Masrik Dahir

Assignment 1 Report

Parallel TSP using MPI

Assumptions: The user has to input two arguments – the <number of blocks> and <number of cities per block>. Each block is a separate MPI process and the <number of cities per block> is the number of coordinates in each block. The city coordinates inside a block are randomly chosen within a range. The intention behind choosing a range is to ensure that the points I generate for each block are within each block's x, the, and y boundaries and are generated separately. Each block has an equal number of points.

The tsp uses multi-thread dynamic programming. Each block is using two threads to handle recursion (individual subproblem) in tsp.

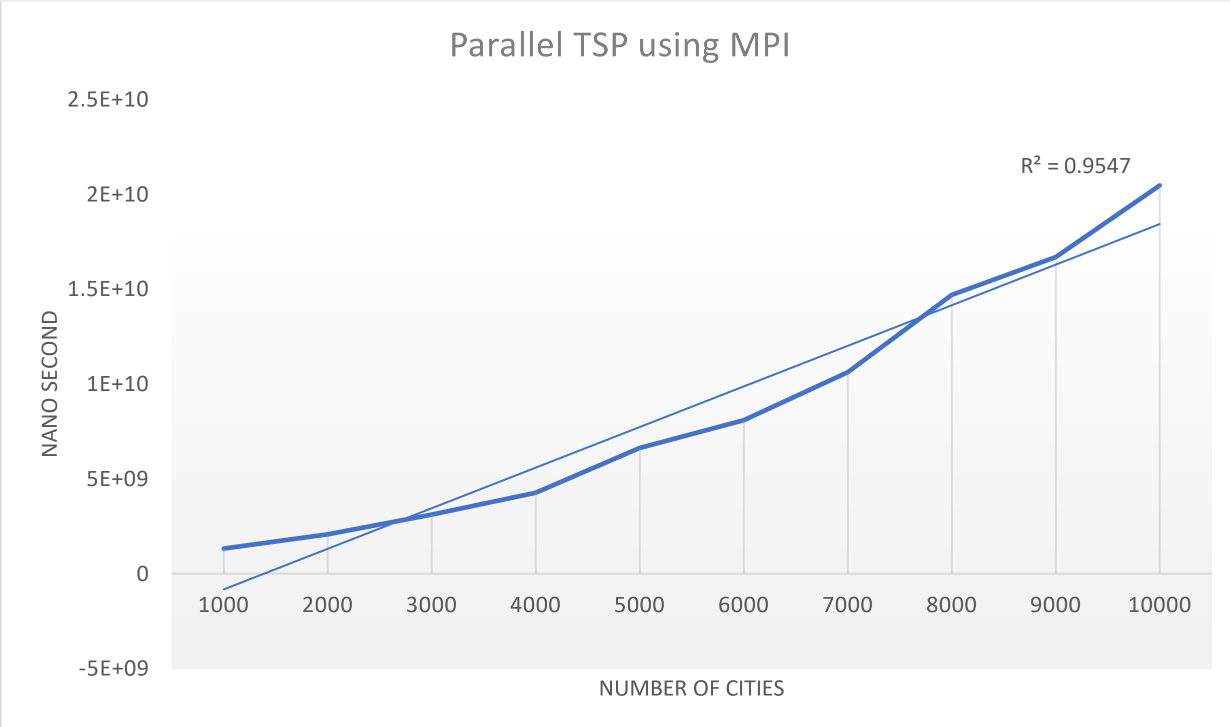
All the blocks are set up in b^2 blocks where each row has a master block. Every block on the row sends the tsp to the master block. The master row blocks use the stitching algorithm to add the tsp from the salve blocks to the master row block. This process is known as row-wise stitching.

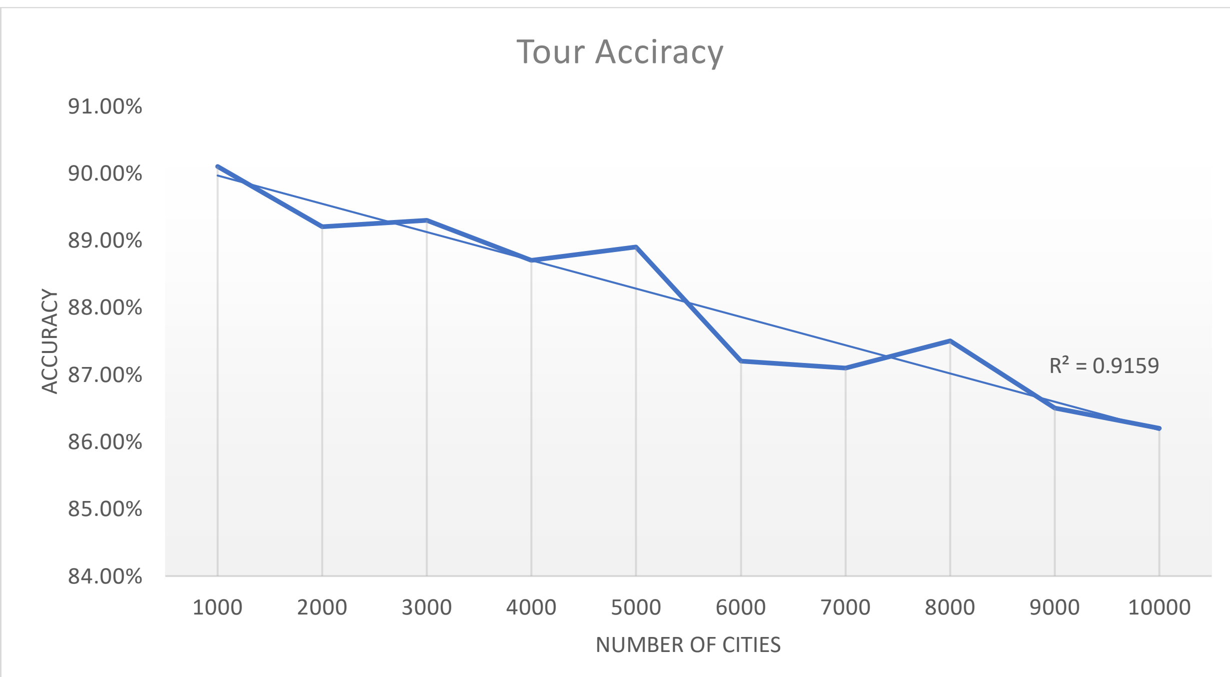
Once all the master row blocks receive the tsp from slave row blocks, the master row blocks send the tsp to the master column block (block 0) and the master column block uses the stitching algorithm to add all the tsp and create a single path. This process is known as column-wise stitching.

I divided the number of blocks by 4 to get the number of rows, so there would be <the number of blocks>/4 row-wise stitching. However, there would be only one column-wise stitching by block 0.

The accuracy drops gradually when the number of cities increases. I timed the execution and took records for a range of cities. The time column shows the execution time for the entire tsp (including stitching) execution time. The accuracy is calculated by dividing the number of current tsp paths after stitching by the length of the tsp and multiplying by 100. Since the Parallel TSP using MPI is not 100% accurate, I have done inversion to make it very close to accurate.

Number of Cities	Time (Nanosecond)	Accuracy
1000	1324557801	90.1%
2000	2072354801	89.2%
3000	3125389800	89.3%
4000	4268125600	88.7%
5000	6632152800	88.9%
6000	8097682399	87.2
7000	10631523301	87.1
8000	14697103600	87.5
9000	16690213601	86.5
10000	20463488600	86.2


Parallel TSP using MPI
2.5E+10
2E+10
1.5E+10
1E+10
5E+09
0
-5E+09
NANO SECOND
R² = 0.9547
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
NUMBER OF CITIES


Tour Accircacy
91.00%
90.00%
89.00%
88.00%
87.00%
86.00%
85.00%
84.00%
ACCURACY
R² = 0.9159
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
NUMBER OF CITIES

Comparing Parallel TSP using MPI with MPI-based TSP, Threaded TSP, and Serial TSP

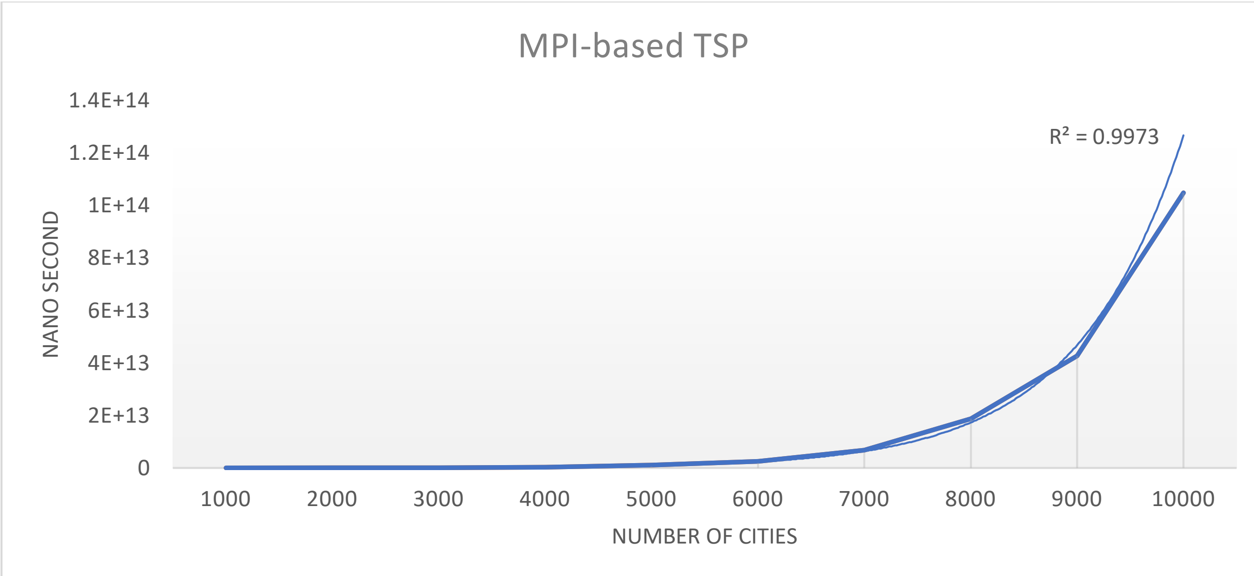
Parallel TSP using MPI is faster than TSP using MPI with MPI-based TSP, Threaded TSP, and Serial TSP (Assignment 1 solutions). Because the algorithm is running the recursion in a separate thread and a single block does not have to wait for stitching all the slave blocks tsp. Therefore, the tsp calculation is faster and the stitching algorithm also has very less wait time. Because each master row block does row-wise stitching and sends it to block 0 for only one column-wise stitching. Also, since separate threads are conducting row-wise stitching, the algorithms are working in parallel which also has less load, making the stitching algorithm overall very fast.

In terms of accuracy, the program is more accurate than MPI-based TSP and Threaded TSP. Because the stitching algorithm is happing at the row level with small arrays of tsp values. Therefore, the number of inversions required in Parallel TSP using MPI is less than MPI-based TSP and Threaded TSP. As the number of cities increases, the accuracy drops gradually. The Serial TSP has perfect accuracy but the algorithm works up to 24 cities. If the number of cities rises to 25, the program breaks (requires enormous time to executant)

MPI-based TSP

The algorithm creates arg1 threads and each thread creates a weighted adjacency matrix with arg2 coordinates. The master process is the main thread and the n-1 threads are the slave threads. The master processes and the n-1 process run in parallel. The master process has an ArrayList that collects the tsp from the slave process when the slave process is done. When the salve processes are done the thread is automatically destroyed. The master process then stitches the TSPs by just adding them arbitrarily (specifically whichever slave process finishes tsp first gets added first). Similar to ThreadedTsp, we use the Inversion algorithm to detect inversion and handle it. First. We detect the crossing of the two edges by using the Line2D library. The library then uses the current and next three coordinates to see if there are any crossings on the two closest edges. For reducing time complexity, we only compared neighbor edges for inversion. Ideally, each edge needs to be compared with all the other edges to find all possible inversions. Once an inversion is found, we swap the second node (city) with the third node (city).

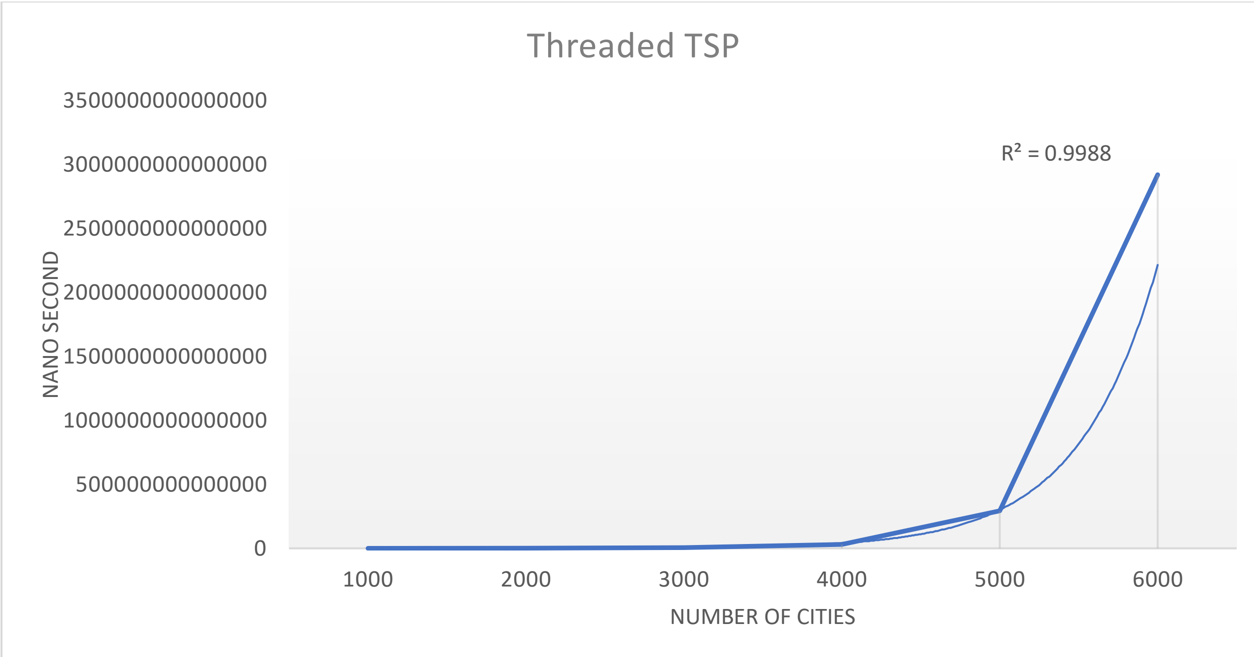
Number of Cities	Time (Nanosecond)	Accuracy
1000	13242378015	79.3%
2000	41447566127	78.6%
3000	125056592005	78.2%
4000	323450048008	78.9%
5000	1061144445600	77.7%
6000	2591258367635	77.8%
7000	6804152512640	77.4%
8000	18812292602640	77.3%
9000	42726463818566	76.1%
10000	102537616324576	76.2%



Threaded TSP

We ask for arg1 thread and arg2 cityPerThread argument from the user. The algorithm then creates arg1 threads and each thread creates a weighted adjacency matrix with arg2 coordinates. Then the threads run in parallel. A global ArrayList adds the individual tsp from the threads once each thread is done. We used a very simple stitching algorithm where we just simply add the tsp of all threads in the ArrayList. Then we use the Inversion algorithm to detect inversion and handle it. First. We detect the crossing of the two edges by using the Line2D library. The library then uses the current and next three coordinates to see if there are any crossings on the two closest edges. For reducing time complexity, we only compared neighbor edges for inversion. Ideally, each edge needs to be compared with all the other edges to find all possible inversions. Once an inversion is found, we swap the second node (city) with the third node (city).

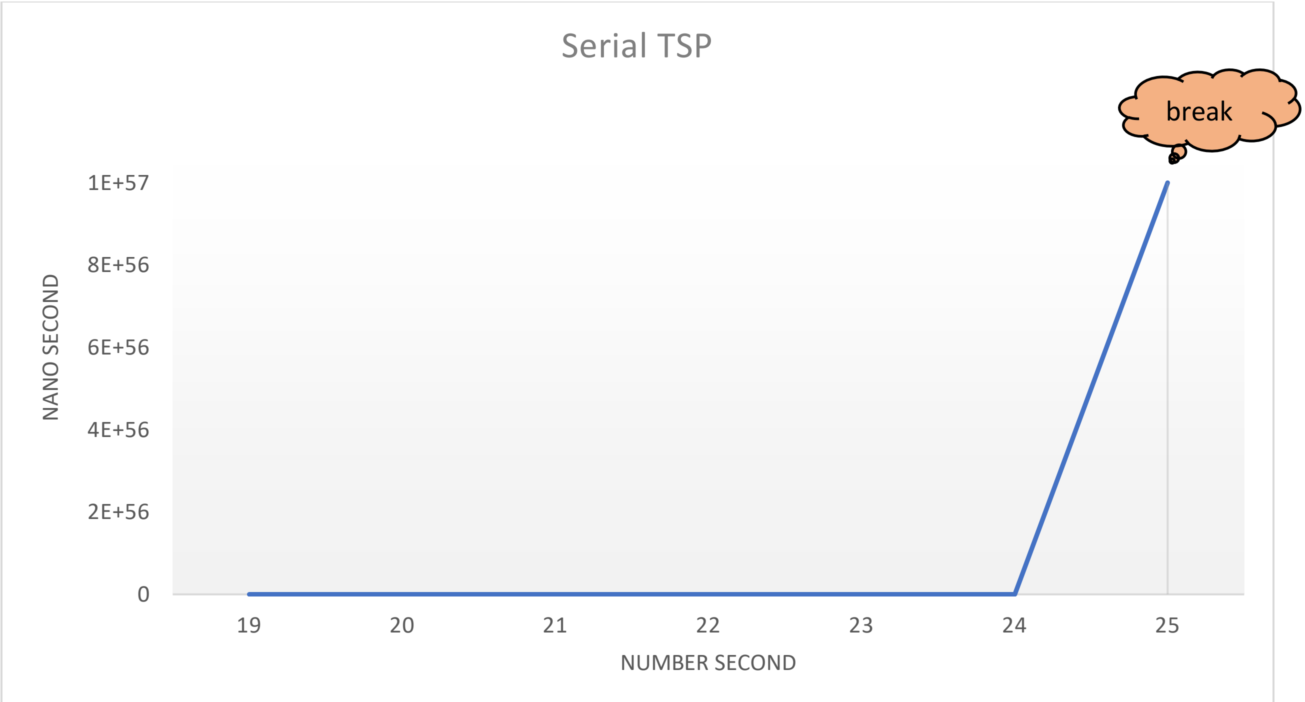
Number of Cities	Time (Nanosecond)	Accuracy
1000	132455780123	71.4%
2000	737045174583	70.9%
3000	4565245970185	70.2%
4000	30651531446312	70.2%
5000	293211335567517	69.7%
6000	2919231455533347	69.5%



Serial TSP

Once the weight is calculated for a coordinate, we calculate the weighted adjacency matrix. Then, we send the distance matrix to SerialTsp.getTour (). The getTour() function runs a while loop less than or equal to r and finds all possible combinations to get the minimum distance. Then it returns the Tour path by the nodes. The complexity of the algorithm is O(n!) to O(n^2 * 2^n). The algorithm is not efficient for a larger number of cities. Here is a sample chart of the number of cities and the amount of time it takes the algorithm to execute the program:

Number of Cities	Time (Nanosecond)	Accuracy
19	1026476200	100%
20	2492251601	100%
21	5642363900	100%
22	12107982300	100%
23	29969632601	100%
24	66730497699	100%
25	1E+57	Breaks!



Observation:

For the SerialTsp, the execution time is very higher but 100% accurate in finding tsp. However, over 24 cities the SerialTsp fails, which is an issue for finding tsp with a higher dataset.

But the threaded algorithm is more time efficient in execution but less accurate in finding tsp. We can do an inversion to make it more accurate but it is still not 100% accurate in finding tsp. Since the blocks are running on threads, there are fewer combinations of paths (for small n value) which reduces time complexity.

MPITsp is more time efficient than ThreadedTsp and has more tour accuracy than ThreadedTsp. Time efficiency depends on the number of cities in the block. More cities in the block lead to less time efficiency but more tsp accuracy. Since in MPITsp has a specific number of cities in every slave process, the accuracy is not as arbitrary as ThreadedTsp. The inversion also significantly improves the accuracy of the MPIthread.

Parallel TSP using MPI is more time-efficient than ThreadedTsp, MPITsp, and SerialTsp. It also has more accuracy than ThreadedTsp and MPITsp. The algorithm is running the recursion in a separate thread and a single block does not have to wait for stitching all the slave blocks tsp and each master row block does row-wise stitching and sends it to block 0 for only one column-wise stitching. Therefore, the parallel nature of the algorithm makes the program very fast in cost of CPU resources. Also, the row-wise and column-wise stitching makes it more accurate in finding tsp. That’s why we needed less inversion in Parallel TSP using MPI than MPITsp.