2 == 0) {
15                // I am a receiver
16                int sender_sum;
17                int sending_rank = rank + (int)pow(2, level);

18
19                MPI_Recv(&sender_sum, 1, MPI_INT, sending_rank,
20                        0, MPI_COMM_WORLD, &status);

21
22                sum += sender_sum;
23            }else {
24                // I am a sender
25                int receiving_rank = rank - (int)pow(2, level);

26
27                MPI_Send(&sum, 1, MPI_INT, receiving_rank, 0,
28                        MPI_COMM_WORLD);
29                still_alive = 0;
30            }
31        }
32        MPI_Barrier(MPI_COMM_WORLD);
33    }
34    *overall_sum = sum;
35 }

```

Listing 4: start\_find\_sum()

```

1 void start_find_psum(int rank, int mysize, int* in,
2                     size_t num_per_proc, int sum){
3     int psum;
4     int level;
5     MPI_Status status;

6
7     if (rank == 0) {
8         psum = sum;
9     }
10    for (level = (int)log2(mysize) - 1; level >= 0; level--) {
11        // only trigger the processes on the current level

```

```

12     if (level == 0 || rank % (int)pow(2, level) == 0) {
13         int position = rank / (int)pow(2, level);

15         if (position % 2 == 0) {
16             int sender_sum;
17             int sending_rank = rank + (int)pow(2, level);

19             MPI_Send(&psum, 1, MPI_INT,
20                     sending_rank, // RIGHT CHILD
21                     0, MPI_COMM_WORLD);

23             MPI_Recv(&sender_sum, 1, MPI_INT,
24                      sending_rank, 0, MPI_COMM_WORLD, &status);

26             // psum <- (prefix sum of parent) - (sum of sibling)
27             psum -= sender_sum;
28         }else{
29             int receiving_rank = rank - (int)pow(2, level);

31             MPI_Recv(&psum, 1, MPI_INT,
32                     receiving_rank, // PARENT
33                     0, MPI_COMM_WORLD, &status);

35             // send sum to receiving_rank so it can fix its sum
36             MPI_Send(&sum, 1, MPI_INT,
37                     receiving_rank, 0, MPI_COMM_WORLD);
38         }
39     }
40     MPI_Barrier(MPI_COMM_WORLD);
41 }
42 // put the prefix sums associated with this node in input array
43 int next_sum = in[num_per_proc-1];
44 in[num_per_proc-1] = psum;

46 for (int j = num_per_proc - 2; j >= 0; j--) {
47     int next_sum_tmp = in[j];

49     in[j] = in[j+1] - next_sum;

51     next_sum = next_sum_tmp;
52 }
53 }

```

Listing 5: start\_find\_psum()

I experimented with different number of threads running and managed to narrow it down to thread counts of 2,4,8 given in Figure 3

With 2 threads running there is noticeably worse running time compared to serial implementation. With 4 and 8 threads running there is significant speedup compared to serial implementation. With smaller input sizes  $n$ , 4 threads performs

better than 8 threads, but as  $n$  increases, performance between 4 threads and 8 threads stays relatively the same. Thus I choose 4 threads as my optimal implementation of MPI parallelization.

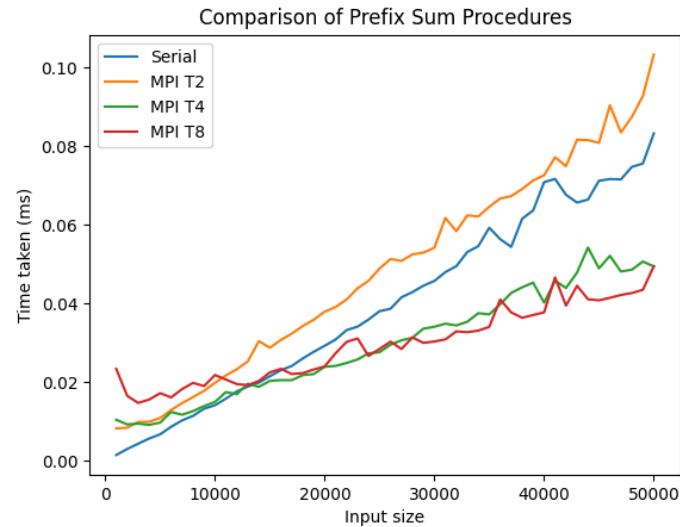


Figure 3: MPI Parallel scan operation average times for different thread counts

## 1.4 Validation:

Checking if elements are correctly summed, through running *run.sh* given in Figure 4

```
-----Serial scan is starting-----
Validate passed.
-----Serial scan is done-----

-----OpenMP scan is starting-----
Validate passed.
-----OpenMP scan is done-----

-----MPI scan is starting-----
Validate passed.
-----MPI scan is done-----
```

Figure 4: Validation of scan operations



## 1.5 Conclusions:

After comparing the results from serial, optimal OpenMP (with 4 threads) and optimal MPI (with 4 threads), I can make the conclusion that serial performs the best at very small array size  $n$ . While at large  $n$ , both OpenMP and MPI have significant speedup from serial, OpenMP performs consistently better than MPI. Comparisons given in Figure 5

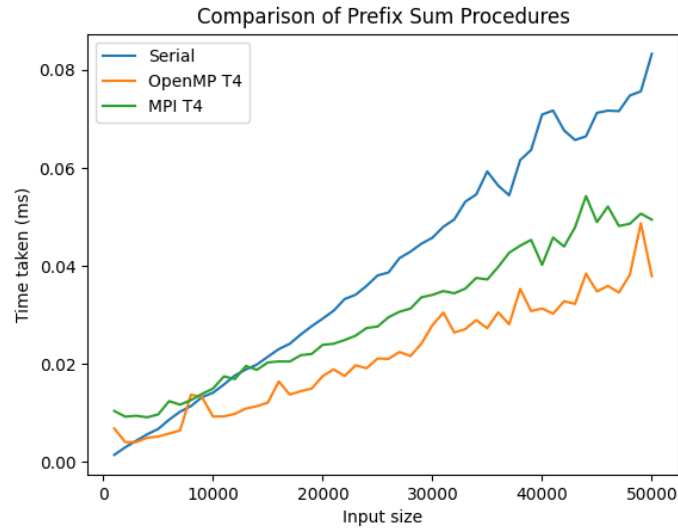


Figure 5: Comparison of all scan operations

## 2 Problem 2: Parallel Bitonic Sort

Not applicable, working alone.

## 3 Problem 3: Parallel Graph Algorithm

Implementing Dijkstra's Single Source Shortest Path (SSSP) Algorithm. An example problem is given in Figure 6.

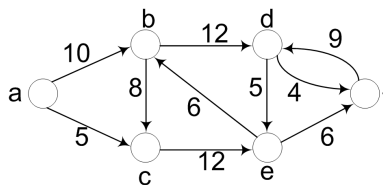


Figure 6: A directed graph

### 3.1 Serial Implementation:

Firstly I started off with the baseline implementation of serial SSSP algorithm given in Listing 6

```

1  int* dijkstra(int **graph, int src){
2      int* dist = (int*)malloc(V * sizeof(int));
3      bool sptSet[V];

5      for (int i = 0; i < V; i++) dist[i] = INT_MAX, sptSet[i] = false;
6      dist[src] = 0;

8      // Find shortest path for all vertices
9      for (int count = 0; count < V - 1; count++) {
10         int u = minDistance(dist, sptSet);
11         sptSet[u] = true;

13         for (int v = 0; v < V; v++)
14             if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
15                 && dist[u] + graph[u][v] < dist[v])
16                 dist[v] = dist[u] + graph[u][v];
17     }
18     return dist;
19 }
```

Listing 6: Sequential algorithm for Dijkstra's SSSP<sup>5</sup>.

This got me a baseline average time to measure and compare parallel implementations. I ran the algorithm on the given graphs (15 times, taking average) with number of vertices ranging from 6 to 512 given in Figure 7

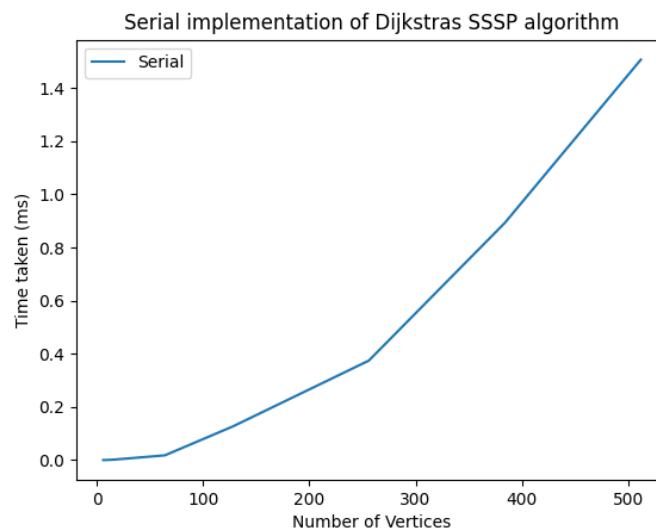


Figure 7: Serial SSSP algorithm average times

### 3.2 OpenMP Parallel Implementation:

Next I implented the SSSP algorithm within OpenMP's interface. The approach I used is based off Dijkstra's SSSP iterative approach, while finding the minimum distance for each node gets assigned a thread to split the work done<sup>6</sup> given in Listing 2

```

1  int* dijkstra(int **graph, int src){
2      int* dist = (int*)malloc(V * sizeof(int));
3      dist[src] = 0; // Distance of source vertex from itself is always 0
4      int i, md, mv;
5      int my_first; // The first vertex that stores in one thread locally.
6      int my_id; // ID for threads
7      int my_last; //The last vertex that stores in one thread locally.
8      int my_md; // local minimum distance
9      int my_mv; // local minimum vertex
10     int my_step; /* local vertex that is at the minimum distance from the
11                  source */
12     int nth; /* number of threads */
13     bool sptSet[V]; // sptSet[i] will be true if vertex i has been
14                  visited
15
16     // Initialize all distances as INFINITE and stpSet[] as false
17     for (i = 0; i < V; i++)
18         dist[i] = INT_MAX, sptSet[i] = false;
19
20     /* OpenMP parallelization starts here */
21     #pragma omp parallel private ( my_first, my_id, my_last, my_md, my_mv
22         , my_step )\
23     shared ( src, md, dist, mv, nth, graph )
24     {
25         my_id = omp_get_thread_num ( );
26         nth = omp_get_num_threads ( );
27         my_first = (my_id * V) / nth;
28         my_last = ((my_id + 1) * V) / nth - 1;
29         for (my_step = 0; my_step < V-1; my_step++) {
30             #pragma omp single
31             {
32                 md = INT_MAX;
33                 mv = -1;
34             }
35             int k;
36             my_md = INT_MAX;
37             my_mv = -1;
38
39             /* Each thread finds the minimum distance unconnected vertex
40              inner of the graph */
41             for (k = my_first; k <= my_last; k++) {
42                 if (!sptSet[k] && dist[k] <= my_md) {
43                     my_md = dist[k];

```

```

40         my_mv = k;
41     }
42 }

44     /* 'critical' specifies that code is only be executed on
45     * one thread at a time, because we need to determine the
46     * minimum of all the my_md here. */
47     #pragma omp critical
48     {
49         if (my_md < md) {
50             md = my_md;
51             mv = my_mv;
52         }
53     }

55     /* 'barrier' identifies a synchronization point at which threads
56     in a
57     * parallel region will wait until all other threads in this
58     section reach
59     * the same point. So that md and mv have the correct value. */
60     #pragma omp barrier

61     #pragma omp single
62     {
63         /* It means we find the vertex and set its status to true. */
64         if (mv != - 1){
65             sptSet[mv] = true;
66         }
67     }

68     # pragma omp barrier

69     if ( mv != -1 ){
70         int j;
71         for (j = my_first; j <= my_last; j++) {
72             if (!sptSet[j] && graph[mv][j] && dist[mv] != INT_MAX
73                 && dist[mv] + graph[mv][j] < dist[j])
74                 dist[j] = dist[mv] + graph[mv][j];
75         }
76     }
77     #pragma omp barrier
78 }
79 }
80 }
81 return dist;
82 }

```

Listing 7: OpenMP Parallel algorithm for computing SSSP<sup>6</sup>

The algorithm works as follows:

- Initialize all variables
- Start omp parallel
- Private variables:  $my\_first, my\_id, my\_last, my\_md, my\_mv, my\_step$
- Shared variables:  $src, md, dist, mv, nth, graph$
- Set:  $my\_id$  to current thread ID;  $nth$  to total thread count;  $my\_first$  to  $(my\_id * V) / nth$ ; and  $my\_last$  to  $((my\_id + 1) * V) / nth - 1$ ;
- Begin a for loop (from 0 to  $V - 1$ )
- Begin an omp single construct, set  $md$  to an arbitrarily large number and  $mv$  to non-existent ( $-1$ )
- Now set  $my\_md$  and  $my\_mv$  in the same way
- Begin a for loop from  $my\_first$  to (and including)  $my\_last$
- Inside the for loop, check 2 things: if node has been visited and if distance of node ( $dist[k]$ ) is less than local minimum distance ( $my\_md$ )
- If true: set  $my\_md = dist[k]$  and  $my\_mv = k$
- Begin an omp critical construct, inside check if  $my\_md < md$ . If true:  $md = my\_md$  and  $mv = my\_mv$
- Initiate omp barrier
- Begin an omp single construct, if  $mv$  is not  $-1$  then set node  $mv$  to visited
- Initiate another omp barrier
- Check if  $mv$  is not  $-1$  then begin a for loop from  $my\_first$  to (and including)  $my\_last$
- Inside the loop check 3 things: node  $[j]$  not visited; cost at  $[mv][j]$  and  $dist[mv]$  are not the default large number we set; finally if  $dist[mv] + cost\ at\ [mv][j]$  are less than  $dist[j]$
- If all conditions are met, set  $dist[j] = dist[mv] + cost\ at\ [mv][j]$
- After the for loop initiate omp barrier
- Return the distance array

I experimented with different number of threads running and noticed any number of threads less than 6 produce similar results. While anything over 6 starts to slow down the algorithm rather than speed it up. From this conclusion I chose thread counts of 2, 3, 4 to compare, given in Figure 8

All 3 parallel implementations deviate very slightly from each other. In parallel speedup is only noticed at around 250 vertices and higher, otherwise serial performs better. With 3 threads running there is a slightly better speedup at higher vertex counts.

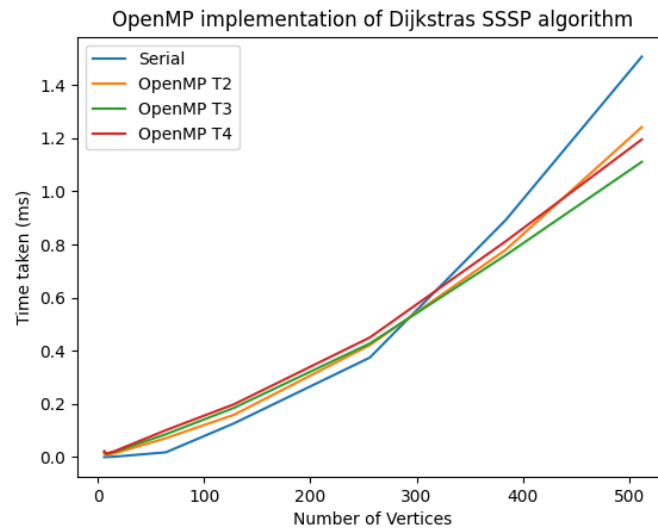


Figure 8: OpenMP Parallel SSSP algorithm average times for different thread counts

### 3.3 Conclusions:

An example of table is given Table 1.

Table 1: An example of a table					
No of vertices	64	128	256	384	512
Serial	0.1	0.2	0.3	0.4	0.5
Parallel					
Speedup	2	3	4	5	6

## References

- [1] Prefix Sum Array | implementation and applications in competitive programming, <https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/>, accessed: 23-10-2022.
- [2] Prefix Sum | algorithm 2: Work-efficient, [https://en.wikipedia.org/wiki/Prefix\\_sum](https://en.wikipedia.org/wiki/Prefix_sum), accessed: 23-10-2022.
- [3] CSE 231 Introduction to Parallel and Concurrent Programming at Washington University | scan, <https://classes.engineering.wustl.edu/cse231/core/index.php/Scan>, accessed: 23-10-2022.
- [4] MPI Parallel Prefix Sum (1A), <https://upload.wikimedia.org/wikiversity/en/2/2f/ParaPrefix.MPI.1.A.20140730.pdf>, accessed: 23-10-2022.
- [5] Dijkstra's Shortest Path Algorithm | greedy algo-7, <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>, accessed: 24-10-2022.
- [6] M. He, Parallelizing dijkstra's algorithm (2021).