Università di Pisa, Master Degree in Computer Science
Course of "Parallel and Distributed Systems: Paradigms and Models" (SPM)

Travelling salesman problem with genetic algorithms

Report of the Final Project for the A.Y. 2019/2020

Francesco Corti

July 7, 2020

Abstract

The purpose of the project was to solve Travelling salesman problem, known as *TSP*, by using a Genetic Algorithms approach. In addition to the sequential version used as baseline, two parallel versions were developed, one using standard C++ threads and one using Fastflow [1].

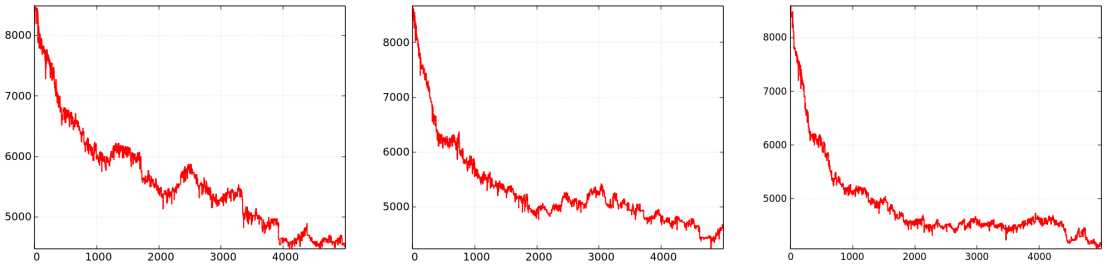
1 Introduction

Travelling salesman problem requires to find the shortest path visiting all the nodes in a graph just once. The *TSP* is an NP-hard problem, namely a problem that cannot be solved in polynomial time, however during past decades many heuristics and algorithms were invented to find an approximate solution of the problem. One of them, used in the project, is a genetic algorithms approach that mimics the genetic evolution of species.

A genetic algorithm starts with a set of chromosomes (*population*) that are typically randomly defined. Then these chromosomes are evaluated and a reproductive opportunity is allocated for each of them in such a way that those chromosomes which represent a better solution to the target problems are given more chance to "reproduce" than those chromosomes which are poorer solution [2]. After the selection phase the crossover and mutations operations are applied to produce a new population for the next generation. It is helpful to view an entire generation phase as a two stage process, it starts with a population that after the selection phase becomes an intermediate population. Then the new population is produced by applying recombination operations, according to probability, on the intermediate population.

In the project *TSP* was modelled as an *undirected weighted graph*, a chromosome represents a route that visits each city and returns to the origin city and our goal was to find a good chromo-

some that minimizes the length of this complete route on the graph. For each chromosome the evaluation function, that provides the performance, was the sum of all the edges length inside it. To verify the correctness of the algorithm for each version developed, the generation number with the current generation best chromosome path value were saved and then plotted. This was done for five thousand generations with chromosomes of length equal to two hundred. As we can see from the plots obtained (figure 1) for every version developed the convergence of the algorithm to a lower route on the graph was obtained.



(a) Sequential (b) Standard thread (c) Fastflow

Figure 1

1.1 Measures

To compare and model the performance of a parallel program two performance indicators were used, *speedup* and *scalability*. For these metrics the speedup asymptote is given by $f(n) = n$.

1.1.1 Speedup

The *speedup* is the ratio between the best known sequential execution time and the parallel execution time. It gives a measure of how good is our parallelization with respect to the best sequential computation. It is a function of n the parallelism degree of the parallel execution.

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

1.1.2 Scalability

The *scalability* is the ratio between the parallel execution with degree equal to 1 and the parallel execution time with parallelism degree equal to n. It measures how efficient is the parallel implementation in obtaining better performance on bigger parallelism degree.

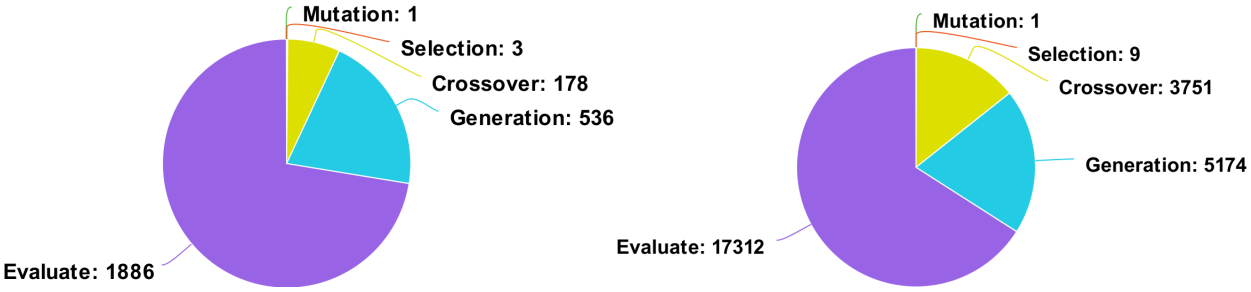
$$scalab(n) = \frac{T_{par}(1)}{T_{par}(n)} \tag{2}$$

2 Parallel Architecture Design

The program was designed to handle large graph as input (that implies large chromosome dimension). This was done by achieving data locality by modelling most of the application with *Scanning* and *Sorting* operation. As mentioned in [3] RAM model fails to capture the running

time for problems that have large data sets because of the I/O bottleneck. Indeed reading and writing data in sequential order or sorting the data to obtain requisite layout is less expansive than accessing data at random. This two operation, respectively *std::for_each* and *std::sort*, were standardized and added to C++ until C++11 and are available inside the *algorithm* header.

To exploit parallelism an initial analysis on the sequential version of the program was done. Each genetic operation was measured, by importing *chrono* header library, to understand which part of the program needed to be parallelized and which not. This analysis was also useful to understand the proportion between the size of the graph and the time taken for each genetic algorithm operation. The following pie charts (figure 2) were obtained by comparing each time spent by the operations that compose the algorithm, the number means the time taken in milliseconds to complete the operation. In both cases, the number of chromosomes taken into consideration was 2000 and the number of generation was 1.



(a) 1000 nodes

(b) 5000 nodes

Figure 2

The first thing we noticed is that the most expansive operations are *generation* (due to the random generation of the graph), *evaluate* and *crossover*. Also, surprisingly, the selection phase that substitutes the old generation with the new one takes an insignificant amount of time. This is because all the index of the new population are sorted before the copy happens to exploit data locality and after the new generation is obtained (through *Stochastic Universal Sampling* [2]) the old generation is swapped with the new one, by moving the pointer, to avoid useless copy.

So, after the initial analysis I implemented the parallel versions by paying attention to the bottleneck created in *evaluate*, *generation* and *crossover* phases. This was done by parallelizing the computation by splitting the works among multiple threads and then join the results obtained to proceed with the sequential computation of *selection* and *mutation* phases. Basically the high level structure of the program has the following operations form:

```
read initial population P
for(all generations) {
       select random parents, enforcing fitness.
       apply cross-over and mutation according to probabilities.
       add new generated individual to new population.
}
```

2.1 Standard thread implementation

Since exception safety is an essential aspect of C++ code, I decided to use a modern C++ approach to spawn the workers to compute the genetic algorithm heavy operations in parallel. Indeed, spawning threads with the standard constructor offer by *std::thread* object doesn't guarantee exception safety and an exception launched by a single thread can terminate the whole application. Also the destruction of the threads, when an exception arises, is not guaranteed and the program can have leak memory thread problems (also known as dangling thread problem).

So, I decided to use *std::async* to spawn the threads with *std::launch::async* to launch it asynchronously, this action produces a result stored in a *std::future<T>* and the operation to be done by each thread is passed in-place as a lambda function with all the parameters needed for the computation as input. This approach guarantees that even if an exception is raised by a thread and its *std::future* object is destroyed the destructor will wait for the thread to complete, so the memory thread leak problem is avoided.

The model used is the fork/join one, this was chosen because each chunk computed by a worker has the same size and amount of work (computational time speaking) to be done. So a dynamic load balancing of the work between threads was not needed. Indeed the communication arrow between masters node (heavy operation of the genetic algorithm) and the workers are implemented as "chunks" (in the code a *std::pair<start,end>*) of the main data structure that has the current population saved in it. The implementation has the skeleton reported in figure 3:

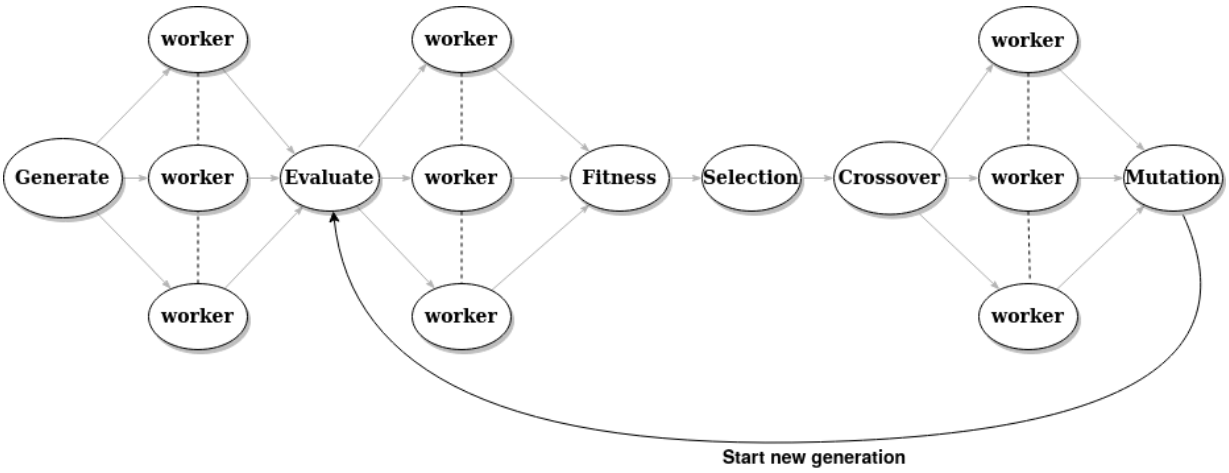


Figure 3: Standard C++ threads structure

As we can see from figure 3 the first master worker spawning is done only at the beginning of the computation to speedup the random generation of the graph. This behaviour can be extended and modified, but basically the skeleton structure remains similar, to let the application parsing in parallel a file and create the graph. As mention above in §2 *fitness* and *selection* were designed as sequential computation because the time spent was too low (see 2) to obtain benefits exploiting parallelism.

2.2 Fastflow implementation

The version developed with Fastlow [1] was modelled using the information obtained from the analysis of the sequential algorithm (figure 2). So, also in the Fastlow implementation the operations of *generation*, *evaluation* and *crossover* were parallelized, moreover the communication between the emitters and collectors were chunks of indexes of chromosomes in the main data structure. The random generation of the graph was handled by creating a temporary farm with an emitter and a collector, but to save computational resources after the creation of the graph it is automatically deleted by the program. So the main skeleton of this version is composed by a pipeline composed by two farms (one for *evalutation* and one for *crossover*) and a sequential stage that computes the *fitness* and *selection* results. After creating the pipeline by injecting in the constructor the farms and the sequential node object the method *wrap_around()* is called to let the computation restarts from the initial emitter after all the current generation operations are done.

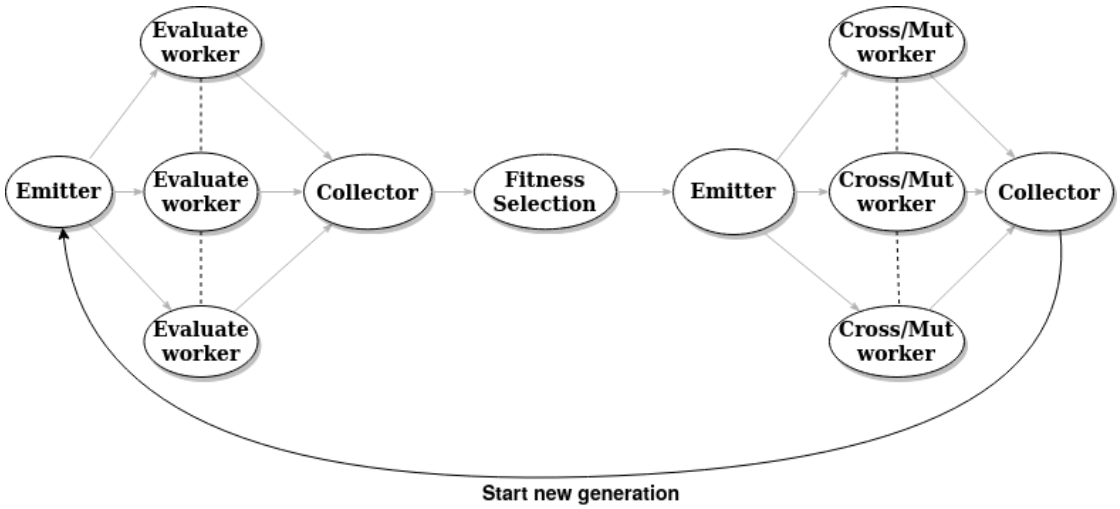


Figure 4: Fastflow structure

As we can see from figure 4 the sequential operations *fitness* and *selection* were merged in a single stage (node inside the pipeline) to save computational resources. Also every worker spawned by the second farm perform *crossover* and immediately after that performs *mutation* according to probabilities to optimize computational resources.

3 Implementation details

The project has three classes: *fastflowTSP*, *parallelTSP* and *sequentialTSP*. If an object of *Graph* type has been created and passed before in the constructor by calling the method *Run* (and passing the parameters) the algorithm starts. The user need to pass all the parameters by command line interface and then the program computes the three versions of the algorithm and for each of them the time spent is printed out. The only check is done on the number of workers

by using *std::thread::hardware_concurrency()* method, if the user pass a number of workers that exceed the number returned by the method (so the supported thread of the machine are lower than the user specified threads), the number of thread is setted to the number returned by the method minus one.

4 Results

Each version of the program was tested on the Xeon PHI machine, the tests were done with 10 generation and 20000 chromosomes and considering [500, 1000, 2000] nodes in the graph. As we can see from the curves, a good speedup is obtained with the first 50 threads because genetic algorithm are generally highly parallelizable algorithms. But as expected from the theory, after spawning more than 100 threads the overhead taken to set up and coordinate the computation is higher than the parallelism benefits obtained by adding threads. Surely with a bigger graph as input, the speedup curve would be increasing even with more than 100 threads.

4.1 Speedup curves

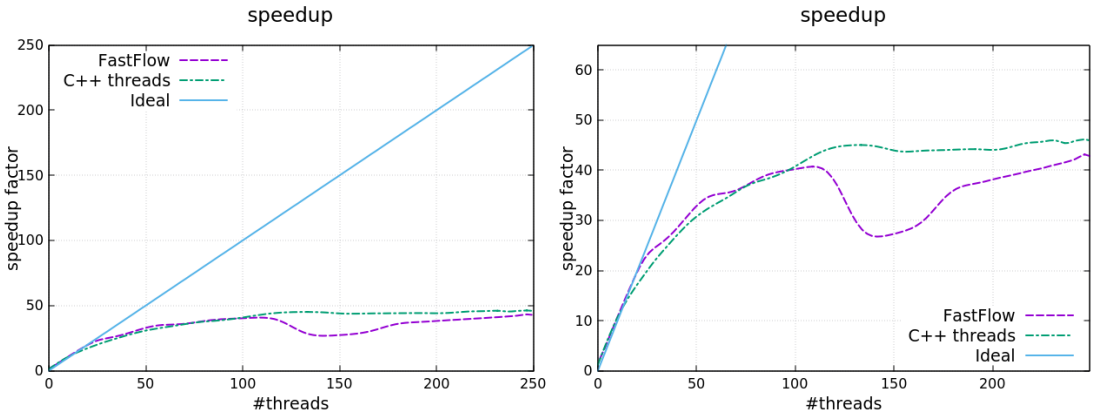


Figure 5: Speedup 2000 nodes

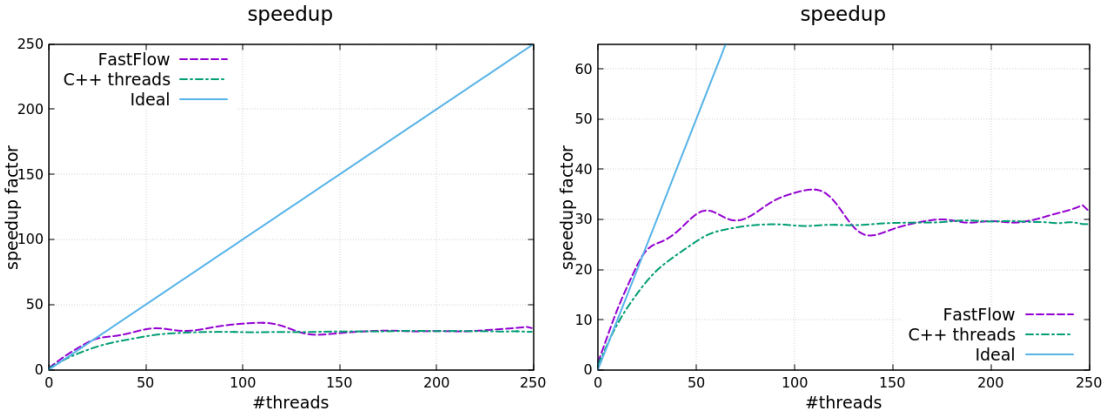


Figure 6: Speedup 1000 nodes

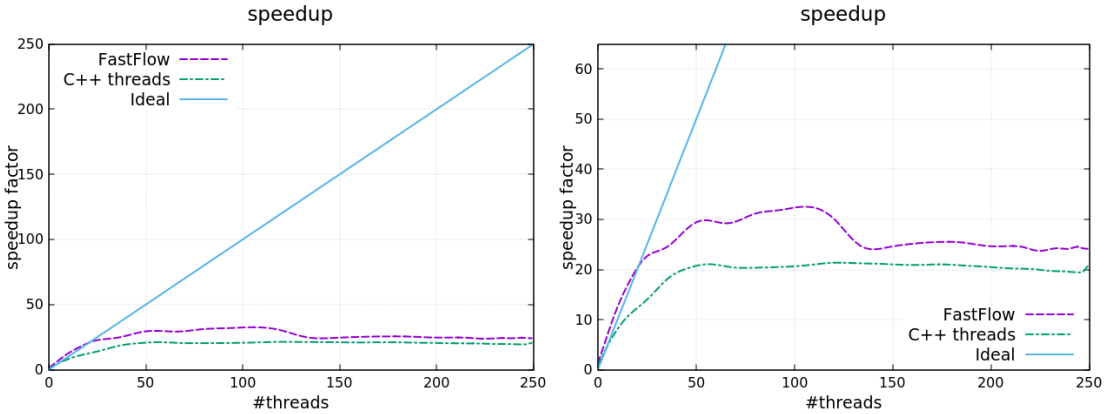


Figure 7: Speedup 500 nodes

4.2 Scalability curves

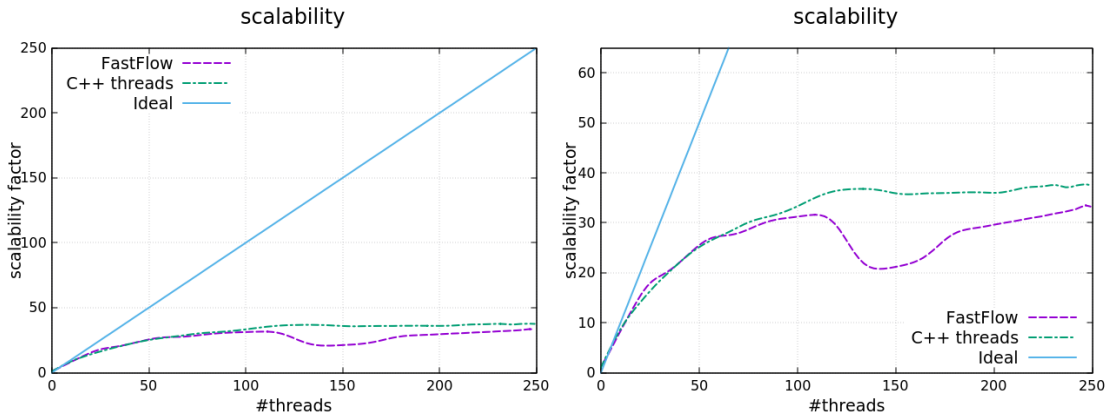


Figure 8: Scalability 2000 nodes

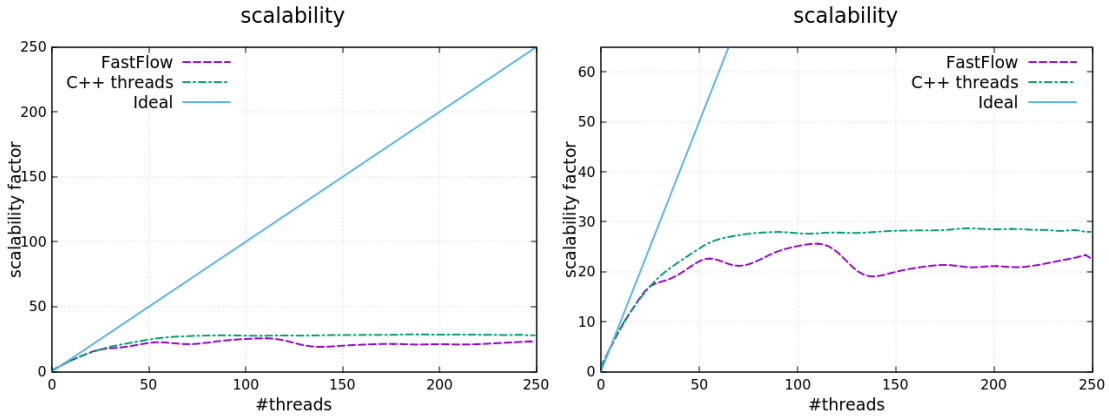


Figure 9: Scalability 1000 nodes

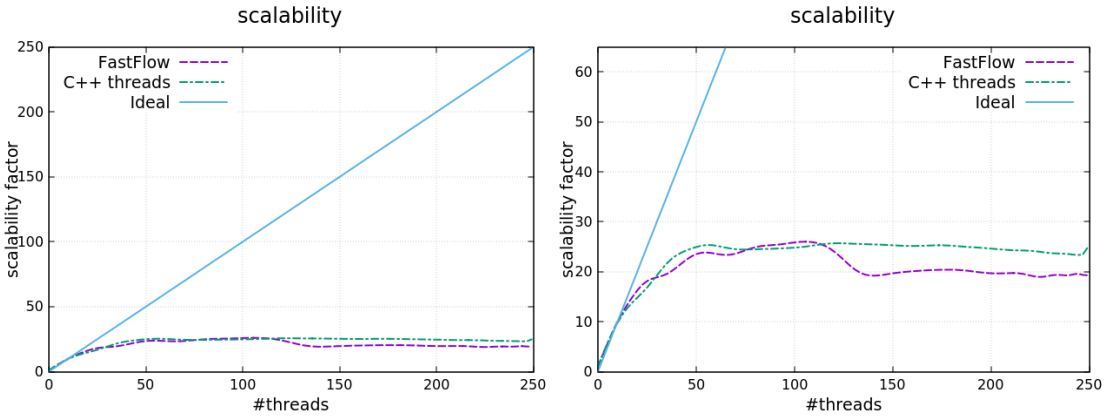


Figure 10: Scalability 500 nodes

5 Conclusion

The project gave me the opportunity to apply the theory seen during the course to a real world problem like Genetic Algorithms. The initial design of the parallel architecture was strictly correlated to the results obtained from the analysis of the sequential program. Also at the beginning it was challenging to deal with the parallel implementation of the algorithm with the standard C++ threads, the error generated by the parallel program were hard to find (and debugging) manually, but thanks to tools like *Valgrind* or *ThreadSanitizer* they were quite easy to found and fix. However, these fact enabled me to understand the purpose of parallel skeleton framework like Fastflow that it hides all the parallelism complexity to the programmer by using object oriented programming technique.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimo Torquati. *Fastflow: high-level and efficient streaming on multi-core* . Programming Multi-core and Many-core Computing Systems
`https://github.com/fastflow/fastflow`

[2] Darrell Whitley. *A genetic algorithm tutorial* . Kluwer Academic Publishers

[3] Müller-Hannemann, Matthias, Schirra, Stefan. *Algorithm Engineering*. Springer