COMS3008A Assignment – Report

Claudio Da Mata - 2128358

23-10-2022

1 Problem 1: Parallel Scan

- Given a set of elements, $[a_0, a_1, \cdots, a_{n-1}]$, the scan operation associated with addition operator for this input is the output set $[a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-1})]$.
- 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[1].

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

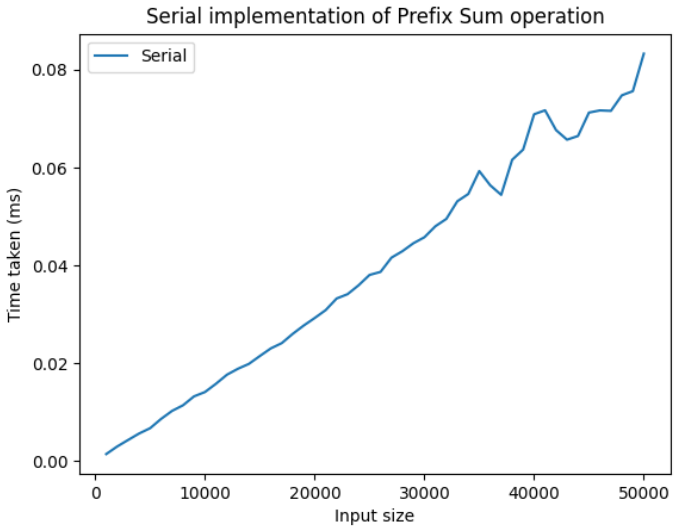

Figure 1: Serial scan operation average times

1.2 OpenMP Parallel Implementation:

Next I implented the scan operation within OpenMP's interface. The algorithm I used is based off the Blelloch[2] 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      }

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      }

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[3].

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 contruct 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.

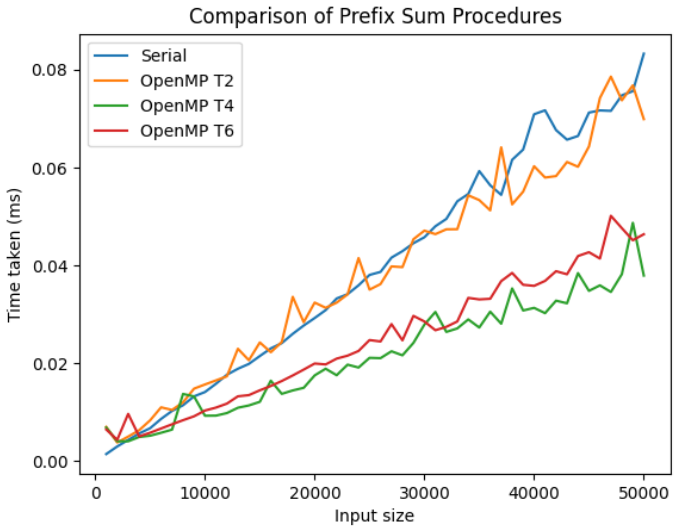

Figure 2: OpenMP Parallel scan operation average times for different thread counts

1.3 MPI Parallel Implementation:

Next I implented the scan operation within MPI's interface. The algorithm I used is based off the Blelloch[2] prescan algorithm given in Listing 3

```
    MPI_Init(&argc, &argv); //Initialize MPI

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    MPI_Barrier(MPI_COMM_WORLD);

    size_t num_per_proc = n / comm_size;
    int sum;

    start_find_sum(my_rank, comm_size, data, num_per_proc,
    &sum);
    start_find_psum(my_rank, comm_size, data, num_per_proc,
    sum);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize(); //Close MPI
```

Listing 3: MPI Parallel algorithm for computing scan operation[4].

The essence of the algorith 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 algoritem 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

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

  int sum = find_sum(in, num_per_proc);

  int still_alive = 1;
  int level;

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

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

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

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

        MPI_Send(&sum, 1, MPI_INT, receiving_rank, 0,
        MPI_COMM_WORLD);
        still_alive = 0;
      }
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }
  *overall_sum = sum;
}
```

Listing 4: start_find_sum()

```
void start_find_psum(int rank, int mysize, int* in,
          size_t num_per_proc, int sum){
  int psum;
  int level;
  MPI_Status status;

  if (rank == 0) {
    psum = sum;
  }
  for (level = (int)log2(mysize) - 1; level >= 0; level--) {
    // 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, perfermance between 4 threads and 8 threads stays relatively the same. Thus I choose 4 threads as my optimal implementation of MPI parallelization.

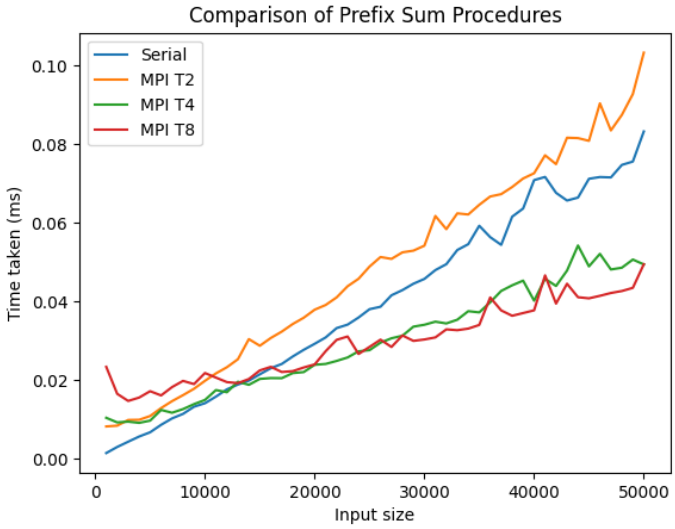

Figure 3: MPI Parallel scan operation average times for different thread counts

1.4 Validation:

Checking if elements are correclty 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

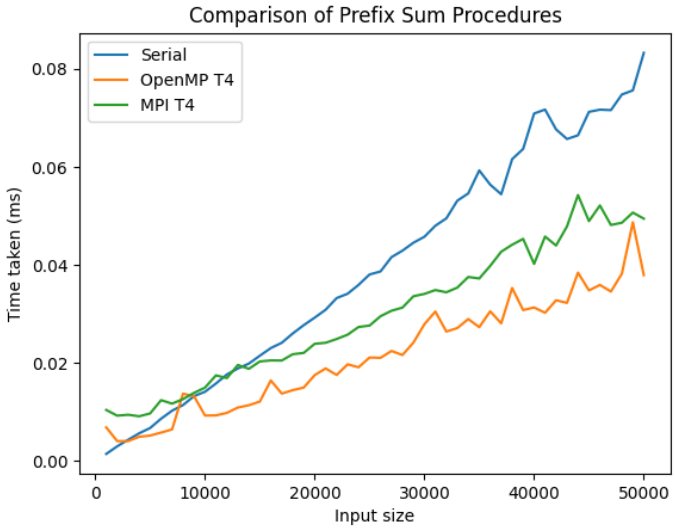

Figure 5: Comparison of all scan operations

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.

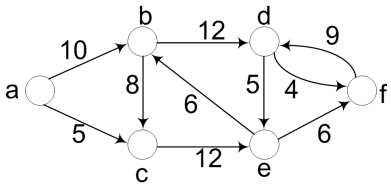

Figure 6: A directed graph

3.1 Serial Implementation:

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

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

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

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

    for (int v = 0; v < V; v++)
      if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
      && dist[u] + graph[u][v] < dist[v])
      dist[v] = dist[u] + graph[u][v];
  }
  return dist;
}
```

Listing 6: Sequential algorithm for Dijkstra's SSSP[5].

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

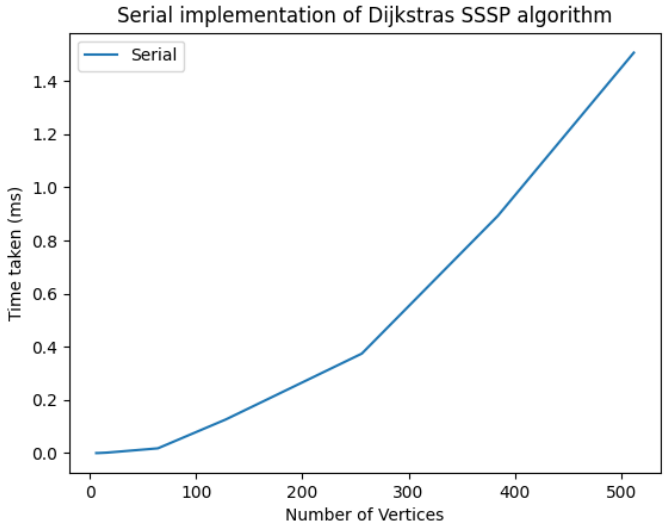

Figure 7: Serial SSSP algorithm average times

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[6] given in Listing 7

```
int* dijkstra(int **graph, int src){
  int* dist = (int*)malloc(V * sizeof(int));
  dist[src] = 0; // Distance of source vertex from itself is always 0
  int i, md, mv;
  int my_first; // The first vertex that stores in one thread locally.
  int my_id; // ID for threads
  int my_last; //The last vertex that stores in one thread locally.
  int my_md; // local minimum distance
  int my_mv; // local minimum vertex
  int my_step; /* local vertex that is at the minimum distance from the
      source */
  int nth; /* number of threads */
  bool sptSet[V]; // sptSet[i] will be true if vertex i has been
     visited

  // Initialize all distances as INFINITE and stpSet[] as false
  for (i = 0; i < V; i++)
    dist[i] = INT_MAX, sptSet[i] = false;

  /* OpenMP parallelization starts here */
  #pragma omp parallel private ( my_first, my_id, my_last, my_md, my_mv
     , my_step )\
  shared ( src, md, dist, mv, nth, graph )
  {
    my_id = omp_get_thread_num ( );
    nth = omp_get_num_threads ( );
    my_first = (my_id * V) / nth;
    my_last = ((my_id + 1) * V) / nth - 1;
    for (my_step = 0; my_step < V-1; my_step++) {
      #pragma omp single
      {
        md = INT_MAX;
        mv = -1;
      }
      int k;
      my_md = INT_MAX;
      my_mv = -1;

      /* Each thread finds the minimum distance unconnected vertex
         inner of the graph */
      for (k = my_first; k <= my_last; k++) {
        if (!sptSet[k] && dist[k] <= my_md) {
          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
                in a
56          * parallel region will wait until all other threads in this
                section reach
57          * the same point. So that md and mv have the correct value. */
58          #pragma omp barrier

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

68          # pragma omp barrier

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

Listing 7: OpenMP Parallel algorithm for computing SSSP[6]

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 contruct, 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 contruct, 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*
- Inisde 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 eachother. 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.

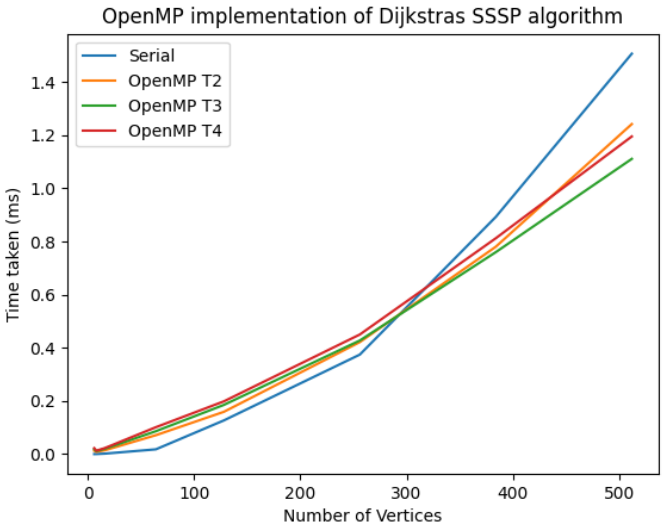

Figure 8: OpenMP Parallel SSSP algorithm average times for different thread counts

3.3 MPI Parallel Implementation:

Next I implented the SSSP algorithm within MPI'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[6] given in Listing 8

```
1   void SingleSource(int source, int *wgt, int *lengths, MPI_Comm comm) {
2     int i, j;
3     int nlocal; /* The number of vertices stored locally */
4     int *marker; /* Used to mark the vertices belonging to Vo */
5     int firstvtx; /* The index of the first local vertex */
6     int lastvtx; /* The index of the last local vertex */
7     int u, udist;
8     int lminpair[2], gminpair[2];
9     int npes, myrank;
10
11    MPI_Comm_size(comm, &npes);
12    MPI_Comm_rank(comm, &myrank);
13
14    nlocal = V / npes;
15    firstvtx = myrank*nlocal;
16    lastvtx = firstvtx + nlocal - 1;
17
18    /* Set the initial distances from source to all the other vertices */
19    for (j = 0; j<nlocal; j++) {
20      lengths[j] = wgt[source*nlocal + j];
21    }
```

```
22      /* This array is used to indicate if the shortest part to a vertex
            has been found or not. */
23      /* if marker [v] is one, then the shortest path to v has been found,
            otherwise it has not. */
24      marker = (int *)malloc(nlocal*sizeof(int));
25      for (j = 0; j<nlocal; j++) {
26        marker[j] = 1;
27      }

29      /* The process that stores the source vertex, mark as seen */
30      if (source >= firstvtx && source <= lastvtx) {
31        marker[source - firstvtx] = 0;
32      }

34      /* The main loop of Dijkstra's algorithm */
35      for (i = 1; i<V; i++) {
36        /* Step 1: Find local vertex with smallest distance from source */
37        lminpair[0] = INT_MAX; /* set it to an arbitrarily large number */
38        lminpair[1] = -1;
39        for (j = 0; j<nlocal; j++) {
40          if (marker[j] && lengths[j] < lminpair[0]) {
41            lminpair[0] = lengths[j];
42            lminpair[1] = firstvtx + j;
43          }
44        }

46        /* Step 2: Compute the global minimum vertex, insert it into Vc */
47        MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC, comm);
48        udist = gminpair[0];
49        u = gminpair[1];
50        /* The process that stores the minimum vertex, mark as seen */
51        if (u == lminpair[1]) {
52          marker[u - firstvtx] = 0;
53        }

55        /* Step 3: Update the distances given that u got inserted */
56        for (j = 0; j<nlocal; j++) {
57          if (marker[j] && ((udist + wgt[u*nlocal + j]) < lengths[j])
58          && wgt[u*nlocal + j] != INT_MAX && udist != INT_MAX) {
59            lengths[j] = udist + wgt[u*nlocal + j];
60          }
61        }
62      }
63      if (myrank == 0) free(marker);
64    }

66    int main(int argc, char *argv[]) {
67      int npes, myrank, nlocal;
68      int i, j, k;
```

```
69      int *localWeight; /*local weight array*/
70      int *localDistance; /*local distance vector*/
71
72      MPI_Init(&argc, &argv);
73      MPI_Comm_size(MPI_COMM_WORLD, &npes);
74      MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
75
76      nlocal = V/npes; /* number of elements per thread. */
77
78      int graph[V*V];
79      int sendbuf[V*V]; /*local weight to distribute*/
80      int dist = (int*)malloc(V * sizeof(int)); /*distance vector*/
81
82      localWeight = (int *)malloc(nlocal*V*sizeof(int));
83      localDistance = (int *)malloc(nlocal*sizeof(int));
84
85      /*prepare send data */
86      for(k=0; k<npes; ++k) {
87        for(i=0; i<V;++i) {
88          for(j=0; j<nlocal;++j) {
89            sendbuf[k*V*nlocal+i*nlocal+j]=graph[i][k*nlocal+j];
90          }
91        }
92      }
93
94      /*distribute data*/
95      MPI_Scatter(sendbuf, nlocal*V, MPI_INT, localWeight, nlocal*V,
            MPI_INT,
96      0, MPI_COMM_WORLD);
97
98      /*Implement the single source dijkstra's algorithm*/
99      SingleSource(0, localWeight, localDistance, MPI_COMM_WORLD);
100
101     /*collect local distance vector at the source process*/
102     MPI_Gather(localDistance, nlocal, MPI_INT, dist, nlocal, MPI_INT,
103     0, MPI_COMM_WORLD);
104
105     MPI_Finalize();
106   }
```

Listing 8: MPI Parallel algorithm for computing SSSP[6]

The algorithm works as follows:

- Initialize all global variables
- Start MPI
- Get comm rank and size
- Set number of local vertices to total vertices (V) / comm size ($npes$)
- Initialize more variables
- Prepare to send localized data
- Distribute data with MPI call Scatter[7]
- Implement Dijkstra's SSSP algorithm in parallel
  - Initialize all local variables
  - Get comm rank and size
  - Set number of local vertices to total vertices (V) / comm size ($npes$)
  - Set starting vertex and ending vertex for current process
  - Get initial distances from source to all other vertices
  - Initialize a marker array
  - Mark the source as seen
  - Find local minimum distances per process
  - Find global minimum distance and insert it into V_c
  - MPI call to Allreduce[8]
  - Mark the vertex with the minimum distance as seen
  - Update all distances from local to global
  - Free marker array from memory
- Collect distance data with MPI call Gather[9]
- End MPI

I experimented with different number of threads running and noticed number of threads less than 6 produce desired speedup. While anything over 6 starts to slow down the algorithm rather than speed it up. From this conclusion I chose thread counts of $2, 3, 6$ to compare, given in Figure 9

All 3 parallel implementations perform quite similarly. With noticeable trends, in 6 threads, speedup is only achieved at 250 vertices and higher, while 2 and 3 threads achieve speedup much earlier. With 3 threads running there is consistently more speedup at all vertex counts.

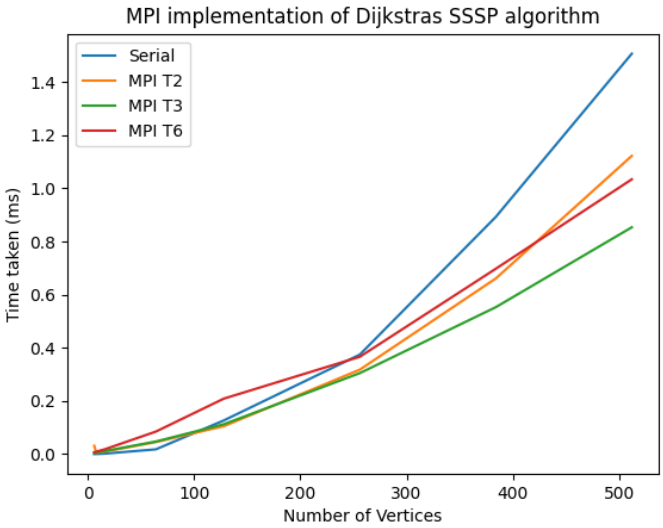

Figure 9: MPI Parallel SSSP algorithm average times for different thread counts

3.4 Validation:

Comparing output with serial counterpart, through running *run.sh* given in Figure 10

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

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

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

Figure 10: Validation of SSSP operations

3.5 Conclusions:

After comparing the results from serial, optimal OpenMP (with 3 threads) and optimal MPI (with 3 threads), I can make the conclusion that speedup at small vertex counts is negligible. While around the 250 vertex count mark, both OpenMP and MPI have significant speedup from serial, with MPI speedup being consistently better than OpenMP. Comparisons given in Figure 11

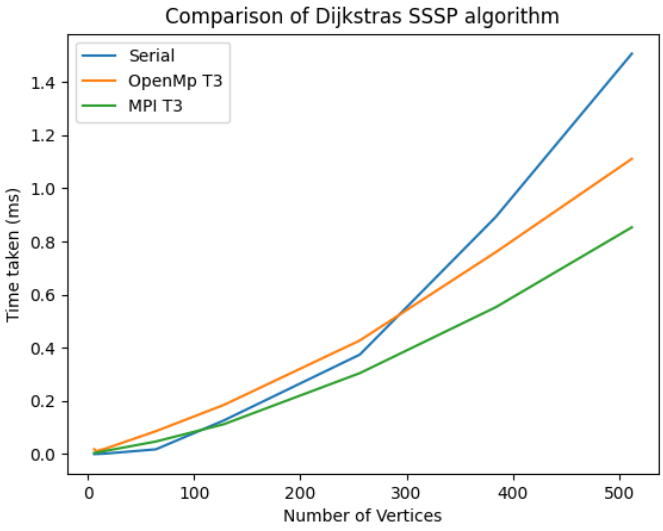

Figure 11: Comparison of all scan operations

The speedup is given in Table 1.

Table 1: Table of speedup (in ms)

No of vertices	6	8	16	64	128	256	384	512
Serial	0.0002	0.0003	0.0018	0.0180	0.1268	0.3746	0.8928	1.5070
Parallel	0.0046	0.0052	0.0117	0.0474	0.1125	0.3048	0.5530	0.8536
Sppedup	0	0	0	0	1.13	1.23	1.62	1.77

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, 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).

[7] MPI Scatter | (3) man page (version 3.1.6), https://www.open-mpi.org/doc/v3.1/man3/MPI_Scatter.3.php, accessed: 25-10-2022.

[8] MPI Allreduce | manual pages, https://www.mpich.org/static/docs/v3.3/www3/MPI_Allreduce.html, accessed: 25-10-2022.

[9] MPI Gather | (3) man page (version 3.1.6), https://www.open-mpi.org/doc/v3.1/man3/MPI_Gather.3.php, accessed: 25-10-2022.