SPM Project: Parallel Implementations of Genetic Algorithm Solutions for the TSP

Andrea Roncoli

April 2024

Abstract

This project investigates the parallelization of genetic algorithms (GAs) to efficiently solve the Traveling Salesman Problem (TSP). Solutions using native threads and the FastFlow library are explored to parallelize the workload. An initial theoretical analysis is conducted, followed by an assessment of experimental timings. The divergence between ideal and practical computation times is discussed, along with factors influencing this disparity.

1 Introduction

1.1 Travelling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic NP-hard optimization problem in which the objective is to determine the shortest possible tour that visits each city exactly once and returns to the starting city. Formally, let $G = (V, E)$ represent a complete graph, where V is the set of cities and E is the set of edges connecting them. Each edge e_{ij} has a non-negative weight w_{ij}, representing the distance between cities i and j. The TSP seeks to find a permutation π of the cities such that the total distance traveled, known as the tour length, is minimized, subject to the constraint that each city is visited exactly once and the tour forms a closed loop.

1.2 Genetic Algorithms

Genetic algorithms are a class of optimization algorithms inspired by the process of natural selection and evolution. They are commonly used to solve complex optimization problems which have no efficient complete solution, including the TSP. The evolution process involves several key operators: initialization, selection, crossover, mutation, and merge. In the **initialization** phase, an initial population of candidate solutions is generated randomly or using a heuristic method. **Selection** involves choosing individuals from the current population to serve as parents for the next generation. During **crossover**, genetic material

from two parent chromosomes is combined to produce offspring. **Mutation** introduces random changes to individual chromosomes, promoting genetic diversity. Finally, in the **merge** phase, offspring are combined with the parent population to form the next generation.

Whilst initialization takes place only once at the start of the algorithm, all other phases are repeated iteratively to evolve the population towards better solutions.

1.3 Data

The data used comes from the World TSP[1], a dataset that puts together data from the National Imagery and Mapping Agency database of geographic feature names and data from the Geographic Names Information System (GNIS), to generate instances of the notorious TSP problem. In particular, the national instance from Canada is used.

2 Analysis of Sequential Algorithm

The phases of the algorithm are reported in Algorithm 1.

Algorithm 1 Genetic Algorithm for TSP

1: **Initialization**: Initialize a population of n_{pop} random routes
2: **Evolution** (repeat n_{gen} times)
3: **Selection**: Select $n_{parents}$ individuals as parents for the offspring of evolution through roulette wheel selection
4: **Crossover**: Perform crossover with two cut points and perform ordered restoration strategy to legalize offspring routes
5: **Mutation**: Swap two random cities in the route in each offspring
6: **Merge**: Sort by fitness the population and the offspring and select the first n_{pop} individuals as the new population

2.1 Sequential Times

Initially, we run the sequential algorithm two times, with different population sizes, to understand mainly two things: the entity of the computation in each phase, and if the algorithm is scaling correctly with population size. We don't run the algorithm multiple times for each population size as the times are already averaged across generations, and initialization, which is the only one performed just once, as we will see, is of little importance to our parallelization efforts. Initialization and each of the evolution steps are performed once on each individual (per generation, in the case of the evolution steps), so we would expect

[1] World TSP (University of Waterloo)

Phase	$n_{pop} = 2048$	$n_{pop} = 8192$
Initialization	3.43×10^5	1.31×10^6
Selection	$(4.78 \pm 2.39) \times 10^3$	$(3.39 \pm 0.87) \times 10^4$
Crossover	$(8.45 \pm 0.08) \times 10^6$	$(3.36 \pm 0.02) \times 10^7$
Mutation	$(1.05 \pm 1.33) \times 10^2$	$(2.57 \pm 7.24) \times 10^3$
Evaluation	$(4.99 \pm 1.14) \times 10^4$	$(2.10 \pm 0.71) \times 10^5$
Merge	$(1.48 \pm 0.77) \times 10^4$	$(6.24 \pm 5.47) \times 10^4$
Total Time	8.55×10^7	3.40×10^8

Table 1: Execution times for different phases of the algorithm, averaged over 10 generations.

the phase times (and thus the total time) to increase proportionally with population size. This may not be exactly the case in practice, mainly for the inherent stochasticity of evolution dynamics, and also for practical computation reasons, such as varying memory allocation times.

As we can observe in Table 1, the highest load of computation, which dominates the computation time, is given by the crossover phase, which is expected since a large amount of copying one route to the other and then legalization time is spent. Furthermore, we are happy to see that the phase and total times scale as expected.

3 Theoretical Analysis

We will proceed with a theoretical analysis of how the parallelization of the computation should take place. We take as reference the sequential algorithm and its times, to both understand where to concentrate our efforts, and what results we should expect.

3.1 Parallelization Setting and Schema

The case of the evolution of individuals of genetic algorithms is a typical data parallel setting. In such a setting, the computational workload is distributed among multiple processing units, each tasked with independently processing different portions of the data. To assess where we may compute in such a parallel way, we must first understand the degree of independence of the various phases.

Initialization. The initial phase of the algorithm can be parallelized, as generating random routes can be done independently by a group of workers who ultimately merge their work to yield the population. However, it is worth noticing that initialization happens once, whilst, in a typical genetic setting, evolution may iterate even thousands of times. For this reason, this is already an amortized cost and we choose to not concentrate our efforts in the parallelization of this phase.

Selection. This phase can be a totally parallelizable process, given the assumption that an individual may be picked more than once as a parent, which is typically the case in genetic algorithms. In this case, since all workers are in read only, they can simply pick from the shared population resource, which needs no protection, choosing a chunk of the total parents each.

Crossover, Mutation and Evaluation. Once parents chunks have been selected, these three phases are totally independent from any other shared resource. [2] The only attention to be made is when, at the end of the mutation and before the merge, all chunks are gathered from the workers.

Merge. The last phase of evolution requires three steps, a concatenation of offspring and population, a sort of the combined vector and a truncation of the resulting sorted vector at the n_{pop} position. Since there is little to be done for concatenation and truncation of the vectors, and sorting with more workers may result in a complex task of synchronization with little gain, we decide not to parallelize this phase.

We can thus imagine that the resulting parallelization schema for the evolution would be, for each iteration, to fork the work at the start of the process and then have a synchronization effort to merge the offspring chunks and join the workers, finishing the evolution with the merge phase performed sequentially.

3.1.1 Serial and Non-Serial Time

The total time of our sequential algorithm is given by:

$$T_{seq} = T_i + n_{gen} \times (T_s + T_c + T_{mu} + T_{me})$$

Defining the serial fraction as the non parallelizable part of the computation and the non serial fraction as the parallelizable one, we can divide our sequential time in two chunks:

$$\begin{cases} T^s_{seq} = T_i + n_{gen} \times T_{me} \\ T^{ns}_{seq} = n_{gen} \times (T_s + T_c + T_{mu}) \end{cases}$$

In our case, this yields a **serial fraction**, calculated as $f = T^s_{seq}/T_{seq}$, of 0.058% and 0.057% for our runs of the sequential algorithm with $n_{pop} = 2048$ and $n_{pop} = 8192$. This result confirms our expectations that there is no reason why the serial fraction should change, as everything scales the same way with the population size.

3.1.2 Parallelization Strategies

For the scheduling of the tasks to the workers, we analyze two of the simplest, yet most used and well known, strategies: static and dynamic scheduling.

[2] Thinking about these three processes in terms of data parallel skeletons, we could see the crossover as a stencil operation, where the neighbourhood of the individual is the next individual, which is the other parent, and the mutation and evaluation as simple maps. In fact, for crossover, particular attention is to be given to the borders of the chunks: in the case the parents are odd, the last is matched with the first one.

Static Scheduling. In static scheduling, we take the data (in our case, the population) and divide it into as many chunks as the number of workers n_w, each of size n_{pop}/n_w. This solution offers an extremely simple scheduling, with very small overhead, but may yield more easily unbalanced loads to the workers. This is of particular interest in our case, as the inherent stochasticity of some of the genetic phases may yield strong imbalances among chunks.

$$T_{stat}(n_w) = T^s_{seq} + \max_{\{i=1,\dots,n_{chunks}\}} T_{chunk_i} + T_{synch}(n_w)$$

Dynamic Scheduling. In dynamic scheduling, a more flexible approach is adopted, by dynamically assigning tasks to workers based on availability. This approach can adapt to varying workloads and mitigate potential load imbalances, however it incurs in higher overhead due to the need for task management and coordination among workers. In our case, the tasks are simply defined as evolving small groups of individual of a given chunk size cs. An interesting trade-off happens in terms of this parameter: lower values gain the advantage of good balancing , but may suffer strong synchronization efforts, as workers are picking up tasks very often and may incur into wait times; higher values, instead, have lower synchronization struggles, but may incur again into unbalance, just as in the static setting. However, this happens only in case the individuals are small enough, to be comparable with the synchronization times. In our setting, tasks are many orders of magnitude higher than synchronization times ($10^6 \mu s$ vs $10 \sim 100 \mu s$), so we expect taking cs as small as possible to be convenient, as we get good balancing without suffering much from synchronization.

$$T_{dyn}(n_w) = T^s_{seq} + \frac{T^{ns}_{seq}}{n_w} + T_{synch}(n_w)$$

3.2 Parallelization Metrics

To evaluate the performance of our parallelization strategies, we consider several metrics, including speedup, scalability and efficiency, defined as:

$$sp(n) = \frac{T_{seq}}{T_{par}(n)}$$

$$sc(n_w) = \frac{T_{par}(1)}{T_{par}(n_w)} = \frac{sp(n_w)}{sp(1)}$$

$$\epsilon(n_w) = \frac{T_{id}(n)}{T_{par}(n_w)} = \frac{sp(n_w)}{n_w}$$

3.3 Amdahl's Law and Ideal Speedup

Amdahl's Law quantifies the potential speedup of a parallel algorithm given a proportion of the algorithm that cannot be parallelized (the serial time).

It is defined as:

$$id\_sp(n_w) = \frac{T_{seq}}{f \times T_{seq} + T_{id}^{1-f}(n_w)} = \frac{1}{f + (1-f)/n_w}$$

where $T_{id}^{1-f}(n_w)$ is the ideal speedup on the non serial fraction of the algorithm.

In a setting with potentially infinite workers, this translates to the limit to which parallelization can speed up the computation:

$$\lim_{n_w \to \infty} id\_sp(n_w) = \frac{1}{f}$$

In our setting, because of the calculated $f = 0.057\%$, this would mean a very unrealistic speedup of 1755x.

4 Experiments Setup

We implement our solutions with native C++ threads and FastFlow. The computation is run in a 32-core architecture with hyperthreading, bounding our analysis' zone of interest in the range of 1-64 workers. All experiments are run on the Canada dataset of cities, and can be replicated running "`source cmd/main.txt`".

4.1 Implementations

4.1.1 Native Threads

In our first implementation of parallelization for the algorithm, we utilize the C++ native threads. To ensure correct behavior and prevent data corruption when multiple threads access shared resources concurrently, we employ mutexes. Mutexes, short for mutual exclusion, are synchronization primitives used to protect shared data from simultaneous access by multiple threads. When a thread acquires a mutex lock, it gains exclusive access to the shared resource, preventing other threads from accessing it until the lock is released. In our implementation, mutexes are used to safeguard critical sections of code, e.g., when pushing offspring chunks into the same vector to aggregate the chunks. By acquiring a mutex lock before accessing the shared resource and releasing it afterward, we prevent race conditions that could lead to unpredictable behavior or data corruption.

We explore both a static and a dynamic solution for our problem. In the dynamic solution, we define a ticket queue, where each worker can ask for a ticket with just the information on how many parents should be picked and evolved. This incredibly simple mechanism of synchronization works because of the independence assumptions exposed in Section 3.1.

4.1.2 FastFlow

Lastly, we implemented the genetic algorithm with the FastFlow library. We make use of the `poolEvolution` class, which is designed for genetic applications. This class requires functions which operate on individuals, which are analogous to the phases we already implemented, reported in brackets when with a different name: selection, evolution (crossover + mutation), filter (merge). The basic idea, reported in the library source code, is reported in Algorithm 2.

Algorithm 2 poolEvolution

```
1: with env as environment
2:   while(not termination()) do
3:     new_population = evolution(selection(population))
4:     population = filter(new_populaiton, population)
5:   end while
```

As we can see, a termination function for the while loop is required, as is typical of FastFlow. To decide on termination, we simply keep track in our environment of the number of generations passed, and once we reach n_{gen} generations, the termination function returns true. Moreover, the evolution process for the `poolEvolution` class is done individual by individual. To make this compatible with the crossover part of the evolution, a random parent is chosen from the environment, where the parents are stored. Again, this is a shared resource, but there is no data race. In fact, investigating the `poolEvolution` by printing some checkpoints, we noticed that the selection is done fully sequential. For this reason, and for the fact that during evolution workers are in read only mode for this variable, these shared mechanisms generate no data races.

4.2 Experiments

With all three implementations (static and dynamic with native threads, fastflow) we run tests with $n_w = 1, 2, 4, 8, 16, 32, 64$ workers, on populations of 2048 and 8192 individuals (with 50% of population selected as parents for crossover, and all offspring being mutated). We run the evolution for $n_{gen} = 10$ generations, to have some significant statistics in terms of the benefits of parallelization, as running just one generation may incur in strong variability due to the stochasticities of the evolution process. As for the chunk size of the dynamic scheduling, initial trials were run to decide this parameter and, confirming the intuition in Section3.1.2, the smallest chunk possible (which is 2, because of crossover) yielded best performance.

4.3 On Implementation and Ideation Process

In this section, some key points of the ideation process for the implementation and experiments are given, to understand the workflow of the project.

Initially, the native thread implementation was meant to be the static one. However, when analyzing the loads of the workers, we noticed a strong imbalance. For this reason, the dynamic solution was implemented.

One of the big issues with the project was that a certain point, dynamic strategy was giving way higher times (tens of millions of microseconds more), and ultimately it was due to not locking a mutex before writing the worker time statistics to a shared vector where each worker wrote at his index. Looking back at the theoretical aspect of this, I think it could be a typical example of false sharing, where because of cache coherence protocols each worker was waiting a lot of time due to other workers actions. Fortunately, the bug was caught thanks to the fact that total times differed in a strange way with respect to statical scheduling strategy, which made it evident that there was a problem.

Moreover, all native threads solutions were implemented as classes, whilst when moving to the FastFlow environment, we realized pretty soon that in the FastFlow context this strategy didn't fit that much. In fact, what we had as class attributes, FastFlow's `poolEvolution` handles with a given environment. For this reason, it made simply more sense in our opinion to just implement the functions in a FastFlow compatible way and abandon the class setting. Fortunately, the underlying evolution functions are basically the same, and during runtime, this implementation yields extremely comparable timings, which helps us be sure that our class implementations were well thought.

5 Results and Discussion

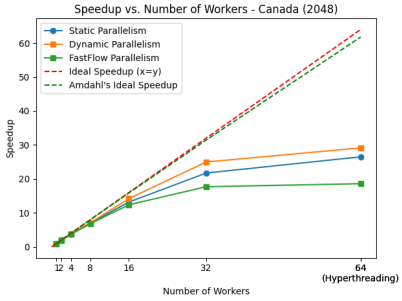
5.1 Incrementing the number of workers

The results of the first experiment are reported in the plots in Figure 1.

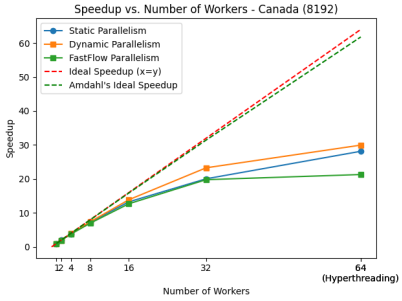
The first thing we notice is that it seems like our algorithm scales decently with the number of workers. In fact, as expected, we have a close to ideal behaviour with a small amount of workers, and the behaviour decays as the number of workers grows. We also see a decrement in the amount of improvement when going into hyperthreading. We observe that dynamic scheduling is the most performing version, in terms of speedup, scalability and efficiency. One thing we wouldn't expect is the fact that the FastFlow solution is outperformed by the static implementation, since the former has dynamic scheduling which, as a strategy, seems to outperform the latter.

To explore how the dynamic scheduling outperforms the static one, we keep track of the load imbalances, and we report them, for $n_w = 2$ and $n_w = 64$ in Figure 1g and 1h. First of all, we notice how the dynamic scheduling is suffering from a very high imbalance, which validates our intuitions from the theoretical analysis. Moreover, we see how the load imbalance diminishes with the size of the population, which is again something that confirms our expectations, since having larger chunks of population stochastically reduces the imbalance of the workload.

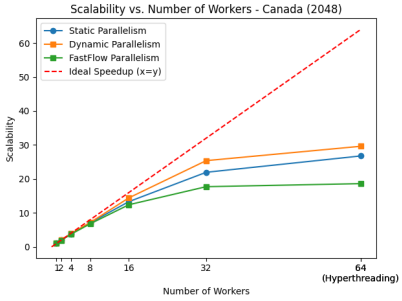
In Figures 1a and 1b, the Amdahl's Law line looks like a straight line. Ac-



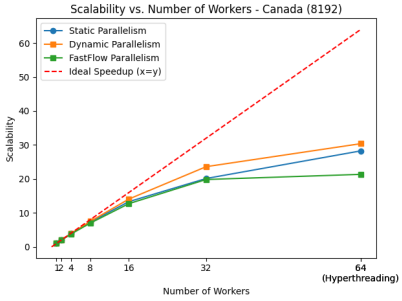
(a) Speedup ($n_{pop} = 2048$)



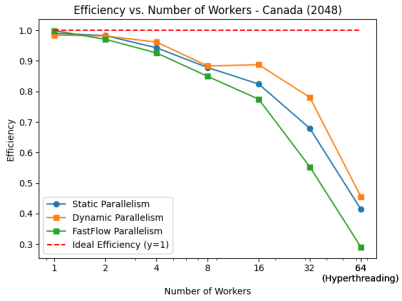
(b) Speedup ($n_{pop} = 8192$)



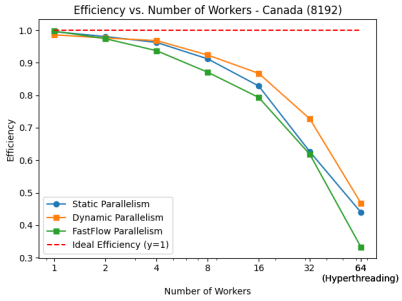
(c) Scalability ($n_{pop} = 2048$)



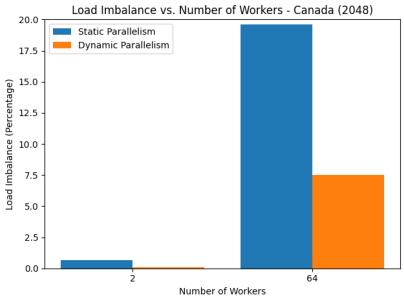
(d) Scalability ($n_{pop} = 8192$)



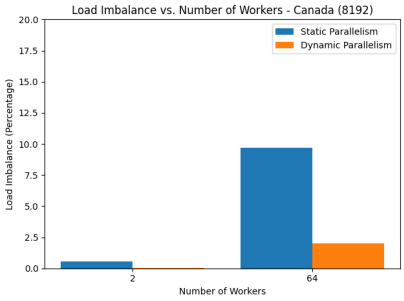
(e) Efficiency ($n_{pop} = 2048$)



(f) Efficiency ($n_{pop} = 8192$)

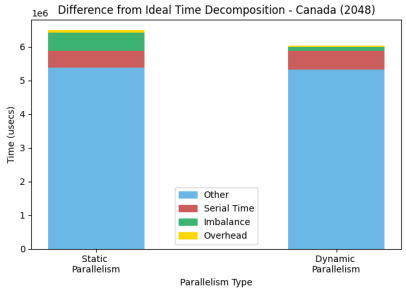


(g) Load Imbalance ($n_{pop} = 2048$)

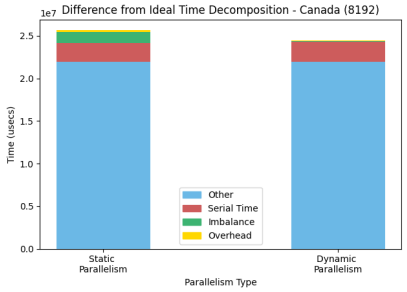


(h) Load Imbalance ($n_{pop} = 8192$)

Figure 1: Plots for Canada with Population Sizes 2048 and 8192



(a) Overhead Decomposition for Static Parallelism



(b) Overhead Decomposition for Dynamic Parallelism

Figure 2: Overhead Decomposition for Static and Dynamic Parallelism

tually, it should look like a function that asymptotically reaches the maximum ideal speed-up (calculated in Section 3.3 to be 1755x), but we are so far from that speedup, and the number of workers to reach somewhere close to it, that the line looks straight.

Another thing we notice is the extremely high efficiency with a small amount of workers, which decays progressively. While in a typical setting we would expect this not to be the case, at least for $n_w = 1$, as we are introducing overheads of the parallel computation (such as forking and joining) without any parallelization gain, this make sense in our case since the computation time is dominated by phases like crossover which are many orders of magnitude more than these overheads.

5.2 Overheads

What factors cause the difference between the ideal case and the actual speedup? We choose to analyze this with $n_w = 16$ since using 32 workers may already bring us into hyperthreading because one thread is in charge of the main program. In Figure 2 we report the breakdown, dividing it in various sources: serial time, total imbalance and overhead, where serial time is defined as in Section 3.1.1, whilst the last two are calculated as:

$$\begin{cases} T^{imb} = n_{gen} \times (\bar{T}^{max}_{w_i} - \bar{T}_{w_i}) \\ T^{oh} = T^{ns} - n_{gen} \times \bar{T}^{max}_{w_i} \end{cases}$$

We can observe things that we expected from our theoretical analysis in both population sizes: imbalance is a lot higher in static parallelism and the serial fraction is pretty much the same in both case. Unfortunately, the largest source of slowdown remains unknown from this analysis. These are probably due to the very frequent timer stops and recordings, which break the computation, memory allocations and stochasticity in the algorithm.