Parallel and distributed systems:
Paradigms and Models

UNIVERSITÀ DI PISA

A parallelized Genetic algorithm for the Travelling salesman problem

Report of the final Project

2022/2023

Samuele Vezzuto

Student ID: 656132

Abstract

The Traveling Salesman Problem [**TSP**] is an NP-Hard problem, its aim is to find the shortest path to visit a set of cities exactly once and return to the starting city. Because of its complexity, it cannot be solved in polynomial time. The solution will be approximated, in the case of this project a genetic algorithm has been used.

The algorithm used starts from a randomly generated population , where each chromosome represents a possible path, and for each iterations it applies three genetic operators to the population.

The three genetic operators applied are: **selection**, **crossover**, and **mutation**.

The algorithm has been implemented in a Sequential version, a Native threads version and a version based on FastFlow.

1 Analysis

1.1 The sequential version

To analyze the performance of the TSP algorithm and determine which operations can be parallelized, the algorithm was first implemented in a sequential version.

This allowed measuring the execution time of each operation.

The execution times obtained from the sequential version were then used to calculate the **improvements** achieved by the parallel implementations.

The structure of the algorithm can be observed in ***Code snippet 1***, it is composed by the following operations:

Population generation, which is achieved by simply shuffling a vector of integers, creating a random path for each chromosome.

Evaluation process, which first evaluates the chromosomes, accordingly to their paths length, and then sorts them. After the sorting only the best elements are kept, accordingly to elitism technique; that's done to maintain the chromosomes dimension constant.

Crossover operation combines the **i-th** and **(i+1)-th** paths to generate new solutions. The calculation of the new path uses the **Partially mapped crossover algorithm**.

The **mutation** operations are performed on percentage of elements, that can be changed, starting from a random index from the chromosomes. This ensures that not only the first elements are mutated and avoids recomputing a random index to mutate repeatedly. The mutation consists of **reversing** the elements between two random indexes.

After the required number of iterations, the **best path** is determined by selecting the first element in the calculated paths, which can then be returned.

1.2 Parallelization considerations

The sequential code has been structured in a way that allows for future parallelization. Specifically, accessing the array directly at a specific position eliminates the need for mutex locks every time the chromosome array is accessed. The evaluation operation must be performed after all the mutations and crossovers are calculated.

From the examination of the sequential algorithm through multiple runs as shown in ***Figure 1***, it becomes evident that the most time-consuming operation is the **evaluation** process.

In addition, the **crossover** operation has a noticeable impact, while the **initial generation** of the chromosome population has a smaller impact, but still relevant.

The **sorting** and **mutation** processes, on the other hand, have a negligible impact.

Given these observations, it is worth to consider parallelizing the **crossover**, the **initial generation** and the **evaluation** to study potential improvements.

The algorithm can be parallelized using the **Data parallel pattern** since the required data for each iteration of the loop are already available. A map pattern can be used for the three most consuming operations. Thanks to the data parallel patter, we have the capability to calculate the chunks before each iteration. This approach ensures that every worker has assigned an equal amount of operations, with each individual operation consuming nearly the same amount of time.

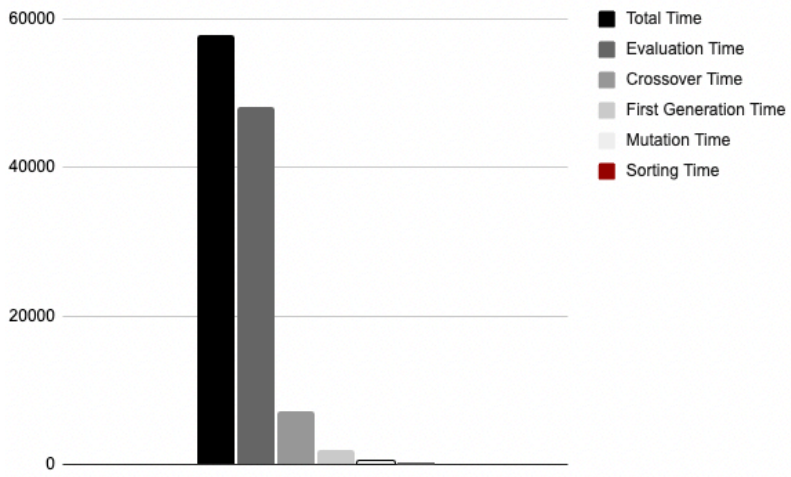


Figure 1

1.3 Time analysis

From the analysis of the algorithm in performed in the previous chapter, we can analyze from a theoretical point of view the times for the two implementation .

Starting from the sequential time:

$$T_{seq}(n) = T_{init} + T_{first-gen} + K * (T_{evolve}(n) + T_{mutation}(n) + T_{sort}(n))$$

Where n is the number of chromosomes and K is the number of iterations.

The T_{init} represents the time to initialise the variables, that can be considered negligible the variable initialisation time.

The time for each evolution is defined by the following formula:

$$T_{evolve}(n) = T_{crossover}(n) + T_{evaluate}(n)$$

From the $T_{seq}(n)$ it can be derived the $T_{par}(n, nw)$, which is defined as:

$$T_{par}(n, nw) = T_{init} + \frac{T_{first-gen}}{nw} + K * (\frac{T_{evolve}(n)}{nw} + T_{split-evo} + T_{merge-evo} + T_{mutation}(n) + T_{sort}(n)) + T_{split-first-gen} + T_{merge-first-gen}$$

Where the $T_{split-evo} + T_{merge-evo}$ represent the overload for each iteration, and $T_{split-first-gen} + T_{merge-first-gen}$ the overload for the first generation.

The possible improvements for the parallelised version of the algorithm have as upper bound the ideal time, which is given by the following formula:

$$T_{Ideal} = \frac{T_{Seq}}{nw}$$

1.4 Metrics

The metrics used to study the actual performances gains from the parallels implementations are the following:

Speed-up which represents the efficiency of parallelization compared to the sequential execution time

$$speedup(nw) = \frac{T_{seq}}{T_{par}(nw)}$$

Scalability which indicates the extent of improvement achieved with a higher degree of parallelism:

$$scalab(nw) = \frac{T_{par}(1)}{T_{par}(nw)}$$

Efficiency: Measures how close the parallel implementation is to the ideal time.

$$\varepsilon(nw) = \frac{T_{ideal}}{T_{par}(nw)} = \frac{T_{Seq}}{nw} * \frac{1}{T_{par}(n)} = \frac{speedup(nw)}{nw}$$

2 Parallel Implementations

2.1 Native threads implementation

The native threads version of the code uses the **std::future**, introduced with C++11, to facilitate the handling of exceptions in the multithreaded implementation. The chunks are calculated to equally distribute the workload to each worker.

The parallel concurrent access to the data structure, which holds the chromosomes, is managed by using appropriate data structures to **temporarily store** the elements. Only when a worker thread completes its calculations, the shared data structure is accessed and the contents of the temporary data structure are **appended** to it.

This is achieved using **std::lock_guard** to ensure thread safety for the concurrent access to the shared data structure.

2.2 FastFlow implementation

The Fast Flow version of the code uses the **ff:ff_Farm** class to handle the first **generation** and the evolutionary operations of **crossover** and **evaluations**.

In this implementation, the workload is partitioned into chunks using emitters, which are connected to the farm via collectors.

The mechanism for managing concurrent access through auxiliary structures has also been implemented in this version of the algorithm.

3 Experiments performed

3.1 Testing environment

All the final tests have been performed on a server equipped with an AMD EPYC 7301 16-Core Processor, provided by the University of Pisa.

For each input was performed multiple runs, and the average times were used, excluding results that deviated significantly from the average.

Each test was based on 10 generations, with 10000 chromosomes, and for an increasing number of cities: 10000, 15000, and 20000.

The tests were executed for the sequential version, the Native threads version and the Fast Flow version. The last two were tested with an increasing number of threads: 1, 2, 4, 8, 32, 40, and 64.

3.2 Algorithm correctness

To test the correctness of the algorithm's solution, it was compared to a brute force solution in multiple runs ~ 50.

The experiment involved a set of 11 cities, and the results showed that the best path identified by the algorithm was **only 7%** worse than the optimal path determined by the brute force method.

This comparison confirmed the algorithm's accuracy in finding near-optimal solutions.

3.3 Results

The results in the graphs show the gain from the use of parallelization. Those improvements grows up with the number of cities. The performance for the parallels implementations are similar, but **FastFlow** showed the **highest peak** for speedup and scalability in the graph **G3**. The scalability and speedup showed similar values, suggesting that the impact of parallelization operations to split the work, and the consequently overhead does not have an huge impact.

As shown in all the graphs, the efficiency tends to decrease when the number of workers exceeds 6.

So, I found the obtained results improvable. I decided to follow Gustafson's Law and scale up the problem to determine if the algorithm's performance improvements would become more interesting.

3.4 Extra tests

The extra tests were conducted on a significantly larger number of chromosomes and cities, spanning across 3 generations. This was necessary due to the bigger computation times involved, and the tests were run multiple times to ensure accuracy and consistency.

The configuration consisted of 50,000 chromosomes and 40,000 cities, pushing the boundaries of the implementation. With this configuration the improvements where tangible, with a peak Speedup of 16 as shown in ***Figure 2***.

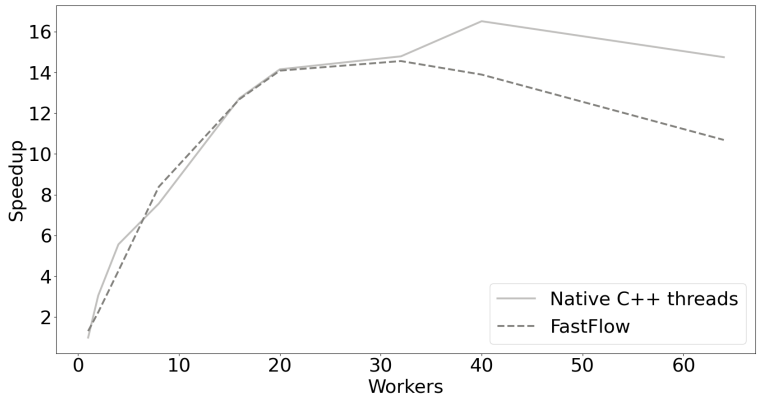


Figure 2

4 Manual

The project is composed by the main.cpp class, the calculator.h which is a virtual class used to choose runtime the actual type of calculator: sequential, fastflow or native threads, implemented in their own file.

The code can be executed using the script.sh file in the src folder to execute the configurations. You can also compile it and run it by using the ./tsp [parameters]. The parameters are in the following order:

- **type**: The type of execution 0 for **sequential**, 1 for **native threads**, 2 for **FastFlow**.
- **number of threads:** The number of threads to use, if sequential put 1 thread.
- **Chromosomes number**: The number of chromosomes.
- **number of cities**: The total number of cities in the problem.
- **number of iterations**: The number of generations for the genetic algorithm.
- **mutation percentage:** The percentage of mutation applied during the genetic algorithm, use a value 1 to 100.

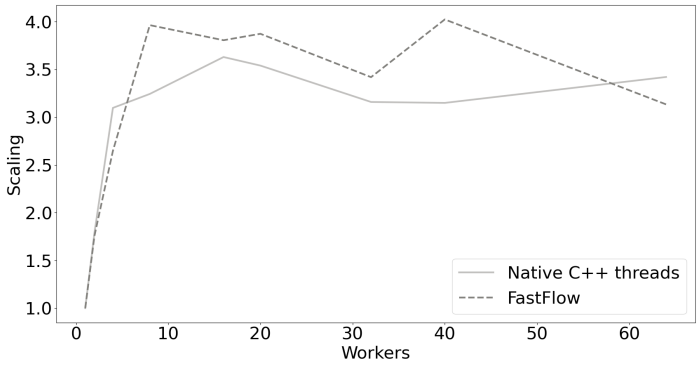
5 Conclusions

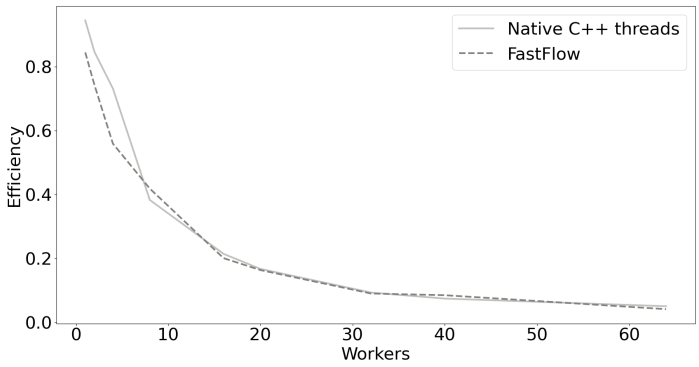
In this project was performed an analysis of the parallelization of the TSP Genetic algorithm, with different implementations: Sequential, Native Threads and FastFlow. The parallel implementations perform similar.

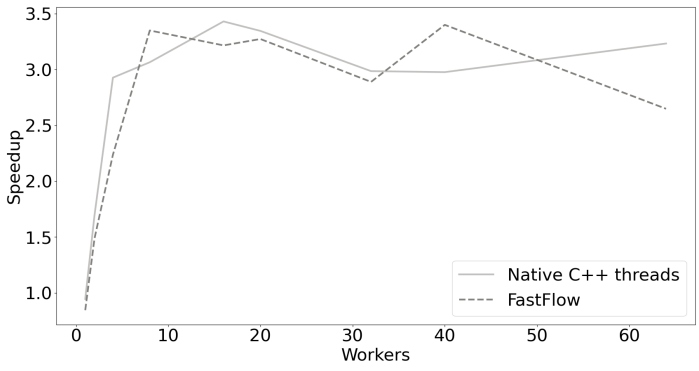
The results reveals also an interesting aspect of the problem. It seems that the initial limit on the performance gain was related to the operations performed by the algorithm, which were not particularly demanding.

This observation suggests that certain factors, such as the implementation of the elitism technique, which keeps only the best chromosomes, and the optimizations made in the mutation calculation (which do not require a large number of random values but just one), influenced the limit in the gain obtained. Consequently, a more substantial problem size was necessary to achieve a better speedup, as described in chapter **3.4**.

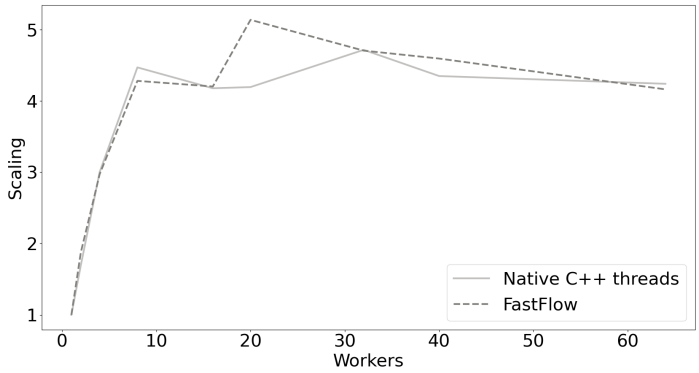
Graphs

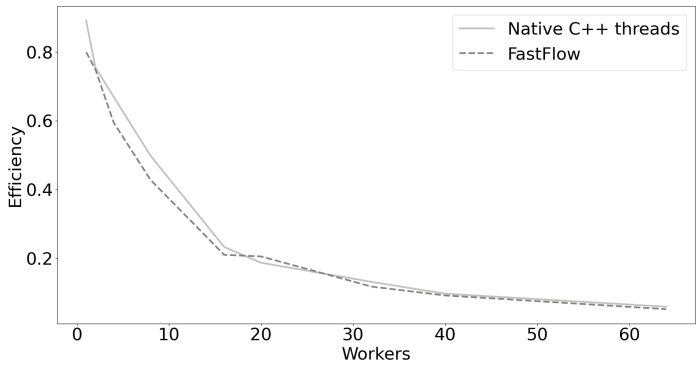


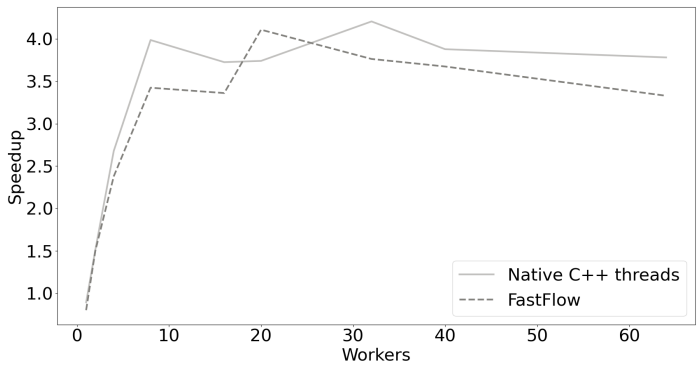




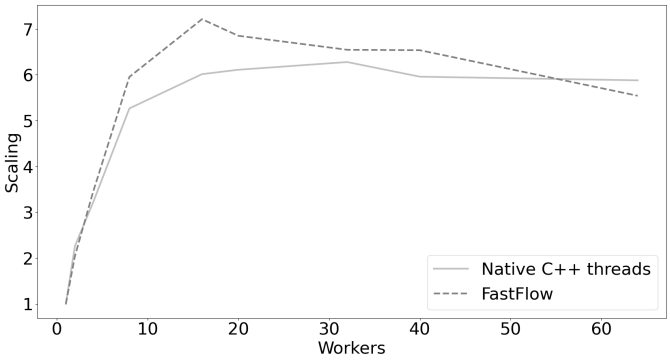
G1: 10k cites, 10k chromosomes

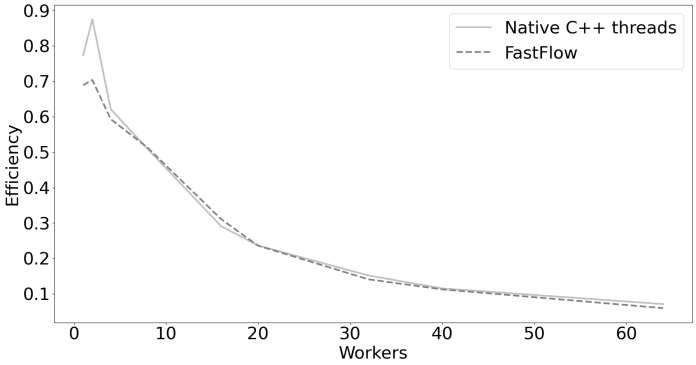


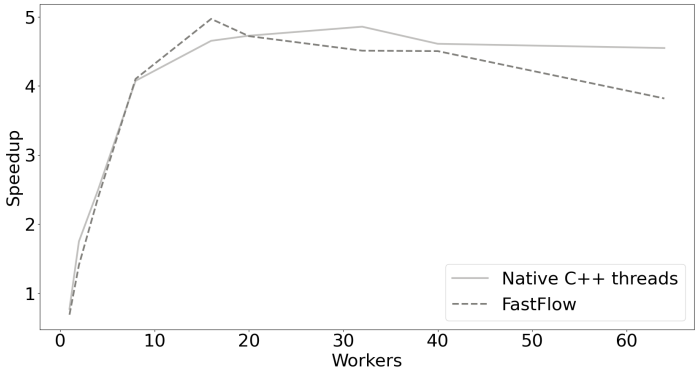




G2: 15k cites, 10k chromosomes







G3: 20k cites, 10k chromosomes

Code snippets

```
calculator->generateFirstChromosomes();

calculator->sortChromosomesByFitness();

int iterationNumber = 1;

while((iterationNumber < maxIterations))

{

    calculator->calculateChromosomesCrossover();

    calculator->calculateChromosomesMutations(mutationPercentage);

    calculator->evaluateAndSortChromosomes();

    iterationNumber++;

}

return calculator->getBestPathLength();
```

Code snippet 1

References

[1] Aldinucci, M. , Danelutto, M. , Kilpatrick, P. and Torquati, M. (2017). Fastflow: High-Level and Efficient Streaming on Multicore. In Programming multi-core and many-core computing systems (eds S. Pllana and F. Xhafa).

[2] Üçoluk G., (2002). Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation