Genetic TSP:
Parallel and Distributed Systems: Paradigms and Models Project

University of Pisa - Computer Science
Jacopo Raffi

Academic Year 2022/2023

Contents

1 Introduction

This report contains the design and implementation choices for the Genetic TSP project. The timing of the sequential algorithm will first be shown, and then the chosen parallel design will be described, giving reasons for the choices. Finally, speedup, scalability and efficiency results will be illustrated.

2 Sequential Algorithm

Before designing the parallel algorithm structure, it was necessary to develop the sequential algorithm(the skeleton is shown in figure 1) in order to assess which parts of the program were convenient to parallelize. The skeleton of the sequential algorithm is as follows:

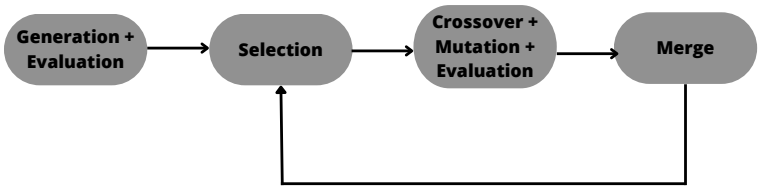

Figure 1: Sequential Algorithm Skeleton

The computation times have been estimated using a graph consisting of 5,000 nodes, with a mutation rate of 30%, a crossover rate of 70% and a population first of 4000 and then of 8000 individuals:

Population size	4000(2800 children)	8000(5600 children)
Generation + Evaluation	410ms	997ms
Selection	8-9ms	17ms
Crossover + Mutation + Evaluation	200-250ms	400-500ms
Merge	300ms	600-650ms

After evaluating these “macro times", the times of the individual functions were also considered, again with a graph of 5000 nodes:

Generation	$40\mu s$
Selection	$8 - 9\mu s$
Crossover	$20 - 50\mu s$
Mutation	$0\mu s$
Fitness	$50\mu s$

After analysing the times of the sequential algorithm, the next step was to assess a potential parallel pattern, taking into account that the most computationally intensive aspects of the program are the evaluation of individuals and the crossover, followed by the selection phase.

3 Parallel Design

First of all, it was considered appropriate to parallelize the $generation + evaluation$ phase. In this case, a map was adopted which returns the first generation of chromosomes. In addition, the load balance of the workers was taken into account; in fact, the number of individuals to be generated was evenly distributed, before the workers were put to work, in order to avoid possible overloading. For example, if it is necessary to generate 11 individuals with three workers, a balanced situation could be one in which 2 workers have 4 chromosomes and the last one has 3 chromosomes, whereas an unbalanced situation would be one in which 2 workers have 3 chromosomes and the last one has 5 chromosomes;

An alternative could have been to use a pipeline with two stages (generation and fitness), which have a similar cost; however, this alternative was rejected for three reasons:

- overhead given by communication/synchronisation;
- the fitness is higher in computation time and it would be a bottleneck for the pipeline;
- would not be a good choice to reduce the completion time.

The reason for the third point can be explained by taking two workers as examples (it can be easily generalised to w workers). In the case of a two-stage (both with cost t) pipeline we would have a completion time of $2 * t + (n - 1) * t$, where n is the size of the population, in the case of a map we would have a completion time of $(n/2) * 2t = n * t$; the pipeline would certainly have a higher throughput, but in this situation it is the completion time that counts. In the end the map pattern was chosen.

The next step was to consider how to parallelize the selection, crossover and evaluation phases. Again, a map was adopted precisely because the objective is to reduce the completion time, as an iteration cannot begin without the previous one being completely finished. In addition, to avoid having two different maps separated by the mutation phase, it was decided to merge these two maps into a single map(figure 2), including the mutation phase, in order to avoid communication/synchronisation overheads, also considering that making the mutation parallel does not introduce any further overheads. Again, the pairs of parents to be chosen and processed were distributed so as to ensure a load balance between the workers.

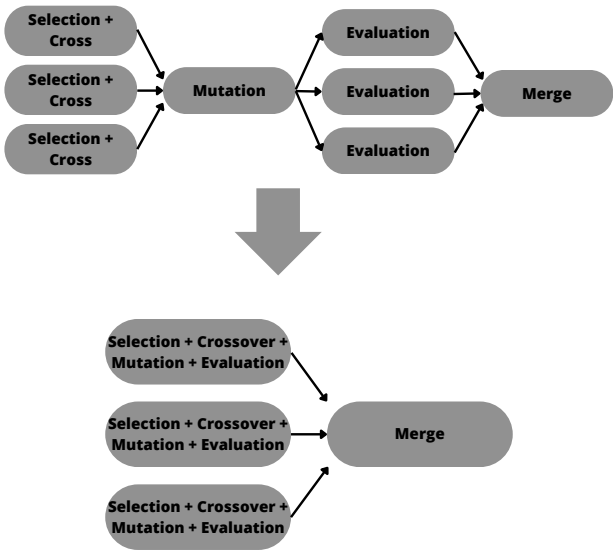

Figure 2: Map Fusion.

Other alternatives considered, but rejected, are the pipeline and the farm. The pipeline approach was rejected because, in general, throughput is not the primary concern. As previously mentioned, the crucial

factor is the completion time of each iteration. For this reason, the pipeline approach was not employed, as it could potentially introduce additional overhead due to synchronisation and communication between the pipeline stages.

The farm approach, involving only emitters and workers, was rejected to avoid excessive overhead in communication between emitters and workers. The decision was made considering that it is possible to divide the indices only once without the need for communication at each iteration.

To avoid excessive communication and synchronisation overhead, different combinations of patterns were avoided, and a simple map pattern was considered sufficient for the purpose. The decision was made to prioritize simplicity and efficiency by minimizing the complexity of the parallel execution model. This approach ensures that the computation is efficiently distributed among the threads while minimizing the need for extensive coordination and communication between them; in addition using stream parallel patterns within a data parallel pattern is an ineffective decision and was not taken into consideration.

Finally the crossover phase consists of two subphases, namely swap and fix, with the weight primarily attributed to the latter. An idea that could have been considered was to make the swap phase sequential to achieve load balancing among the workers for the fix phase. This is because the time taken for the fix phase increases with the number of fixes to be performed. It was deemed unnecessary to perform load balancing for the fix phase after selection and swap for two reasons:

- the selection process is randomized, it is unlikely that a worker would have a significantly higher number of chromosome pairs requiring a fix after the swap;
- observing the sequential algorithm, the tendency is that initially all offspring require a large number of fixes, but as the iterations progress, the number of fixes tends to decrease or even disappear.

3.1 Native Threads C++

The initial implementation was developed using the native threads and the standard synchronisation mechanisms provided by C++.

The generation+evaluation phase was realised by means of a simple map introducing only the fork and join of worker threads as overhead.

In the implemented solution, the thread main is responsible for executing the merge phase. The thread main initially, before forking the workers, distributes the number of parents that the workers will have to choose and process (trying to maintain load balance); the workers also know the number of iterations of the program, so there is no need for the thread main to communicate the “End of Work". The only synchronisation needed is to understand when the workers can start the new iteration and when they must wait for the merge phase; a barrier and a condition variable were used to do this. The workers wait for the main thread to complete its work via a “wait()", the main thread then wakes them up via a “notify_all()" call. The main thread on the other hand waits for the workers using a barrier, so that when the barrier “opens," the main thread knows that all the workers have completed their iteration.

3.2 FastFlow

The second implementation of the project was developed using the FastFlow library, a programming framework specifically designed for parallel programming in C++. The parallel design remains the same in the second implementation, and it is realized using an “ff_Farm" object with only the emitter and the workers.

The emitter component in the second implementation is realized using an “ff_monode" object, specifically designed for the merge phase. On the other hand, the worker nodes are implemented as “ff_node" objects, which handle the selection, crossover, mutation and evaluation phases. The emitter component communicates the completion of its work by broadcasting a task signal, specifically using the “broadcast_task(GO_ON)" method; the number of pairs to be selected and processed is given to the workers during their creation. On the other hand, individual worker nodes signal the completion of their tasks by executing the “ff_sendout(GO_ON)" operation. Furthermore, the emitter component continues its task only after receiving a number of signals equal to the number of worker nodes. This ensures that all worker nodes have completed their respective tasks.

4 Results

The final phase of the project involved performance analysis of the parallel programs, focusing on speedup and scalability. Three different input dimensions were considered:

- 500 nodes and a population size of 3000;
- 1000 nodes and a population size of 6000;
- 2000 nodes and a population size of 12000.

For each input dimension, the parallel programs were executed multiple times to obtain averaged results; the programs ran with 2, 4, 8, 16, 32 and 64 threads. In all cases the crossover-rate is 70%, the mutation rate 30% and 1000 iterations are performed.

In general, there is a significant difference between the ideal and actual execution times in both parallel versions of the program. In the native thread version of the program, the observed overhead primarily stems from the use of condition variable and barrier. These synchronization mechanisms are necessary to coordinate the execution of threads and ensure proper sequencing of operations. However, they introduce additional computational costs, including thread blocking and waking up, which impact the overall performance of the program. In FastFlow, one potential source of overhead is the increased communication between the worker threads and the emitter. Another issue may be the presence of other FastFlow programs. In fact, when attempting to run two instances of the FastFlow implementation concurrently, the execution times showed a significant increase compared to running a single instance of the program. On the other hand, running an instance of the program implemented with FastFlow simultaneously with an instance of the program implemented with native threads does not lead to a significant increase in the execution times.

In general, both in the case of native threads and FastFlow, the difference between the achieved speedup and the ideal speedup tends to decrease as the input size increases (same for the scalability and efficiency).

Speedup Plots

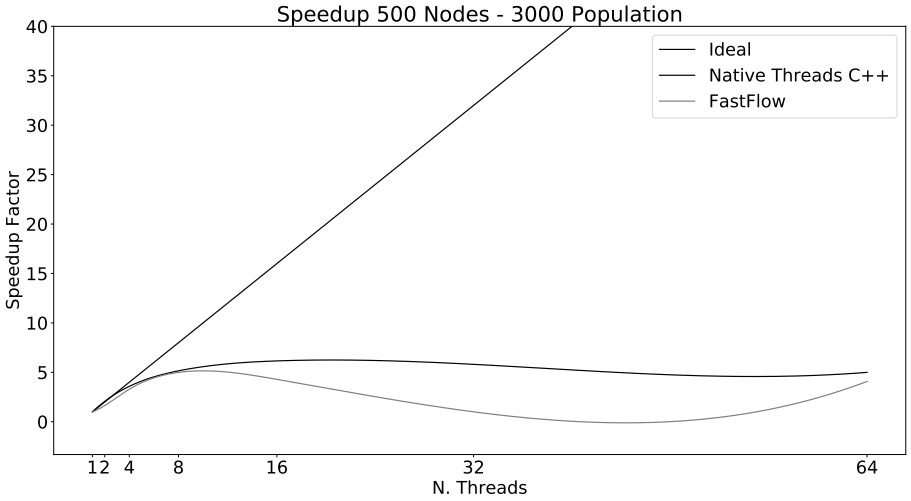

Figure 3: Speedup small input size.

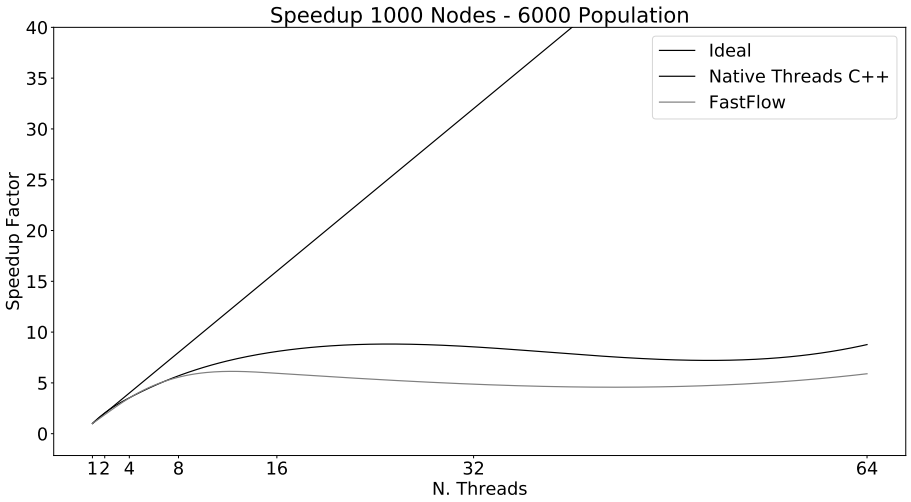

Figure 4: Speedup medium input size.

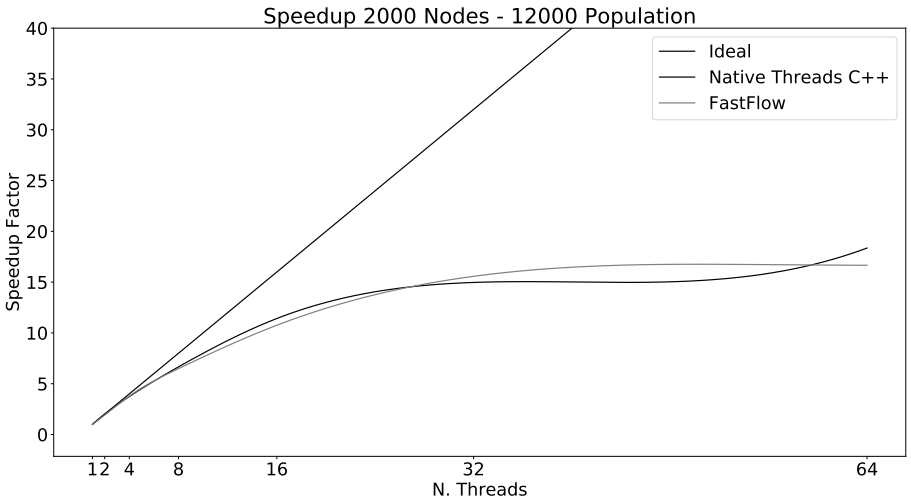

Figure 5: Speedup large input size.

Scalability Plots

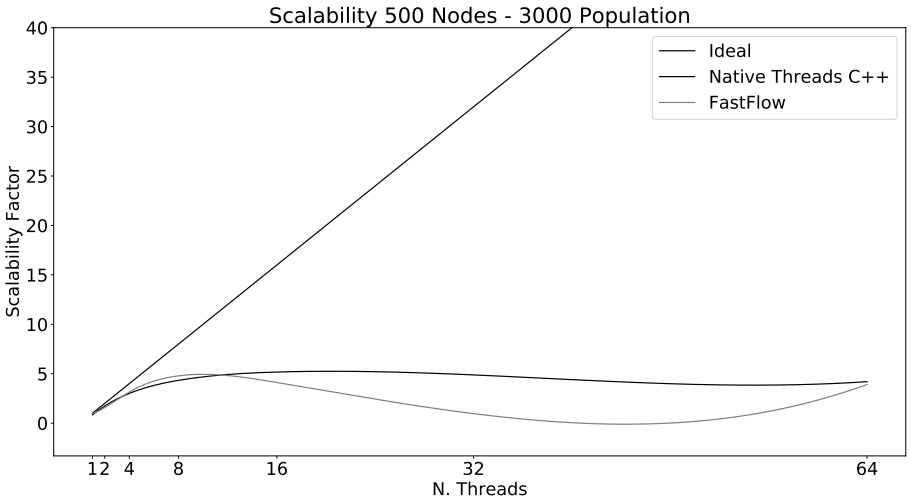

Figure 6: Scalability small input size.

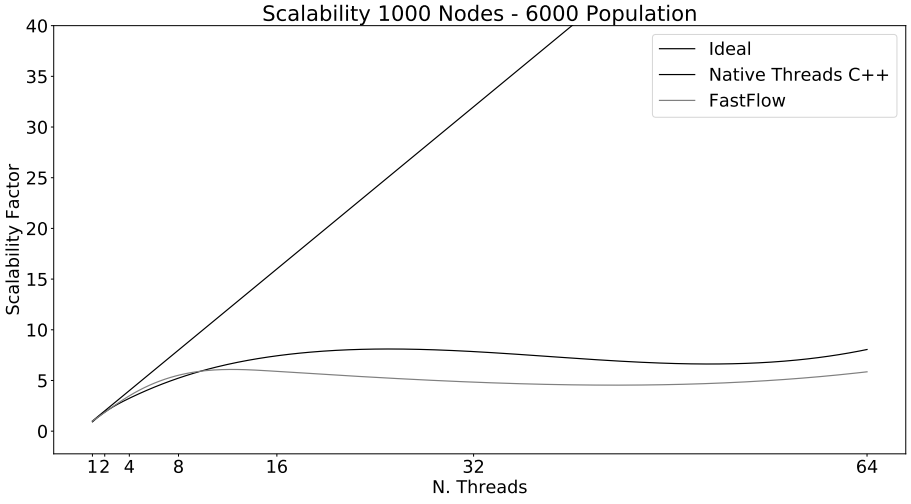

Figure 7: Scalability medium input size.

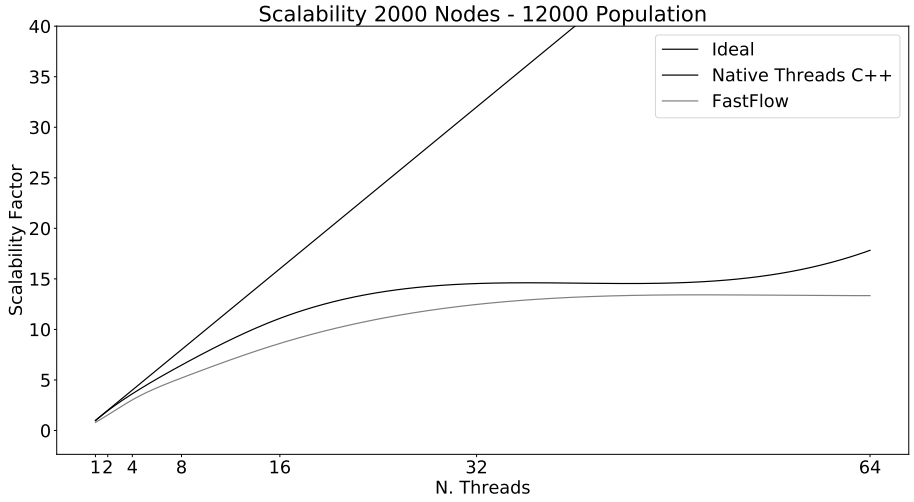

Figure 8: Scalability large input size.

Efficiency Plots

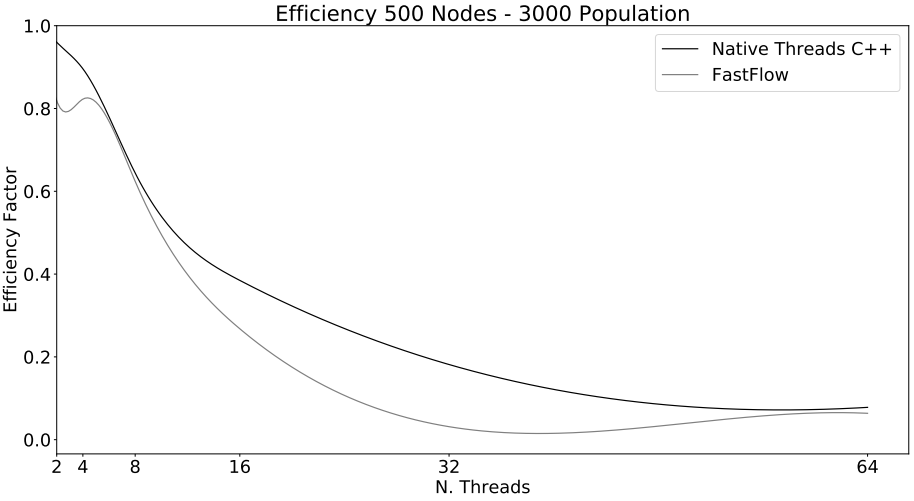

Figure 9: Efficiency small input size.

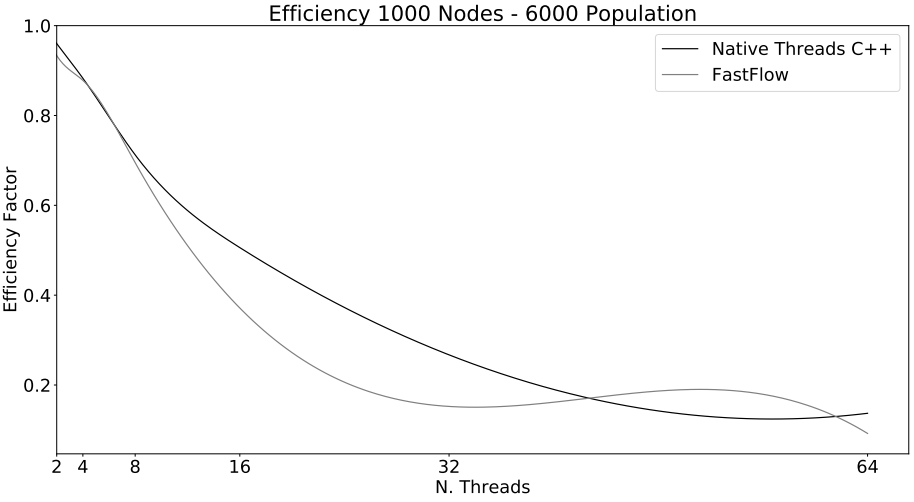

Figure 10: Efficiency medium input size.

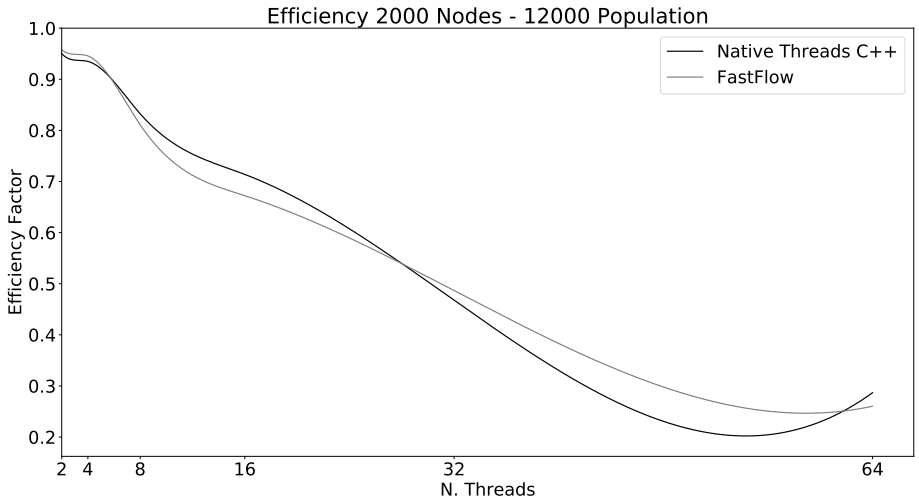

Figure 11: Efficiency large input size.

5 How to Compile

To compile and run the various versions of the programme go to the “src" folder. The commands to compile are:

- **sequential**: g++ -O3 -std=c++20 tsp_sq.cpp -o sq;
- **native threads**: g++ -O3 -pthread -std=c++20 tsp_par.cpp -o par;
- **FastFlow**: g++ -O3 -pthread -std=c++20 -I fastflow/path tsp_ff.cpp -o ff.

The execute the programs the commands are (with examples):

- **sequential**: ./sq file population mut_rate cross_rate generations:
  - ./sq ../cities_500.txt 3000 0.3 0.7 1000;
  - ./sq ../cities_1000.txt 6000 0.3 0.7 1000;
  - ./sq ../cities_2000.txt 12000 0.3 0.7 1000.
- **native threads**: ./par file population mut_rate cross_rate generations workers:
  - ./par ../cities_500.txt 3000 0.3 0.7 1000 2;
  - ./par ../cities_1000.txt 6000 0.3 0.7 1000 4;
  - ./par ../cities_2000.txt 12000 0.3 0.7 1000 16.
- **FastFlow**: ./ff file population mut_rate cross_rate generations workers:
  - ./ff ../cities_500.txt 3000 0.3 0.7 1000 2;
  - ./ff ../cities_1000.txt 6000 0.3 0.7 1000 4;
  - ./ff ../cities_2000.txt 12000 0.3 0.7 1000 16.

The use of the “jemalloc" library does not lead to any increase in performance except for the first phase (generation + evaluation); this is somewhat expected as only the first phase has a large number of “alloc" calls to the heap (when creating “vector<int>" for each member of the population).

6 Conclusions

In conclusion, it has been examined the sequential algorithm phases, followed by the selection of an appropriate parallel design(map pattern), with completion time of a single iteration being the crucial factor. Subsequently, implementations using C++ threads and FastFlow were presented. Finally, speedup, scalability and efficiency results are shown.