Genetic Travelling Salesman Problem

SPM Final Project

Simone Marzeddu

Academic Year 2022-2023

1. Introduction

This document contains a report about all the designs and implementations related to the final project for the "Parallel and Distributed Systems: Paradigms and Models" course, Master Degree in Computer Science of the University of Pisa. The chosen project's assignment required three different implementation of a solution for the "Traveling Salesman Problem" exploiting "Genetic Algorithms".

1.0.1 Document's Structure

The rest of the document is organised as follows:

- **Chapter 2**, where is discussed the sequential implementation of the project.
- **Chapter 3**, which inspects the main implementation choices for the parallel design of the program, discussing pros and cons of the possible patterns. The chapter also details the two parallel implementations presented for the project, the first exploiting native C++ threads and the second making use of the library "FastFlow".
- **Chapter 4**, which analyses the empirical performance of the codes, comparing the measures of "Speedup", "Scalability" and "Efficiency" for the two parallel implementations.
- **Chapter 5**, where is presented a brief guide about compiling and executing the code presented in the final submission of the project.

2. Sequential Implementation

The sequential implementation had a key role for identification and analysis of the possible bottlenecks of the algorithm, the study of these features has been, for this reason, mandatory in order to perform the design choices that characterize the two other parallel implementations.

The algorithm implemented is a direct application of the standards of “Genetic Algorithms”, consisting in the phases of “Initialisation”, “Selection”, “Crossover” and “Mutation”, followed by the merging of the generated specimens with the global population, where the new solutions replace the previous worst solutions, according to a fitness function evaluation.

2.0.1 Initialisation and Data Representation

Considering the focus of the project on “TSP”, a specimen (for what concerns the genetic algorithm) consist of a path in the reference graph, which is randomly generated according to the number of nodes expressed as an input parameter at run time. From an implementation point of view, specimens representation in the code exploits the “C++ Struct” named “popMember”. The core internal variables of objects of this type are: “fitness”, where is stored the output of the fitness function on which the path is evaluated, and “specimen”, a vector of integers that represent the order of nodes of which consists the specimen path.

After the generation of the graph of interest (expressed by a adjacency matrix), the population is initialised according to the number of specimens requested by program input. These starting specimens are randomly generated and stored in a **ordered** vector of “popMember” representing the population. The seed exploited is fixed and input independent.

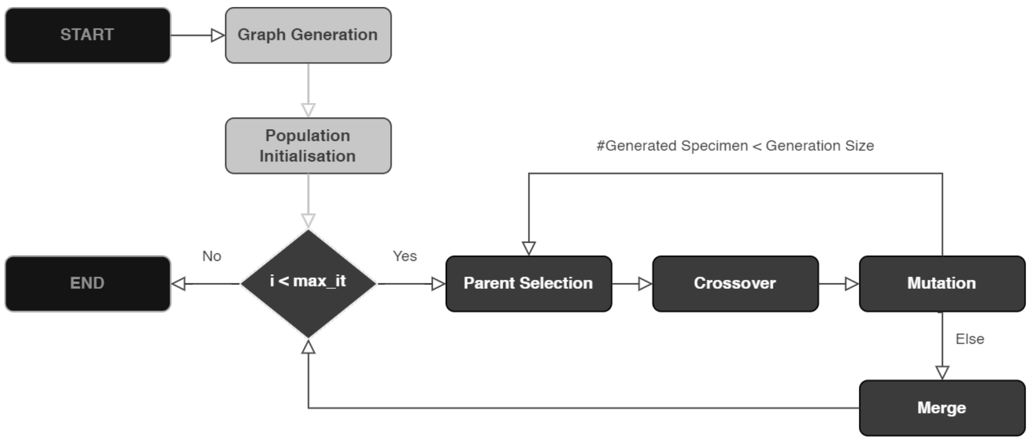
2.0.2 Iterations and Genetic Algorithm

The core of the implementation consists of two nested loops (a for and a while loop). The for loop iterates according to the number of generations on which the genetic algorithm is initialised. The while loop, nested in the former, implements the phases of “Selection”, “Crossover” and “Mutation”, iterating until the number of new specimens generated coincides with the required new generation size (input parameter). All the specimens generated by these phases are stored in a second vector of “popMember”s. After the execution of the while loop is terminated, the elements of this children specimens vector are swapped with the worst evaluated population members and, after the population have been rearranged respect to the fitness of its specimens, the for loop will continue its iteration, leading to the begin of a new generation from the genetic algorithm point of view.

2.0.3 Crossover and Mutation

The crossover that occurs at the generation of each pair of specimens consists of the exchange of two portions of the vectors representing the paths of the two “popMember”s. Given this implementation, as the paths resulting from this operation may contain duplicates, a correction is required in order to produce significant specimens to represent “TSP” solutions. The correction takes place as the substitution of all the duplicate nodes in the generated path with the nodes which are not found in the former, so that at the end of every iteration the resulting specimens’ paths appear as permutations of the integers from 0 to n (size of the reference graph) without repetitions..

Considering, therefore, that all the paths produced after a crossover are meaningful specimens, the mutation simply consists of the swapping of two nodes in the given path. Mutation is performed on a random basis according to a mutation rate established during initialisation.



3. Design and Implementation

The design of the parallel applications has been strongly guided by the empirical evaluation of the execution times of the sequential implementation. What emerged from this first analysis are mostly insights about the relative time of execution for each of the phases of the genetic algorithm. As an example, referring to a case of execution with a graph of 7000 nodes, a population of 10.000 specimens and a reproduction rate of 70%, considering the time consumed by the fastest phases as 1U (one **"Unit of Time"**), the relative times can be approximated as: **1U** for the (fastest) "Selection" phase, **3U** for the "Crossover" phase, **6U** for the "Correction" phase, **0U** for the "Mutation" phase (which for implementation reason is always negligible compared to the other phases) and **43U** for the "Evaluation" phase (calculation of the fitness function for the generated specimens).

Considering this data, calculating the times obtained with different input sizes and looking at the code structure itself, facts emerge that must be taken into account during the design phase:

- **"Evaluation" phase is a bottleneck**: The evaluation generally has a larger time cost than the other phases of the algorithm.
- **Different complexity of phases**: the genetic algorithm phases have different time complexities, fact that makes some phases relative times different depending on the input size.
- **Stochastic variability**: the execution times of "Selection" and "Correction" phases are subject to randomness.
- **Intergenerational dependence**: the genetic algorithm implemented is founded on the invariant for which one generation (iteration of the algorithm) can only be computed after the completion of the previous. The need to include new specimens in the population only at the end of the generation creates a natural "barrier" between iterations.

On this basis, the three general patterns analysed for the parallel implementations of the project are "Pipeline", "Farm" and "Map".

3.0.1 Pipeline

In addition to the overheads generated by the communication between the various stages, the pipeline model suffers from all the insights listed above. Indeed, the temporal variability generated by points two and three would lead to the creation of inconsistencies ("bubbles") in the pipeline flow. A similar problem would then be generated by the fourth point and, since at each iteration of the algorithm the pipeline would be subject to a new inconsistency, it is inferred that the pipeline would fail to maintain a stable flow regime. Due to the bottleneck found in the "Evaluation" stage, finally, the pipeline model would require a generous number of workers in order to be able to relieve its cost (implementation of the stage via a pattern farm).

3.0.2 Farm

Consider the model of a farm in which an emitter thread assigns pairs of specimens for the construction of the next generation to a number of parallel workers, while a collector gather their output before updating the population at the end of each iteration. The system would suffer from the already analysed “intergenerational dependence”, as all the threads not involved in the population update (merge) will necessarily have to wait for the end of the operation, however, this would not damage the model performance as much as in the previous pipeline case. The farm would also be subject to the overhead deriving from the emitter-workers and workers-collector communications. Also, the presence of this two active threads (which can be avoided in other patterns) reduces the total number of workers that can be supported by the machine.

Through special policies, such as the ”job stealing” technique, this model allows optimisable workload balances between the various workers.

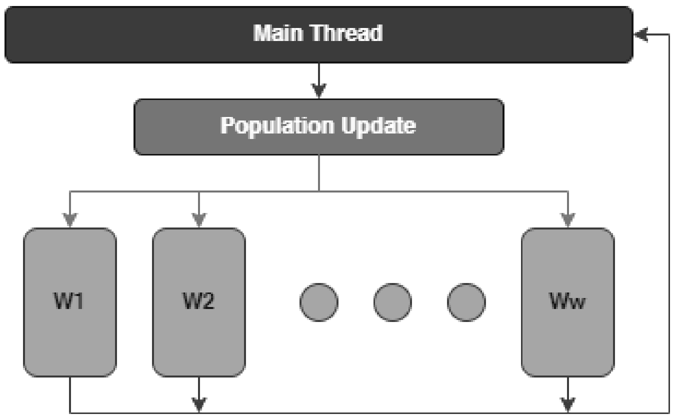
3.0.3 Map

Given the previous analysis, it is possible to consider an iteration of the genetic algorithm as the application of a map function on the population, which given as input the i-th generation returns the i+1-th generation. The idea behind a map would therefore be to define the number of specimens to be produced for each worker during the initialization phase, so that they can execute independently and consistently. This model would retain the advantages of the previous case, keeping the workers loads naturally balanced in the long run thanks to the fact that the large input size required by genetic algorithms for good performance would distribute between different threads the computations affected by random fluctuations during the generation of each new specimen, bringing worker execution times towards a common convergence. At the same time, this model loses most of the disadvantages deriving from the overheads of internal communications present both in pipelines and farms for the analysed cases (a more detailed study follows in the next section). For these reasons, during the design phase, the implementation of an high level ”map-like” pattern appeared as an effective model for the parallel implementation of the algorithm.

3.0.4 C++ Native Threads Parallel Implementation

Exploiting C++ native threads, the implementation consist of a map similar to the aforementioned one. In particular, the main thread initialise the first random population, forks the workers’ threads and implements what was (in the sequential implementation) included in the external for loop (updating the population after every generation). The workers implement what before was encapsulated in the internal while loop (“Selection”, “Crossover”, “Correction”, “Mutation” and “Evaluation”, phases of the genetic algorithm).

Synchronising thanks to a condition variable, workers wait for the population to be updated by the main thread at the start of each iteration, and thanks to a barrier, instead, the threads wait for all the workers to finish their execution. The main thread will then update the current population, according to what is produced by workers (mappers). In addition to what is introduced by the condition variable (with associated lock operations) and the barrier, both only respectively at the beginning and at the end of each iteration, there are no other direct sources of overhead, as the workers exploits a shared array where their results are inserted without competition, knowing from initialisation their own dedicated vector partitions.



3.0.5 “FastFlow” Parallel Implementation

The second parallel version of the code developed exploiting the “FastFlow” library (open source on GitHub at the link: `https://github.com/fastflow/fastflow`) respects analogues assumptions and shows an implementation similar to the former. In particular, the personalised map pattern is built through an object of type “ff_Farm”, initialized with a pool of “ff_node” (workers) and a custom (multi-output broadcaster node) emitter, implemented through an object of type “ff_monode”.

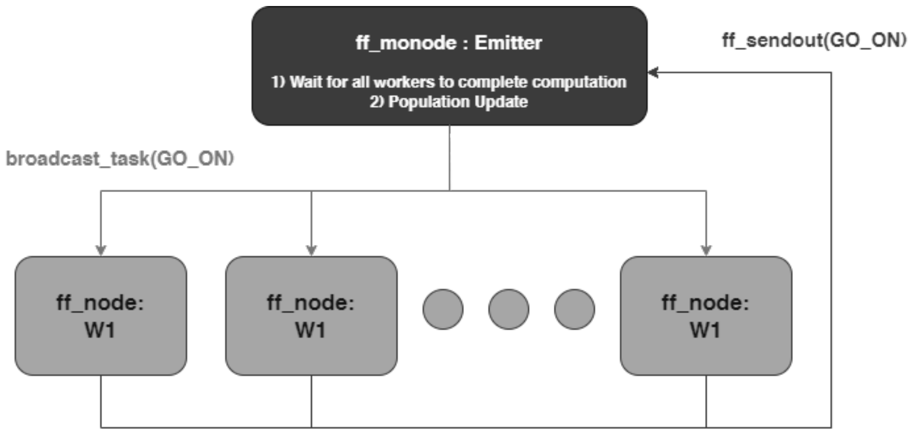
During initialisation, on the “ff_Farm” object is called the method “wrap_around()”, thanks to which the workers’ outputs are redirected to the emitter.

The emitter node acts like the main thread of the previous section, waiting for the end of workers’ computations in order to perform the population update and broadcasting the “GO_ON” task in order to restart worker execution for the next generation.

In a similar manner, a worker’s output communication consists of calling the “ff_-sendout(GO_ON)” method, notifying the emitter that the activity of the current node is completed.

As in the C++ native threads implementation, the exchange of data (specimens generated by workers) between threads never happens directly and does not directly generate overhead. Again the threads work on a shared array but in different indexes ranges (defined in the program initialisation phase) without ever accessing the same memory locations.

The emitter node will maintain the iteration count, propagating the “EOS” to the workers once this is sufficient for the termination of the algorithm.



4. Empirical Analysis

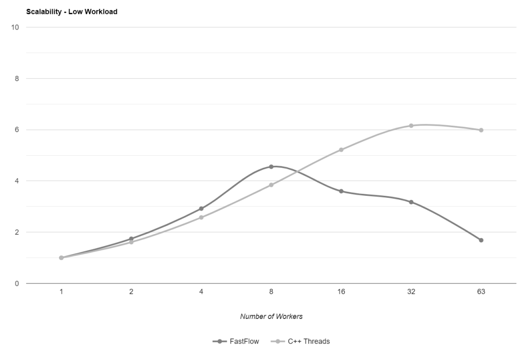
With the aim of evaluating performance, all implementations discussed in this report were subjected to experiments in the virtual machine. These empirical tests were designed so that data could be collected on three different categories of execution:

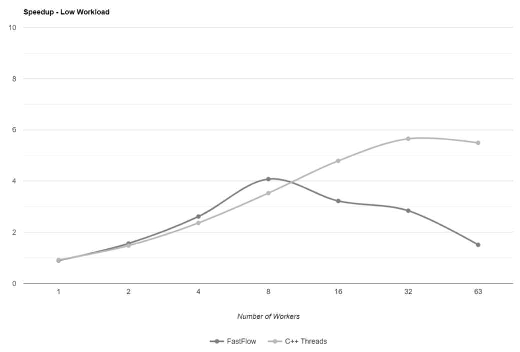
- **Low Workload**, with a graph of 500 nodes, a population size of 2500 and 1000 iterations.
- **Medium Workload**, with a graph of 1000 nodes, a population size of 5000 and 1000 iterations.
- **High Workload**, with a graph of 2000 nodes, a population size of 10.000 and 1000 iterations.

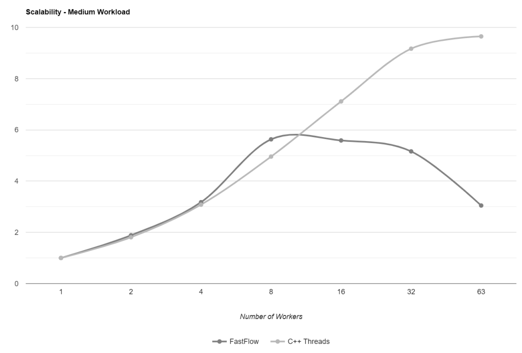
For each of these categories, the parallel implementations were subjected to seven tests with **one**, **two**, **four**, **eight**, **sixteen**, **thirty-two** and **sixty-three** workers, respectively. All tests were repeated several times in order to collect statistically more robust values, averaging the performance of each test so that natural fluctuations in execution times due to the underlying machine and its third-party workload could be transcended.

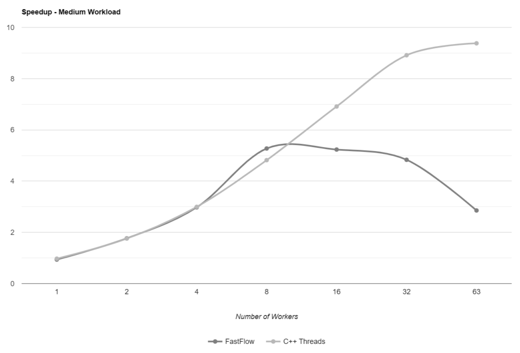
Comparing the measures of "Speedup", "Scalability" and "Efficiency" achieved by the two implementations during the empirical evaluation, interesting insights emerge:

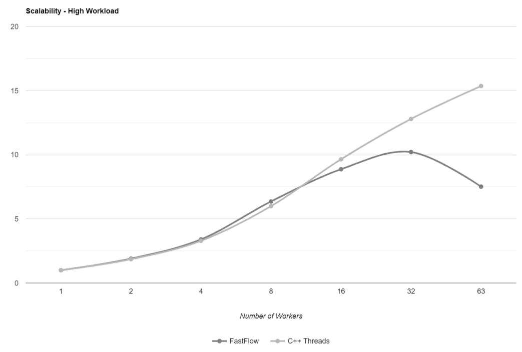
- **Distance from ideal time**: the presence of overheads generated by the use of the condition variable, the associated lock operation and the barrier in the C++ native threads version, as well as the presence of overheads introduced by the "FastFlow" library, make actual completion times worse than the ideal one.
- **Performance degradation of "FastFlow"**: as the number of threads involved increases (with even more noticeable effects with lower workloads), "FastFlow" appears to suffer greater performance degradation respect to the C++ native threads implementation. Trying to investigate the causes of this degradation, in particular trying to run the program on the remote machine in several simultaneous instances, it appears that execution times are heavily influenced by the presence of other (especially when similarly using "FastFlow") activities running on the machine. One theory, which should certainly be investigated in greater depth, could see as the cause of this phenomenon a negative combination between context switches at the level of the cores and the internal implementation of the library.
- **The "FastFlow" implementation works best with lower workloads**: with low workloads, as the number of workers remains below the aforementioned "FastFlow" degradation threshold, the implementation manages to obtain better performance than the version based on C++ native threads.

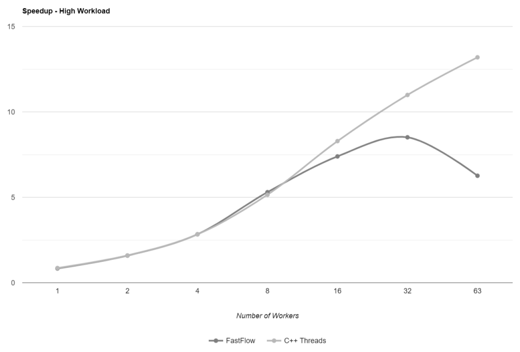

Scalability - Low Workload
0 2 4 6 8 10
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads

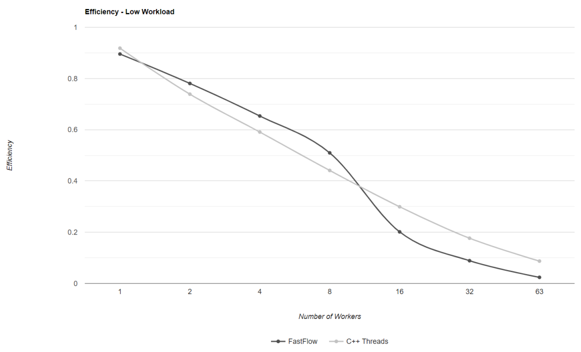

Speedup - Low Workload
0 2 4 6 8 10
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads

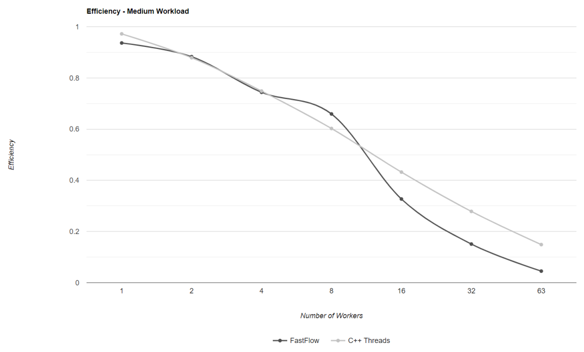

Scalability - Medium Workload
0 2 4 6 8 10
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads

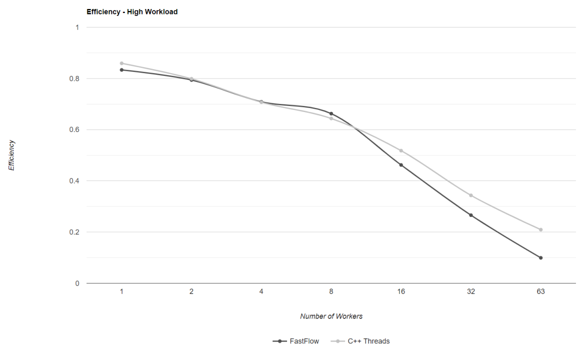

Speedup - Medium Workload
0 2 4 6 8 10
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads


Scalability - High Workload
0 5 10 15 20
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads


Speedup - High Workload
0 5 10 15
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads


Efficiency - Low Workload
Efficiency
0 0.2 0.4 0.6 0.8 1
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads


Efficiency - Medium Workload
Efficiency
0 0.2 0.4 0.6 0.8 1
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads


Efficiency - High Workload
Efficiency
0 0.2 0.4 0.6 0.8 1
1 2 4 8 16 32 63
Number of Workers
FastFlow
C++ Threads

5. User Instruction

Inside the main folder **"Project"**, the files are organised in the folders: **"lib"**, which contains the external resources useful for the implementations (excluding the "FastFlow" library), **"src"**, which contains the source files of the implementations discussed, and **"plots"**, which contains the plots representative of the experimental analysis applied to the implementations.

Once the user is positioned in the **"Project"** folder, the following shell commands will compile the source codes of the three implementations:

- **Sequential Implementation:**

  ```
  g++ -O3 ./src/Genetic_TSP_Sequential.cpp
  ```

- **C++ Native Threads Parallel Implementation:**

  ```
  g++ -O3 -std=c++20 -pthread ./src/Genetic_TSP_Parallel_NativeThreads.cpp
  ```

- **"FastFlow" Parallel Implementation:**

  ```
  g++ -O3 -I "FastFlow Location" ./src/Genetic_TSP_Parallel_FastFlow.cpp
  ```

The inputs required for the execution of all programs are in order: **"n"** (the number of nodes for the generation of the TSP instance graph), **"start"** (the starting node for the paths searched by the algorithm), **"pop_size"** (the population size), **"ng_percentage"** (the number of specimens generated at each iteration expressed as a percentage, integer value from 1 to 100, with respect to the population size), **"mut_rate"** (the probability expressed as a percentage, integer value from 0 to 100, that a generated specimen will undergo a mutation) and **"maxit"** (number of generations and iterations of the algorithm). In addition to the previous input parameters, parallel implementations require the final parameter **"w"**, number of worker threads.

For example, a typical shell command to run the executable file of a parallel implementation would have a structure such as:

-
  ```
  ./a.out n start pop_size ng_percentage mut_rate maxit w
  ```