The Wandering Salesman Problem

High Performance Technical Computing

Paul Fontanges; Stu No. : 461720

Prepared for 10 February , 2025

Paul Fontanges
Student Number : 461720

Abstract

This report explores the application of distributed-memory parallel programming techniques to solve the Wandering Salesman Problem (WSP), a combinatorial optimization challenge that requires visiting a set of cities exactly once without returning to the starting point. The study first implements a sequential Branch-and-Bound algorithm, leveraging pruning techniques to reduce computational overhead. It then introduces a parallelized version using MPI (Message Passing Interface), focusing on task distribution, load balancing, and communication optimization. The results highlight significant performance improvements through dynamic subproblem distribution and show how the efficiency of parallelization depends on problem size and task granularity. Additionally, the report discusses the environmental impact of high-performance computing (HPC), emphasizing sustainability challenges and the potential of HPC to drive innovations in areas like climate modeling and energy optimization. This comprehensive study underscores the trade-offs between computational efficiency and environmental impact, providing insights into the scalability and sustainability of HPC solutions.

Paul Fontanges
Student Number : 461720

Contents

List of Figures

1 Introduction

The assignment focuses on the application of distributed-memory parallel programming techniques to solve a combinatorial optimization problem: the Wandering Salesman Problem (WSP). Unlike the classic Traveling Salesman Problem, the WSP requires a traveler to visit a given set of cities exactly once without returning to the starting point. This problem is computationally challenging, as the number of possible routes grows factorially with the number of cities, making it infeasible to solve using brute force for large inputs.

The primary goal of this project is to design and implement efficient algorithms to solve the WSP, initially through a sequential Branch-and-Bound approach and subsequently through a parallelized version using MPI (Message Passing Interface). The Branch-and-Bound method leverages intelligent pruning to reduce the number of computations required by eliminating paths that cannot improve the current best solution.

The broader context of this assignment is situated within high-performance computing (HPC), where solving large-scale combinatorial problems requires scalable parallelization techniques. By exploring load balancing, communication patterns, and performance metrics, this project aims to address critical challenges in distributed-memory parallel programming. Additionally, the environmental impact of HPC systems, inspired by discussions on platforms like Green500 and LUMI supercomputers, will be analyzed to understand the sustainability of such computational approaches.

This assignment will deliver a comprehensive solution that encompasses algorithm development, implementation, and performance analysis while reflecting on the role of HPC in effectively solving optimization problems.

2 Methodology

2.1 Experimental Design

2.1.1 Sequential Algorithm for the Wandering Salesman Problem (WSP)

The Wandering Salesman Problem (WSP) is a combinatorial optimization problem that consists of determining the shortest path for a traveler to visit a list of cities exactly once, without returning to the starting city. Given the factorial complexity of this problem ($N!$ possible routes for N cities), an exhaustive solution quickly becomes impractical. The chosen approach to solve this problem is the Branch-and-Bound method, which significantly reduces the search space by pruning non-promising paths. Here, we will take city 1 (or city 0, depending on the index in the input file "city") as the starting city.

2.1.2 Steps of the Sequential Methodology

Reading Input Data: The program begins by reading a file containing:

- The number of cities N.
- The distances between pairs of cities are provided as a lower triangular matrix.

These data are transformed into a symmetric matrix $d[N][N]$, where $d[i][j]$ represents the distance between cities i and j.

Exploration of Paths Using Branch-and-Bound: The algorithm recursively explores possible paths while adhering to the following constraints:

- **Pruning of Non-Promising Branches:** If the cumulative distance of a current path ($current\_distance$) exceeds the current best distance ($best\_distance$), the branch is abandoned.
- **Memorization of Optimal Solutions:** When a complete path is found, its distance is compared with $best\_distance$. If this path is shorter, $best\_distance$ and $best\_path$ are updated.
- **Backtracking:** After each attempt, the visited cities are reset to allow the exploration of other combinations.

Global Resolution Process: The algorithm starts by testing each city as a starting point. For every second city visited i:

- The city is marked as visited.
- A recursive function explores all possible paths starting from this city using the Branch-and-Bound approach.
- Once all possibilities for a starting city have been explored, the visit markers are reset.

2.1.3 Parallelization of the Algorithm

To parallelize this algorithm, we can take advantage of the fact that each starting city represents an independent sub-problem. Since the goal is to explore all possible combinations of paths, an efficient approach would be to group the computations by the starting city.

The problem can be decomposed as follows.

- The first group consists of all the paths starting from city 2 (or city 1, depending on the index in the input file "city").
- The second group includes all paths starting from city 3 (or city 2, depending on the index in the input file "city").
- And so on, until the last city.

Each group is then assigned to a processor, depending on the total number of cities (N) and the number of processors available. Each processor independently processes the paths in its assigned group using the sequential algorithm (Branch-and-Bound).

Once all processors complete their computations, they send their best distance ($best\_distance$) and the corresponding path to the master processor. The master processor compares the results of all processors to determine the global minimum distance and the associated path.Here is an example Figure 1.

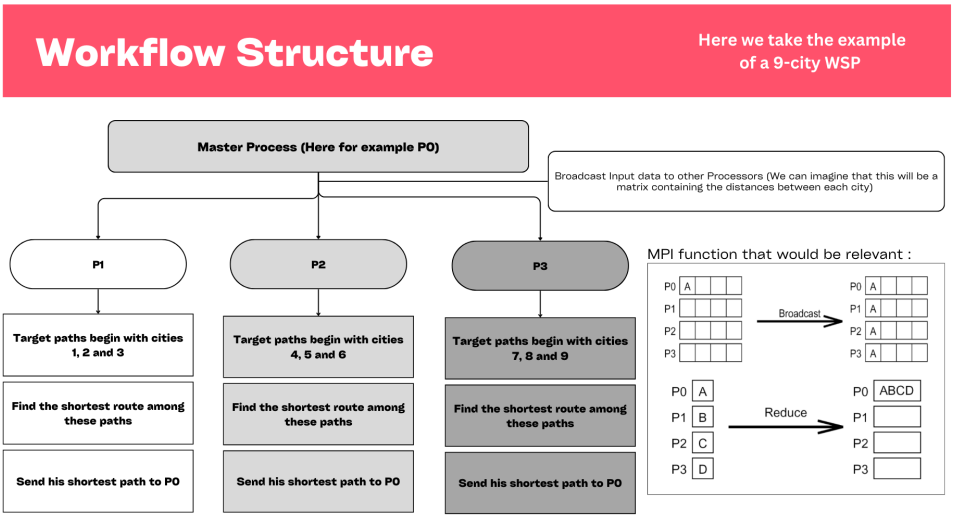

Figure 1: Parallelization Workflow

2.1.4 Advantages of this Parallelization Approach

1. **Independence of Subproblems:** Each starting city corresponds to a distinct subproblem, eliminating the need for synchronization during computations. Processors can work in parallel without interference.

2. **Load Balancing:** The number of groups to process is equal to the number of cities minus one $(N-1)$, ensuring a fair distribution of tasks among processors, especially if the number of processors (P) divides $N-1$ evenly.

3. **Scalability:** This approach is scalable for large problems, as increasing the number of cities or processors does not require modifications to the core algorithm.

2.2 Implementation

2.2.1 Code Structure

The program is structured around four main functions, each with a specific role in solving the Wandering Salesman Problem (WSP):

2.2.2 Initialization of the Search:

The variable *best_distance* is initialized to an infinite value (∞) to store the best distance found. An array *best_path* is used to store the optimal path.

2.2.3 `load_input` Function :

- Reads the input file containing:
  - The number of cities (N).
  - The distances between city pairs in an upper triangular format.
- Constructs a symmetric distance matrix $distances[N][N]$ where $distances[i][j]$ represents the distance between cities i and j. Symmetry is ensured by copying $distances[i][j]$ to $distances[j][i]$.
- Initializes the diagonal ($distances[i][i]$) to 0, as the distance from a city to itself is 0.

Let us consider the following input file example:

```
4
 54
 76 30
 24 51 64
```

The resulting distance matrix is:

$$\begin{bmatrix} 0 & 54 & 76 & 24 \\ 54 & 0 & 30 & 51 \\ 76 & 30 & 0 & 64 \\ 24 & 51 & 64 & 0 \end{bmatrix}$$

2.2.4 `branch_and_bound` Function :

- Implements the recursive Branch-and-Bound algorithm to explore all possible paths:
  - **Pruning:** Branches where the cumulative distance (`current_distance`) exceeds the best local distance (`local_best_distance`) are abandoned.

  - **Updating Local Results:** When a complete path is found (`level == N`) and is shorter than the current local best, `local_best_distance` and `local_best_path` are updated.
  - **Backtracking:** After exploring a path, the visited cities are reset to allow exploration of other combinations.

- Key parameters:
  - `level`: Depth of exploration (number of cities visited so far).
  - `visited[]`: Array marking the cities that have already been visited.
  - `current_distance`: The current distance being explored.
  - `local_min_distance`: The best distance available at this moment in this processor.

2.2.5 `solve_wsp` Function :

- Orchestrates the parallel execution of the WSP solution using MPI.
- Key steps:
  1. **Data Broadcasting:** The master process (`myrank = 0`) reads the input via `load_input`, and the number of cities (N) and distance matrix are broadcast to all processes using `MPI_Bcast`.
  2. **Task Distribution:** The cities are divided into subsets:
  $$cities\_per\_process = \frac{N}{P}$$
  Each process works on the cities within its assigned range [`start`, `end`).
  3. **Local Computation:** Each process executes `branch_and_bound` on its subset of cities and updates its `local_best_distance` and `local_best_path`.
  4. **Reduction and Collection:** Local best distances are aggregated via `MPI_Allreduce`.
  5. **Identifying the Global Best Path:** The master process identifies the path corresponding to the global minimum distance.

2.2.6 `main` Function :

- Initializes MPI with `MPI_Init`.
- Parses command-line arguments to specify the input file.
- Calls `solve_wsp` to solve the WSP.
- Finalizes MPI with `MPI_Finalize`.

2.2.7 Parallelization Management

The parallelization is based on distributing starting cities among MPI processes, ensuring independent execution for each subset of cities:

- **Independent Local Computations:** Each process executes `branch_and_bound` on its subset of cities without communication during the computation phase, minimizing communication overhead.
- **Load Balancing:** The task distribution ensures a nearly equal workload among processes, calculated as:

$$cities\_per\_process = \frac{N}{P}$$

2.2.8 Finding the Best Path

- **Reducing Distances:** Each process sends its `local_best_distance` via `MPI_Reduce`, and the master process identifies the global minimum (`global_best_distance`).
- **Matching the Path:** Once the global minimum distance is determined, the master process gathers all the local paths using `MPI_Bcast` and identifies the path corresponding to the global minimum distance.

2.2.9 Communication Between Processes

MPI functions are used extensively for coordinating and optimizing communication between processes:

- `MPI_Bcast`**:** Used to broadcast the number of cities and the distance matrix to all processes, ensuring that each process has consistent input data.
- `MPI_AllReduce`**:** Collects the local minimum distances from all processes and reduces them to determine the global minimum distance.
- `MPI_Bcast`**(again):** After determining the global best path, it is broadcasted to all processes so that every process has the best path information.h.

2.3 Improvement through Dynamic Subproblem Distribution

To improve the processing time, we introduce an additional approach aimed at dynamically generating subproblems and efficiently distributing them among processes. Instead of directly calculating all paths from each city as in the previous approach, we divide the paths into fixed-size sub-segments, which allows for more granular task management and optimized parallelization.

The idea here is to break down the entire problem into subproblems of depth n, which corresponds to subpaths of length n, and distribute them among the different processes. This reduces the size of each subproblem and better utilizes processors to calculate different independent parts of the path.

2.3.1 Generation of Depth n Subproblems:

In this method, each subproblem consists of a partial path of length n, meaning we generate all possible combinations of n distinct cities from the N cities. These subproblems are then organized into a list, with each entry corresponding to a subproblem with a partial path and the cities already visited.

2.3.2 Task Distribution and Calculation:

Once the subproblems are generated, the work is distributed among the different processes using the `MPI_Bcast` function. Each process handles a portion of the subproblems depending on its rank in the MPI process ordering. Each task involves exploring a partial path of length n, and the process uses the `branch_and_bound` algorithm to calculate the minimum possible distance for that subproblem.

2.3.3 Advantages of this Approach:

- **Reduction in the Size of Subproblems:** By dividing the problem into depth-n subproblems, we significantly reduce the size of each individual task, enabling each process to work more efficiently.

3 Results

3.1 Optimal Path for the Wandering Salesman Problem

Figure 2 represents the optimal solution found for the Traveling Salesman Problem using the `branch_and_bound` method with (and without) dynamic subproblem distribution. Each city is identified by a number, and the optimal path is plotted using red segments connecting the cities. The arrows indicate the direction of the journey.

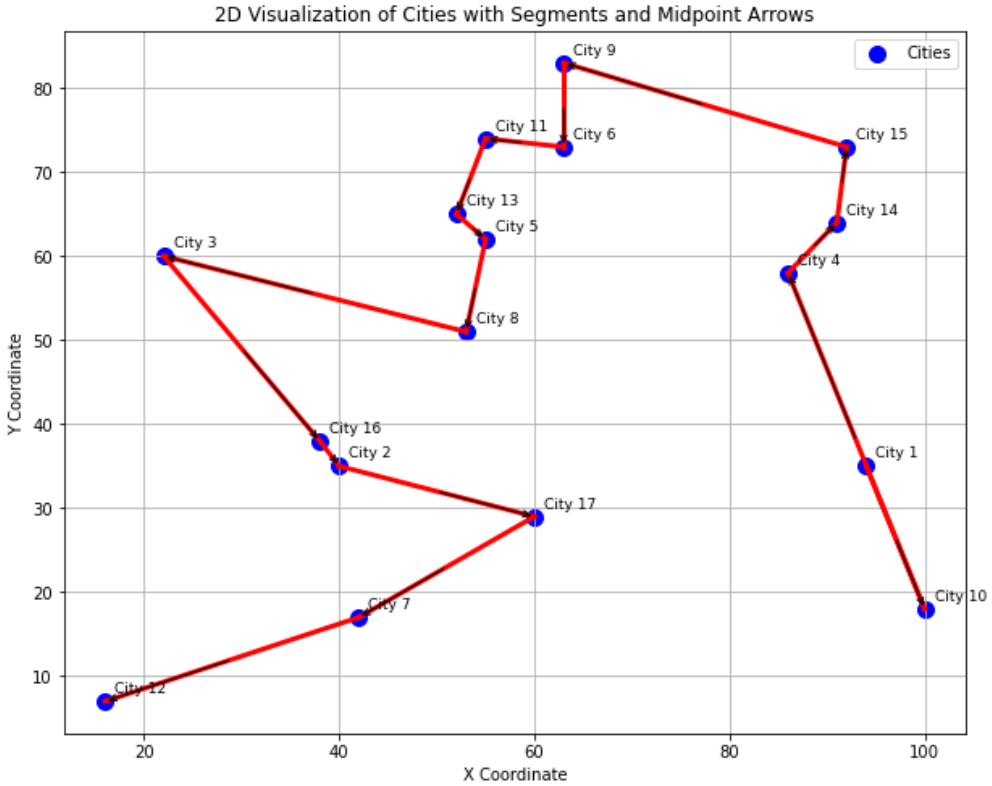

Figure 2: Optimal path for the Wandering Salesman Problem starting from city 1.

This path minimizes the total distance while ensuring that each city is visited exactly once. The observed path respects the constraints of the problem while optimizing the traveled distance.

3.2 Performance Analysis: Speed-up

Figure 3 shows the speed-up achieved as a function of the number of processors for different subproblem distribution methods.

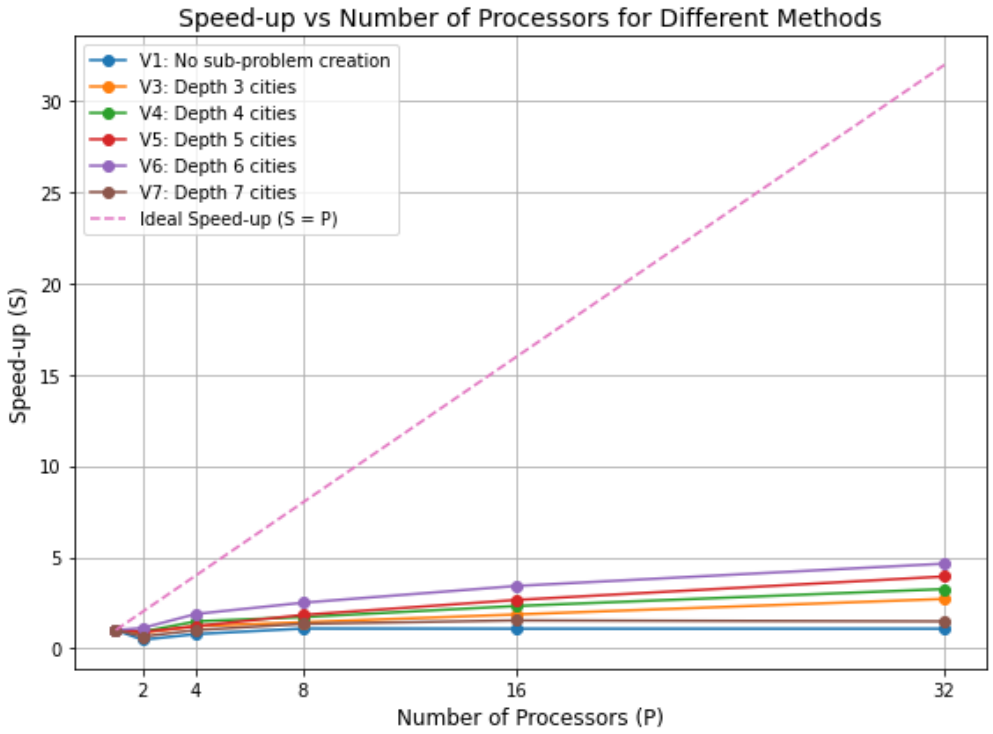

Figure 3: Speed-up as a function of the number of processors and the different distribution methods.

Key observations:

- The **no subproblem creation** method (V1) exhibits the lowest speed-up, even as the number of processors increases.
- Methods with subproblems of increasing depth (V3 to V7) show progressive improvement in speed-up, particularly for depths greater than 4 (V4 to V7).

- The ideal linear speed-up is partially achieved by the most efficient methods (V6 and V7), demonstrating good parallelization.

3.3 Execution Time Analysis

Figure 4 illustrates the execution time as a function of the number of processors for different methods.

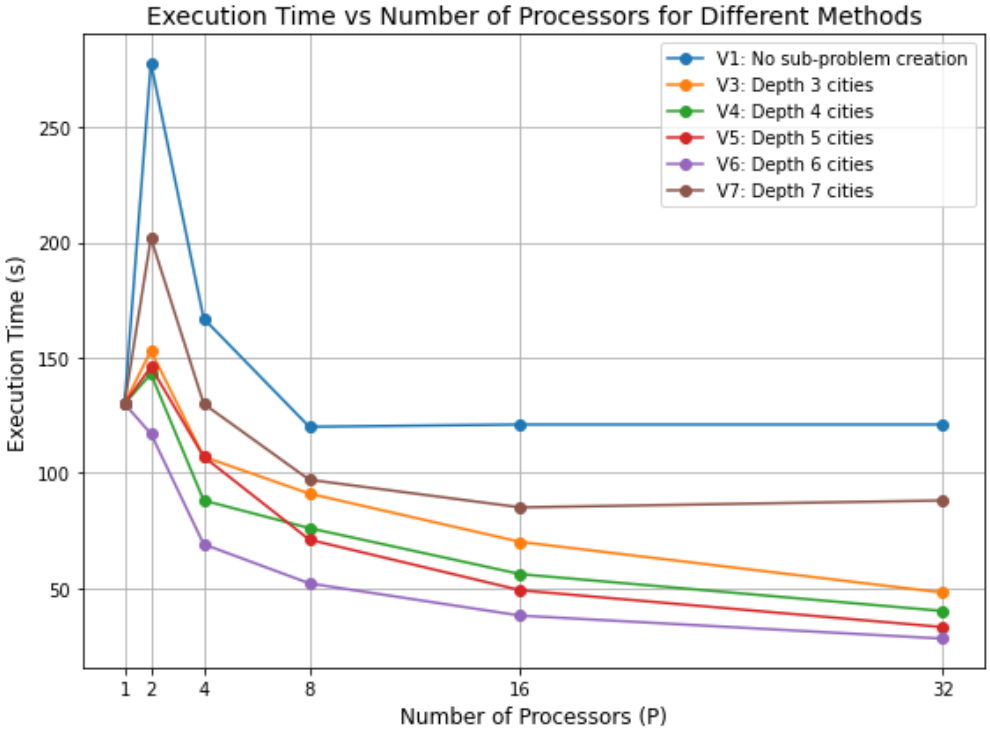

Figure 4: Execution time as a function of the number of processors and the different methods.

Key insights:

- The V1 method, without subproblem creation, has a significantly high execution time, particularly for a small number of processors.
- Methods with subproblems (V3 to V7) drastically reduce the execution time, especially with optimized subproblem depths (V5 to V7).

- Execution time decreases as the number of processors increases, showing good scalability.
- However, for a very large number of processors (e.g., 32), performance gains diminish, likely due to increased communication overhead between processors.

3.4 Discussion of Results

The results obtained reveal critical insights into the performance of the parallel implementation of the Wandering Salesman Problem (WSP) using MPI and its comparison with theoretical expectations.

3.4.1 Comparison with Theoretical Performance

Theoretically, the speed-up of a parallel program should follow the ideal linear trend, where the speed-up S is equal to the number of processors P used. However, as observed in Figure 3, the actual performance deviates from this ideal case due to various factors inherent to parallel computation:

- **Communication Overhead:** In MPI-based parallelization, processors must exchange information (e.g., subproblem distribution and results aggregation). As the number of processors increases, the communication cost becomes more significant, leading to a sublinear speed-up, particularly for larger processor counts.
- **Load Imbalance:** Depending on the problem size and the method of subproblem distribution, certain processors may handle more complex tasks, leading to idle time for others and a reduction in overall efficiency.
- **Granularity of Tasks:** In methods with shallow subproblems (e.g., V3 or V4), the tasks assigned to each processor are not computationally intensive enough to justify the overhead of parallelization. This results in diminished performance compared to deeper subproblems (e.g., V6 or V7).

Although the parallel approach improves performance, the speed-up falls short of the theoretical ideal, especially as the number of processors increases. The most efficient methods (e.g., V6 and V7) approach linear speed-up for smaller numbers of processors but begin

to plateau as processor count increases beyond 16, indicating the diminishing returns of parallelization.

3.4.2 Expected vs Observed Performance

The observed performance aligns with expectations for this type of parallel algorithm. The methods incorporating deeper subproblems (V6 and V7) achieve significantly better performance due to their ability to divide the workload into sufficiently granular tasks, reducing idle time for processors. However, even with these improvements, the theoretical linear speed-up remains unattainable due to the factors mentioned above. This is a common limitation in parallel programming, particularly for algorithms with complex data dependencies like the WSP.

3.4.3 Problem Size and MPI Efficiency

Based on the results, the efficiency of MPI parallelization depends heavily on the problem size:

- For smaller problem sizes (e.g., fewer cities), the computational workload per processor is insufficient to offset the communication overhead. In such cases, the parallel approach is not efficient, and the execution time may not decrease significantly with the addition of more processors.
- For larger problem sizes (e.g., 17 or more cities), the workload becomes substantial enough to justify parallelization. The deeper subproblems (V6 and V7) particularly benefit in this context, as they can distribute the computational burden more effectively among processors.
- To achieve high efficiency, the problem size must be large enough to ensure that the computational workload dominates the communication cost. In practice, this means that the number of cities should scale with the number of processors to maintain a balanced workload distribution.

3.4.4 Conclusion

While the parallel approach demonstrates clear performance improvements, its efficiency is constrained by the problem size and the overhead of communication between processors. For the algorithm to achieve near-linear speed-up, it is essential to:

- Work with sufficiently large problem sizes to ensure that computation outweighs communication.
- Use deeper subproblem distributions (e.g., V6 or V7) to maximize the granularity of tasks and reduce idle time for processors.

These considerations highlight the trade-offs inherent in parallelizing algorithms like the WSP and the importance of tailoring the approach to the specific problem size and computational resources available.

4 Discussion

High-Performance Computing (HPC) is a double-edged sword when it comes to its environmental impact. On the one hand, it is an indispensable tool for addressing global challenges, such as climate change, by enabling advanced research and innovation. On the other hand, its immense power consumption raises concerns about sustainability. Reflecting upon Irina Kupiainen's article and the historical data from Green500, it is evident that HPC must evolve to balance its benefits and environmental footprint.

4.1 HPC's Contribution to Environmental Sustainability

HPC has the potential to drive sustainability in multiple ways. For example, as outlined by Irina Kupiainen, systems like LUMI in Finland play a critical role in climate research through high-precision modeling and digital twins. These tools allow researchers to simulate complex climate systems, predict environmental phenomena, and design more efficient clean energy solutions. This demonstrates the handprint of HPC, its positive impact on the environment and society by addressing grand challenges like climate change.

In addition, HPC systems are being used to optimize industrial processes, such as improving the efficiency of renewable energy technologies, reducing waste in manufacturing,

and advancing the circular economy. The ability to process large datasets from various disciplines accelerates the development of sustainable technologies. For example, HPC can help design energy-efficient materials or optimize transportation networks to reduce emissions.

4.2 Challenges and Negative Impacts of HPC

Despite its benefits, HPC systems face significant challenges due to their high energy consumption. As noted in Kupiainen's article, the ICT sector, including HPC, has a substantial carbon footprint. Supercomputers require massive amounts of electricity to operate, with additional energy demands for cooling systems. According to Green500, even the most energy-efficient supercomputers have power demands in the megawatt range, equivalent to the electricity usage of small towns.

Efforts to mitigate this negative impact include innovations like powering HPC systems with renewable energy, using free cooling in colder climates, and reusing waste heat. For instance, LUMI stands out as a pioneer in eco-efficiency by running entirely on renewable energy and using its waste heat to warm up to 20% of the houses in its surrounding city. These initiatives show that it is possible to significantly reduce the carbon footprint of HPC systems, but they are not yet universally adopted.

Another critical issue is the lifecycle of HPC systems. Supercomputers typically have a lifespan of five to seven years, after which their hardware becomes obsolete. The construction, maintenance, and decommissioning of these systems involve significant material and energy costs. Addressing this requires a shift towards modular and recyclable designs, as well as the reuse of existing infrastructure, as seen in Finland's practice of replacing decommissioned supercomputers with new systems in the same location.

4.3 Balancing the Trade-offs

To ensure that HPC contributes positively to the green transition, a holistic approach is required. This includes not only improving the energy efficiency of supercomputers but also expanding their role in sustainability efforts. For example:

- Energy Efficiency and Innovation: Technologies like quantum computing hold promise for reducing energy consumption in some computational tasks, though they cannot entirely replace traditional HPC.

- Lifecycle Sustainability: From construction to recycling, the entire lifecycle of supercomputers must align with circular economy principles.
- Cross-disciplinary Collaboration: HPC must facilitate multi-disciplinary research to address global challenges more effectively. This involves fostering collaboration across fields and industries to leverage HPC's full potential.

Moreover, governments and institutions must continue to invest in policies and strategies that prioritize both the handprint and footprint of HPC. For example, Finland's climate and environmental strategy for the ICT sector recognizes both the heavy footprint of HPC and its transformative potential for solving global problems. Such frameworks are essential for guiding the development and deployment of sustainable HPC systems.

4.4 Future Outlook

As the demand for computational power grows, HPC will remain at the forefront of both digital and green transitions. However, achieving full climate neutrality in HPC requires concerted efforts across multiple dimensions. The use of renewable energy, innovations in cooling and waste heat recovery, and advancements in computational efficiency will all play a role. Additionally, the role of HPC in enabling data-intensive, multi-disciplinary research will become increasingly important in addressing complex global challenges.

In conclusion, HPC has a dual role in the fight against climate change: as a significant energy consumer and as a key enabler of sustainable innovation. By addressing its environmental impact and leveraging its potential for positive contributions, HPC can play a leading role in driving the green transition while supporting the digital transformation.

5 Conclusions

This study demonstrates the effectiveness of parallel computing in solving computationally intensive problems like the Wandering Salesman Problem (WSP). By transitioning from a sequential Branch-and-Bound algorithm to a parallelized MPI-based implementation, significant reductions in execution time and improvements in scalability were achieved. The introduction of dynamic subproblem distribution further enhanced performance by balancing workloads across processors and reducing idle time.

Despite these advancements, the study highlights the limitations of parallelization, such as diminishing returns with excessive processors and communication overhead. These challenges underline the importance of carefully designing task granularity and balancing computational workload to maximize efficiency.

Beyond the technical achievements, the report sheds light on the environmental impact of high-performance computing (HPC). While HPC enables groundbreaking innovations in sustainability and optimization, its energy consumption and carbon footprint remain critical concerns. The findings advocate for continued efforts in improving the energy efficiency of HPC systems and leveraging renewable energy sources.

In conclusion, this project not only provides a scalable and efficient solution to the WSP but also emphasizes the dual role of HPC in driving innovation and addressing its environmental implications. Future work could explore hybrid computing approaches, such as combining traditional HPC with emerging technologies like quantum computing, to further enhance performance while minimizing environmental impact.

6 Reference

1. Irina Kupiainen. *HPC at the core of green and digital transition.* CSC-IT Center for Science in Finland, 2023. Available `https://www.scientific-computing.com/viewpoint/hpc-core-green-and-digital-transition`.

2. Siméon Ferez. *Solving the Traveling Salesman Problem with Parallel Computing.* Mar 25, 2023. Available `https://medium.com/@simeon.ferez/solving-the-traveling-salesman-pro`

3. Izzatdin A. Aziz, Nazleeni Haron, Mazlina Mehat, Low Tan Jung, Aisyah Nabilah. *Solving traveling salesman problem on high performance computing using message passing interface.* January, 2008. Available `https://www.researchgate.net/publication/228416450_Solving_traveling_salesman_problem_on_high_performance_computing_using_message_passing_interface`.

A Pseudo Code

```
distances ← matrice[MAX_CITIES][MAX_CITIES]
num_cities ← 0

function read_file(filename)
    open file
    num_cities ← lire le nombre de villes
    for i ← 0 to num_cities - 1 do
        for j ← 0 to i do
            if i = j then
                distances[i][j] ← 0
            else
                distances[i][j], distances[j][i] ← lire distance
    close file
    return num_cities

function branch_and_bound(path, level, visited, current_distance,
                          local_min_distance, local_best_path)
    if level = num_cities then
        if current_distance < local_min_distance then
            local_min_distance ← current_distance
            local_best_path ← copie de path
        return
    for i ← 1 to num_cities - 1 do
        if visited[i] = false then
            new_distance ← current_distance + distances[path[level-1]][i]
            if new_distance < local_min_distance then
                visited[i] ← true
                path[level] ← i
                branch_and_bound(path, level + 1, visited, new_distance,
                                 local_min_distance, local_best_path)
                visited[i] ← false
```

```
function solve_wsp(filename)
    start_time ← MPI_Wtime()
    rank, num_procs ← MPI_Comm_rank(), MPI_Comm_size()
    local_min_distance ←
    local_best_path ← tableau[MAX_CITIES]
    if rank = 0 then
        num_cities ← read_file(filename)
        if num_cities = 0 then
            exit
    MPI_Bcast(num_cities, distances)
    path ← tableau[MAX_CITIES]
    visited ← tableau[MAX_CITIES]
    for i ← rank + 1 to num_cities - 1 step num_procs do
        visited[i] ← true
        path[1] ← i
        branch_and_bound(path, 2, visited, distances[0][i],
                         local_min_distance, local_best_path)
        visited[i] ← false
    global_min_distance ← MPI_Allreduce(local_min_distance, MPI_MIN)
    if local_min_distance = global_min_distance then
        rank_with_min_distance ← rank
    MPI_Bcast(global_best_path, rank_with_min_distance)
    if rank = 0 then
        print global_min_distance, global_best_path
        print MPI_Wtime() - start_time

function main(argc, argv)
    filename ← lire argument "-i"
    if filename = null then
        print "Usage: ./prog -i <fichier>"
        exit
```

```
    solve_wsp(filename)
```

B Parallel code

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <mpi.h>
#include <string.h>

#define MAX_CITIES 17  // Maximum number of cities

int distances[MAX_CITIES][MAX_CITIES];  // Distance matrix.
int num_cities;

// Function to read the file and load the distance matrix.
int read_file(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        fprintf(stderr, "Error : Unable to open the file..\n");
        return 0;
    }

    fscanf(file, "%d", &num_cities);
    if (num_cities > MAX_CITIES) {
        fprintf(stderr, "Error : The number of cities exceeds the maximum limit. (%d).\n
        fclose(file);
        return 0;
    }

    for (int i = 0; i < num_cities; i++) {
        for (int j = 0; j <= i; j++) {
```

```
            int distance;
            if (i == j) {
                distances[i][j] = 0;
            } else {
                fscanf(file, "%d", &distance);
                distances[i][j] = distance;
                distances[j][i] = distance;  // Symmetric filling
            }
        }
    }

    fclose(file);
    return num_cities;
}

// Function for shortest path search with pruning
void branch_and_bound(int *path, int level, int visited[], int current_distance, int *lo
    if (level == num_cities) {  // All nodes have been visited
        if (current_distance <= *local_min_distance) {
            *local_min_distance = current_distance;
            memcpy(local_best_path, path, num_cities * sizeof(int));
        }
        return;
    }

    // Try each unvisited city
    for (int i = 1; i < num_cities; i++) {  // City 0 is always the starting point
        if (!visited[i]) {
            int new_distance = current_distance + distances[path[level - 1]][i];

            // Pruning: skip if the distance exceeds the local minimum solution
            if (new_distance >= *local_min_distance) continue;
```

```
            // Mark the city as visited
            visited[i] = 1;
            path[level] = i;

            // Continue the exploration
            branch_and_bound(path, level + 1, visited, new_distance, local_min_distance,

            // Undo the changes to explore other branches
            visited[i] = 0;
        }
    }
}

void solve_wsp(const char *filename) {
    int rank, num_procs;
    double start_time, end_time;
    double local_computation_time = 0.0, local_communication_time = 0.0;
    double total_computation_time, total_communication_time;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    // Timing
    start_time = MPI_Wtime();

    int local_min_distance = INT_MAX;
    int global_min_distance;
    int local_best_path[MAX_CITIES] = {0};
    int global_best_path[MAX_CITIES] = {0};

    if (rank == 0) {
```

```
    num_cities = read_file(filename);
    if (num_cities == 0) {
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
}

MPI_Bcast(&num_cities, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(distances, MAX_CITIES * MAX_CITIES, MPI_INT, 0, MPI_COMM_WORLD);

// Task distribution: generation of subproblems (depth 4)
int tasks[MAX_CITIES * MAX_CITIES * MAX_CITIES*  MAX_CITIES][4];
int total_tasks = 0;

if (rank == 0) {
    for (int i = 1; i < num_cities; i++) {
        for (int j = 1; j < num_cities; j++) {
            if (j == i) continue;
            for (int k = 1; k < num_cities; k++) {
                if (k == i || k == j) continue;
  for (int l = 1; l < num_cities; l++) {
                    if (l == i || l == j || l == k) continue;
          tasks[total_tasks][0] = i;
          tasks[total_tasks][1] = j;
          tasks[total_tasks][2] = k;
   tasks[total_tasks][3] = l;
          total_tasks++;
  }
            }
        }
    }
}
```

```
    MPI_Bcast(&total_tasks, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(tasks, total_tasks * 4, MPI_INT, 0, MPI_COMM_WORLD);

    int path[MAX_CITIES] = {0};
    int visited[MAX_CITIES] = {0};
    path[0] = 0;
    visited[0] = 1;

    for (int t = rank; t < total_tasks; t += num_procs) {
        int i = tasks[t][0], j = tasks[t][1], k = tasks[t][2], l = tasks[t][3];
        path[1] = i;
        path[2] = j;
        path[3] = k;
path[4] = l;
        visited[i] = visited[j] = visited[k] = visited[l] = 1;

        branch_and_bound(path, 5, visited, distances[0][i] + distances[i][j] + distances

        visited[i] = visited[j] = visited[k] = visited[l] = 0;
    }

    MPI_Allreduce(&local_min_distance, &global_min_distance, 1, MPI_INT, MPI_MIN, MPI_CO

    int rank_with_min_distance = -1;
    if (local_min_distance == global_min_distance) {
        rank_with_min_distance = rank;
    }

    MPI_Allreduce(MPI_IN_PLACE, &rank_with_min_distance, 1, MPI_INT, MPI_MAX, MPI_COMM_W

    if (rank == rank_with_min_distance) {
```

```
        memcpy(global_best_path, local_best_path, num_cities * sizeof(int));
    }
    MPI_Bcast(global_best_path, MAX_CITIES, MPI_INT, rank_with_min_distance, MPI_COMM_WO

    end_time = MPI_Wtime();

    if (rank == 0) {
        printf("Best distance : %d\n", global_min_distance);
        printf("Best path : ");
        for (int i = 0; i < num_cities; i++) {
            printf("%d ", global_best_path[i] + 1);
        }
        printf("\n");
        printf("Execution Time : %f secondes\n", end_time - start_time);
    }
}

int main(int argc, char *argv[]) {
    char *filename = NULL;
    MPI_Init(&argc, &argv);

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-i") == 0 && i + 1 < argc) {
            filename = argv[i + 1];
        }
    }

    if (!filename) {
        fprintf(stderr, "Use: %s -i <input file>\n", argv[0]);
        return EXIT_FAILURE;
    }
```

```
    solve_wsp(filename);

    MPI_Finalize();
    return 0;
}
```

C Scheduler Script

```
    #!/bin/bash
##
## MPI submission script for PBS on CR2
## ------------------------------------
##
##"MPI-sub2022v1"
## Follow the 6 steps below to configure your job
##
## STEP 1:
##
## Enter a job name after the -N on the line below:
##
#PBS -N wsp_result
##
## STEP 2:
##
## Select the number of cpus/cores required by modifying the #PBS -l select line below
##
## Normally you select cpus in chunks of 16 cpus
## The Maximum value for ncpus is 16 and mpiprocs MUST be the same value as ncpus.
##
## If more than 16 cpus are required then select multiple chunks of 16
## e.g. 16 CPUs: select=1:ncpus=16:mpiprocs=16
```

```
## 32 CPUs: select=2:ncpus=16:mpiprocs=16
## ..etc..
##
#PBS -l select=1:ncpus=16:mpiprocs=16
##
## STEP 3:
##
## Select the correct queue by modifying the #PBS -q line below
##
## half_hour -  30 minutes
## one_hour -   1 hour
## three_hour   -   3 hours
## six_hour     -   6 hours
## half_day -  12 hours
## one_day -  24 hours
## two_day -  48 hours
## five_day - 120 hours
## ten_day - 240 hours (by special arrangement)
##
#PBS -q half_hour
##
## STEP 4:
##
## Replace the hpc@cranfield.ac.uk email address
## with your Cranfield email address on the #PBS -M line below:
## Your email address is NOT your username
##
#PBS -m abe
#PBS -M paul.fontanges@cranfield.ac.uk
##
## ===================================
## DO NOT CHANGE THE LINES BETWEEN HERE
```

```
## ====================================
#PBS -j oe
#PBS -W sandbox=PRIVATE
#PBS -k n
ln -s $PWD $PBS_O_WORKDIR/$PBS_JOBID
## Change to working directory
cd $PBS_O_WORKDIR
## Calculate number of CPUs
export cpus=‘cat $PBS_NODEFILE | wc -l‘
sort -u $PBS_NODEFILE -o mpi_nodes.$$
export I_MPI_HYDRA_IFACE=ib0
export I_MPI_HYDRA_BOOTSTRAP=ssh
export I_MPI_HYDRA_RMK=pbs
## Debug options - only enable when instructed by HPC support
##export I_MPI_HYDRA_DEBUG=1
##export I_MPI_DEBUG=6
##export I_MPI_DEBUG_OUTPUT=%h-%r-%p-debug.out
## set some MPI tuning parameters to use the correct transport
## ========
## AND HERE
## ========
##
## STEP 5:
##
##  Load the default application environment
##  For a specific version add the version number, e.g.
##  module load intel/2016b
##
module use /apps/modules/all
module load intel/2021b
##
## STEP 6:
```

```
##
## Run MPI code
##
## The main parameter to modify is your mpi program name
## - change YOUR_EXECUTABLE to your own filename
##

mpirun -genvall -hostfile mpi_nodes.$$ -np ${cpus} ./a.out -i distances

## Tidy up the log directory
## DO NOT CHANGE THE LINE BELOW
## ============================
rm $PBS_O_WORKDIR/$PBS_JOBID
#
```