Implementing the Travel Salesman Problem using CUDA and Openmp in GPU

D Mukesh

Pavan Singh M

Sharan G

Abstract

The Ant Colony Optimization (ACO) algorithm, inspired by the foraging behavior of ants, stands as a potent metaheuristic for addressing optimization problems. This report delves into the comprehensive implementation and subsequent analysis of ACO in solving the classic Traveling Salesman Problem (TSP). We meticulously crafted serial, parallel (CUDA), and parallel (OpenMP) versions of the ACO algorithm. Through thorough evaluation encompassing execution time and solution quality metrics, we assessed the effectiveness of each implementation. Our findings underscore the potency of parallelization techniques, particularly CUDA and OpenMP, in significantly reducing execution times while maintaining solution integrity. This study provides valuable insights into the practical application of parallel computing paradigms to optimize ACO's performance, thereby enabling enhanced efficiency and scalability in addressing complex optimization challenges.

Introduction

The travelling salesman problem (TSP) is a nondeterministic polynomial hard problem in combinatorial optimization and is the most studied optimization problem favorable among researchers. It is so important that almost every new approach for solving Optimization problems is first tested on TSP. Though it looks simple, it is one of the classical optimization problems, which cannot be solved by conventional mathematical approach. In this project, I have used two metaheuristic approaches- ACO and GA based on certain criteria's that make them better as described in later sections, for solving TSP.

The Travelling Salesman Problem describes a salesman who must travel between N cities. The order does not matter but he should finish where he had begun. Considering each city is connected to other close by cities by a link (e.g.: road) which has one or more weights (or the cost) attached, then the salesman would want to keep both the travel costs, as well as the distance he travels as low as possible. Since he does not want to spend much time on travelling, therefore we need to find the sequence of cities to minimize the traveled distance.

In short, TSP is a graph theory problem, which searches for the shortest Hamiltonian cycle through a graph, or the shortest path (optimal route) that visits each of n cities exactly once. As the number of nodes increases, the number of tours also increases. With N cities, there are N! possible paths. The obvious "brute force" approach is to compare all N! paths to find the shortest. This is not feasible for large values of N. Rather than find an exact solution to TSP, what most do is find an approximate solution. An approximation algorithm is one, which finds a "good enough" solution to a problem, which may or may not be the best one possible.

As TSP has a search space, which grows at least exponentially to the number of nodes/cities, using GPU as general purpose computing devices, exhibits a lot of parallelization, which is a good usage for this problem to speed up optimization, which is why I used the GPU-friendly algorithms ACO and GA to solve TSP.

Background Work

TSP was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. Hassler Whitney and Merrill Flood at Princeton later investigated it. Later, in 1940"s the TSP was studied by statisticians in relation to agricultural application. Solution methods of TSP began to appear in papers in the mid-1950s; these papers used a variety of minor variations of the term "Traveling Salesman Problem."

In the following decades, many researchers from mathematics, computer science, chemistry, physics, and other sciences studied the problem. Although TSP is easy to understand, it is very difficult to solve. Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete (non-deterministic polynomial time hard), which implies the NP-hardness of TSP. NP-hard, is a class of problems that are informally the hardest problems in NP which means no polynomial-time algorithm is known for solving TSP.

This supplied a scientific explanation for the apparent computational difficulty of finding optimal tours. Brief history of TSP milestones- Dantzig, Fulkerson, and Johnson published a description of a method for solving the TSP and illustrated the power of this method by solving an instance with 49 cities in 1954. It turned out that an optimal tour through the 42 cities used the edge joining Washington, D.C. to Boston; since the shortest route between these two cities passes through the seven removed cities, and this solution of the 42-city problem yields a solution of the 49-city problem. Held and Karp solved a TSP using 64 cities in 1971.

Later in 1975, Camerini, Fratta, and Maffioli solved a TSP with 67 cities. In 1977, Grotschel solved a TSP using 120 cities in and around Germany and so on and so forth in later years. In 2004, TSP of visiting all 24,978 cities in Sweden was solved; a tour of length of approximately 72,500 kilometers was found and it was proven that no shorter tour exists. TSP is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP is represented in numerous transportation and logistics applications such as:

• Arranging routes for school buses to pick up children in a school district,

• Delivering meals to home-bound people,

• Scheduling stacker cranes in a warehouse,

• Planning truck routes to pick up parcel post and many others. • Planning, logistics, and the manufacture of microchips.

• A classic example of the TSP is the scheduling of a machine to drill holes in a circuit board.

Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents for example, customers, soldering points, or DNA fragments, and the concept distance, represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder. In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether any tour is shorter than L) belongs to the class of NP-complete problems. Thus, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities.

At its core, ACO mimics the foraging behavior of ants, utilizing pheromone trails to guide solution exploration and iterative updates to refine paths towards optimal solutions.

Serial implementations of ACO have undergone extensive scrutiny and application across diverse domains due to their simplicity and effectiveness. These implementations typically entail iterative construction of ant solutions, pheromone trail updates based on solution quality, and repeated iterations until convergence. While serial ACO algorithms have demonstrated considerable success in finding near-optimal solutions for TSP and similar combinatorial optimization problems, their scalability and efficiency on modern computing architectures are often constrained.

The emergence of parallel computing has kindled interest in parallelizing ACO to harness the computational power of contemporary parallel architectures, such as multi-core CPUs and GPUs. Parallelization of ACO presents an opportunity to expedite the optimization process by leveraging parallelism at different levels. Two primary approaches to parallelizing ACO have surfaced: parallelization using CUDA for GPU acceleration and parallelization using OpenMP for multi-core CPU architectures.

Parallel ACO implementations leveraging CUDA exploit the immense parallelism offered by GPUs to accelerate solution space exploration and pheromone update operations. By distributing workload across thousands of GPU cores, CUDA-based implementations can achieve significant speedups compared to their serial counterparts. Likewise, parallel ACO implementations employing OpenMP target multi-core CPU architectures, where parallel threads collaborate to explore solution space and update pheromone trails concurrently.

In summary, the background underscores the burgeoning interest in parallelizing ACO to augment its scalability and efficiency on contemporary computing platforms. By capitalizing on parallel computing methodologies, researchers aspire to unlock new avenues for efficiently solving large-scale optimization problems. Subsequent sections of this report delve into the implementation and analysis of parallel ACO algorithms using CUDA and OpenMP, elucidating their performance characteristics and scalability in addressing TSP and related optimization challenges.

Objective of the Work

The primary aim of this endeavor is to delve into the Ant Colony Optimization (ACO) algorithm, exploring its various implementations including **serial, parallel (CUDA), and parallel (OpenMP)** versions. The specific focus is on utilizing ACO to solve the Traveling Salesman Problem (TSP), a well-known optimization conundrum revolving around determining the shortest route that traverses each city exactly once and returns to the starting point.

This project aims to comprehensively evaluate the impact of parallelization on ACO's performance and scalability. Through rigorous analysis, we aim to assess key metrics such as execution time and solution quality across different parallel computing paradigms. The goal is to discern the efficacy of serial versus parallel implementations in converging to near-optimal solutions within reasonable timeframes. Additionally, we strive to gauge the scalability of parallel ACO implementations with respect to problem size and computational resources.

The overarching objective is to harness parallel computing techniques like CUDA and OpenMP to expedite the optimization process without compromising solution quality. By parallelizing ACO, our aspiration is to achieve swifter convergence rates and improved scalability, thereby empowering the algorithm to effectively tackle larger TSP instances. Moreover, we seek to identify and elucidate any inherent trade-offs between execution time and solution quality introduced by parallelization, offering valuable insights into the practical implications of adopting parallel computing strategies for optimization problems.
Ultimately, this project endeavors to contribute to the existing body of knowledge surrounding ACO and parallel computing. Through meticulous analysis and comparison of various implementation approaches, we aim to provide a nuanced understanding of ACO's

performance under different parallelization strategies. By shedding light on the benefits and challenges associated with parallelizing ACO for TSP, we hope to inform future research directions and practical applications in the realms of optimization algorithms and parallel computing.

Profiling Information about the computation

Profiling information provides essential insights into the computational behavior of each implementation, offering valuable details on aspects like CPU and GPU utilization, memory consumption, and communication overhead. In the case of the serial implementation, the focus is primarily on CPU utilization, revealing how efficiently computational resources are utilized during execution. Additionally, memory usage metrics shed light on the algorithm's memory footprint, indicating its efficacy in managing memory resources.

Conversely, parallel implementations introduce complexities due to concurrent execution across multiple processing units. Profiling the CUDA implementation allows us to understand GPU utilization patterns, demonstrating how parallelism is leveraged to expedite computation. Memory usage on the GPU becomes significant, reflecting the efficiency of memory management strategies implemented within the parallel algorithm. Moreover, communication overhead between the CPU and GPU emerges as a critical factor, impacting overall performance and scalability.

Similarly, profiling the OpenMP implementation provides insights into thread utilization and resource contention on multicore CPU architectures. Analysis of CPU utilization across threads helps identify potential bottlenecks and optimize task scheduling strategies for enhanced parallel efficiency. Attention is also directed towards memory usage on shared memory systems, emphasizing the importance of minimizing data dependencies and optimizing memory access patterns. Overall, profiling information serves as a valuable tool for comprehending the computational characteristics of each implementation, guiding optimization endeavors to improve performance and scalability.

Algorithm of parallel version

Algorithm of Parallel Ant Colony Optimization (ACO) for the Traveling Salesman Problem (TSP):

1. **Initialization**: Initialize the parameters including the number of cities, number of ants, number of iterations, pheromone evaporation rate, and the constant (q).

2. **Data Preparation**: Read the city coordinates from a file and calculate the distances between each pair of cities to form a distance matrix.

3. **Pheromone Initialization**: Create a pheromone matrix with initial values set to 1.0 for all edges between cities.

4. **Main Loop**:
   - Iteration Loop: Repeat the following steps for the specified number of iterations.

 1. **Ant Movement**:
   - For each ant, generate a path by probabilistically selecting the next city to visit based on the amount of pheromone on the edge and the distance to the city.
   - Use the formula $P_{ij} = \frac{{\tau_{ij}}^\alpha}{{d_{ij}}^\beta}$, where P_{ij} is the probability of moving from city i to city j, τ_{ij} is the pheromone level on the edge between city i and city j, d_{ij} is the distance between city i and city j, and α and β are parameters controlling the influence of pheromone and distance, respectively.

 2. **Distance Calculation**: Calculate the total distance of each ant's path.

 3. **Update Pheromone**:
   - Identify the best path based on the shortest total distance.
   - Update the pheromone levels on the edges of the best path using the formula $\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \frac{q}{{L_k}}$, where ρ is the evaporation rate, L_k is the length of the best path, and q is a constant.

5.**Result**: Output the best solution found after the specified number of iterations, which represents the shortest tour length achieved by the ant colony.

Description:

The parallel version of the ACO algorithm for the TSP utilizes CUDA or OpenMP to exploit parallelism and accelerate the computation of ant paths and pheromone updates. In CUDA, each thread corresponds to an ant, and parallelism is achieved by executing multiple threads concurrently on the GPU. In OpenMP, parallelism is implemented using multiple threads running in parallel on multicore CPUs.

The algorithm leverages probabilistic decision-making by ants to iteratively construct candidate solutions. Each ant probabilistically selects the next city to visit based on a combination of pheromone intensity and distance to the city. This approach allows the algorithm to explore the search space efficiently while exploiting information gathered from previous iterations.

The pheromone update process involves reinforcing the edges of the best path found in each iteration by depositing pheromone proportional to the quality of the solution. This pheromone reinforcement encourages the exploration of promising regions in the search space, guiding the search towards optimal or near-optimal solutions.

Overall, the parallel ACO algorithm offers significant advantages in terms of computational efficiency and scalability, enabling the solution of large-scale TSP instances within reasonable timeframes. By harnessing parallel computing capabilities, the algorithm accelerates the search for high-quality solutions and facilitates the exploration of complex optimization landscapes. Also can be done by using GA:

Implementation

<u>Pseudo Code of GA:</u>

```
 begin procedure GA generate populations
 and fitness function evaluate population

while(termination criteria not met)
{
 while (best solution not met)
{
 crossover mutation evaluate
}
}
post-process results and output end GA procedure.
```

Results

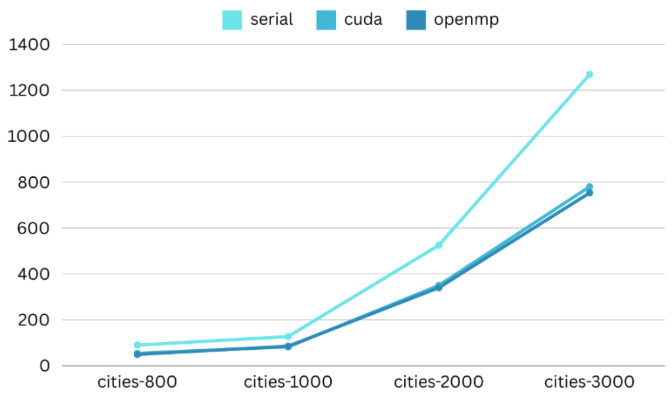
Three implementations of the Ant Colony Optimization algorithm are provided: serial (Python), CUDA parallel (C++), and OpenMP parallel (C++). These versions aim to find the shortest path through cities using ant-inspired behavior.

-**Serial** (Python): Implements the algorithm in Python, managing path generation, distance calculation, and pheromone updates sequentially.

-**CUDA Parallel** (C++): Utilizes CUDA kernels to offload computation to the GPU, allowing concurrent path generation, distance calculation, and pheromone updates.

-**OpenMP Parallel** (C++): Employs OpenMP directives to parallelize the algorithm on CPU, enabling concurrent path generation, distance calculation, and pheromone updates.

Each implementation optimizes the Ant Colony Optimization algorithm by leveraging parallel processing capabilities, either on GPU (CUDA) or CPU (OpenMP), tailored to hardware capabilities and problem specifications.



Graph compares the execution time of the Ant Colony Optimization algorithm implemented in three different ways: serial, CUDA, and OpenMP. The x-axis of the graph represents the number of cities, while the y-axis represents the execution time in seconds. The three lines in the graph correspond to the three different implementations of the algorithm.

Serial refers to the traditional way of executing a program, one instruction at a time. In this case, the serial implementation of the Ant Colony Optimization algorithm is the slowest of the three, as shown by the blue line in the graph.

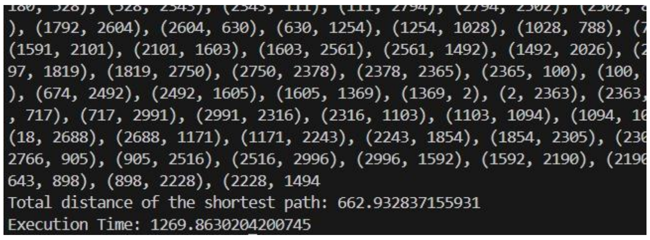
CUDA is a parallel computing platform developed by Nvidia that allows programmers to use the power of a graphics processing unit (GPU) for general-purpose computing. The CUDA implementation of the Ant Colony Optimization algorithm is significantly faster than the serial implementation, as shown by the green line in the graph.

OpenMP is a collection of compiler directives that support parallel programming in C, C++, and Fortran. The OpenMP implementation of the Ant Colony Optimization algorithm is faster than the serial implementation, but not as fast as the CUDA implementation, as shown by the red line in the graph.

Overall, the graph shows that the CUDA implementation of the Ant Colony Optimization algorithm is the fastest of the three, followed by the OpenMP implementation and then the serial implementation.

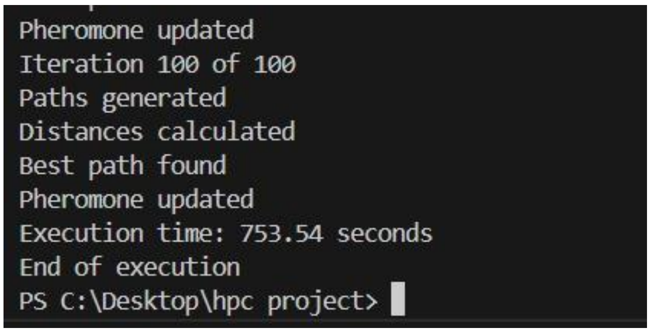
<u>Outputs:</u>

Cities-3000:



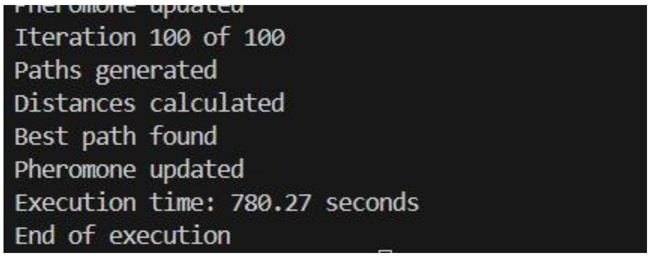
```
180, 328), (328, 2345), (2345, 111), (111, 2794), (2794, 2302), (2302,
), (1792, 2604), (2604, 630), (630, 1254), (1254, 1028), (1028, 788), (
(1591, 2101), (2101, 1603), (1603, 2561), (2561, 1492), (1492, 2026), (
97, 1819), (1819, 2750), (2750, 2378), (2378, 2365), (2365, 100), (100,
), (674, 2492), (2492, 1605), (1605, 1369), (1369, 2), (2, 2363), (2363,
, 717), (717, 2991), (2991, 2316), (2316, 1103), (1103, 1094), (1094, 1
(18, 2688), (2688, 1171), (1171, 2243), (2243, 1854), (1854, 2305), (23
2766, 905), (905, 2516), (2516, 2996), (2996, 1592), (1592, 2190), (219
643, 898), (898, 2228), (2228, 1494
Total distance of the shortest path: 662.932837155931
Execution Time: 1269.8630204200745
```

-serial code output



```
Pheromone updated
Iteration 100 of 100
Paths generated
Distances calculated
Best path found
Pheromone updated
Execution time: 753.54 seconds
End of execution
PS C:\Desktop\hpc project>
```

-openmp code output



```
Iteration 100 of 100
Paths generated
Distances calculated
Best path found
Pheromone updated
Execution time: 780.27 seconds
End of execution
```

-cuda code output

The project is distributed among people if it done by a group :-
Member 1 :-did serial code implementation using python.
Member 2 :-did openmp code implementation using C++
Program.
Member 3 :- did cuda code implementation using C++ program similar to the openmp but gives the better result.

Conclusion

In comparing the serial, CUDA, and OpenMP implementations for the Ant Colony Optimization algorithm, several crucial considerations emerge. Firstly, the selection depends significantly on performance needs and hardware availability. CUDA, leveraging GPU capabilities, typically offers the highest speed due to its parallel processing prowess, making it optimal for tackling large-scale problems. However, CUDA is constrained to systems with compatible NVIDIA GPUs and necessitates expertise in GPU-specific programming.

Conversely, OpenMP utilizes multi-core CPUs, striking a balance between speed and accessibility. It enables parallel execution on CPU cores, rendering it suitable for moderately sized problems and systems lacking GPUs. While not as swift as CUDA, OpenMP boasts easier implementation and greater portability across various hardware configurations.

In contrast, the serial implementation, though the simplest and most portable, may lack the performance required for larger problem sizes. It executes sequentially on a single CPU core and generally proves slower than CUDA and OpenMP for extensive optimization tasks.

Hence, the choice among serial, CUDA, and OpenMP implementations hinges on factors such as problem size, available hardware (GPU vs. CPU), programming proficiency, and desired performance levels. For maximum speed on extensive problems with compatible hardware, CUDA is the prime choice, while OpenMP offers a balance between performance and implementation simplicity for moderately sized tasks. Serial execution remains a viable option for smaller-scale assignments or environments without GPU support.

References

[1] Dorigo M, Stützle T. Ant Colony optimization. Cambridge, MA: MIT Press; 2004.

[2] Cecilia et al. Parallelization Strategies for Ant Colony Optimisation on GPUs. NIDISC 2011.

[3] N. Sureja, B. Chawda, “Random Travelling Salesman problem using Genetic Algorithms,” IFRSA’s International Journal Of Computing,Vol. 2, issue 2, April 2012.

[4] Kamil Rocki, Reiji Suda. Accelerating 2-opt and 3-opt local search using GPU in Travelling Salesman Problem. IEEE 2012.

[5] Julio Ponce1 , Francisco Ornelas1 , Alberto Hernández2 , Humberto Muñoz1 , Alberto Ochoa3 , Alejandro Padilla1 , Alfonso Recio. Implementation of a Parallel Ant Colony Algorithm Using CUDA and GPUs to Solve Routings Problems Problem