Parallel Genetic Traveling Salesman Problem

Francesco Gargiulo
f.gargiulo2@studenti.unipi.it

1 A Genetic Algorithm for the TSP

Given a set of n cities indexed by integers in $[0, n-1]$, we have that $dist(i, j)$ is the distance between any pair of cities (i, j). The TSP algorithm aims at finding the shortest possible route that visits each city exactly once and returns to the initial city.

A population P is a sequence of individuals. An individual $p \in P$ is a vector that specifies an ordering (a permutation) of the n cities: $p[k] = h$ indicates that city h is at the k-th position of the ordering.

The path distance of an individual p is calculated as

$$path\_distance(p) = \sum_{i=0}^{n-1} dist\Big(p[i],\ p[(i+1) \bmod n]\Big)$$

and it indicates the total length of p's path. The fitness score of p is given by

$$fitness(p) = \frac{1}{1 + path\_distance(p)}$$

The fitness is the inverse of the distance because the goal is to maximize the fitness score and thus to minimize the distance of the path. A +1 is added to the denominator because a path distance can possibly be 0.

The fitness scores must then be normalized; this allows to map each fitness score to a probability value such that the total sum of probabilities is 1. Let

$$fitness\_sum = \sum_{p \in P} fitness(p)$$

then

$$normalized\_fitness(p) = \frac{fitness(p)}{fitness\_sum}$$

1.1 Weighted Random Selection

Our goal is to pick individuals at random from the population so that individuals with a higher fitness are more likely to be chosen (and thus they are more likely to pass their genetic information to the next generation).

Each individual $p \in P$ has a probability given by $normalized\_fitness(p) \in [0,1]$ such that

$$\sum_{p \in P} normalized\_fitness(p) = 1$$

The weighted random selection works as follows: a random floating-point number $r \in [0,1]$ is generated. Consider P as a sequence $p_0, p_1, \dots, p_{m-1}$. The individual we pick is the first $p_i \in P$ such that the prefix sum of probabilities is at least r. In other words, p_i is such that

$$\sum_{k=0}^{i} normalized\_fitness(p_k) \geq r \quad \text{and} \quad \sum_{k=0}^{i-1} normalized\_fitness(p_k) < r$$

1.2 Cycle Crossover

Given two individuals p_1, p_2, a cycle crossover generates two children c_1, c_2. First of all, we need to identify cycles in p_1, p_2 and assign them a progressive number. We identify a single cycle as follows:

1. start in p_1 from the first index that has no cycle number assigned, say $p_1[initial]$. Let $curr = initial$.
2. look at $p_2[curr]$.
3. find index $next$ such that $p_1[next] = p_2[curr]$.
4. add index $next$ to the cycle and set $curr = next$.
5. repeat steps 2 through 4 until you arrive back at $p_1[initial]$.

The complete cycle identification is illustrated in Figure 1.

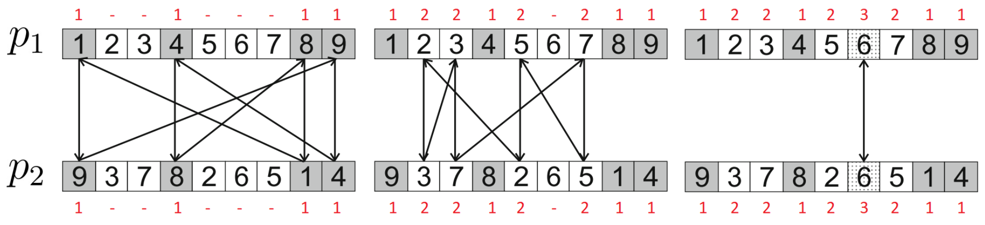

Figure 1: Three cycles are identified and numbered progressively.

Then, c_1 is created by taking indexes with an odd cycle number from p_1 and indexes with an even cycle number from p_2. c_2 is generated in the opposite way. The generation of c_1, c_2 is illustrated in Figure 2.

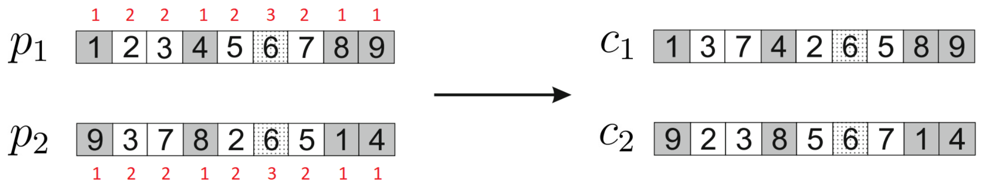

Figure 2: Generating c_1, c_2 from p_1, p_2 after cycles are identified and numbered.

1.3 Mutation

After the crossover operator generates the offspring of a new generation, the mutation operator is applied with a certain probability to each individual. The mutation operator selects a subsequence of an individual and reverses it.

2 Sequential Solution

The sequential solution is composed by the following steps:

1. Create an initial random population $P = p_0, p_1, \ldots, p_{m-1}$.
2. For each individual $p \in P$ compute its fitness score and normalize it.
3. For $\frac{m}{2}$ times, pick two parents at random from the current population via weighted selection and create two children using the crossover operator.
4. Apply mutation.
5. Replace old population with new one.
6. For each individual $p \in P$ compute its fitness score and normalize it.
7. Terminate if the number of generations meets some upper bound; otherwise go to step 3.

Figure 3 shows that this genetic algorithm is able to improve the population statistics generation after generation.

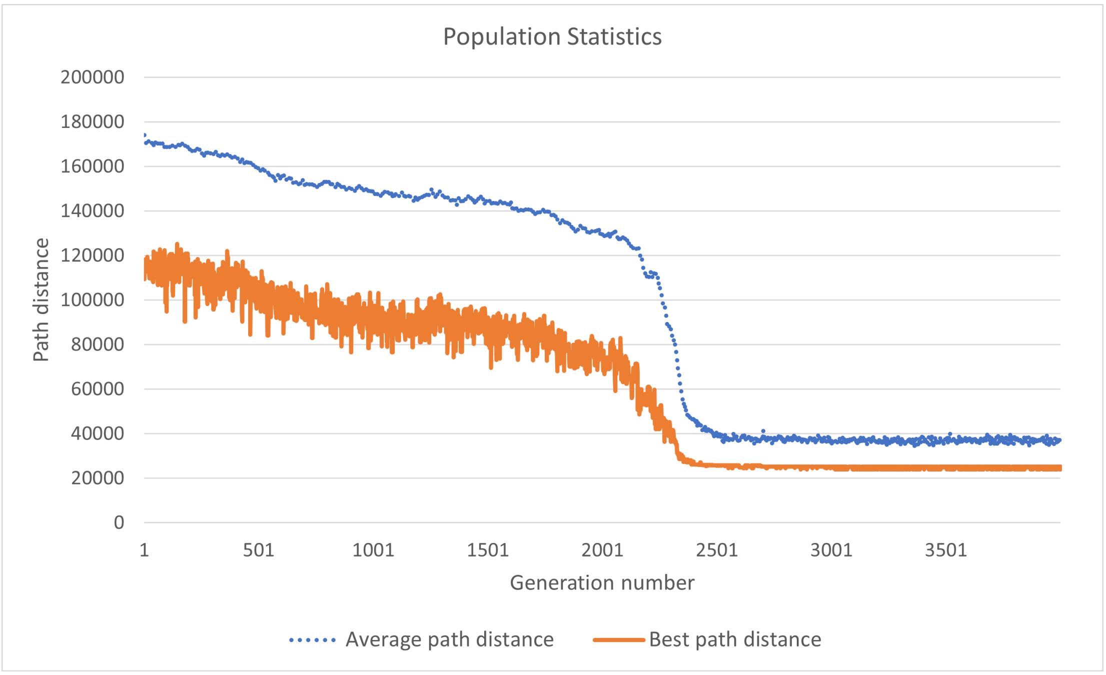

Figure 3: Executing the proposed algorithm with parameters $n = 35$, $m = 2000$. We can notice how both average and best path distance improve over time until a local minimum is reached.

Let T_i indicate the latency of step i. In order to compute each T_i on the designated machine, we need to fix the number of cities n and the population size m. Table 1 indicates latencies for $n = 100$ and $m = 20000$.

T_1	T_2	T_3	T_4	T_5	T_6
47737 μs	10835 μs	239239 μs	568 μs	0 μs	10840 μs

Table 1: Estimating each latency T_i by taking averages over 1000 executions of *sequential.cpp* with parameters $n = 100$, $m = 20000$, $seed = 123$, $n_{gen} = 1$.

Step 5 takes 0 μs because at the start of the program we allocate memory for two populations: current population and next population. Then, the replacing can be done in constant time by swapping pointers to these areas of memory.

The computational cost is dominated by step 3, i.e. the generation of offspring via crossover. In particular, the loop from step 3 to step 6 will be executed for a certain number of generations, which can be in the order of thousands. If we indicate with n_{gen} the number of generations, then

$$T_{SEQ} = T_1 + T_2 + n_{gen}(T_3 + T_4 + T_6)$$

3 Parallel Solution

3.1 Farm with Feedback

In order to significantly reduce the time needed to execute the program, we need to parallelize the generation of each next population. Consider the architecture illustrated in Figure 4.

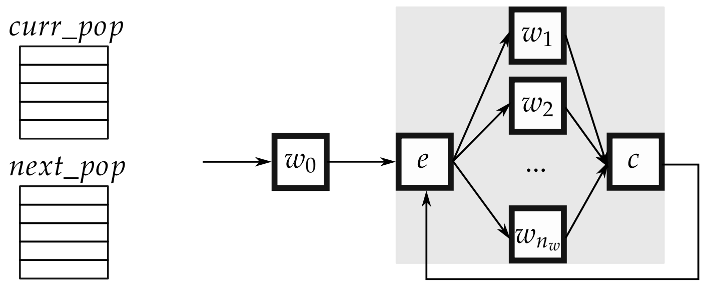

Figure 4: A farm with feedback that can be used to parallelize the generation of successive generations.

The workflow is the following:

- w_0 creates an initial population, computes the fitness scores and their sum.
- e assigns to each farm worker an interval of $next\_pop$ of size $\frac{m}{n_w}$ to store children generated using parents in $curr\_pop$.
- each farm worker generates the required children, applies the mutation operator, computes their fitness scores, and computes some local statistics (sum of fitness scores, average/best path distance).
- c reduces the local statistics, obtaining $fitness\_sum$. Then, it replaces the old population with the new one. Finally, it terminates the program if the number of generations equals n_{gen}, otherwise it starts a new generation.

Using a *read-only* access to the current population and an *owner-writes* access to the next population, there is no need for synchronization between parallel workers.

Let T_{split}, T_{merge} be the latencies (for a single element) of e, c, respectively. Let the latency of a single parallel worker be defined by

$$T_w = \frac{T_3 + T_4 + T_6}{\hat{n}_w}$$

where $\hat{n}_w = \min\{\frac{m}{2}, n_w\}$ is the maximum number of parallel workers given by the fact that the crossover operator generates two children and it must be executed sequentially. Then, the completion time is

$$T_{PAR}(n_w) = T_1 + T_2 + n_{gen}\left(T_{split} + T_w + T_{merge} + (\hat{n}_w - 1)\max\{T_{split}, T_{merge}\}\right)$$

T_{split} consists in assigning consecutive intervals, while T_{merge} consists in a few constant time operations. If we assume $T_{split} \approx T_{merge} \approx 0\ \mu$s, then we can approximate the speedup as follows

$$\begin{aligned} speedup(n_w) &= \frac{T_{SEQ}}{T_{PAR(n_w)}} \\ &= \frac{T_1 + T_2 + n_{gen}(T_3 + T_4 + T_6)}{T_1 + T_2 + n_{gen}\left(T_{split} + T_w + T_{merge} + (\hat{n}_w - 1)\max\{T_{split}, T_{merge}\}\right)} \\ &\approx \frac{T_1 + T_2 + n_{gen}(T_3 + T_4 + T_6)}{T_1 + T_2 + n_{gen}\left(\frac{T_3+T_4+T_6}{\hat{n}_w}\right)} \end{aligned}$$

which becomes closer to the ideal speedup as n_{gen} grows (as long as $n_w \leq \frac{m}{2}$).

3.2 Master-Worker

The farm with feedback architecture proposed in the previous section provides an almost-ideal speedup. Given that $T_{split} \approx T_{merge} \approx 0$, a master-worker architecture allows us to achieve the same performances as using less parallel agents. Consider the Master-Worker architecture illustrated in Figure 5.

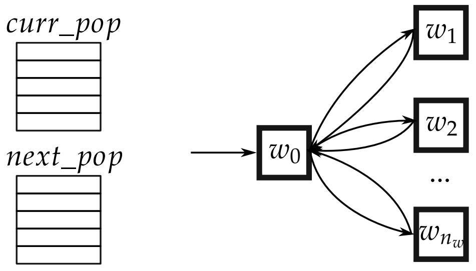

Figure 5: A master-worker architecture that can be used to parallelize the generation of successive generations.

The master w_0 substitutes the work done by w_0, e, c in the farm with feedback architecture. For this reason, the implementation uses a master-worker architecture.

4 Reducing Overheads

This section presents the solutions adopted to address the overhead sources identified during the development of the parallel solution.

4.1 Memory Allocation

Memory has been allocated once and for all during the initial stage of the computation. However, if a worker is the only one using a data structure (e.g. vectors used during crossover), then memory allocation is postponed to the start of the thread so that data will be closer to the thread itself. Functions on such data structures take as input references in order to avoid copying data.

4.2 Random Numbers Generation

The program takes a seed as input for reproducibility purposes. Random numbers are used both for graph initialization and for the genetic algorithm operators. The C++ *rand()* function is not thread-safe since it uses an internal state that is modified on each call. A multithreaded program should use a synchronization mechanism to use *rand()* on the same state. However, this would introduce overhead and, because of thread scheduling, numbers generated by a single thread can change between different executions. For this reason the *rand_r()* function is used: it takes as input the address of a seed value that is updated on each successive call. Each thread uses the input seed to generates its own local seed, which is modified only by the thread itself after each *rand_r()* it executes.

5 Results

Tests were performed using the following parameters:

$$n = 100,\ m = 20000,\ n_{gen} = 1000,\ seed = 123$$

Figure 6 shows the completion times of the sequential and parallel solutions. Figure 7 shows the speedup of the parallel solution. During the initial analysis we expected a roughly linear speedup. Initially ($1 \leq n_w \leq 15$), this estimate looks correct. However we can see how, as the parallelism degree grows, the overheads affect the speedup. These overheads can probably be explained by the fact that threads are executed on different NUMA nodes and thus the memory access cost grows with the parallelism degree.

After $n_w > 29$, the speedup slows done significantly. The machine has 32 cores, each supporting 2-way hyper-threading. When $n_w = 30$ there are 30 worker nodes each on a distinct core, there is the master node on another core, and the machine will occupy another core. Starting more threads means that some cores will have 2 threads running.

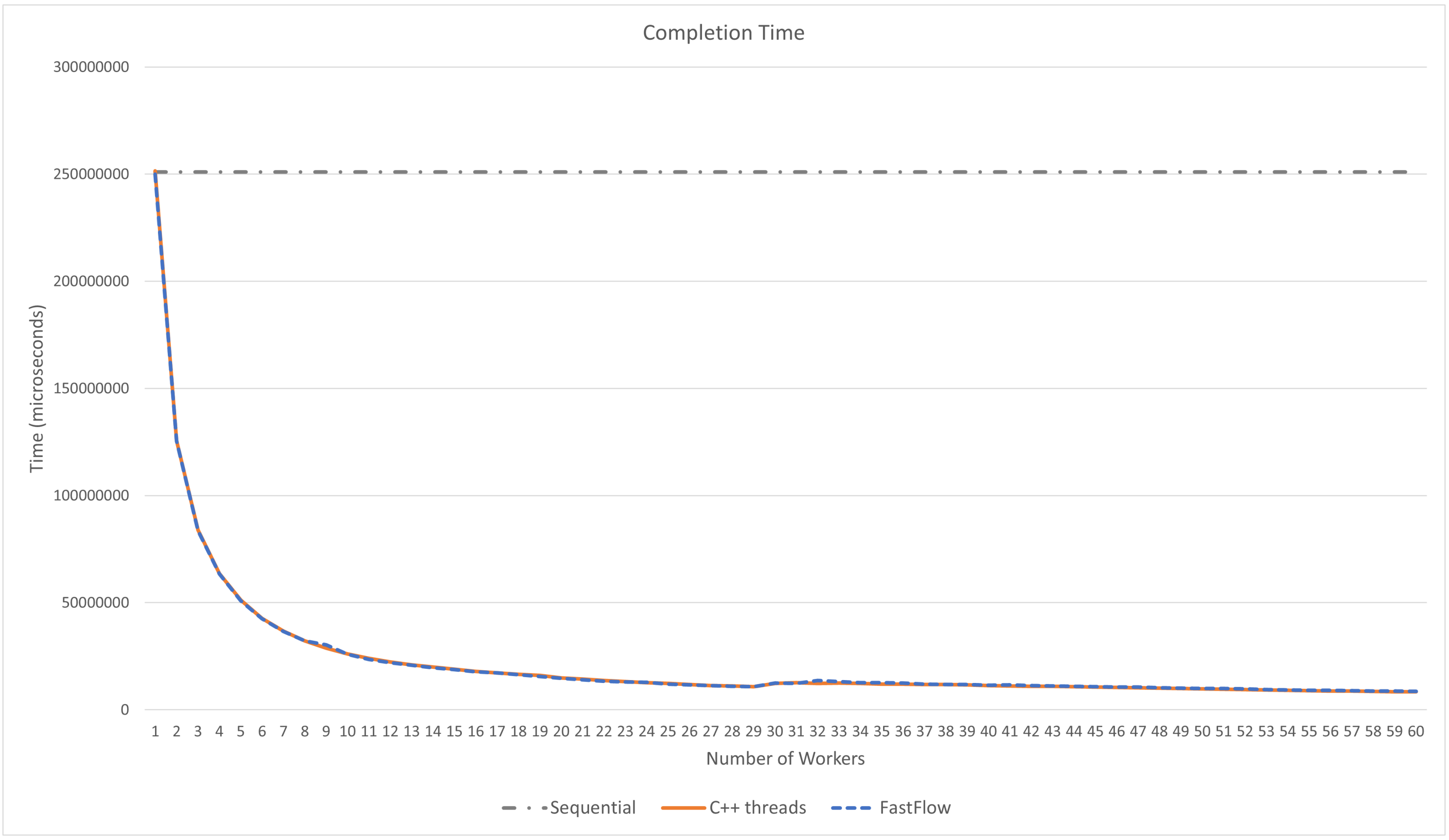

Figure 6: Completion time of the sequential and parallel implementations.

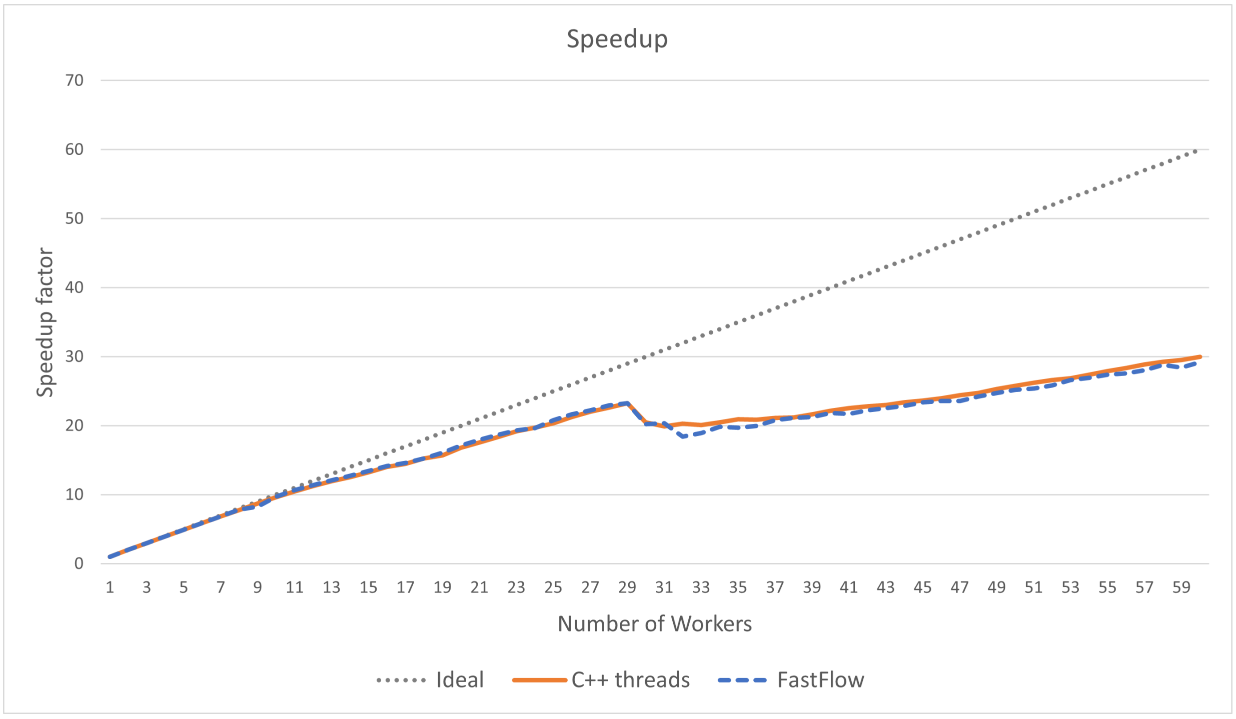

Figure 7: Speedup of the parallel solution.

6 How to Compile and Run the Solutions

6.1 Sequential Solution

It is possible to compile the sequential solution using

```
g++ -O3 -o sequential sequential.cpp
```

and it can be executed using

```
./sequential cities population generations seed
```

An example of execution is

```
./sequential 100 20000 1000 123
```

6.2 Parallel Solution

It is possible to compile the parallel solution using

```
g++ -pthread -O3 -o parallel parallel.cpp
```

and it can be executed using

```
./parallel cities population generations seed n_w
```

An example of execution is

```
./parallel 100 20000 1000 123 32
```