Parallel Programming Final Project: Optimizing the Traveling Salesman Problem with Parallel Algorithms

Ray Tsai
110550093
National Yang Ming Chiao Tung University
Taiwan

Charles Tsai
110550041
National Yang Ming Chiao Tung University
Taiwan

Kyle Lin
110550110
National Yang Ming Chiao Tung University
Taiwan

ABSTRACT

The Traveling Salesman Problem (TSP) is a classic NP-hard optimization problem in combinatorial mathematics and computer science, requiring the shortest route that visits a set of cities exactly once and returns to the starting point. Despite its simple formulation, TSP becomes computationally challenging as the number of cities grows, making efficient solutions crucial for its wide-ranging applications. This project focuses on parallelizing metaheuristic algorithms to solve TSP more effectively. By implementing parallel strategies, we significantly reduce computation time while maintaining solutions close to optimal compared to non-parallel approaches. We release our implementation at https://github.com/Shukkai/Parallel-Programming-Final.

KEYWORDS

Traveling Salesman Problem, Metaheuristic Algorithms, Parallel Computing

1 INTRODUCTION

The Traveling Salesman Problem (TSP), first introduced in the 1930s, has been extensively studied due to its wide range of applications, including logistics, transportation planning, circuit board drilling, and DNA sequencing. The problem is defined as follows: "**Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the starting city?**" Its name originates from the analogy of a traveling salesman tasked with finding the most efficient route to visit a set of cities and return to the starting point. TSP belongs to the class of NP-hard problems and is of significant interest to researchers due to its computational complexity. As the number of cities increases, the number of possible routes grows factorially, leading to an exponential rise in the computational effort needed to determine the optimal solution. For large problem instances, exhaustively exploring all possible solutions becomes practically infeasible.

There are various key algorithms for solving TSP problems. Two fundamental approaches are dynamic programming (DP) and greedy algorithms. Dynamic programming solves TSP by breaking it down into smaller subproblems, computing optimal solutions for all possible subsets of cities and using those results to build up the final solution. While DP guarantees finding the optimal tour, it has exponential time complexity $O(n^2 * 2^n)$, thus making it practical only for small instances up to 15-20 cities. The greedy approach, most commonly implemented as the Nearest Neighbor algorithm, works by always choosing the closest unvisited city as the next destination. While much faster with $O(n^2)$ complexity than DP, this method can produce tours significantly longer than optimal.

Due to the tradeoffs between these two algorithms, metaheuristic algorithms are introduced to provide another powerful approaches to solving TSP. A metaheuristic algorithm is a type of algorithms that is characterized by its ability to solve complex optimization problems by mimicking natural phenomena or human intelligence. Though metaheuristics don't guarantee optimal solutions, they often find high-quality solutions in reasonable time for large-scale TSP instances. Even with the implementation of metaheuristics, the algorithms often still require time to converge to high-quality solutions. Fortunately, we have discovered that some metaheuristic algorithms are suitable for parallelization to reduce computational overhead. Therefore, we apply several parallelization methods to these metaheuristic algorithms across different TSP test cases, including real-world instances.

2 RELATED WORK

2.1 Ant Colony Optimization

Ant Colony Optimization (ACO) is a nature-inspired metaheuristic that has been widely applied to solve the Traveling Salesman Problem[1]. The Ant System is inspired by the behavior of real ant colonies. When ants search for the shortest path from their nest to a food source, they leave pheromones along the route they take. These pheromones guide other ants to follow the same path. Over time, because pheromones naturally evaporate, longer paths accumulate less pheromone and become less appealing to the ants. As a result, the colony gradually converges on the shortest path.

Traditional ACO demonstrated promising results but often struggled with stagnation, where solutions converged prematurely. To address this limitation, Max-Min Ant System (MMAS)[2] introduced stricter pheromone control mechanisms. MMAS bounds the pheromone levels to prevent excessive exploitation of specific paths. These advancements have made MMAS a highly effective approach for solving TSP, outperforming traditional ACO in terms of both accuracy and robustness.

2.2 Genetic Algorithm

Genetic algorithms (GA) are a type of evolutionary algorithm inspired by the principles of natural selection. They are particularly

well-suited for solving optimization problems, such as the traveling salesperson problem[3], where the search space is large and exhaustive search is infeasible. The key concept of GA is to evolve the best outcomes from a population of individuals. This evolution occurs through five main stages: initialization, evaluation, selection, crossover, and mutation. Over successive iterations, the algorithm converges toward an optimal or near-optimal solution.

3 PRELIMINARIES

3.1 Solving TSP with ACO

Initially, each ants randomly choose a city to start with. At each construction step, ant k choose the next traverse city by the probability describe as follow

$$p_{ij}^k(t) = \frac{\left[\tau_{ij}(t)\right]^\alpha \cdot \left[\eta_{ij}\right]^\beta}{\sum_{l \in \mathcal{N}_i^k} \left[\tau_{il}(t)\right]^\alpha \cdot \left[\eta_{il}\right]^\beta} \quad \text{if } j \in \mathcal{N}_i^k \tag{1}$$

, where $\tau_{ij}(t)$ represents the pheromone strength of city i and j in t iteration, η_{ij} is the priori available heuristic value of city i and j (usually set to the inverse of the distance), and $\mathcal{N}_i^k$ is the feasible neighborhood of ant k, that is, the set of cities which ant k has not yet visited. After all ants have constructed their tours, the pheromone trails are updated:

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t) \tag{2}$$

, where $0 < \rho \leq 1$ is the pheromone trail evaporation rate. With evaporation, the algorithm can forget previous done bad decisions. And the $\Delta\tau_{ij}^k(t)$ here is defined as:

$$\Delta\tau_{ij}^k(t) = \begin{cases} 1/L^k(t) & \text{if } (i,j) \text{ is used by ant } k \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

, and $L^k(t)$ is the path distance of ant k in t iteration.

Above is how traditional ACO works. The main modification of MMAS is that it adopted the lower and upper bound for pheromone level, the lower and upper bound is set as τ_{min} and τ_{max} respectively. After pheromone update, any value that is smaller than τ_{min} will be increased to τ_{min} and any value that is greater than τ_{max} will be decreased to τ_{max}.

3.2 Solving TSP with GA

Here's how GA applied to solve the TSP: In the initialization stage, we assign random paths (permutations of vertices) to each individual. This ensures the population contains diverse path combinations, which is crucial for GA to explore a wide range of possibilities. In the evaluation stage, individuals are assessed based on the length of their paths. Shorter paths indicate better solutions and are given higher probabilities of being selected for reproduction. During the selection stage, individuals with shorter paths are more likely to pass their traits to the next generation. In the crossover stage, selected individuals randomly combine parts of their paths to create new offspring. Finally, the mutation stage introduces randomness by swapping segments of paths within individuals, preventing the algorithm from prematurely converging to suboptimal solutions.

4 PARALLEL COMPUTING SOLUTION

4.1 Multi-thread

Multi-threading is a method based on running multiple threads within a single process. We expect that dividing workloads across different threads and making them work concurrently can save a significant amount of time. Moreover, since a thread is a lightweight version of a process, it should provide better performance as well. In this task, we use the standard **<thread>** library provided by C/C++.

4.2 OpenMP

OpenMP, or Open Multi-Processing, is a widely used API for parallel programming in shared-memory systems. Similar to multi-threading, we expect that dividing workloads across different processes and executing them concurrently will improve performance. Moreover, OpenMP supports optimizations during compilation, such as loop parallelization and other hardware-level enhancements that we might otherwise overlook. Therefore, it serves as an excellent method for comparison due to its ability to leverage more parallelism at the hardware level. In this task, we use the **<omp.h>** library provided by C/C++.

4.3 MPI

MPI, or Message Passing Interface, is a standardized message-passing system designed for parallel computing in distributed memory architectures. It enables multiple processes running on different nodes to communicate and coordinate to solve large-scale computational problems. Unlike the other two parallel methods, MPI doesn't use shared memory but relies on communication protocols to share data among independent processes. This approach can be advantageous if the dataset is not too large, as shared memory may introduce synchronization issues that could offset the response time gains. In this task, we use the **<mpi.h>** library provided by C/C++.

5 EXPERIMENTAL METHODOLOGY

5.1 Parallel ACO

Ant Colony Optimization (ACO) is naturally suited for parallel processing because it mimics how real ant colonies work, with many ants operating independently. First, each ant is assigned to its own processing thread or CPU core. These ants work completely independently as they build their individual solution paths. Since they don't need to communicate during this phase, there's no need for synchronization between threads (Fig. 1). Only after all ants have finished constructing their solutions does the algorithm require coordination. At this point, the threads or processes synchronize to update the pheromone trails based on the quality of all solutions found. This synchronization step ensures that future ants can benefit from the combined knowledge of the entire colony.

5.2 Parallel GA

Our primary focus on parallelizing GA is to compute individuals and generate new generations concurrently. Since the path calculations for individuals are independent, we can assign workers to evaluate all individuals simultaneously. During the selection and crossover stages, as long as random selection methods are applied, different workers remain independent, as there is no shared state

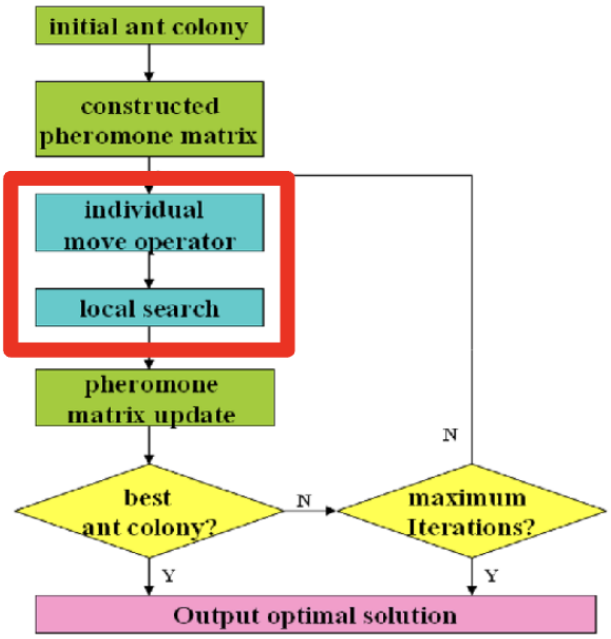

Figure 1: ACO Parallel Focus Section

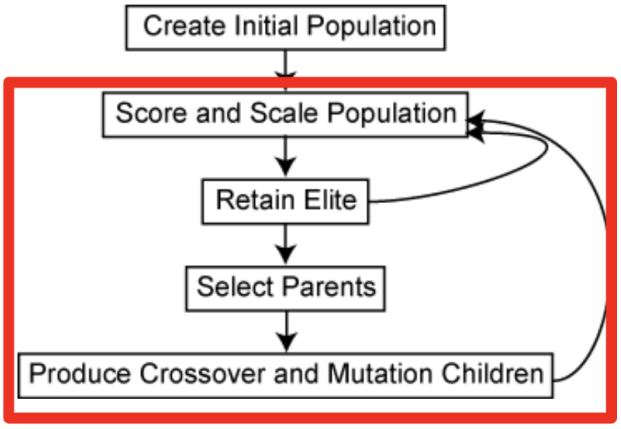

Figure 2: GA Parallel Focus Section

to update. The only bottleneck occurs during generation iteration, as workers cannot continue until a specific number of individuals have been generated. The mutation stage is also parallelizable, as we aim for random effects on path changes, making synchronization unnecessary. For the initialization stage, the computational workload is relatively small, so we do not make any modifications.

As the flow chart shown in Fig. 2, our parallel method for GA involves assigning multiple workers—ideally close to the number of individuals—to execute the Evaluate, Select, Crossover, and Mutation stages concurrently.

Algorithm 1 2-Opt Algorithm

Require: current_route: constructed TSP solution
Require: distance_matrix: provides the distances between all pairs of cities
improved ← **true**
while improved = **true do**
  improved ← **false**
  for i from 1 to $length$(current_route) − 2 **do**
    for j from $i+1$ to $length$(current_route) − 1 **do**
      delta ← (distance_matrix[current_route[i]][current_route[j]]+ distance_matrix[current_route[i+1]][current_route[j+1]])− (distance_matrix[current_route[i]][current_route[i + 1]] + distance_matrix[current_route[j]][current_route[j + 1]])
      if delta < 0 **then**
        current_route[i+1 : j+1] ← Reverse(current_route[i+1 : j + 1])
        improved ← **true**
      end if
    end for
  end for
end while

5.3 2-Opt Integration

To enhance the performance of metaheuristic algorithms, we incorporate a local search algorithm into the constructed path. Applying the local search algorithm not only further improves the solution but accelerates convergence as well. We chose the 2-Opt algorithm[4] as our local search method due to its simplicity and effectiveness.

The **Algorithm 1** examines pairs of edges in the current tour iteratively for every possible combinations of two edges. If swapping two edges reduces the overall distance, the algorithm replaces the edges and updates the route. By iteratively refining routes and eliminating inefficient crossings, the 2-opt algorithm provides a practical approach to tackling the computational challenges of the Traveling Salesman Problem.

5.4 Environment

- **OS** : Ubuntu 22.04
- **CPU** : AMD Ryzen Threadripper 3970X 32-Core
- **GPU** : NVIDIA RTX A6000
- **Compiler** : g++ 17
- **Interfaces** : MPI 4.0, OpenMP 201511

5.5 Dataset

We utilized TSP instances from TSPLIB[5], a well-known library of benchmark problems for the Traveling Salesman Problem. It is a widely recognized resource in the optimization community, providing a standard set of problem instances for researchers and practitioners to test and compare their algorithms. The library contains a diverse collection of problem instances, stored in a standardized format that specifies city coordinates and other relevant metadata such as problem type, dimension, and known optimal solutions where available.

The library includes both both real-world instances, such as "berlin52" which represents actual locations in Berlin, as well as

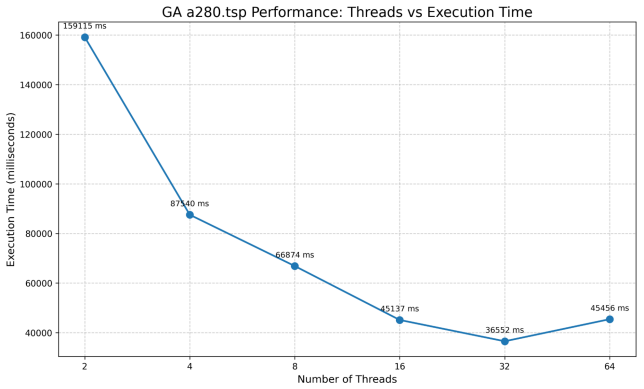

Figure 3: Genetic Algorithm with different thread number

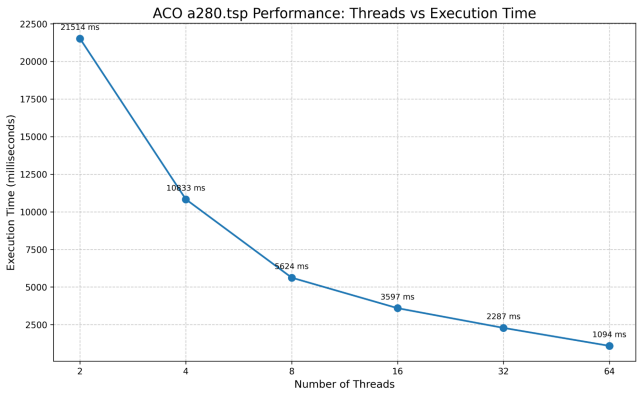

Figure 4: Ant Colonial Algorithm with different thread number

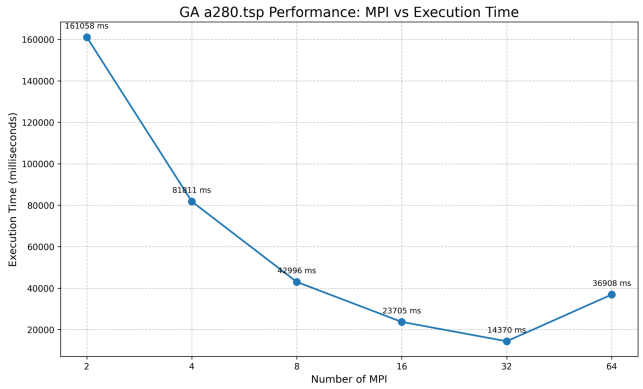

Figure 5: Genetic Algorithm with different Process number

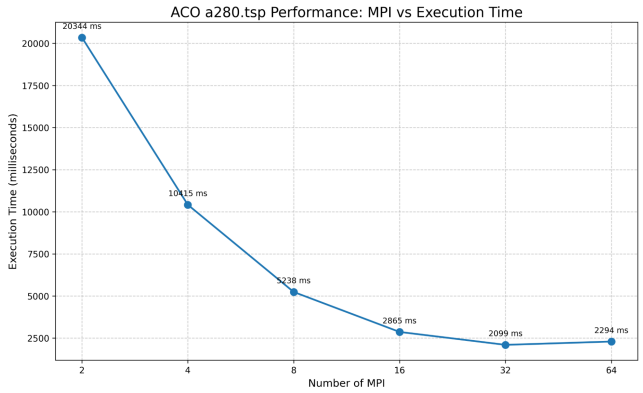

Figure 6: Ant Colonial Algorithm with different Process number

synthetic instances like "a280". These problems vary significantly in size and complexity, enabling researchers to thoroughly evaluate their algorithms' effectiveness and scalability under different conditions.

6 EXPERIMENTAL RESULTS

6.1 Comparison with Different Numbers of Threads

In our experiments, we observed that as the number of threads doubled, the execution time for the ACO algorithm consistently decreased by half. This scaling behavior highlights the efficiency of ACO in utilizing parallel resources. In contrast, the GA did not show the same level of improvement. While performance increased with more threads, the speedup was not proportional to the number of threads. Notably, when the thread count reached 64, performance began to degrade. This decline is likely attributed to the overhead associated with duplicating resources. GA requires the replication of population data across each thread, introducing significant overhead as the thread count grows. Conversely, ACO's parallelization mainly involves distributing the ants across threads, resulting in lower resource duplication overhead.

6.2 Comparison with Different Numbers of MPI

In the case of MPI, when the number of processes is relatively small, the performance scales proportionally with the increase in processes. However, as the process count continues to grow, the performance gains become less pronounced. This diminishing return is likely caused by the overhead associated with resource duplication and context switching, which increases as more processes are introduced.

6.3 Comparison with Different Parallel Methods

The results show that MPI generally outperforms multithreading and OpenMP, especially when the number of processes or threads is moderate. MPI benefits from process-level parallelism, avoiding contention for shared resources by using separate memory spaces. This leads to better scalability compared to threads, which rely on shared memory and are prone to synchronization overhead.

However, as the number of processes grows, MPI's performance gains diminish due to communication overhead and context switching. In contrast, multithreading performs efficiently at smaller scales, with lower overhead and faster context switching, making it a suitable option for lighter workloads.

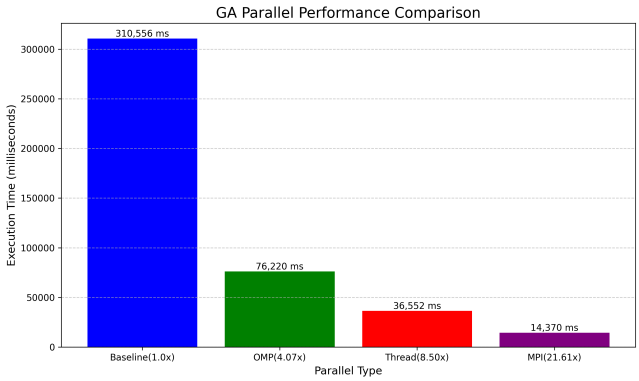

Figure 7: Genitic Algorithm all methods comparison

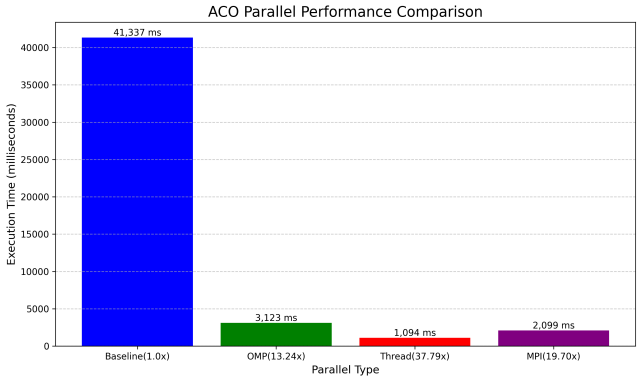

Figure 8: Ant Colonial Algorithm all methods comparison

Table 1: Comparisons of GPU and CPU parallelization methods using ACO on the a280.tsp problem

Method	Run Time(ms)	Speedup	Total Distance
Serial	41337	1.0x	**2637.63**
OpenMP	3123	13.24x	2640.68
Multi-thread (64)	**1094**	**37.79x**	2651.24
MPI (32)	2099	19.70x	2776.77
CUDA	21942	1.88x	2776.60

6.4 CUDA Implementations

We have also implemented a CUDA version to parallelize the TSP problem. Our CUDA implementation focuses solely on the ACO algorithm, because we believe that ACO relies more heavily on mathematical calculations, which make it well-suited for CUDA. GA, on the other hand, require more complex operations like crossover and mutation. The results are shown in Table 1, indicating that CUDA can indeed improve runtime performance, but it is not as ideal as other parallelization methods. As a result, we believe that ACO has to synchronize the current best solution to update the pheromone in each iteration, which becomes the bottleneck in CUDA implementations.

Table 2: Ablation of the 2-Opt integration. We compare serial and multi-thread ACO on the a280.tsp problem

parallel methods	2-Opt	Optimal Distance	Total Distance
Serial	✗	2579	3954.53
	✓		**2637.63**
Multi-thread(64)	✗		3481.77
	✓		**2651.24**

6.5 Ablation Study

In Sec.5.3, we integrate 2-Opt algorithm as local search to improve the total performance.To validate the effectiveness of the 2-Opt integration, we conducted experiments comparing the solution performance with and without the use of 2-Opt in same iterations. As indicated in Table 2, the addition of 2-Opt significantly reduces the total distance. These results demonstrate that integrating 2-Opt improves the solution quality and accelerate the metaheuristics convergence speed.

7 CONCLUSIONS

In this project, we parallelized metaheuristic algorithms to solve the Traveling Salesman Problem (TSP) using multi-threading, OpenMP, MPI, and CUDA. The results show significant performance improvements over serial implementations, with ACO benefiting the most from parallelization.

MPI consistently outperforms multithreading and OpenMP at moderate process counts, while multithreading excels at smaller scales due to lower overhead. OpenMP, although easy to implement, showed the slowest performance due to synchronization overhead in shared memory. CUDA provided some speedup but faced bottlenecks during pheromone updates.

Integrating the 2-Opt algorithm further improved solution quality and accelerated convergence. Overall, MPI is the most effective for large-scale workloads, while multithreading and OpenMP remain viable for smaller tasks. Future work could explore hybrid CPU-GPU approaches to optimize performance further.

REFERENCES

[1] T. Stützle, M. Dorigo, *et al.*, "Aco algorithms for the traveling salesman problem," *Evolutionary algorithms in engineering and computer science*, vol. 4, pp. 163–183, 1999.

[2] T. Stützle and H. H. Hoos, "Max–min ant system," *Future generation computer systems*, vol. 16, no. 8, pp. 889–914, 2000.

[3] S. Chatterjee, C. Carrera, and L. A. Lynch, "Genetic algorithms and traveling salesman problems," *European Journal of Operational Research*, vol. 93, no. 3, pp. 490–510, 1996, ISSN: 0377-2217. DOI: https://doi.org/10.1016/0377-2217(95)00077-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377221795000771.

[4] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958, ISSN: 0030364X, 15265463. [Online]. Available: http://www.jstor.org/stable/167074 (visited on 12/21/2024).

[5] G. Reinelt, "Tsplib—a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.