Parallel implementations of a genetic algorithm for the Travelling salesman problem

Lapo Toloni, 568235
l.toloni-at-studenti.unipi.it

June 14, 2020

1 Problem description

In this report we analyze in theory and in practice the resolution of the famous NP-HARD optimization problem TSP (travelling salesman problem) via a genetic approach.
Given a weighted graph $G = (V, E)$ and its weight function $w : E \longrightarrow \mathbb{N}$, a solution of TSP is an Hamiltonian tour of G whose cost is minimum. All along this report and in the code we will represent these Hamiltonian tours of G as permutations ρ of all the nodes of V (repetitions are not allowed!). An Hamilton tour of minimum cost is a permutation ρ^* of V such that the sum of the weights of successive nodes in ρ^* is minimal, namely:

$$\min_{\rho} (\sum_{i=0}^{|V|} w(\rho(i), \rho(i+1))) + w(\rho(|V|), \rho(0))$$

Genetic algorithms are iterative process that aim to find a feasible near-optimal solution to hard optimization problems deploying heuristics that emulate the behaviour of biological systems.
Such a complex system is simulated by keeping in memory a collection of chromosomes (i.e. candidate solution) and by trying to improve their fitness value at each iteration.
Every genetic algorithm has a common flow that repeats every iteration (usually called generation/epoch in this field's jargon)

- $crossover$: take chromosomes in the current population two-by-two and for every such couple produce two new chromosomes similar to the parents as a result of some combinatorial process.
- $mutate$: take chromosomes in the current population one-by-one and change some part of it.
- $evaluate$ population: exploit a predefined fitness function to assess how well our population is behaving.
- $selection$: replace ill-fitted chromosomes with copy of the current best ones.

Note that both the first two operations above are applied with some given probability for each iteration.

2 Development of a framework to work with genetic algorithms

In order to ease the development of a working solution for this work and to be more open to future enhancements we built a tiny framework. The file *genetic.hpp* contains a templatic C++ class, named `Genetic_Algorithm` parametrized by the type of the population, the type of a single chromosome and the type of the used fitness function. `Genetic_Algorithm` offers a lightweight interface and few protected members to be inherited by its subclasses:

- a constructor asking for the maximum number of epochs, the number of chromosomes in the population, the size of a single chromosome and a fitness function.
- a function `void run()` to start the algorithm on the given instance and a function `get_current_opt()` returning a pair $(optimum\_value, optimum\_chromosome)$.
- four protected functions to obtain the flow presented in the previous section, namely: `crossover, mutate, eval_population, selection`. These `void` functions all require as input parameters the start and the end of the range of chromosomes of the population to be deal with by such function calls.

3 Sequential algorithm for genetic TSP

3.1 Input graphs

In order to test the algorithm instead of working on pre-computed graphs that can be scraped from the web, we developed a little class `TSP_Graph` in the module *tsp_graph.hpp* to generate random graphs. Object instances of `TSP_Graph` are completely connected (each node has an edge to any other node, no self-loops allowed) undirected random weighted graphs. Here the randomness is introduced by the process exploited to assign weights to edges. In fact each edge is given a weight in the interval $[1, 9]$ drawn i.i.d. from a uniform distribution. In memory the graph is stored as an array of adjacency lists that is as a `std::vector<std::vector<int>>`. Given the properties of the used graph we have an upper triangular matrix that is every element below the diagonal is zero. We did so to spare some time in process of generaing the random graph.

3.2 Problem representation

We decided to represent the population of chromosomes as a `std::vector<std::vector<int>>` and so the single chromosome as a `std::vector<int>`. Every element of a chromosome is a natural number $\in [0, chromosome\_size]$, where the upper bound here used is one of the parameters used by constructor of `Genetic_Algorithm` and is also the same parameter that defines the size of V for the `TSP_Graph` to be generated for this run of the algorithm.

3.3 Algorithm initialization

The initialization phase consists in the generation of n random permutations that will represent the starting population.

3.4 Crossover

Many well-refined algorithms to implement crossover functions for genetic TSP can be found in the literature. However we preferred to implement a customize solution. We want to note that the implementation provided does not want to produce better iterates but its main focus is not being too simple and computationally light.
The procedure scan the population by rows, two-by-two, and copy the central part of the first chromosome into the central part of the second. Since this procedure may yield candidate solutions that are not feasible (i.e.: chromosomes with repetitions) we apply some sort of sanitization. Through a second linear scan of the two permutations we count numbers appearing twice and we greedly replace them with numbers that never appear in the newly produced children-permutation.

3.5 Mutate

It simply consists in a pointwise swap of two elements of a chromosome.

3.6 Evaluate population

We scan the matrix representing the population row by row and, using the given fitness function, we compute the cost of the tour represented by that chromosome. Successive elements of a single chromosomes are used as index to access the respective edge weight in the adjacency list of the graph. The fitness value of each chromosome is stored in a vector `chromosomes_fitness`.

3.7 Selection

It consists in a linear scan of the vector `chromosomes_fitness`. During the scan we keep track of the minimum cost tour and of the maximum cost tour in the current generation. In the end we update the current global optimum tour we found so far. Moreover we inject the current global optimum in the current population in place of the worst chromosome of the current generation.

3.8 The algorithm as a whole

We can summarize the algorithm as

```
while current_epoch < max_epoch
  crossover
  mutate
```

```
evaluate_population
selection
```

4 Parallel solutions design

Now that all the basic steps that need to be performed by the algorithm were layed down we are able to see from which angle attack the problem. The parallel kind of computation that emerges from this application is the data pallel one.

If we look at a single iteration of the algorithm we will see that the subtask of performing a mutation, a crossover and a fitness evaluation on a subgroup of chromosomes is independent from the task of doing the same operations on a different subgroup.

Indeed, when we look at the single iteration, the computations here involved are embarassingly parallel. This is implied by the fact that it is just required to split the single-iteration-task: "manipulate the current population and evaluate it" in many equally sized subtasks and then give each subtask to a different worker.
Note that we are not yet taking into account the $selection$ part of the algorithm. If we choose to parallelize this part too we will introduce the classical overhead present in data parallel computations that is the one associated to the time spent in merging the results coming from the subtasks. Otherwise by choosing to let a single thread perform all alone the selection task we introduce some form of overhead in the time we have to wait for all the subtasks becomes completed before starting doing selection. Moreover this single "selecting-worker" will have to deal with a larger chunk of data. Summarizing the introduction of parallelism in the implementation of genetic tsp aims to reduce the latency of the task corresponding to a single iteration of the whole genetic algorithm.

Finally we can say that this kind of parallelism should reduce the latency of two consecutives iterations and so the whole completion time of the algorithm.

5 Performance modelling

Having described the sequential version of the algorithm and having presented a tentative design for the parallel implementations, we paved the way to produce some assessment about the performances of the model.

5.1 In theory

Let $n = population\_size$, $m = chromosome\_size$, $I = max\_epochs$, then we can state:

$$t_{seq} = I * (t_{cro} + t_{mut} + t_{fit} + t_{sel})$$

where t_{cro}, t_{mut}, t_{fit}, t_{sel} are respectively the time spent in performing crossovers, mutations, population evaluations and selections.

- $t_{cro} \approx \mathcal{O}(m) * \mathcal{O}(n) \approx \mathcal{O}(m * n)$
- $t_{mut} \approx \mathcal{O}(1) * \mathcal{O}(n) \approx \mathcal{O}(n)$
- $t_{fit} \approx \mathcal{O}(m) * \mathcal{O}(n) \approx \mathcal{O}(m * n)$
- $t_{sel} \approx \mathcal{O}(n)$

Glueing pieces together, we obtain

$$\begin{aligned} t_{seq} &= I * (t_{cro} + t_{mut} + t_{fit} + t_{sel}) \\ &\approx I * ((\mathcal{O}(m * n) + \mathcal{O}(n) + \mathcal{O}(m * n)) + \mathcal{O}(n)) \\ &\approx I * (\mathcal{O}(m * n) + \mathcal{O}(n)) \\ &\approx I * \mathcal{O}(m * n) \\ &\approx \mathcal{O}(m * n) \end{aligned}$$

With the last $\approx$ coming from the fact that I is usually orders of magnitude less then both m and n.

Recalling that we will deploy parallelism only for crossovers, mutations and evaluation, what we are trying to enlight with this reasoning is that even if we chose to not parallelize the selection phase the performance loss should remain negligible since its computational complexity is cancelled by the complexity of the three previous tasks.

5.2 Ideal Parallel performances

Following everything we stated so far about

- the points from where the problem of parallelizing our sequential algorithm can be attacked;
- t_{cro}, t_{mut}, t_{fit}, t_{sel} theoretical time complexities;

Then assuming that p is the number of workers at disposal for our parallel computation, we can define the ideal parallel time:

$$t_{par\_ideal} = \frac{t_{seq}}{p}$$

and we will use it as benchmark for our experimental results.
We define the ideal $speedup$, the ideal $scalability$ and the ideal efficiency using the defini-

tion of such measures and our definition of t_{par_ideal}:

$$s(p) = \frac{t_{seq}}{t_{par}(p)} = \frac{t_{seq}}{t_{seq}/p} = p \quad scalab(p) = \frac{t_{par}(1)}{t_{par}} = \frac{t_{seq}/1}{t_{seq}/p} = p \quad \varepsilon(p) = \frac{t_{seq}}{p * t_{par}(p)} = \frac{t_{seq}}{p * t_{seq}/p} = 1$$

6 Parallel implementations: structure and details

We provide three parallel implementation of the algorithm. Each one is implemented in C++17, two rely on native C++ threads while the third one exploits the FastFlow framework. The implementations are thought to avoid race conditions between workers dealing with different parts of the data. This let us avoid the explicit use of locks and any other special synchronization mechanism except for the ones hidden behind the FastFlow front-end and the one used to manage the `condition_variable` used in the task queue needed by the thread pool.
Each implementation is available as an .hpp header file in the directory ./include. The directory ./src contains .cpp file that are the drivers of the relative .hpp files. To compile the project is enough to run the bash script $compile.sh$ located in the root of the project folder. While to run a default batch of experiments use the bash script $run.sh$. More details on the directory-structure of the project and on how to play with this piece of software can be find in $README.md$ file.

6.1 Naive C++ thread implementation

MEASURE TIME TAKEN TO SPAWN THREADS and OVERHEADS.!!!

6.2 Threads pool C++ implementation

6.3 FastFlow implementation

7 Experimental validation

marta

8 Conclusions and final remarks

fine