CS433
Design Assignment - 1

190772
Sarthak Rout
sarthakr@iitk.ac.in
Group - 21

March 8, 2022

Q1

Description of Algorithm:

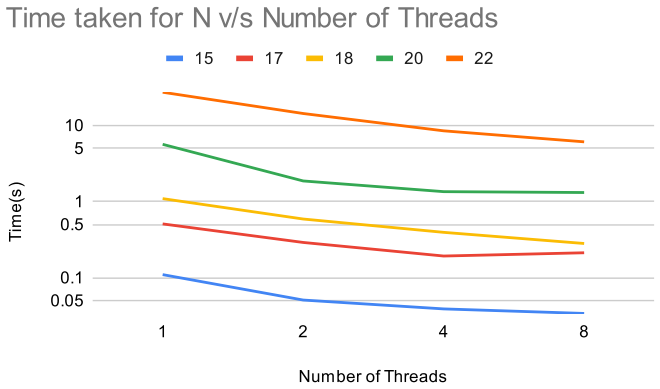
- For the symmetric TSP problem, the solution approach involves dynamic programming where we consider pairs of subsets of visited vertices and the vertex that the tour ends after starting from the source vertex.
- An initial implementation consisted of iterating over states and possible end vertices in which we iterate over vertices that we wish that should be visited next. This is a $\mathcal{O}(n^2 2^n)$ algorithm consisting of $\mathcal{O}(n2^n)$ subproblems which is equal to the memory complexity of the algorithm. Also, the states are stored using bitmasks where each bit signifies whether a particular vertex is present in that subset. The source vertex is always considered present.
- To parallelise it, we divided the algorithm into phases where we process states of same size together as these updates are independent of each other. This is due to the fact that in each update, the new state contains a new vertex that is visited (which increases the size of that state). The states can be possibly of sizes ranging from 0 to n and for each size k, there can be $\binom{n}{k}$ possibilities of subsets/states.
- To generate bitmasks, at bit hack was used that generated permutation of bits, having the same number of set bits. The state updates for a particular size sz are completely independent and identical due to which they can be completely parallelized. Then, OMP parallel directive with default static scheduling was used while iterating over those states.
- At the end, to find the minimum cost and the corresponding tour, we needed to check each vertex starting from the complete subset in a backwards manner; to find the vertex that was added in the last. A stack was used to store the vertex and print in the reverse order.

Results

Computer Specifications:

- Intel 8th Gen i5 Quad Core processor with 8 Threads
- x86-64 architecture
- 32kB L1 cache; data and instruction each
- 8GB RAM

Logarithmic plot of the time taken by the algorithm

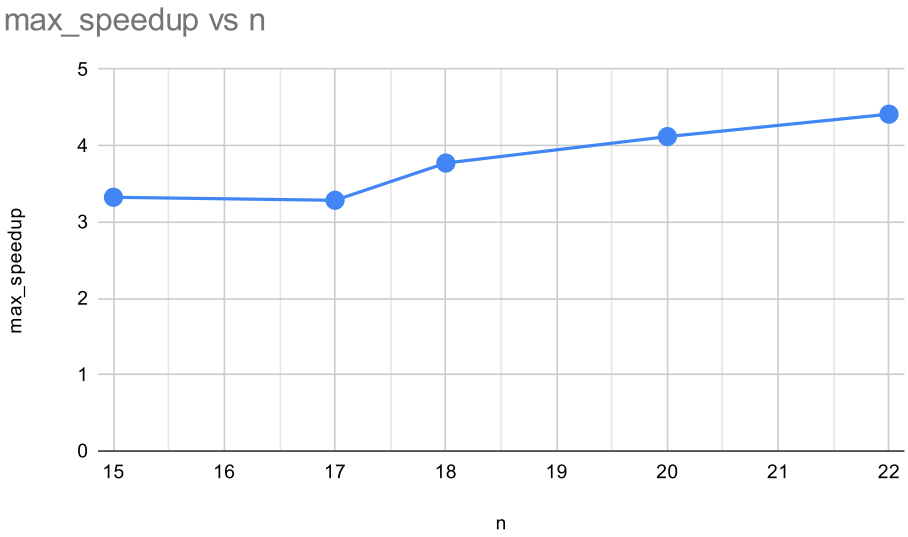


Results of experiments with different number of threads and different graph sizes in seconds:

n	1	2	4	8
15	0.113	0.059	0.039	0.034
17	0.509	0.292	0.18	0.155
18	1.139	0.616	0.394	0.302
20	5.531	2.994	1.77	1.343
22	27.897	14.451	8.247	6.323

Observations

- This problem is not inherently parallelizable; The sequential algorithm couldn't be directly parallelized. We had to break it over steps that were independent of each other.
- As the dynamic programming algorithm is exponential in time complexity, there is a lot of potential for parallelization to provide significant speedup. Using 8 threads with OpenMP gives around a speedup of a factor of 4. We also verify the exponential time complexity from the table of experimental times with a single thread.



$$\text{Max Speedup} = \frac{\text{time for 1 thread}}{\text{time for 8 threads}}$$

- But in our approach, the logarithmic plot shows that we are saturating on the amount of speedup which will be bounded by 5 or 6. This shows that a significant fraction of the task is still sequential. We also observe that we parallelize with the OMP directive $\mathcal{O}(n)$ times which is expensive due to creation of implicit barriers.
- We also observe that we are unable to take full advantage of caches as the states that are updated are not closer to new states as they are indexed by the bit representation. We only take partial advantage with the knowledge that the new state will only have one more set bit than the previous state. But, the set of new states is very large (of exponential order).

Q2

Description of the algorithm

- The algorithm to solve the system of equation requires to solve for the values of x sequentially from the top to bottom where x[i] is solved as:

$$x[i] = \frac{y[i] - \sum_{j=1}^{i-1} L[i][j] \cdot x[j]}{L[i][i]}$$

- Initially a parallel program was implemented using OMP with default scheduling that reduced the value of $y[i]$ with $L[i][j]*x[j]$ whenever $x[j]$ is solved by a thread. Herein, the value of $x[i]$ is computed by dividing the residual value of $y[i]$ with $L[i][i]$. It displayed some amount of speedup.
- To further improve the performance, we solve a block of values of $x[i]$; a small lower triangular block using a single thread. After solving those values; say from $x[b*block\_size]$ to $x[(b+1)*block\_size-1]$, the values of $y[i]$ are correspondingly decreased by the $L[i][j]*x[j]$ terms for each $j \in [b*$block_size, $(b+1)*$ block_size $- 1]$ and for each $i >= (b+1)*block\_size$. These operations are independent of each other and can be completely parallelized.

Optimal Block Size

The optimal value for the size of a block is of the order of the square root of the size of the matrix; $\mathcal{O}(\sqrt{n})$ as evident from the derivation of the expression for speedup s:

$$s = \frac{\frac{n(n+1)}{2}}{\frac{n}{b}\left(\frac{b(b+1)}{2}\right) + \frac{\frac{n(n+1)}{2} - \frac{n}{b}\cdot b(b+1)/2}{T} + Z}$$

where b stands for block size, T stands for number of threads, and Z represents the overhead for L1 cache misses for all blocks.
Expression for Z as sum of number of rows in each iteration divided with number of rows present in a cache block:

$$Z = \sum_{i=1}^{\frac{n}{b}} \frac{n - i\cdot b}{\left\lceil \frac{L_1}{n * \text{sizeof(float)}} \right\rceil} \propto \frac{n}{b} \cdot \frac{n-b}{2} \cdot \left\lceil \frac{L_1}{n * \text{sizeof(float)}} \right\rceil^{-1}$$

When $n << L_1$, much of the data fits into the L0 cache and we needn't worry about cache misses; then

$$Z \propto \frac{n^3}{bL_1}$$

. But, when n is comparable to L_1, or much larger than L_1, there are going to be lots of cache misses every now and then;

$$\left\lceil \frac{L_1}{n * \text{sizeof(float)}} \right\rceil = \mathcal{O}(1) \implies Z \propto \frac{n^2}{b}$$

Therefore, when n is comparable or larger than L_1,

$$\implies s = \frac{\frac{n(n+1)}{2}}{\frac{n(b+1)}{2} + \frac{\frac{n(n+1)}{2} - \frac{n(b+1)}{2}}{T} + \frac{n}{b}\cdot\frac{n-b}{2}}$$

$$\implies s = \frac{n+1}{b+1+\frac{n-b}{T}+\frac{n-b}{b}}$$

The expression involving b in the denominator:

$$b \cdot (1 - \frac{1}{T}) + \frac{n}{b}$$

Using AM-GM Inequality, we can find the b which minimises the expression:

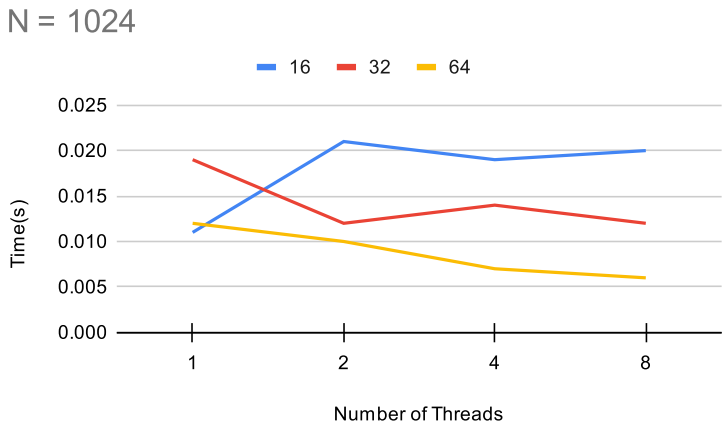
$$b = \sqrt{\frac{n}{1 - \frac{1}{T}}}$$

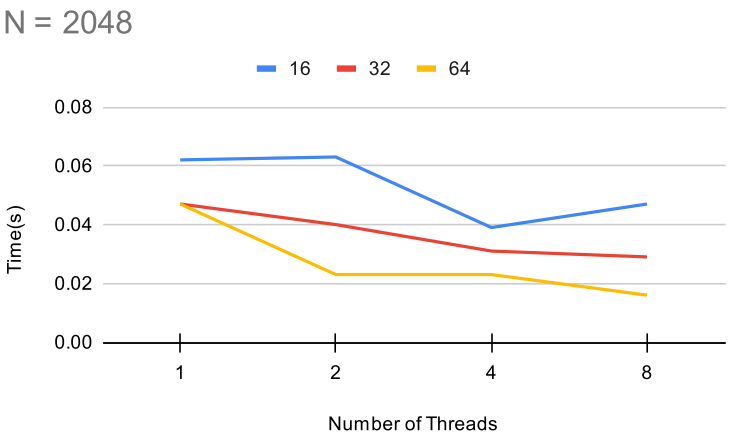
Results

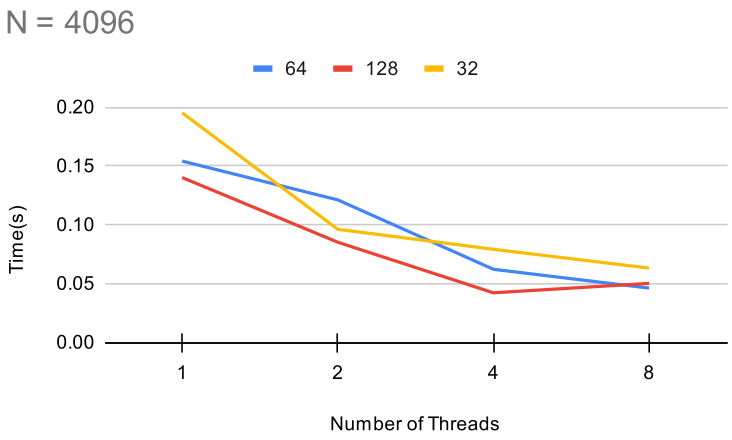
Computer Specifications:

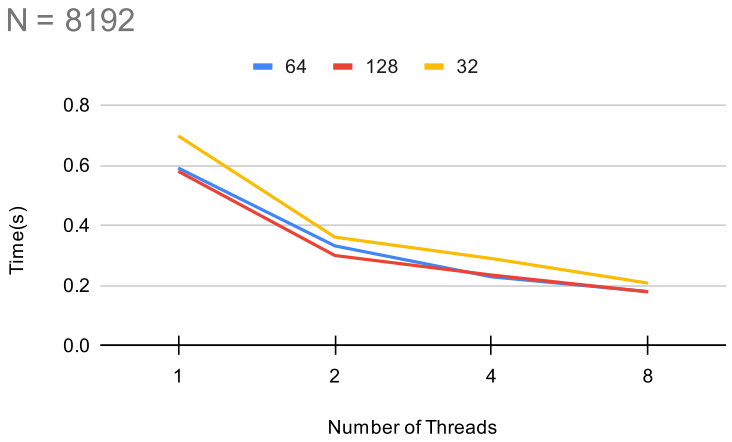
- Intel 8th Gen i5 Quad Core processor with 8 Threads
- x86-64 architecture
- 32kB L1 cache; data and instruction each
- 8GB RAM

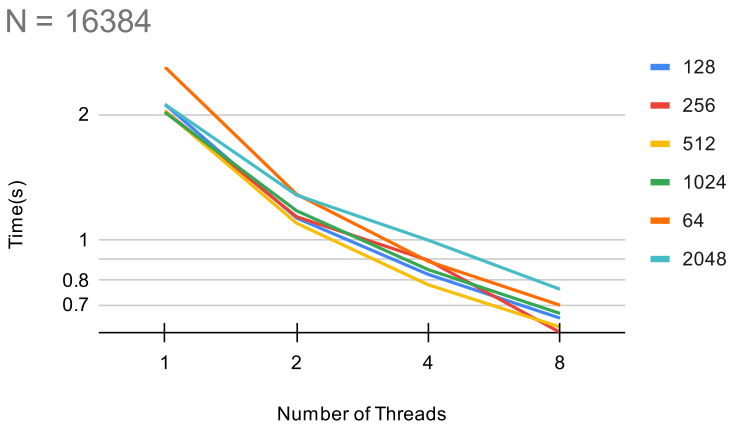
Variation of the time taken by the algorithm for different values of N with different block sizes and different number of threads:



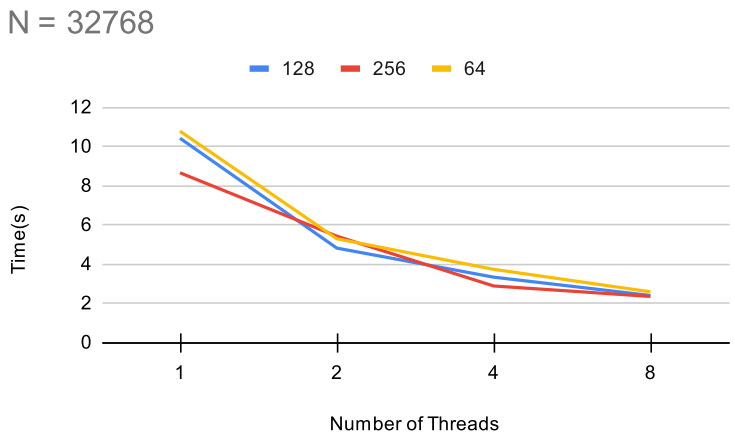








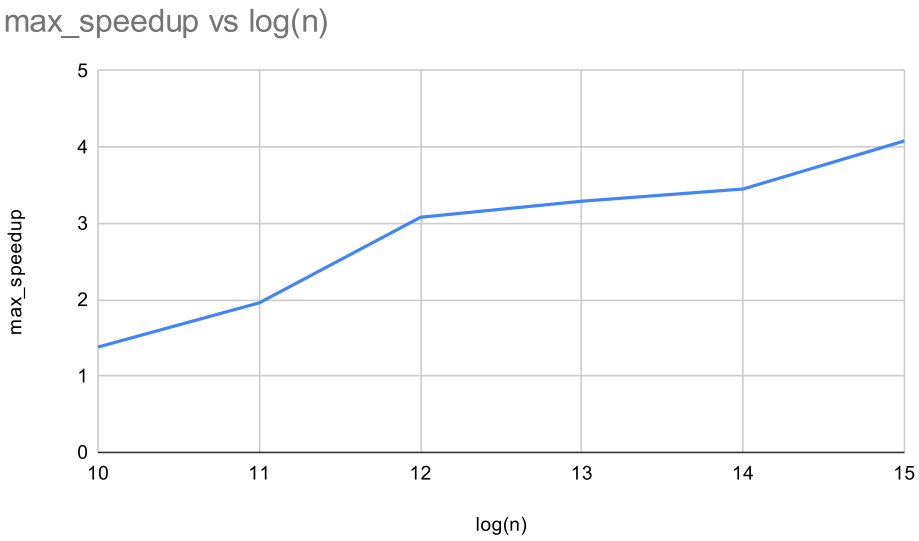
Logarithmic Plot for N = 16384



Data is presented in this spreadsheet
Table is present at the end of the report

Observations:

- We can observe that speedup is significant with larger values of n and larger number of threads as there is larger amount of parallelization. It is possible to obtain more speedup ratios with increasing n with respect to memory and cache size constraints. Smaller values of n don't exhibit good amount of speedup.
  For smaller sizes of matrices, it is beneficial to avoid parallelisation altogether and perform sequential operations or use smaller block sizes for parallelisation.



$$\text{Max Speedup} = \frac{\text{time for 1 thread}}{\text{time for 8 threads}}$$

- In all the plots, there is variation in performance even without parallelisation with $num\_threads = 1$, due to cache effects of choosing block size.
  Also, For $n = 16384(2^{14})$, we have a logarithmic plot of performance timings for several different block sizes and they clearly illustrate that optimal choice of block size does affect performance.

- With increase in n by factor of 2, the time for single thread roughly increases by a factor of 4 which shows that the underlying sequential algorithm has quadratic time complexity.
- For badly-conditioned system of equations, it is possible that the errors get accumulated and the values of x blow up. Such cases arise when the condition number is large; when pairs of rows/columns are close to linearly dependent.

Table of Time Taken by the Algorithm in seconds

n	blk	1	2	4	8
1024	16	0.011	0.021	0.019	0.02
1024	32	0.019	0.012	0.014	0.012
1024	64	0.012	0.01	0.007	0.006
2048	16	0.062	0.063	0.039	0.047
2048	32	0.047	0.04	0.031	0.029
2048	64	0.047	0.023	0.023	0.016
4096	32	0.195	0.096	0.079	0.063
4096	64	0.154	0.121	0.062	0.046
4096	128	0.14	0.085	0.042	0.05
8192	32	0.698	0.361	0.29	0.208
8192	64	0.591	0.332	0.229	0.18
8192	128	0.58	0.3	0.235	0.179
16384	64	2.597	1.287	0.889	0.698
16384	128	2.113	1.129	0.827	0.65
16384	256	2.038	1.137	0.894	0.601
16384	512	2.036	1.097	0.781	0.619
16384	1024	2.022	1.175	0.849	0.667
16384	2048	2.113	1.281	0.999	0.762
32768	64	10.766	5.279	3.718	2.571
32768	128	10.405	4.801	3.318	2.384
32768	256	8.643	5.41	2.869	2.334
32768	512	8.255	4.79	3.36	2.352