Parallel implementation of Travelling Salesman Problem with genetic algorithm

Matteo Massetti

Università di Pisa (Artificial Intelligence); m.massetti2@studenti.unipi.it

Version July 6, 2020 submitted to Parallel and Distributed System 2019-2020

1. Introduction

This project face up the parallel implementation of a solution to the Travelling Salesman Problem using genetic algorithm. TSP is a routing problem defined as follows: *given a set of cities (x,y) find the shortest path that allows the salesman to visit all the cities once and return to the starting point*.
Genetic algorithm instead is a method for solving optimization problems that is based on natural selection, the process that drives biological evolution.
Three different Implementation are provided:

- Sequential Implementation, used as a baseline for comparing performances
- Parallel Implementation using C++ thread
- Parallel Implementation using FastFlow

2. Design Choices

Genetic Algorithm repeatedly evolves the population obtaining a new one by selecting half of individuals in the previous generation (ones with optimal solutions) and by creating new individuals until the new generation has the same size as the old one. New individuals are created by randomly selecting (according to a probability distribution called fitness) two chromosomes from previous generation, combining them (crossbreeding) and mutate the resulting individual with a certain frequency (mutation rate). After a fixed number of generation the best chromosome found among last generation's ones is chosen as final solution.
So in order to use this kind of algorithm for solving Travelling Salesman Problem, it is necessary to define some key elements, in particular:

- Chromosome: it represent a path which contain all cities only ones and where the last position is meant to be connected to the first.
- Fitness function: this function represent how good is the solution, so each chromosome's fitness score will be inversely proportional to the length of the path.
- Reproduction: the cross-breeding of two chromosomes A and B correspond to removing a part from A and insert missing cities in the order in which they appear in B.
- Mutation: a mutation consist in randomly choose two position in the path and swap them.

Parameters that can be chosen by the user are: number of cities, number of chromosomes, number of generations, mutation rate, number of workers (only in parallel implementation) and maximum values for x and y of each city.
The choice of how introduce parallelism fell on the division of the number of individuals to be created (the half that is created by crossbreeding and mutation) among a set of workers, in particular a scheduler assigns to each worker the number of chromosomes that it must create and each new

individual will be inserted in a common queue that will form the new generation. This schema is graphically represented in Figure 1.

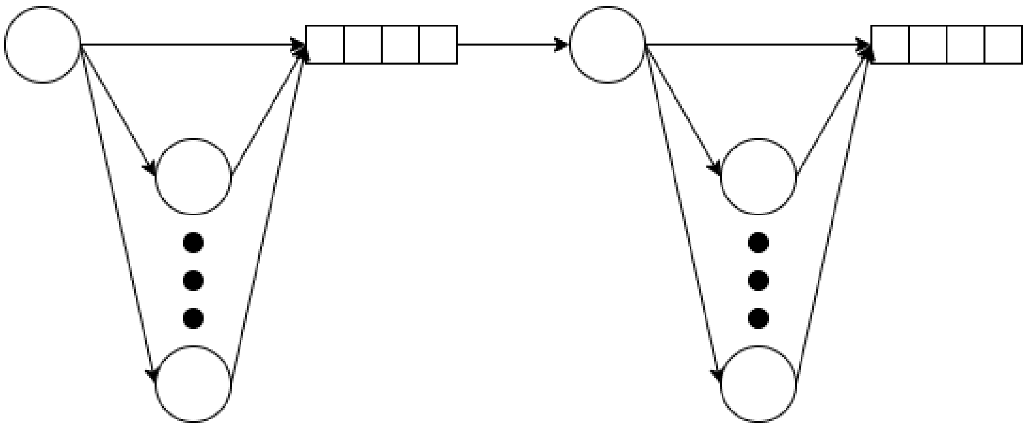
Figure 1. Parallel implementation design of two consecutive generations

3. Implementation

As mentioned in Section 1, 3 different implementations have been reported, they differ only in the way the population evolution is implemented. For this reason shared code was written in a header file called "utilities.h" accessible by all implementations. This file contain class and function that represent key elements of genetic algorithm:

- Chromosome, implemented with a class that has two attributes: path and path length.
- Fitness function, it has been implemented through the *computePathLenght* and *computeFitness* functions, the first compute the path's length using the Euclidean distance while the latter uses the set of lengths of all chromosomes to calculate fitness scores.
- Reproduction, with the *randomSampling* function two random chromosomes are picked up and with *reproduce* they are combined.
- Mutation, implemented with *mutate* function which random switch two position in the path.

In addiction to these there are *findBest* and findNBests functions which respectively find best chromosome in a generation (used at the end to find the final solution) and find the best n chromosomes in a generation (used for evolve the population).

3.1. Parallel implementation using C++ Thread

In the first parallel implementation a set of threads is used to compute in parallel the creation of new population (the part created with crossbreeding and mutation). It is not used in other parallelizable parts, such as the function *findBest*, since the overhead introduced is greater that the time saved.
Each thread receives the number of chromosomes it has to create and each time it produces a new individual it is inserted in a shared queue protected by a mutex. Accesses to the previous generation are not synchronized because these consist only in reading operation so there is no risk of of making data inconsistent.
With a view to load balancing no particular precautions have been taken, as the cost to create a new chromosome is the same for each iteration, so each thread receives a more or less equal number of individuals to create (depending on whether the cardinality of the population is divisible or not by the number of threads).

3.2. Parallel implementation using FastFlow

For this implementation it has been used the parallel_for instruction which allows parallelization of loops with independent iterations. This is implemented on top of the farm building block and it has

a master-worker structure where the master is the Scheduler of loop iterations.
As before only the creation loop has been parallelized since it was not convenient to parallelize anything else. Also in this case accesses to the shared queue are synchronized by a mutex, while the old generation is not protected because only reading operations are carried out.

4. Performances evaluation

In order to compare performances of parallel implementations, two measures are adopted: completion time and speedup.

$$CompletionTime = T_{end} - T_{start}$$

Where T_{start} is the time when the computation start and T_{end} is the time when it finish, completion time so is the time between the start and the end of the computation.

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

Where T_{seq} is the completion time spent in sequential implementation and $T_{par}(n)$ is the completion time spent in the parallel implementation with parallel degree equal to n.

The genetic algorithm imposes constraints between a generation and the next one, in particular to generate new individuals is necessary the probability distribution of the previous generation. These constraints are graphically represented in Figure 2, where each node represent a task, such as creation of new chromosome or finding best one (last layer), and arcs connect pairs of nodes where the output of one is necessary for the computation of the other.

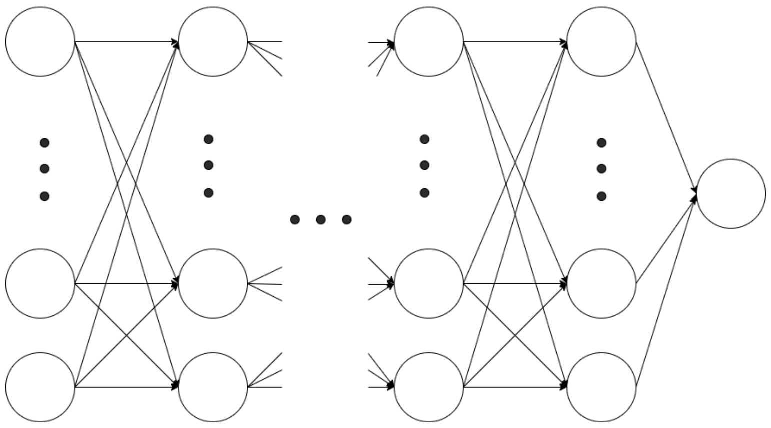

Figure 2. Direct Acyclic Graph describing dependencies in computation

Supposing that the time spent in creation of a new chromosome is t while the time spent in finding the best chromosome in a generation is t' and suppose to have k generations and n chromosomes for each generation; it is possible to define:

- Work, total amount of time spent executing the whole application which correspond to $tkn + t'$
- Span, cost of the longest path (called critical path) which correspond to $tk + t'$

These two elements can provide an upper bound of the achievable speedup, indeed if it were possible to use an infinite number of parallel units, it would still be necessary a time equal to the critical path. If t' is a negligible time it's possible to write:

$$speedup(n) \leq \frac{tkn}{tk} = n$$

This bound represent the constraints of computing all n generations one after the other without the possibility of computing two of them at the same time.
This upper bound however is overoptimistic because it doesn't take into account the overhead, in other words it doesn't take into account costs that derives from the parallelization of code which introduces operation like creation/elimination of thread, synchronization in critical part, communication and so on. Moreover, as it has been implemented, the creation of n new individuals is not completely parallelized because only $\frac{n}{2}$ chromosomes are created in parallel, and this further lowers the achievable speedup.

5. Results and Comparison

As mentioned in Section 4, Speedup and Completion Time are the metrics chosen for comparing different implementations, in Figure 3 are graphically reported values concerning Completion Time while in Figure 4 the ones about Speedup.
The results refer to an execution with 8000 chromosomes evolved for 2000 generation.

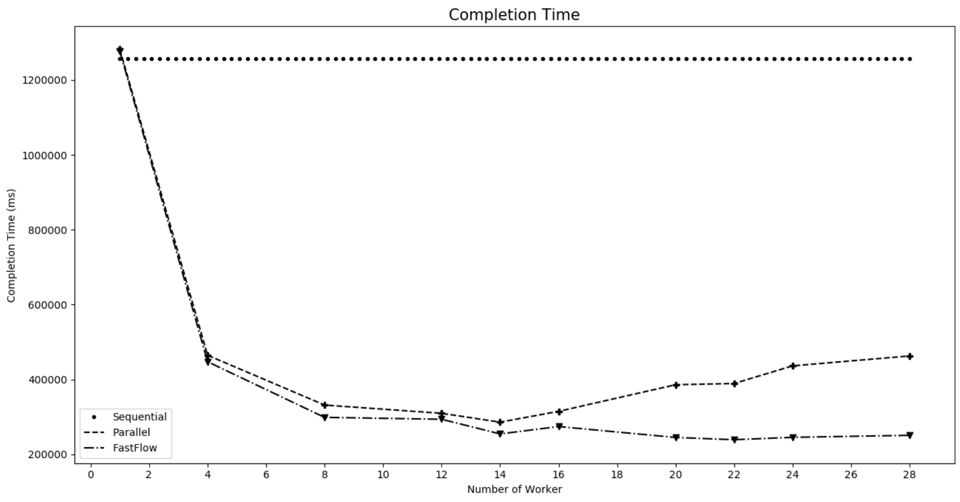

Figure 3. Completion Time plotting

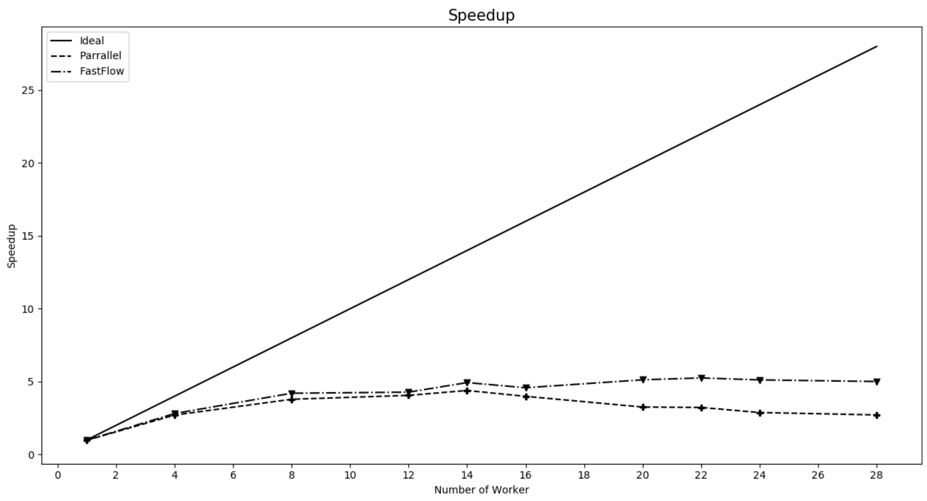

Figure 4. Speedup plotting

From these graphs it is possible to observe a situation in which with a degree of parallelism equal to 1 the sequential implementation obtains better performances, but increasing the number of processing units better times are achieved by parallel implementations up to the point where going further performances worsen.
This is caused by overhead, indeed with parallelism degree equal to 1, even if there is only one running activity, mechanisms for managing parallel computation are introduced and this lead to longer computation time. Beyond a certain point instead tasks assigned to each worker are so simple that they require less time to be computed than the time spent for setting up and manage concurrent activities, leading to increase completion time.

Another interesting thing that came out from these graphs is the fact that FastFlow implementation outperform much better than the implementation with c++ threads. In particular the FastFlow implementation can reach a speedup of 5.25 with a parallelism degree equal to 22 while the other can only reach a speedup of 4.24 using 16 workers.
The causes can be found in the nature of FastFlow, which is an optimized framework for implementing parallelism, while the implementation with c++ threads require a higher time for management operations of parallel activities.

The achieved speedup is much lower than the ideal one, this because the parallel portion of the program is less than 50% of total amount of work leaving sequential most of the program. Indeed if instead of copying half of previous generation to the next one, all new individuals are generated with crossbreeding and mutation, the achieved speedup is much higher. As reported in Figure 5, FastFlow implementation was able to achieve (with same number of chromosomes and generations) a speedup equal to 13.93 using 32 workers and the c++ thread implementation a speedup of 9.22 using 20 workers.
With this solution it is possible to obtain better performances in terms of time, but on the other hand the effectiveness of the algorithm in finding the best path drops drastically, in fact given that all new individuals are generated by probability distributions and random draws an huge number of generations is necessary to achieve good results.

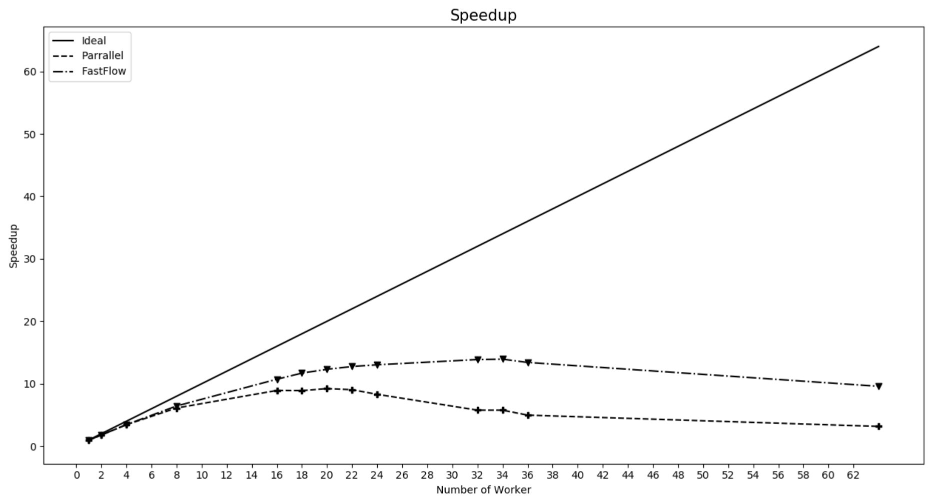

Figure 5. Speedup plotting

6. Algorithm Effectiveness

The Travelling Salesman Problem is a non-polynomial hard problem, in particular with n cities there are $n!$ possible path.
With the provided implementation it's possible to find the optimal path in a reasonable amount of time, in Figures 6 is reported the evolution of the best path found in a situation with 12 cities, 400 chromosomes for generations and mutation rate equal to 0.01. It possible to see how just after 50 generations the algorithm was able to find a good solution and with other 100 it was able to find the best one. So with just 400 path evolved for 150 generations a solution has been found, without having to check all the $4.79 \cdot 10^8$ possible path.

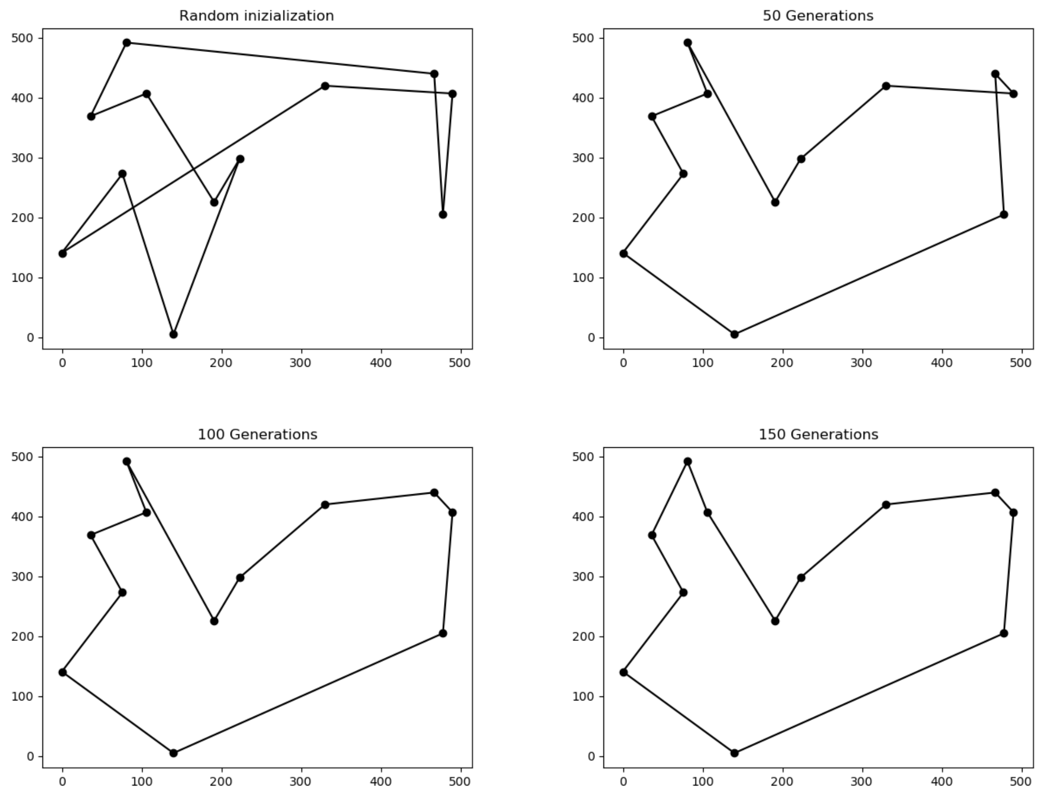

Figure 6. Best path found at different generation