COMP528 Individual Continuous Assignment 1

1. Approach

- Serial Solution for Cheapest Insertion:

  Code- [Below fig is just presenting logic of cheapest function]

cInsertion.c

Final > cInsertion.c > cheapest_insertion(double **, int)

```c
int* cheapest_insertion(double** dist_matrix, int number_of_coordinates) {
    //We will use malloc function allocating memory in our one-d array
    int* path = (int*)malloc((number_of_coordinates + 1) * sizeof(int));
    //using calloc function to allocating memory at starting point
    int* cityvisited = (int*)calloc(number_of_coordinates, sizeof(int));
    //setting flag 1 since its starting point
    cityvisited[0]=1;
    double  min_dist, insert_distance;
    int nearest, insertion_position;
    nearest = -1;
    min_dist = DBL_MAX;

    //since tour will start from zero, setting zero at all index at intial phase
    for(int a=0; a<number_of_coordinates; a++){
        path[a]=0;
    }//finding first minimum cost from zero
        for (int v = 1; v < number_of_coordinates; v++) {
                insert_distance = 0.0;
                insert_distance = dist_matrix[0][v] ;
                //comparing distance of vth position
                //insert_distance calculated at each iteration is getting check if true then at vth position is updated in nearest and min_dist is now having new value
                if (insert_distance < min_dist) {
                    nearest = v;
                    min_dist = insert_distance;
                }
        }
        //placing new nearest city in path[] and changing flag to 1 in cityvisited[]
        path[1]=nearest;
        cityvisited[nearest]=1;

    for(int index=1;index<number_of_coordinates-1;index++) {
         //reinitialising min distance for rest of the traversing
         min_dist = DBL_MAX;
         insertion_position=0;
         //comparing distance for cities not visited by using cordinates added in path[]
    for (int i = 0; i < number_of_coordinates-1; i++) {
        for (int v = 1; v < number_of_coordinates; v++) {
            if (!cityvisited[v]) {
              insert_distance=0.0;
              insert_distance = dist_matrix[path[i]][v] + dist_matrix[v][path[i+1]] - dist_matrix[path[i]][path[i+1]];
              //we are finding out the cost for each vertex between two index of path[]
              // insert_distance calculated at each iteration is getting check if true then at vth position is updated in nearest and min_dist is now having new value
              if (insert_distance < min_dist) {
                    nearest = v;
                    insertion_position=i;
                    min_dist = insert_distance;
              }
            }
        }
    }
    //changing flag to 1 in cityvisited[]
    cityvisited[nearest]=1;
    //shift the old cities by 1 position so that new city can be added
    for (int i = number_of_coordinates; i > insertion_position+1; i--) {
        path[i] = path[i - 1];
    }
    //placing new nearest city in path[]
    path[insertion_position+1]=nearest;
    }
return path;
```

Strategy-

1. The function uses a distance matrix 'dist_matrix' and the number of coordinates as input and returning an array representing the cheapest insertion path.
2. Allocating memory for two arrays: 'path' and to store the final path and 'cityvisited' to keep track of visited cities by setting flags 0/1.
3. Starting with the initial city (0), the algorithm determines the closest city with the help of distance matrix to create the initial path.
4. In main loop it then iteratively insert the next city into path[] at the location that gives minimum distance, chosen repeatedly.
5. The procedure keeps going until every city is seen, creating a finished trail. An Array of integers containing the final path is returned.
6. Also, the code used function and constants (such DBL_MAX) and allocates dynamic memory using the malloc, calloc.

- Serial Solution for Farthest Insertion:

  Code- [Below fig is just presenting logic of farthest function]

fInsertion.c ompcInsertion.c ompfInsertion.c

Final > fInsertion.c > farthest_insertion(double **, int)

```c
int* farthest_insertion(double** dist_matrix, int number_of_coordinates) {
    //We will use malloc function allocating memory in our one-d array
    int* partial_tour = (int*)malloc((number_of_coordinates + 1) * sizeof(int));
    //using calloc function to allocating memory at starting point
    int* visited = (int*)calloc(number_of_coordinates, sizeof(int));
    //setting flag 1 since its starting point
    visited[0] = 1;
    double  min_dist, insert_distance,max_dist;
    int nearest_vertex, insertion_position,max_vertex;
    nearest_vertex = -1;min_dist = DBL_MAX;max_dist = DBL_MIN;
  //since tour will start from zero, setting zero at all index at intial phase
    for (int i = 0; i < number_of_coordinates; i++) {
                partial_tour[i]=0;}
        for (int vertex = 1; vertex < number_of_coordinates; vertex++) {
                insert_distance=0.0;insert_distance = dist_matrix[0][vertex] ;
                //comparing distance of vth position
                //insert_distance calculated at each iteration is getting check if true then at vth position is updated in nearest and min_dist is now having new value
                if (insert_distance > max_dist) {
                    nearest_vertex = vertex;
                    max_dist = insert_distance;}}
        partial_tour[1]=nearest_vertex;
        visited[nearest_vertex]=1;
     for(int index=1;index<number_of_coordinates-1;index++) {
         //reinitialising min&max distance for rest of the traversing
         min_dist = DBL_MAX;
         max_dist = DBL_MIN;
         insertion_position=0;
        for (int i = 0; i < number_of_coordinates-1; i++) {
          for (int vertex = 1; vertex < number_of_coordinates; vertex++) {
            if (!visited[vertex]) {
              insert_distance = 0.0;
                insert_distance = dist_matrix[partial_tour[i]][vertex] ;
                //we are finding out the cost for each vertex between two index of tour
                // insert_distance calculated at each iteration is getting check if true then at vth position is updated in farthest and max_dist is now having new valu
                if (insert_distance > max_dist) {
                    max_vertex = vertex;
                    max_dist = insert_distance;
                }}}}
          //comparing distance for cities not visited by using cordinates added in tour
    for (int i = 0; i < number_of_coordinates-1; i++) {
            insert_distance = 0.0;
            insert_distance = dist_matrix[partial_tour[i]][max_vertex] + dist_matrix[max_vertex][partial_tour[i+1]] - dist_matrix[partial_tour[i]][partial_tour[i+1]];
        //we are finding out the cost for each vertex between two index of tour
        // insert_distance calculated at each iteration is getting check if true then at vth position is updated in nearest and min_dist is now having new value
        if (insert_distance < min_dist) {
                            nearest_vertex = max_vertex;
                            insertion_position=i;
                            min_dist = insert_distance;}
} //changing flag to 1 in cityvisited[]
        visited[nearest_vertex]=1;
        //shift the old cities by 1 position so that new city can be added
      for (int i = number_of_coordinates; i > insertion_position+1; i--) {
            partial_tour[i]=partial_tour[i-1];
      }//placing new nearest city in tour
      partial_tour[insertion_position+1]=nearest_vertex;
     }
return partial_tour;
```

Strategy-

1. Initialized the partial tour with zeros which will store resulting tour.
2. Identify the initial city which is giving max distance from starting point city 0.
3. With that set the state of the tour by placing the farthest city in partial tour array and change flag of visited array to 1.
4. In main loop, iteratively insert the farthest cities into the current tour. Later on checking if condition we will find the farthest vertex from the current tour.
5. Moving ahead we determine the position to insert the farthest vertex in the current tour by small logic used in cheapest also (i.e. find the minimum cost).
6. Finally, return the partial tour array.

- Parallel Solution for Cheapest Insertion:

Code- [Below fig is just presenting logic of distance matrix &cheapest function in parrallelising way]

Distance matrix part-

```
//With the coordinates we will use distance formula to find distance and store it in matrix
double **distance_matrix_calculate(double **points, int number_of_coordinates){
    //We will use malloc function allocating memory in our matrix
    double **dist_matrix = malloc(sizeof(double *) * number_of_coordinates);
    //parallelising dist calculation
    #pragma omp parallel for
    for (int i = 0; i < number_of_coordinates; i++) {
    dist_matrix[i] = malloc(sizeof(double) * number_of_coordinates);

    for (int j = 0; j < number_of_coordinates; j++) {
      if(i==j){
        dist_matrix[i][j]=0.0;
      } else {
          double x = points[i][0]-points[j][0];
          double y = points[i][1]-points[j][1];
        dist_matrix[i][j]=sqrt(x*x+y*y);
      }
    }
```

Cheapest part-

```
        //comparing distance for cities not visited by using cordinates added in path[]
        //parallelising minimun distance finding between elements of tour
        #pragma omp parallel
        {
            //reinitialising local min distance for rest of the traversing
            double local_min_dist = DBL_MAX;
            int local_nearest = -1, local_insertion_position = 0;

            #pragma omp for nowait
            for (int i = 0; i <= index; i++) {
                for (int v = 1; v < number_of_coordinates; v++) {
                    if (!cityvisited[v]) {
                        double local_insert_distance = dist_matrix[path[i]][v] + dist_matrix[v][path[i + 1]] - dist_matrix[path[i]][path[i + 1]];
                        //we are finding out the cost for each vertex between two index of path[]
                        //insert_distance calculated at each iteration is getting check if true then at vth position is updated in nearest and
                        //min_dist is now having new value
                        if (local_insert_distance < local_min_dist) {
                            local_nearest = v;
                            local_insertion_position = i;
                            local_min_dist = local_insert_distance;
                        }
                    }
                }
            }
            //using critcal , also substituting private variables to global to reduce effort
            #pragma omp critical
            {
                if (local_min_dist < min_dist) {
                    nearest = local_nearest;
                    insertion_position = local_insertion_position;
                    min_dist = local_min_dist;
                }
            }
        }
        //changing flag to 1 in cityvisited[]
        cityvisited[nearest] = 1;
```

Strategy-

1. Parallelizing the outer loop of the algorithm that iterates over the cities to be visited for each iterations it seeks the least expensive way to add city to the current tour.
2. #pragma omp parallel directives initiates parallel region, nowait clause indicate that threads are not required to wait until the end of loop to proceed.
3. Using local variables local_min_dist, local_nearest and local_insertion_position are unique to each threads inside parallel region and track minimum distance.
4. To guarantee that only one thread at a time updates the global variable (min_dist,nearest,insertion_position), #critical section is used.
5. The #pragma omp nowait directive uses the private clause to indicate that each thread get copy of the loop variable (i), data dependencies are avoided.
6. Also, have parallelised in the distance matrix function to outer loop, in order to execute it in parallel, several threads will share their iterations.

- Parallel Solution for Farthest Insertion:

  Code-[Below fig is just presenting logic of farthest function in parrallelising way]

ompfInsertion.c

Final > ompfInsertion.c > ...

```c
// Finding farthest insertion tour
int* farthest_insertion(double** dist_matrix, int number_of_coordinates) {
    //We will use malloc function allocating memory in our one-d array
    int* partial_tour = (int*)malloc((number_of_coordinates + 1) * sizeof(int));
    //using calloc function to allocating memory at starting point
    int* visited = (int*)calloc(number_of_coordinates, sizeof(int));
    if (!partial_tour || !visited) {
        // Free memory allocation, getting segmentation error at tour
        free(partial_tour);
        free(visited);
        return NULL;
    }

    visited[0] = 1;
    double max_dist = DBL_MIN;
    int nearest_vertex = -1;
    //since tour will start from zero, setting zero at all index at intial phase
    for (int i = 0; i < number_of_coordinates; i++) {
        partial_tour[i] = 0;
    }
    //finding first max cost from zero
    for (int vertex = 1; vertex < number_of_coordinates; vertex++) {
        double insert_distance = dist_matrix[0][vertex];
        //comparing distance of vertex th position
        //insert_distance calculated at each iteration is getting check if true then at vertex th position is updated in nearest and max_dist is now having new value
        if (insert_distance > max_dist) {
            nearest_vertex = vertex;
            max_dist = insert_distance;
        }
    }//placing new nearest city in partial_tour[] and changing flag to 1 in visited[]
    partial_tour[1] = nearest_vertex;
    visited[nearest_vertex] = 1;

    int nearest_vertex = -1;
    int insertion_position = 0;
    max_dist = DBL_MIN;
    int max_vertex = -1;

    #pragma omp parallel
    {
        double thread_max_dist = DBL_MIN;
        int thread_max_vertex = -1;

        #pragma omp for nowait
        for (int vertex = 1; vertex < number_of_coordinates; vertex++) {
            if (!visited[vertex]) {
                for (int i = 0; i < index; i++) {
                    double current_distance = dist_matrix[partial_tour[i]][vertex];
                    //we are finding out the cost for each vertex between two index of partial tour[]
                    //insert_distance calculated at each iteration is getting check if true then at vertex th position is updated and
                    //max current_distance is now having new value
                    if (current_distance > thread_max_dist) {
                        thread_max_dist = current_distance;
                        thread_max_vertex = vertex;
                    }
                }
            }
        }
        //using critcal , also substituting private variables to global to reduce effort
        #pragma omp critical
        {
            if (thread_max_dist > max_dist) {
                max_dist = thread_max_dist;
                max_vertex = thread_max_vertex;
            }
        }
    }
```

Strategy-

1. In this first memory allocation failure is handled to prior execution of algorithm in parallelised version. It frees the allotted memory and return NULL if memory allocation fails.
2. Parallelized the outer loop, the task split across several threads for execution while the loop iterates over vertices which need to be add into the tour.
3. #pragma omp parallel starts a parallel section, #pragma omp for nowait divides the loop iterations among the available threads.
4. Finding the max distance and matching vertex for each thread inside the parallel zone.
5. The variables max dist, max vertex and nearest vertex have been moved inside the loop in parallelised region, thereby treating them local to each thread this helps in solving race condition by updating them at same time while multiple thread runs.

2. Speedup Plot

For Plotting Speedup v/s Number of threads, we need to find Speedup between serial and parallel code.

Formula for Speedup: Serial time / parallel time (n) where n is number of threads

[A] Serial Speedup for cheapest

Serial/sequential cheapest time: 349.369 seconds

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 1 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392689-sequential-4096.out
Number of threads or processes :
Running iteration 1
Exceution time 349.341865 seconds
Writing output data

real    5m49.369s
user    5m48.458s
sys     0m0.085s
----end---
```

Parallel cheapest time for 1 thread: 207.637 seconds

Speedup: 349.369/207.637 = 1.682

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 1 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392560-4096-1.out
Number of threads or processes :
Running iteration 1
Exceution time 207.613070 seconds
Writing output data

real    3m27.637s
user    3m27.046s
sys     0m0.099s
```

Parallel cheapest time for 2 threads: 119.362

Speedup: 349.369/119.362= 2.926

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 2 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392548-4096-2.out
Number of threads or processes :
Running iteration 1
Exceution time 119.334063 seconds
Writing output data

real    1m59.362s
user    3m57.870s
sys     0m0.248s
```

Parallel cheapest time for 4 threads: 61.239 seconds

Speedup: 349.369/61.239 = 5.705

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 4 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392541-4096-4.out
Number of threads or processes :
Running iteration 1
Exceution time 61.212865 seconds
Writing output data

real    1m1.239s
user    4m3.880s
sys     0m0.339s
```

Parallel cheapest time for 8 threads: 31.489 seconds

Speedup: 349.369/31.489 = 11.094

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 8 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392109-4096-8.out
Number of threads or processes :
Running iteration 1
Exceution time 31.460839 seconds
Writing output data

real    0m31.489s
user    4m10.325s
sys     0m0.500s
```

Parallel cheapest time for 16 threads: 17.163 seconds

Speedup: 349.369/17.163 = 20.355

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 16 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392079-4096-16.out
Number of threads or processes :
Running iteration 1
Exceution time 17.134766 seconds
Writing output data

real    0m17.163s
user    4m31.453s
sys     0m1.091s
```

Parallel cheapest time for 32 threads: 9.268 seconds

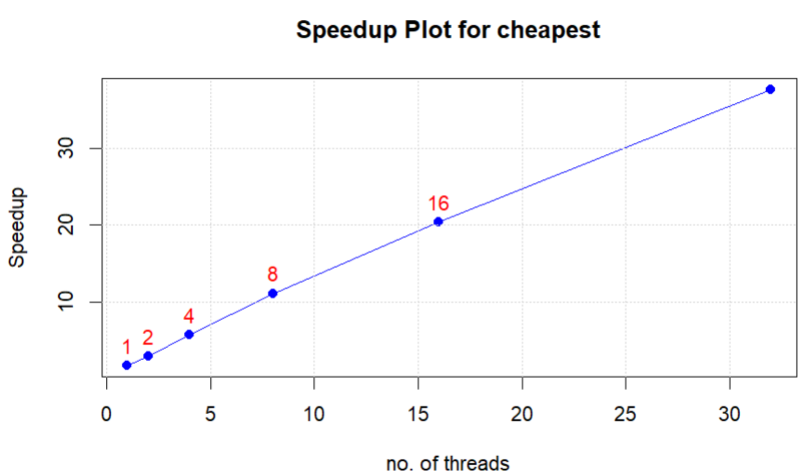
Speedup: 349.369/9.268 = 37.696

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 32 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-c-46392065-4096-32.out
Number of threads or processes :
Running iteration 1
Exceution time 9.236157 seconds
Writing output data

real    0m9.268s
user    4m51.060s
sys     0m1.173s
```

Graph: Upwards trend in speedup by increasing threads



Noted:[Graph has been made with the help of r, using concept of plot() from COMP563]

[A] Serial Speedup for farthest

Serial/sequential farthest time:166.814 seconds

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 1 Batch.sh
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46392765-sequential-4096.out
Number of threads or processes :
Running iteration 1
Writing output data

real    2m46.814s
user    2m46.339s
sys     0m0.054s
----end---
```

Parallel farthest time for 1 thread: 65.070 seconds

Speedup: 166.814/65.070 = 2.563

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 1 Batch.sh
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46392033-4096-1.out
Number of threads or processes :
Running iteration 1
Exceution time 65.046211 seconds
Writing output data

real    1m5.070s
user    1m4.803s
sys     0m0.109s
-------
```

Parallel farthest time for 2 threads: 35.511 seconds

Speedup: 166.814/35.511 = 4.697

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 2 Batch.sh
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46392030-4096-2.out
Number of threads or processes :
Running iteration 1
Exceution time 35.480079 seconds
Writing output data

real    0m35.511s
user    1m10.573s
sys     0m0.229s
```

Parallel farthest time for 4 threads: 18.331 seconds

Speedup: 166.814/18.331 = 9.100

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 4 Batch.sh
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46392027-4096-4.out
Number of threads or processes :
Running iteration 1
Exceution time 18.287823 seconds
Writing output data

real    0m18.331s
user    1m12.594s
sys     0m0.348s
```

Parallel farthest time for 8 threads: 9.996 seconds

Speedup: 166.814/9.996 = 16.680

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 8 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46392002-4096-8.out
Number of threads or processes :
Running iteration 1
Exceution time 9.996294 seconds
Writing output data

real	0m10.025s
user	1m18.959s
sys	0m0.577s
```

Parallel farthest time for 16 threads: 5.574 seconds

Speedup: 166.814/5.574 = 29.927

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 16 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46391997-4096-16.out
Number of threads or processes :
Running iteration 1
Exceution time 5.574896 seconds
Writing output data

real	0m5.612s
user	1m27.086s
sys	0m0.884s
```

Parallel farthest time for 32 threads: 3.837 seconds

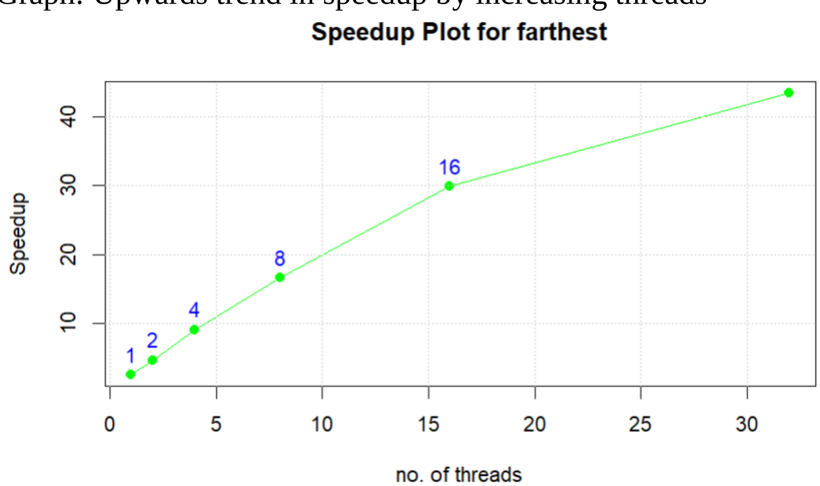
Speedup: 166.814/3.837 = 43.475

```
[sgyraora@viz02[barkla] Assignment01]$ sbatch -c 32 Batch.sh
```

```
[sgyraora@viz02[barkla] Assignment01]$ cat slurm-f-46391985-4096-32.out
Number of threads or processes :
Running iteration 1
Exceution time 3.837250 seconds
Writing output data

real	0m3.876s
user	1m58.324s
sys	0m1.476s
```

Graph: Upwards trend in speedup by increasing threads



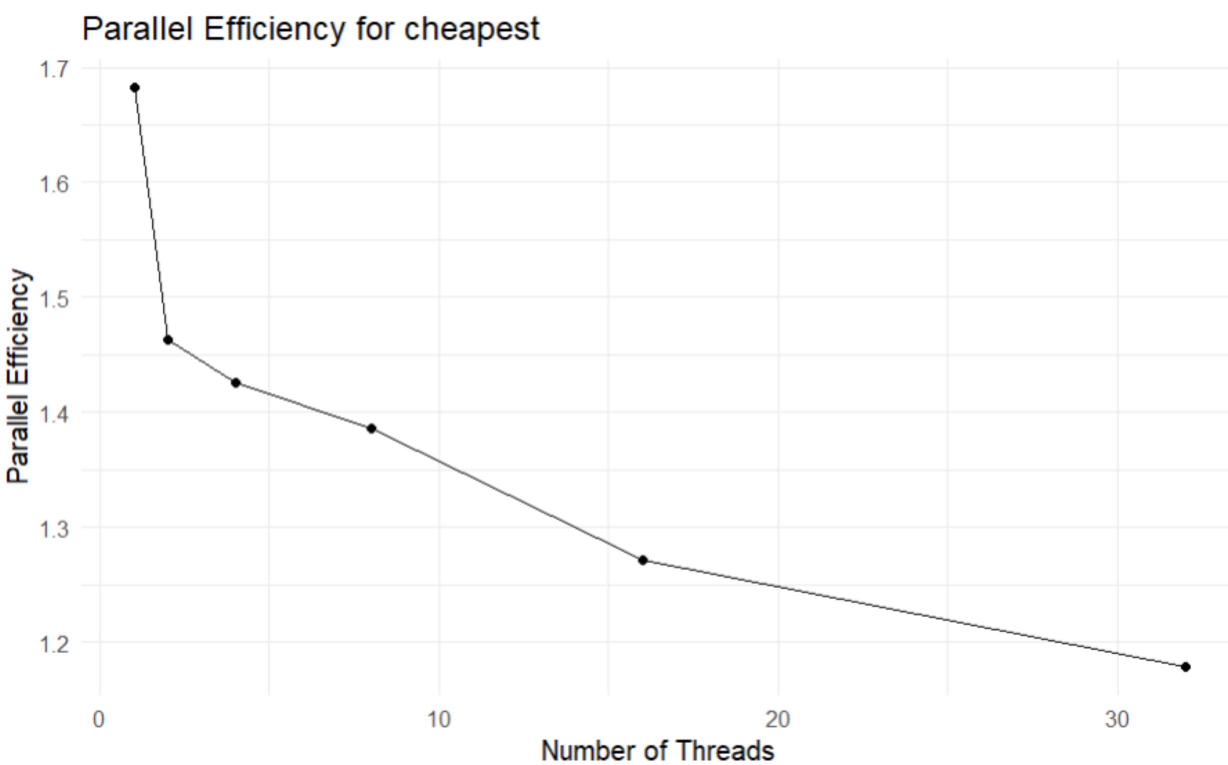
Noted:[Graph has been made with the help of r, using concept of plot() from COMP563]

3. Efficiency Plot

For Plotting Efficiency v/s Number of threads
Formula for Parallel Efficiency: Speedup /number of threads

[A] Parallel Efficiency (P.E) for cheapest
Calculation:
P.E(1): 1.682/1=1.682
P.E(2): 2.926/2=1.463
P.E(4): 5.705/4=1.426
P.E(8): 11.094/8=1.386
P.E(16): 20.355/16=1.272
P.E(32): 37.696/32=1.178

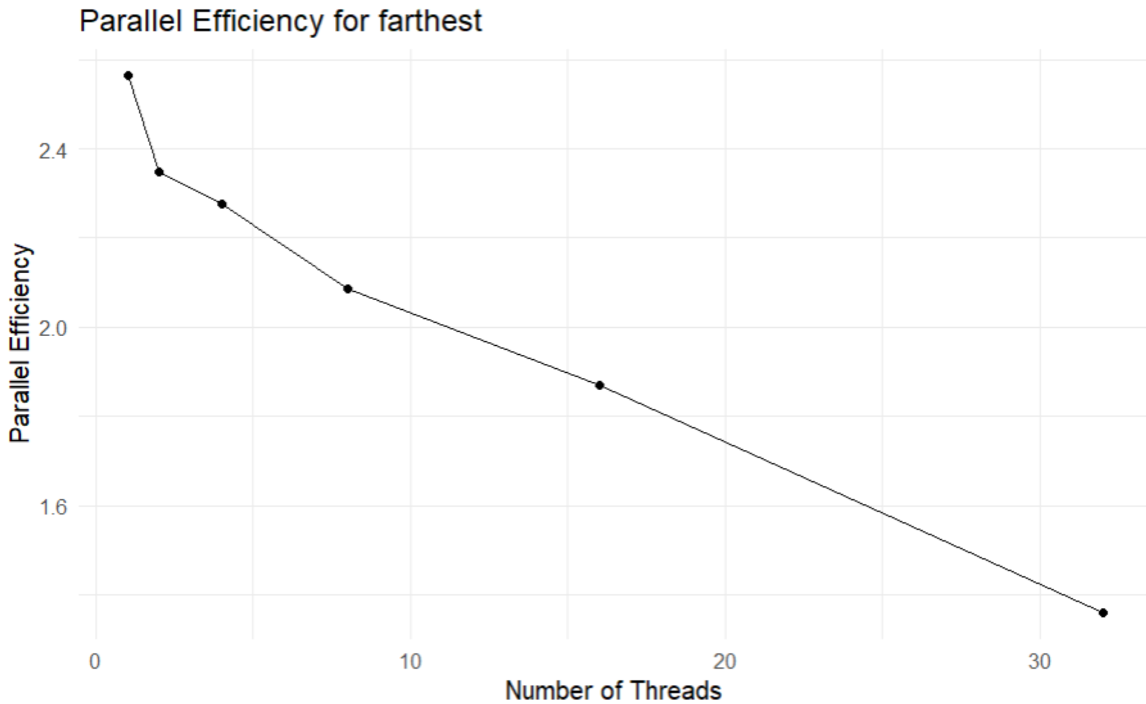
Graph: Downward trend in efficiency by increasing threads



Noted:[Graph has been made with the help of r, using concept of plot() from COMP563]

[B] Parallel Efficiency for farthest
Calculation:
P.E(1): 2.563/1=2.563
P.E(2): 4.697/2=2.348
P.E(4): 9.100/4=2.275
P.E(8): 16.680/8=2.085
P.E(16): 29.927/16=1.870
P.E(32): 43.475/32=1.358

Graph: Downward trend in efficiency by increasing threads



Noted:[Graph has been made with the help of r, using concept of plot() from COMP563]

4. Conclusion

All the tasks are completed. Plotted Speedup and Efficiency for both cheapest and farthest, determine the speed of program. Also, explained the code/strategy used.