UNIVERSITY OF PISA,
PARALLEL AND DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS

Parallel genetic algorithm for traveling salesman problem

Andrea Rosasco

Computer Science (Artificial Intelligence)

July 21, 2020

CONTENTS

1 Introduction

The aim of the project is to figure out how to efficiently parallelize a Genetic Algorithm (GA) applied to the Traveling Salesman Problem (TSP) and to implement the solution both with the FastFlow framework and the standard Còà.++ parallel mechanisms. The performance of the two implementations (e.g. speedup, scalability) are then compared and discussed.

Before starting to discuss how to solve the problem let's lay down a common terminology

1.1 Traveling Salesman Problem

This is how we define the Traveling Salesman Problem that we will try to efficiently solve during the rest of the project.

Definition 1. Given a complete weighted graph $G = \{V, E\}$ and a start node $s \in E$ the Traveling Salesman Problem consists in finding the shortest possible path p_* which starts and finishes at the same node s and visits all other nodes exactly once.

It exists another formulation of the problem which deals with non-complete graphs, but since a graph can be completed without changing the TSP solution by adding arbitrary length edges between the nodes that are not connected, we will stick to Definition 1.

It's important to state that the TSP problem is NP-Complete and that one of the fastest known exact algorithm takes $O(n^2 2^n)$. Also, the number of possible path defined by a TSP of dimension n is $\frac{(n-1)!}{2}$.

1.2 Genetic Algorithm

A genetic algorithm is a stochastic search algorithm which mimics Darwin's theory of evolution. This kind of algorithm is not guaranteed to find the best possible solution but it can efficiently find near optimal solutions. In this particular context a candidate solution is called chromosome. Let us now define the algorithm the basic components:

Fitness function
: Function which takes as input a chromosome and return a goodness score.

Crossover operator
: Function which takes as input two chromosomes and recombines them into two new chromosomes.

Mutation operator
: Function which takes as input one chromosomes and generates a new one by applying a random mutation

To create a genetic algorithm for our problem we just have to define these three functions. Once we have them we can run the algorithm

Algorithm 1 Generic Algorithm

```
1: population ← random population of k chromosomes
2: for i = 0 to g do
3:     offspring ← evolve(population)
4:     population = select(population, offspring)
5: end for
```

In Algorithm 1, the function $evolve$ generates k new chromosomes by applying, depending on a given probability, the crossover operator or the mutation operator. After that, the $select$ select takes the best k chromosomes among the old and the new population using the fitness function as a metric.
The main parameter of this algorithm are the population dimension k and the number of generations (iterations) g.

2 SOLUTION

In this section we will see the most important design choices in the implementation of the sequential version. Then, by measuring the time spent by such implementation, I will look for an efficient way to parallelize it.

2.1 SEQUENTIAL DESIGN

The implementation of the sequential version of the algorithm is important mainly for two reasons:

- The main components of the sequential implementation are shared by the parallel one. Developing them in the context of a sequential program is easier.
- Measuring the performance of the sequential version is the first step in developing a parallel one.

TSP

The first step is the implementation of the problem. What I did was to represent the graph as a symmetric matrix initialized with random values between 2 and 100 with zeros on the diagonal.

To understand whether the algorithm finds the optimal solution, one random path is always initialized with weight 1 between each node. This way we know that an optimal unique solution is present and we can evaluate the ability of the algorithm at finding it. If the number of nodes is n we know that a single optimal path with length n is present in the graph.

CHROMOSOME AND FITNESS FUNCTION

Without loss of generality we assume the starting node to always be the first one (i.e the one at index 0 inside the matrix). Then, we can represent a chromosome by a path of $n-1$ nodes.

To evaluate one path the fitness function just need to sum the weights of the edges connecting the nodes in the path, plus the weights of the edges from and to the starting node.

EVOLUTION OPERATORS

There are a lot of different ways to implement the two evolution operators. This is what I did:

Crossover
: consists in taking a random section of the first chromosome and changing the order of the nodes in that section to match the one of the same nodes in the second chromosome. The same thing is done for the second chromosome using the same indices.

Mutation
: consists in swapping the position of two nodes inside a chromosome.

Also, I set the crossover probability to 70%.

2.2 SEQUENTIAL PERFORMANCE

I measured the performance of the sequential implementation running the code on the Xeon Phi machine provided by the University of Pisa and timing different sections of the code. The parameter used to run the code are the following:

- Generations (g): 1000
- Population size (k): 10,000
- Number of nodes: 20

These first two values were chosen to get a more stable average measurement of the time spent to process a chromosome. The number of nodes was set to 20 to show how the algorithm can efficiently solve the problem exploring just a fraction of the $6.08e16$ possible solutions.

Important: the execution time doesn't change linearly with the parameters g and k. This is because the $selection$ step is done trough a min-heap and, on average, popping an element from a binary heap takes $O(\log n)$. What's more, the number of insertion and deletion from the heap is higher during the first generations.

	Time for k = 1000 and g = 10,000	**Time for k = 10,000 and g = 1000**
Total	21,510,769 us	25,290,849 us
Evolve	10,774,351 us	10,931,503 us
Select	10,722,666 us	14,591,934 us

Table 2.1: Time measurement of the sequential implementation for different values of g and k. Despite the number of operations is the same, the different heap dimension causes the first version to be faster.

2.3 Parallel Solution

As we can see from Table 2.1 the total execution time is almost equally split between the functons *evolve* and *select*.

My first idea was to create a pipeline of two stages: a farm of *evolve* and a farm of *select*. Using the notation presented during the course:

$$Pipe(Farm(Seq(evolve)), Farm(Seq(select)))$$

Using this architecture the chromosomes from the selection stage could be directly fed back to the first pipe stage with no need to wait for all the chromosomes from the previous generation to be processed. Unfortunately this architecture has a problem.

To operate exactly as the sequential algorithm, all the workers parallelizing *select*, should access to a synchronized shared heap of best chromosomes. Unfortunately, when sharing a state between more workers the speedup is limited to $1 + \frac{t_{local}}{t_{state}}$. What a worker does is to check if a chromosome can be inserted in the heap, and if it can, pops the worst chromosome and insert the current one. This means that $t_{local} \approx t_{state}$ so the maximum speedup of that stage is about 2.

Actually, to be more precise, we should also consider that not all of the chromosomes get inserted in the heap. Most of them are just tested, hence they just have to acquire a shared mutex to read from the heap. Given the peculiarity of the situation I decided to run some tests to see how costly is the bottleneck.

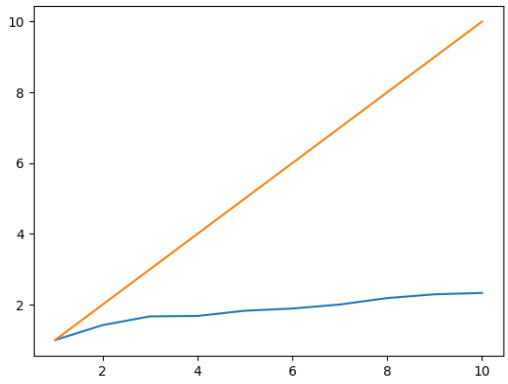

Figure 2.1: Scalability of the workers implementing *evolve* with a shared state

As we can see from Figure 2.1 we clearly have a bottleneck that prevent us to efficiently parallelize the algorithm.

An alternative solution might be to remove the shared state and let each worker to work on their own heap. Then, to maintain consistency, the collector should merge all the heaps and feed the resulting one back to the workers. This wouldn't work. What this implementation is doing is just postponing the problem: the collector would become the bottleneck and, as before, it's not parallelizable.

It seems that there is no way to keep the chromosomes population consistent without having a bottleneck somewhere. Then, to achieve a better speedup, we drop the constraint of having a unique best population.

With that out of the way we can simply parallelize the problem by replicating the entire algorithm nw times and executing each instance on a different worker. This is actually the normal form and, other than be simple to implement, it has linear speedup with minimum waste of resources. I'm going to implement it with Fastflow and with the standard C++ concurrency mechanisms and see how the two solutions perform in practice.

3 Fastflow Implementation

The Fastflow implementation consist just in a farm and in a modified collector. Each worker execute the for loop in Algorithm 1 with parameters g and $\frac{k}{nw}$ (i.e. the population size divided the number of workers). When a worker finishes its search it sends the best found result to the collector, which in turn select the best among all the solutions and returns it as the final output.

I run the program with nw ranging from 1 to 256 and measured the performance.

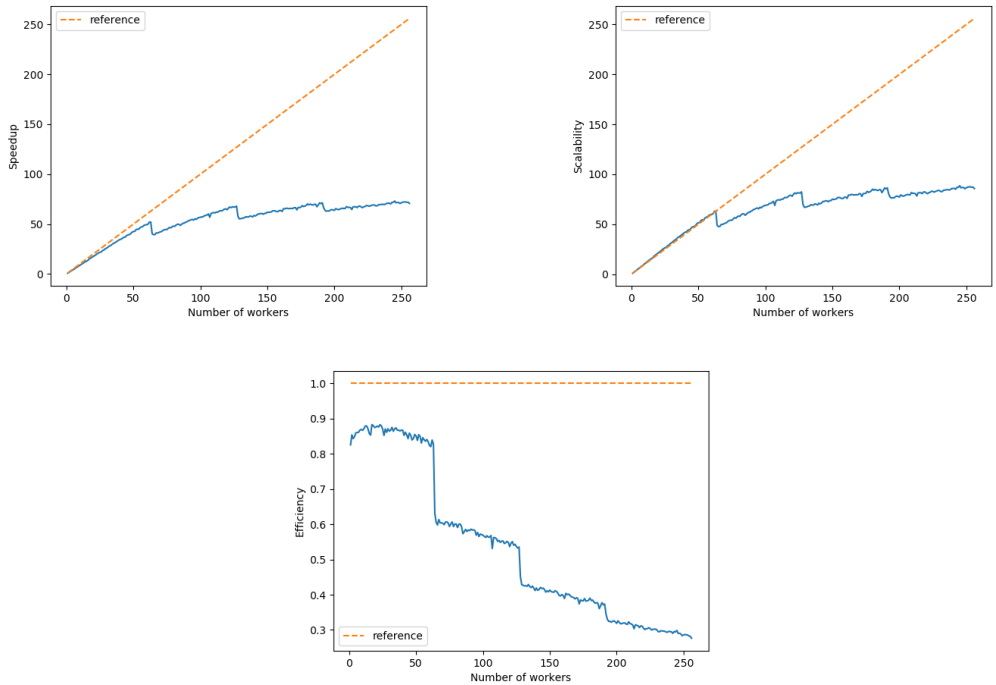

Figure 3.1: Speedup, scalability and efficiency of the Fastflow parallel implementation with $g = 1000$ and $k = 10,000$

.

As we can see from the graphs in Figure 3.1 it managed to achieve almost linear speedup up to 64 workers. After that we start using hyper-threading and we have a sudden decrease in speedup due to the higher time spent by the workers sharing the same cores.

This happens at interval of 64 workers as the Xeon Phi machine has 64 cores with 4 way hyper-threading. The same behavior can be observed in the efficiency graph where we have a big drop at 64 workers and smaller drops at 128 and 192.

It's also worth noting how, in the scalability graph, we have a bit of super-linear speedup for the first 64 workers. The reason is probably that using more cores we have more L1 cache available and less cache misses.

4 STD C++ IMPLEMENTATION

The standard C++ implementation is actually pretty similar to the Fastflow one, except for the parallel mechanisms used to implement it.

My first attempt was to use the parallel function *async* to spawn *nw* workers, each implementing an instance of the genetic algorithm. After that I would wait for the results while keeping track of the best one. This is how this implementation's speedup looks like.

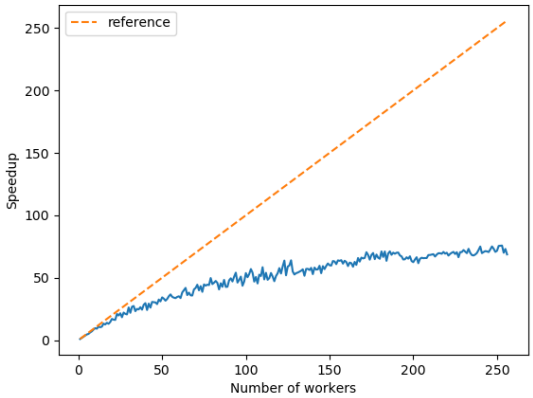

Figure 4.1: Speedup of the parallel implementation using *async*

Clearly something isn't working as it should work. After thinking about what could cause the low speedup and the "noisy" graph, I realized that one thing that Fastflow was automatically doing was pinning the workers to the cores. I decided to do the same on the std C++ version.

To implement the thread pinning I had to switch from using *async* to using *thread*, since the returned thread handle can be used to pin the worker to a core. I mapped the first worker on core 1, the second on core 2 and so on. This works on the university machine since the "real" cores have index from 0 to 63 but with other architectures it may be different. This is not a big portability problem though, as a script like the one shipped with Fastflow can be run to get the correct mapping.

After pinning the workers we got the speedup and scalability shown in Figure 4.2.

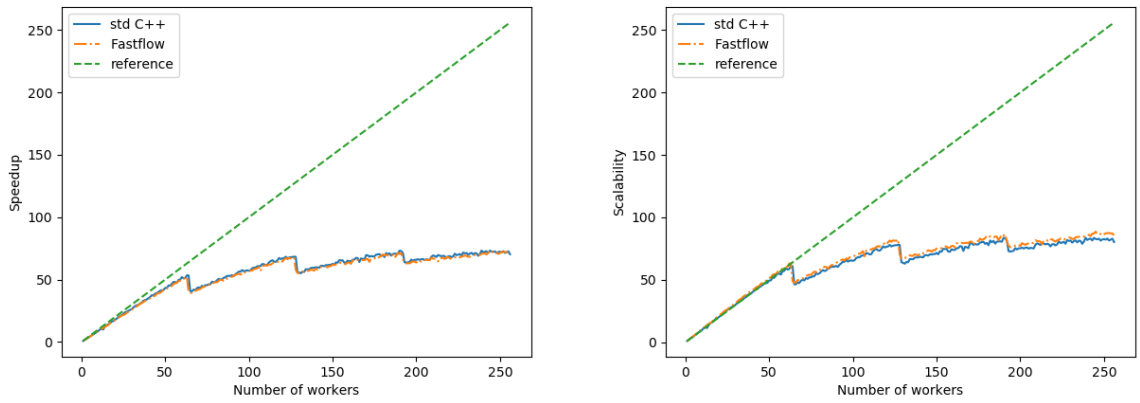

Figure 4.2: Speedup and scalability of the parallel implementation using *thread* compared with the Fastflow implementation.

It's surprising how the performance of the two parallel implementations are almost exactly the same. That's probably because the underlying architecture, other than being the same for both implementation, is also very simple, giving not much room for divergence in performance.

5 Result Evaluation

In this section we want to evaluate if we can use the speedup introduced by the parallelization to actually find the best path faster.

To evaluate this, I slightly changed the interface so that running the application with k and nw means having nw parallel populations of k chromosomes. This way, changing the value of nw doesn't change (a lot) the computational time.

I run the experiments with $k = 2000$, $g = 100$ and $p = 20$ using each time a different seed. The results are shown in the following table.

	1 worker	**50 workers**	**100 workers**
Time	1.361 s	1.541 s	2.018 s
Best solution ratio (on 100 runs)	6%	91%	97%

Table 5.1: Performance of different runs of the algorithm varying the number of workers.

As we can see, increasing the number of parallel instances of the algorithm that are run in parallel, we can increase the percentage of success. In this setup, to measure the speedup, we should compare our algorithm to a sequential one with the same success rate.

Unfortunately, I wasn't able to do it because, to measure the success rate, its required to run the algorithm at least 100 times and the time of a single run with enough chromosomes and generations to achieve that performance it's too high.

I measure the success rate for the sequential algorithm running with $k = 10,000$ and $g = 1000$ and I got the best path 56% of the times, with every run taking about 30 seconds. This shows how much this application benefits from parallelization.