University of Pisa, Master Degree in Computer Science
Course of "Parallel and Distributed Systems: Paradigms and Models"

Parallelizing the Genetic Travelling Salesperson Problem

Final Project, A.Y. 2020/2021

Davide Barasti - 607003

Gen 2022

Summary

The Travelling Salesperson Problem (TSP) is a well-known optimization problem requiring to find the shortest possible path visiting a list of nodes exactly once. Being the TSP a NP-hard problem, a common solution is to find a "good enough" path using some optimization algorithm. One of the flavours of the TSP is to use a genetic algorithm to find a solution. In this report, I describe two implementations of the genetic TSP, parallelizing the stages of each generation of the algorithm: the first parallel implementation uses unstructured, low-level mechanisms of C++ (*futures* and *async*), whereas the second exploits the FastFlow library, approaching the problem in a more structured way. As a baseline for the performance evaluation, I developed a sequential version of the genetic algorithm as well. After a description of the implementations, I used a performance model to evaluate and compare the proposed solutions.

1 The Genetic Algorithm

Genetic algorithms are based on three basic principles that need to be present in every solution. These are called *Darwinian natural selection principles*:

- **Heredity**: traits that give individuals a reproductive advantage are passed from parent to offspring;
- **Variation**: throughout generations, slight changes in the traits occur;
- **Selection**: if variations are useful, they are selected and preserved through heritable traits.

In practice, the genetic algorithm implemented, whose base logic is common to all three implementations, works as follows: the first part is executed only once, and consists in the random generation of the initial population. Each element of the population is referred to as *chromosome*, and consists in an order in which the cities are visited. The second part is iterated numerous times and is made of:

1. **Lower bound computation**: this value is used later to compute the *fitness score* of each chromosome. It is an approximation of the shortest path for the current population;
2. **Evaluation**: measures the performance of each chromosome in the population with respect to the total length of the path. Each chromosome evaluation is independent of others;
3. **Fitness score computation**: transforms the measure of performance into an allocation of reproductive opportunities. This value is defined with respect to the whole population. It is computed as a function of evaluation and lower bound;
4. **Selection**: an intermediate population is selected, elements with higher fitness score are more likely to be included in the intermediate population;
5. **Crossover**: the next generation is created from the intermediate population. Crossover is applied to randomly-paired strings with a parametric probability. The selected pairs are recombined into a chromosome, and become part of the next generation;
6. **Mutation**: a mutation operator is applied to the chromosomes with a parametric probability. Mutation is fundamental, especially during the initial iterations, to introduce variation in order to effectively evolve the population.

2 Implementation Details

This section summarizes the design choices that have brought to the final version of the program. I iterated through several ideas, some of them still available as git branches, and the implementation presented is what I reached after several trial and error cycles. The implementation details, as they describe the main logic of the algorithm, are valid for both sequential and parallel versions. To represent the cities that the traveller visits, I used a vector of `Point`s. Each point represents coordinates (x and y values) on a plane. Each chromosome composing the population is made of a vector of integer numbers representing the order in which the cities are visited. Hence, the points representing the cities are not altered during the execution; what changes in the population is solely the order of visit. Figure 1 helps to visualize the concept of "moving orders" instead of cities. The order of visit is indicated left-to-right inside the i-th chromosome. The arrows indicate exactly the order of visit imposed by the example chromosome.

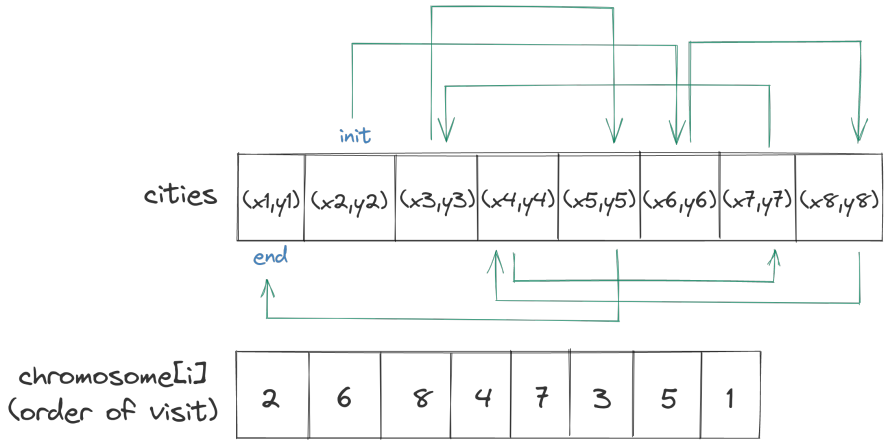


Figure 1: Chromosomes represent the order to use when visiting the cities.

The evaluation for each chromosome simply consists in the total path length that the chromosome imposes. Then, the program computes a fitness score for each chromosome. The formula used is the following:

$$fitness[i] = \frac{LowerBound}{evaluation[i]}$$

The selection phase uses the fitness values to perform a weighted selection, producing the intermediate population. The higher the fitness score of a chromosome, the higher the probability for it to be inserted in the intermediate population. To obtain the next generation, the crossover phase is executed with probability P_c for each chromosome. Given two randomly selected mates, the recombination produces a chromosome having traits of both mate A and mate B. Finally, a mutation is applied to each chromosome with probability P_m. The mutation, if applied, randomly swaps two elements of a chromosome. To summarize, a high-level pseudocode of the algorithm can be:

```
geneticTSP(){
    generateCities()
    generateInitialPopulation()
    for(gen in generations){
        lb = calculateLowerBound()
        eval = evaluate()
        fitness = calculateFitness(evaluation, lb)
        selection(fitness)
        crossover()
        mutate()
    }
}
```

The pseudocode is actually a good approximation of the `run` method of the `SequentialTSP` class. To validate the functional requirement of the implementation, meaning, to verify that the genetic algorithm is able to find better solutions as time advances, I collected and plotted the data relative to the average evaluation score for every generation. Figure 2 shows the resulting plot.

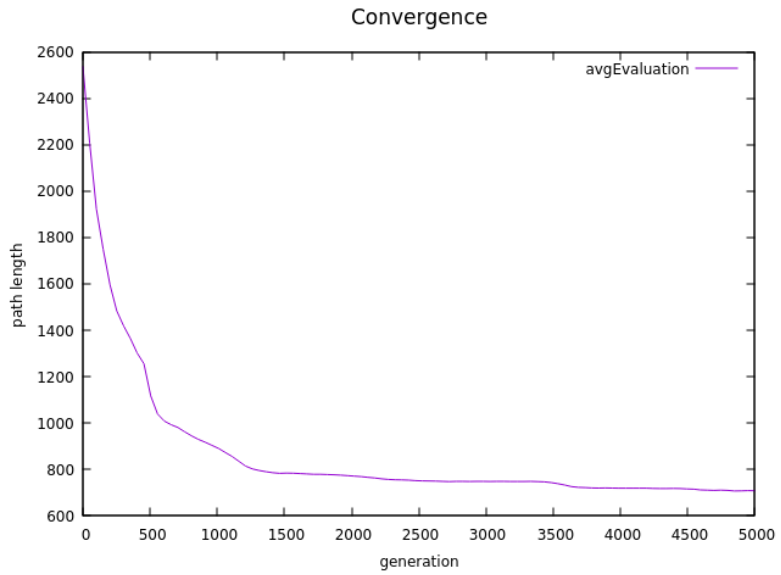


Figure 2: Convergence plot after 5000 iterations with 50 cities and population size of 20000.

To conclude this section, Figure 3 shows how the various tasks impact on the final computation. Where tasks like fitness computation, mutation and selection have low impact on the total computation time, lower bound computation, evaluation, and crossover have a higher overall impact (crossover makes up more than 85%). The analysis does not include the initialization phase, as it is executed sequentially, only once, at the beginning of the algorithm.

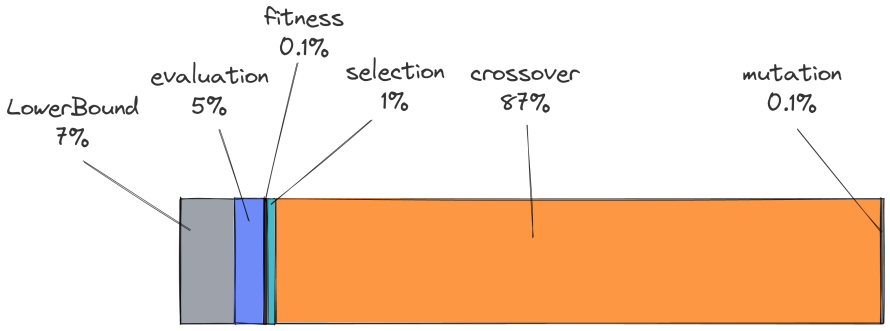


Figure 3: Task impact on computation time.

3 Parallel Implementation

Once I achieved satisfying results in the base implementation, I proceeded with the development of the parallel versions of the algorithm. The standard threads and FastFlow approaches differ in a very precise way: the former is obtained using unstructured parallel programming, whereas the latter makes use of the high-level parallel patterns of the FastFlow library, allowing for a structured approach to the parallel implementation.

3.1 Standard threads

With a solid sequential implementation, the first parallel version can be described as an increment over the base version. In fact, the main logic remains the same, but the code to handle the parallelization is now needed. To avoid using and managing C++ threads directly, I opted for a modern C++11 solution, that makes use of `std::async` and `std::future` to compute parallel jobs. The core difference stands in the fact that the work to be done at each stage will be divided in chunks, that will be executed by a pool of workers available at runtime. The approach follows a fork/join model. A core part of the algorithm is the *setup* phase, that determines how big each chunk will be. Specifically, a chunk refers to a pair of indexes, representing the section of the population on which the worker will operate. The chunk size is a function of the population size and the amount of resources (threads in this case) that are made available by the user. The chunk size is fixed, as the amount of work (the heaviness) of each task is the same; hence, the solution applies static scheduling over the tasks. Then, each main phase of the genetic algorithm, instead of computing the work on the whole population, will assign to each worker an asynchronous task; these tasks will be awaited before the phase terminates. The standard threads implementation can be visualized in the schema of Figure 4. The *fitness computation* and *mutation* phases remained sequential, as little or no speed-up could be achieved due to the low impact of the tasks on the overall completion time (refer again to Figure 3).

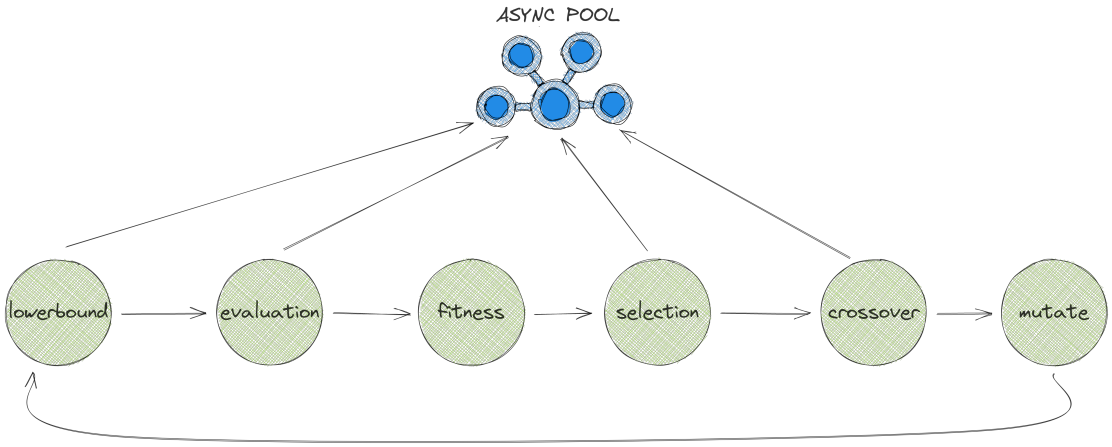


Figure 4: Diagram of the standard threads parallel implementation.

The use of async allows to implicitly optimize the use of computing resources, as the decision on whether to spawn a new thread for any new task, or to use an available worker from a thread pool, is delegated to the async library. Also, thanks to the mechanisms of async, there is no need for explicit mutexes or messaging: all the synchronization between the thread launching the async task and the background thread is performed behind the scenes. When the initial thread requests to access the future-value, it is automatically paused until the value is ready. Of course, the fact that the user has no control over the amount of threads created to run the functions, and, in general, how many async functions are being run, can become an issue. A not-so-complicated solution, as outlined in [1], would require the user to manually manage a thread pool and some work queues, to gain more control over what is happening behind the scenes. In my specific case, the amount of functions to be executed (the tasks) are strictly related to the amount of processing units made available by the program user. As mentioned earlier in the paragraph, the

chunk size is a function of population size and amount of available resources; specifically:

$$chunkSize = \frac{|population|}{nWorkers}$$

Hence, there are exactly `nWorkers` chunks for each phase of the algorithm to be distributed among the same amount of threads. Therefore, with relative confidence, there will not be processor overloading problems in the implementation.

3.2 FastFlow

After experimenting with the unstructured, standard threads approach, I used the FastFlow library [2] with the objective to reach similar, if not better, results, compared to the unstructured approach, this time using a high-level parallel programming framework. The implementation consists in a pipeline of farms, replicating (with some minor changes) the standard threads version: the lower bound computation, evaluation, selection and crossover phases, each have their own farm, and the mutate phase has been merged into the crossover phase, exploiting the fact that the mutation of a chromosome is independent of other chromosomes being mutated. The FastFlow design looks like the schema of Figure 5. Since the problem at hand is solved with a data parallel computation split in five phases, the streaming nature of FastFlow pipelines is forced to a staged data parallel computation, thanks to the collector nodes, waiting for the partial results to be delivered by all the workers before resuming the stream of tasks. The tasks moving around the FastFlow pipeline represent the same chunks of the standard threads implementation: to each worker is passed a pair of indexes that indicate the slice of the problem they need to work on. As said, the collectors/emitters synchronize the work before sending the tasks to the next stage, but some also need to compute the inherently sequential parts of some phases (e.g., the lower bound collector that needs to sort the partially computed results). By default, FastFlow nodes have a busy wait cycle that looks for new tasks available on the communication channels; as this behaviour have resulted in less overall performance, I enabled the `BLOCKING_MODE` flag at compile time. As a final note, to optimize the number of nodes composing the FastFlow graph, I removed the default collectors, apart from the ending one, and made the farm emitters a multi-input node.

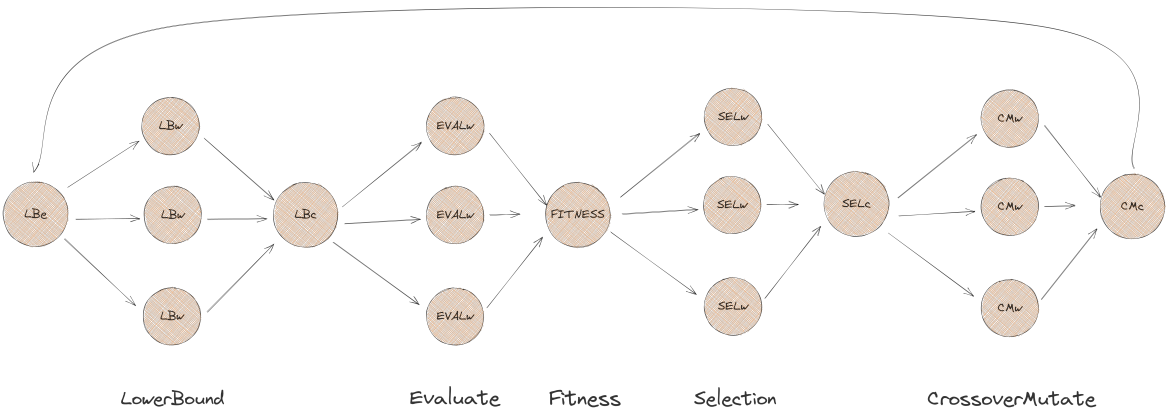


Figure 5: Diagram of the FastFlow parallel implementation.

4 Performance Model

To measure the performance of the proposed implementations and be able to make comparisons, I defined two commonly used indicators: *speedup* and *scalability*. Speedup relates the (best) sequential time to the parallel time obtained using n processing units. The formula is the following:

$$sp(n) = \frac{T_{seq}}{T_{par}(n)}$$

Ideally, the speedup value grows linearly with the amount of resources available; in practice, the measured values are below the ideal speedup. Overhead and inherently sequential code inside parallel programs are the main reasons for non-ideal speedup. I used the Amdahl's law to estimate the maximum speedup obtainable in the three scenarios; for each of these I defined a constant obtained with the formula

$$maxSp = \frac{1}{s}$$

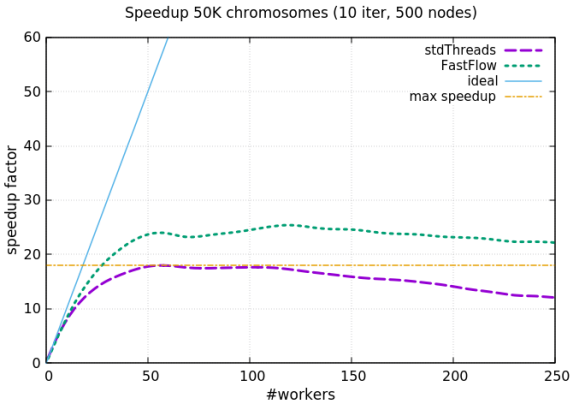
where s is the serial fraction of the code. The speedup, instead, measures how efficient the parallel implementation is, as the parallelism degree becomes larger. The formula is the following:

$$scal(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

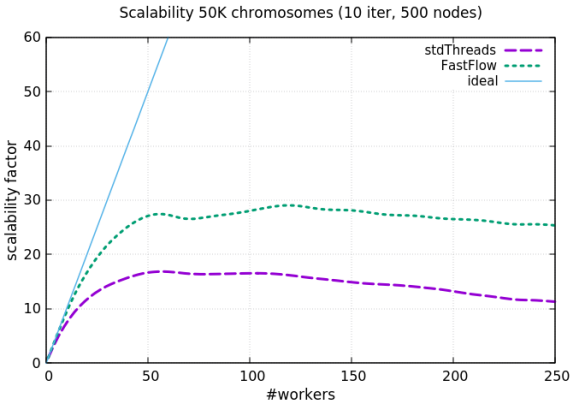
To obtain the timings necessary to compute speedup, scalability and the Amdahl constants, I plugged in the code several meters, enabled by activating the `DETAILED_MEASURE` and `MEASURE` flags inside the `main.cpp` file.

5 Results

The results have been obtained by running a benchmark against the two parallel implementations. I considered three scenarios, each obtained by varying the size of the population. The fixed parameters are the number of cities (500) and the number of iterations (10). 50k, 100k and 200k chromosomes constitute the variable parameter. By analysing the benchmark, the main argument is that the larger the population, the better the performance gets. In fact, in the first scenario (Figure 6), the speedup and scalability factors reach a maximum at around 120 threads, and then start to decrease. In the second scenario (Figure 7), the two factors stabilize at around 60 threads for the standard threads implementation, whereas the performance increases up to 120 threads and only then stabilizes for the FastFlow implementation. Finally, the best performance is achieved with the maximum population size considered. Figure 8 shows that both speedup and scalability have a sublinear behaviour until 50 threads, then continue increasing slightly, without reaching an inflection point. Despite showing similar behaviour, the FastFlow version beats the standard thread implementation in every benchmark, except for the speedup of the third scenario, where the performance shown is practically the same (see again Figure 8). Another point that draws attention is that the FastFlow implementation reaches and exceeds the maximum speedup threshold imposed by the Amdahl's law.

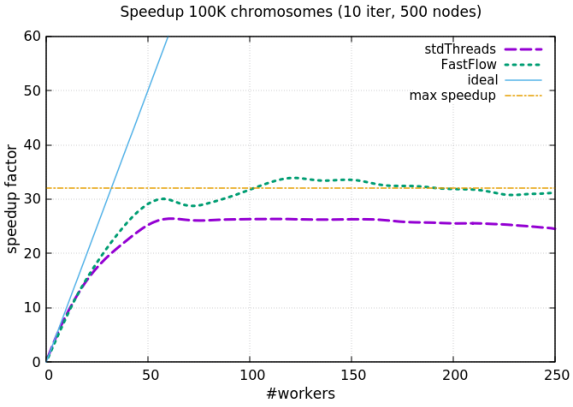


(a) Speedup

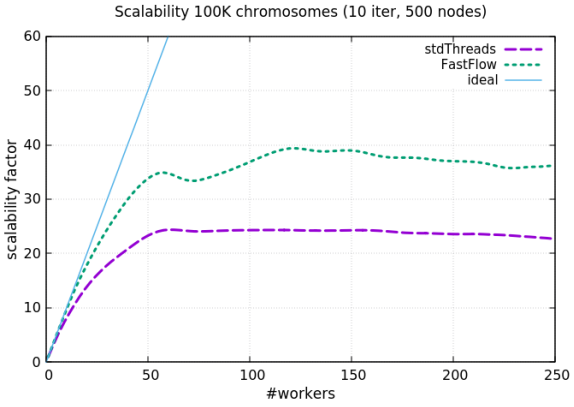


(b) Scalability

Figure 6: Scenario 1: 50k population size, 500 cities, 10 iterations

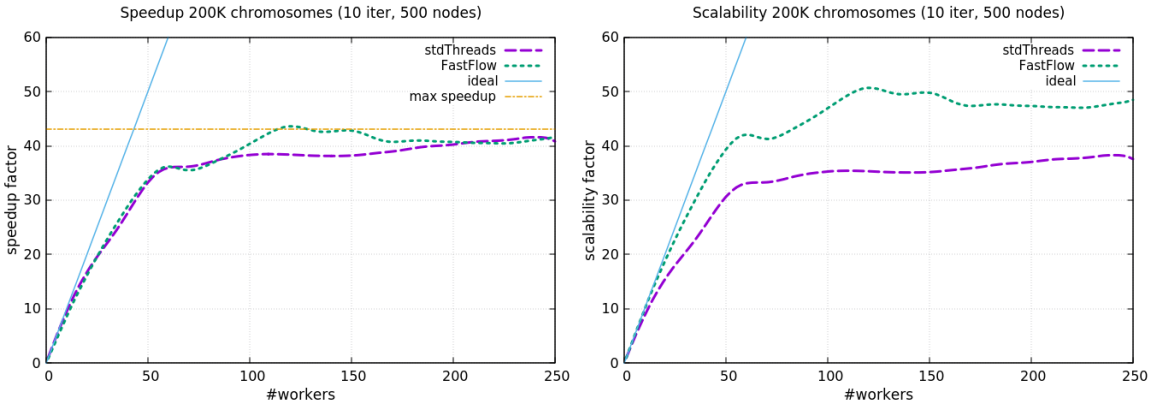


(a) Speedup



(b) Scalability

Figure 7: Scenario 2: 100k population size, 500 cities, 10 iterations



(a) Speedup

(b) Scalability

Figure 8: Scenario 3: 200k population size, 500 cities, 10 iterations

6 Discussion and Conclusion

As shown in the previous section, the performance got better as the population size improved; this result is strictly related to the decreasing impact of the overhead, and the ability of the implementation to adapt to larger data exploiting the amount of resources available. As a consequence, not only the degradation in performance is minimal or none in the first two scenarios (Figure 6 and 7), but it also steadily and continuously improved in the third scenario (Figure 8). As of why FastFlow performed generally better than standard threads, this is possibly due to the optimizations built-in the framework, the efficient communication mechanisms between the stages of the pipeline, and the fine-grained management of threads behind the scenes. The benefits of a well-built framework, where concurrency and communication details are abstracted-away from the user, emerge from both the benchmark results and the programming experience during the development process. Despite the use of async and futures helped with hiding some burdens typical of dealing directly with threads, I found it much easier to use a framework with declarative-style definition of pipeline and farms, and not needing to think about optimizations and low-level tweaks. Finally, in the previous section, I highlighted how the FastFlow implementation surpassed the Amdahl's limit several times. To justify this behaviour, it is important to consider that the limit imposed by the Amdahl's formula is theoretical, and can vary in practice. Also, the formulation does not account for the number of processing resources available to the program. Another, possibly more accurate, speculation on the maximum speedup achievable could have been obtained using the Gustafson's law, that evolves from the Amdahl's theory and brings into the formulation the number of processors, as well as the serial fraction.

7 Building and Running

The project can be compiled and executed using either Cmake or GCC. Before, you may want to edit the `main.cpp` file to alter the implementations to be executed. If CMake is installed, type: `mkdir build && cd build`. Now to build the project type: `cmake --build ..` If you encounter any problem, to use GCC run from the project root folder: `g++ -std=c++17 -O3 main.cpp -o tsp-gen -pthread -I <path/to/fastflow> -DBLOCKING_MODE` specifying the FastFlow path. In order to run the program, you need to pass to the executable the following parameters:

```
nCities populationSize generations mutationProbability crossoverProbability nWorkers
[seed]
```

References

[1] Pierre Baillargeon. *Solving the problems with the "futures" in C++*. URL: https://www.spiria.com/en/blog/desktop-software/solving-the-problems-with-the-futures/. (accessed: 14.01.2022).

[2] Marco Danelutto Massimo Torquati Marco Aldinucci. *The FastFlow library*. URL: http://calvados.di.unipi.it/. (accessed: 14.01.2022).