Travelling salesman problem

Giovanni Sorice - 606915

July 2020

Abstract

The assignment required to implement three different solutions for Travelling salesman problem. I initially developed a *sequential* version, needed as a baseline and performance comparison for further implementations. The parallel versions developed are based on c++ standard thread and on FastFlow framework.

1 Introduction

This report describes the framework developer to execute the traveling salesman problem with genetic algorithm. The TSP is a NP-hard problem and for this reason during the year many heuristic and approximation algorithms were invented and used. One of the most successfully category of algorithms for the TSP problem are the genetic algorithms. A genetic algorithm works by building a population of chromosomes, usually randomly generated, which is a set of possible solutions to the optimization problem. During each iteration of the algorithm, the population is randomly altered in hopes of creating new chromosomes that have better evaluation scores. The next generation population of chromosomes is randomly selected from the current generation with selection probability based on the evaluation score of each chromosome, adding some randomly changing to the previous population chromosomes. There are several variant of genetic algorithms structure. I choose also to take in account of the best previously population chromosomes for the current population.
Usually a genetic algorithm is divided in more phases:

- Initialization;
- Evaluation;
- Selection;
- Reproduction;
- Crossover;
- Mutation.

1.1 Measures

We need some measures to understand the real advantages of using a parallel implementation. The common measures used in this case are speedup and scalability.

1.1.1 Speedup

The speedup is used to compare the time needed to execute a specific task sequentially with the time needed to do the same task in parallel. The ratio among the time spent by the sequential and the time spent by the parallel is called Speedup. We hope that it was linearly proportional to the number of parallel degree used and for this reason the time spent doing the task decrease as 1/k where k is the parallelism degree. Unfortunately, there are more things to keep in mind (overhead) and usually the rate of speedup is not the expected. The speedup is computed as follows.

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

1.1.2 Scalability

The scalability is the ration between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

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

2 Implementation structure

I defined three classes: *TSPGeneticAlgorithm*, *TSPGeneticAlgorithmST*, *TSPGeneticAlgorithmFF* in which there are respectively the implementation of the sequential code, parallel code with c++ standard thread used as fork-join, parallel code with FastFlow. All the code could be found at https://github.com/GiovanniSorice/TSPGeneticAlgorithm. I also implemented *undirectedGraph* class, it define the graph structure and expose the main methods with which to interact with it. In the last few year, the importance of reduce the I/O is grown every year. This because at the moment it is more expensive to move data than execute instruction with the CPU. For this reason, i made particularly attention to use algorithms that try to minimize the cache miss. This can be seen for example in the *selectionReproduction* method where i try to use the scan and sort programming paradigm to minimize the I/O made by the application. Also this approach and attention are used in all the implemented classes.

2.1 Parallel implementation structure

The core of the parallel implementation structure reside in see the execution like a pipeline. Each node of the pipeline can be developed as a farm, in this way we can reach an important speedup as we can see in the graph (indicare numero img). Moreover, i try to understand where could the framework have a bottle neck. This analysis help me to decide where i need a parallel execution and where it would not be necessary to parallelize it.

In the c++ standard thread i used a fork-join implementation because all the task of each node have the same complexity, so a thread can not steal some work to another, because in theory they have the same execution time. Each node of the pipeline, divide the work in equal size task and assign to each thread a specific range of shared memory. Assigning range to a thread could help to preserve the locality of the memory and save cache miss. Assuming that

all thread have the right range size (Magari spiegare meglio in che senso) the probability that two different thread were sharing the same line of cached memory are low and as a consequence cache sharing problem are low too.

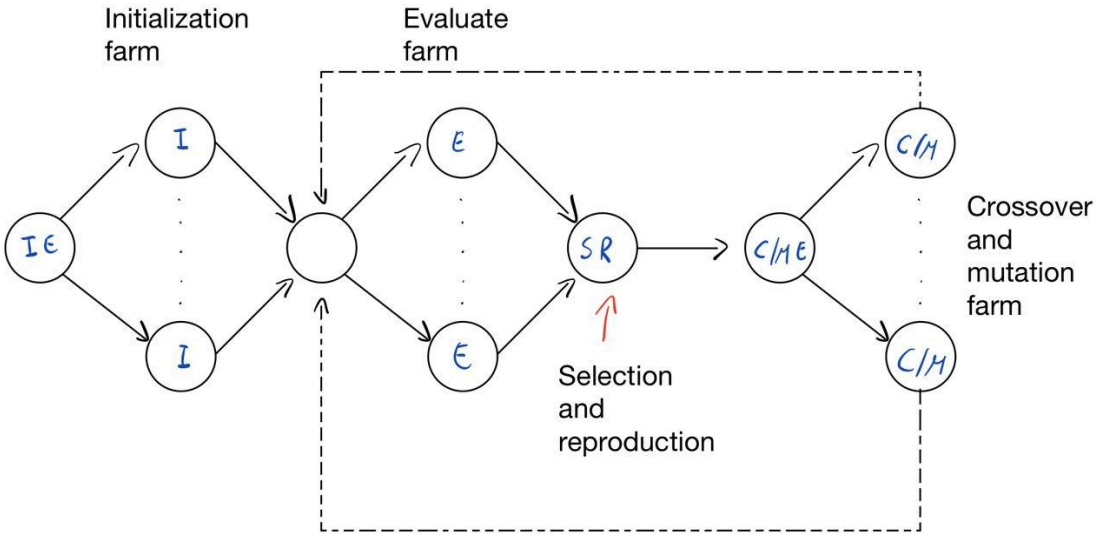

Figure 1: FastFlow pipeline.

3 Results

The results are shown in graphs, all the execution are made on square matrices. The size tested are 500x500, 1000x1000, 5000x5000 and 1000x1000 with 15 iterations. The test are made on Xeon Phi.

3.1 Speedup graph

Figure 2: Speedup curves for 500x500 boards.

Figure 3: Speedup curves for 1000x1000 boards.

Figure 4: Speedup curves for 5000x5000 boards.

Figure 5: Speedup curves for 10000x10000 boards.

3.2 Scalability graph

Figure 6: Scalability curves for 500x500 boards.

Figure 7: Scalability curves for 1000x1000 boards.

Figure 8: Scalability curves for 5000x5000 boards.

Figure 9: Scalability curves for 10000x10000 boards.

4 Conclusions

We can see in the graphs that as expected the speedup and the scalability do not increase linearly, but at a certain point reach an inflection point and start decrease or stabilize itself. This can be attribute to the increasing of the overhead of the splitting. In conclusion, it is important to find the right number of parallelism degree and not underestimate the overhead.