Parallel Genetic Algorithm Implementation for Solving the Travelling Salesman Problem

Parallel and Distributed Systems: Paradigms and Models

UNIVERSITÀ DI PISA

Lorenzo Leuzzi

1 Introduction

The Traveling Salesman Problem (TSP) is a famous combinatorial optimization challenge. Its fundamental premise involves determining the shortest possible route, while visiting each city exactly once and returning to the starting city, given a list of cities and the distances between them. Despite its seemingly straightforward description, the TSP's computational complexity is underscored by its classification as NP-hard. As the number of cities increases, finding an optimal solution becomes exponentially more intricate.

This report focuses on demonstrating methods for parallelizing the evolution process of genetic algorithms in order to expedite the search for optimal solutions to the Traveling Salesman Problem. The goal is to showcase how parallelization can significantly enhance the algorithm's performance and its ability to handle complex instances of the problem.

1.1 Problem Formulation

The project was developed in C++ programming language, and consists in three different version: Sequential Version, Parallel Version using Native Threads Libraries and another Parallel Version using FastFlow's framework [1]. In order to shape the problem I considered a real-world scenario where Italian cities serve as nodes in a fully connected graph. I retrieved data from the International TSP Library [2] and used it to create a distance matrix represented as a two-dimensional vector. This matrix encapsulates the Euclidean distances between the actual geographical positions (coordinates) of the specific cities relevant to the problem. This initial phase is managed within the `Graph` class, which is defined in the header file named `graph.hpp`.

The population is represented as a vector of chromosome (`std::vector<Chromosome>`) where each chromosome is a struct containing the path (`std::vector<int>`) and the fitness (`int`) corresponding to the distance required to visit all the cities in the given path. After the random creation of the initial population, the evolutionary process unfolds as a cyclic iteration of the following operations applied to each chromosome of the population:

- **Selection**: individuals are chosen for the next generation. The selection is executed based on a probability distribution that is directly proportional to the fitness of each individual i.e. individuals with superior fitness are more likely to be selected as parents for the next generation.
- **Crossover**: this operation results in the creation of a new individual combining two other chromosomes (the parents) with a given probability. To ensure that every path remains a valid permutation of the cities, devoid of duplicate cities and ensuring the inclusion of all cities, the Partially Mapped Crossover (PMX) technique is employed.
- **Mutation**: randomly swaps two points of the chromosome.
- **Fitness Evaluation**: calculating the distance between the initial city and the final city, while following the path that circumnavigates all cities and returns to the starting point.

The algorithm requires some parameters to run: number of cities in the path, size of the population, maximum generations, crossover rate, mutation rate and seed for random number generation.

2 Sequential Version

The initial version of the project lays the foundation for our exploration into solving the Traveling Salesman Problem (TSP). This version represents a sequential execution of the genetic algorithm, allowing us to establish a baseline performance metric against which we can measure the improvements achieved through parallelization.

The first phase involves creating an initial population. This is done by assigning to each path a vector containing every city from 1 to `chromosomeSize` and randomly shuffling it. The use of a random seed ensures that the same set of initial populations is obtained when the same seed value is employed, resulting in reproducible problem initialization. For each generation in the evolutionary algorithm, the population undergoes a series of operations. First, we select the best individuals, as described earlier (function `populationSelection`). Once we have a pool of parent individuals

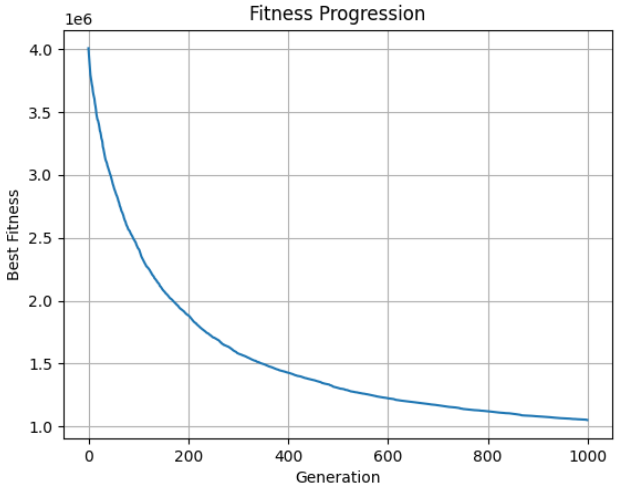


Figure 1: Best chromosome's fitness through generations.

(`std::vector<Chromosome>`), we sequentially pick near pairs of parents to create new offspring. Each new individual substitutes the same position as its corresponding first parent in the original population. The creation of a new individual involves probabilistic processes, including crossover, mutation, and the subsequent evaluation of the path to calculate its fitness (`evolution`). Following this, we proceed to the next generation.

To demonstrate the convergence of the genetic algorithm, it was executed with the following configuration: 1000 `chromosomeSize`, 1000 `populationSize`, 1000 `maxGenerations`, 0.8 `crossoverRate`, 0.5 `mutationRate`, 123 `randomeSeed`. The results, as depicted in Figure 1, indicate that the fitness of the best individual in the population improved significantly over the generations, decreasing from approximately $4 \cdot 10^6$ to 10^6. This demonstrates that the algorithm progressively enhanced the solution with each generation. It's worth noting that the choice of a large number of generations (1000) was specific to showcase convergence in this particular experiment. In practice, a smaller number of generations may be employed to work with larger problem sizes or expedite experimentation.

3 Parallel Design

In a genetic algorithm for the TSP, each solution is represented as a chromosome. These chromosomes can be independently evolved and evaluated. This means that multiple chromosomes can be processed in parallel without interfering with each other. The genetic operations like crossover, and mutation are applied to each chromosome separately, making it conducive to parallel execution. Therefore the chosen approach involves dividing the population into, non-overlapping chunks, with each thread assigned to operate its dedicated portion of the vector and apply to its elements the evolution steps. Remarkably, this approach minimizes synchronization requirements and eliminates concerns about data races within the population.

On the other hand, the selection phase includes sorting the population and probabilistically selecting a random individual from the entire population. Parallelizing this process presents challenges due to the possibility of selecting an individual from another thread's chunk, making it difficult to guarantee that the chosen individual has not already undergone evolution or been modified in some manner - it's necessary that the selection involves only the individual from the previous generation. It's important to note that even though this process does not inherently lead to data races, as it mainly involves just reading from other chunks and modifying within each thread's designated chunk, it is problematic in ensuring the algorithmic integrity of the selected individuals. Trivially, parallelizing the sorting operation by having each thread sort its sub-portion was not used as an option. This is primarily because sorting is typically a relatively fast operation, especially since the population is already somewhat already sorted after the first iteration of the algorithm.

Given that each thread in the parallelized implementation performs identical operations over all

elements within its assigned chunk, the need for load balancing mechanisms was deemed unnecessary. This assertion is rooted in the improbable scenario that any single worker would execute a significantly greater number of operations than others, primarily due to the large population size and the uniform distribution used for random number sampling. In particular, from an experiment, it was observed that, for instance, when utilizing a crossover rate of 80%, the average number for the crossover operation to occur was approximately 791 units, with a corresponding standard deviation of roughly 16, accounting for approximately 2% of the data spread, not a significant amount.

3.1 Performance Modelling

Let m be the chromosome size, n the population size and L the maximium number of generations, then the sequential time is:

$$t_{seq} = L \times (t_{popSel} + t_{popEvo}) \tag{1}$$

$$t_{popSel} = n \times t_{sel} \tag{2}$$

$$t_{popEvo} = n \times (t_{cross} + t_{mut} + t_{fit}) \tag{3}$$

and considering the complexity.

1. $t_{popSel} \approx \mathcal{O}(n)$
2. $t_{popEvo} \approx \mathcal{O}(m \times n) + \mathcal{O}(n) + \mathcal{O}(m \times n) \approx \mathcal{O}(m \times n)$
3. $t_{seq} \approx L \times (\mathcal{O}(n) + \mathcal{O}(m \times n) + \mathcal{O}(n) + \mathcal{O}(m \times n))$
   since $L << m, n$
   $t_{seq} \approx \mathcal{O}(m \times n)$

These observations indicate that the majority of the computational load arises during the evolution phase. This, along with the previously exposed issues, validates the earlier decision to exclude from parallelization the selection phase. Furthermore, the recorded timings from the sequential version highlight that the selection process is considerably faster than the evolution phase, with this sequential fraction f accounting for only small percentage of the overall computation time. Although we are aware that the theoretical maximum speedup achievable with p workers is potentially p it's crucial to acknowledge Amdahl's Law. According to it, even in the hypothetical scenario of having infinite resources and zero overheads, the maximum speedup attainable would still be limited to a value not exceeding $1/f$. However, achieving this level of speedup, especially with large input sizes, is practically impossible.

	t_{popSel} μs	t_{popEvo} μs	f	$1/f$
smaller input size	605	31622	1.9%	52.63
bigger input size	15892	2112962	0.7%	142.86

4 Parallel Version using Native Threads Libraries

Following the completion of the selection process, we can perceive the evolution as a data parallel *Map* operation that transforms the parents into the new population. This operation can be parallelized by spawning p threads and assigning an equal-sized chunk of work to each thread. Once each thread has completed its task, they wait for all threads to finish (join operation), and subsequently, the vector of worker threads cleared, proceeding to the next generation.

```
for(int i = 0; i < numWorkers; ++i)
    workers.push_back(new std::thread(body, i));
for(auto w : workers) w->join();
workers.clear();
```

5 FastFlow Version

The parallel implementation, utilizing the Fastflow framework, focuses on a Source-Filter mapping of `TASK`s. `TASK` represents a helper structure that includes both pointers to the parent's and the population's vectors. This setup ensures that any modification made by one node is reflected in the other node. To match the design requirements, we have defined two FastFlow nodes:

1. **Source**: This class is a subclass of `ff::ff_node_t<TASK>`. Its primary role is to generate tasks and dispatch them to the filter node. It keeps track of the current generations and, upon reaching the maximum allowed, it sends an End-Of-Stream (EOS) pointer to indicate completion.

2. **Filter**: It is also a `ff::ff_node_t<TASK>` class where the core algorithmic operations take place within the `svc` method. Initially, the population is sequentially selected. Subsequently, using the `ff::ParallelFor` object from the class, the evolution process is executed in parallel. The `parallel_for` method is invoked with parameters that include the index of the first and last elements, the step size, and the chunk size. The functions applied both for the selection and evolution are passed as parameters when creating the `Filter` object.

These nodes are integral components of a pipeline in which the final node (Filter) is linked to the initial one (Source), ensuring the continuous progression of generations.

```
Source source(&population, &parents, populationSize, maxGenerations);
Filter filter(populationSelection, evolution, numWorkers);
ff::ff_Pipe<> tsp(source, filter);
tsp.wrap_around();
tsp.run_and_wait_end();
```

6 Experiments and Results

6.1 Experiments Settings

The experiments were run on the remote machine available for this the course which is a dual socket AMD EPYC 7301 machine. Each socket has 16 cores (total 32 cores), 2 way hyper threading for a total of 64 hw threads.

The algorithm was run with both smaller input size (1000 chromosome size, 1000 population size) and bigger input size (5000 chromosome size, 5000 population size). Each of this version was tested with different parallel degrees $p \in 2^i, \forall i \in 1, ..., 6$. Throughout every execution, the remaining parameters remained constant: a maximum of 10 generations, a 0.8 crossover rate, and a 0.5 mutation rate. The results presented below are obtained by averaging the outcomes from five consecutive runs of the identical instances.

6.2 Performance Measures

6.2.1 Smaller Input Size

With a smaller input size, the maximum speedup, which also aligns with the peak scalability, is attained with 16 workers for the native parallel version and 32 workers for the fastflow version. Beyond these points, for both versions, completion times increase, indicating that we are initiating more threads than required. Naturally, the highest efficiency is observed when utilizing just 2 workers because the speedup is nearly optimal, suggesting that the resources are almost fully utilized. We anticipate more favorable outcomes with larger input sizes of chromosome size and population. The known sequential execution time with these settings is 359.33 ms, and Table 1 presents all the corresponding performance metrics.

6.2.2 Bigger Input Size

When the input size is increased from 1000 to 5000 for both chromosome size and population size, we observe that the quickest execution for both the *par* and *ff_map* versions occurs when utilizing the maximum number of threads, which is 64. The parallel version using the native library reaches

par	2	4	8	16	32	64
time (ms)	215.33	122.33	75.00	**60.00**	62.33	88.33
speedup	1.68	2.93	4.78	**5.68**	5.11	4.07
scalability	1.91	3.34	5.45	**6.49**	6.22	4.29
efficiency	**0.84**	0.73	0.60	0.38	0.18	0.06
ff_map	2	4	8	16	32	64
time (ms)	225.67	128.33	80.67	65.33	**59.67**	98.67
speedup	1.60	2.81	4.21	5.49	**6.11**	3.75
scalability	1.76	3.09	4.90	6.13	**6.61**	4.53
efficiency	**0.80**	0.70	0.56	0.34	0.19	0.06

Table 1: Performance measures for smaller input size (1000 chromosome size and 1000 population size)

a notable speedup of nearly 21. The performance of the fastflow version is slightly lower but still within acceptable ranges. For both versions, the speedup remains close to the ideal until we utilize 32 threads, after which the efficiency begins to diverge more noticeably. Table 2 displays the performance metrics with the larger input size. The average time for sequential execution in these runs is 22799.8 milliseconds.

par	2	4	8	16	32	64
time (ms)	12613.0	6920.0	3759.0	2374.6	1357.4	**1134.6**
speedup	1.83	3.71	6.69	9.71	17.37	**20.7**
scalability	2.16	4.31	8.4	12.1	21.49	**27.48**
efficiency	**0.92**	0.83	0.79	0.62	0.54	0.32
ff_map	2	4	8	16	32	64
time (ms)	12711.2	7751.6	4092.2	2147.0	1517.6	**1335.0**
speedup	1.82	3.0	5.53	10.92	15.24	**17.17**
scalability	2.21	3.7	6.7	13.21	18.53	**21.27**
efficiency	**0.91**	0.76	0.72	0.68	0.44	0.27

Table 2: Performance measures for bigger input size (5000 chromosome size and 5000 population size)

6.2.3 Overheads

The fork-join model, characterized by its simplicity, allows for the spawning of worker threads on-demand, which perform tasks and then terminate, necessitating a join operation to synchronize with the main thread. However, this model can incur overhead, particularly when dealing with a large pool of workers. Each thread creation and join operation introduces a latency that, while individually measured in microseconds (typically 10-100 μs), can accumulate to a total of 6.4 to 64 ms given a maximum thread count of 64. This overhead can impact performance, although it may not be as noticeable with larger input sizes where the computation time exceeds the overhead by a considerable margin.

An alternative to this approach is the implementation of a thread pool, which can offer improved efficiency by reusing a fixed set of threads to perform various tasks. This method avoids the cost of frequent thread creation and destruction. However, it also introduces complexity related to synchronization. Ensuring that tasks are executed in an orderly fashion without race conditions requires meticulous management of task queues and worker synchronization, which can be challenging.

Another potential source of overhead is false sharing, an instance where multiple threads inadvertently impact each other's performance through the cache system, leading to unnecessary cache misses and memory traffic. I considered padding data structures to mitigate false sharing, aiming to ensure that each thread operates on a distinct cache line. However, this technique did not yield a noticeable improvement in speedup for this particular case, suggesting that false sharing may not be a significant concern in this context or that the data access pattern does not lend itself to easy optimization through padding.

6.3 Plots

To enhance the comprehension of performance improvements when scaling up parameter degrees, I graphically represented the data. The resulting plots for Speedup, Scalability, and Efficiency are displayed in Figures 2, 3, and 4, respectively.

7 Usage

To compile the individual versions, use the following commands:

```
g++ -O3 -std=c++17 seq.cpp -o seq
g++ -O3 -std=c++17 parallel.cpp -o par -pthread
g++ -O3 -std=c++17 ff_map.cpp -o map -pthread -I path/fastflow
```

And to run each program, use these commands:

```
./seq <chromosomeSize> <populationSize> <maxGenerations>
      <crossoverRate> <mutationRate> <seed>
./par <numWorkers> <chromosomeSize> <populationSize> <maxGenerations>
      <crossoverRate>  <mutationRate> <seed>
./map <numWorkers> <chromosomeSize> <populationSize> <maxGenerations>
      <crossoverRate> <mutationRate> <seed>
```

If no parameters are specified, the algorithms will run with default values.
To execute the entire experiment, including running a sequential version and various parallel versions for each parallel degree value, and then comparing the times to calculate speedup, scalability, and efficiency, use the following command:

```
sh run.sh <chromosomeSize> <populationSize> <maxGenerations>
          <crossoverRate> <mutationRate> <randomeSeed> <c>
```

here, *c* is a flag to compile the source code first if needed.

8 Conclusion

The exploration of parallelizing the genetic algorithm for solving the Traveling Salesman Problem (TSP) has demonstrated significant insights and noteworthy results. This report presented a comprehensive study that involved three distinct versions of the algorithm: a sequential version, a parallel version utilizing native threads libraries, and another parallel version leveraging the FastFlow framework.

The experiments conducted on different input sizes and parallel degrees illustrated the scalability and efficiency of the parallel approaches. The native threads version and the FastFlow version both demonstrated decent speedups and scalability, with the FastFlow version slightly worst. Notably, the FastFlow version, with its structured approach to parallel programming, allowed me for better understanding of parallel skeletons.

In conclusion, the study reinforces the effectiveness of parallel computing in solving complex, computationally intensive problems like the TSP. The results I obtained drew my attention to the potential for improving performance measures by paying closer attention to overheads. This could potentially lead to further optimized solutions for parallel problems like the Travelling Salesman Problem.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: High-level and efficient streaming on multicore. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-Core and Many-Core Computing Systems*. Wiley, 2017.

[2] University of Waterloo. The traveling salesman problem for world countries.

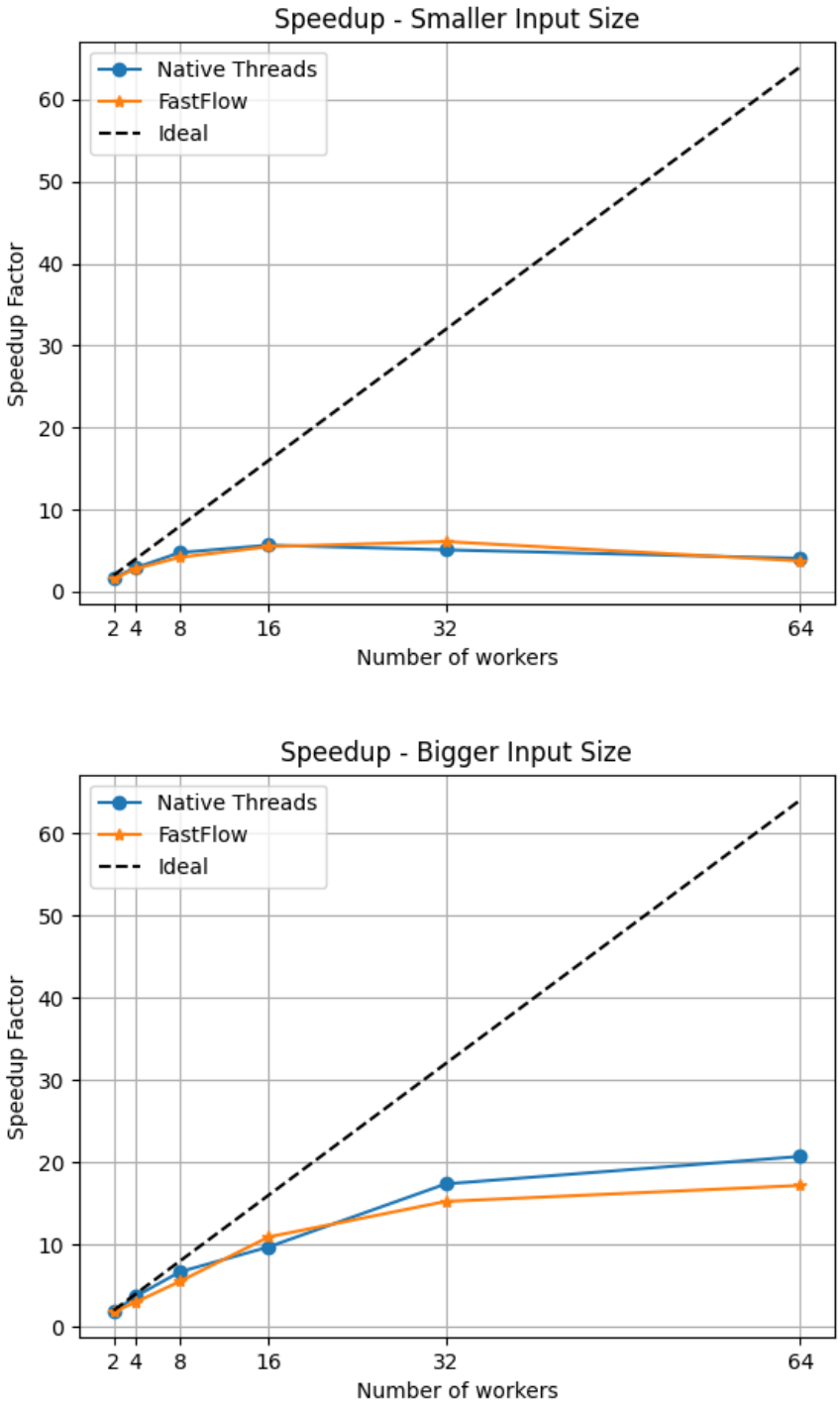


Figure 2: Speedup Plots

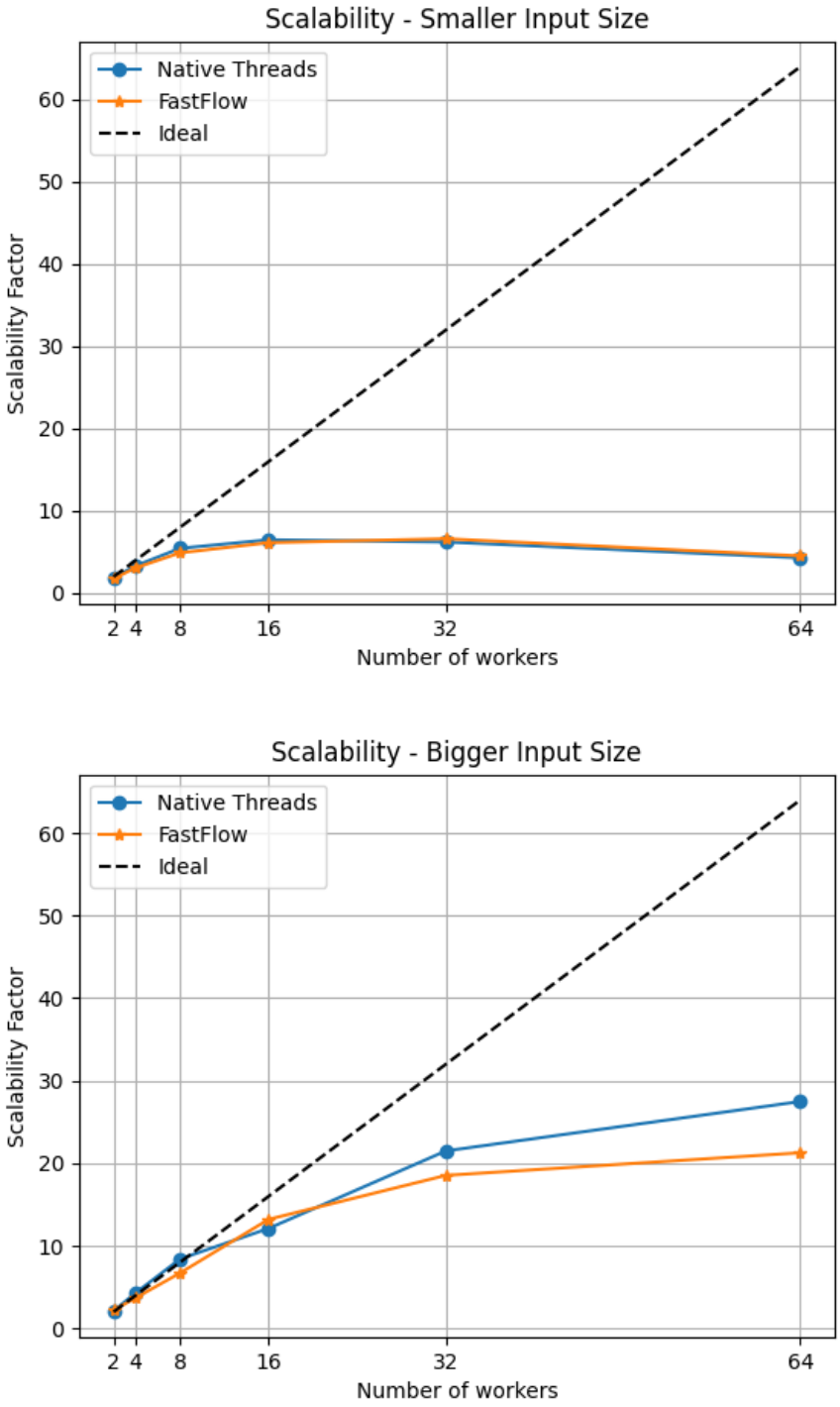
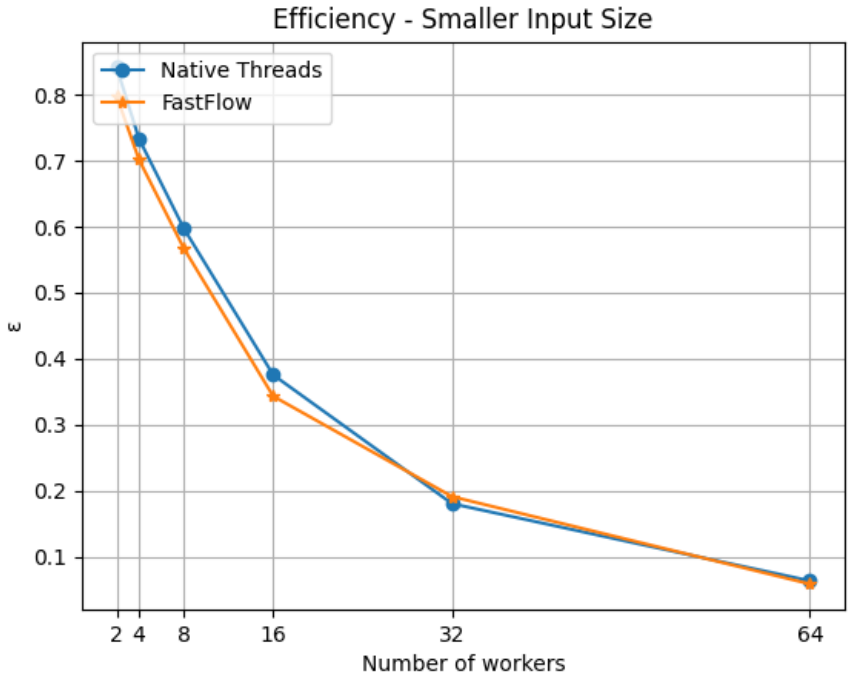


Figure 3: Scalability Plots



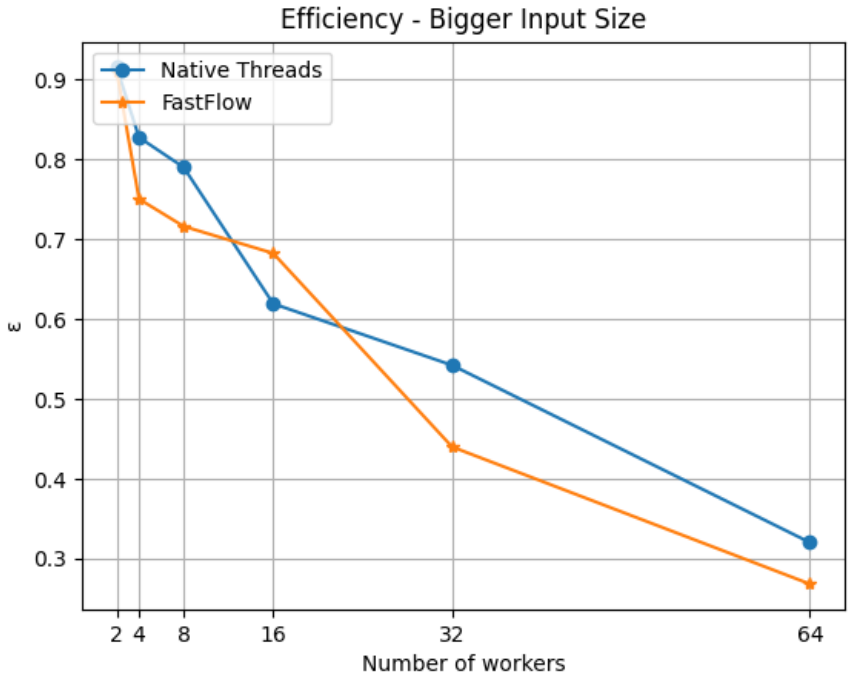


Figure 4: Efficiency Plots