DISTRIBUTION AND PARALLELIZATION OF A GENETIC ALGORITHM

Sebastian Kurella, Justin MacPherson, Hannes Pfammatter, Valentin Anklin

Department of Computer Science
ETH Zürich
Zürich, Switzerland

1. CURRENT STATUS OF THE PROJECT

In the subsections below we highlight our different genetic algorithm (GA) versions for the traveling salesman problem (TSP) implemented since the submission of our project proposal. Our code can be found on Github [1].

1.1. Sequential Implementation

Our current sequential version of the GA runs for a fixed number of epochs. Each epoch consists of three functions, whose run times are shown in Figure 1. We also measured resulting fitness per epoch and used those measurements to adapt the design of the GA and determine the weaknesses of the implementation.

We define baseline performance to be the performance of a single node running the sequential implementation. We point out that this is the underlying algorithm executed locally in other implementations.

Because of non-linearity of problem complexity, we are primarily interested in strong scaling, which we define as the ratio of the mean time to achieve a certain fitness to the mean time taken to achieve the same fitness by the baseline algorithm.

1.2. Embarrassingly Parallel (Naive) Implementation

The most naive way to parallelize the GA is to run p independent instances on p cores, initialized with different random seeds.

We compared the best fitness value per epoch $f^*_{p,i}$ on p processes to the best fitness value per epoch $f^*_{1,i}$ on one process. Example results are shown in Figure 3a. We found the improvement to be marginal in most cases.

1.3. CUDA Implementation

High variance in epoch execution times shown in Figure 1 justifies use of GPUs in our implementation. Partial progress has been made in CUDA acceleration of the parallel implementation. Work in progress.

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

1.4. Synchronous Island Model Implementation

To improve the parallel version of the GA, we use an island model with migration where the instances exchange individuals at regular intervals.

The migration uses 5 parameters: *network topology* G_M, *migration amount* N_M, *selection policy* S_M, *replacement policy* R_M and *migration period* T_M. Example values are displayed in Figure 2. The inter-process communication is synchronous and done with MPI. It is determined by N_M and G_M which specify how much data each process receives and from which process the data comes from.

Figure 3 shows the performance of the embarrassingly parallel model compared to that of the island model.

2. DISTRIBUTION OF PROJECT WORK

2.1. Coordination

We started this project by outlining in detail, what needed to be done to successfully complete the task at hand. We have since met weekly to discuss progress.

We use git and GitHub extensively for version control, managing issues and milestones and getting an overview of the entire project.

2.2. Contributions

The initial sequential version was written by Valentin. Other contributions include exploring different designs of the sequential version of the GA and creating an experiment running infrastructure on the cluster to both run jobs automatically and evaluate the results with performance plots. Hannes extended the sequential version to the embarrassingly parallel implementation and has been working on the island model with all topologies, migration and replacement policies. Justin profiled and optimized the sequential version and is in charge of the CUDA implementation. Sebastian implemented the logging infrastructure for the performance measurements and plotting.

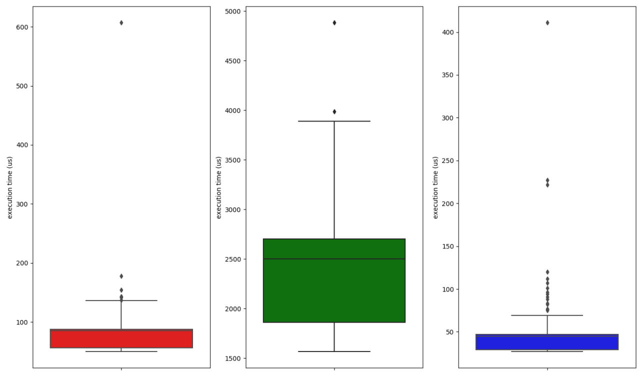
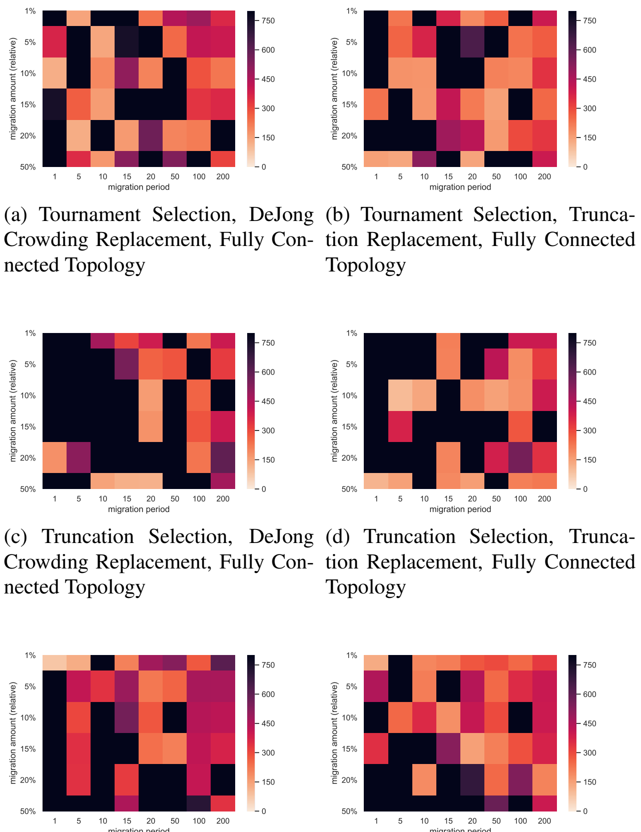


Fig. 1: Box plots showing the three functions of an epoch and their runtimes. Red: Fitness evaluation; Green: Crossover; Blue: Mutation. Notice the difference in scales.



(a) Tournament Selection, DeJong Crowding Replacement, Fully Connected Topology

(b) Tournament Selection, Truncation Replacement, Fully Connected Topology

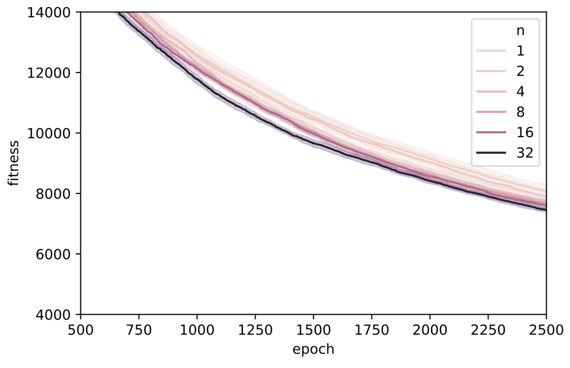
(c) Truncation Selection, DeJong Crowding Replacement, Fully Connected Topology

(d) Truncation Selection, Truncation Replacement, Fully Connected Topology

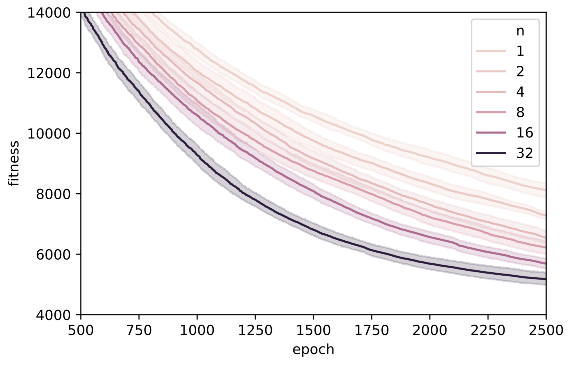
(e) Stochastic Universal Sampling Selection, Dejong Crowding Replacement, Fully Connected Topology

(f) Stochastic Universal Sampling Selection, Truncation Replacement, Fully Connected Topology

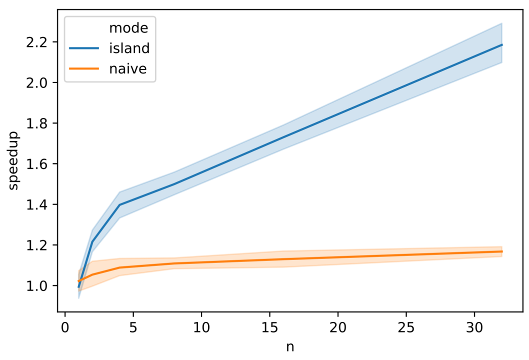
Fig. 2: Grid search over the 5 parameters of the island model. The colormap corresponds to the number of evolution steps to find a fitness value smaller than 110% of the best known solution to problem *berlin52*. The population size is 512. The number of islands is 8.



(a) Naive



(b) Island



(c) Speedup for both Island and Naive.

Fig. 3: Scaling of the naive (embarrassingly parallel) model (a) vs. the island model (b) on the *a280* TSP graph, which has 280 nodes. The speedup (c) is based on the time it takes to reach a fitness threshold of 7737. The population size is 512, the elite size 16. For the island, we use a fully connected topology. For the 95% confidence intervals we use 10 runs.