Prallelization of Genetic Algorithms to solve the Travelling Salesman Problem

Marinelli Alberto Roberto (638283) - a.marinelli9@studenti.unipi.it
Master Degree Computer Science, AI curriculum
Course of Parallel and Distribuited Systems (305AA),
Academic Year: 2022-2023

Abstract

This project focuses on parallelization of a genetic algorithm to solve the **Travelling Salesman Problem** (TSP). A sequential version is developed, followed by an analysis to identify components suitable for parallelization. Two parallel implementations are created using **standard threads** and **FastFlow**. Subsequently, the performance gains from parallelization are evaluated and the sequential and parallel versions are compared using metrics such as execution time and solution quality. The results provide insights into the benefits and tradeoffs of standard threads and FastFlow, showing how the field of **parallel optimization** algorithms is used to efficiently solve complex combinatorial problems.

Contents

1 Introduction

The **Travelling Salesman Problem** (TSP) is a complex combinatorial problem that involves finding the shortest possible route for a salesman to visit a set of cities and return to the starting point. **Genetic algorithms**, which simulate the process of natural selection to find optimal solutions, have proven to be effective in solving the TSP.

This report focuses on the **parallelization of a genetic algorithm** to tackle the TSP, aiming to leverage parallel computing techniques for enhanced efficiency. The objective is to exploit the inherent parallelism within the genetic algorithm and utilize multiple computational resources to expedite the solution search process.

To establish a foundation for the parallelization efforts, a sequential version of the genetic algorithm is initially developed. This sequential implementation serves as a baseline against which the parallel versions will be compared.

Chapter 2, titled *"Problem Setting"*, delves into the formal definition of the TSP and explores the utilization of genetic algorithms to solve it. The chapter also conducts an analysis to identify specific components within the algorithm that are best suited for parallelization. By closely examining the algorithm's structure and characteristics, this analysis aims to pinpoint the areas where parallelization can offer the greatest potential for optimization.

Within Chapter 3, titled *"Parallel Architectures Design"*, two distinct parallelization architectures are defined. These architectures are specifically designed to optimize the execution time of crucial functions within the genetic algorithm, including the initial generation, evaluation, and crossover processes. The first implementation utilizes **standard threads**, while the second leverages the **FastFlow** framework. Both implementations are constructed to harness the benefits of parallel computing, with the goal of achieving significant performance improvements.

Chapter 4, *"Parallel Algorithms Performance"*, is dedicated to evaluating and analyzing the performance gains attained through parallelization. To accomplish this, two essential performance indicators—speedup and scalability—are employed. Speedup measures the improvement in execution time achieved by parallelizing the algorithm, while scalability assesses the algorithm's ability to maintain efficiency as the number of computational resources increases. The chapter also presents the results of experiments conducted to compare the sequential and parallel versions in terms of execution time and solution quality.

Finally, the *"Conclusions"* in Chapter 5 highlights the insights and results of this study on the advantages and tradeoffs associated with using standard threads and the FastFlow framework in the field of parallel optimization algorithms. By examining the performance metrics and considering the experimental results, this report demonstrates the advantages and disadvantages of increasing the number of workers in parallelization problems.

2 Problem Setting

This chapter formally defines the Travelling salesman problem, then provides an overview of how to exploit genetic algorithms to solve it. Finally, by analyzing some high-level implementation details, an analysis is made on which parts of the algorithm are best suited for parallelization.

Formally, the **Traveling Salesman Problem** (TSP) can be defined as follows:
Given an undirected, complete graph $G = (V, E)$, where V represents a set of vertices (cities) and E represents the set of edges connecting pairs of vertices, the objective is to find a Hamiltonian cycle (a cycle that visits each vertex exactly once) with the minimum possible weight. The weight of an edge in the graph represents the distance or cost associated with traveling between two cities.

In this project, a **genetic algorithm** is used to address this problem.
The steps of a genetic algorithm are implemented following the guidelines listed in "A genetic algorithm tutorial" by Darrel Whitley [3], which offers a comprehensive guide to understanding and implementing genetic algorithms. It covers the fundamental concepts, explores the different components and techniques, and provides practical recommendations for designing effective GA to solve optimization problems.
In particular, the execution of GA is viewed as a two-step process. Selection is applied to the current population to create an intermediate population, using reproduction probabilities based on the quality of each chromosome calculated through the evaluation and fitness functions. Once the intermediate population is constructed, recombination (crossover and mutation) is applied to chromosomes with a certain probability. This generates new chromosomes that form the next population.
For more details, refer to Appendix A for a high-level overview of how the different phases work.

Once the problem is defined, the preliminary analysis is concerned with ensuring that a solution is reached through the iterative process. Figure 1 shows an experiment performed to empirically demonstrate the **convergence properties** of the described (sequential) algorithm and its parallel implementations.

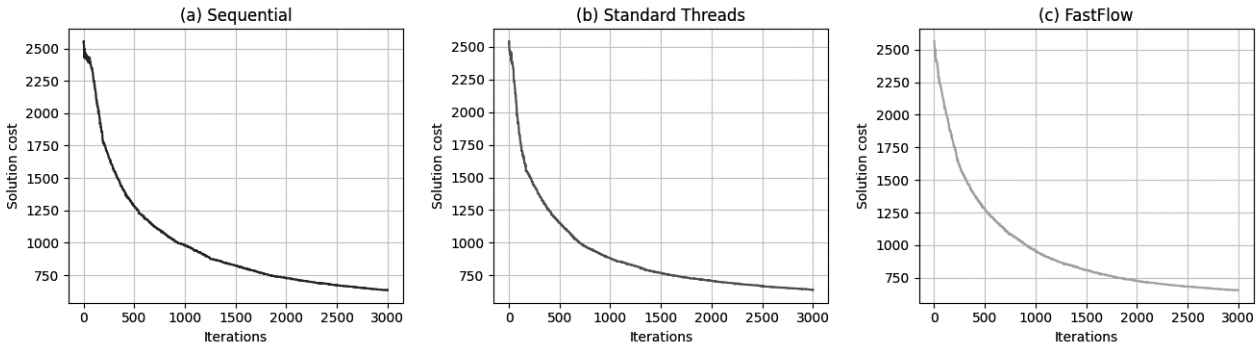

Figure (1) The results of 3.000 generations with the starting problem set with: 500 vertices (cities), edge weights between [1, 10], 2.000 initial chromosomes, mutation rate of 0.4, crossover rate of 0.8. In each graph, the quality of the solution on the y-axis and the number of iterations (generations) on the x-axis are shown.

Considering that the weights of the edges are between [1, 10] and that there are 500 vertices, the minimum solution potentially obtainable is cost 500 (if an edge of weight 1 is created between each node in the graph in the random generation), the algorithm achieves about 600 in each configuration as the final solution. Although the effective optimal solution of the generated problem in the experiment is not known, the results can be considered sufficiently close to the theoretical minimum solution.

2.1 Algorithm cost analysis

The first fundamental step that must be addressed to start a parallelization process is the analysis of the parts of the algorithm that require more execution time, which therefore could be optimized.

As described earlier, the proposed algorithm is divided into five basic components: initial generation, evaluation, fitness, selection, corssover and mutation. It should be kept in mind that it may not be necessary to parallelize all of them because parallelization of operations that are already fast enough may introduce overhead that leads to performance degradation. To understand which ones were selected for parallelization, experiments were performed in which the size of the initial problem was varied to see what percentage each operation affected the total execution time.

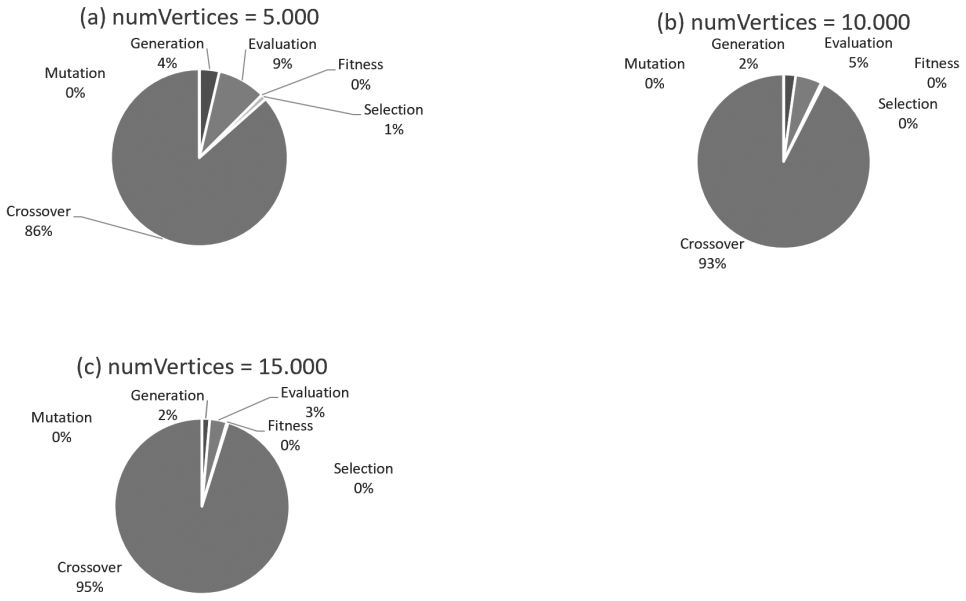

Figure (2) In each pie chart there are the percentages, calculated by an the average time of three experiments per configuration, of each stage in the sequential algorithm on a single generation of starting problems with different scales of size. The number of vertices is set to 5.000, 10.000 and 15.000 respectively, in addition the starting population is 15.000 chromosomes and the probabilities of crossover and mutation are set to 1 (100% probability of these operations occurring).

The experiments in the above Figure 2, show that as the problem size increases, the operation that dominates computation is the **crossover operation** that involves the exchange of genetic material between two parent individuals to create offspring individuals. Specifically, the implemented PMX crossover [2] involves randomly choosing a starting crossover point in the parents. The first operation performed is, starting from the crossover point in the chromosome of interest, to locate the corresponding element in the second chromosome. Then, a search operation takes place within the first chromosome to look for the corresponding element of the second chromosome and swap it (each swap takes place on the chromosome itself). Moving on to the next position, this process is repeated for about a twenty percent portion of the string length, for both chromosomes. So it is easy to realize that perform this operation within larger chromosomes becomes an increasingly expensive task. Note that in the experiment settings the probability of corssover occurring is set to 1 to force the algorithm to perform it obligatorily but generally this operation may also not be performed for some chromosomes, thus taking a smaller percentage of the execution time out of the total.

Then, **initial generation** and **evaluation** are two costly operations that depend heavily on the number of nodes in the graph and the size of the population to be evaluated.

The remaining operations, on the other hand, turn out to be fast even in more complex settings, meaning that there is no need to parallelize them both to achieve performance improvement and to avoid additional overhead.

Based on these considerations, it was considered essential to parallelize only the initial generation, evaluation and crossover operations.

3 Parallel Architectures Design

In this chapter, two parallelization architectures are defined to optimize the execution time of the initial generation, evaluation and crossover functions. In particular, standard threads and FastFlow are exploited to produce the two different implementations.

3.1 Standard thread implementation

The first implementation doesn't explicitly use native threads (`std::thread`), but instead leverages `std::async` with the `std::launch::async` policy to achieve parallelism. This choice is driven by several advantages provided by `std::async`:

First, it offers **automatic thread management**, where the C++ implementation determines whether to execute the task asynchronously in a new thread or synchronously within the calling thread. This decision is based on factors such as workload, available resources, and the threading policy of the implementation. By handling thread creation and management automatically, manual intervention is minimized, resulting in cleaner and less error-prone code.

Second, **asynchronous execution**, which allows the calling thread to continue its execution without waiting for the task to complete. This is beneficial when multiple tasks need to be initiated in parallel, enabling the calling thread to proceed with other computations or operations while awaiting the results.

Additionally, `std::async` returns a **future object** representing the result of the asynchronous task. Futures provide a convenient mechanism for result retrieval, completion status checking, and handling exceptions or errors that occurred during task execution. This facilitates synchronization and coordination between the calling thread and the asynchronous task.

By utilizing `std::async`, the code achieves parallelism, simplifies thread management, supports asynchronous execution, and provides a structured approach for result retrieval and synchronization.

In this implementation, parallelism is achieved by dividing the task into smaller portions assigned to workers.

Specifically, to explain how it works, consider the function for initial generation '*generatePopulation*' that takes three parameters: `chromosomeNumber`, `numVertices`, and `workers`, specifying the desired number of chromosomes, the number of vertices, and the number of parallel workers to be used.

A lambda function named `generateChromosomes` is defined inside *generatePopulation* to generate a portion of the chromosomes. It takes a start and end index and produces pairs consisting of fitness values and shuffled sequences of vertices.

The number of chromosomes per worker (`chromosomesPerWorker`) is determined by dividing the total chromosome number by the number of workers, with any remaining chromosomes assigned as `extraChromosomes` to be distributed.

A vector of `std::future` objects named futures is created to store the asynchronous results of each worker's computation.

A loop is used to create and launch worker tasks using `std::async` with the `std::launch::async` policy, ensuring asynchronous execution. Within the loop, start and end indices are calculated for each worker, with the last worker handling any `extraChromosomes`.

Each worker's task is initiated using `std::async` and the `generateChromosomes` lambda function, with the calculated start and end indices. After launching the tasks, another loop collects the partial results from each `std::future` using `future.get()`, appending them to the population vector.

Similarly, the functions for evaluation and crossover have been coded. By exploiting the idea that chromosomes could be processed in independent blocks (each assigned to each thread), access to shared resources can be guaranteed without the explicit use of lock mechanisms by accessing them using specific index ranges to prevent each worker from concurrently accessing the spaces of other workers. In addition, using this fork/join model avoids exchanging information between workers and with a master, which could become a heavy operation as the problem size increases.

3.2 Fastflow implementation

Using **FastFlow** [1], a parallel programming framework, this implementation follows a data-parallel approach, where the chromosome generation task is divided into chunks and distributed among worker nodes for parallel execution. FastFlow parallel patterns, such as emitter, worker, and collector nodes, are used to structure the parallel computation and handle the communication between nodes. The code takes advantage of modern hardware concurrency by dynamically adjusting the number of worker nodes based on the available resources.

To go into the details of the implementation, let us again consider the initial gener-

ation function '*generatePopulation*'. In this case, as mentioned above, several components have been defined such as the `ChunksEmitter`, the `ChunksCollector`, the node for worker execution (`ChromosomeGenerationWorker`), and the `setupComputation` function for resource allienement.

The `setupComputation` method calculates the number of workers based on the available hardware concurrency and the desired number of chromosomes. Note that this method is called every time a parallel operation is started because due to stochastic selection the algorithm may generate an intermediate population of varying size that is resized only after the evaluation process is performed.

`ChunksEmitter` is a FastFlow node (`ff_node_t`) responsible for emitting chunks of chromosome generation tasks to the worker nodes. It inherits from `ff_node_t<std::pair<int, int>>`, indicating that it processes and emits pairs of integers. It has a reference to a vector of `std::pair<int, int>` called `chunks` and an integer `generationsNumber`. The `svc` method is the main execution method, it decrements the `generationsNumber` and sends out each chunk in the `chunks` vector using the `ff_send_out` method. When `generationsNumber` becomes zero, it returns the end-of-stream (`EOS`) signal.

Correspondingly, the `ChunksCollector` is a FastFlow node with the same properties of emitter and is responsible for collecting the generated chromosomes from the worker nodes. The `svc` method processes each chunk received. If all chunks have been processed, it sends out the chunk to the next stage (if any), otherwise, it deletes the chunk. It uses the `GO_ON` signal to continue processing.

Each chunk is processed using `ChromosomeGenerationWorker` that performs the actual generation of chromosomes. It also inherits from `ff_node_t<std::pair<int, int>>` and has references to a graph and vectors of chromosome populations. The `svc` method receives a chunk (pair of integers) indicating the range of chromosomes to generate, then generates random chromosomes and stores them in the local `partialPopulation` vector to avoid highly competitive access to the shared structure. After generating the chromosomes, it transfers the values from `partialPopulation` to the main population vector. Once these operations are completed, it sends out the processed chunk to the next stage using `ff_send_out`.

Finally, the *generatePopulation* method sets up the **FastFlow Farm** and executes the chromosome generation. It creates instances of `ChunksEmitter`, `ChunksCollector`, and multiple `ChromosomeGenerationWorker` nodes and then wraps these nodes in an `ff_Farm` object, which manages the execution and communication between the nodes.

The above logic is replicated in both the evaluation and crossover functions. The use of farms offers several significant advantages, in addition to the parallelization of processes, that contribute to the overall efficiency and effectiveness of the system.

A key advantage is scalability. As system requirements grow, farms can easily handle increased workloads by adding more worker nodes or adding it within other parallelization systems. This scalability allows for continuous expansion of computational resources,

enabling the system to efficiently handle larger and more complex tasks.

Then, by employing farms, the complexities associated with managing parallel execution and inter-node communication are abstracted away. This abstraction simplifies the development process and enhances code readability. It is possible to focus on the logic of each phase without having to deal with low-level details such as thread creation, synchronization, or communication mechanisms.

Finally, error handling is another essential feature offered by farms. If an error occurs during the execution of a worker node, the farm can handle the error and propagate it appropriately. This built-in error handling capability ensures the integrity of the overall computation and allows for graceful error recovery. By providing error resilience, farms enhance the robustness of the system.

4 Parallel Algorithms Performance

4.1 Measures

To evaluate and model the performance of a parallel program, two important performance indicators are utilized: speedup and scalability.

Speedup is a measurement that compares the execution time of the sequential program with the execution time of the parallel program. It quantifies how much faster the parallel program is in comparison to the sequential counterpart. The speedup is a function of the parallelism degree, denoted by n. The formula for calculating speedup is as follows:

$$\text{speedup}(n) = \frac{T_{\text{seq}}}{T_{\text{par}(n)}} \tag{1}$$

In this formula, $Tseq$ represents the execution time of the sequential program, and $Tpar(n)$ represents the execution time of the parallel program with a parallelism degree of n.

Scalability measures the efficiency of the parallel implementation when dealing with increasing levels of parallelism. It quantifies the improvement in performance achieved by increasing the parallelism degree from 1 to n. The formula for calculating scalability is as follows:

$$\text{scalability}(n) = \frac{T_{\text{par}(1)}}{T_{\text{par}(n)}} \tag{2}$$

Here, $Tpar(1)$ denotes the execution time of the parallel program with a parallelism degree of 1, and $Tpar(n)$ represents the execution time of the parallel program with a parallelism degree of n.

Both speedup and scalability are essential metrics for evaluating the performance of parallel programs. Speedup assesses the overall improvement achieved through parallelization, while scalability focuses on the efficiency of parallelism with varying degrees. These metrics help in understanding and modeling the behavior of parallel programs and optimizing their performance.

Furthermore, for speedup the expectation on an **ideal time** where $Tpar(n) = Tseq/n$ is calculated, this implies that the parallel execution time is directly proportional to the degree of parallelism, n. This scenario represents perfect parallelism, in which an increase in the number of parallel resources or processors leads to a proportional decrease in execution time. In the graphs it is reported to have an upper bound on what maximum theoretically can be achieved as the best parallelization time.

4.2 Experiments

Considering the results of the experiments, it can be seen that increasing the number of workers in parallelization problems can have both advantages and disadvantages.

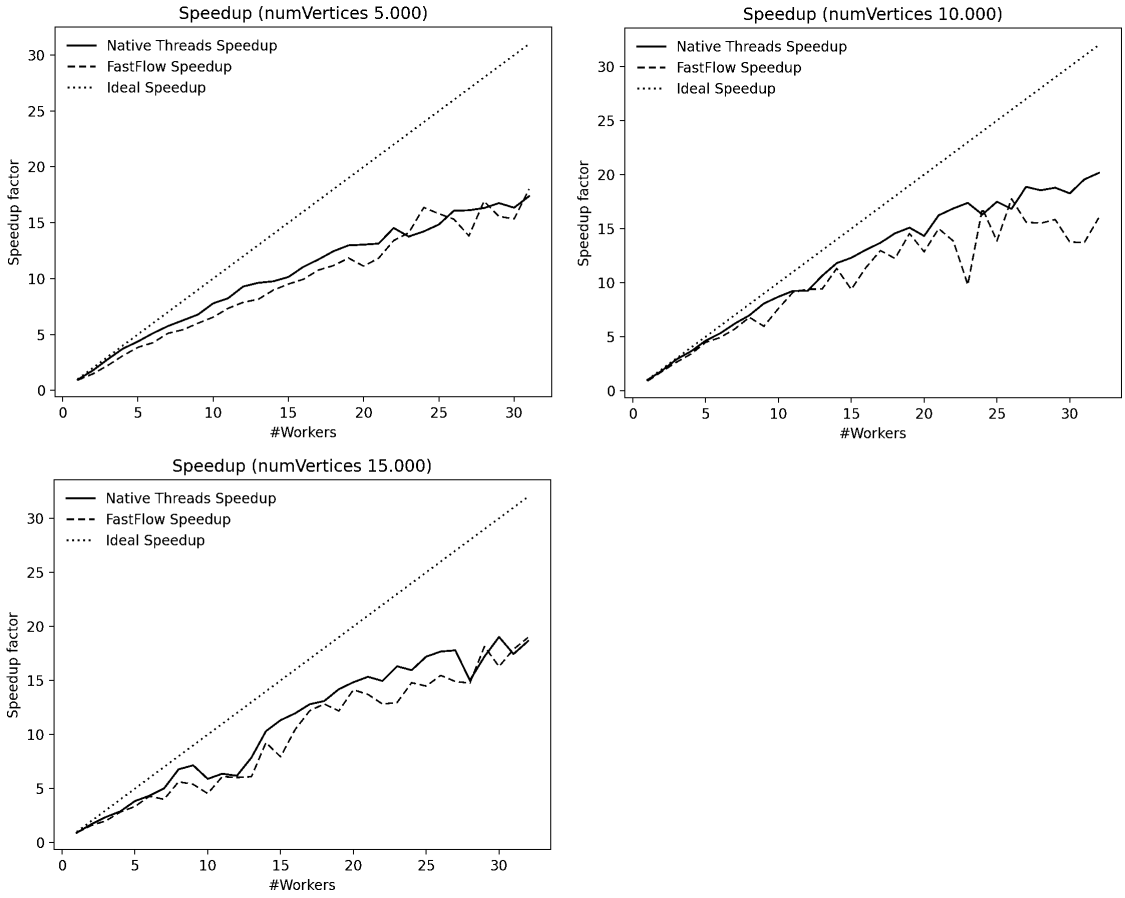

Figure (3) The graphs depict the speedups of the FastFlow and the standard threads implementations on different magnitudes of the starting problem. The number of vertices is set to 5.000, 10.000 and 15.000 respectively, one generation, mutation and crossover probability at 1 (100% probability of these operations being performed), starting population of 15.000 chromosomes. On the y-axis is the speedup factor, on the x-axis is the number of workers (threads).

In Figure 3, **speedup** indicates that increasing the number of workers can lead to improved performance and faster execution times. Especially in the case of TSP with genetic algorithms, where tasks can be executed independently without the need for frequent communication or synchronization. With more workers, the workload can be divided more finely, enabling better resource utilization and reduced computation time.

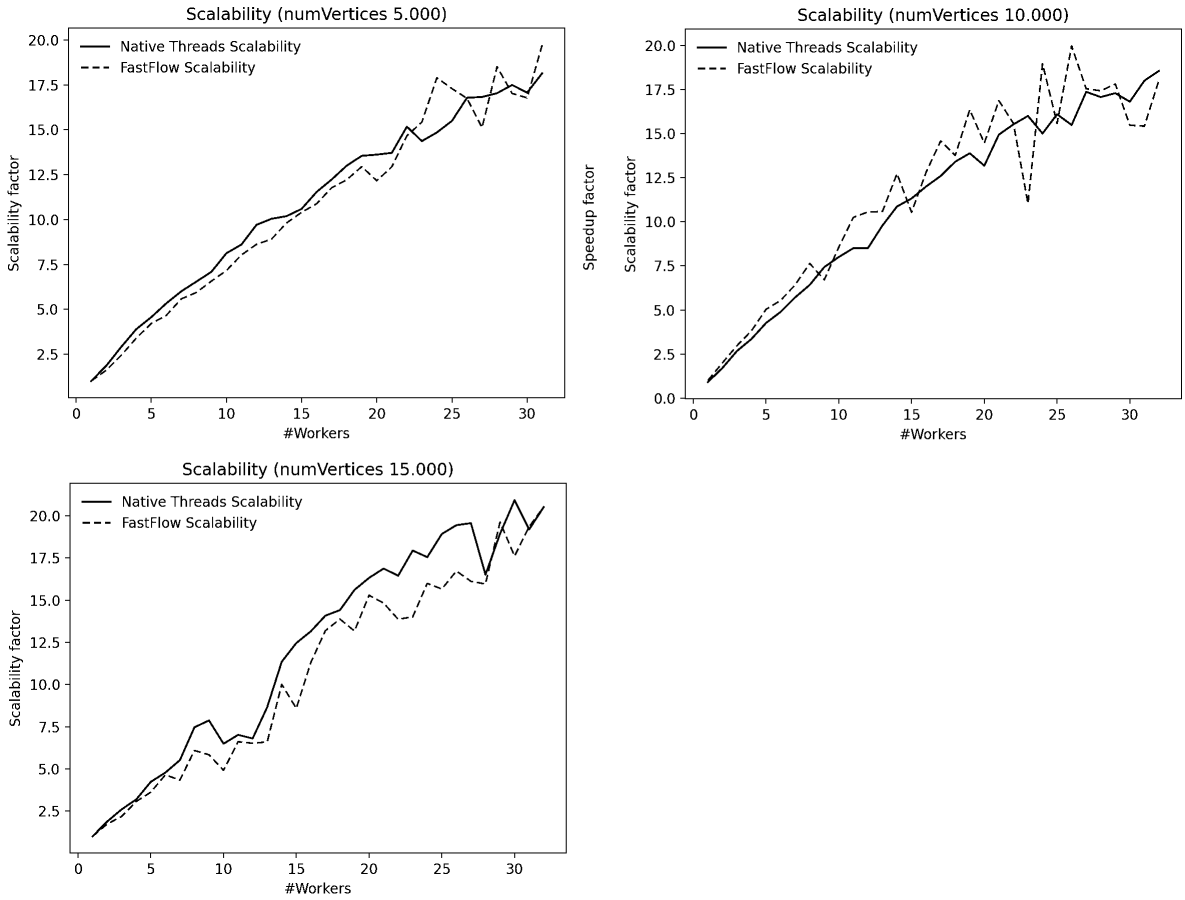

Figure (4) The graphs depict the scalability of the FastFlow and the standard threads implementations on different magnitudes of the starting problem. The number of vertices is set to 5.000, 10.000 and 15.000 respectively, one generation, mutation and crossover probability at 1 (100% probability of these operations being performed), starting population of 15.000 chromosomes. On the y-axis is the speedup factor, on the x-axis is the number of workers (threads).

As for **scalability**, in Figure 4, it allows the problem to be solved efficiently on larger systems or with larger data sets. By increasing the number of workers, available computational resources can be leveraged more effectively and larger workloads can be handled. This can be particularly advantageous for tasks involving big data processing or complex simulations.

Looking at the results in general, **overhead** and **communication** should be noted, in fact increasing the number of workers may introduce additional overhead due to communication and coordination among them, especially if the problems are small. Thus, by observing the above metrics that are conducted on single-generation experiments, a trade-off can be established to balance the ideal number of threads to perform the specific task.

Finally, the last experiment performed is the measurement of times in milliseconds, on different problem sizes, with both sequential and parallel implementations.

Implementation	**#Threads**	**Generation**	**Evaluation**	**Crossover**	**Total**
Sequential	-	698 ms	2.371 ms	16.929 ms	20.684 ms
Standard Thread	8	500 ms	629 ms	2.441 ms	3.762 ms
FastFlow	8	262 ms	425 ms	2.375 ms	3.188 ms
Standard Thread	16	265 ms	522 ms	1.160 ms	2.089 ms
FastFlow	16	209 ms	290 ms	1.150 ms	1.783 ms
Standard Thread	32	134 ms	347 ms	630 ms	**1.216** ms
FastFlow	32	97 ms	361 ms	1063 ms	1657 ms

Table (1) Performance comparison on number of vertices set to 5.000, the starting population is 15.000 chromosomes and the probabilities of crossover and mutation are set to 1 (100% probability of these operations occurring)

Implementation	**#Threads**	**Generation**	**Evaluation**	**Crossover**	**Total**
Sequential	-	1.326 ms	3.362 ms	51.110 ms	56.061 ms
Standard Thread	8	403 ms	683 ms	76.43 ms	8.995 ms
FastFlow	8	490 ms	1.023 ms	11.109 ms	12.960ms
Standard Thread	16	340 ms	716 ms	4.131 ms	5.476 ms
FastFlow	16	459 ms	618 ms	6.452 ms	7.853 ms
Standard Thread	32	307 ms	780 ms	2.304 ms	**3.670** ms
FastFlow	32	270 ms	815 ms	3.841 ms	5.174 ms

Table (2) Performance comparison on number of vertices set to 10.000, the starting population is 15.000 chromosomes and the probabilities of crossover and mutation are set to 1 (100% probability of these operations occurring)

Implementation	**#Threads**	**Generation**	**Evaluation**	**Crossover**	**Total**
Sequential	-	2.013 ms	5.184 ms	127.834 ms	135.509 ms
Standard Thread	8	740 ms	1.443 ms	16.755 ms	19.317 ms
FastFlow	8	1.018 ms	2.375 ms	18.851 ms	22.817 ms
Standard Thread	16	612 ms	1.217 ms	10.427 ms	12.660 ms
FastFlow	16	826 ms	1.977 ms	9.425 ms	12.733 ms
Standard Thread	32	544 ms	1.172 ms	5.510 ms	**7.589** ms
FastFlow	32	610 ms	1.330 ms	6.680 ms	9.048 ms

Table (3) Performance comparison on number of vertices set to 15.000, the starting population is 15.000 chromosomes and the probabilities of crossover and mutation are set to 1 (100% probability of these operations occurring)

As can be seen from Tables 1, 2 and 3, the goal stated in Chapter 2.1 can be considered reached. In fact, a net improvement is achieved as the number of threads increases and the problem become bigger, this implies lower execution times, in example in Table 3, up to a reduction by a factor of about *18x*, if we go from sequential to parallel with 32 threads.

5 Conclusion

In conclusion, parallelization is highly advantageous for speeding up genetic algorithms used to solve TSP problems, especially when dealing with large instances that are difficult to solve sequentially. It offers improved computational efficiency, increased exploration of the search space, faster convergence, and scalability.

While, considering implementation aspects, standard thread-based parallelization using async/await mechanisms provides benefits, but using a specialized framework like FastFlow offers additional advantages. FastFlow provides a high-level abstraction, efficient task scheduling, optimized communication and extensibility of the parallelization structure.

A Genetic Algorithm Pseudocode

Algorithm 1 runGeneticAlgorithm

Require: $generationNumber, chromosomeNumber, numVertices, crossoverRate, mutationRate$

```
generatePopulation(chromosomeNumber, numVertices)
for generation = 1 to generationNumber do
    evaluate(evaluationsAverage, chromosomeNumber)
    fitness(evaluationsAverage)
    selection(intermediatePopulation)
    crossover(intermediatePopulation, crossoverRate)
    mutation(intermediatePopulation, mutationRate)
    evaluationsAverage = 0
    population.swap(intermediatePopulation)
end for
```

Algorithm 2 generatePopulation

Require: $chromosomeNumber, numVertices$

```
sequence = createSequence(numVertices) {Create a vector with values from 0 to numVertices}
for i = 0 to chromosomeNumber do
    shuffle(sequence) {Shuffle the sequence}
    keyValue = 0
    population.add(keyValue, sequence) {Add the keyValue and sequence to the population}
end for
```

Algorithm 3 evaluate

Require: $evaluationsAverage, cutValue$

1: $chromosomeSize \leftarrow$ graph.getNumVertices()
2: **for each** individual **in** population **do**
3: $chromosomeScoreCurrIt \leftarrow 0$
4: **for** $j = 0$ to $chromosomeSize - 2$ **do**
5: $chromosomeScoreCurrIt$ += graph.getWeight($individual.sequence[j]$, $individual.sequence[j + 1]$)
6: **end for**
7: $chromosomeScoreCurrIt$ += graph.getWeight($individual.sequence[0]$, $individual.sequence[chromosomeSize - 1]$)
8: $individual.keyValue \leftarrow chromosomeScoreCurrIt$
9: **end for**
10: sort($population$)
11: $totalScore \leftarrow 0$
12: $numElements \leftarrow \min(cutValue, \text{size}(population))$
13: **for** $i = 0$ to $numElements - 1$ **do**
14: $totalScore$ += $population[i].keyValue$
15: **end for**
16: $evaluationsAverage \leftarrow \frac{totalScore}{numElements}$
17: resize($population, numElements$)

Algorithm 4 fitness

Require: $evaluationsAverage$

1: **for** $i = 0$ to size($population$) $- 1$ **do**
2: $individual \leftarrow$ population[i]
3: $individual.keyValue$ /= $evaluationsAverage$
4: **end for**

Algorithm 5 selection

Require: *intermediatePopulation*

distribution ← createUniformDistribution(0.0, 1.0)
for each individual **in** population **do**
  numCopies ← integerPart(*individual.keyValue*)
  for $i = 0$ to *numCopies* − 1 **do**
    intermediatePopulation.add({individual.keyValue, individual.sequence})
  end for
  fractionalPart ← individual.keyValue − numCopies
  if getRandomValue(*distribution*) < fractionalPart **then**
    intermediatePopulation.add({individual.keyValue, individual.sequence})
  end if
end for

Algorithm 6 crossover

Require: *intermediatePopulation*, *crossoverRate*

distribution ← uniform distribution between 0.0 and 1.0
elementSize ← size of chromosome path in intermediatePopulation
twentyPercent ← integer value of (elementSize * 0.2)
if getRandomValue(distribution) < crossoverRate **then**
  for $i \leftarrow 0$ to size of intermediatePopulation − 1 **do**
    vec1 ← reference to the chromosome path of intermediatePopulation[i]
    if $i + 1 <$ size of intermediatePopulation **then**
      vec2 ← reference to the chromosome path of intermediatePopulation[i+1]
      startIndex ← random integer between 0 and size of (vec1 - twentyPercent)
      endIndex ← startIndex + twentyPercent
      for $j \leftarrow$ startIndex to endIndex **do**
        pos_1 ← find the position of vec2[j] in vec1
        pos_2 ← find the position of vec1[j] in vec2
        swap vec1[pos_1] with vec1[j]
        swap vec2[pos_2] with vec2[j]
      end for
    end if
  end for
end if

Algorithm 7 mutation

Require: $intermediatePopulation, mutationRate$
1: $distribution \leftarrow$ createUniformDistribution$(0.0, 1.0)$
2: **for each** individual **in** intermediatePopulation **do**
3: **if** getRandomValue$(distribution) <$ mutationRate **then**
4: $size \leftarrow$ size$(individual.sequence)$
5: **if** size > 1 **then**
6: $index1 \leftarrow$ getRandomIndex$(0, \text{size} - 1)$
7: $index2 \leftarrow$ getRandomIndex$(0, \text{size} - 1)$
8: swap$(individual.sequence[index1], individual.sequence[index2])$
9: **end if**
10: **end if**
11: **end for**

B Run configurations

The code can be built (inside the folder *genTSP*, use the command: `g++ -O3 -std=c++17 main.cpp -o genTSP`) and executed by passing the correct parameters to the main function.

The following are the parameters and their respective domains to be passed as input to the main.cpp function.

Parameter	**Domain**
numVertices	integer > 0
chromosomesNumber	integer > 0
generationNumber	integer > 0
crossoverRate	double $\in [0, 1]$
mutationRate	double $\in [0, 1]$
workersNumber	integer > 0
[seed]	integer (optional value)

Table (4) main.cpp parameters

References

[1] Marco Aldinucci et al. "FastFlow: High-Level and Efficient Streaming on Multi-Core". In: *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2009, pp. 173–180. DOI: 10.1109/PDP.2009.26.

[2] Göktürk Üçoluk. "Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation". In: *Intelligent Automation and Soft Computing* 3 (Jan. 2002). DOI: 10.1080/10798587.2000.10642829.

[3] Darrell Whitley. "A genetic algorithm tutorial". In: *Statistical Computing* 4.1 (June 1994), pp. 65–85. DOI: 10.1007/BF00175354. URL: https://doi.org/10.1007/BF00175354.