Parallel Solution for TSP with Genetic Algorithm

Valeria Montagna
505481

University of Pisa — A.A. 2019/2020

The Problem

The Travelling Salesman Problem (TSP) is a well-known problem which proposes a list of cities with their distances between each other and has the purpose to find the minimum path that, from a starting city, acrosses every city exactly once and returns to the starting point. It is required to find a parallel solution that solves TSP, using the following genetic algorithm.

Algorithm 1: `GeneticAlgorithm`

```
read initial population P ;
for all generations do
    select random parents, enforcing fitness ;
    apply cross-over and mutation according to probabilities ;
    add new generated individual to new population ;
end
```

Every city can be numbered as an integer. All possible permutations of n distinct integers are considered to be *tours* between n cities. At the same time, from a genetic algorithm's point of view, a tour can be viewed as a chromosome. All chromosomes have associated a score called *fitness* that says how good they are as solutions of the problem: in TSP, a nice fitness score is the length of the tour related to chromosome. A group of m chromosomes constructs a *population*.

Crossover and *mutation* are operations that outputs a new chromosome by applying a series of transformations to one or two chromosomes in input. The idea behind a genetic algorithm is that a starting population evolves throught these transformations, passing from a generation to another, until it is found the best solution, according to a certain criteria.

A crossover operation can be implemented selecting from a chromosome $c1$ all numbers between a random range $[i, j]$ and taking all missed numbers from another chromosome $c2$ in the order at which they appear in it. Then, the very same process is repeated revering roles of chromosomes $c1$ and $c2$. At the end, they are obtained two new chromosomes.

Mutation operation can be executed simply swapping the positions of two random numbers of an incoming chromosome.

Design

The purpose of this project is to outline a parallel solution that implements the algorithm described before. To that end, there's a first *sequential version*, used as a baseline to elaborate a parallel version.

Two parallel solutions will be presented:

1. A first parallel solution is based on the usage of standard library's threads.
2. A second parallel solution is designed with FastFlow, a parallel programming framework.

Implementative choices

There's a common base that affects all the solutions. First of all, it is assumed that distance between a couple of cities is independent from the direction of the tour, so $d_{i,j} = d_{j,i} \forall i, j$. This implies that all solutions solve *symmetric TSP* and the graph that models the problem is undirected.
Weighted matrix that represents distances between any couple of cities could be built in two different ways:

1. Either *reading* a text file reporting weights related to edges.
2. Or, given the number of nodes and the maximum weight between two nodes, *generating them randomly*, according to a given seed.

Another universal decision is about how to select candidate chromosomes that probably partipate to crossover and mutation. The criteria chosen is called *roulette wheel selection* and is the following:

Algorithm 2: `Roulette Wheel Selection`

```
for all members of population do
    sum += fitness[i] ;
end
for all members of population do
    probability[i] = sum of probabilities + (fitness[i]/sum) ;
    sum of probabilities += probability[i] ;
end
number = random between 0 and 1 ;
for all members of population do
    if number > probability[i] and number < probability[i+1]
     then
        return i ;
    end
end
return size ;
```

The first for computes the total sum of the fitness. The latter is exploited by the second for to compute the cumulative probability distribution: this is going to favor the selection of individuals with better fitness (i.e. with a lower value).
Another thing in common between the various solutions is the model adopted for the replacement of the new individual in the population. Each new child will be part of the next generation and is not immediately available for reproduction. They will actually join the population at the next loop iteration, only if their fitness value is better than the current ones (i.e. lower).
Parameters like the probability of mutation/crossover, the number of individuals, the number of cities and so on, are provided by the user, via line command.

Other choices (not adopted). Another choice taken into account, but not adopted, concerns the adoption of a stopping criteria. A genetic algorithm will stop to run when there will not be a new best solution for 200 *generations*.
This choice, however correct, does not allow an easy comparison between the various parallel solutions proposed. As we will see, implementations work on subpopulations: decreasing the number of individuals to work on, one converges first to a local optimum. With a degree of parallelism of k, the risk is to obtain a fast parallel solution at more than k. Consequently, it would not have been possible to compute the quality parameters shown below, namely speedup, scalability and so on. Hence the decision to maintain a fixed number of iterations, here defined as generations.
Another choice then discarded is the criterion that determines which genetic transformations to subject the selected individuals. The most intuitive choice was the following: generate a random number and see if it is less than or equal to the probability of obtaining a mutation / crossover.
To give more stability to the completion times of the parallel solutions, I decided to give a fixed value to

the number of transformations that must occur in each generation. This number is however dependent on the probability of mutation / crossover, as it is computed as follows:

$$\#\text{transformations} = \#\text{individuals} \cdot \mathcal{P}(\text{transformations})$$

Parallel Solution

Criticality of the algorithm. For designing the parallel architecture an initial study about algorithm's phases in the sequential version was carried out. The sequential solution can be divided in the following steps:

- *Build* - This phase outputs distance matrix.
- *Breeding* - This stage actually selects a portion of existing population to breed a new generation. It can be further divided into some sub-phases:
  - *Selection (for mutation or crossover)* - This phase computes to which individuals of the population algorithm's transformations are applied.
  - *Mutation or crossover* - Mutation or crossover operation.
- *Evaluation* - This phase takes the new chromosomes, combines them with the actual population and, having ranked it, cuts the excess of individuals with higher fitnesses.

All together, the algorithm to be parallelized will be the following:

Algorithm 3: `Step-by-step sequential solution`

```
Build distance matrix ;
for all generations do
    repeat
        Select chromosomes ;
        Apply mutation ;
    until done all mutations;
    repeat
        Select chromosomes ;
        Apply crossover ;
    until done all crossovers;
    Evaluate new chromosomes ;
end
return Best solution for fitness-value
```

A first analysis about how much time is spent in these phases with respect to total sequential time execution was accomplished. More tests were carried out, varying the significant parameters of the algorithm, mainly cities number or population's cardinality.

Times divided between GA phases (30 generations)						
Population	Nodes	Prob. Mutation	Prob. Crossover	Build	Breeding	Evaluation
1000 individuals	50	30%	70%	62 μs	272556 μs	25659 μs
	100	30%	70%	381 μs	328981 μs	23787 μs
	500	30%	70%	6988 μs	2521495 μs	36150 μs
	1000	30%	70%	50289 μs	7722050 μs	55165 μs
3000 individuals	500	30%	70%	14850 μs	9210963 μs	147073 μs
	1000	30%	70%	35930 μs	27889882 μs	159808 μs
5000 individuals	500	30%	70%	11406 μs	19166669 μs	347938 μs
	1000	30%	70%	41381 μs	45927578 μs	265457 μs

As we can see, with few chromosomes and 1000 population members, the operations that impacts the most on the algorithm are *Breeding* and *Evaluation*.
Investigating more on this phenomenon, however, we discover that indeed the phase whose cost increases as the parameters of the GA is the *Breeding* phase. This partly depends on the probability percentages of the genetic operators. In fact, in terms of single cost, the crossover operation is potentially the most expensive action of the whole algorithm. What gives it this primacy is the fact that its cost increases significantly with the increase in the number of nodes. If we have a huge population as input, then, combined with its probability, this operator alone could really become the algorithm's bottleneck.

Impact on single-times of genetic operators (30 generations)			
Population	Nodes	Crossover	Mutation
50 individuals	100	4 μs	1 μs
	500	63 μs	2 μs
	1000	450 μs	2 μs
	2000	916 μs	20 μs

On the other hand, *Evaluation* could increase significantly in weight when the population and the chances of transformation are high, even more than *Breeding*. For example, with a population of 5000 and a mutation and crossover probability of 100% and 100% respectively, hopefully at this stage the algorithm will have to sort a vector of 15000 items, and then erase 10000 of them. Not to mention that, in the hypothesis of dealing with long chromosomes 2000, only crossover phase at 70% will last on average something like $916 \times (5000 \times \frac{70}{100}) \times 30 = 96'180'000\,\mu s \approx 96,1\,s$.

Designing a solution. The concrete solution to the problem is based on an **embarrassingly parallel computation**. If we look at a single generation, we will see that *Breeding* and *Evaluation* subtasks could be performed independently on different subgroups of chromosomes. With this assumption, it will be as if each entity were working on independent solutions.
Unfortunately, choosing to parallelize the next generation selection part or not will have disadvantages in any case:

- Choosing not to parallelize this part, would lead to introduce overhead for waiting for the iteration to be completed on subpopulations and moreover, a single entity will have to deal with analyzing $(nCros + nMut) \times nw$ individuals and deciding whether to include them in a large population n.
- Choosing to parallelize this operation, however, would introduce the overhead associated with the time spent merging the sub-results from the subtasks.

All together, the main core takes care of dividing the population into chunks to give to its workers. Each entity works on a particular subpopulation, applying to it the various steps that are needed to build the next generation, including *Evaluation* phase. Once finished, a worker returns the resulting subpopulation to the master.
A parallel pattern that naturally models this reasoning pattern is a master/worker farm. This appears to be the same approach suggested by the `rplsh` support tool.

```
rplsh> crossover = seq()
rplsh> mutation = seq()
rplsh> new_gen = seq()
rplsh> annotate crossover with latency 10
response: annotated!
rplsh> annotate mutation with latency 1
response: annotated!
rplsh> annotate new_gen with latency 20
response: annotated!
rplsh> breed = comp(mutation, crossover)
rplsh> GA = pipe(breed, new_gen)
rplsh> rewrite GA with allrules, allrules
rplsh> optimize GA with farmopt, pipeopt, maxresources
rplsh> show GA by servicetime, resources + 5
2.214286        16       [3] : farm(comp(breed,new_gen)) with [ nw: 14]
2.428571        16       [0] : comp(farm(breed) with [ nw: 11],farm(new_gen) with [ nw: 14])
2.750000        16       [2] : pipe(farm(breed) with [ nw: 4],farm(new_gen) with [ nw: 8])
```

```
2.857143        16       [4] : farm(pipe(breed,new_gen)) with [ nw: 7]
3.333333        16      [16] : farm(farm(pipe(breed,new_gen)) with [ nw: 6]) with [ nw: 1]
```

Speaking of implementations details, with ST threads it was logical to consider an approach based on `std::async` function template, because it is natural to use it when a thread (in this case the main thread) needs for a return value from another thread. However, the standard behavior of this function does not allow you to control the number of threads spawned to complete tasks. To save resources and to allow you to control the degree of parallelism, it was necessary to use a threadpool, which is incompatible with the use of `std::async`. At the same time, the `std::future`-based mechanism was convenient for not introducing explicit synchronization mechanisms such as barriers, which are always a cause for concern in parallel programs.

For all these reasons, I wrote a struct `ThreadpoolTask<T>`, which implements a threadpool, whose threads wait on a blocking queue for tasks to be done. I used a single shared task queue, on which all threads wait for tasks. By completing the various tasks, a result of type `T` is obtained. On the other hand, the main thread computes the various subpopulation chunks and builds `std::packaged_task<T()>` to submit to the threadpool, passing it the function to execute that takes the subpopulation as a parameter. The return value, computed later by a thread of the threadpool, will be stored in a shared state, accessible via a `std::future<T>` object.

Once the various subpopulations have been obtained, the main thread can build the population that constitutes the next generation and thus prepare for the next iteration. Simply, it is enought to merge the $\frac{\mathrm{n}}{nw}$ chunks of population and re-sort them.

The obvious goal of this solution is to attempt to minimize the overall overhead introduced by thread management by reusing existing threads in combination with a task queue locking mechanism. This variant of task queue will perform well, because this application creates few tasks.

Intuitively, this parallel solution gives its best for instances of the TSP problem with many cities (i.e. many nodes), a huge amount of individuals in the population and an high percentage of crossover and mutation. This is because in such circumstances the overhead introduced by thread management would be compensated by the increased work to be parallelized.

FastFlow Solution

The solution developed with the FastFlow library follows the same guidelines as the solution that uses ST threads. It use the parallel design farm, in particular it is a master-worker farm.
Two FastFlow nodes are defined:

1. `Master: ff_node_t<Task>` is a node that emits and collects `Task`, i.e. an helper struct that encapsulated the chunk of population on which the receiver will work on. The master node deals substantially with task dispatching.
2. `Slave: ff_node_t<Task>` is a node that receives tasks from the master, performs an iteration of the algorithm in order to build the new generation and sends back its results.

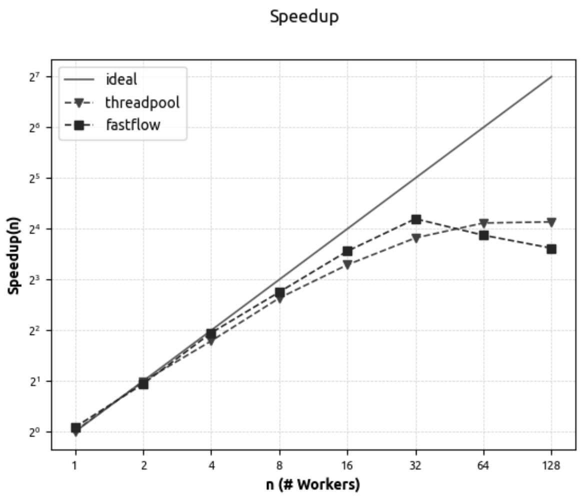
Experimental Results

To conduct the experiments and thus evaluate the performance of the solutions, the execution times were computed on the following experimental values:

- The number of nodes are 2152 and weights of edges are taken from file `u2152.tsp`.
- Number of individuals are 2048.
- Number of generations are 30.
- Seed is 123.
- Percentages of crossover and mutation are respectively 70% and 30%.

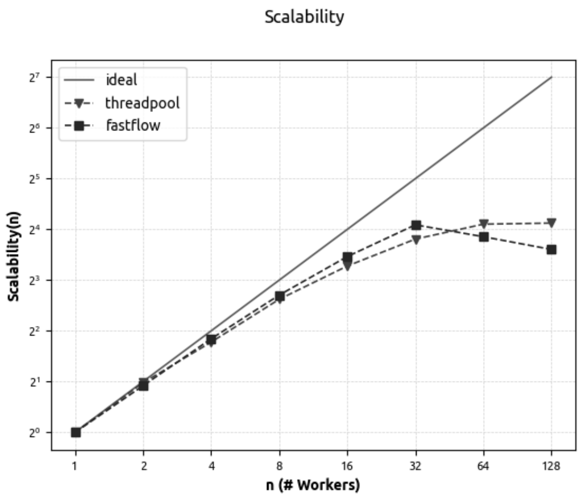
As you can guess, from these parameter values, based on the observations made before, the algorithm is placed in beneficial conditions in terms of parallelism. With these values, we expect a long time from the sequential algorithm, because it will have to deal with long operations on vectors.
Now we just have to compute how long our solutions take to give us the result with parallelism degree 2^i, with $i \in [0 \ldots 7]$.

Speedup results. The situation that arises with the following plot representing the speedup is clear:



As long as we keep to a degree of parallelism $nw = 8$, the speedup scales almost exactly like ideal case. It gets further and further away from the ideal as the number of threads increases. In the case of the implementation with the threadpool, with 128 there is still an improvement, compared to the previous case, even if minimal. With fastflow, on the other hand, the benefits introduced by parallelism begin to diminish even earlier, i.e. with more than 32 workers. If we had increased the degree of parallelism to more than 128 workers, most likely the version with ST threads would also have had the same trend reversal as FastFlow.
Let's try to explain why this happens. Assuming you have 128 workers available, since the master / main thread performs a static division of the population, with this particular instance, each worker will apply an iteration of the algorithm on $\frac{2048}{128} = 16$ individuals. So on the one hand, the job to complete for the single thread does not compensate for the overhead involved in activating that thread, waiting on the task queue, and waiting for the result from the main thread. On the other hand, with this TSP instance, we come to a situation where the master / main thread become a bottleneck: workers are idle waiting for the master to send them a task.
Also, using a FastFlow farm involves introducing an input queue and a feedback queue for each FastFlow node. This could be why FastFlow performance deteriorates earlier: the bottleneck effect for the master gets worse, because there is a lot more overhead for the master in handling communications for 128 channels than handling a single queue of `std::future` results returned by 128 threads.

Scalability results. The results related to scalability also allow us to arrive at the same conclusion.



Conclusions

The root problem lies in tasks division. Inevitably, a static division of the chunks leads to the loss of the advantages of performing essentially independent tasks in parallel, when the degree of parallelism is too high. Dealing with large TSP instances mitigates bottleneck effects, but sooner or later this problem shows up.

We can try to sketch an improvement to the parallel algorithm. Certainly, we must leverage the creation task policy. Workers could benefit from more dynamic handling of chunk splitting. Monitoring the completion time of tasks and increasing the size of the chunks in case of too short service times could be a good thing. This causes imbalances in terms of computation heaviness in the task to be completed. It might be interesting to change FastFlow's scheduling policy (which is round robin by default) to see further improvements.

Usage

It is worth mentioning the introduction of some conditional compilation directives that I've include to my project:

- `-DFILEMATRIX` gives the possibility include a file from which the program builds the graph on which to apply the TSP genetic algorithm.
  If compiled with `-DFILEMATRIX`, executable's usage is:

  ```
  ./seq <seed> <n-population> <n-iteration> <prob-mutation> <prob-crossover> <filename>
  ./par <nw> <seed> <n-population> <n-iteration> <prob-mutation> <prob-crossover> <filename>
  ```

  If the file is compiled without this flag, then it will be necessary to provide the number of nodes and maximum weight on the arcs and the program will generate the graph at random.
  With no `-DFILEMATRIX`, usage become:

  ```
  ./seq <seed> <n-population> <n-iteration> <prob-mutation> <prob-crossover> <n-nodes> <max-weigth>
  ./par <nw> <seed> <n-population> <n-iteration> <prob-mutation> <prob-crossover> <n-nodes> <max-weigth>
  ```

- `-DVERBOSE` makes sure to print the solution found by the algorithm at the end.
- With `-DTIME`, it will be printed in detail the time spent in total in the various phases of the algorithm. Attention: with this flag, there will be lots of prints.
- With `-DTIMETHREAD` the time taken by the threads in the different phases of the TSP genetic algorithm is printed at each iteration.

The last two were useful in sketching the parallel solution.
To compile the files it is enought to write for example the following command:

```
g++ -O3 -o TSP_GA_par TSP_GA_par.cpp -pthread -DFILEMATRIX
```

If you want to pass as an instance one of the files of the `tests_tsplib95` folder and you just want to know the completion time of the program.