Assignment-1

Tushar Singla (190918)
Ketan Chaturvedi (190428)

Q1 Travelling Salesman Problem

We have solved the problem using a dynamic programming algorithm. Without loss of generality we assume to start our tour from the first city. We define C(i, S) as the cost of the minimum cost path visiting each vertex in set S exactly once and starting at 1 and ending at i.
Clearly set of size 2 will be {1, i}, and C(i, S) = weight[1][i]
Else
C(i, S) = min {C(j, S-{i}) + weight[j][i] } where j ∈ S and j != i and j != 1
We have found all the C(i, S) recursively and store them in a 2D matrix state.

To optimise for memory we have used bitmasks to represent S in our program.
We have noticed that if we choose different ending points then all the subproblems would be independent and we could easily parallelize our program by parallelizing the first for loop.
Also, notice critical section is required during updation of state[0][1] and path[0][1] as multiple threads could try to update this at the same time. Also, this critical section will be accessed maximum V(number of vertices) times. To optimise wait time we are first checking if (currWeight < state[currPos][visited]) before entering the critical section and then checking the condition on entering it.
We have used static scheduling because work in all the iterations of the for loop is roughly the same.

We have parallelized the program using OpenMP directives.

Below is the table representing the time taken by our program for different graph sizes and different number of threads on a 12 core Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz shared machine with 64 GB of RAM being used by other users as well.

Number of Vertices	1 thread	2 threads	4 threads	8 threads
22	6.440771	3.246125	1.760826	1.305106
23	14.960079	7.681821	4.119568	2.987830
24	33.823790	17.267171	9.336339	6.681912
25	76.351581	39.666622	21.554723	14.894642
26	176.278856	92.230756	48.311171	34.727093

As we can see from above results, time taken halves roughly with doubling core counts meaning we have efficiently parallelized the program.
One issue with the dynamic programming approach is that space complexity is exponential meaning we cannot execute it for large graph sizes. For a graph of size 26 our program required around 16 GB of memory and it roughly doubles on increasing graph size by 1.

Solution - 2

Sequential Algorithm

We are given following matrix equation:

$$\begin{matrix} L & & x & = & y \end{matrix}$$

$$\begin{bmatrix} l_{00} & 0 & 0 & \dots \\ l_{10} & l_{11} & 0 & \dots \\ l_{20} & l_{21} & l_{22} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \end{bmatrix}$$

Following algorithm is being used to calculate matrix X:

1. On multiplying matrices L and x, we get following equations:

$$l_{00}x_0 = y_0$$

$$l_{10}x_0 + l_{11}x_1 = y_1$$

$$l_{20}x_0 + l_{21}x_1 + l_{22}x_2 = y_2$$

$$\dots$$

2. Now x_0 can be easily obtained as y_0/l_{00}. Subtract the $l_{i0}x_0$ expression from of all equations to get following equation set.

$$l_{11}x_1 = y_1 - l_{10}x_0$$

$$l_{21}x_1 + l_{22}x_2 = y_2 - l_{20}x_0$$

$$l_{31}x_1 + l_{32}x_2 + l_{33}x_3 = y_3 - l_{30}x_0$$

$$\dots$$

3. Now similar to above step, x_1 can be obtained and $l_{i1}x_1$ can be subtracted from all equations to get further equation set. This can be recursively used to found all the elements of matrix X.

4. So the pseudo code of the above algorithm boils down to the following:

Algorithm 1 Find matrix X

```
procedure MAIN
    n ← Dimension of array
    L[n][n] ← L matrix of dimension n × n
    X[n] ← Empty array of dimension n × 1
    Y[n] ← Y of dimension n × 1
    for i ← 1 to N do                                   ▷ Loop 1
        X[i] = Y[i]/L[i][i]
        for j ← i to N do                               ▷ Loop 2
            Y[j] -= X[i] * L[j][i]
        end for
    end for
end procedure
```

Optimizations

1. For storing L, we are not allocating $n \times n$ space. As L is lower-triangular, actual space used by it is $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$. So we allocated a pointer ptr with this size and $L[i]$ is defined as $L[i] = ptr + \frac{i(i+1)}{2}$. So in this manner $L[i]$ actually gets the size i instead of n.
2. We used **OpenMP** directive to parallelize the inner loop (Loop 2). The $l_{ji}x_i$ subtraction from equations is independent and can be done in parallel. We used static assignment to assign a fixed number of rows to each thread.

Results

Experiments were run on 12 core Intel i7-9750H processor with 16GB of RAM. For each thread and dimension combination, experiment was performed 10 times and the average values (in microseconds) are presented below:

Dimension	1 thread	2 thread	4 thread	8 thread	12 thread
4096	35695	16520	9700	9129	11587
8192	157358	82873	35562	32657	31796
16384	729133	417389	209271	182691	172636
32768	3582387	2042858	1094201	1009373	954365
45000	7412348	4306751	2256774	2172779	1878931

In case of $n = 4096$, after 8 thread, performance reduces. The reason might be as 12 threads are created on a 12 core machine, other processes might deschedule the threads resulting in bad performance. Other reason can be that thread creation/destruction overhead is considerable.
In other cases, performance continues to increase as threads are increased. This is because the subtraction operations can be further divided and completed in shorter time.