Parallelization of Traveling Salesman Problem

Sanketh N S
221IT060
Department of Information Technology
National Institute Of Technology Karnataka, Surathkal.

Vrushank T S
221IT083
Department of Information Technology
National Institute Of Technology Karnataka, Surathkal.

Nithin S
221IT085
Department of Information Technology
National Institute Of Technology Karnataka, Surathkal.

Abstract*—The Traveling Salesman Problem (often called TSP) is a classic algorithmic problem in the field of computer science and operations research. It is an NP-Hard problem focused on optimization. TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips; and can be slightly modified to appear as a sub-problem in many areas, such as DNA sequencing. In this paper, a study on parallelization of the Dynamic Programming approach (under several paradigms) of the Travelling Salesman Problem is presented. Detailed timing studies for the serial and various parallel implementations of the Travelling Salesman Problem have also been illustrated.

***Keywords*—Parallel computation, TSP, OpenMP, MPI, CUDA, Time Analysis**

Introduction

The Travelling Salesman Problem (TSP) is 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 traveling between each pair of destinations. Stated formally, given a set of N cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. The problem is an NP-Hard problem, that is, no polynomial-time solution exists for this problem. The brute force solution for the problem is to consider city1 as a starting city and then generate all the permutations of the remaining N−1 cities and return the permutation with the minimum cost. The time complexity for this solution is $O(N!)$. Another solution using the Dynamic programming paradigm exists with the time complexity $O(N^2 2^N)$ which is much less than $O(N!)$ but it has exponential space complexity so impractical to implement. The goal of this study is to parallelize the *Dynamic Programming* algorithm for solving TSP using a variety of paradigms (using OpenMP, MPI & CUDA), and to critically compare and analyze the differences in their performance.

Literature survey

Recent research has explored various parallel algorithms and heuristic techniques to improve the efficiency of solving the Traveling Salesman Problem (TSP). A 2023 study implemented the Branch and Bound (BnB) algorithm combined with OpenMP parallelization and hash tables for shared memory management[7], resulting in significant speed-up, enhanced accuracy, and consistent results across simulations. Key parameters such as runtime, efficiency, core count, and speed-up were analyzed, though further research is needed to assess the approach on larger instances and other optimization problems. In 2017, another study utilized a BnB algorithm with multithreading and priority queues, achieving improved performance and scalability for TSP with up to 20 cities. However, it highlighted the need for greater scalability and evaluation in distributed environments.

A second study from 2017 presented a modified Balas' and Christofides' algorithm using sparse matrices[8], achieving high performance for TSP instances with up to 800 nodes. The study also demonstrated memory efficiency but pointed to high memory consumption for larger instances and suggested refining node selection strategies. Further advancements were made in 2014 with Ant Colony Optimization (ACO) on GPUs via OpenCL[9], showing notable speedup and optimal pheromone update strategies, though hybrid ACO implementations and GPU resource utilization remain areas for improvement. Lastly, in 2013, parallel ACO using OpenMP and CUDA demonstrated significant speed-up, particularly on GPUs[10], reaching fifty times faster execution for larger colonies. This study highlighted GPU memory limitations and the potential for multi-GPU approaches to accommodate larger problem sizes.

Together, these papers underscore the promise of parallel computing and heuristic methods in TSP solutions, revealing both the benefits of speed and efficiency and the ongoing challenges in scalability, memory management, and hybrid implementation across diverse architectures.

Serial Implementation

A. Brute Force

The brute-force approach to solving the Traveling Salesman Problem (TSP) involves exhaustively evaluating all possible routes that visit each city exactly once and return to the starting point.

For a set of n cities, we generate every possible route (or permutation) that visits each city in a unique order. Fixing one city as the starting point, there are (n-1)! possible routes to evaluate, reducing the total permutations. For each generated route, the total travel cost is calculated by summing the distances between consecutive cities, including the return distance to the starting city. During the route evaluations, the minimum route cost encountered is stored. If a cheaper route is found, it updates the minimum cost. At the end of the process, this minimum cost represents the shortest possible route for the given set of cities. Once all

Algorithm 1 Brute Force Serial TSP

Input: *city_list, cost_matrix*
optimal_cost ←∞
optimal path ← *null*
city_list ← *list of cities excluding* $city_1$
while *next_permutation(city_list)* **do**
  temp_cost ← *get_path_cost(city_list, cost_matrix)*
  {Cost of traveling cities in order of cities in city list}
  if *temp_cost* < *optimal_cost* **then**
    optimal_cost ← *temp_cost*
    optimal_path ←*city_list* {with $city_1$ appended at start & end}
  end if
end while
Output: *optimal_path, optimal_cost*

Algorithm 2 Dynamic Programming Serial TSP

Input: *cost_matrix, num_cities*
$dp[2^{num\_cities}][num\_cities]$
optimal cost ← *tsp_dp(start_city_mask, start_city, num_cities)*

tsp_dp(visited_mask, current_city, num_cities)
  if $visited\_mask = 2^{num\_cities} - 1$ **then**
    return *cost_matrix[current_city][start_city]*
  end if
  if $visited\_mask \neq 2^{num\_cities} - 1$ **then**
    return *dp[visited_mask][current_city]*
  end if
  optimal_cost ←∞
  for *each next_city in all_cities* **do**
    if *next_city is not visited in visited_mask* **then**
      new_cost ← *cost_matrix[current_city][next_city] + tsp_dp(visited_mask | (1 << next_city), next_city, num_cities)*
      optimal_cost ← *min(optimal_cost, new_cost)*
    end if
dp[visited_mask][current_city] ← *optimal_cost*
return *optimal_cost*

Output: *optimal_cost*

routes have been analyzed, the optimal route and its cost are reported as the solution to the TSP.

Algorithm 1 shows the use of the *next_permutation(...)* function that takes in a list of destinations, i.e., *city_list* & checks if there exists a next lexicographic permutation for the given permutation of destinations. If it does exist, modify the *city_list* to store this next permutation, so that it may be used to calculate the path cost. The *get_path_cost(...)* function calculates the cost of traveling through the cities based on the order mentioned in *city_list*, & finding the individual trip costs from the *cost_matrix*. At the end, we would have exhausted all possible (N−1)! permutations & arrive at the optimal path with the least path cost

This approach, while conceptually simple, has a factorial time complexity, $O((N-1)!)$, making it computationally infeasible for larger instances of TSP.

B. Dynamic Programming

The dynamic programming (DP) provides an efficient approach in TSP to handle the exponential growth of possible paths by breaking the problem down into overlapping subproblems and storing their results to avoid redundant calculations. This method, often known as Held-Karp or DP-TSP, leverages a bitmask to track visited cities and a recursive function to calculate the minimum cost of visiting remaining cities from each possible state.

In Algorithm 2, we define a recursive function *tsp_dp(visited_mask, current_city)* that computes the minimum travel cost to complete the tour from *current_city* with certain cities marked as visited. By using a 2D DP table, *dp[visited_mask][current_city]*, to store and retrieve the minimum cost for each state, we avoid recalculating costs for previously visited states, reducing the computational overhead significantly. This DP approach thus optimally solves TSP within feasible limits for moderate numbers of cities, as each state and transition is computed only once.

The time complexity of the Dynamic Programming (DP) approach for the Traveling Salesman Problem (TSP) is $O(N.2^N)$, where N is the number of cities.

C. Timing Analysis

Figure 1 demonstrates the contrasting trends in execution times and the number of nodes handled by the brute force and dynamic programming algorithms. It is evident that the brute force approach takes significantly more time to process the same number of nodes compared to the dynamic programming approach. Additionally, the brute force algorithm reaches its computational limits at a considerably lower number of nodes than the dynamic programming approach. This comparison clearly highlights that the dynamic programming approach offers superior performance in terms of both time complexity and scalability.

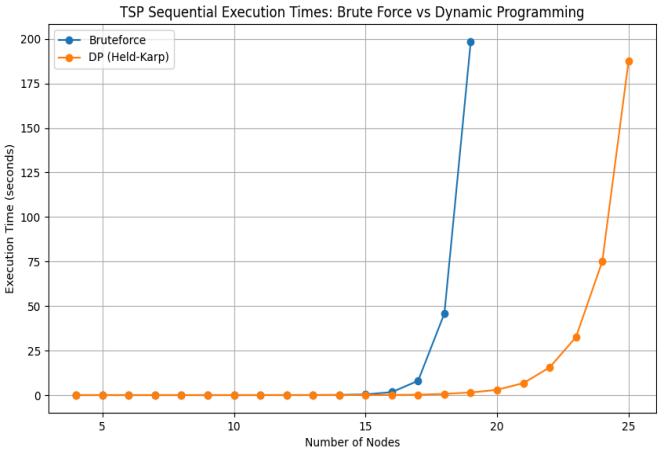

Fig. 1: Comparison of Brute Force and DP approaches based on execution times vs. number of nodes

Parallel Implementation

Motivation to parallelize serial TSP

In both brute force and dynamic programming approaches for TSP, the core computations involve evaluating numerous possible paths or subsets of cities. In the brute force method, each permutation of cities must be independently assessed, making it inherently parallelizable. Similarly, the dynamic programming approach requires evaluating subproblems defined by city subsets, which can also be processed independently before aggregation. Thus, parallelizing these computations can significantly reduce runtime, making both approaches highly suitable for parallel execution and improving scalability on larger datasets.

I. OpenMP (Shared Memory Processing)

A. Overview of OpenMP

OpenMP (Open Multi-Processing) is a widely-used API for parallel programming in C, C++, and Fortran, designed to simplify the process of running code across multiple processors or cores. Using compiler directives, OpenMP allows developers to parallelize code easily, especially in loops, without extensive modifications. Key features include flexible workload distribution, with scheduling options like Static, Dynamic, and Guided to optimize load balancing, and abstracted thread management, which lets developers focus on dividing tasks rather than handling threads directly. This makes OpenMP code both scalable and portable, running efficiently on systems from multi-core desktops to high-performance computing clusters, ultimately enhancing execution speed and resource utilization.

B. Approach for Parallelization

Logically, the algorithm requires evaluating each city as a potential next destination, a task that is both repetitive and independent for different cities. Each thread can therefore handle calculating paths to different cities without depending on the results of other threads. This independence allows us to divide the task of evaluating cities across threads, minimizing redundant work and avoiding unnecessary dependencies.

The challenge in TSP parallelization lies in the imbalance of workloads due to varying path costs and branching factors. Some cities will have more complex paths with higher costs and take longer to compute, while others might resolve quickly. To address this, the workload is distributed among threads dynamically, assigning each thread a portion of the recursive calls. Each thread is tasked with calculating the minimum path cost for a subset of cities, iteratively updating the current minimum cost.

To avoid re-evaluating the same subproblems, results are stored in a shared memoization table. The table records the minimum cost to reach each city with a given set of visited cities, allowing threads to access previously computed results instead of re-computing them. This shared access to memoized results reduces redundant calculations across threads.

In this approach, threads independently evaluate the cost of paths to each city and update a shared minimum cost variable. To avoid data conflicts when multiple threads attempt to update the minimum path cost, each thread works with its own local copy of the cost before synchronizing results. After each thread completes its calculations for a subset of cities, the results are merged to update the global minimum cost. This reduces the need for threads to wait on each other, leading to more efficient parallel execution. By experimenting with scheduling types, namely static, guided and dynamic; we optimize for minimal runtime and demonstrate the flexibility and power of OpenMP in tackling high-complexity tasks.

As each thread completes its portion of the task, it contributes its computed minimum path cost to a global result. Threads access the shared memoization table to store their intermediate results, ensuring the results are accessible to other threads for future calculations. The final result aggregates each thread's minimum path contributions, producing the optimal solution for the TSP.

C. Timing Analysis

Table I & II summarize the time taken for execution and speedup achieved using OpenMP for various counts of threads and nodes of the graph.

Nodes (Threads=8)	Execution Times (seconds)		
	Static	Dynamic	Guided
4	0.000607	0.000779	0.000726
6	0.001365	0.001312	0.001155
8	0.001890	0.002085	0.001862
10	0.003652	0.005271	0.003072
12	0.005595	0.007477	0.005586
14	0.120673	0.130202	0.127392
16	0.439528	0.474669	0.450434
18	1.963927	2.179286	2.059359
20	9.885923	11.574163	10.274660

a.

Table I. Variation of execution times for different number of nodes with different schedule types (max_threads=8)

Threads (Nodes=20)	Execution Times (seconds)		
	Static	Dynamic	Guided
2	7.476974	8.562577	7.921701
4	8.110252	9.048673	8.172363
6	8.653343	9.567813	8.857911
8	9.885923	11.574163	10.274660

b.

Table II. Variation of execution times for different number of threads with different schedule types (nodes=20)

Nodes	Time (s)		Speedup
	Serial Code	OMP Code	
4	0.000008	0.000704	0.11x
6	0.000031	0.001227	0.02x
8	0.000203	0.001945	0.10x
10	0.000959	0.003998	0.24x
12	0.004654	0.006219	0.75x
14	0.021587	0.126089	0.17x
16	0.181658	0.454877	0.40x
18	1.531165	2.067524	0.74x
20	7.900504	10.578249	0.75x

TABLE III: Execution times & speedups achieved using OpenMP for various number of nodes

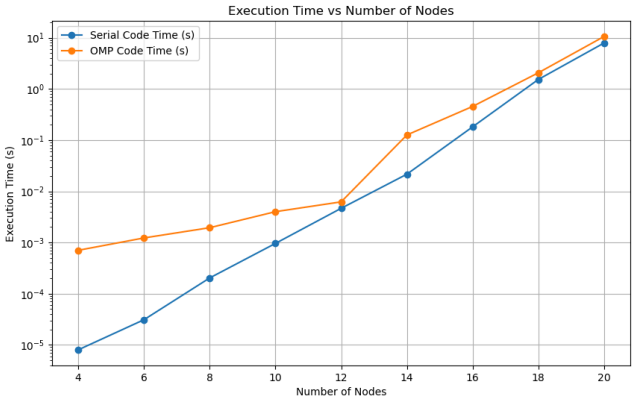

Fig. 2: Comparison of Serial and OMP Parallel implementations based on execution times vs number of nodes

D. *Performance Analysis*

- As the number of nodes rises from 4 to 20, all scheduling types show a trend of increasing execution time due to the greater workload.
- Contrary to expectations that adding more threads would reduce execution time, all three scheduling strategies—Static, Dynamic, and Guided—show an increase in execution time as the thread count grows. This pattern suggests that overhead from context switching, increased synchronization, and potential contention among threads outweighs the benefits of parallelism in this scenario, particularly as threads approach the workload size.
- Static scheduling shows the lowest execution times overall across the thread counts. It appears to handle the workload most efficiently, possibly due to minimal load balancing overhead, but still experiences an increase in time as threads increase, which could be attributed to thread contention or cache inefficiencies.
- Both dynamic and guided scheduling exhibit slightly higher execution times than static, especially as thread count increases. dynamic scheduling, in particular, shows the most pronounced increase, possibly due to the added overhead of dynamic load balancing as more threads are used. Guided scheduling, while similar to dynamic, slightly outperforms dynamic, suggesting that its decreasing chunk size strategy is slightly more efficient under this workload.

The results indicate a nuanced relationship between execution time, threads, and scheduling strategy in OpenMP parallelization. While parallel execution initially outperforms serial execution, the advantage decreases at higher node counts due to increased overhead. Static scheduling generally performs best with lower overhead, whereas dynamic and guided scheduling handle larger workloads better but suffer from thread management costs. For optimal performance, selecting the appropriate number of threads and scheduling type is crucial, balancing load balancing benefits with overhead.

II. CUDA (NVIDIA GPUs)

A. *NVIDIA's GPU architecture*

GPUs have a parallel architecture with massively parallel processors. The graphics pipeline is well suited to the rendering process because it allows the GPU to function as a stream processor. NVIDIA's GPU with the CUDA programming model provides an adequate API for non-graphics applications. The CPU treats a CUDA device as a manycore co-processor. At the hardware level, CUDA-enabled GPU is a set of SIMD stream multiprocessors (SMs) with 8 stream processors (SPs) each. Each SM contains a fast shared memory, which is shared by all its processors. A set of local 32-bit registers is available for each SP.[3] The SMs communicate through the global/device memory. The global memory can be read from or written to by the host, and are persistent across the kernel launches of the same application. Shared memory is managed explicitly by the programmers. It's much slower than shared memory. Compared to the CPU, the peak floating-point capability of the GPU is an order of magnitude higher, as well as the memory bandwidth. We have used the default GPU provided by Google Colab , **NVIDIA Tesla K80** with 12 GB of VRAM. This GPU can run continuously for up to 12 hours.

B. *CUDA programming model*

At the software level, CUDA model is a collection of threads running in parallel. The unit of work issued by the host computer to the GPU is called a kernel. CUDA program is running in a data-parallel fashion. Computation is organized as a grid of thread blocks. Each SM executes one or more thread blocks concurrently.[4] A block is a batch of SIMD-parallel threads that runs on the same SM at a given moment. For a given thread, its index determines the portion of data to be processed. Threads in a common block communicate through the shared memory. CUDA consists of a set of C language extensions and a runtime library that provides APIs to control the GPU. Thus, CUDA programming model allows the programmers to better exploit the parallel power of the GPU for general purpose computing.

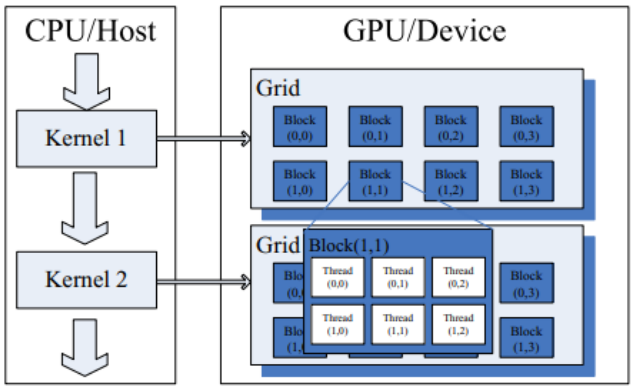

Figure 1. The threads organization architecture of CUDA.

C. *Approach for Parallelization*

Parallelizing the dynamic programming approach for the Traveling Salesman Problem (TSP) on NVIDIA GPUs with CUDA requires a different strategy than the brute force method. In dynamic programming, we typically use a memoization table to store intermediate results, which allows us to avoid redundant calculations and reduce the overall computational complexity. When parallelizing this approach, careful attention is needed to handle dependencies between subproblems to ensure correctness and efficiency.

To begin, we divide the recursive subproblems among threads by mapping parts of the memoization table (which stores solutions for specific subsets of cities) across multiple threads. This mapping allows each thread to work on calculating the optimal path cost for distinct subsets of cities, effectively enabling simultaneous calculations for many of the required subproblems.[5] However, unlike the brute force approach, where each thread can independently evaluate a path, the dynamic programming approach requires synchronization between threads at each level of subset size due to dependencies: calculations for a particular subset size depend on solutions for smaller subsets. This structure necessitates frequent synchronization within each block and between blocks to ensure that required subproblem results are available before advancing to larger subsets.

After computing the costs for smaller subsets of cities, we proceed iteratively to larger subsets, utilizing shared memory within each block to store interim results. Shared memory enables fast data access, which is essential for maintaining efficiency in a dynamic programming approach. Within each block, threads calculate the minimum path cost for subsets assigned to them, storing results in shared memory. Once threads within a block complete computations for a subset size, synchronization is performed to ensure all threads have access to the completed results before advancing to larger subsets.[6] To further reduce global memory accesses and improve performance, the block's threads work collectively to update a shared global data structure with the minimum costs for the subsets they've completed

Finally, after all subset sizes have been processed, the GPU writes the computed minimum path costs to the global memory structure, accessible by both the GPU and CPU. The CPU then reads the final computed values from the global data structure and extracts the optimal tour cost and path from the last cell, corresponding to the full set of cities. This collaborative approach between GPU and CPU allows the dynamic programming solution to take advantage of GPU parallelism while correctly managing dependencies between subproblems.

D. *Pseudocodes of CUDA specific functions*

```
__global__ void initDP(long long *dp, int size);

__global__ void tspLauncher(int *adj, long long *dp, int n, long long *result);

__device__ long long tspDP(int mask, int pos, const int *adj, long long *dp, int n);
```

E. *Timing Analysis*

Table IV summarizes the time taken for execution and speedup achieved by leveraging the power of NVIDIA GPUs

N	Time (s)		Speedup
	Serial Code	CUDA Code	
4	0.000008	1.023e-06	7.82x
6	0.000031	7.472e-06	4.15x
10	0.000959	0.000346	2.77x
15	0.073654	0.0265484	2.77x
18	1.531165	0.375428	4.08x
20	7.900504	2.504	3.16x
25	252.816128	80.12800	3.16x
26	505.632256	160.256000	3.16x
27	1011.264512	320.512000	3.16x
30	8090.116096	2564.09600	3.16x

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

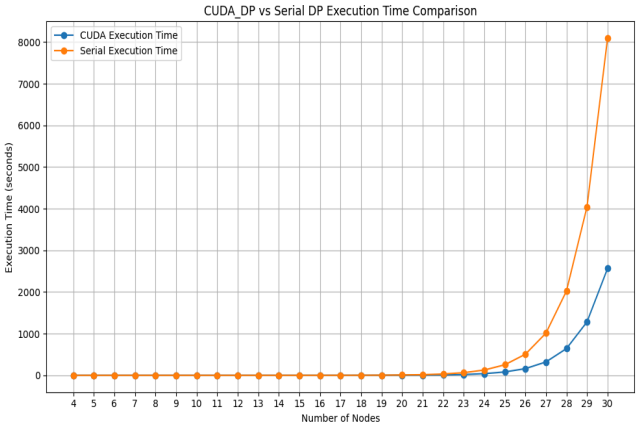

Fig. 4: Plot of CUDA execution time vs Serial Code w.r.t problem size (N) with DP algorithm.

F. *Performance Analysis*

Some observations made from the above study include:

- Execution time increases exponentially with node count, characteristic of TSP's NP-hard complexity.For node sizes 4–20, GPU computation times range from 0.00028 to 0.00485 seconds, showcasing CUDA's speed advantage for smaller node sizes.
- CUDA leverages up to 256 threads per block to handle parallel computations across states, significantly reducing time for subset calculations.
- GPU implementation is limited to 20 nodes due to exponential DP table growth, emphasizing GPU efficiency for smaller, highly parallelizable problems.
- DP table initialization on the GPU incurs overhead; data movement between CPU and GPU is minimized but could be further optimized.
- Large state tables limit performance at higher node counts; optimizations could include recursive division or memory-efficient DP methods.
- GPU outperforms CPU, completing tasks in milliseconds instead of seconds, highlighting CUDA's advantage for dynamic programming with TSP.

III. Message passing interface (MPI)

A. Overview of MPI

MPI (Message Passing Interface) is a standardized API designed for parallel computing, allowing processes to communicate with each other in a distributed computing environment. Unlike shared-memory approaches, MPI supports parallelism in a distributed memory model where each process has its own memory space. By distributing computations across multiple processors in a cluster, MPI enables efficient data sharing and task management across nodes, making it a strong candidate for high-performance parallel processing.

In this study, we used MPI on a cluster of CPUs to parallelize the TSP problem, distributing computational tasks across processes. Each process was assigned a distinct subset of calculations to minimize overlap and reduce execution time. This approach contrasts with CUDA's GPU-based architecture, focusing on multi-core CPU parallelism in a distributed environment.

B. MPI Programming Model

The MPI model divides the main program into multiple processes, with each process handling specific tasks and communicating via message-passing functions. MPI provides several critical communication methods:

- **Point-to-Point Communication**: Used to send and receive data between pairs of processes, facilitating fine-grained control over inter-process communication.
- **Collective Communication**: Involves communication across all processes. For instance, *MPI_Reduce* and *MPI_Bcast* are used to aggregate results and broadcast data, respectively.

MPI programs are typically executed on a cluster of CPUs, with each core or node acting as an independent process. To parallelize the TSP algorithm, we divided the workload of computing path costs among available processes and used collective communication functions to combine results.

C. Approach for Parallelization

The master process (rank 0) generates a graph with randomly assigned distances between cities. The graph is then broadcasted to all processes using *MPI_Bcast.* This ensures each process has access to the same distance matrix for consistent calculations.

The DP approach uses a recursive function with memoization to avoid redundant calculations. The memoization table stores previously computed subproblem results, enabling efficient reuse and reducing computational redundancy. Each process maintains its own memoization table, allowing for local computation without interference from other processes.

Each process is assigned a subset of cities to start its TSP calculation.This division of labor allows each process to calculate the minimum path cost for different subsets, improving computational efficiency and balancing the workload.

Each process calculates a *local_min_cost* based on the paths it has examined. Using *MPI_Reduce* with *MPI_MIN*, the local minimums from each process are combined to determine the global minimum TSP cost, which is stored in the master process.This reduction operation efficiently aggregates results, ensuring that only the minimum cost across all processes is retained as the global solution.

D. Timing Analysis

We measured execution times for both serial and MPI-based implementations, comparing the two for varying city counts. The results, shown in Table IX, indicate that MPI significantly reduces runtime compared to the serial implementation.

Nodes	Time (s)		Speedup
	Serial Code	MPI Code	
4	0.000008	0.000060	0.13x
6	0.000031	0.000827	1.17x
8	0.000203	0.000064	0.04x
10	0.000959	0.000346	2.77x
12	0.004654	0.001993	2.34x
14	0.021587	0.0130599	1.65x
16	0.181658	0.066281	2.74x
18	1.531165	0.476050	3.21x
20	7.900504	2.928814	2.70x

Table V: Execution times and speedups achieved using MPI for different number of nodes

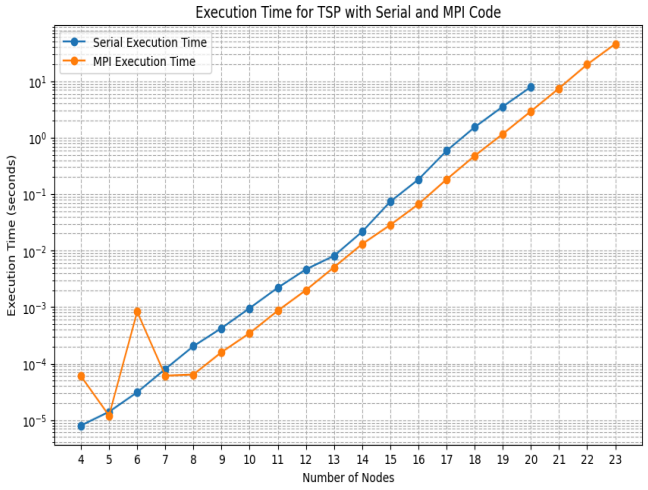

Fig. 5: Plot of MPI execution time vs Serial Code w.r.t problem size (*N*) with DP algorithm.

E. Performance Analysis

- The MPI implementation manages to reduce execution time significantly compared to the sequential approach, although its effectiveness varies with node size.
- For smaller nodes, such as 4 to 6, the MPI speedup is low, even showing a slow down in some cases. This can be due to communication overhead in the MPI implementation, where time spent managing tasks across multiple processes outweighs the benefits of parallelization for such small data sizes.
- However, as the node count grows (8–18), the speedup increases significantly, with speedup factors ranging between 2.5x and 3.2x, highlighting MPI's benefits for moderate/larger problem sizes.
- For nodes larger than 18, the efficiency of MPI decreases as the time required increases exponentially. It still performs better than the sequential solution, but the speedup factor is not high and it is not an efficient solution. The exponential growth in the DP table and subset states makes it increasingly challenging to achieve high efficiency as the problem size increases.

Conclusion and Future Work

In this project, we explore the Travelling Salesman Problem & implement several approaches to parallelize the Dynamic Programming TSP algorithm, including OpenMP, MPI & CUDA programming. We perform detailed timing analysis for all the approaches & also critically compare the performance of OpenMP & MPI codes through their efficiency..

This project can be extended to parallelize & compare other algorithms to solve the TSP such as Branch-&-Bound, Genetic Algorithms, etc. This would provide a better idea about the relative importance of selecting a good algorithm & parallelizing an algorithm.

References

1. Understanding The Travelling Salesman Problem (TSP)
2. Burkhovetskiv V & Steinberg B; Parellelizing an exact algorithm for the traveling salesman problem; Procedia Computer Science, Volume 119, Pg 97-102-2017
3. Sharing global and local variables using Unified Memory in CUDA
4. NVIDIA CUDA Programming Guide. Version 2.3. // 2009
5. Nickolls J., Buck I., Garland M., Skadron K., “Scalable parallel programming with CUDA”, ACM Queue 6 (2), 2008, 40 - 53
6. J. Nickolls, "GPU parallel computing architecture and CUDA programming model," 2007 IEEE Hot Chips 19 Symposium (HCS), Stanford, CA, USA, 2007, pp. 1-12, doi: 10.1109/HOTCHIPS.2007.7482491. keywords: {Instruction sets;Graphics processing units;Parallel processing;Computer architecture;Programming;Computational modeling},
7. Barreto, L.s & Bauer, Michael. (2010). Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations. Journal of Physics: Conference Series. 256. 012018. 10.1088/1742-6596/256/1/012018.
8. Genova, Kyle & Williamson, David. (2015). An Experimental Evaluation of the Best-of-Many Christofides' Algorithm for the Traveling Salesman Problem. 10.1007/978-3-662-48350-3_48.
9. Eric Papenhausen and Klaus Mueller. 2018. Coding Ants: Optimization of GPU code using ant colony optimization. Comput. Lang. Syst. Struct. 54, C (Dec 2018), 119–138. https://doi.org/10.1016/j.cl.2018.05.003
10. Ahmed, Ashraf & Ahmed, Walid & Elbialy, Ashraf. (2015). Performance Analysis and Tuning for Parallelization of Ant Colony Optimization by Using OpenMP. 9339. 10.1007/978-3-319-24369-6_6.