# Traveling Salesman Heuristics: CPU Parallelization

**Ethan Baldwin, Bryden Mollenauer, Dominic Ortolano**

**May 2, 2024**

# 1 Abstract

The aim of this report is to analyze the Ant Colony Optimization heuristic, which is used to generate an approximate solution to the Traveling Salesman Problem. Our goal was to improve on the work of a previous research group, who analyzed heuristics for the Traveling Salesman Problem in 2017. That research group (referred to afterwards as "the 2017 group") examined how an Ant Colony Optimization approach to the TSP performs when run in parallel on multiple CPU cores. We attempted to improve on their work, first by attempting an implementation with GPU parallelization, and then by improving their CPU-parallelized implementation. We found that our CPU-based TSP-ACO implementation displayed better performance in scaling when compared to the 2017 implementation. Although we could not complete a functional GPU implementation of our solution, we believe this to be possible, and we expect it to yield an even greater performance increase.

# 2 Background

## 2.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a problem about finding the shortest Hamiltonian cycle (i.e. a cycle that visits all vertices exactly once) in a complete graph. Exact-solver brute-force algorithms are in O(n!) time and heavily computation intensive. The TSP is a famous NP-hard problem, meaning it is an optimization problem for which no known algorithm can find a globally optimal solution in polynomial time, but if a solution is provided, its optimality can be verified in polynomial time.

The TSP has a variety of applications, such as in planning, logistics, and transportation, that make improving its performance desirable. Because its exact solution has a factorial runtime, however, most of this effort is put into developing heuristic solutions that yield a suboptimal answer at a fraction of the runtime.

## 2.2 Ant Colony Optimization

Ant Colony Optimization (ACO) is a technique for finding the best path through a graph with the use of "ants" which traverse the graph leaving a pheromone trail for future ants to follow, eventually leading to the most optimal path. It is modeled off of pheromone-based communication, one of the most effective ways of communication widely observed in nature.

The time it takes to find the optimal solution for the TSP skyrockets as the problem size increases, rendering the algorithm unfeasible for large problem sets. ACO can be used as a heuristic to find a nearly optimal solution within a certain threshold of the optimal solution, making the runtime more manageable. ACO-TSP implementations work by having ants walk through the graph. As ant travels throughout the graph, it leaves behind pheromones. Once an ant completes a full Hamiltonian cycle, creating a solution to the TSP, it will stop its travels.

If the solution is the best so far, it will be remembered. Better solutions will leave behind stronger trails of pheromones on each of the edges they traversed, encouraging future ants to reinforce those edges. In the 2017 group's implementation, this process will continue to improve until the solution is within 5% of the optimum, which is known for the problem sets used, and then terminates the algorithm and returns the solution and runtime. Our modified implementation follows the same process, with the same threshold.

# 3 Methods

## 3.1 GPU Parallelization with CUDA

Our initial approach was to use CUDA to parallelize "construct ant solutions" (see Figure 1), the same part of the algorithm that the 2017 group parallelized with OpenMP. According to them, it made up 75% of the total algorithm's runtime, making it a good target for parallelization. The 2017 group also theorized that a CUDA implementation of ACO would result in powerful performance improvements. However, the ACO-TSP codebase is ill-suited to parallelization with CUDA. A high quantity of global variables, decentralized memory allocations, and coupling between files all made it difficult to establish the firm partition between host and device memory that makes writing CUDA code feasible. As a result, we concluded that a CUDA implementation was beyond the scope of this project.

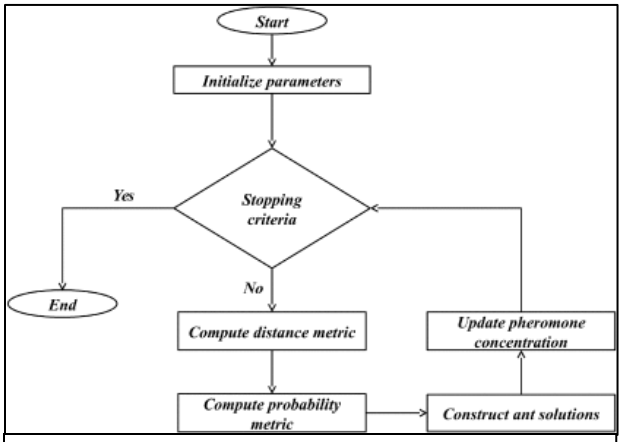



Figure 1: ACO Flowchart. Image from https://www.sciencedirect.com/topics/engineering/ant-colony-optimization

## 3.2 CPU Parallelization with OpenMP

We instead used OpenMP to parallelize the "update pheromone concentration" section of the algorithm, as it was serial in the starting codebase. While not as dominant in the runtime as "construct ant solutions", it consisted of several inner methods that contained sections of code which could be improved by parallelization.

# 4 Experiments

The graphs below were created from data obtained by a consistent method. For each TSP problem set, we ran five trials. In each trial, we first measured the performance of ACO-TSP running serially, ignoring all OpenMP directives. Then, we measured the performance of the 2017 group's parallel ACO-TSP implementation with a set number of OpenMP threads: 2, 4, 8, 16, 32, and finally 64. Finally, we measured the performance of our group's ACO-TSP implementation on the same number of threads. In both the 2017 and 2024

implementations, the program is designed to stop after reaching a solution to the problem set with a trail length within 5% of the optimal solution's trail length. The optimal solution's trail length is passed to the program ahead of time.

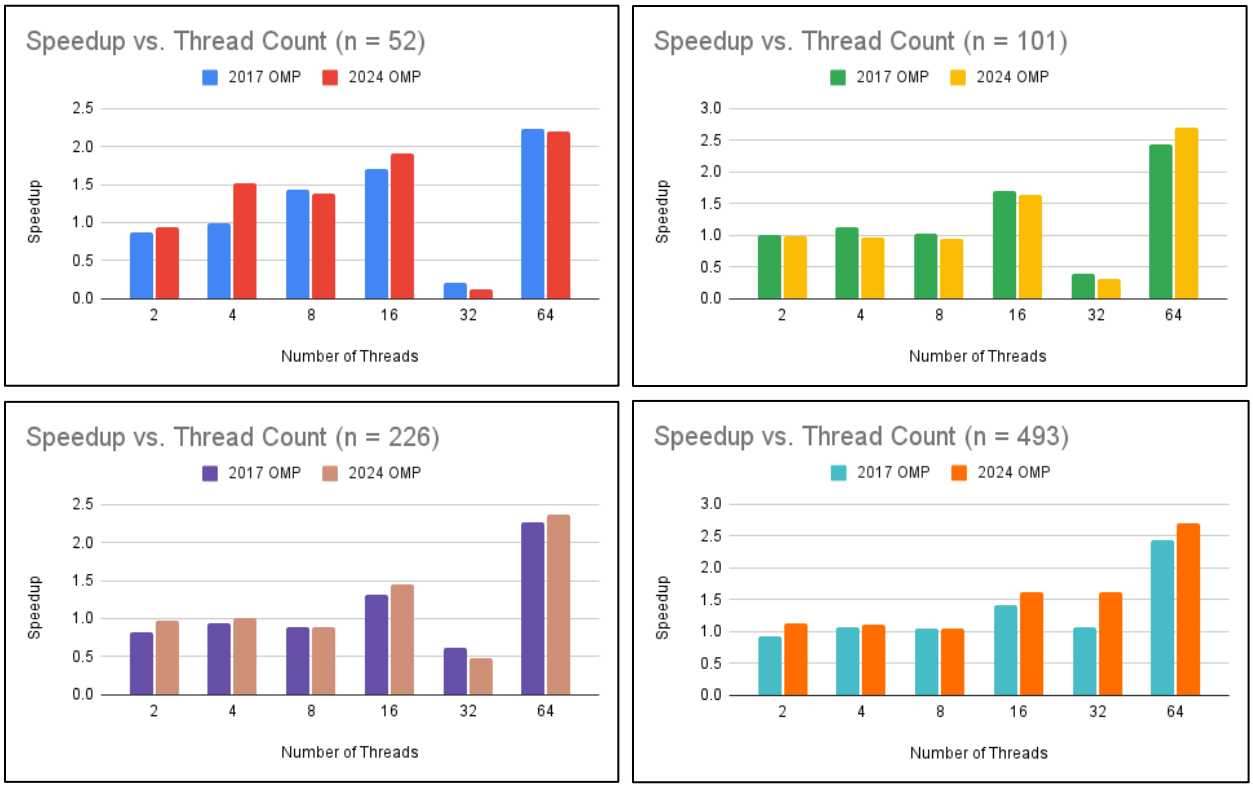
To calculate the speedup for each test, we divided the problem set's lowest serial runtime between the five trials by the lowest parallel runtime between the five trials (i.e. $S = T_{serial} / T_{parallel}$). The lowest values were chosen from each trial, rather than the average, to minimize the amount of possible noise or interference on the JMU CS 470 cluster (which all of these trials were run on).

The TSP problem sets used are located in the 2017 group's repository on GitHub, in the "tsplib" submodule. The number of nodes in the problem set is identical to the number at the end of their names. By increasing node count, the problem sets are: berlin52, eil101, pr226, and d493.

# 5 Results

## 5.1 Data

The full set of data collected can be viewed here:
https://docs.google.com/spreadsheets/d/1XtmbMeHZEH81v3XtGR1gtJDxx6l-XQha-O8L_nPGUmY/edit#gid=0




Figures 2-5: ACO-TSP Comparison Results

## 5.2 Commentary

The most noticeable consistent trait throughout Figures 2-5 is the sharp drop off in speedup at 32 threads. Across all four TSP problem sets, the tests at 32 threads show a deviation from the expected trend; while tests at lower thread counts (2, 4, 8) have little to no speedup compared to the serial version as a result of parallelization overhead, tests at higher thread counts (16, 64) are universally faster than the serial version by a significant degree. Only at 32 threads is the parallel version significantly slower than the serial version. This anomaly was seen consistently at tests at 32 threads. One possible reason lies in the JMU CS 470 cluster's hardware structure, in which 32 threads is the first threshold of hyperthreading, potentially indicating that ACO-TSP gains no advantage from hyperthreading alone. Regardless of the cause, the performance decrease at 32 threads appears to be communication-bound rather than computation-bound, as the severity of the performance drop decreases as the node count of the problem set increases. It's possible that at an even higher node count (>1000), the performance drop will be unnoticeable compared to the increase that parallelization brings.

Aside from tests at 32 threads, the speedup tends to increase as the number of threads does, while the problem size stays the same. Both the 2017 group's implementation and our implementation exhibit this, particularly with 16 or more threads, and thus show evidence of strong scaling. This is consistent with the 2017 group's findings.

Our implementation shows minimal or no improvement over the 2017 group's implementation at both lower thread counts and lower node counts, likely due to greater communication overhead for less performance increase. However, with the d493 problem set at 64 threads (Figure 5), our implementation demonstrated a 10.9% increase in maximum speedup. Given the trend, it is likely that our implementation will demonstrate even greater improvements with higher thread counts and/or higher problem sets, and thus scales better than the 2017 group's implementation.

# 6 Conclusion

Our group's contribution is a more comprehensively parallelized implementation of ACO-TSP compared to previous implementations, along with perspective on a possible CUDA implementation on the GPU. Because our ACO-TSP implementation parallelizes a greater portion of the algorithm, it exhibits better scaling than the 2017 group's, showing greater increases in performance as both problem size and thread count scales. While we attempted a CUDA implementation, and we do believe it is possible, the 2017 group's codebase is ill-suited for GPU parallelization. Future attempts at a CUDA implementation would likely have more success at building a new implementation of ACO-TSP designed specifically for CUDA, especially with concern to memory management.

# 7 Future Work

For future research and development, there exists considerable potential to boost the efficiency of ACO-TSP through the utilization of GPU-based technologies. Despite our team's challenges in achieving a successful CUDA implementation of the ACO algorithm, we are confident in the viability of this approach for enhancing computational performance.

Two promising strategies could be explored further. First, a "coarse-grained" implementation where each CUDA thread acts as an autonomous agent, representing an individual ant. This method leverages the specific aspects of GPU parallelization to simulate multiple ants simultaneously, potentially accelerating the convergence towards optimal solutions. Second, a "fine-grained" approach could involve assigning each CUDA block to mimic the behavior of a single ant, with each thread within the block dedicated to executing different components of the probability calculations required for path selection and pheromone updates. This method could allow for more intricate interactions within the model, accelerating finer aspects of the ACO process individually.

However, a significant hurdle in our current framework is the extensive use of global variables, which introduces complexities in memory management when interfacing with CUDA kernels. These issues often lead to inefficiencies and can hamper the scalability of the algorithm on GPU architectures. To address this, we suggest a substantial overhaul of the existing code base. Starting afresh or at least undertaking a partial rewrite would minimize dependencies on global variables and help in architecting a more robust and scalable GPU-ready solution.

Adopting these recommendations would not only pave the way for a successful CUDA implementation but also significantly advance the state of research in parallel computing solutions for combinatorial optimization problems like the TSP. This could ultimately lead to more efficient and faster solving techniques, opening new avenues for practical applications and further academic exploration.

# 8 References


Compton, A., Rogers, I. (2006, December 4). *Ant Colony Optimization: The Traveling Salesman Problem*. Rose-Hulman Institute of Technology. https://www.rose-hulman.edu/class/cs/csse453/archive/2011-12/presentations/TSP.pdf

Folks, G., Mast, Q., Nasrawt, Z. (2017, April 27). *Traveling Salesman: Heuristic Scaling Analysis*. https://github.com/mastqe/470-gputsp/blob/master/Traveling%20Salesman%3A%20Heuristic%20Scaling%20Analysis.pdf

Menezes, B. A. M., Kuchen, H., Amorim Neto, H. A., & de Lima Neto, F. B. (n.d.). Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem. *2019 IEEE Congress on Evolutionary Computation (CEC)*, pp. 3094-3101. IEEE Xplore. https://doi.org/10.1109/CEC.2019.8790073

Tripathy, A. (2020, September 5). *Travelling Salesman Problem (Basics + Brute force approach)* (U. Kiao, Ed.) [Review of *Travelling Salesman Problem (Basics + Brute force approach)*]. OpenGenus IQ: Computing Expertise & Legacy; OpenGenus IQ. https://iq.opengenus.org/travelling-salesman-problem-brute-force/