CS433: Parallel Programming
Assignment I Report

Aditi Goyal - 190057
Yatharth Goswami - 191178

March 10, 2022

Contents

I Travelling Salesman Problem - OpenMP

In this problem, we were asked to solve the famous travelling salesman problem. We are given a complete undirected graph of Ns cities with positive integer weights on edges. We need to determine the shortest tour (and it's cost) covering each city exactly once and returning back to the starting city. We solved the problem using **C** programming language and **OpenMP** framework.

Let us define some terminologies that will be used:

- The graph is $G = (V, E)$ where V denotes the set of vertices and E denotes the set of edges.
- Edge weight between vertex i and vertex j is represented as $cost[i][j]$

Note : In the discussion below, we use 0 indexing.

1.1 Previous attempts

1.1.1 Permutations

- **Sequential Algorithm:** Brute force approach where we iterate over all permutations of vertices and pick the path with minimum cost. This approach is a pure recursive brute force approach and has time complexity of order of $O(N!)$.
- **Parallel Algorithm:** We assign different permutations to different threads (based on starting vertex) and compute the costs parallelely. We assume that we start from the vertex 0 and then travel all the other vertices and come back to 0. For making it parallel, we assigned different second vertices to all the threads. In this way, each thread explores one of the sub-trees in level 1 of the complete recursion tree. The scheduling was kept dynamic to keep the threads busy.

Problems with this approach: There is a lot of work to be done by a thread. Even if we fix the first and second vertex, $(N-2)!$ permutations still need to be explored by a thread. Even the cost calculations are not reused in this way and therefore there is a lot of redundant computation done by each of the threads. Although the solution was able to achieve some speedup but this approach took large amount of time even for small graphs of size around 10 and hence we left exploring this approach any more.

1.1.2 Bitmasking

- **Sequential algorithm:** This approach [1] uses dynamic programming paradigm. Let $M = 2^N$. We will represent the set of vertices visited using a bitmask, where we store an array of 1s and 0s denoting whether a vertex has been visited yet or not (1 denoting that the vertex has been visited and 0 otherwise). Notice, that each number from 0 to $M-1$ can be represented in it's binary form of size N containing 1 and 0. Therefore, we can represent any set of vertices visited till now, by a number in the range 0 to $M-1$ by converting the bitmask represented by the set of visited vertices to the decimal equivalent. Next, we create a dp array of size $M \times N$ where dp[i][j] denotes minimum cost to visit the set of vertices represented by the number i (converting to the corresponding bitmask) and j representing the last visited vertex among the set of visited ones. The algorithm runs as follows:

We assume that we always start from vertex 0 and compute dp[i][j] iteratively for all the is where vertex 0 is always visited. In the end the minimum cost obtained will be

$$\min_j dp[M-1][j] + cost[j][0]$$

The way to compute $dp[i][j]$ is that we will iterate over the second last vertex visited in the set of vertices visited and we get the following relation

$$dp[i][j] = \min_{sl}(dp[i-(1<<j)][sl] + cost[sl][j]) \quad \text{where i \& (1 << sl)} \neq 0$$

- **Parallel algorithm:** There can be two approaches to parallelize the above sequential algorithm. Either we can parallelize the iteration over the second last vertex (the innermost loop), since there is no loop carried dependence in this loop or we can either parallelize the iteration over the last vertex chosen. The second approach can be done because the computations for $dp[i][j]$ and $dp[i][k]$ where $j \neq k$ do not depend on each other and can be done parallely. So, we parallelize it and computations of $dp[i][j]$ for different j proceed simultaneously.

Problems with the approaches: The first approach parallelizes the innermost loop and hence leads to a lot of synchronization overheads as there is implicit barrier at the end of the loop and the other problem is that it involves calculation of minimum value and hence it requires to be done inside a critical section. Therefore, there will be need of taking locks which highly compromises the performance. Also, there will be false sharing in this case. Updates by different threads may be made to dp[subset][j] and dp[subset][k] where $j \neq k$ but these might lie in the same cache block (resulting in false sharing).

For the second approach some threads do not work much. For ex, assume some thread is computing $dp[i][j]$ where the j^{th} bit in i is not set to 1. This thread will have much less work as compared thread assigned $dp[i][k]$ where k^{th} bit in i is set to 1. This would lead to load imbalance. Also, there is overhead in synchronization (implicit barrier) because no $dp[i+1][k]$ is started before all $dp[i][j]$ have been computed. So, though we can run this till $n = 24$, the speedup achieved is not great.

1.2 Final attempt

1.2.1 Sequential algorithm

This algorithm again works on the bitmasking approach discussed earlier. Only the order in which dp matrix is filled is different. In the above algorithm, we iterated in the order of masks from 0 to $2^N - 1$ whereas now, we will iterate in the order of number of 1s in the mask. Here, first costs for subsets of size 2 are computed, then for subsets of size 3 and so on. In other words, the outermost loop iterates over the number of 1s in the mask or equivalently the size of the visited set and inside each loop, the calculation of $dp[i][j]$ is exactly the same as described in the previous solution. Here is the pseudo code of the sequential algorithm:

Algorithm 1: Sequential algorithm for TSP

```
   Input: Number of vertices : n, Edge weights : cost
   Output: Min cost path : MinPath, min cost : MinCost
1  Subsets → Stores subsets of masks of a given size as an array of integers.
2  Done → Stores the size of each set of masks for a given size of subset.
3  Function SolveTSP():
4      for SubsetSize in 3 to N do
5          for SubsetIndex in 0 to Done[SubsetSize]-1 do
6              Subset ← Subsets[SubsetSize][SubsetIndex]
7              for last in 1 to N-1 do
8                  if Subset & (1<<last) then
9                      dp[Subset][last] ← INF
10                     prev ← Subset-(1<<last)
11                     for secondLast in 1 to N-1 do
12                         if prev & (1<<secondLast) then
13                             if dp[Subset][last] > dp[prev][secondLast] +
                                cost[secondLast][Last] then
14                                 dp[Subset][last] ←dp[prev][secondLast] +
                                    cost[secondLast][Last]
15                                 parent[Subset][last] ← secondLast
16     for last in 1 to N-1 do
17         if minCost > dp[1<<N-1][last]+cost[last][0] then
18             minCost ← dp[1<<N-1][last]+cost[last][0]
19             lastIndex ← last
20 Function GenPath():
21     MinPath[N-1] ← lastIndex
22     last ← lastIndex
23     Subset ← 1<<N-1
24     for i in N-2 to 0 do
25         MinPath[i] ← parent[Subset][last]
26         subset ← subset-(1<<last)
27         last ← parent[Subset][last]
```

1.2.2 Parallel algorithm

Notice the fact that for all the subsets of same size, we can compute the costs for these masks in parallel as all of these computations depend on smaller sized subsets which are already calculated before. Therefore, for each subset size, we parallelize the calculation of dp matrix for all the subsets of that size. We chose to do static scheduling with equal number of tasks given to each thread. Since the number of subsets for each of the size is greater than number of threads always there will be no thread which will remain idle for most of the time. Also, since calculation for each subset is just dependent on the number of ones present in the mask of the subset, there will be no load imbalance either as all the subsets in a parallel for have same number of bits set.

We tried the simple parallelization initially with '#pragma omp parallel for' above the loop in which we are iterating on SubsetIndex. But the problem with this is that after every iteration of the loop on SubsetSize, we will create new set of threads for the next iteration. Instead we optimized the code by creating the threads first and then using these threads in parallel for the inner for loop. Another optimisation that we did was to reduce the false sharing across the threads. For this, we malloced the dp array for each subset distinctly instead of allocating the array statically. In this manner, the dp array for each subset will be allocated in different regions and since the threads are independently working on different subsets at a certain point of time, the false sharing will not occur. We parallelized the SolveTSP() function in the pseudo code and rest of the code is similar to the sequential code.

Algorithm 2: Parallel algorithm for TSP

```
Input: Number of vertices : n, Edge weights : E
Output: Min cost path : MinPath, min cost : MinCost
1  Subsets → Stores subsets of masks of a given size as an array of integers.
2  Done → Stores the size of each set of masks for a given size of subset.
3  omp_set_num_threads(n_threads)
4  Function SolveTSP():
5      #pragma omp parallel
6      for SubsetSize in 3 to N do
7          #pragma omp for
8          for SubsetIndex in 0 to Done[SubsetSize]-1 do
9              Subset ← Subsets[SubsetSize][SubsetIndex]
10             for last in 1 to N-1 do
11                 if Subset & (1<<last) then
12                     dp[Subset][last] ← INF
13                     prev ← Subset-(1<<last)
14                     for secondLast in 1 to N-1 do
15                         if prev & (1<<secondLast) then
16                             if dp[Subset][last] > dp[prev][secondLast] +
                                cost[secondLast][Last] then
17                                 dp[Subset][last] ←dp[prev][secondLast] +
                                    cost[secondLast][Last]
18                                 parent[Subset][last] ← secondLast
19     for last in 1 to N-1 do
20         if minCost > dp[1<<N-1][last]+cost[last][0] then
21             minCost ← dp[1<<N-1][last]+cost[last][0]
22             lastIndex ← last
```

1.3 Performance Evaluation

The code was run on the cse machine 172.27.19.49. Here is the overall experimental setup:

- The execution time is only taken for the function 'Solve()' in the code.
- Sample test cases with random edge weights between 1 and 100 and for $N = 10$ to $N = 24$ were taken for testing and evaluation.

- The execution times are measured for thread count = 1, 2, 4 and 8. Although we tried for bigger thread counts as well but it seemed as if machine didn't support these many threads (had less number of cores) and hence the time was almost always greater than smaller thread counts.
- The execution times are measured as average of 10 executions.

Here are the performance statistics for the program described above (and submitted as final submission)
Note: The way we have parallelized the program allows calculation of TSP of upto 24 node graphs only. Beyond this size, the program runs into memory related issues. Execution times for $N = 10$:

Number of Threads	Execution Time (in μs)
1	255
2	253
4	242
8	1525

Execution times for $N = 15$:

Number of Threads	Execution Time (in μs)
1	11318
2	6360
4	3985
8	3450

Execution times for $N = 20$:

Number of Threads	Execution Time (in μs)
1	622361
2	331834
4	182209
8	123202

Execution times for $N = 21$:

Number of Threads	Execution Time (in μs)
1	1407986
2	748124
4	405789
8	270962

Execution times for $N = 22$:

Number of Threads	Execution Time (in μs)
1	3140183
2	1684550
4	904696
8	605850

Execution times for $N = 23$:

Number of Threads	Execution Time (in μs)
1	6983380
2	3725164
4	2022612
8	1335585

Execution times for $N = 24$:

Number of Threads	Execution Time (in μs)
1	15515952
2	8295695
4	4424397
8	2956555

Visualisation of performance:

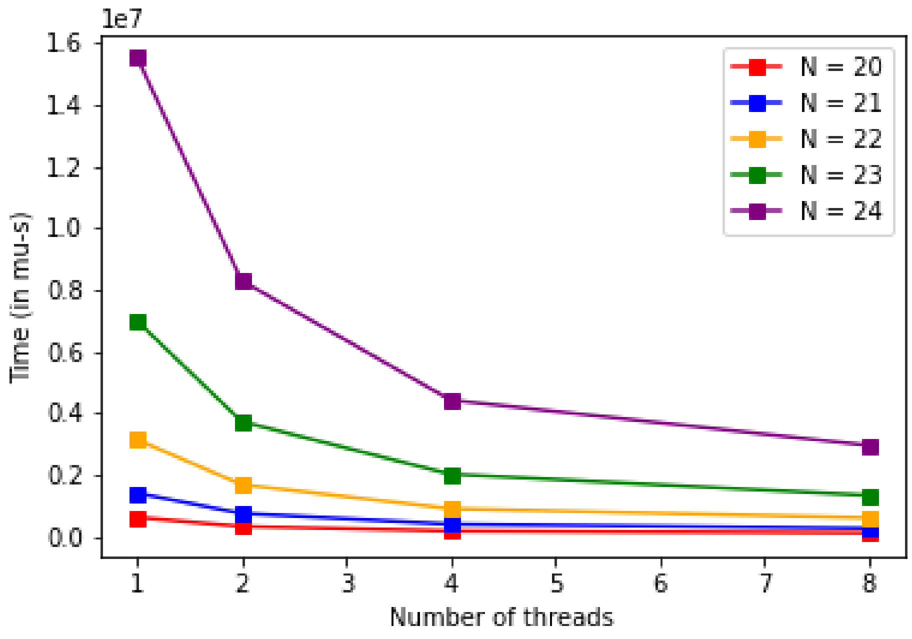

Figure 1: Time vs Number of threads

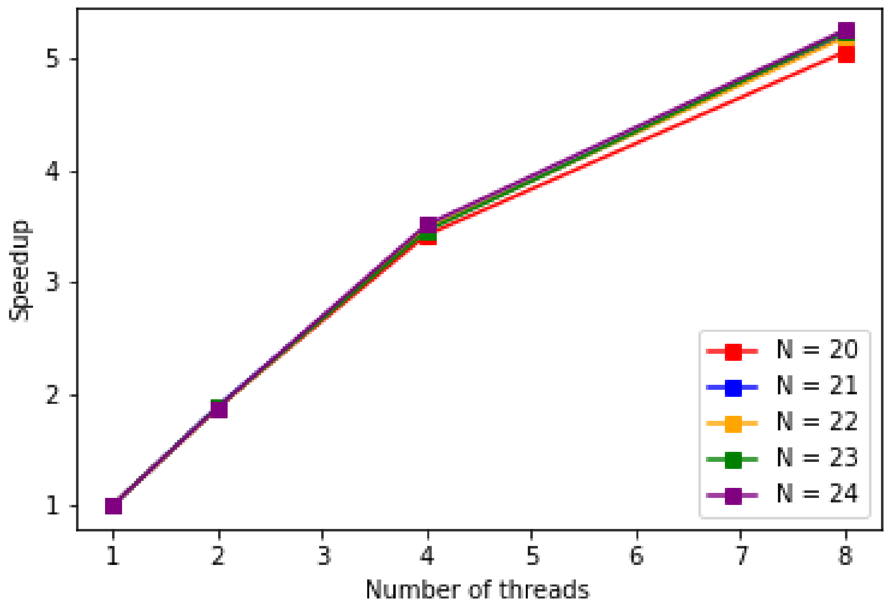

Figure 2: Speedup vs Number of threads

Discussion on trends:

- For a given graph size (which is large enough), as we increase the number of threads, execution time decreases (till a certain thread count). This because computation per thread reduces.
- As graph size increases, speedup for a given number of thread increases. This is because when we increase the graph size, the amount of computation to be done starts dominating over the overheads of parallelization : synchronization, creation, joining of threads etc.
- For a given graph size (which is large enough), the achieved speedup increases with the number of threads. However, the factor by which the speedup increases shows a decreasing trend. This is because even though the work is divided more but the overheads of parallelization are also increasing with increasing number of threads. Hence, if we have large amount of computation we are likely to notice increase in relative speedup compared to smaller amount of computation.

II Lower Triangular System of Equations - OpenMP

In this problem, we were asked to solve lower triangular system of equations. We are given a lower triangular matrix L with non zero diagonal entries and a column vector y. We need to determine column vector x such that $Lx = y$. We solved the problem using **C** language and using **OpenMP** framework.

2.1 Previous attempts

2.1.1 Forward substitution

- **Sequential Algorithm:** We tried the most basic approach initially which was just a normal forward substitution algorithm. We first calculate x[0] from the first equation (where the only variable is x[0]). We then substitute x[0] in the remaining equations. Now, equation 2 has one variable (x[1]). We keep repeating this process, substituting x[i] in equations i+2 to N to obtain all unknown entries. This ends the description of the sequential algorithm for forward substitution.

- **Parallel Algorithm (1D Flag synchronization):** We discuss the first way to parallelize the sequential algorithm described above. We maintain a 1D flag array of size N (all entries initialized to false). Flag[i] = true means that x[i]'s value has been computed and is stored in y[i]. We assign different rows to different threads (scheduling static, 1 to assign consecutive rows) and within a row, we iterate over columns (we wait till the corresponding x value for that column has been calculated using while loop). Once calculated, we subtract the corresponding entry from the output vector (y[row]) with the product of the x[col] calculated and the corresponding matrix entry (L[row][col]). In the end, we divide y[row] by L[row][row] to get x[row] and set the corresponding entry of this row in the flag array to true.

Problems with the approach: This approach looks promising as we are using (staticc, 1) scheduling and hence tackling the issue of load imbalance in this case of triangular matrix. However, because of the technique used for synchronization i.e. keeping the threads waiting till the time calculation of a row is done using while loop, there is a lot of busy waiting time in each thread. This leads to less efficient parallelization. Also, using an extra array to store flag entries leads to memory overhead and updation of adjacent entries of flag array by different threads everytime (since rows are distributed to threads) also leads to lot of false sharing in this array.

2.1.2 Gaussian elimination

- **Sequential Algorithm:** This algorithm is a standard procedure to apply gauss elimination for a lower triangular matrix. We set pivots as diagonal elements and with the help of those reduce the other entries in the column to be zero. The algorithm runs as follows: We first compute x[0] (divide y[0] by L[0][0]) . Then, subtract L[row][0]×x[0] from y[row] for row in 1 to N-1. Then, we compute x[1] (divide y[1] by L[1][1]). Then, we subtract L[row][0]×x[1] from y[row] for row in 2 to N-1. We keep repeating this process till we obtain all x[i]'s.

- **Parallel Algorithm:** Notice the fact that we can parallelize the process of zeroing of entries of a column with the help of the diagonal pivot. For this, we iterate over columns sequentially for all threads. One of the thread computes x[0] (as discussed earlier). Next we distribute the rows to different threads in static fashion and using this, subtracting L[row][0]×x[0] from y[row] for row in 1 to N-1 is done parallely by threads (different threads subtract for different rows). This process continues, the subtraction of y[row] for different rows is parallelized.

Problems with this approach: An observation on execution times and speedup here was that for small matrices speedup was more as compared to larger matrices. This should be mainly due to cache effects. If the matrix dimension is large, when a thread does multiple subtractions, it loses/evicts cache blocks of the rows that were in it's cache earlier!. When it tries to subtract from the same row in the next column, we no longer have the block in cache. The final attempt discussed below circumvents this by working on chunks of columns at one time.

2.2 Final attempt

2.2.1 Sequential algorithm

We adapt the above explained algorithm for gaussian elimination to behave in a more cache friendly manner. The main issue with the above explained algorithm was that there will be lot of eviction/cache misses in the case of large size of matrix and this affects the attempted parallel algorithm to quite an extent. To resolve the issue, we tried a different sequential algorithm. In this algorithm, we divide the lower triangular matrix into regions of trapezoids. We first solve for the unknowns $x[i]$ using just the upper right angled triangle regions and then propagate these values down the trapezoid to solve for the unknown left. The algorithm runs as follows with Figure 3 being our matrix:

1. Make the diagonal elements of triangle 1 as pivot and make all other entries inside the triangle 1 as 0 and also perform similar operation with the output vector y (Just like in Gauss Jordan Elimination, same operation is performed on both the sides of equation). Obtain the value of unknowns corresponding to rows in triangle 1.

2. Use the obtained values in step 1, to make the entries in the rectangle 2 as zeros, while performing similar operation with output vector y as well. In this step, the difference is that we will move from right to left for each row and make the corresponding entries 0 instead of doing this thing for a single column at a time.

3. Repeat the above two steps, till the point when all the unknowns have been found.

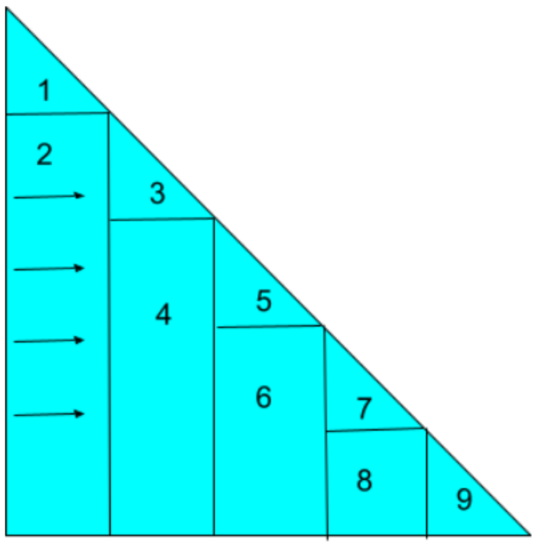

Figure 3: Order of execution

Algorithm 3: Sequential algorithm for Lower Triangular Matrix Solver

```
Input: Matrix dimension : N, Matrix L, Vector y
Output: Vector x such that Lx = y
1  C is chunk size
2  Function SolveLowerTriangle():
3      if $N\%C == 0$ then
4          times ← N/C
5      else
6          times ← N/C + 1
7      for i in 0 to times - 1 do
8          for col in $i \times C$ to min(N, $(i+1)\times C)-1$ do
9              $y[col] \leftarrow (y[col])/(L[col][col])$
10             for row in col+1 to min(N, $(i+1)\times C)-1$ do
11                 $y[row] \leftarrow y[row] - y[col] \times L[row][col]$
12         for row in $(i+1)\times C$ to N-1 do
13             for col in $i \times C$ to min(N, $(i+1)\times C)-1$ do
14                 $y[row] \leftarrow y[row] - y[col] \times L[row][col]$
```

2.2.2 Parallel algorithm

In the parallel counterpart, we still compute the pivots in a sequential fashion in each iteration (only one of the thread computes this). In other words, computation of triangle is sequential as described in the previous sub section. Once we have all the pivots computed for that chunk, notice that we can now do parallel subtractions. In the rectangles, we assign threads to rows. This will resolve the cache getting evicted problem because a thread processes all elements of the block while it is in the cache. Since the workload is similar across rows, we adapt the default scheduling for the '#pragma for' directive. The order of execution can be understood from the Figure 3:

- Compute Triangle 1 sequentially as described above

- Distribute rows of Rectangle 2 to different threads, arrows represent how a thread proceeds.
- Compute Triangle 3 sequentially
- Distribute rows of Rectangle 4 to different threads, arrows represent how a thread proceeds.
- Repeat the above steps till all unknowns have been found.

One of the parameters that is of importance here is chunk size : number of columns that we are processing in one iteration. We can see a trade-off here. If the chunk size is too large, the triangular portion becomes too large and hence the computation becomes more sequential (in the limit when chunk size is dimension of the matrix, this becomes completely sequential algorithm!). If the chunk size is too less, we will suffer cache misses on large matrices. The case of chunk size = 1 is essentially the parallelized Gauss algorithm described above. We evaluated performance at chunk sizes = 8, 16, 32 and 64. The results in the performance section are for chunk size = 16 (and in the submitted code). On running the algorithm, it was also found that the execution time decreased by quite a good factor as compared to the naive parallel execution of this algorithm. We tried the simple parallelization initially with '#pragma omp parallel for' above the loop in which we are iterating on the rows of rectangle, but the problem with this is that after every iteration of the loop, we will create new set of threads for the next iteration. Instead we optimized the code by creating the threads first and then using these threads in parallel for the inner for loop. To avoid false sharing, we padded the *y* array but results were almost similar (hence not in final submission).

Algorithm 4: Parallel algorithm for Lower Triangular Matrix Solver

```
Input: Matrix dimension : N, Matrix L, Vector y
Output: Vector x such that Lx = y
C is chunk size
Function SolveLowerTriangle():
    if N%C == 0 then
        times ← N/C
    else
        times ← N/C + 1
    #pragma omp parallel nun_threads (n_threads)
    for i in 0 to times - 1 do
        #pragma omp single
        {
        for col in i × C to min(N, (i+1)× C)-1 do
            $y[col] \leftarrow (y[col])/(L[col][col])$
            for row in col+1 to min(N, (i+1)× C)-1 do
                $y[row] \leftarrow y[row] - y[col] \times L[row][col]$
        }
        #pragma omp for
        for row in (i+1)×C to N-1 do
            for col in i × C to min(N, (i+1)× C)-1 do
                $y[row] \leftarrow y[row] - y[col] \times L[row][col]$
```

2.3 Performance Evaluation

The code was run on the cse machine 172.27.19.49. Here is the overall experimental setup:

- The execution time is only taken for the part of code where solving of $Lx = y$ is done.
- Sample test cases with size of matrix as $N \times N$ and value of N as powers of 2 from 2^{10} to 2^{15} were taken for testing and evaluation with a custom initialization function (not provided with the final submission).
- The execution times are measured for thread count = 1, 2, 4 and 8. Although we tried for bigger thread counts as well but it seemed as if machine didn't support these many threads (had less number of cores) and hence the time was almost always greater than smaller thread counts.
- The execution times were measured as average of 15 executions with chunk size 16.

Here are the performance statistics for the program described above (and submitted as final submission)
Execution times for $N \times N$ matrix with $N = 1024$:

Number of Threads	Execution Time (in μs)
1	1279
2	848
4	703
8	2280

Execution times for $N \times N$ matrix with $N = 2048$:

Number of Threads	Execution Time (in μs)
1	6939
2	3973
4	3020
8	2842

Execution times for $N \times N$ matrix with $N = 4096$:

Number of Threads	Execution Time (in μs)
1	28763
2	15850
4	8793
8	8906

Execution times for $N \times N$ matrix with $N = 8192$:

Number of Threads	Execution Time (in μs)
1	119254
2	65666
4	36472
8	33178

Execution times for $N \times N$ matrix with $N = 16384$:

Number of Threads	Execution Time (in μs)
1	540615
2	292941
4	169551
8	132000

Execution times for $N \times N$ matrix with $N = 32768$:

Number of Threads	Execution Time (in μs)
1	3489050
2	1990395
4	1097816
8	819875

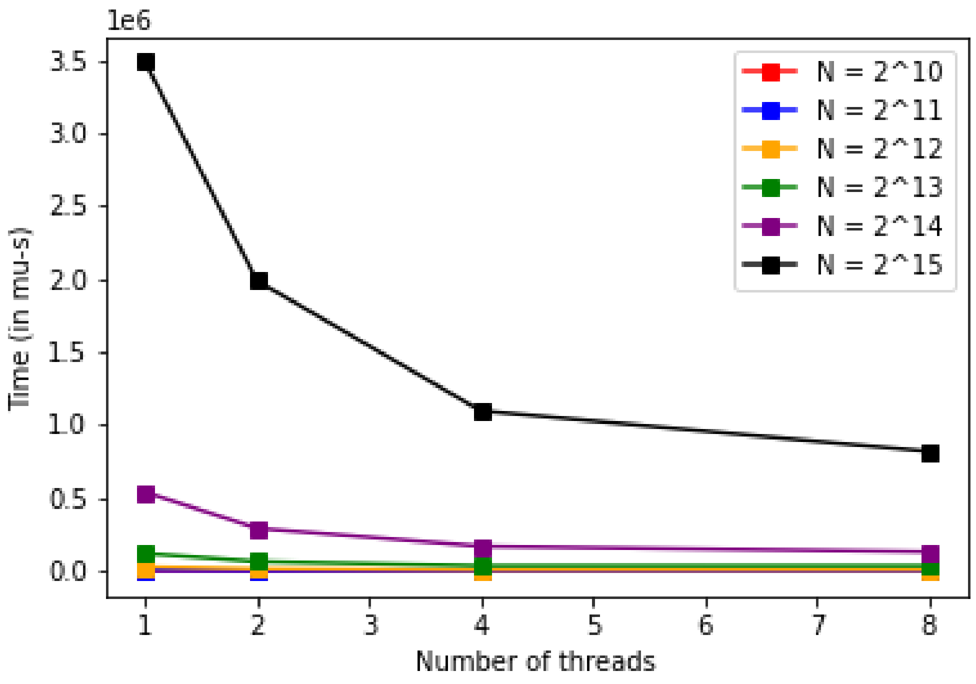

Figure 4: Time vs Number of threads

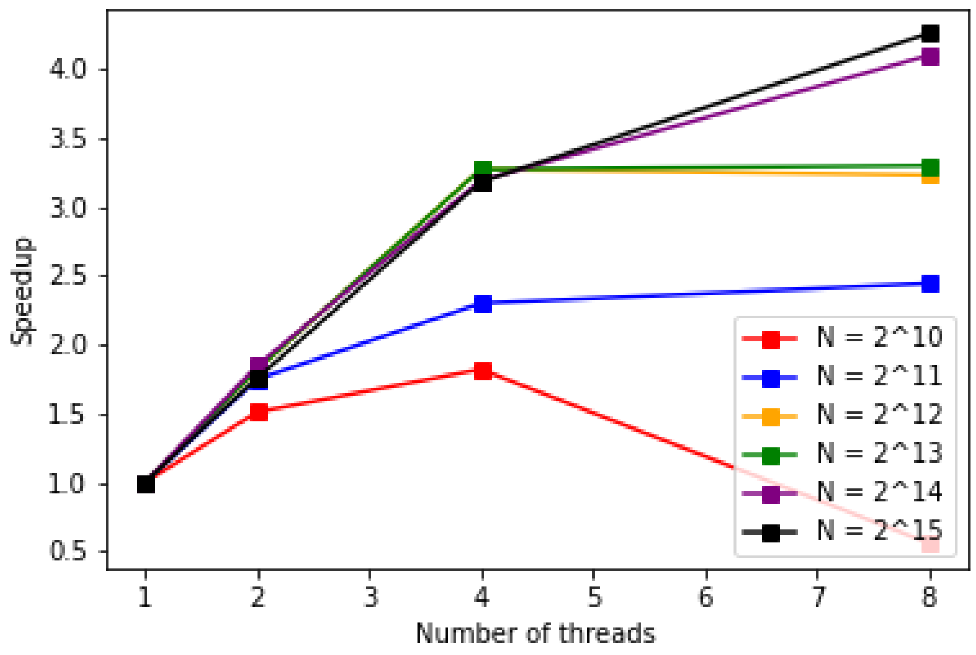

Figure 5: Speedup vs Number of threads

Discussion on trends:

- For a given matrix size (which is large enough), as we increase the number of threads, execution time decreases (till a certain thread count). This because computation per thread reduces.
- As matrix size increases, speedup for a given number of thread increases. This is because when we increase the matrix size, the amount of computation to be done starts dominating over the overheads of parallelization : synchronization, creation, joining of threads etc.
- For a given matrix size (which is large enough), the achieved speedup increases with the number of threads. However, the factor by which the speedup increases shows a decreasing trend. This is because even though the work is divided more but the overheads of parallelization are also increasing with increasing number of threads. Hence, if we have large amount of computation we are likely to notice increase in relative speedup compared to smaller amount of computation.
- It was also observed that the best execution time for different matrix sizes was achieved with different number of threads. This is also because of the tradeoff between computation size and synchronization overheads.

III Compilation procedure

3.1 For Question 1

Compile with the following command ‘gcc -O3 -fopenmp A1_Q1_final.c -o A1_Q1_final’

3.2 For Question 2

Compile with the following command ‘gcc -O3 -fopenmp A1_Q2_final.c -o A1_Q2_final’

References

[1] Travelling salesman problem using Dynamic Programming. https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/.