UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA – UTEC
Lima, Perú

Parallel Resolution of the Traveling Salesman Problem Using Branch and Bound: Application to VLSI Data

Final Project - Parallel and Distributed Computing
CS4052

Mariana Capuñay – 202120625 100%
Virginia Puente – 202110281 100%
Romina Romani – 202110320 100%

Professor
José Antonio Fiestas Iquira

July, 2025

CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

The traveling salesman problem (TSP) is a classic problem widely studied across numerous fields, including algorithms, optimization, and computational complexity. According to Hoffman et al. [1], this well-known algorithm has garnered significant interest from mathematicians and computer scientists, specifically due to its concise description and the complexity of its solutions.

The problem is as follows:

> A traveling salesman has a predefined $N-1$ cities to visit (stopping at each one exactly once) before returning to his home city. The challenge is to determine the least costly route for the traveling salesman to complete his journey.

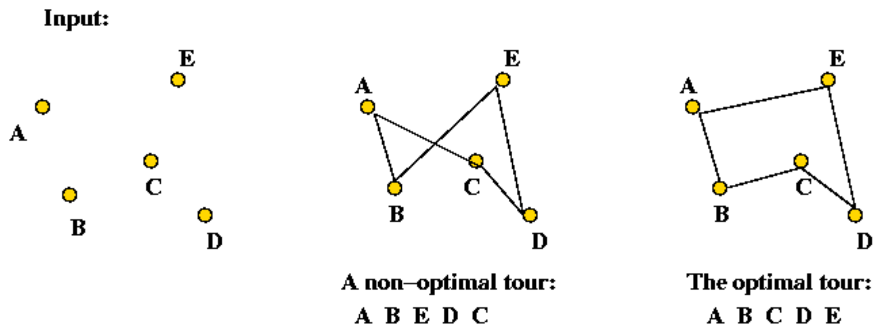


Figure 1.1: Comparison of a non-optimal and the optimal tour for a 5-city TSP instance

Junger et al. [2] indicate that the Traveling Salesman Problem (TSP) was one of the first problems established as NP-Hard in 1972. Consequently, various algorithmic methods – including branch and bound, cutting plane techniques, and local optimization – were developed to demonstrate their effectiveness in solving the TSP.

1.1 Classification

The algorithm can be classified as either a symmetric or asymmetric traveling salesman problem (sTSP and aTSP, respectively). We will focus on the first one, where the distances between cities satisfy $d(c_i, c_j) = d(c_j, c_i)$ for $1 < i, j < N$.

1.2 Complexity

Matai et al. [3] describe the complexity of the TSP problem as $(N-1)!/2$, where N represents the number of cities to be visited. This expression is derived by considering the total number of possible routes that encompass all cities in the problem.

1.3 The branch and bound approach

The method operates through two main processes: branching and bounding. Branching involves dividing the problem into smaller subproblems by making a decision at each level, thus expanding the search tree. Bounding calculates an estimate (a lower or upper bound) of the best possible solution that can be obtained from a given node. If this bound is worse than the best solution found so far, that branch is pruned, meaning it is not explored any further.

The goal of Branch and Bound is to avoid exhaustive enumeration of all possibilities by discarding suboptimal paths early. This is especially useful in problems where the number of possible solutions grows exponentially.

For example, in the Traveling Salesman Problem, the algorithm starts at a city and tries different orders of visiting the other cities. At each partial path, it computes a lower bound on the total tour cost. If this bound exceeds the cost of the best known complete tour, the path is pruned.

Chapter 2

SEQUENTIAL IMPLEMENTATION OF THE TSP ALGORITHM

2.1 Sequential Brute Force Algorithm for TSP

The brute force algorithm for the Traveling Salesman Problem (TSP) works by generating all possible routes that visit each city exactly once and return to the starting city. One city (in this case, city 0) is fixed as the starting point, and the algorithm generates all permutations of the remaining cities. For each permutation, it calculates the total cost by summing the distances between consecutive cities and the return leg to the origin. It compares all these possible tour costs and keeps track of the minimum-cost route.

This method guarantees the optimal solution, but its major drawback is its computational cost: the number of permutations grows factorially with the number of cities. As a result, it becomes infeasible for N greater than around 10, since it must evaluate millions of possible paths without applying any intelligent pruning or heuristics.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Point {
    double x, y;
};

int dimension = 0;

vector<Point> parseTSPLIB(const string& filename) {
    ifstream infile(filename); string line;
    while (getline(infile, line)) {
        if (line.find("NODE_COORD_SECTION") != string::npos) break;
        if (line.find("DIMENSION") != string::npos) {
            stringstream ss(line); string tmp;
            while (ss >> tmp) {
                if (isdigit(tmp[0])) {
                    dimension = stoi(tmp); break;
                }
            }
        }
    }

    int index; double x, y;
    vector<Point> coords(dimension);

    for (int i = 0; i < dimension; ++i) {
        infile >> index >> x >> y;  coords[i] = {x, y};
    }

    return coords;
}

vector<vector<int>> computeDistanceMatrix(const vector<Point>& points)
↪ {
    int n = points.size();
    vector<vector<int>> adj(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j) {
                double dx = points[i].x - points[j].x;
                double dy = points[i].y - points[j].y;
                adj[i][j] = round(sqrt(dx * dx + dy * dy));
            }
    return adj;
}

int solveTSPBrute(const vector<vector<int>>& adj, vector<int>&
↪  best_path) {
    int N = adj.size();   vector<int> cities;
    for (int i = 1; i < N; ++i)
        cities.push_back(i); // Start from 0, permute the rest

    int min_cost = INT_MAX;

    do {
        int cost = 0, prev = 0;
        for (int i = 0; i < cities.size(); ++i) {
            cost += adj[prev][cities[i]];
            prev = cities[i];
        }
        cost += adj[prev][0]; // Return to origin

        if (cost < min_cost) {
                min_cost = cost;   best_path = {0};
                best_path.insert(best_path.end(), cities.begin(),
                ↪  cities.end());
            }
        } while (next_permutation(cities.begin(), cities.end()));

        return min_cost;
}

int main(int argc, char* argv[]) {
    if (argc < 2) return 1;

    vector<int> best_path;
    string filename = argv[1];
    vector<Point> points = parseTSPLIB(filename);
    vector<vector<int>> adj = computeDistanceMatrix(points);

    auto start = chrono::high_resolution_clock::now();
    int min_cost = solveTSPBrute(adj, best_path);
    auto end = chrono::high_resolution_clock::now();

    cout << "File: " << filename << endl;
    cout << "Min Distance: " << min_cost << endl;
    cout << "Execution time: " << chrono::duration<double>(end -
    ↪  start).count() << " segundos" << endl;

    ofstream times_out("times_brute_force.txt", std::ios::app);
    times_out << dimension << " " <<
    ↪  chrono::duration<double>(end-start).count() << "\n";
    times_out.close();

    ofstream out("route_brute_force" + to_string(dimension) + ".txt");
    for (int i : best_path)
        out << points[i].x << " " << points[i].y << "\n";
    out << points[best_path[0]].x << " " << points[best_path[0]].y <<
    ↪  "\n";
    out << "Distance: " << min_cost << endl;
    out.close();
    return 0;
}
```

Listing 1: Brute-force TSP solver implementation

Listing 1 presents the implementation of a brute-force algorithm for solving the Traveling Salesman Problem (TSP). The program begins by reading city coordinates from a TSPLIB-formatted file to build a symmetric distance matrix. It then evaluates all possible permutations of tours starting from city 0, identifying the minimum-cost path, which is subsequently written to an output file along with the execution time. Due to its factorial time complexity, this approach is only practical for small instances and serves primarily to validate the correctness of later implementations.

To assess the time complexity of this solution, we executed the program on uniformly distributed datasets, varying the number of cities from $N = 2$ to $N = 14$.

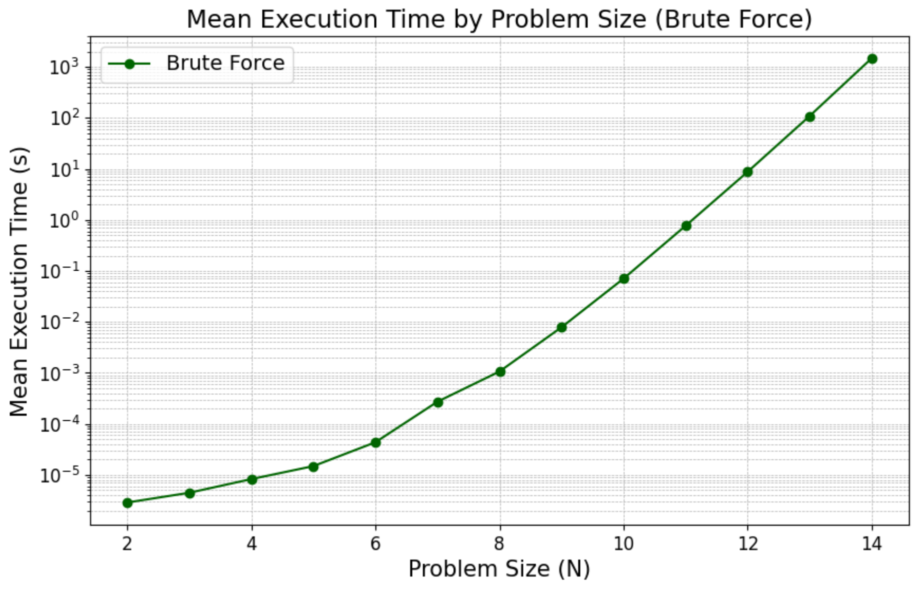


Figure 2.1: Execution time of the brute-force TSP algorithm on a logarithmic scale for problem sizes from 2 to 14 cities

As shown in Fig. 2.1, the execution time increases factorially with N (the number of cities in the problem). Consequently, we can improve execution time by applying a branch-and-bound strategy to avoid exploring non-promising routes.

2.2 Branch and Bound Algorithm for TSP

Our implementation is based on the approach described in [4]. This algorithm aims to find the shortest route that visits each city exactly once and returns to the starting point, but does so more efficiently than brute-force methods. It constructs a search tree of partial tours and uses lower bound estimates to

eliminate paths that cannot yield a better solution than the best one found so far. If the current cost plus the bound exceeds the best-known solution, the corresponding node is pruned.
The lower bound is estimated using a heuristic based on the sum of the two smallest outgoing edges from each city. As the tour progresses, the bound is updated accordingly. Although the worst-case time complexity remains factorial, in practice, Branch and Bound explores significantly fewer paths and is considerably more scalable—particularly when combined with a well-designed bounding function.

```
#include <bits/stdc++.h>
using namespace std;

int dimension = 0;

void copyToFinal(const vector<int>& curr_path, vector<int>& final_path)
↪ {
    int N = curr_path.size();
    final_path.resize(N + 1);
    for (int i = 0; i < N; i++) final_path[i] = curr_path[i];
    final_path[N] = curr_path[0]; // Closing the loop
}

struct Point {
    double x, y;
};

int firstMin(const vector<vector<int>>& adj, int i) {
    int min = INT_MAX;
    for (int k = 0; k < adj.size(); ++k)
        if (adj[i][k] < min && i != k) min = adj[i][k];
    return min;
}

int secondMin(const vector<vector<int>>& adj, int i) {
    int first = INT_MAX, second = INT_MAX;
    for (int j = 0; j < adj.size(); ++j) {
        if (i == j) continue;
        if (adj[i][j] <= first) {
            second = first;
            first = adj[i][j];
        } else if (adj[i][j] <= second && adj[i][j] != first) {
            second = adj[i][j];
        }
    }
    return second;
}

void TSPRec(const vector<vector<int>>& adj, int curr_bound, int
↪   curr_weight,
            int level, vector<int>& curr_path, vector<bool>& visited,
            ↪   int& final_res, vector<int>& final_path) {

    int N = adj.size();
    if (level == N) {
        if (adj[curr_path[level - 1]][curr_path[0]] != 0) {
            int curr_res = curr_weight + adj[curr_path[level -
            ↪   1]][curr_path[0]];
            if (curr_res < final_res){
                final_res = curr_res;
                final_path = curr_path; // save the best path
            }
        }
        return;
    }

    for (int i = 0; i < N; ++i) {
        if (adj[curr_path[level - 1]][i] != 0 && !visited[i]) {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];

            if (level == 1)
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) +
                ↪   firstMin(adj, i)) / 2);
            else
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) +
                ↪   secondMin(adj, i)) / 2);

            if (curr_bound + curr_weight < final_res) {
                curr_path[level] = i; visited[i] = true;
                TSPRec(adj, curr_bound, curr_weight, level + 1,
                ↪   curr_path, visited, final_res, final_path);
            }

            curr_weight -= adj[curr_path[level - 1]][i];
            curr_bound = temp;
            fill(visited.begin(), visited.end(), false);
            for (int j = 0; j <= level - 1; ++j)
                visited[curr_path[j]] = true;
        }
    }
}

int solveTSP(const vector<vector<int>>& adj, vector<int>& final_path) {
    int N = adj.size();
    vector<int> curr_path(N + 1, -1);

    vector<bool> visited(N, false);
    int curr_bound = 0;

    for (int i = 0; i < N; ++i)
        curr_bound += (firstMin(adj, i) + secondMin(adj, i));

    curr_bound = (curr_bound & 1)? curr_bound / 2 + 1 : curr_bound / 2;
    visited[0] = true;  curr_path[0] = 0;

    int final_res = INT_MAX;
    TSPRec(adj, curr_bound, 0, 1, curr_path, visited, final_res,
    ↪  final_path);
    return final_res;
}

vector<Point> parseTSPLIB(const string& filename) {
    ifstream infile(filename);  string line;

    while (getline(infile, line)) {
        if (line.find("NODE_COORD_SECTION") != string::npos)
            break;
        if (line.find("DIMENSION") != string::npos) {
            stringstream ss(line);  string tmp;
            while (ss >> tmp) {
                if (isdigit(tmp[0])) {
                    dimension = stoi(tmp); break;
                }
            }
        }
    }

    vector<Point> coords(dimension);
    for (int i = 0; i < dimension; ++i) {
        int index;  double x, y;
        infile >> index >> x >> y; coords[i] = {x, y};
    }

    return coords;
}

vector<vector<int>> computeDistanceMatrix(const vector<Point>& points)
↪ {
    int n = points.size();
    vector<vector<int>> adj(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j) {
                double dx = points[i].x - points[j].x;
                double dy = points[i].y - points[j].y;
                adj[i][j] = round(sqrt(dx * dx + dy * dy));
            }
    return adj;
}

int main(int argc, char* argv[]) {
    if (argc < 2) return 1;

    vector<int> final_path; // to save the optimal route
    string filename = argv[1];
    vector<Point> points = parseTSPLIB(filename);
    vector<vector<int>> adj = computeDistanceMatrix(points);

    auto start = chrono::high_resolution_clock::now();
    int min_cost = solveTSP(adj, final_path);
    auto end = chrono::high_resolution_clock::now();

    cout << "File: " << filename << endl;
    cout << "Min Distance: " << min_cost << endl;
    cout << "Execution Time: " << chrono::duration<double>(end -
    ↪  start).count() << " segundos" << endl;

    ofstream times_out("time_branch_bound.txt", std::ios::app);
    times_out << dimension << " " <<
    ↪  chrono::duration<double>(end-start).count()<< " " <<  1 <<"\n";
    times_out.close();

    ofstream out("ruta_"+to_string(dimension)+".txt");
    final_path.resize(points.size());

    for (int i : final_path)
        out << points[i].x << " " << points[i].y << "\n";
    out << points[final_path[0]].x << " " << points[final_path[0]].y <<
    ↪  "\n";
    out << "Distance: "<<min_cost<<endl;
    out.close();
    return 0;
}
```

Listing 2: Branch&Bound TSP solver implementation

Listing 2 shows the implementation of the Branch-and-Bound algorithm for the problem. The algorithm, as the previous one, parses the dataset and make the symmetric matrix of distances. Then, it builds a search tree of partial tours, using lower bound estimates to prune suboptimal paths early, significantly reducing the number of computations compared to brute-force.

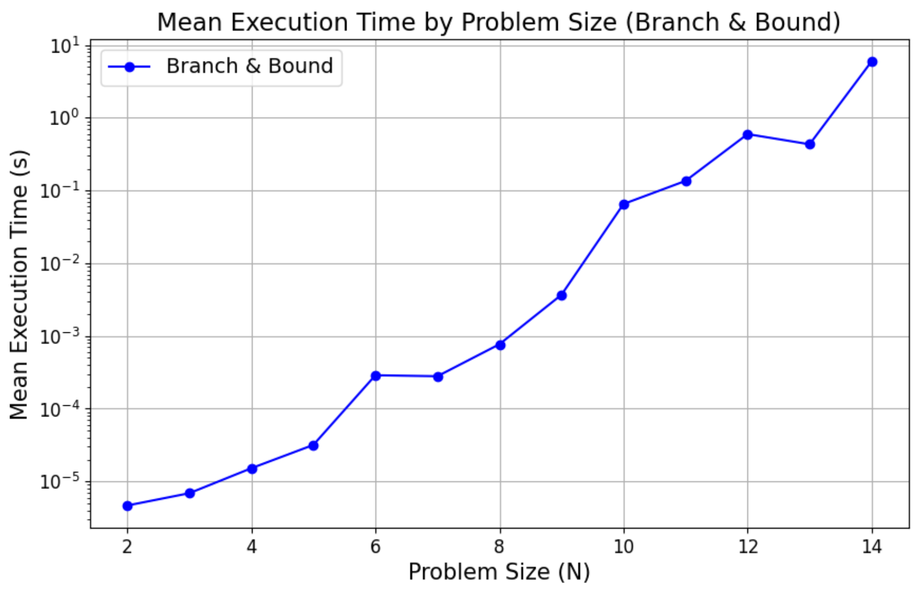


Figure 2.2: Execution time of the branch and bound, sequential TSP algorithm on a logarithmic scale for problem sizes from 2 to 14 cities

As shown in Fig. 2.2, the execution time still increases with N (the number of cities in the problem), but at a slower rate. Consequently, we can improve execution time by applying a parallel branch-and-bound strategy to better balance the workload.

2.3 Sequential Brute Force vs Sequential Branch and Bound

Both programs produce the same optimal paths, but the most notable difference lies in their execution times. As can be seen, in most cases the Branch and Bound algorithm maintains a significantly lower execution time, while the brute-force approach only increases more. Figure 2.3 shows the time optimization achieved by using the Branch and Bound approach.

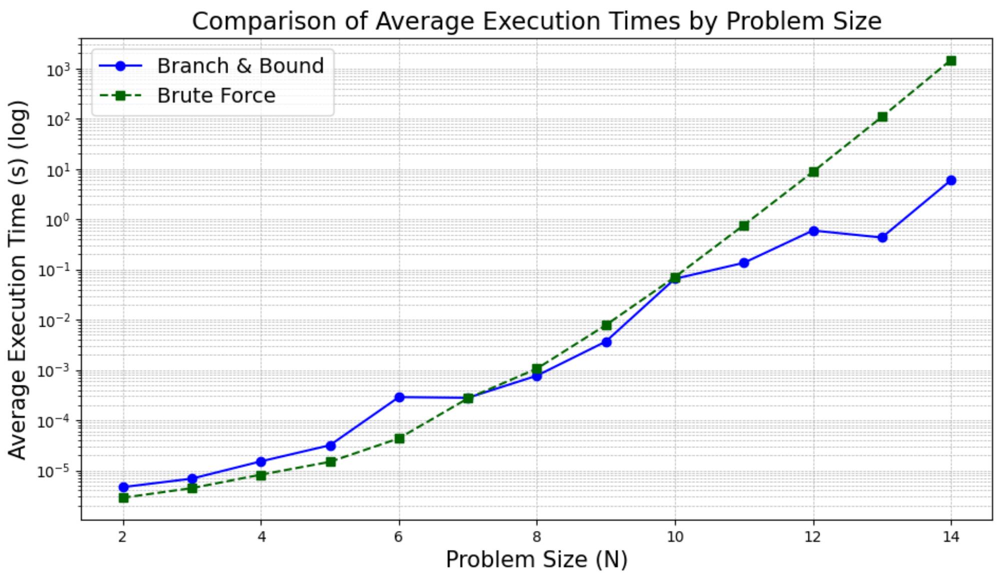
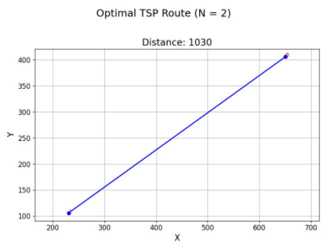


Figure 2.3: Comparison of execution times between brute-force and branch-and-bound TSP algorithms on a logarithmic scale for problem sizes from 2 to 14 cities

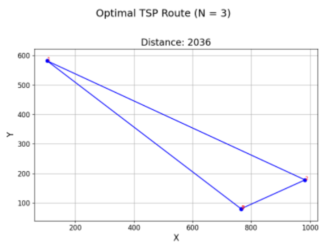
In order to ensure a fair comparison between both sequential programs, we used randomly generated datasets. Each program was executed five times for every problem size, and the average execution time was computed. All experiments were carried out on the Khipu supercomputer at UTEC to guarantee consistent performance conditions.
Figure 2.3 shows that particularly, for 6 cities, the brute force approach is even faster than the Branch and Bound. However, as the problem size increases, the optimization provided by Branch and Bound becomes evident, with execution times remaining significantly lower in comparison.

2.4 Visualization of Optimal Routes

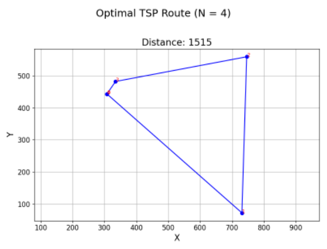
The optimal routes found by both algorithms are presented below (see Fig. 2.4).



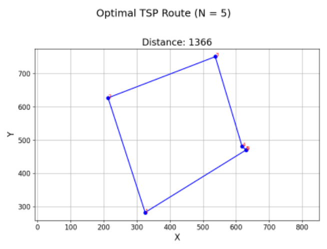
(a) Route for N = 2



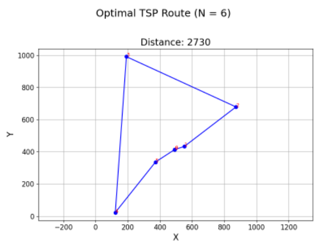
(b) Route for N = 3



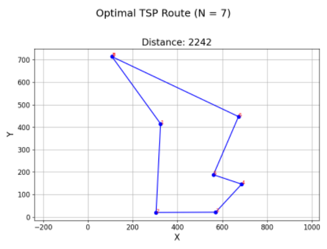
(c) Route for N = 4



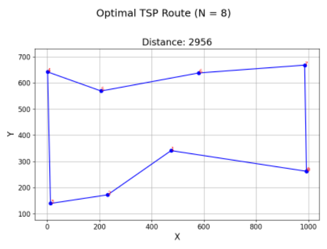
(d) Route for N = 5



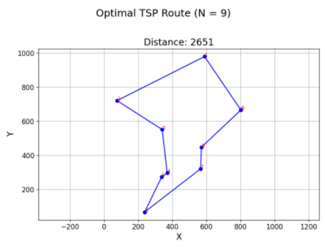
(e) Route for N = 6



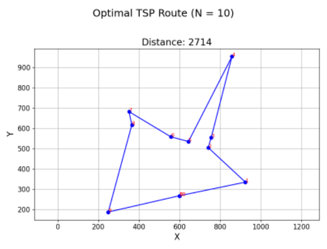
(f) Route for N = 7



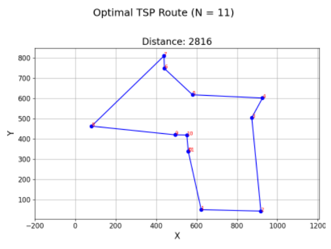
(g) Route for N = 8



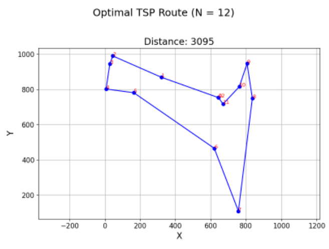
(h) Route for N = 9



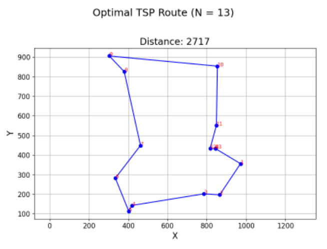
(i) Route for N = 10



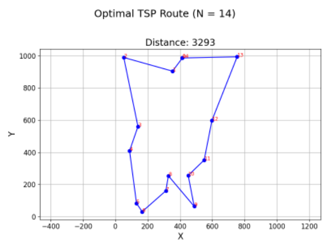
(j) Route for N = 11



(k) Route for N = 12



(l) Route for N = 13



(m) Route for N = 14

Figure 2.4: Optimal TSP routes obtained for problem sizes from $N = 2$ to $N = 14$ using Brute Force and the Branch and Bound algorithms.

Chapter 3

PARALLEL IMPLEMENTATION OF THE TSP USING BRANCH AND BOUND

3.1 Parallel Paradigm and PRAM Model

Due to the rapid growth in execution time with increasing problem sizes, even the Branch and Bound approach becomes inefficient. This motivated the development of a parallel and more efficient version to improve performance on larger TSP instances.
To design our parallel solution, we adopted the **CREW PRAM** model (Concurrent Read, Exclusive Write), which allows multiple processors to read shared variables concurrently, while only one processor may write to a shared location at any given time. This model provides a theoretical abstraction that fits well with our practical implementation using OpenMP.
The core idea behind our parallel strategy is to reduce the search space exploration time by assigning subtrees of the decision tree to different processors. Specifically, the parallelism is structured in two levels:

- **Step 1 (Bound Initialization):** All N processors participate in the parallel computation of the initial lower bound. Each processor P_i computes the sum of the smallest and second smallest outgoing edges for city i. The resulting values are reduced to obtain the global bound, which is used to guide pruning decisions during the search.

- **Step 2 (Subtree Distribution):** Once the initial bound is calculated, the root of the search tree (i.e., city 0) is expanded to generate $N-1$ possible first moves. Each of these paths is assigned to a distinct processor (in total, $N-1$ processors), where each processor continues the exploration of its assigned subtree independently, using the same Branch and Bound logic recursively from level 2 onward.

To prevent race conditions during updates to the shared variables `final_res` and `final_path`, we use a critical section (mutex or lock) every time a processor finds a potentially better solution. This ensures that concurrent writes do not lead to inconsistencies in the best-known solution.

The following pseudocode illustrates our PRAM model for parallelizing the Branch and Bound algorithm:

PRAM Model for Parallel Branch and Bound TSP

```
Input:
$adj[N][N]$                          // Symmetric cost matrix between cities

Output:
$final\_res$                         // Minimum total distance found
$final\_path[N]$                     // Corresponding optimal path

Shared variables:
$final\_res := \infty$
$final\_path := [-1, -1, \ldots, -1]$

  $curr\_bound := 0$
Step 1: Bound Initialization ($N$ processors)
for $i$ from 0 to $N-1$ pardo:
        $curr\_bound += firstMin(adj, i) + secondMin(adj, i)$
if $curr\_bound$ is odd: $curr\_bound := curr\_bound/2 + 1$
else $curr\_bound := curr\_bound/2$

Step 2: Level-1 parallelization ($N-1$ processors)
for each $i$ from 1 to $N-1$ pardo:
        if $adj[0][i] \neq 0$:
          Initialize $curr\_path, visited, curr\_weight, local\_bound$
          Call TSPRecParallel(...) at level 2

TSPRecParallel(...):
if level == $N$ and $adj[curr\_path[N-1]][0] \neq 0$:
      Compute $curr\_res$. Update $final\_res, final\_path$ with lock
for each $j$ from 0 to $N-1$:
if not visited and $adj[curr\_path[level-1]][j] \neq 0$:
         Compute $temp\_bound, temp\_weight$
         if cost is promising: recurse deeper
```

This parallelization scheme makes efficient use of the PRAM model and allows the algorithm to scale with the number of available processors. The CREW constraint is satisfied because only read operations occur concurrently (e.g., reading from the adjacency matrix), while updates to shared results are serialized via locking mechanisms.

3.2 Parallel Algorithm

Listing 3 presents the full implementation of the parallel Branch and Bound algorithm using OpenMP. The program is organized into several modular functions, each responsible for a key component of the TSP resolution process. The most important elements are described below:

- **Data Structures:** The algorithm defines a `Point` struct to store city coordinates and uses 2D matrices (`adj`) to represent symmetric distance matrices between cities.
- **First and Second Minimum:** The functions `firstMin()` and `secondMin()` compute the two shortest outgoing edges for each city. These values are used to calculate lower bounds that guide the pruning process during search.
- **Recursive Search (`TSPRec`):** This is the heart of the Branch and Bound algorithm. It recursively explores the search tree from a given level, pruning paths whose cumulative cost (current weight + lower bound) exceeds the best solution found so far. The recursion ends when a complete path is constructed (i.e., `level == N`), at which point the cost is evaluated and potentially updates the global best.
- **Parallel Execution (`solveTSP`):**
  - **Parallel Bound Initialization:** The computation of the initial lower bound (`curr_bound`) is distributed among N threads using the directive `#pragma omp parallel for reduction(+:curr_bound)`. Each thread P_i calculates the sum of the smallest and second smallest distances (`firstMin` and `secondMin`) for a distinct city i, and the results are accumulated using a reduction operation. This step corresponds to the **Step 1** of the PRAM model and is embarrassingly parallel, as each computation is independent and involves only reading from the shared adjacency matrix.
  - **Parallel Subtree Exploration:** Once the bound is initialized, the main parallelism is introduced in the exploration of subtrees from

the root (city 0) to each of the remaining $N-1$ cities. This corresponds to the **Step 2** of the PRAM model. The loop:

```
#pragma omp parallel for shared(final_res,
        best_path) schedule(dynamic)
```

assigns to each thread a unique index i from 1 to $N-1$, representing the second city in the path. Each thread builds a partial solution `0 → i` and explores the corresponding subtree recursively using the function `TSPRec()` from level 2 onward. This structure enables task-level parallelism and avoids overlap between threads, as each thread works on a disjoint portion of the search tree.

  - **Local vs Shared Variables:** To ensure thread safety, each thread operates on its own local copies of the following variables: `curr_path`, `visited`, `curr_weight`, `local_res` and `local_path`. This isolation prevents data races during recursion and avoids the need for locks within the recursive search.
  - **Critical Section for Solution Update:** When a thread finds a better solution (i.e., a tour with lower total cost than `final_res`), it must update the shared best result. This operation is protected by a critical section using `#pragma omp critical`, ensuring that only one thread at a time modifies the global variables `final_res` and `best_path`. Although this introduces some synchronization overhead, the frequency of such updates is typically low, since only promising paths reach the leaf level with improved cost.

- **I/O Utilities:** The function `parseTSPLIB()` reads a TSPLIB-formatted file and extracts city coordinates. `computeDistanceMatrix()` constructs the symmetric distance matrix based on Euclidean distances between points.

- **Main Function:** The `main()` function handles input parameters, reads the problem instance, computes the optimal route using the parallel solver, and writes results to both the console and output files. The number of threads is set via command-line argument using `omp_set_num_threads()`.

This parallel approach reduces overall runtime by distributing subtrees among processors and avoids redundant computation through effective pruning.

Although the theoretical complexity remains factorial, the parallelism allows tackling larger instances than the sequential versions.

```cpp
#include <bits/stdc++.h>
#include "omp.h"
using namespace std;

int dimension = 0, th = 0;

void copyToFinal(const vector<int>& curr_path, vector<int>& final_path)
↪ {
    int N = curr_path.size();
    final_path.resize(N + 1);
    for (int i = 0; i < N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0]; // Closing the loop
}

struct Point {
    double x, y;
};

int firstMin(const vector<vector<int>>& adj, int i) {
    int min = INT_MAX;
    for (int k = 0; k < adj.size(); ++k)
        if (adj[i][k] < min && i != k)
            min = adj[i][k];
    return min;
}

int secondMin(const vector<vector<int>>& adj, int i) {
    int first = INT_MAX, second = INT_MAX;
    for (int j = 0; j < adj.size(); ++j) {
        if (i == j) continue;
        if (adj[i][j] <= first) {
            second = first;
            first = adj[i][j];
        } else if (adj[i][j] <= second && adj[i][j] != first) {
            second = adj[i][j];
        }
    }
    return second;
}

```

```
void TSPRec(const vector<vector<int>>& adj, int curr_bound, int curr_weight,
            int level, vector<int>& curr_path, vector<bool>& visited, int& final_res, vector<int>& final_path) {

    int N = adj.size();
    if (level == N) {
        if (adj[curr_path[level - 1]][curr_path[0]] != 0) {
            int curr_res = curr_weight + adj[curr_path[level - 1]][curr_path[0]];
            if (curr_res < final_res){
                final_res = curr_res;
                final_path = curr_path; // save the best path
            }
        }
        return;
    }

    for (int i = 0; i < N; ++i) {
        if (adj[curr_path[level - 1]][i] != 0 && !visited[i]) {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level - 1]][i];

            if (level == 1)
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2);
            else
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) + secondMin(adj, i)) / 2);

            if (curr_bound + curr_weight < final_res) {
                curr_path[level] = i;
                visited[i] = true;
                TSPRec(adj, curr_bound, curr_weight, level + 1, curr_path, visited, final_res, final_path);
            }

            curr_weight -= adj[curr_path[level - 1]][i];
            curr_bound = temp;
            fill(visited.begin(), visited.end(), false);
            for (int j = 0; j <= level - 1; ++j)
                visited[curr_path[j]] = true;
        }
    }
```

```
}

int solveTSP(const vector<vector<int>>& adj, vector<int>& final_path) {
    int N = adj.size();
    vector<int> curr_path(N + 1, -1);

    vector<bool> visited(N, false);
    int curr_bound = 0;

    #pragma omp parallel for reduction(+:curr_bound) schedule(dynamic)
    for (int i = 0; i < N; ++i)
        curr_bound += (firstMin(adj, i) + secondMin(adj, i));

    curr_bound = (curr_bound & 1) ? curr_bound / 2 + 1 : curr_bound /
    ↪  2;
    int final_res = INT_MAX;
    vector<int> best_path;

    #pragma omp parallel for shared(final_res, best_path)
    ↪  schedule(dynamic)
    for (int i = 1; i < N; ++i) {
        vector<int> curr_path(N + 1, -1);
        vector<bool> visited(N, false);
        int local_res = INT_MAX;
        vector<int> local_path;

        visited[0] = visited[i] = true;
        curr_path[0] = 0;
        curr_path[1] = i;
        int curr_weight = adj[0][i];
        int bound = curr_bound - ((firstMin(adj, 0) + firstMin(adj, i))
        ↪  / 2);

        TSPRec(adj, bound, curr_weight, 2, curr_path, visited,
        ↪  local_res, local_path);

        #pragma omp critical
        { // critical section
            if (local_res < final_res) {
                final_res = local_res;
                best_path = local_path;
            }
        }
    }

    best_path.push_back(best_path[0]);
    final_path = best_path;
    return final_res;
}

vector<Point> parseTSPLIB(const string& filename) {
    ifstream infile(filename); string line;
    while (getline(infile, line)) {
        if (line.find("NODE_COORD_SECTION") != string::npos) break;
        if (line.find("DIMENSION") != string::npos) {
            stringstream ss(line); string tmp;
            while (ss >> tmp) {
                if (isdigit(tmp[0])) {
                    dimension = stoi(tmp); break;
                }
            }
        }
    }
    vector<Point> coords(dimension);
    for (int i = 0; i < dimension; ++i) {
        int index;
        double x, y;
        infile >> index >> x >> y;
        coords[i] = {x, y};
    }

    return coords;
}

vector<vector<int>> computeDistanceMatrix(const vector<Point>& points)
↪ {
    int n = points.size();
    vector<vector<int>> adj(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j) {
                double dx = points[i].x - points[j].x;
                double dy = points[i].y - points[j].y;
                adj[i][j] = round(sqrt(dx * dx + dy * dy));
            }
    return adj;
}

```

```
int main(int argc, char* argv[]) {
    if (argc < 2) return 1;

    vector<int> final_path;
    string filename = argv[1];
    th = stoi(argv[2]); omp_set_num_threads(th);

    vector<Point> points = parseTSPLIB(filename);
    vector<vector<int>> adj = computeDistanceMatrix(points);

    auto start = omp_get_wtime();
    int min_cost = solveTSP(adj, final_path);
    auto end = omp_get_wtime();

    cout << "File: " << filename << endl;
    cout << "Min Distance: " << min_cost << endl;
    cout << "Execution Time: " << end - start << " segundos" << endl;

    ofstream times_out("time_omp.txt", std::ios::app);
    times_out << dimension << " " << end-start << " " <<  th <<"\n";
    times_out.close();

    ofstream out("ruta_"+to_string(dimension)+".txt");
    final_path.resize(points.size());

    for (int i: final_path)
        out << points[i].x << " " << points[i].y << "\n";

    out << points[final_path[0]].x << " " << points[final_path[0]].y <<
    ↪  "\n";
    out << "Distance: "<<min_cost<<endl;
    out.close();

    return 0;
}
```

Listing 3: Parallel Branch&Bound TSP solver implementation

3.3 Development Iterations of the Parallel Algorithm

To ensure the robustness and correctness of the parallel implementation, the algorithm was progressively refined through four successive iterations. Each version addressed different issues, from validation and correctness to performance optimization.

- **Version 1 – Correction of the Bound Function in the Sequential Algorithm:** During early testing of the sequential Branch and Bound implementation, a discrepancy was found between the results of our code and the brute-force algorithm for small instances (e.g., $N = 4$). After reviewing the code, we identified that the bound calculation —originally based on the GeeksforGeeks implementation [4]— occasionally overestimated the true cost. This occurred because the selection of the two smallest outgoing edges did not properly exclude duplicates or correctly apply the bound formula. We corrected the function to ensure it computes a valid (admissible) lower bound, guaranteeing correctness and improving pruning efficiency. This correction became the foundation for all subsequent versions.

- **Version 2 – Basic Parallelization (Initial Bound Only):** The first parallel version introduced concurrency solely in the computation of the initial lower bound using `#pragma omp parallel for reduction`. The rest of the algorithm was left sequential. This version served to validate the parallel environment setup and the correctness of reductions in OpenMP.

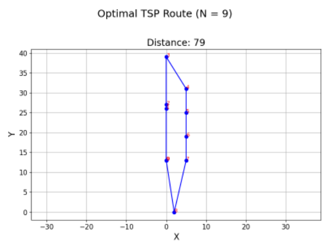
- **Version 3 – Subtree Distribution Without Synchronization:** The next version distributed subtrees rooted at each possible second city (i.e., starting from node 0 to node i) across multiple threads. However, the version lacked synchronization when updating shared variables like `final_res`, which caused race conditions and inconsistent results.

- **Version 4 – Full Parallelization with Synchronization:** In the final version, a critical section was introduced to protect updates to `final_res` and `final_path`, using `#pragma omp critical`. Moreover, each thread operated on its own local copies of recursive variables to avoid shared memory conflicts. This version produced correct and stable results and was used for the final performance measurements and scalability tests.

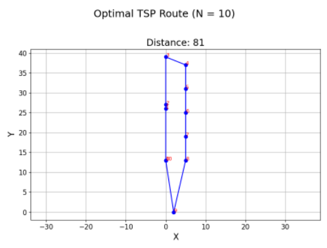
All versions were validated through controlled experiments. In particular, correctness was verified by comparing outputs with the brute-force method on small instances and by checking symmetry and cost consistency on larger inputs. The benchmark datasets from [**?**] were used for all evaluations.

3.4 Visualization of Optimal Routes

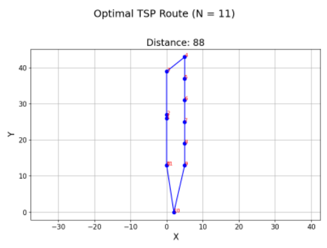
The optimal routes found by this parallel algorithm are presented below (see Fig. 3.1).



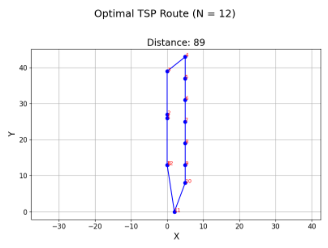
(a) Route for N = 9



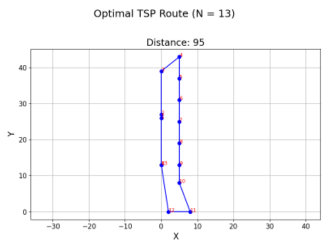
(b) Route for N = 10



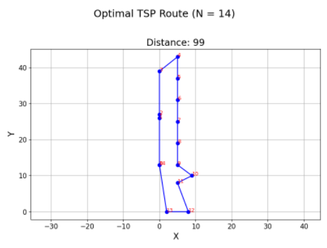
(c) Route for N = 11



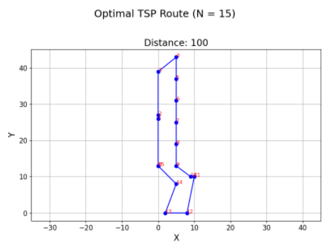
(d) Route for N = 12



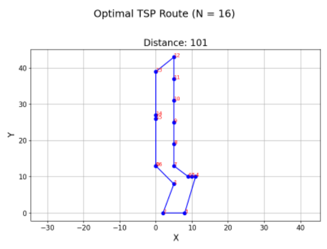
(e) Route for N = 13



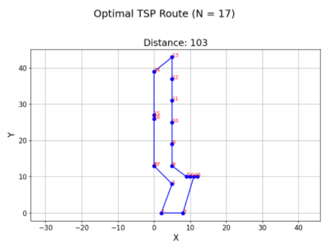
(f) Route for N = 14



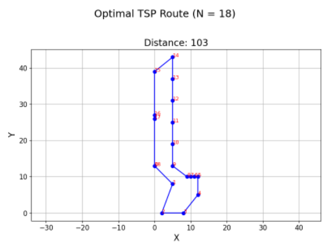
(g) Route for N = 15



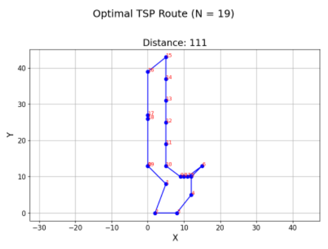
(h) Route for N = 16



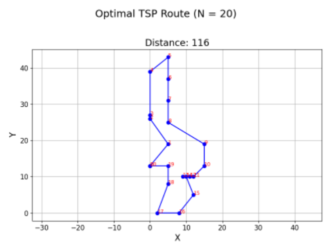
(i) Route for N = 17



(j) Route for N = 18



(k) Route for N = 19



(l) Route for N = 20

Figure 3.1: Optimal TSP routes obtained for problem sizes from $N = 9$ to $N = 20$ using the parallel Branch and Bound algorithm.

To assess how well the algorithm scales, we started with a common base dataset and gradually added more cities to create larger problem scenarios.

This approach ensured that each increase in size preserved the original data's distribution and integrity.

3.5 Time Analysis and Theoretical Comparison

To assess the performance of our implementation, we measured execution times across a range of input sizes. To evaluate our parallel implementation, we present execution times for thread counts ranging from 1 to 16. While tests were conducted using up to 32 threads, we found that performance improvements beyond 16 threads were minimal for the selected problem sizes. This phenomenon is likely attributed to the relatively small size of the problems, where the overhead associated with thread management and synchronization surpasses the advantages of increased parallelism at higher thread counts.

The results are presented in the figures below.

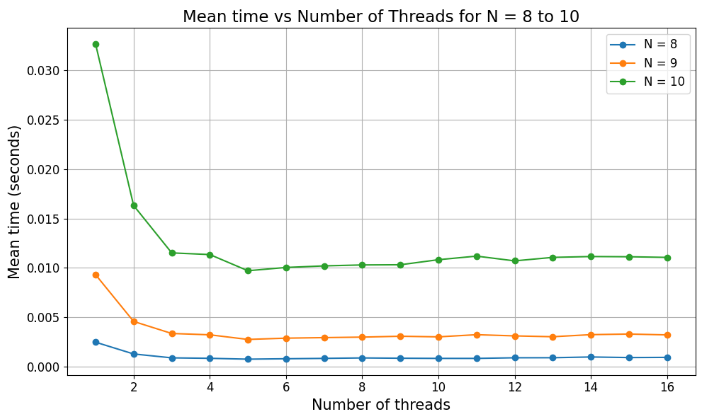


Figure 3.2: Execution times for $N = 8$ to $N = 10$ using the parallel Branch and Bound algorithm.

The results shown in Fig. 3.2 indicate that, for the largest problem size, a reduction in execution time is observed when employing up to three threads. In contrast, the smaller problem sizes($N = 8, 9$) show a decrease in execution time when utilizing up to two threads. It is important to note, however, that the three plotted results do not exceed half of a second; this suggests that the reductions in execution time are relatively modest.

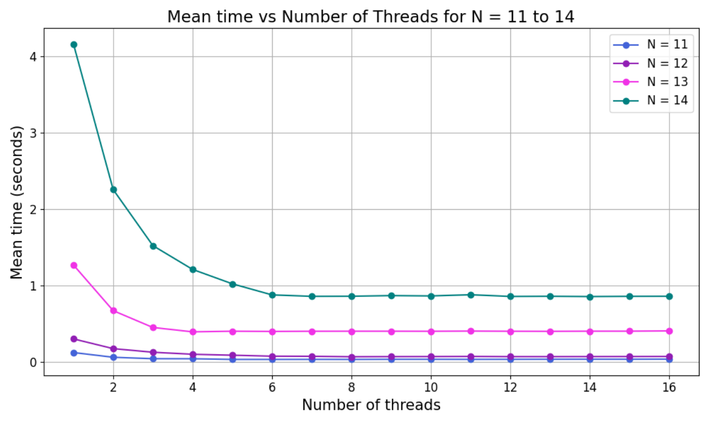


Figure 3.3: Execution times for $N = 11$ to $N = 14$ using the parallel Branch and Bound algorithm.

According to Fig. 3.3, the execution time for $N = 14$ shows a significant reduction as the number of threads is increased to six. In contrast, for the other three problem sizes ($N = 11, 12, 13$), there is no evidence of reduced execution time when utilizing more than two or three threads. Additionally, the time measured in seconds for this graph is longer than that of the previous one, making the observed reduction in execution time for $N = 14$ even more clear.

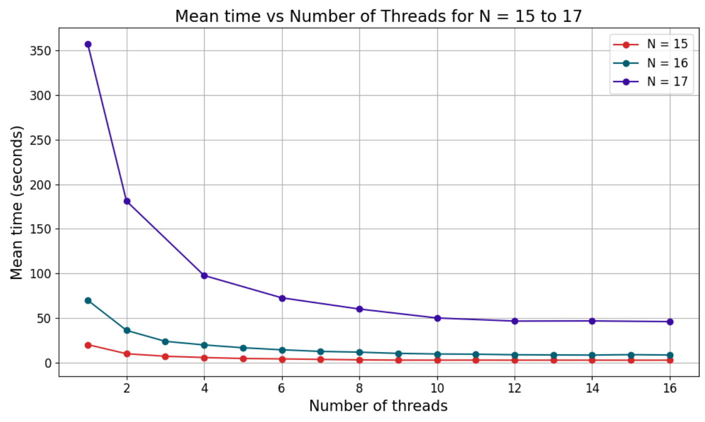


Figure 3.4: Execution times for $N = 15$ to $N = 17$ using the parallel Branch and Bound algorithm.

The execution times illustrated in Fig. 3.4 show that, for the largest problem size ($N = 17$), utilizing up to six threads results in a significant enhancement in performance. In contrast, the smaller problem sizes ($N = 15, 16$) present a decline: as the number of thread increases, the decrement in time execution becomes less visible.

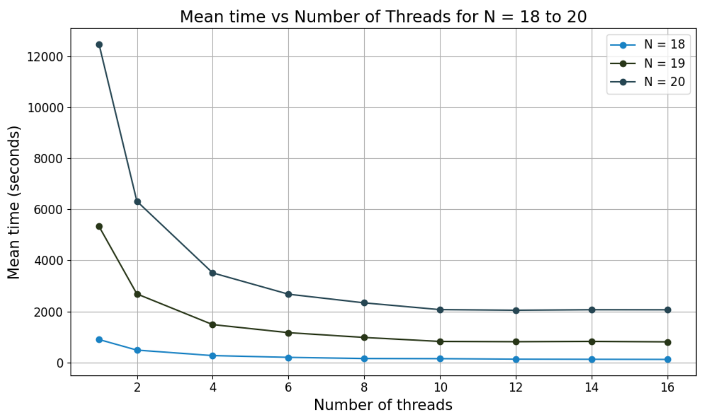


Figure 3.5: Execution times for $N = 18$ to $N = 20$ using the parallel Branch and Bound algorithm.

Figure 3.5 shows that the execution time for the largest problem size ($N = 20$) drops considerably when using up to six threads. Meanwhile, the reduction in execution time for $N = 18$ and $N = 19$ becomes less pronounced.

In Figs. 3.2 to 3.5, we provide the average execution times, derived from three separate runs for each input size, using VLSI datasets from [5]. All tests were conducted on the Khipu supercomputer at UTEC, ensuring a consistent computational environment throughout the evaluation.
For smaller problem sizes, there is no significant improvement in parallel execution time. However, as the problem size increases, the parallel execution time shows a marked decrease. This trend is evident across all plots with varying thread counts. One notable observation is that the most substantial reduction in execution times is seen in the last two plots when the number of threads is increased from 2 to 6.

The graphs indicate that although the system has the potential to handle more threads, the parallelization based on the Parallel Window Search for the TSP branch-and-bound does not consistently enhance parallel execution time. The performance improvement when using 16 threads compared to 8 threads is not significantly noticeable for the problem sizes we tested. However, for larger problem sizes, increasing the number of threads could yield better performance. We did not assess larger problem instances due to the high sequential complexity of the algorithm. Although employing Branch and Bound reduces the growth from exponential to polynomial, the execution time still increases rapidly as the number of cities increases.

3.6 Speedup Evaluation

To evaluate the parallel performance of the TSP branch-and-bound algorithm, we examined the speedup obtained when increasing the number of threads. The goal is to determine the optimal number of threads needed to attain the ideal speedup, which is outlined as follows:

$$\text{Speedup}_{\text{ideal}} = \frac{T_s}{T_p} = p \tag{3.1}$$

where T_s or T_1 is the execution time of the sequential algorithm that uses only one thread, T_p denotes the execution time using p threads, and p indicates the total number of threads employed. In the ideal case, doubling the number of threads results in halving the execution time, which leads to the expected linear speedup.

The speedup plots for each problem size are shown below.

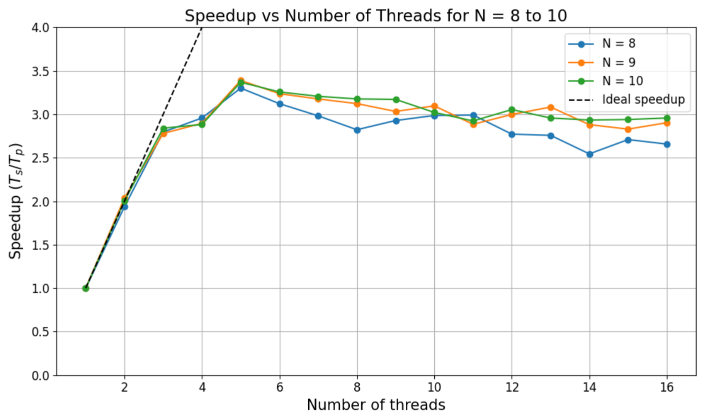


Figure 3.6: Speedup achieved for $N = 8$ to $N = 10$ using the parallel Branch and Bound algorithm.

In Fig. 3.6, the achieved speedup approaches the ideal speedup (see Eq. (3.1)) with a maximum of three threads. Beyond this point, there is no improvement in performance, and in some cases, even declines.

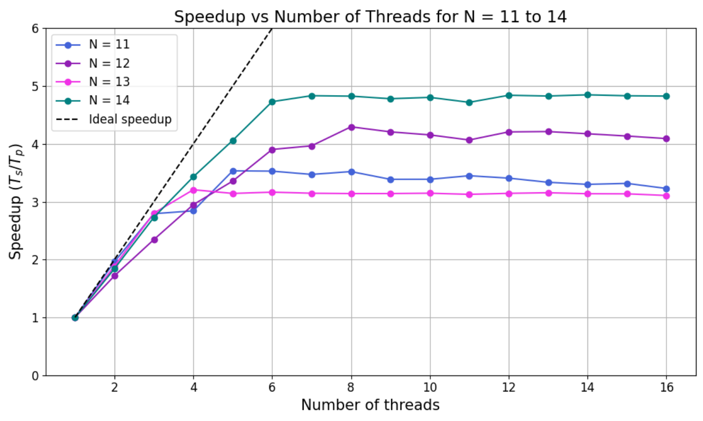


Figure 3.7: Speedup achieved for $N = 11$ to $N = 14$ using the parallel Branch and Bound algorithm.

As illustrated in Fig. 3.7, the observed speedup closely aligns with the ideal speedup curve (see Eq. (3.1)) up to four threads, while the speedup achieved by the algorithm continues to increase when utilizing up to ten threads; however, the increment rate is slower than the expected one.

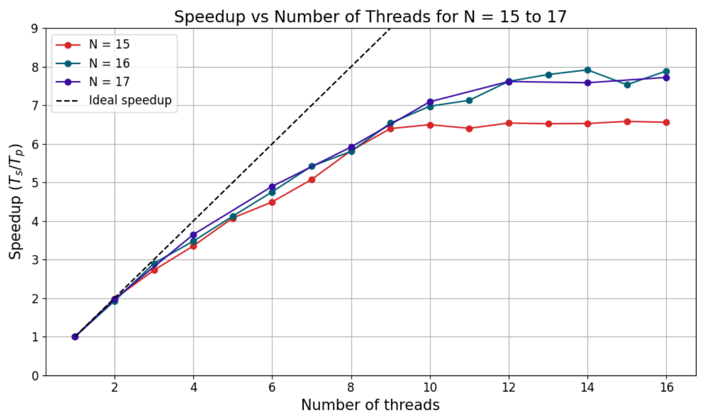


Figure 3.8: Speedup achieved for $N = 15$ to $N = 17$ using the parallel Branch and Bound algorithm.

As observed in Fig. 3.7, the problem sizes shown in Figs. 3.8 and 3.9 reach their maximum speedup, closely aligning with the ideal up to six threads. Additionally, in both graphs, the speedup continues to increase up to ten threads, but at a slower rate.

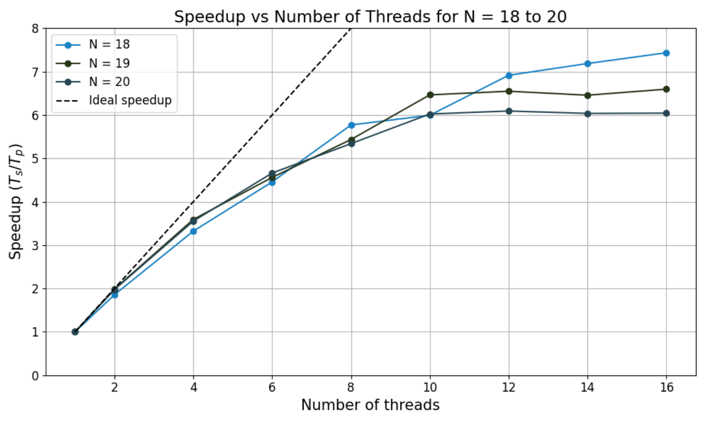


Figure 3.9: Speedup achieved for $N = 18$ to $N = 20$ using the parallel Branch and Bound algorithm.

The plots illustrated in Figs. 3.6 to 3.9 provide additional insight into the efficiency of the parallelization. For smaller problem sizes, the speedup remains limited, with only slight improvements from using multiple threads. As the problem size increases, the speedup curves begin to approach the ideal, particularly up to six threads. This trend suggests that larger problem instances benefit more from parallelism, due to the greater computational work to distribute. Nevertheless, once the thread count surpasses six to eight, the gains in speedup begin to diminish, and in some instances, there may be a slight stagnation or decline, suggesting bottlenecks due to synchronization, load imbalance, or the intrinsic limitations of the Parallel Window Search strategy for TSP. Furthermore, the recursive nature of the Branch and Bound algorithm for solving the TSP may also restrict scalability, as deep recursive calls can hinder efficient thread utilization and limit opportunities for concurrent execution.

3.7 Efficiency and Scalability Evaluation

To effectively evaluate the parallel performance of the TSP branch-and-bound algorithm, we also evaluated its parallel efficiency, which is outlined as follows:

$$\text{Efficiency} = \frac{T_s}{T_p \times p} = \frac{Speedup}{p} \tag{3.2}$$

where T_s, T_p, and p are as defined previously in Eq. (3.1).

Given that achieving the ideal efficiency (Efficiency $= O(1)$) is inherently challenging, we will establish the following as a bound for our consideration:

$$\text{Efficiency}_{\text{expected}} = 0.8 \tag{3.3}$$

This value will serve as a benchmark for evaluating our performance going forward.
The following figures illustrate the efficiency observed for different problem sizes and thread counts.

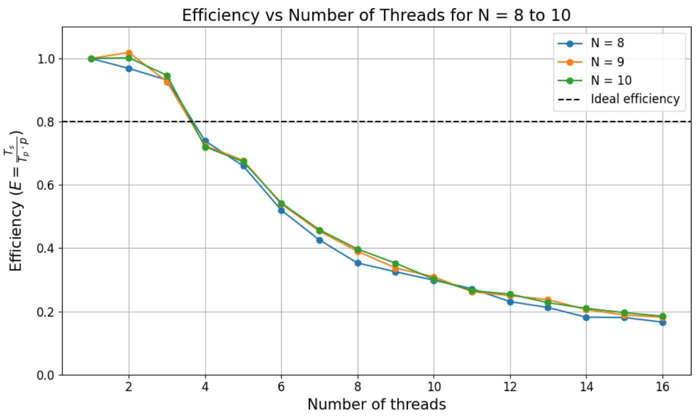


Figure 3.10: Parallel efficiency for $N = 8$ to $N = 10$ using the parallel Branch and Bound algorithm.

As depicted in Fig. 3.10, the efficiency obtained with up to three threads remains above 0.8; beyond that point, the efficiency decreases abruptly, reaching a value of 0.2 when employing sixteen threads. It is noteworthy that as the problem size ranges from $N = 8$ to $N = 10$, the improvement in the efficiency when using multiple threads is not noticeable.

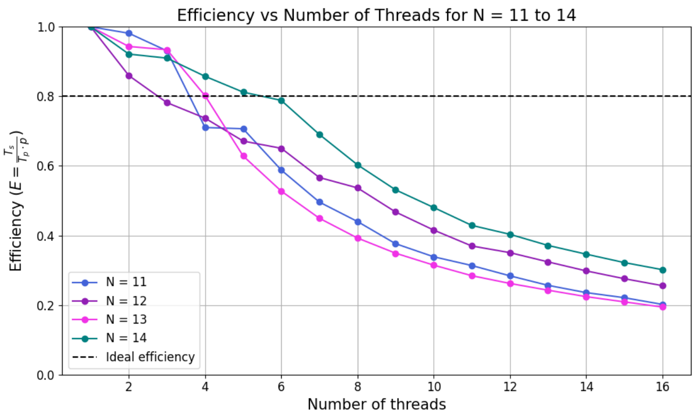


Figure 3.11: Parallel efficiency for $N = 11$ to $N = 14$ using the parallel Branch and Bound algorithm.

In Fig. 3.11, the efficiency achieved for the larger problem size ($N = 14$) is nearly close to the ideal efficiency (see Eq. (3.3)). Meanwhile, with a problem size of $N = 13$, the efficiency remains substantial using up to three threads. Oppositely, for the remaining problem sizes, the efficiency stays above 0.8 with a maximum of three threads, and in some cases, only two. These findings suggest that as the problem size increases, the efficiency obtained consistently exceeds the expected one.

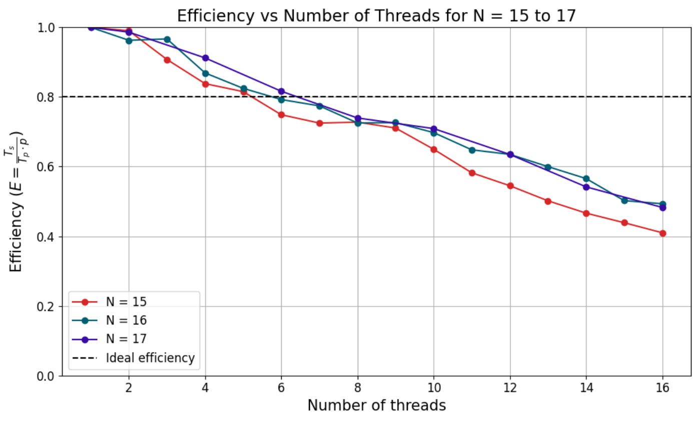


Figure 3.12: Parallel efficiency for $N = 15$ to $N = 17$ using the parallel Branch and Bound algorithm.

As illustrated in Fig. 3.11, the efficiency shown in both Figs. 3.12 and 3.13 reaches nearly the expected value when utilizing up to six threads.However, beyond six threads, the efficiency declines, ultimately reaching a value of 0.4.

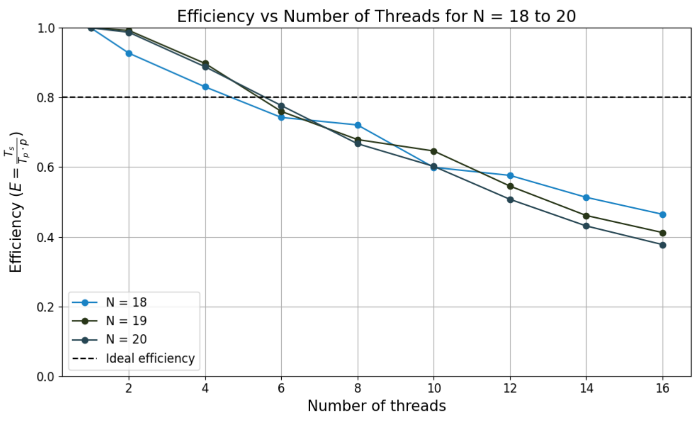


Figure 3.13: Parallel efficiency for $N = 18$ to $N = 20$ using the parallel Branch and Bound algorithm.

The plots presented in Figs. 3.10 to 3.13 provide valuable insights into the scalability and efficiency of the parallel TSP with the branch-and-bound approach. For smaller problem sizes ($N = 8$ to $N = 10$), the efficiency rapidly declines beyond three threads, with limited improvement from adding more computational resources. As the problem size increases, the efficiency consistently approaches or even exceeds the expected value of 0.8 (see Eq. (3.3)) when using up to six threads, especially for $N \geq 14$. This suggests that larger problem instances are better suited for parallel execution, as they provide sufficient workload to maintain thread utilization effectively. However, as the number of threads continues to grow beyond this point, a notable drop in efficiency occurs, revealing the impact of synchronization overhead. The recursive nature of the algorithm may restrict optimal thread distribution and hinder the exploitation of full parallel potential.

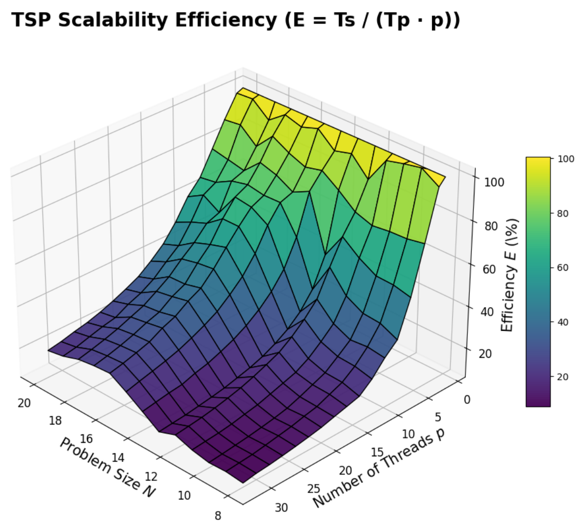


Figure 3.14: Scalability efficiency surface for the TSP using the parallel Branch and Bound algorithm.

Figure 3.14 presents a 3D surface that illustrates the changes in efficiency for different problem sizes and thread counts. The graph shows that efficiency remains high for larger problems when using a moderate number of threads (around six). Beyond that point, efficiency tends to decline, especially for smaller problems where the overhead of managing multiple threads outweighs the benefits of parallelism. In summary, the plot effectively outlines the trends observed in earlier figures and reinforces the idea that improved efficiency is achieved when the workload is large enough and can be evenly distributed across multiple threads.

3.8 FLOPs Analysis

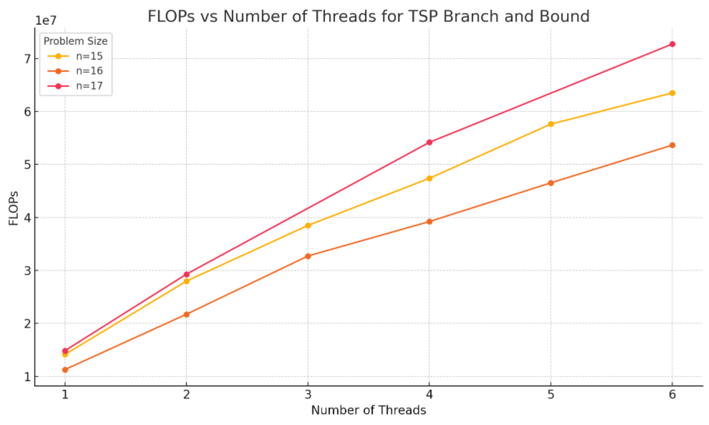


Figure 3.15: FLOPs for $N = 15$ to $N = 17$ using the parallel Branch and Bound algorithm.

The graph presented in Fig. 3.15 shows the number of floating point operations per second (FLOPs) obtained for three different problem sizes ($N = 15$, $N = 16$, and $N = 17$), as the number of threads increases from 1 to 6. To obtain these results, we measured the average execution time over three independent runs for each configuration, and then computed:

$$\text{FLOPs} = \frac{\text{Total Floating Point Operations}}{\text{Average Execution Time (in seconds)}} \tag{3.4}$$

The total number of floating point operations was estimated from instrumentation of key operations inside the recursive Branch and Bound process, including distance calculations, bound updates, and pruning decisions.
As shown in the figure, the FLOPs increase significantly as the number of threads increases, especially for the larger problem size $N = 17$. This indicates an effective utilization of computational resources when parallelism is introduced. For smaller problem sizes ($N = 15$ and $N = 16$), the FLOPs also increase with the number of threads, but the slope is less steep. This suggests that, while parallelism helps, the benefit is less dramatic for problems with smaller search trees.
In particular, the curve for $N = 17$ shows a steep increase up to 6 threads, reflecting a significant gain in performance due to deeper search trees and higher computational load, which allows threads to work more independently and efficiently.

Overall, this analysis confirms that parallelism improves the computational throughput of the algorithm, and that FLOPs is a useful metric to assess how well the algorithm scales with respect to both problem size and thread count.

Chapter 4

RESULTS AND ANALYSIS

4.1 Critical Discussion

The experimental results demonstrate that the parallel Branch and Bound algorithm significantly reduces execution time compared to its sequential counterpart, particularly for problem sizes beyond $N = 14$. However, a detailed analysis of parallel efficiency reveals that performance gains diminish as the number of threads exceeds six.

This phenomenon is primarily explained by the *overhead associated with thread management and synchronization mechanisms*. While the CREW PRAM model ensures thread-safe operations, the use of `#pragma omp critical` sections to update shared variables (`final_res` and `final_path`) introduces serialization points that limit scalability. As more threads are introduced, the probability of concurrent access to these shared resources increases, leading to contention and idle time, which reduces overall efficiency.

Moreover, the nature of the problem imposes a significant *computational imbalance*: not all subtrees in the search space are equally complex. Some threads may complete early while others explore deeper subtrees, resulting in *load imbalance* and poor resource utilization. Additionally, the recursive structure of the algorithm limits parallelism beyond the top levels of the search tree, as the inner recursive calls are executed serially within each thread.

These combined effects explain why, despite the theoretical benefits of parallelization, the practical efficiency plateaus and even degrades beyond 6–8 threads.

4.2 Algorithm Limitations

Despite its improvements over brute-force approaches, the parallel Branch and Bound algorithm has several inherent limitations:

- **Execution Time:** The algorithm still has exponential time complexity in the worst case. For instances beyond $N = 20$, the required computation grows rapidly, even with pruning strategies.

- **Memory Consumption:** The algorithm maintains multiple recursive stacks, state vectors, and distance matrices in memory. When executed with many threads, this can lead to high memory usage, especially for large problem sizes.
- **Scalability Constraints:** The recursive nature of Branch and Bound inhibits deep parallelism. Once the subtrees are distributed at the top level, each thread explores its assigned subtree serially, limiting fine-grained parallel execution.
- **Synchronization Overhead:** Shared data updates introduce critical sections, which become bottlenecks with increasing thread counts.
- **Limited Parallel Granularity:** As the number of cities increases, deeper levels of the search tree offer more parallelizable work, but the current implementation only exploits parallelism up to the first decision level.

4.3 Possible Improvements

To enhance the performance and scalability of the algorithm, several improvements can be considered:

- **Deeper Parallelization:** Instead of distributing only the first level of subtrees, parallel recursion could be implemented at deeper levels using task-based parallelism (e.g., `OpenMP tasks` or thread pools), allowing dynamic load balancing.
- **Use of GPUs:** Offloading part of the computation (e.g., bound calculations or distance matrix generation) to GPUs could dramatically accelerate the algorithm, leveraging their massive parallelism.
- **Hybrid Parallel Models:** Combining shared-memory parallelism (OpenMP) with distributed-memory approaches (MPI) could allow the algorithm to scale across multiple nodes in a high-performance computing cluster.
- **Heuristics or Metaheuristics Integration:** Incorporating approximation methods (e.g., nearest neighbor, genetic algorithms, simulated annealing) as preprocessing steps can provide tighter initial bounds, thus improving pruning efficiency.

- **Memory Optimization:** Implementing memory pooling and reducing redundant data structures could help scale the algorithm for very large problem instances.

4.4 Conclusions

In this study, we have designed, implemented, and evaluated both sequential and parallel versions of the Branch and Bound algorithm for solving the Traveling Salesman Problem. The results demonstrate the effectiveness of parallelization in reducing execution times and increasing computational throughput, especially for instances with $N \geq 14$.
The use of the CREW PRAM model and OpenMP enabled scalable parallel execution up to a moderate number of threads. However, efficiency analysis revealed diminishing returns beyond 6–8 threads, mainly due to synchronization overhead, load imbalance, and limited recursive parallelism.
Despite these limitations, the algorithm proved capable of solving TSP instances up to $N = 20$ in reasonable time frames, highlighting its practical utility when combined with parallel computing. Future improvements should focus on exploiting deeper parallelism, reducing synchronization costs, and integrating GPU acceleration or hybrid strategies.
This project showcases the potential of parallel algorithms in addressing combinatorial optimization problems and underscores the importance of thoughtful design choices in balancing theoretical scalability with real-world performance.

The complete source code, dataset samples, and plotting scripts used in this project are publicly available at the following repository:

`https://github.com/Mycodeiskuina/TSP`

BIBLIOGRAPHY

[1] Karla L Hoffman, Manfred Padberg, Giovanni Rinaldi, et al. Traveling salesman problem. *Encyclopedia of operations research and management science*, 1:1573–1578, 2013.

[2] Michael Jünger, Gerhard Reinelt, y Giovanni Rinaldi. The traveling salesman problem. *Handbooks in operations research and management science*, 7:225–330, 1995.

[3] Rajesh Matai, Surya Prakash Singh, y Murari Lal Mittal. Traveling salesman problem: an overview of applications, formulations, and solution approaches. *Traveling salesman problem, theory and applications*, 1(1): 1–25, 2010.

[4] GeeksforGeeks. Traveling salesman problem using branch and bound, April 2023. URL `https://www.geeksforgeeks.org/dsa/traveling--salesman-problem-using-branch-and-bound-2/`.

[5] University of Waterloo. Vlsi data sets, May 2013. URL `https://www.math.uwaterloo.ca/tsp/vlsi/index.html`.