Parallel Implementation of Genetic Travelling Salesman Problem

Parallel and Distributed Systems: Paradigms and Models

Giacomo Cignoni, Mat: 581112

Academic Year 2022/2023

UNIVERSITÀ DI PISA

1 Introduction

The aim of this project consists in developing a solver for the Travelling Salesman Problem (TSP) using a genetic algorithm in C++ and parallelize it using 2 different implementations, using native C++ threads and using the FastFlow library.

1.1 Travelling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic optimization challenge in computer science and mathematics. It involves finding the shortest possible route that visits a given set of cities exactly once and returns to the starting city, minimizing the total distance traveled.

In more mathematical terms, it is a combinatorial optimization task defined as follows:

Given a set of n cities represented by their coordinates (x, y), the goal is to find the permutation π of the cities that minimizes the total distance traveled along the route. This can be mathematically expressed as:

$$\underset{\pi}{\operatorname{argmin}} \ \text{distance}(\pi(n), \pi(1)) + \sum_{i=1}^{n-1} \text{distance}(\pi(i), \pi(i+1)) \tag{1}$$

where $\text{distance}(i, j)$ is the Euclidean distance between cities i and j and π it is subject to the constraint that each city is visited exactly once.

The objective is to determine the optimal permutation π that results in the shortest total distance traveled.

Approaching the problem in terms of a graph, the Traveling Salesman Problem (TSP) focuses on identifying the shortest Hamiltonian cycle within a complete undirected graph $G = (V, E)$. In this context, vertices V symbolize cities, and edges E symbolize the distances associated with traveling between those cities. Again, the aim is to discover a cyclic path that visits each city precisely once, returning to the initial city, and achieving the lowest possible total distance or cost of travel.

1.2 Genetic Algorithm

A Genetic Algorithm (GA) is a heuristic optimization technique inspired by the process of natural selection and evolution. It involves iteratively evolving a population of potential solutions to a problem by simulating processes like selection, crossover, and mutation. GAs are used to find approximate solutions to complex optimization and search problems across various domains, by mimicking the survival-of-the-fittest principle to progressively improve solutions over generations.

1.3 Cities Data

One of the freely available TSP datasets from https://www.math.uwaterloo.ca/tsp/world/countries.html was used for this project. Each dataset from the source website is a collection of cities from a specific world nation, with great variability among the number of cities for each country dataset. The Zimbabwe dataset was chosen, having 929 cities. For initial trials, the Western Sahara dataset was chosen due to its small size (29 cities).

2 Implementation Algorithm

This section explains the algorithm implementation, focusing on the preparation steps and the details, parts and functioning of the genetic algorithm itself. Details on the parallel implementation are in the following Section.

2.1 Preparation steps

Coordinates of the cities are read from the file and the adjacency matrix of the cities distances is calculated in advance, in order to avoid computation time at each iteration. The cost of the generation of the adjacency matrix is of $O(n^2)$ pairwise distance calculations.

For the progression of the genetic algorithm, at each time step (each generation) two population are kept: the old *parent* population and the new *child* population. Each individual (chromosome) of the population contains its path of the cities (its proposed solution) and the relative fitness to that path.

The parent population is fixed and only the child population is kept generated. At the end of each generation, the two populations are swapped, with the current child population becoming the next parent population (and previous parent population gets overwritten by next child population). The number of chromosomes in the population is fixed generation after generation, it is *num_population*. For these reasons, the initialization of both populations is needed. Each chromosome in the parent population is initialized with a random path among the cities, while the child generation is initialized with empty paths.

To summarize, we have the following preparation steps:

1. **Adjacency Matrix Generation**
2. **Empty Population Initialization**
3. **Random Population Initialization**

2.2 Genetic algorithm implementation

As hinted before, the population of chromosomes is updated entirely at each generation, with the child chromosomes replacing entirely the parent chromosomes. The same process for generating the child population is used for a fixed number of iterations (*num_iterations*). It is composed of the following steps:

1. **Sorting -** The parent population is ordered by descending fitness. This is necessary in order to find the best parent chromosome to select for "reproduction"; they are chosen by taking the first *elitism_rate* of the sorted parent population.

2. **Crossover -** For each new chromosome in the child population, two among the previously selected "best" parents are chosen for passing their "genes" (the path of the proposed solution). The crossover among the two parent paths to generate the child path uses the Ordered Crossover (OX) strategy [1]. In short, the process involves selecting a continuous segment of genetic material (genes) from one parent and preserving the order of these genes, then filling in the missing genes from the other parent, still preserving the order, while avoiding duplicate genes by simply skipping them; in our case each gene is a city from the path. Figure 1 offers a visual example of this algorithm.

3. **Mutation -** Once a new child chromosome has been generated at the previous step, we can subject it to mutations. Mutations are implemented as random swaps among cities in the path. The maximum number of mutations is set as $m = mutation\_ratio \times num\_cities$, and up to m cities are swapped in the child path.

4. **Fitness Calculation -** The finalized child chromosome needs its fitness to be calculated for the sorting step of the next iteration. We define fitness as the inverse of the total path distance:

$$fitness = \frac{1}{\text{distance}(path[n], path[1]) + \sum_{i=1}^{num\_cities-1} \text{distance}(path[i], path[i+1])}$$

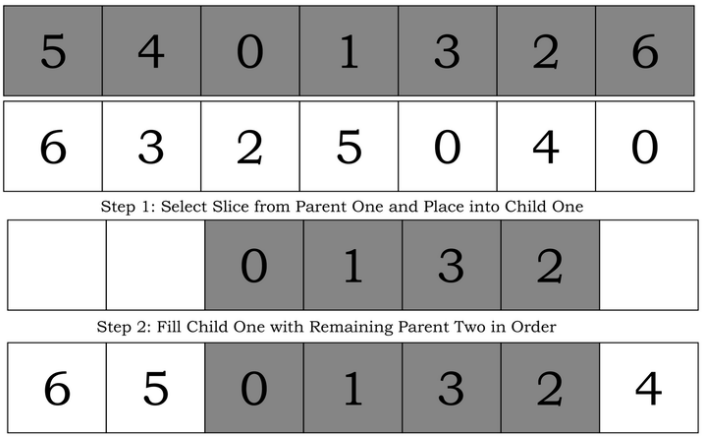


Figure 1: Example of Ordered Crossover algorithm (OX). The two parent chromosomes are at the top of the image.

Correctness of the previously explained genetic algorithm is easily proven in Figure 2, which shows a constantly increasing fitness.

3 Parallelization

3.1 Phases

Before entering in the parallelization strategies details, I analyzed the times employed by the different phases in both the genetic algorithm and preparation steps. This was done by analyzing the execution times of each program phase, while using a strictly sequential strategy. Figure 3a shows that the great majority of the execution time is occupied by the *evolutionTime* which includes steps 2-4 (no sorting) of the genetic algorithm. This is most probably due to the consistent amount of iterations done for simulating the evolving generations.

Excluding the *evolutionTime* in Figure 3b, we have a clearer view of the remaining execution times of the other phases. *otherTime* includes the sorting step (1.) of the evolution process.

For sake of completeness, Figure 3c includes the internal times of *evolutionTime*, that are the individual execution times of genetic algorithm steps 2-4.

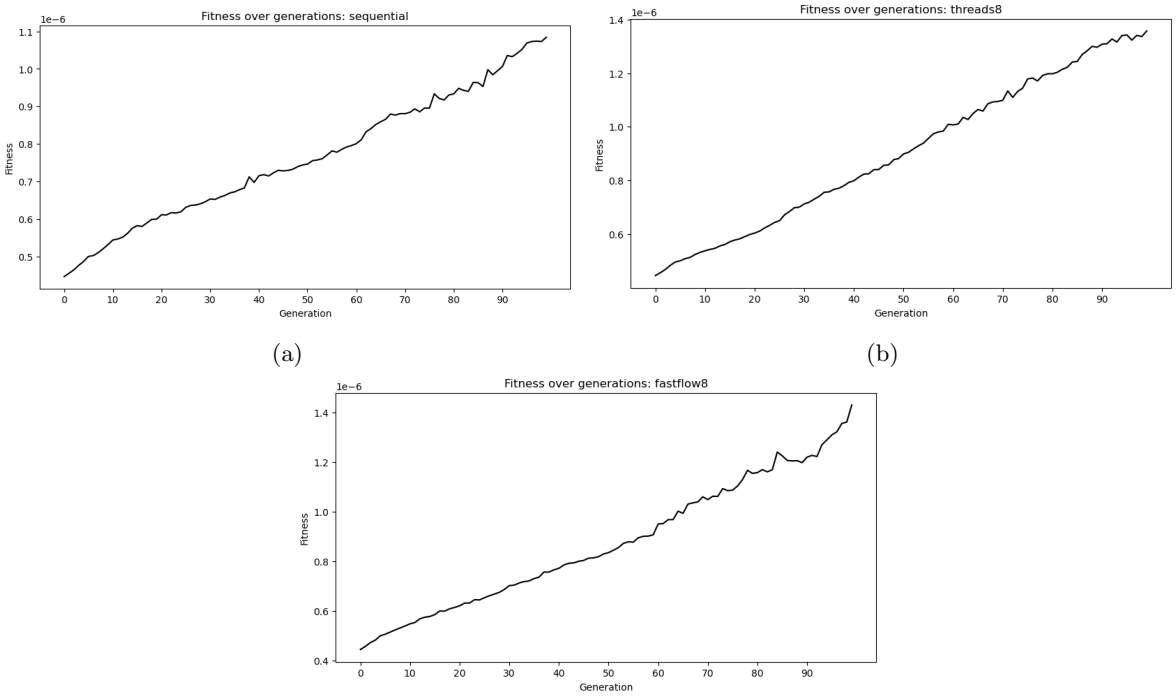
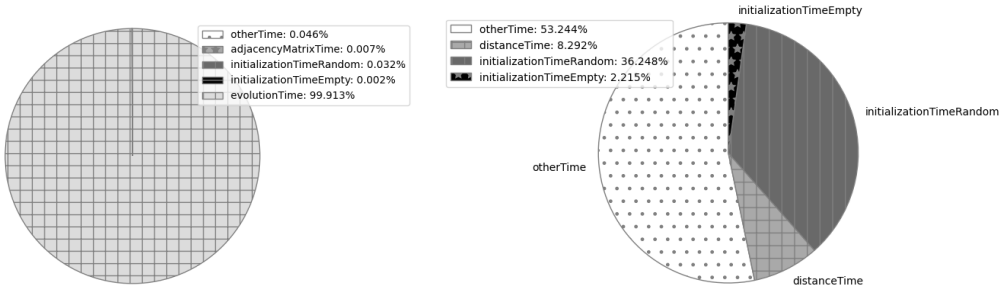
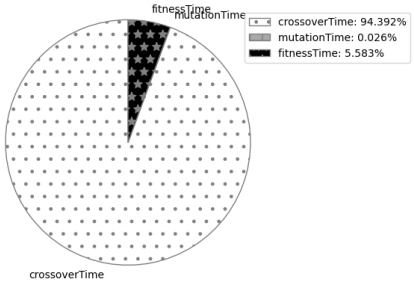


Figure 2: Best fitness across generations for sequential (a), threads (b) and FastFlow (c) implementations.



(a) Ratios of the execution times of the various phases of the program

(b) Ratios of the execution times of the phases, excluding *evolutionTime*



(c) Internal times of *evolutionTime*

Figure 3

3.2 Parallelization strategy

The subdivision in clearly defined phases in the previous Section, allows the parallelization of each task independently.

First of all, due to the working of the algorithm, we recognize being in a data-parallel scenario, as there are no streams of data. The preparation steps are clearly data-parallel, while the genetic algorithm is data parallel as all the parent population has to be completed before generating the child population. Each parallelization phase considers the individual chromosomes as the minimum decomposition of the collection to parallelize, except for the parallelization of the adjacency matrix which considers rows of the matrix (as it is assumed that $num\_cities << num\_workers$).

Moreover, parallelization is possible only inside each iteration of the genetic algorithm, as each generation is fully dependent on the previous one: we cannot generate chromosome from different generations at the same time.

The independent parallelization of the phases follows a fork-join model [2], in which parallel part of the computations alternate with sequential ones. To summarize, the following four phases are parallelized: **evolution** (crossover, mutation, fitness), **adjacency matrix generation**, **empty population initialization**, **random population initialization**.

3.3 Native C++ threads

The parallelization using the C++ native threads was implemented following a map parallel pattern. A `ParallelMap` class was implemented, which role is to divide an `input` collection in *num_workers* chunks (using a static chunking strategy), then to spawn a new thread (`std::thread`) for each chunk that applies a function f to each item in the chunk. The `ParallelMap` class also has the option to save the return value of f in a target `output` collection.

3.4 FastFlow

The FastFlow implementation of the parallelization makes uses of the `parallel_for` construct, initialized with *num_workers*.

The choice of this parallelization approach was made for its intuitivity: being in a data parallel context, the loops over the collections in the sequential variant of the program were easily parallelized using `parallel_for`.

4 Experiments and Results

4.1 Setting of experiments

For the experiments on the Zimbabwe cities dataset, I used a fixed *num_population* of 1000, and the number of generations was fixed to 100. Moreover, I set *elitism_rate* and *mutation_rate* respectively to 0.1 and 0.02.

The program was run initially in its sequential variant, then in both parallel variants, with even number of workers varying from 1 to 32. Times of execution were recorded both of the total execution time and of each of the phases of the computation. All experiments were run on the course's NUMA multicore machine. The code was built using CMake.

4.2 Metrics

In order to evaluate the performance of the parallel implementations, two important metrics are used:

- **Speedup:** is the measure that compares the execution time of the sequential program with the execution time of the parallel program. It indicates how much faster a parallel program is compared to its sequential counterpart. The speedup function is denoted by n, which is the parallelism degree of the parallel program (*num_workers*). The speedup formula is the following:

$$\text{speedup}(n) = \frac{T_{seq}}{T_{par}(n)} \tag{2}$$

with T_{seq} being the execution time of the sequential program and $T_{par(n)}$ the execution time of the parallel program with parallelism n.

- **Scalability:** measures the efficiency of the parallel program when increasing its parallel degree. It indicates the performance improvement of the parallel program with n as parallel degree in respect to 1 as having parallel degree. The scalability formula is:

$$\text{scalability(n)} = \frac{T_{par}(1)}{T_{par}(n)} \tag{3}$$

- **Efficiency:** it compares the performance of the parallel program with parallelism degree n with the ideal parallel execution time. The ideal time with parallelism degree n is defined as:

$$T_{id}(n) = \frac{T_{seq}}{n} \tag{4}$$

so we can define the efficiency as:

$$\text{efficiency}(n) = \frac{T_{id}(n)}{T_{par}(n)} \tag{5}$$

4.3 Results

Table 1 shows the execution times and measures of both parallel implementation with varying parallelism degree.

	Threads		**FastFlow**	
numWorkers	Exec. Time	Speedup	Exec. Time	Speedup
1	291s	1.00	291s	1.00
2	153s	1.90	186s	1.56
4	91s	3.22	112s	2.61
6	77s	3.80	96s	3.03
8	82s	3.54	90s	3.24
10	**55**s	**5.30**	60s	4.89
12	58s	4.98	44s	6.66
14	67s	4.31	**42**s	**6.98**
16	76s	3.81	67s	4.35
18	79s	3.68	63s	4.60
20	85s	3.41	77s	3.80
22	88s	3.30	52s	5.59
24	84s	3.47	49s	5.93
26	88s	3.30	46s	6.37
28	83s	3.51	60s	4.84
30	85s	3.43	78s	3.72
32	85s	3.41	55s	5.29

Table 1: Results of total execution times and speedup for sequential and parallel implementations of the program (threads and FastFlow) with varying parallelism degree. Best time and speedup in bold.

Figure 4 is plot of the the total execution times for sequential, threads and FastFlow implementations of the program, with varying parallelism degree.

Figure 5, instead plots the speedup of the parallel implementations and compares them to the *ideal speedup*. The ideal speedup is calculated using Amdhal law's upper bound on the speedup:

$$\text{speedup}(n) \leq \frac{T_{seq}}{fT_{seq} + (1-f)\frac{T_{seq}}{n}} \tag{6}$$

with f being the inherently sequential and not parallelizable portion of the program. As previously seen in Figure 3a, the program execution times are largely predominated by $(1-f)$; we can then approximate f to 0 and obtain an estimated (slightly lower) speedup upper bound of n.

Figures 6a and 6b, instead, are plots respectively of the scalability and efficiency of the parallel implementations.

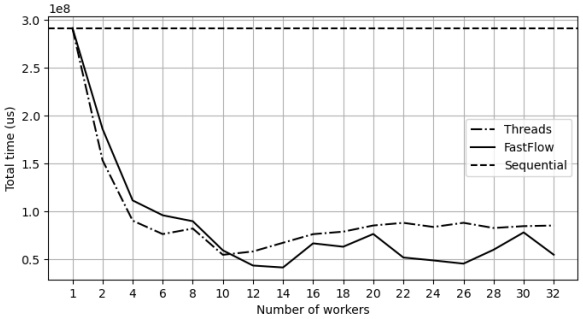


Figure 4: Total execution times for sequential and parallel implementations, with *num_workers* varying from 1 to 32.

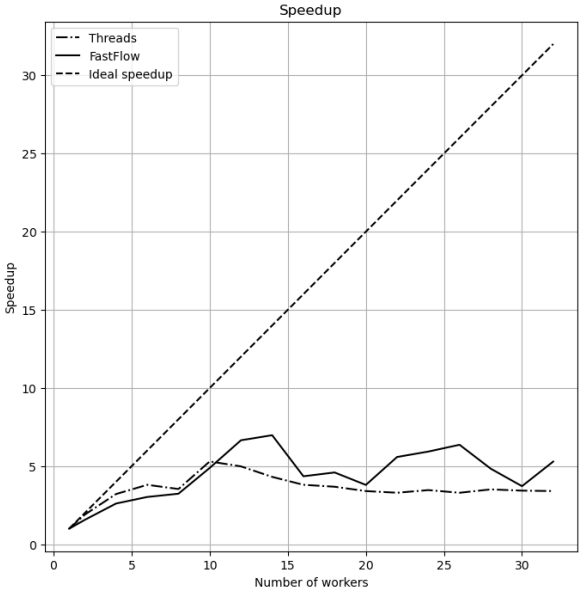


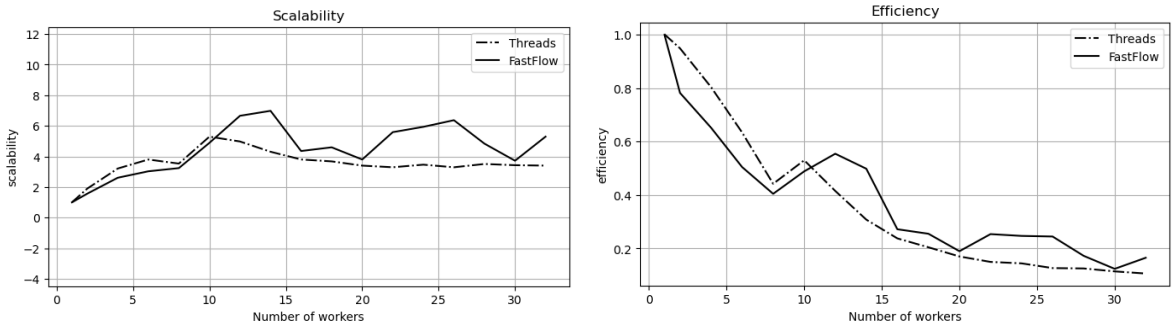
Figure 5: Speedup of parallel implementations, with *num_workers* varying from 1 to 32. Ideal speedup is also shown.

4.4 Considerations

Looking at the plots, it is easy to note that the benefits of parallelization are much higher with a lower parallel degree for both implementations; the speedup plots distinctly diverge from the ideal speedup for *num_workers* > 4. Nontheless, improvement in execution times and speedup are reached up to a parallel degree of 14. Beyond that point, the performances begin to slowly drop, as the overheads introduced by parallelism grow.

It is important to analyze also the difference in performances between native C++ threads and FastFlow implementations. The first has slightly superior performances with lower parallalelism degree, but the second gains the upper hand with *num_workers* > 10 hand reaches the highest speedup of 6.98 with 14 workers. FastFlow also seems to suffer less from increasing overheads with high *num_workers*. Overheads in the threads implementation are most probably due to the static chunking method, the creation of a vector of `std::thread` at each execution of the map and the wait for threads to join.

Another consideration is on the similarity between the scalability and speedup graph: they are practically identical, as there is no significant overhead in the computation when using a parallel degree of 1.



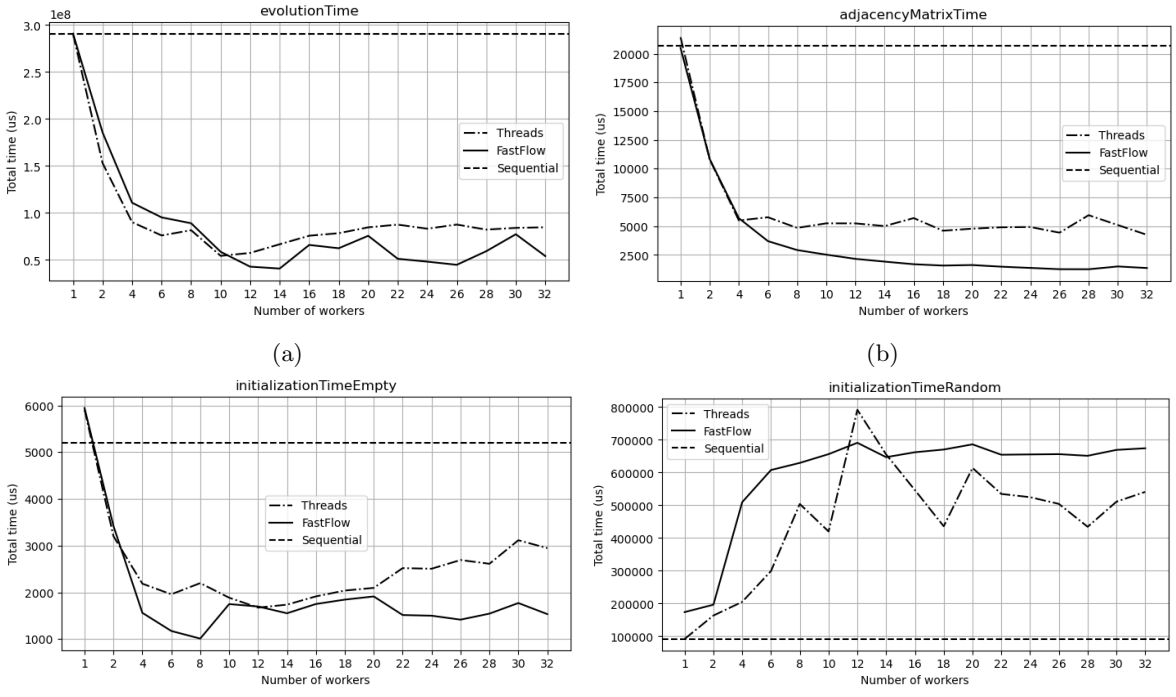
(a) (b)

Figure 6: Scalability (a) and efficiency (b) of the parallel implementations, with *num_workers* varying from 1 to 32.

4.5 Analysis on separated phases

Figures 7 and 8, contains independently the execution times and speedups of the four parallelized phases of the program. It is easily noticeable that the plots relative to the evolution phase are almost identical to the ones relative to the entire program, as it takes the great majority of the execution.

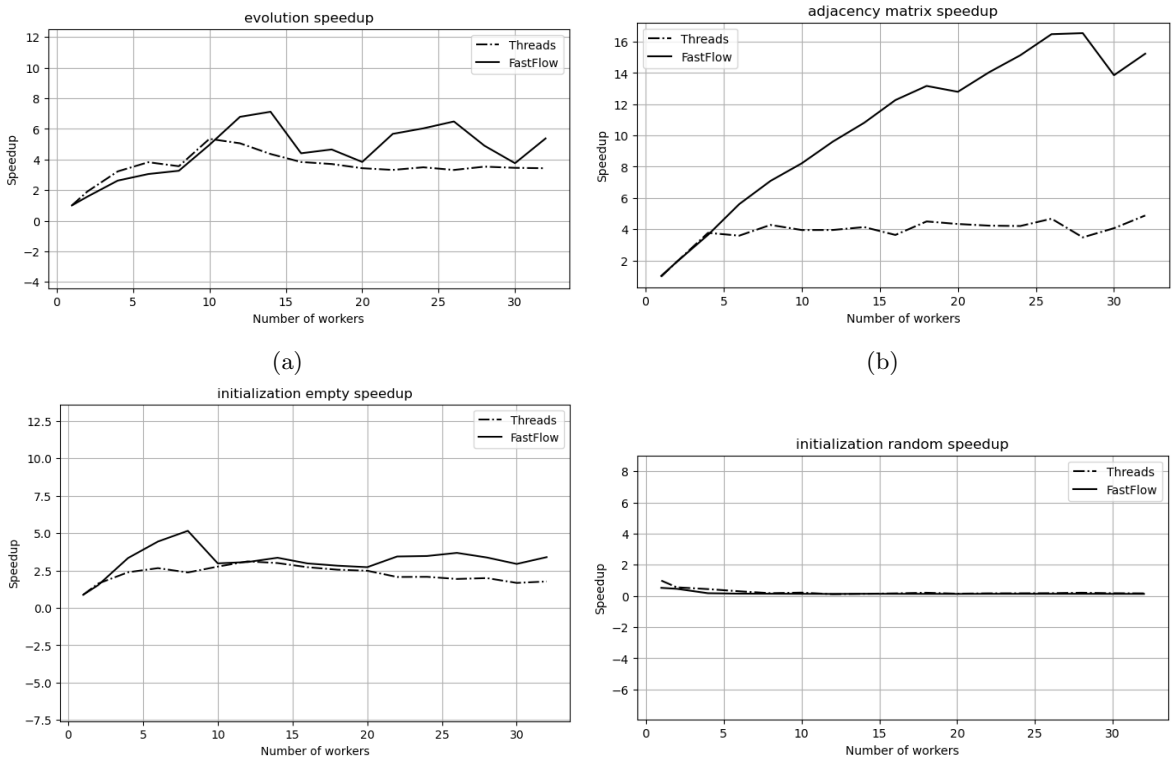
Parallelization of the other phases has practically no influence on the performance of the entire program due to their relative shortness in term of execution times, nonetheless this analysis can be interesting. Regarding the other phases, it is worthy of note that the adjacency matrix phase benefits much more from a parallelization using FastFlow than native threads, especially with an higher parallelism degree. Instead, the parallelization of the random population is not beneficial, as execution times greatly increase when introducing more workers.



(a) (b)

(c) (d)

Figure 7: Execution times of different parallelized phases of the program.



(a) (b)

(c) (d)

Figure 8: Speedups of different parallelized phases of the program.

References

[1] Lawrence Davis. *Handbook of genetic algorithms*. New York: Van Nostrand Reinhold, 1991. ISBN 0–442–00173–8.

[2] Wikipedia. Fork–join model — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Fork%E2%80%93join%20model&oldid=1157279085, 2023. [Online; accessed 23-August-2023].