BEE COLONY OPTIMIZATION FOR TRAVELLING SALESMAN PROBLEM

MASTER OF TECHNOLOGY

In

COMPUTER SCIENCE ENGINEERING

Submitted By

Bandaru Lakshmi Tulasi
(23535004)

Mohammed Safwan
(23535011)

Ribhu Mondal
(23535025)

Vuda Jaswanth
(23535030)

Under the guidance of

Dr. Neteesh Kumar

(Professor, CSE Department)

Department of Computer Science and Engineering,
Indian Institute of Technology, Roorkee

Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem that has been extensively studied in the field of combinatorial optimization. In the TSP, a salesman needs to find the shortest possible route that visits each city exactly once and returns to the origin city. This problem has numerous real-world applications, including logistics, transportation planning, and circuit design.

To solve the TSP efficiently, various optimization algorithms have been developed, and one such approach is Bee Colony Optimization (BCO). BCO is a metaheuristic optimization algorithm inspired by the foraging behavior of honeybees. It mimics the process of bees searching for food sources and communicating information about good solutions to other bees in the colony.

In this report, we will discuss the implementation of the Bee Colony Optimization algorithm for solving the Traveling Salesman Problem in a sequential manner and in a parallel programming environment using OpenMP. We will explore how parallel programming can be utilized to improve the performance of the algorithm by leveraging multiple threads to search for solutions concurrently.

The goal of the Traveling Salesman Problem is to find the shortest tour that visits each city exactly once and returns to the starting city. Mathematically, given a set of n cities and the distance between each pair of cities, the objective is to minimize the total distance traveled.

Bee Colony Optimization:

Bee Colony Optimization is a population-based metaheuristic algorithm that mimics the foraging behavior of bees. In the context of the TSP, bees represent potential solutions (tours), and the algorithm iteratively improves these solutions by simulating the behavior of bees in a colony.

The algorithm consists of three types of bees:

Collector Bees: These bees explore the search space by iteratively improving the solutions they find.
Observer Bees: These bees evaluate the quality of solutions found by collector bees and share information about good solutions with other bees.
Scout Bees: These bees explore new regions of the search space if the algorithm becomes stuck in a local minimum.

The parallel implementation of the BCO algorithm using OpenMP involves distributing the workload across multiple threads to search for solutions concurrently. Each thread simulates the behavior of a subset of bees, allowing for faster exploration of the search space.

To evaluate the performance of the parallel program, we measure the time taken by both the sequential and parallel algorithms to find a solution for the TSP. We also analyze how the performance of the parallel program varies with the change in the number of threads used for parallel execution. By experimenting with different thread counts, we can determine the optimal number of threads that maximizes the performance of the parallel algorithm while taking into account factors such as CPU cores and system resources.

Implementation:

Struct Bee: This structure represents a bee in the colony. It contains fields to store the behavior of the bee ('F' for collector, 'O' for observer, 'S' for scout), the route it explores, the distance of the route, and an iterator to track iterations.

Global Variables: Various global variables are declared to configure the algorithm's parameters, such as mutation limit, number of bees, percentages of collector, observer, and scout bees, number of cycles, and a distance array to store distances between cities.

CalculateDistance: This function calculates the total distance traveled for a given route using the distances between cities provided in the distanceArray.

MutatePath: This function performs mutation on a bee's route by swapping two cities randomly. It excludes the first and last cities to maintain the route integrity.

Scout : This function implements the behavior of scout bees. It mutates the current route and evaluates the new route's distance. If the new route is shorter, it updates the bee's route and distance; otherwise, it increments the bee's iterator or marks it as a scout if the mutation limit is reached.

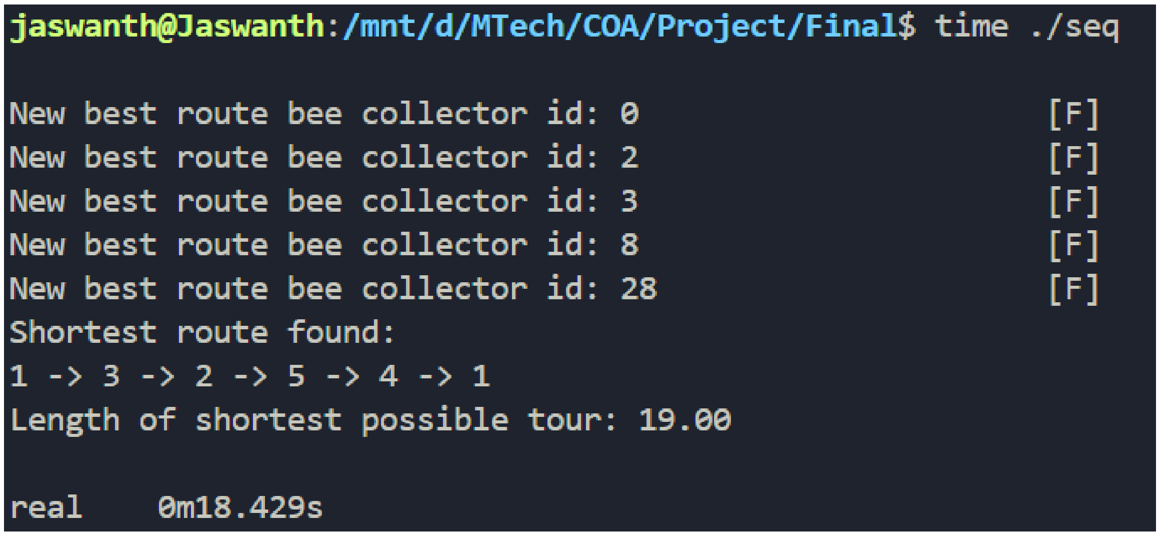
findNewPath: This function implements the behavior of collector bees. It shuffles the bee's route to explore new paths.

Dance: This function simulates the behavior of bees in the colony.

It iterates over all bees, allowing them to explore and improve their routes. If a collector bee finds a shorter route, it updates the best solution.

Results:

Sequential code output :

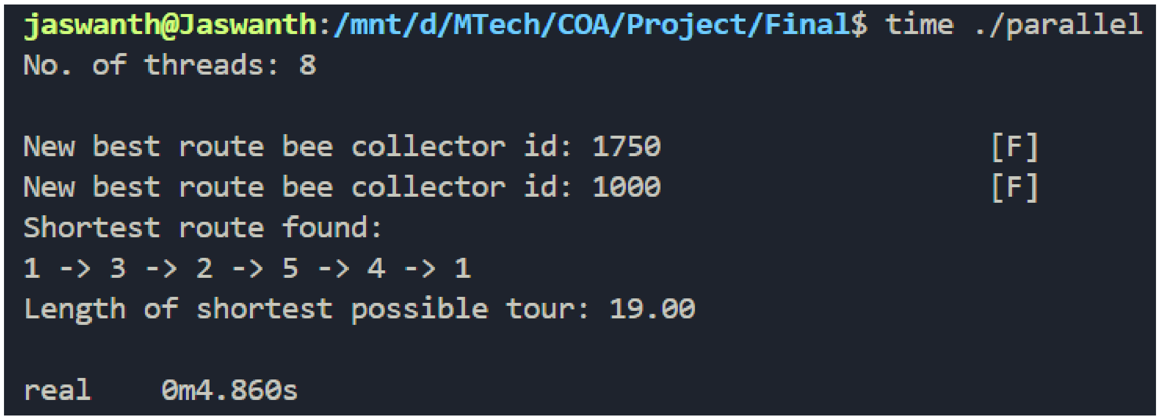


```
jaswanth@Jaswanth:/mnt/d/MTech/COA/Project/Final$ time ./seq

New best route bee collector id: 0                         [F]
New best route bee collector id: 2                         [F]
New best route bee collector id: 3                         [F]
New best route bee collector id: 8                         [F]
New best route bee collector id: 28                        [F]
Shortest route found:
1 -> 3 -> 2 -> 5 -> 4 -> 1
Length of shortest possible tour: 19.00

real    0m18.429s
```

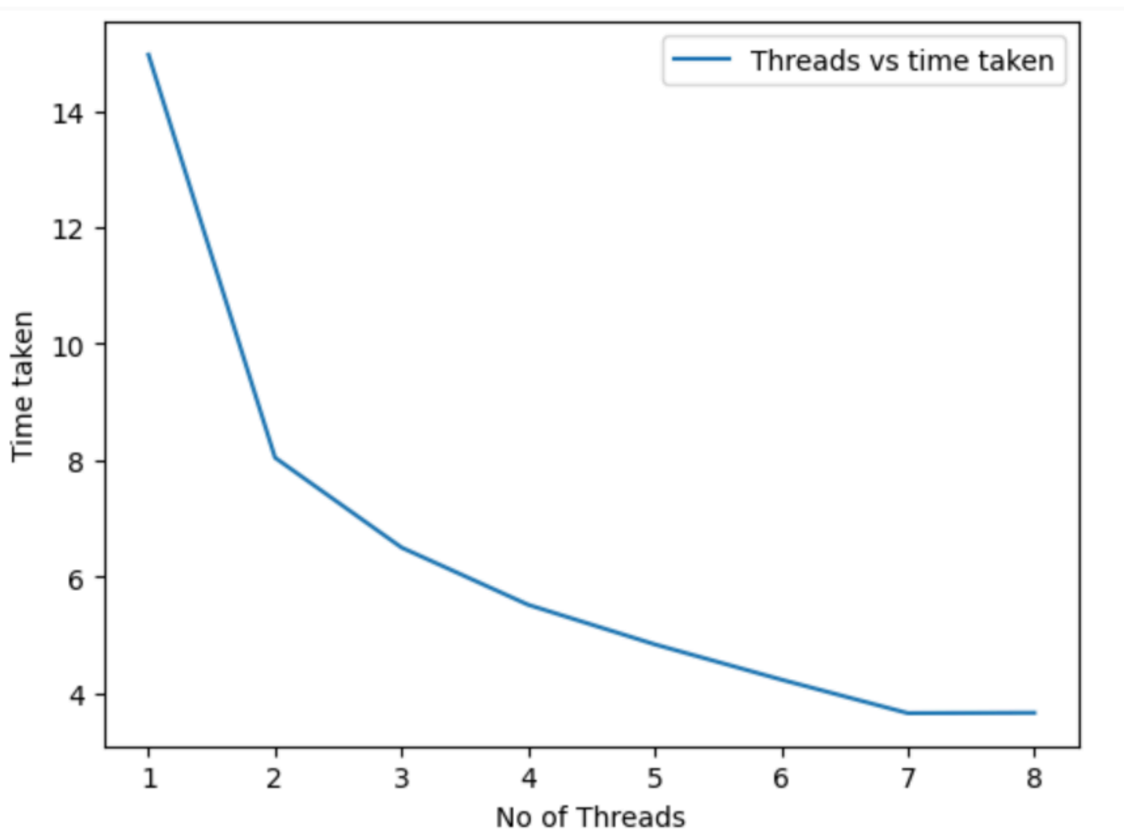
Parallel code output :



```
jaswanth@Jaswanth:/mnt/d/MTech/COA/Project/Final$ time ./parallel
No. of threads: 8

New best route bee collector id: 1750                      [F]
New best route bee collector id: 1000                      [F]
Shortest route found:
1 -> 3 -> 2 -> 5 -> 4 -> 1
Length of shortest possible tour: 19.00

real    0m4.860s
```



Conclusion:

In conclusion, the Bee Colony Optimization algorithm offers an effective approach to solving the Traveling Salesman Problem by leveraging the collective intelligence of a bee colony. By parallelizing the algorithm using OpenMP, we can further enhance its performance by exploiting parallelism and distributing the computational workload across multiple threads. Through experimentation and analysis, we can gain insights into how the performance of the parallel program varies with different thread counts, ultimately optimizing the algorithm's efficiency for solving large-scale instances of the TSP.