Travelling Salesman Problem: Parallel Implementations & Analysis

ME 766: High Performance Scientific Computing

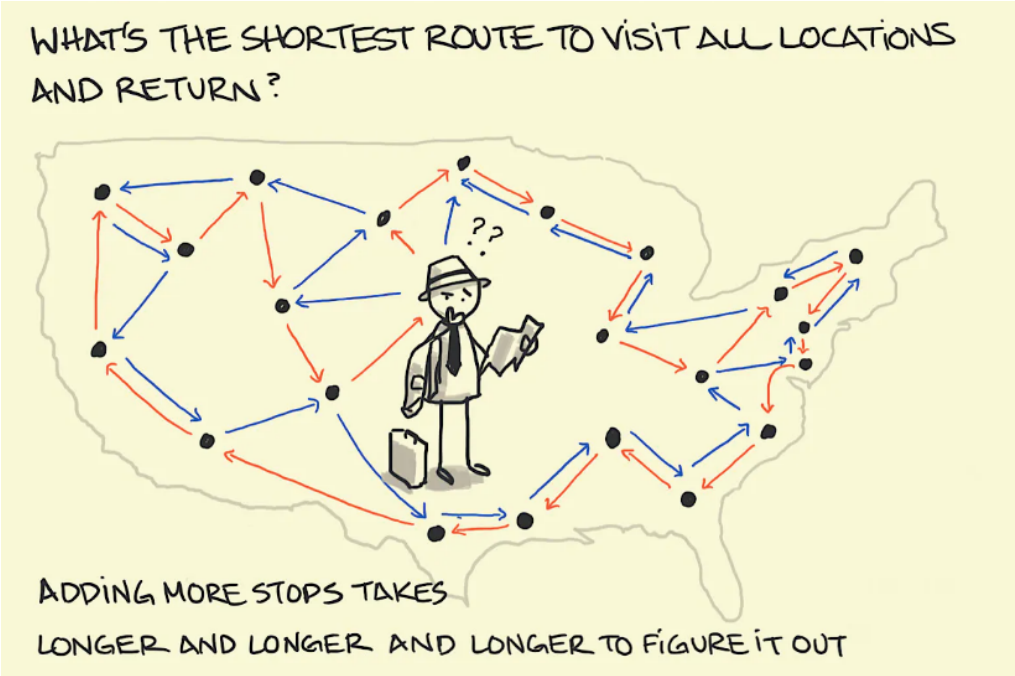
Amey Gohil	**Manan Tayal**	**Tezan Sahu**	**Vyankatesh S.**
170100019	170100005	170100035	170100034

12 May 2021

Travelling Salesman Problem

What is the problem?

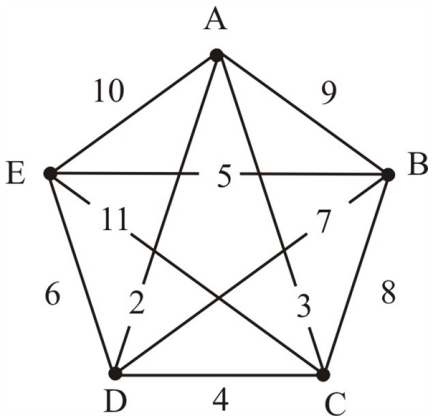
- The challenge of finding the shortest yet most efficient route for a person to take given a list of specific destinations along with the cost of travelling between each pair of destinations.
- The problem is an NP-Hard problem that is, no polynomial-time solution exists for this problem.



Solving TSP

Modeling the problem & Brute Force Approach to solve it

Graph Modelling



0	9	3	2	10
9	0	8	7	5
3	8	0	4	11
2	7	4	0	6
10	5	11	6	0

Table: Adjacency Matrix

Brute Force Algorithm

- Consider city A as a starting and ending city of the path
- Generate all $(N-1)!$ permutations of remaining cities *(here $N = 5$)*
- Calculate the cost of traversing a path corresponding to every permutation and keep track of minimum cost permutation (requires $O(N)$ time)
- Return the permutation with minimum cost *(*optimal path)
- Total Time Complexity: $O(N!)$

OpenMP (Shared Memory Processing)

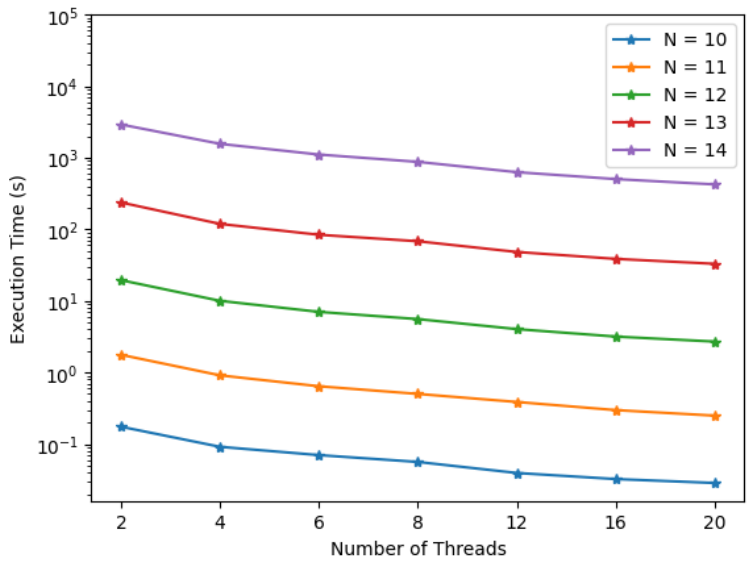


Figure: Variation of execution times with number of OpenMP threads for different problem sizes (N=Number of cities)

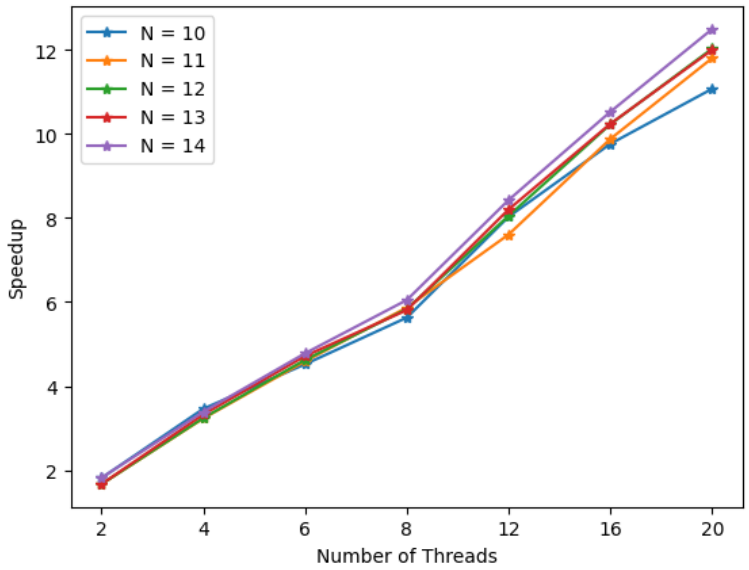


Figure: Variation of speedup achieved with number of OpenMP threads for different problem sizes (N=Number of cities)

- Parallelization done by dividing $(N-1)!$ permutations among available threads.
- Execution time decreases with increase in # of threads.
- The difference between the execution time for two consecutive cities is large because it depends on $N!$.
- Speedup increases with increase in the number of threads, while the trend remains similar across all $N's$.

MPI (Message Passing Interface)

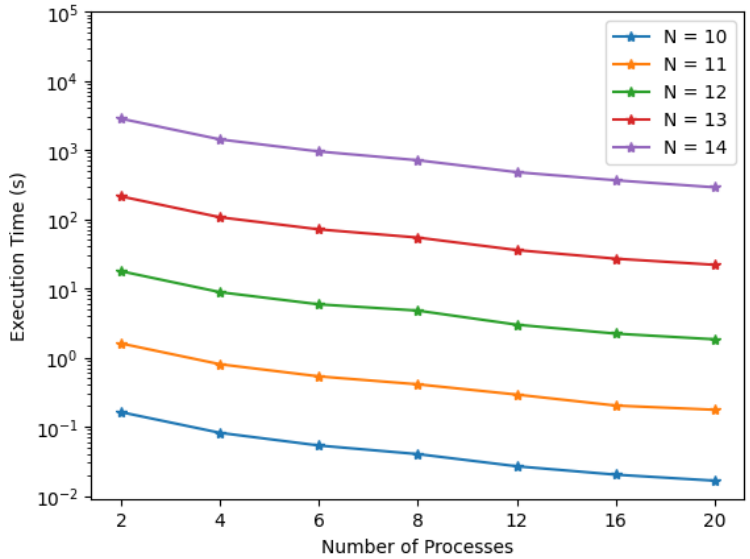


Figure: Variation of execution times with number of MPI processes for different problem sizes (N=Number of cities)

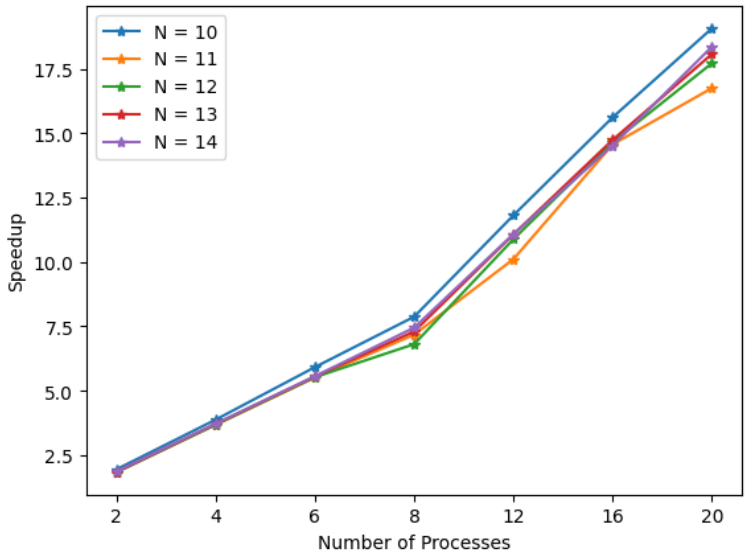


Figure: Variation of speedup achieved with number of MPI processes for different problem sizes (N=Number of cities)

- Parallelization done by dividing $(N-1)!$ permutations among available PEs.
- Execution time decreases with increase in # of PEs.
- The difference between the execution time for two consecutive cities is large because it depends on $N!$.
- Speedup increases with increase in the number of PEs, while the trend remains similar across all $N's$.

Comparative Analysis of OpenMP and MPI

Speedups & Efficiency (η)

		No. of Threads (p)				
		2	**4**	**8**	**16**	**20**
N	**10**	0.907	0.865	0.704	0.610	0.554
	11	0.834	0.808	0.734	0.618	0.590
	12	0.834	0.810	0.729	0.639	0.601
	13	0.834	0.831	0.726	0.640	0.600
	14	0.909	0.848	0.750	0.658	0.618

Table: Efficiency (η) of the OpenMP implementation for various *N* & *p* (no. of OpenMP threads)

		No. of PEs (p)				
		2	**4**	**8**	**16**	**20**
N	**10**	0.975	0.970	0.984	0.976	0.953
	11	0.922	0.920	0.898	0.910	0.838
	12	0.918	0.922	0.851	0.916	0.886
	13	0.927	0.929	0.913	0.921	0.904
	14	0.934	0.931	0.931	0.908	0.918

Table: Efficiency (η) of the MPI implementation for various *N* & *p* (no. of MPI Processes)

- Speedups achieved through MPI are greater than those achieved through OpenMP
- Using 20 PEs (in MPI) results in speedups of $\sim$ 18.
- Using 20 threads (in OpenMP) results in speedups of merely $\sim$ 12.
- Efficiency for MPI code is always greater than its OpenMP counterpart
  - Potentially due to extra over-heads of spawning & maintaining OpenMP threads compared to the inter-process communications in MPI.

Comparative Analysis of OpenMP and MPI (Contd.)

Karp-Flatt Metric $e(N, p)$ & Parallelizability

Given the speedup $\Psi(N, p)$ using p parallel elements, $e(N, p)$ is given by:

$$e(N, p) = \frac{\frac{1}{\Psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

		No. of Threads (p)				
		2	**4**	**8**	**16**	**20**
N	**10**	0.103	0.052	0.060	0.043	0.042
	11	0.199	0.079	0.052	0.041	0.037
	12	0.199	0.078	0.053	0.038	0.035
	13	0.198	0.068	0.054	0.037	0.035
	14	0.100	0.060	0.048	0.035	0.033

Table: $e(N, p)$ of the OpenMP implementation for various N & p (no. of OpenMP threads)

		No. of PEs (p)				
		2	**4**	**8**	**16**	**20**
N	**10**	0.025	0.010	0.002	0.002	0.003
	11	0.085	0.029	0.016	0.007	0.010
	12	0.089	0.028	0.025	0.006	0.007
	13	0.078	0.026	0.014	0.006	0.006
	14	0.070	0.025	0.011	0.007	0.005

Table: $e(N, p)$ of the MPI implementation for various N & p (no. of MPI Processes)

- For a given N, $e(N, p)$ usually decreases or remains constant with increase in p, indicating that the brute force TSP algorithm is rather embarrassingly parallel.
- Comparing the corresponding $e(N, p)$ values for OpenMP & MPI, highlights that OpenMP code contains higher fraction of serial component, leading to lower speedups.

CUDA (NVIDIA GPUs)

N	Time (s)		Speedup
	Serial Code	CUDA Code	
8	0.00453	0.01714	0.264
9	0.06515	0.04238	1.537
10	0.31636	0.02012	15.725
11	2.941675	0.05737	51.278
12	32.44037	0.04818	673.260
13	395.165115	0.08671	4557.187
14	5289.317205	0.82298	6427.022

Table: Execution times & speedups achieved using CUDA for various values of N

- Parallelization achieved by using fixed number of *THREADS_PER_BLOCK* & *BLOCKS*, and dividing the permutations of cities among the threads in all the blocks.
- As N increases, it is evident that the GPU provides immense benefits through parallelization, as speedups rise exponentially.

Hybrid Approach

Combining OpenMP and MPI

Parallelization Paradigm		Execution Time (s)
OpenMP (20 threads)		32.934
MPI (20 PEs)		21.866
Hybrid	2 PEs $\times$ 10 threads	124.736
	4 PEs $\times$ 5 threads	62.219
	5 PEs $\times$ 4 threads	50.492
	10 PEs $\times$ 2 threads	25.822

Table: Comparison of hybrid code execution times against OpenMP & MPI codes with same number of parallel elements for $N = 13$

- Spawn multiple OpenMP threads (*num_threads*) in each of the multiple MPI processes (*num_PEs*).
- Permutations are divided equally among the *num_PEs* $\times$ *num_threads* parallel elements.
- With same total number of parallel elements, increasing the number of MPI processes reduces the execution time.
  - Larger overheads of maintaining OpenMP threads compared to the MPI communication overhead.
- Using lesser number of PEs (with same total parallel elements) proves to be worse even compared to OpenMP
  - Higher costs of maintaining threads & their synchronization across only a small number of PEs compared to single processor.
  - Once the number of PEs becomes large enough, thread maintenance costs are overcome by MPI communication.

Wrapping Up

Individual Team Member Contributions	
Amey	MPI Code, Hybrid Code, Report, PPT
Manan	OpenMP Code, Report, PPT
Tezan	CUDA Code, Performance Analysis, Report, Presenting
Vyankatesh	Serial Code, CUDA Code, Bug-fixes, Report, Presenting

Thank You!

Questions?