IMPLEMENTATION AND PARALLELIZATION OF A GENETIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM

Valentin Anklin, Hannes Pfammatter, Justin MacPherson, Sebastian Kurella

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

We implemented and optimized a sequential genetic algorithm for the traveling salesman problem and extended it to a parallel genetic algorithm. We use the island model for our algorithm and run it on multiple CPU cores using MPI. Depending on the problem instance, we achieve a close-to linear speedup for up to 24 cores. In addition, we use CUDA to run our algorithm on a GPU. The CPU version of the algorithm outperforms the GPU version and reaches a path length that is 8.75 times smaller in less time on the d1291 problem.

1. INTRODUCTION

The use of genetic algorithms (GAs) as heuristics has helped to solve real-world optimization problems in a wide range of areas such as engineering [1]. The ability to run a GA in parallel can help solve these problems much faster than before. In this project, we set out to parallelize the classical GA for a specific problem instance: the travelling salesman problem (TSP). The TSP is a practically relevant problem that arises in many real-world situations, such as in the production of printed circuit boards.

Related work. A good overview of parallel genetic algorithms (PGAs) is given in [2], where 25 PGAs from the literature and implementations of PGAs are presented. About 70% of them are from the year 2000 or earlier. Some of the implementations are outdated or consist of emulating a PGA on a single CPU core. In [3], the most important models for distributed evolutionary algorithms (DEAs) in the literature, such as the master-slave model, island model, cellular model, hybrid model and pool model, are summarized. Libraries for PGAs are [4] and [5]. We used these as performance baselines. Finally, [6] is a state-of-the art TSP solver that uses constraint programming. We use MPI to implement a PGA that runs on multiple CPU cores and compare its performance to [4, 5, 6]. In addition, we use the CUDA runtime to implement our PGA on a GPU.

2. SEQUENTIAL GENETIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM

In this section, we give an overview of the implementation of our sequential GA for the TSP.

Traveling salesman problem. For the TSP we consider a complete graph with weighted edges. We refer to the nodes of the graph as cities. The edge weights correspond to the distances between the cities. Solving the TSP consists of finding a closed path with minimum length across all cities. The TSP is NP-complete.

Sequential genetic algorithm. We use the sequential GA outlined in Algorithm 1 as a heuristic to find an approximate solution for the TSP. It is briefly explained below.

Algorithm 1 Sequential GA

```
1:  Population ← ∅
2:  INITIALIZE(Population)
3:
4:  for Evolution = 1, ..., Number of Evolutions do
5:      RANK(Population)
6:      Next Population ← CROSSOVER(Population)
7:      MUTATION(Next Population)
8:
9:      Population ← Next Population
10: end for
11:
12: return Best Individual in Population
```

Individual. An individual is represented by its gene. A gene is a permutation of the city indices which corresponds to a closed path with fixed path length.

Fitness. Each individual has an associated fitness. We use the length of the path associated with its gene as fitness, where a smaller length value is better.

Population. The population is the set of all individuals. The number of individuals in the population is the population size. The population members are initialized with random permutations of city indices.

Elitism. We keep the fittest population members in the course of the algorithm. This is called elitism. This mechanism causes the smallest length value over all individuals in the population to decrease monotonically. Here the design choice is how many population members should be carried over into the next iteration unchanged.

Ranking. This step of the sequential GA ranks the individuals based on their length value. The ranking is needed during crossover and mutation and to determine the best individual of the population.

Crossover. This step of the sequential GA mixes the genes of two population members called parents to form a child. Parents are chosen randomly using roulette wheel selection, where population members are weighted according to their length values. Crossover is performed by selecting a contiguous sequence of city indices from the first parent and copying it into the child. The rest of the new gene is filled up by taking the city indices from the second parent that are not already in the child's gene in the order in which they appear in the second parent. The entire population is rebuilt via crossover, leaving only the elite members intact.

Mutation. In this step of the sequential GA, each population member is mutated with a certain probability, which is a parameter. A mutation is a random swap of two city indices. The elite members are not mutated.

3. PARALLEL GENETIC ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM

In this section we describe the CPU implementations of the two versions of our PGA, and our GPU implementation.

Naive parallel genetic algorithm. As the population is initialized randomly, the simplest strategy to speed up the computation is to run several isolated instances of the sequential GA in parallel on separate CPU cores. The rationale is that each execution of the sequential GA should find different individuals. Therefore, aggregated over all isolated executions, a smaller length value should be found after a certain runtime. We implemented this approach using MPI.

Island model genetic algorithm. To distribute the work of evolving the whole population, one could in principle assign a different CPU core to each subset of the population. However, as crossover operations involve all individuals of the population, this would quickly lead to a lot of communication between the CPU cores. We instead use a strategy that lies in between this and naively running isolated instances: the island model. Instead of isolating the separate executions of the algorithm, good individuals are regularly exchanged between the separate executions, also referred to as subpopulations or islands. This exchange is called migration. We implemented migration using MPI.

Migration parameters. There are various parameters for the migration, see for example [2]. The migration topology G_M can be seen as a directed graph. It determines to which subpopulations migrants of a certain subpopulation migrate. We implemented a fully connected topology and a ring topology. The selection policy S_M determines how migrants are selected from a subpopulation. We implemented pure random selection, roulette wheel selection, stochastic universal sampling and truncation selection. In the first three methods, individuals are randomly selected. In the fourth method, the individuals with the smallest associated length values are selected. The replacement policy R_M determines how incoming migrants are integrated into a subpopulation. We have implemented pure random replacement, DeJong crowding replacement and truncation replacement. In the first two methods, individuals are randomly replaced. In the third method, the individuals with the largest associated length values from the union of the subpopulation and all migrants are discarded. The migration period P_M determines after how many iterations of the GA a migration between subpopulations takes place. The migration amount N_M determines how many migrants arrive at a subpopulation during a migration. Finally, the type of MPI communication C_M used for the migration also a parameter. We implemented blocking communication and non-blocking communication. Migrants are always copied and remain in the original subpopulation.

Grid search for parameter values. As we have a lot of parameters, we have implemented infrastructure to parse values for the parameters from the command line. We have also implemented infrastructure to specify and automatically run experiments. We used this for grid searches as well. The motivation was that we did not know which parameter values of the sequential GA and the island model GA were suitable for the TSP and how different parameter values affect the found length values.

Single core optimizations. To improve the runtime of the island model GA, we have analyzed and optimized the sequential GA which is executed for a subpopulation on a single CPU core. As first optimization we have tried different compiler flags. For the other optimizations we collected profiling data. As second optimization, we replaced a set data structure used during crossover to store the genes that are already present in a child with a lookup table (LUT). As third optimization we used 16-bit integers for the genes and single precision floating point numbers for the fitness values. As fourth optimization we implemented parts of the sequential GA with AVX2 instructions. The third and fourth optimizations are exclusive because it is harder to use AVX2 instructions for 16-bit integers. For AVX2 we additionally use the compiler flag -mavx2. If not stated otherwise, we use the third code version for our experiments.

GPU Implementation. The source of parallelism leveraged in the CPU implementation, namely the plurality and homogeneity of processing individuals within a population,

was also chosen as the basis for the GPU implementation in CUDA. More parallelism exists in the fitness evaluation of an individual, however we chose not to extract it due to its marginal overall performance improvement.

GPU-accelerated code can and often does execute some of its processing on the CPU. As our algorithm runs many iterations of homogeneous epochs, using this hybrid model of computation presents a significant communication and control overhead. For this reason, we implemented certain routines on both the GPU and the CPU and were able to compare single- and multiple-kernel performance. Specifically, elitism requires the population to be sorted by fitness, a task for which CPUs ought to be well-suited. Random initialization of the population is done offline and not included in any performance measurements. CUDA thread blocks correspond to islands and each CUDA thread processes one individual in fitness evaluation and one child is crossover.

Every epoch begins with a per-individual fitness evaluation. Then, each island's population is sorted by fitness and the prefix sum over the fitness values is performed in order to perform roulette wheel selection. On the GPU, the thrust library is used [7]. Finally, a crossover kernel is invoked, in which each thread randomly selects two parents using the weights calculated above to produce a child. The crossover kernel also performs the mutation. Migration has not been implemented as the naive model already failed to outperform the CPU version.

4. EXPERIMENTAL RESULTS

In this section, we give an overview of our experimental results.

Input graphs. To test our implementations, we use the publicly available graph dataset TSPLIB [8], which includes several graphs of different sizes. As the runs take a long time to complete, we only use a subset of the graphs, namely *a280*, *d1291* and *u2319*, which are graphs with 280, 1291 and 2319 nodes respectively.

Experimental setup. All experiments were run on the Leonhard cluster on nodes with dual 18 core Intel Xeon E5-2697v4 CPUs @ 2.3 GHz, unless specified otherwise. The code was compiled with g++ 8.2.0 with the following flags: -mavx2 -O3 -funroll-loops. To run the experiments within the given usage restrictions of the cluster, we use a Python script to automatically schedule the jobs required for an experiment. Such an experiment needs to be specified in a JSON file, further details on how we conducted our experiments can be found on our GitHub repository [1].

Randomness. In all experiments, the individuals are randomly initialized. We justify this by the fact that despite the random initialization of the individuals, the progression of the smallest length value found over the runtime is consistent for different program executions. This can be explained by the fact that there are many individuals with similar length values, while the distribution of all possible length values is constant and ultimately determines the progression of the smallest length value.

Speedup definition. We use the definition of speedup from [2]. We define the runtime of the sequential implementation $T_1(\alpha)$ and the runtime of the parallel implementation on n CPU cores $T_n(\alpha)$ as the wall clock time (WCT) after which a predefined problem-specific length value α is reached. We define the speedup s_n as

$$s_n(\alpha) \coloneqq \frac{T_1(\alpha)}{T_n(\alpha)}. \tag{1}$$

Population size. We start by evaluating the effect of the population size parameter on the achieved path length. The result of this experiment can be found in Figure 1 for a single graph, the remaining results of this experiment can be found in Figure 8 in the appendix. We observe that for small problems that run for a relatively long time, a large population is advantageous, while for larger problems that run for a relatively short time, a small population is preferred. This can be explained by the fact that with a small population a depth search is conducted, as good individuals are often combined with each other. With a large population, a breadth search is conducted, which on the other hand reduces the risk of a convergence to a local minimum.

Island scaling. Based on the last experiment, we take the ideal population size for sequential execution and compare our two parallelization strategies: the naive model, in which the population size is duplicated, and the island model, in which the population is split among the CPU cores and migration takes place. The result of this experiment can be found in Figure 2. One can see that the island model outperforms the naive model by far.

Speedup. In this experiment we investigate the speedup of our two parallelization approaches. To get to a speedup

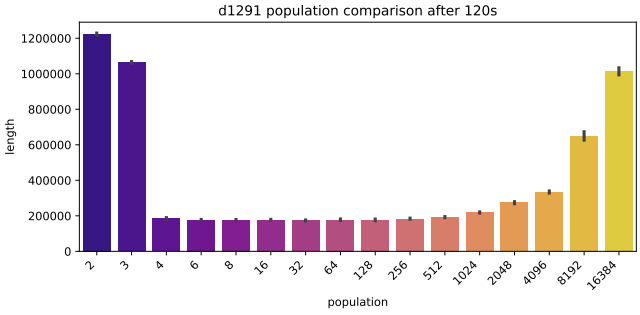


Fig. 1: **The effect of the population size.** The plot shows the path length achieved by the sequential GA after a runtime of 120s on the graph *d1291*.

[1] https://github.com/anklinv/Scalable_Genetic_Algorithm

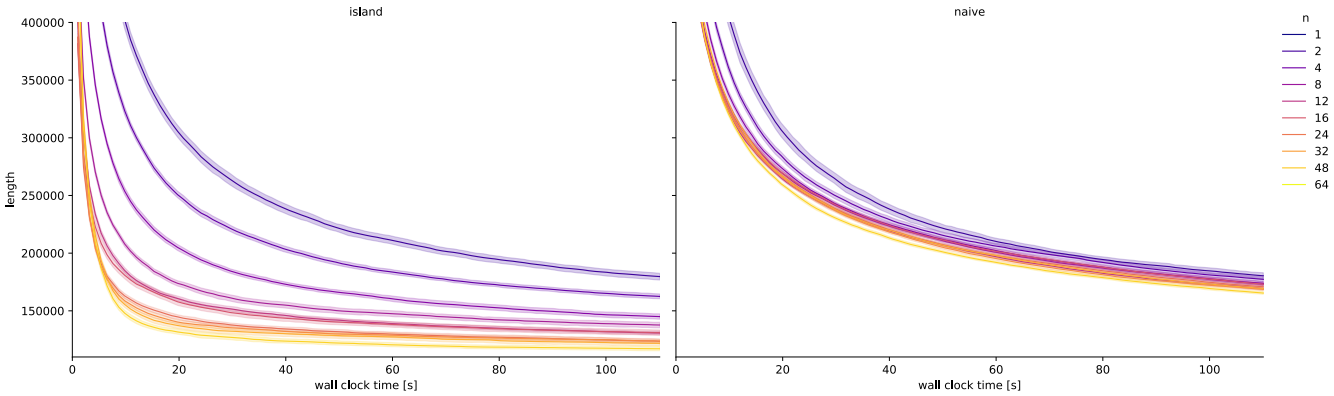


Fig. 2: **Scaling of the island model.** Path length reached over time on the *d1291* graph for the island model (left) and the naive model (right). The experiment is conducted using 30 repetitions, the error bands indicate 95% confidence intervals.

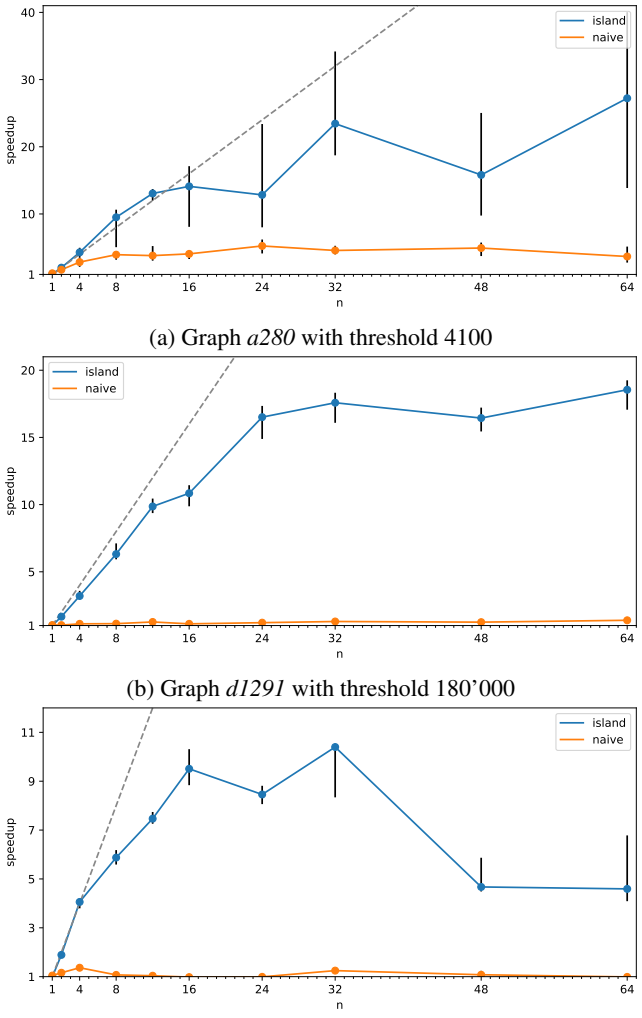
from the scaling plot, we need to define a target length value to be reached. For each graph we take the smallest length value achieved by all executions as a threshold. We then take as base case the time needed by the sequential execution to get to the speedup plots in Figure 3.

Communication overhead. To test whether it is worth to use asynchronous communication, we analyze our runs of the island model and measure the time needed for computation and the time needed for communication between islands. We do this for each migration period. We find that the fraction of the runtime spent on communication is small for a small number of islands and that we only run into problems if we have more than 32 islands (at the point where the program gets distributed across multiple machines), as can be seen in Figure 4.

Single core. The runtimes of the functions ranking, crossover and mutation, which are executed during an iteration of the sequential GA, are shown for different code versions in Figure 7. Due to the improved runtime of the sequential GA, we were able to run our parallel implementations on larger graphs and to run more program executions.

GPU results. The timing analysis of the GPU implementation in Figure 5 shows the effect that increasing the population size has on a NVIDIA GTX 1080 GPU on the d1291 problem. The population members per island stays constant at 64 members. Notice the step down in timing as the islands increase from 16 to 32 (population increases from 1024 members to 2048). Recall that a population member corresponds to a thread. The timing indicated in the graph is the time the algorithm took to each a fitness value of below 10^6. The reason why more threads seem to work better is the latency hiding that NVIDIA performs by warping threads together and performing latency hiding. The fewer threads exist, the less effective thread warping is.

Baselines. We first compare the performance of our CPU implementations to the performance of libraries for GAs. We use PGAPack, release 1.1.1 [4] and GALib, ver-



(a) Graph *a280* with threshold 4100

(b) Graph *d1291* with threshold 180'000

(c) Graph *u2319* with threshold 1'000'000

Fig. 3: **Speedup comparison.** The dashed line indicates linear speedup. The points show the median over 30 repetitions. The error bars show 95% confidence intervals.

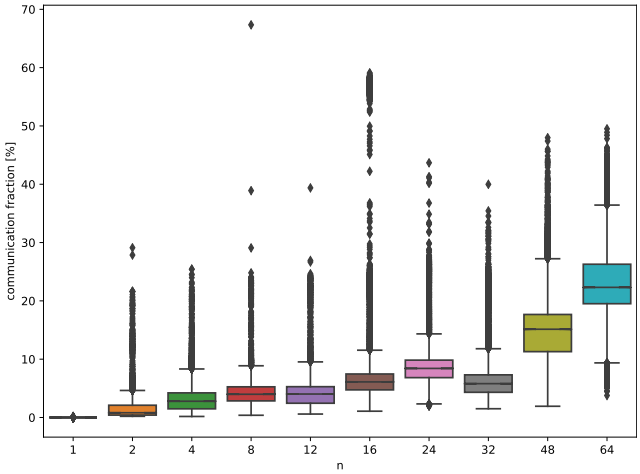


Fig. 4: **Communication overhead.** Fraction of the runtime spent on communication for different numbers of islands (n) for the graph *1291*. Each box shows 22'000 measurements.

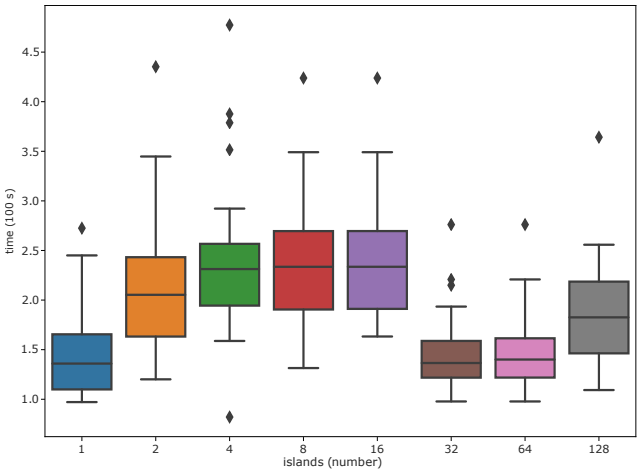


Fig. 5: **Runtimes on GPU.** Timing analysis of various population sizes on GPU. The experiment is run using 30 repetitions.

sion 0.1850-0 [5]. Both allow to specify input data, parameter values and callback functions for fitness evaluation, mutation and crossover. The result of the comparison can be seen in the speedup plot in Figure 6. The reason for the poor performance of PGAPack and GAUL for the TSP is that these libraries only parallelize the fitness evaluation. The island model is emulated on a single CPU core. This is reasonable for many optimization problems in which complex models must be evaluated based on parameter values. In the sequential GA, the fitness evaluation is done during ranking. Finally, we compare our approach with Google OR Tools [6], a state-of-the-art library to solve the TSP using constraint programming. The results of the comparison can be seen in Table 1. In comparison to the state-of-the-art solver our solution does not fare that well.

graph	path length	time [s]	method
d1291	$177'210 \pm 1'896$	117	Ours (sequential)
	$138'301 \pm 5'807$	129	Ours (parallel)
	$1'210'000 \pm 54'508$	166	Ours (CUDA)
	$53'837$	60	Google OR Tools
u2319	$1'207'016 \pm 13'578$	171	Ours (sequential)
	$631'174 \pm 63'001$	208	Ours (parallel)
	$240'734$	83	Google OR Tools

Table 1: **Comparison to TSP baseline.** The CUDA implementation is run using 6 islands. The algorithm used by Google OR Tools is deterministic.

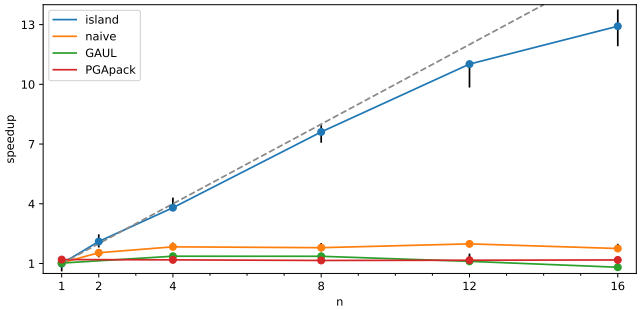


Fig. 6: **Comparison to GA library baselines.** Speedup on the graph *d1291*. The dashed line indicates linear speedup. The points indicate the median over 10 repetitions and the bars indicate 95% confidence intervals.

5. CONCLUSIONS

5.1. Discussion

In this project, we explored the application of a parallel GA to the TSP and have implemented various versions of it from scratch. Our implementations include a naive parallel implementation and an island model implementation using MPI as well as an island model implementation using CUDA. We have demonstrated that our parallelization using the island model improves the speed to get to an approximative solution of the TSP considerably.

Overall, the CUDA implementation fails to outperform the MPI implementation the tested graph d1291. As can be seen in Table 1, the reached path length is far worse than even the sequential CPU implementation.

We have also shown that in comparison to the state-of-the-art in TSP solving our implementations do not look as

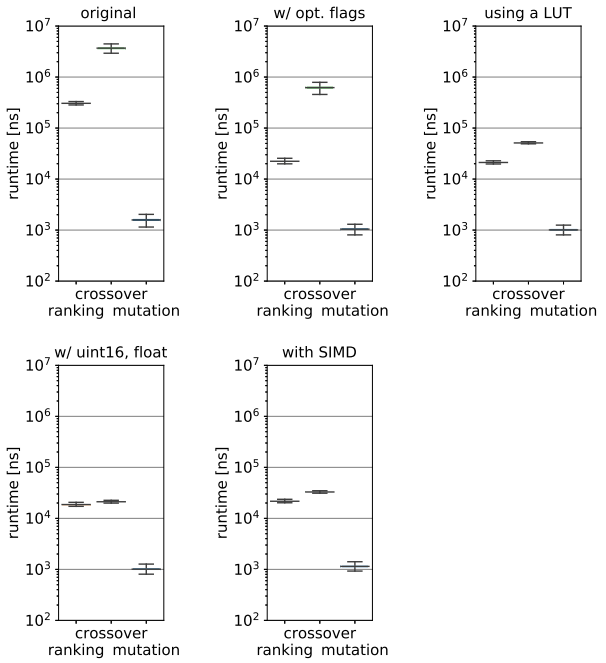


Fig. 7: **Profiling data for the sequential GA.** The number of evolution steps is 10000, which corresponds to the sample size. Outliers are not shown. The graph used is *a280*.

appealing. The reason for this is that using a GA to solve the TSP is not necessarily the best approach. As the goal of this project was to take an existing algorithm and to try to parallelize it, and not to break the records in approximately solving the TSP, we are still satisfied with the results.

5.2. Future Work

In the future, we would like to analyse which parts of the algorithm worked well in the CUDA runtime on GPU. With this information one could use the GPU to accelerate certain parts of the GA. This would allow the incorporation of MPI as well if every CPU in the cluster is also equipped with a GPU. Another direction worth exploring is hierarchical island systems, where each CPU hosts multiple islands. For each CPU, there is frequent migration and at a smaller migration rate individuals can migrate between CPUs.

6. REFERENCES

[1] Bogdan Tomoiagă, Mircea Chindriş, Andreas Sumper, Antoni Sudria-Andreu, and Roberto Villafafila-Robles, "Pareto optimal reconfiguration of power distribution systems using a genetic algorithm based on nsga-ii," *Energies*, vol. 6, no. 3, pp. 1439–1455, 2013.

[2] Gabriel Luque and Enrique Alba, *Parallel Genetic Algorithms*, Springer Berlin Heidelberg, 2011.

[3] Yue Jiao Gong, Wei Neng Chen, Zhi Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing Jing Li, "Distributed evolutionary algorithms and their models: A survey of the state-of-the-art," *Applied Soft Computing Journal*, vol. 34, no. 2013, pp. 286–300, 2015.

[4] Ralf Schlatterbeck, "Parallel Genetic Algorithm Library," online: https://github.com/schlatterbeck/pgapack, 2017.

[5] Stewart Adcock, "GAUL: Genetic Algorithm Utility Library," online: http://gaul.sourceforge.net, 2009.

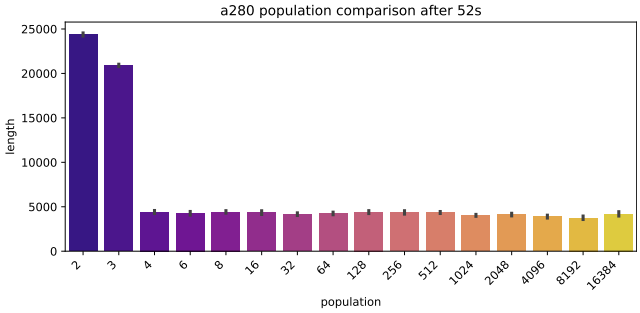
[6] Google Developers, "OR-Tools," online: https://developers.google.com/optimization, 2019.

[7] Jared Hoberock and Nathan Bell, "Thrust: A parallel template library," 2010, http://code.google.com/p/thrust/.

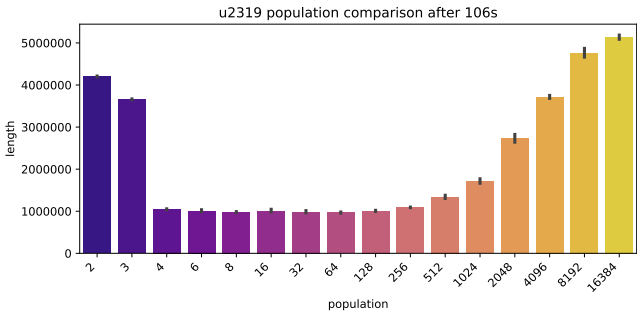
[8] Universität Heidelberg, "TSPLIB," online: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/, 2018.

A. EFFECT OF THE POPULATION SIZE

We tested the effect of the population size also on the two other graphs for which we provide speedup plots in Figure 3. As can be seen in Figure 8a for smaller problems we are tending towards the convergence region, which leads to larger populations showing their benefit. On the other side of the spectrum, we have Figure 8b, which is a large problem with 2319 nodes which we only run for a relatively short time. In this case having a large population is not beneficial.



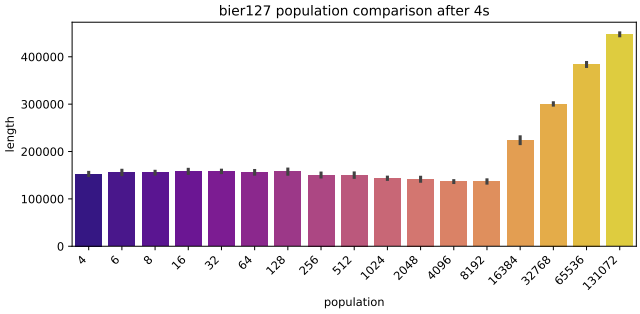
(a)



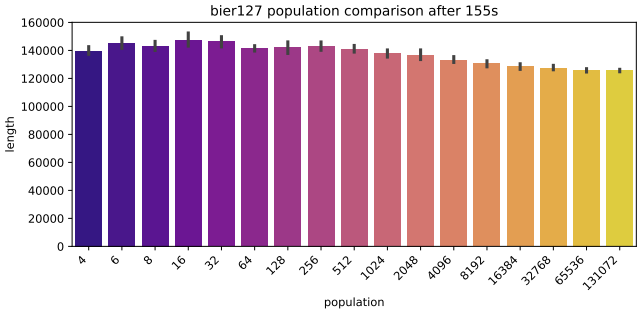
(b)

Fig. 8: **The effect of the population size on the performance.** (a) shows a graph with 280 nodes stopped after 52s, (b) shows a graph with 2319 nodes stopped after 105s

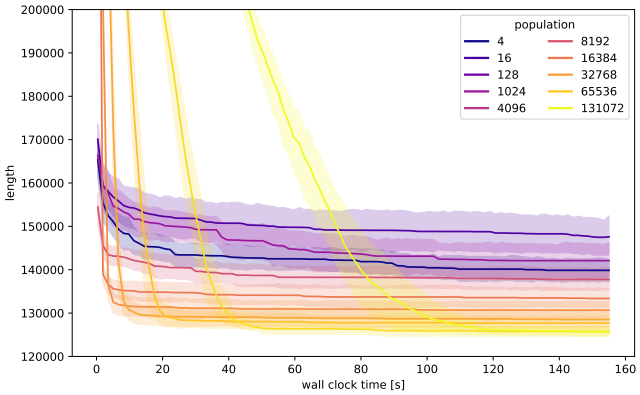
In the following, a brief explanation for the above observations is given. Having a large population leads to less evolution steps being performed in the same time compared to a small population. At the same time, a larger population allows to potentially explore more of the search space. For short runs this leads to the effect that small populations are beneficial. For long runs larger population sizes give good results. Note that the runtime is highly dependent on the size of the problem, as the search space grows at least exponentially with the size of the problem. As we were not able to run the larger problems for a long time, this is demonstrated on a the small problem *bier127* with only 127 nodes that we run for about 3 minutes. The results are summarized in Figure 9.



(a)



(b)



(c)

Fig. 9: **The effect of the population size at convergence** (a) shows that after 4s the benefits of a large population do not yet outweigh the costs, hence having a smaller population is better. (b) shows that after waiting long enough, having a larger population improves the result considerably. (c) shows how the smallest length value over time for the *bier127* problem.

B. SCALING OF THE ISLAND MODEL

We also compare the island model to the naive model on the two other graphs for which we provide speedup plots. The island model outperforms the naive model on these graphs as well, as can be seen in Figure 10.

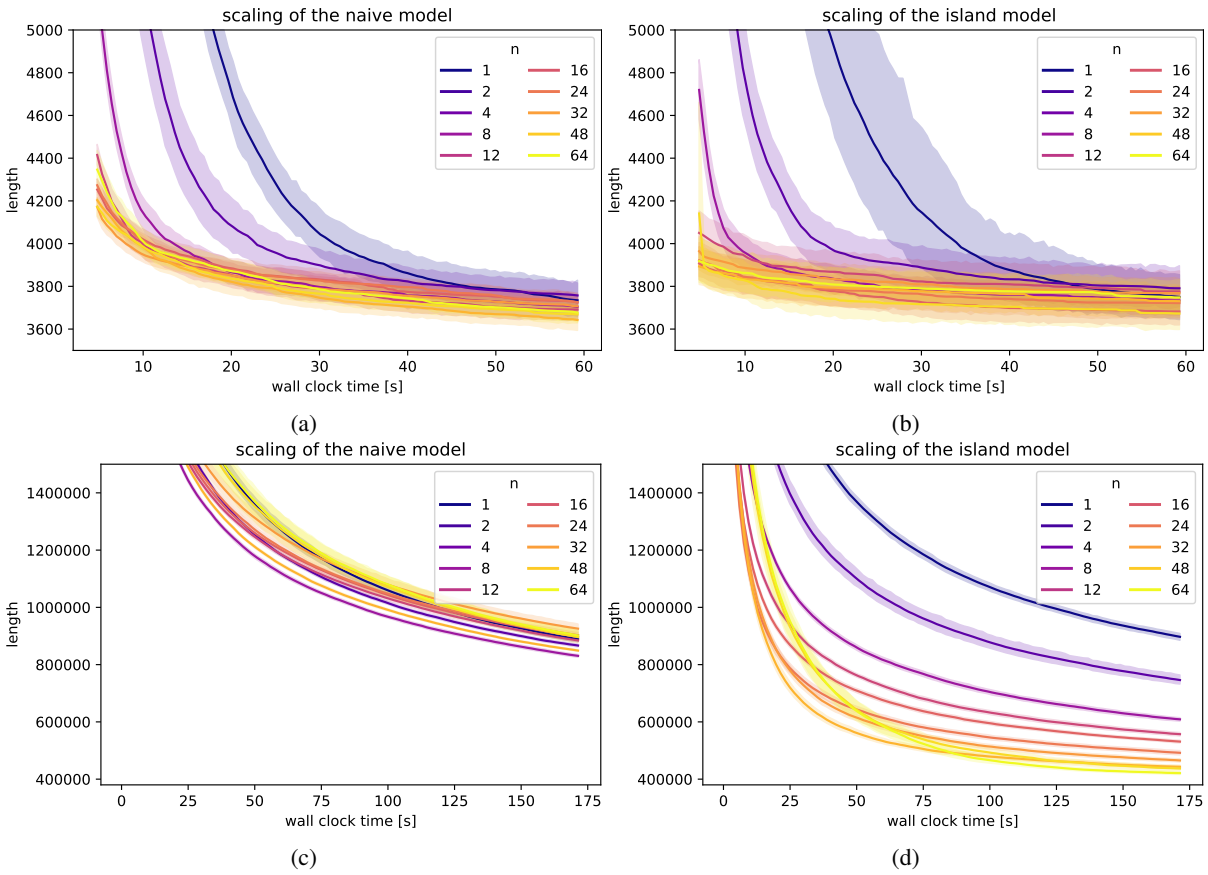


Fig. 10: **Scaling of the island model.** (a) and (b) show the progression of the smallest length value for a graph with 280 nodes for the naive model and the island model respectively. (c) and (d) show the same for a graph with 2319 nodes. The experiment is conducted using 30 repetitions, the error bands indicate 95% confidence intervals.

C. MIGRATION PARAMETER VALUES

The parameters of the migration are summarized in Tables 2 and 3. For the underlying communication, the MPI functions used are also specified.

Migration topology	G_M
Selection policy	S_M
Replacement policy	R_M
Migration period	P_M
Migration amount	N_M

Table 2: Adjustable parameters of the migration of the island model genetic algorithm excluding C_M.

C_M	G_M	MPI function
Blocking	Ring	Sendrecv
	Fully connected	Allgather
Nonblocking	Ring	Isend / Irecv
	Fully connected	Iallgather

Table 3: MPI functions used for the combinations of underlying communication C_M and migration topology G_M.

The results of three different grid searches for parameter values are shown in Figures 11, 12 and 13. For the migration topology, a fully connected topology works better than a ring topology. For the selection policy and the replacement policy, it is difficult to find good values. Good results are for example obtained by using truncation and DeJong crowding. Finally, one can see that there is an optimum for a migration period of about 40 and a migration amount of about 50% of the population size for the investigated setting.

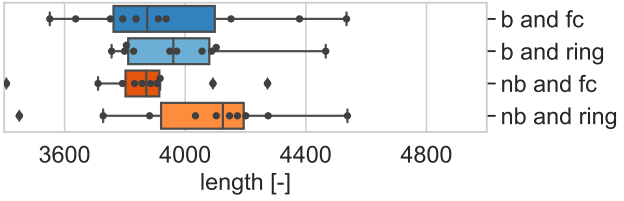


Fig. 11: Box plots which summarize a grid search over the parameters C_M and T_M of the island model genetic algorithm. The migration amount is 50% of the population size and the migration period is 20. Truncation is used for selection and DeJong crowding is used for replacement.

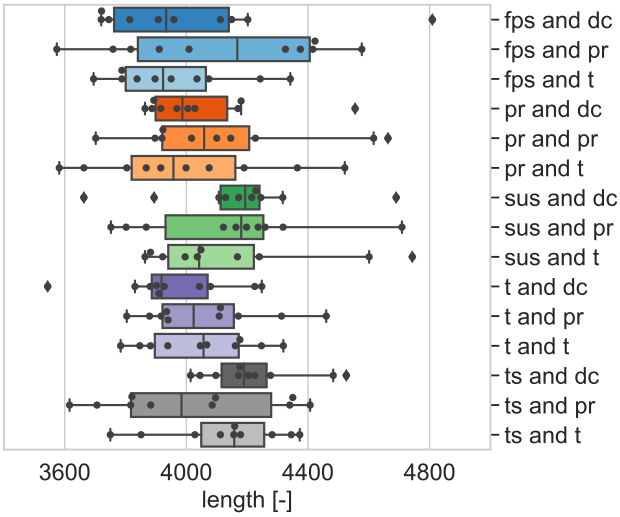


Fig. 12: Box plots which summarize a grid search over the parameters S_M and R_M of the island model genetic algorithm. A fully connected topology is used and the underlying communication is blocking.

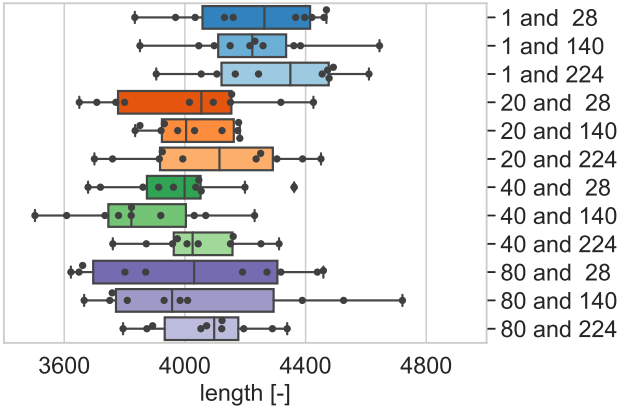


Fig. 13: Box plots which summarize a grid search over the parameters P_M and N_M of the island model genetic algorithm. A fully connected topology is used and the underlying communication is blocking. Truncation is used for selection and DeJong crowding is used for replacement.

For all combinations of parameters shown, the island model genetic algorithm is executed 10 times. The underlying TSP graph has 280 nodes and the number of islands is 8. The population size is 280, the elite size is 50% of the population size and the mutation rate is 10%. The length values shown are the length values reached after 30 seconds. Because the data is not normally distributed, it is also shown as black dots.

D. SINGLE CORE OPTIMIZATIONS

The runtimes of the different code versions of the sequential genetic algorithm are investigated in Figure 14.

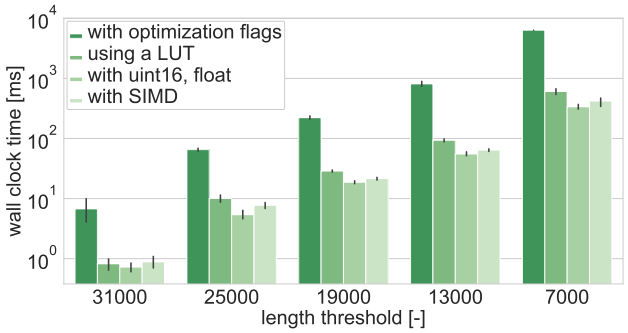


Fig. 14: Bar plot showing the runtimes needed by the different code versions of the sequential genetic algorithm to reach five evenly spaced length thresholds. The black vertical lines show 95% confidence intervals. The underlying TSP graph has 280 nodes. The size of the working set is such that it approximately fits into L2 cache.

Additionally, Figure 15 shows the effects of varying the population size while keeping the total number of crossovers fixed. First, a smaller population achieves smaller length values. This can be seen, for example, from the fact that the largest population does not reach the third length threshold. This mainly holds because the total number of crossovers and the total runtime are small. Second, cache effects are visible. This can be seen from the fact that the two smallest populations both reach the smallest length threshold, but the smallest population reaches it faster.

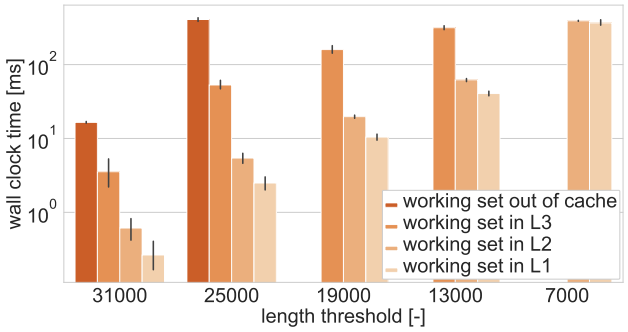


Fig. 15: Bar plot showing the effect of varying the population size from 6000 to 6 in powers of 10 while fixing the total number of crossover to 300'000. Note that the total number of crossovers is not the number of evolution steps. The population size is such that the working set approximately fits into the indicated cache levels.