SPM 2020, project report
Parallel genetic algorithms for TSP: Design, implementation and experimental analysis

Emanuele Cosenza, 536836
e.cosenza3@studenti.unipi.it

September 2020

Introduction

This report is about the design, implementation and analysis of a parallel genetic algorithm for the Travelling Salesman Problem (TSP). The report will be organized in the following way. Section 1 will briefly describe the main assumptions made on the TSP and the genetic algorithm. Section 2 will contain a discussion on the design of the parallel genetic algorithm, deriving the main theoretical expectations. Section 3 will be about the mandatory and additional implementations of the algorithm outlined in Section 2. Finally, Section 4 will contain the experimental results of the project and it will be followed by a brief conclusion.
For any information regarding the use of the project, please refer to the file `README.md` in the project folder.

1 A genetic algorithm for TSP

1.1 Assumptions about the TSP

The Travelling Salesman Problem (TSP) considered in this project is defined over undirected weighted graphs of any kind (also non-complete). The goal of the problem is to find the hamiltonian path with minimum length of the input graph. This means that the solution path must contain every node of the graph exactly once. Also, in this version of the problem, the path does

not have to be a cycle, meaning that it may end on a node different from the starting one.

1.2 Genetic algorithm design

A genetic algorithm can be instantiated in many possible ways. In the genetic TSP algorithm considered here, evaluation, selection, crossover and mutation operators are applied in each generation to a population of chromosomes until a maximum number of generations is reached. The population has a fixed size which is kept the same throughout different generations.
Chromosomes, evaluation, selection, crossover and mutation are all defined as follows:

Chromosomes Chromosomes are hamiltonian paths of the input graph. These paths are encoded as lists of integers, where each integer represents a node in the graph.

Evaluation operator The evaluation operator is the first to be applied at the start of each generation. It is applied to each chromosome in the population, returning a fitness score that determines how likely it is for the chromosome to be picked by the selection operator. In the context of TSP, it simply returns the length of the path represented by the chromosome.

Selection operator Tournament selection has been chosen as the selection operator. First, a random subpopulation of K chromosomes is selected. Then, a "tournament" is run among the K chromosomes, returning the one with the minimum path length.

Crossover and mutation operators These two operators are applied one after the other on two selected parent chromosomes a and b. Two probability values, p_c and p_m, regulate respectively how often crossover and mutation take place.
The crossover operator is applied as follows. First, two indices $i > j$ are chosen randomly. Then, for each k such that $i \leq k \leq j$, the elements $a[k]$ are repositioned in the list in the order they appear in the parent b. The same procedure is applied to the parent b with the same indices.
The mutation operator is applied on both chromosomes returned by the crossover operator as follows: given a single chromosome a, two indices i and j are chosen randomly and the element $a[i]$ is exchanged with $a[b]$.

The resulting chromosomes are added to the new population only if they represent hamiltonian paths.

2 Discussion on parallel design

The genetic algorithm introduced in Section 1.2 is an iterative process in which a new population is produced from the previous one in each generation. In particular, the new population at generation i must be completely produced in order to start the generation $i + 1$. This means that different generations must be necessarily run sequentially in order to preserve the functional semantics of the algorithm.
For what concerns the single generation, the analysis is more complex. Each generation is composed of two "macro" steps: the evaluation (`E`) step and the selection + crossover/mutation (`SCM`) step. The fitness scores computed in the `E` step must be completely computed before starting the `SCM` step. In fact, to pick a single parent, the chosen selection mechanism takes into account all the chromosomes with their fitness scores. For this reason, the steps must be run sequentially in the generation.
On the other hand, the two steps considered separately may be exectuted in parallel by different workers. In fact, in the `E` step the fitness scores of the chromosomes do not depend on each other, while in the `SCM` step, the selection operator does read only operations on the population vector and crossover/mutation is applied on separate pairs of chromosomes to produce a new one.
The `E` step is a perfect candidate for the application of a standard `Map`, with each parallel entity computing fitness scores relative to a subpopulation. On the other hand, in the `SCM` step, every parallel worker should receive in input the entire population and the relative fitness scores, producing a new subpopulation by applying sequentially the selection operator and the crossover/-mutation operators. Since this can be thought of as functional replication, it would be reasonable to apply a standard `Farm` in a non-streamed data parallel fashion. The computation of a single generation can be then formalized as follows (indicating a sequential composition with `Comp`):

```
Comp(Map(E), Farm(SCM))
```
(1)

Completion time analysis Given the nature of the problem, the main focus here will be on the completion time of the computation in 1 (the fact that the computation is repeated for a number of generations will not influence the analysis). Finally, the goal will be to derive $speedup(n)$ and

scalability(n) with respect to the completion time. In this way, it will be possible to predict the behaviour of a concrete implementation of the parallel algorithm under different circumstances.
From now on, n will be the parallelism degree, while $T_o(n)$ will be the overhead time. The overhead time depends on the particular implementation of the algorithm: it includes explicit overheads, such as the time spent in possible split and merge operations, and implicit overheads that depend on the specific architecture the algorithm is being executed on.
We have:

$$T^{par}(n) = T_E^{par}(n) + T_{SCM}^{par}(n) + T_o(n) \tag{2}$$

$$T^{par}(n) = \frac{T_E^{seq}}{n} + \frac{T_{SCM}^{seq}}{n} + T_o(n) = \frac{(T_E^{seq} + T_{SCM}^{seq})}{n} + T_o(n) = \frac{T^{seq}}{n} + T_o(n) \tag{3}$$

where $T^{par}(n)$ is the parallel completion time of 1 with n workers, $T_E^{par}(n)$ and $T_{SCM}^{par}(n)$ are the completion times respectively of `Map(E)` and `Farm(SCM)`, while T^{seq}, T_E^{seq} and T_{SCM}^{seq} are the completion times of the sequential counterparts of the terms mentioned previoulsy.
At this point, *speedup*(n) and *scalability*(n) can be both derived as follows:

$$speedup(n) = \frac{T^{seq}}{T^{par}(n)} = \frac{T^{seq}}{\frac{T^{seq}}{n} + T_o(n)} \tag{4}$$

$$scalability(n) = \frac{T^{par}(1)}{T^{par}(n)} = \frac{T^{seq} + T_o(n)}{\frac{T^{seq}}{n} + T_o(n)} \tag{5}$$

What emerges from the equations is that when the overhead time is negligible with respect to T^{seq}, both *speedup*(n) and *scalability*(n) approach n. In the context of the TSP parallel algorithm, T^{seq} can be varied by varying the population size. On the other hand, given a fixed architecture and a fixed algorithm implementation, $T_o(n)$ increases with increasing values of n because of the higher number of parallel entities to be orchestrated.
Just by analyzing the algorithm from a theoretical perspective, we can then expect the following facts:

1. For fixed values of n, increasing the population size will increase both the speedup and the scalability.

2. For fixed values of the population size, increasing n will decrease both the speedup and the scalability.

Since overheads essentially correspond to serial parts of the program which cannot be parallelized, what we have derived from this discussion is simply an istantiation of Gustafson's law, and 1. and 2. are its consequences. The experimental analysis will hopefully enforce these facts.
For the sake of simplicity, we did not consider here the overhead of creating and destroying the n parallel activities. Again, the same type of argument can be applied here. If the algorithm is run for a sufficient number of generations, the time associated with these overheads will be negligible with respect to the total computation time. If that is not the case, we will see a degradation of both $speedup(n)$ and $scalability(n)$.

3 Implementation

Following the assignment text, the project contains two main mandatory `C++` implementations of the genetic parallel algorithm for TSP, one with `C++` threads and one based on the `FastFlow` library. Moreover, two slight variations of the previous versions have been developed, one with `C++` threads and one with the `FastFlow` library. So, in total we have the following 4 parallel versions of the algorithm:

1. `C++` threads version with a spin barrier (`tsppar_th.cpp`).
2. `C++` threads version with a blocking barrier (`tsppar_th.cpp` compiled with `-DBLOCKING`).
3. `FastFlow` version with `ParallelFor` (`tsppar_fffor.cpp`).
4. `FastFlow` version building on the already existing `poolEvolution` pattern (`tsppar_ffpool.cpp`).

An important thing to notice here is that while versions 1 and 2 will likely have different performances, versions 3 and 4 will behave more or less in the same way, since they employ the same type of computation under the hood. Just to be clear, version 4 has been implemented just to experiment with a higher level parallel pattern.
Other than the parallel related code, the project contains also a sequential version of the algorithm (`tspseq.cpp`) needed to compute the speedup, as well as modules relative to the genetic and graph domains (in `genetic.hpp` / `genetic.cpp` and `graph.hpp` / `graph.cpp`). Finally, a bruteforce algorithm has been implemented to evaluate the genetic algorithm approximations against the exact solutions (`tspbrute.cpp`).
As already mentioned in the Introduction, for any information regarding the use of the project please refer to the file `README.md` in the project folder.

3.1 Parallel versions with C++ threads

For the C++ threads version, two possible implementations were first taken into account: one with a thread pool and one with a thread barrier. At the end, the choice fell on the barrier version, not because of any a priori consideration, but just to simplify and shorten the analysis.
The project contains two different implementations of the barrier: a spin barrier and a blocking barrier (in sync.hpp/sync.cpp). The spin barrier is implemented simply by using a std::atomic<int> counter, making the threads spin in a while loop until the last thread enters the barrier. The blocking barrier is implemented with the use of a mutex, making the threads wait on a condition variable.
In the barrier versions, each thread works on portions of the population vector, with ranges that are statically computed beforehand. All threads have a reference to the population vector, the fitness scores vector and a third vector which is filled in each generation with new individuals resulting from the evolution.
In the context of a single generation, the threads first compute the E step and when they finish they enter the first barrier. When the last thread enters this barrier, all threads can then proceed to compute the SCM step, filling up their portion of the new population vector and entering a second barrier as they finish. When all threads get past this barrier, they can finally swap the pointers to the old population vector and the new one.

3.2 Parallel versions with the FastFlow library

The main FastFlow version has been developed by simply concretizing the abstract computation in 1 with FastFlow's high level mechanisms. Both Map and Farm patterns have been substituted by a parallel for, using FastFlow's ParallelFor.
The second minor version uses FastFlow's poolEvolution, which is a general parallel pattern applicable on any kind of evolving set of individuals. The pattern has been adapted to the TSP genetic algorithm simply by defining the selection and filter functions with the appropriate business code. Under the hood, the pattern uses a ParallelFor and it can therefore be considered equivalent to the previous version.

3.3 Overheads and expected performances

In all the implementations, possible sources of overhead are:

- Forking and joining of threads.

- Operations relative to barriers.
- Inevitable load imbalances both in the `E` step and `SCM` step.
- Phenomena of false sharing, especially with a high parallelism degree.

Moreover, in the blocking barrier version, when the threads reach the barrier they are put to sleep and then awaken by the operating system. This will result in more overhead. Because of this, it is reasonable to expect that the spin barrier version will perform better than the blocking one.
For what concerns the `FastFlow` versions, the amount of overhead will depend strictly on the internal mechanisms of `FastFlow`. Keeping in mind that the library is highly optimized, we expect to see the `FastFlow` versions outperform the barrier based versions, especially with high parallelism degrees.

4 Experimental analysis

For the sake of simplicity, all the experiments were performed on a complete graph with 10 nodes and integer weights between 0 and 9 [1][2].
Genetic algorithms generally work best when p_c is high and p_m is low (too much mutation would result in a random search). Testing different values on different graphs, and in particular on the graph mentioned above, $p_c = 0.9$ and $p_m = 0.1$ gave good results and have therefore been fixed as such for all the experiments.
To check experimentally the theoretical analysis, the 4 parallel implementations were all tested on a population of 500 and 3000 individuals for 20 generations. The number of generations was chosen high enough so that the forking and joining times would not influence the analysis that much. Each parallel implementation was tested with a parallelism degree $n \in \{1, 2, 4, 8, 16, 32, 64\}$ picking only powers of 2 to get lower computation times, giving nonetheless an idea of the asymptotic behaviour of the 4 implementations. To avoid biased results, each pair `<implementation, par.deg.>` was tested for 4 runs, computing then the average of the relative completion time. Finally, $scalability(n)$ was computed with different values of the parallelism degree n.
Even if the theoretical analysis of section 2 also considered speedup, from now on we will only focus on scalability. This will keep the analysis short and

[1]The optimal solution of the TSP for this graph is a hamiltonian path with length 4. With reasonable parameters, the genetic implementations all return the optimal solution when they are run for a sufficient number of generations.

[2]Used architecture: Xeon Phi machine, x86_64, CPUs: 256, Cores: 64, 1.30 GHz.

simple while still providing insight on the performance of the 4 implementations. Speedup plots can anyway be produced with the `data_analysis.py` script.

Results The scalability plot relative to a population of 3000 individuals is shown in Figure 1. When $n \leq 16$, the four implementations all reach an

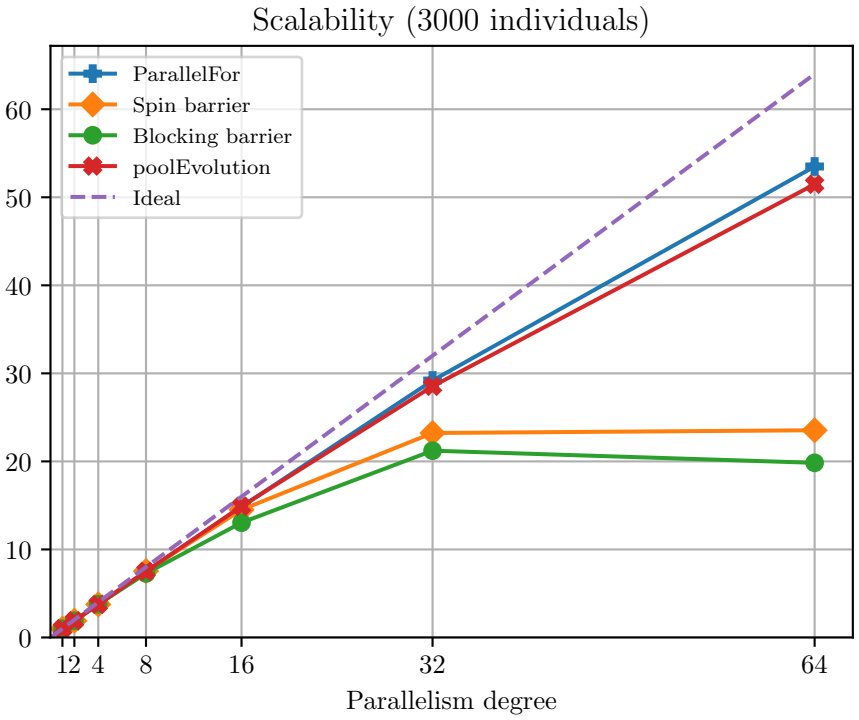

Figure 1: Scalability values of the 4 parallel implementations with different parallelism degrees and a population size of 3000.

almost ideal scalability. The main differences show up when $n \geq 16$. Both `FastFlow` versions turn out to be much more performing than the barrier based versions. When $n = 64$, the `FastFlow` versions reach scalability values greater than 50, while the barrier based versions do not even reach half of the ideal scalability. As expected, the `FastFlow` versions have similar performances, with the `ParallelFor` version outperforming the `poolEvolution` one just by a little. Similarly, the barrier versions have similar performances, with the spin barrier version always outperforming the blocking barrier for reasons already explained in Section 3.3.
Following what has been said in section 2, we will now check what hap-

pens with lower values of the population size, expecting to observe a general degradation of the performances. The scalability plot in Figure 2 shows what happens when the population size is shrunk to 500. Since the ratio between the actual computation time and the overhead time got lower, the general performances got worse as expected in section 2. As before, the `FastFlow` versions show a similar behaviour, still reaching decent scalability values (around 35) even at $n = 64$. On the other hand, the impact on the barrier based versions is much more evident. The spin barrier version keeps up with the `FastFlow` implementations up until $n = 16$, but then scalability values start to decrease, meaning that completion times actually got higher even with more parallel workers. At the same time, the blocking barrier version starts to underperform by a lot even at $n = 8$, with scalability values decreasing after the $n = 32$ mark.

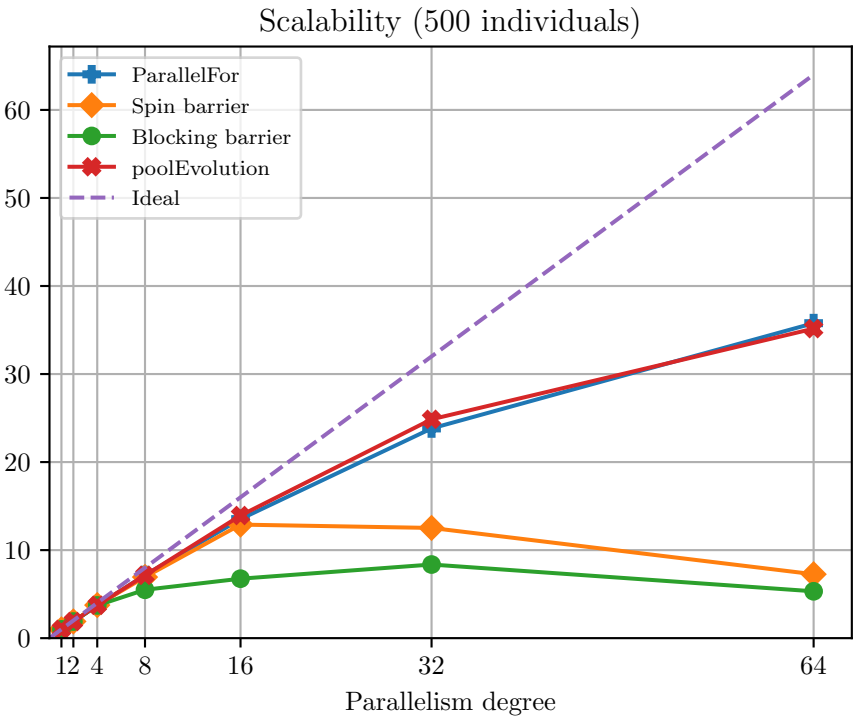

Figure 2: Scalability values of the 4 parallel implementations with different parallelism degrees and a population size of 500.

Conclusion

The experimental analysis confirmed the main theoretical expectations and intuitions outlined in the previous sections: the `FastFlow` versions outperformed the barrier based versions and a higher population size resulted in better performances.
To conclude, using a high level library allowed to map the abstract model of the computation to the actual code in a much more direct way, eliminating the burden of dealing with low level mechanisms such as barriers, mutexes and condition variables, producing at the same time a much more scalable code.