Tools of High Performance Computing

Simo Tuomisto
013456353

Final project -

Implementation of a parallel genetic algorithm for the travelling salesman problem

1 Introduction

In this report I will present an implementation of a genetic algorithm that can be used to solve the travelling salesman optimization problem. The source code is written in Fortran and it uses OpenMPI for parallelization. The goal of this project is to create a program that accurately and efficiently will solve this notoriously hard optimization problem.

This introductory section is followed by a section giving details of the genetic algorithm used. Some detail is also given to random number generation and pitfalls of the algorithm.

Section 3 contains review of the Fortran implementation. Instructions of use are also given.

In section 4 I will address the initial value dependency of the program and its accuracy.

In section 5 there are measurements on the efficiency of the parallelization.

Section 6 holds conclusions.

2 Genetic algorithm for travelling salesman problem

In the travelling salesman problem the goal is to find the shortest route a salesman can take between N cities. He has to return to his starting location after his round.

In genetic algorithms a population of different solutions is created and through selective breeding the optimal solutions eventually surface. There are two parts that really define implementations of this algorithm:

1. How does the algorithm decide which members of the population to mate?
2. How does the mating proceed?

The first question is usually solved by defining a partition function of states Z for all states x:

$$Z = \sum_i e^{-\frac{E(x_i)}{T}} \tag{1}$$

The optimal state of the system can now be found at the state where the energy finds its minimum value. Usually the whole space of possible states is huge (in travelling salesman problem there are $N!$ routes to take, when N is the number of cities) and thus the goal is to limit the search into smaller and smaller subspaces of the whole space.

In the genetic algorithm a population is drawn from the whole space. This population will have a partition function Z^*. Within this subspace the probability of finding the system in a state X is

$$P = \frac{e^{-\frac{E(X)}{T}}}{Z^*} \tag{2}$$

and thus the states with smaller energies will be picked more often. By now choosing two random numbers from this probability distribution, picking states that correspond to them from the population and mating them with a

suitable algorithm that prefers the energy minimizing qualities of the parents, a new (low energy) state is born. By replacing high-energy states in the population with low-energy states the algorithm chooses a smaller subspace and "zooms" closer to the minimum state.

Random number generation from this distribution is done by calculating the sum of the states Z and creating a probability distribution function

$$F_j = \sum_{i=1}^{j} P(X_i) \tag{3}$$

Now as $F_j \in [0, 1]$ when $j \in [0, N_{pop}]$, by creating a random number $u \in [0, 1]$ the values of F_j can be found. Comparing this value to the sum on the right side of the equation gives a random number $w = \dfrac{j}{N_{pop}} \in [0, 1]$ from the distribution.

The energy function, that measures the quality of states, was the sum of squared distances

$$L^2 = \sum_{n=1}^{N} \left[(x_{R(i+1)} - x_{R(i)})^2 + (y_{R(i+1)} - y_{R(i)})^2 \right] \tag{4}$$

where $R(i)$ is a route function that defines an ordering between cities. It's periodic and thus $R(N + 1) = R(1)$.

The "temperature" of the system T was chosen to be the range of lengths in the system $T = \max L - \min L$. Thus the system cools down when the algorithm closes near the minimum.

The second problem, mating, was done in the following fashion:

1. City 1 was random: either City 1 from Route 1 or City 1 from Route 2.

2. Distance from previous city to City 2 in Route 1 was compared to distance from previous city to City 2 in Route 2. The city closest to the previous one was picked. If it was already in the route, the other

one was picked. If both were in route, a random non-picked city was picked.

3. Step 2 continues until all cities are picked.

At a defined rate there were mutations: swapping of adjecent cities. These were introduced in order to keep the population varying and vibrant.

3 The code

The code was written in Fortran and OpenMPI was used for parallelization. There were additional programs written with Python and its scientific library NumPy. These support programs were used to create the configuration files for the code and to visualize the results. Some Bash-scripts were also written for batch data generation. The main program is fairly straightforward.

Initialization

The program initializes and root process reads the configuration file. If no problems are encountered, root process will distribute these to other processes.

Population generation

Each process will initialize a random number generator with different seeds. After this they will generate population of routes.

Route creation

At each time step the code removes the lowest energy state. Two routes breed as defined in the previous chapter. After that a mutation is done with the probabilty given.

Migration

In a paralellized code it's important to utilize each population properly. Thus each population will send a number of their best routes to other tasks. This enables genetic information to pass between processes. Parallelization creates

isolated environments for each population, but this can possibly help with local minima as each population can find partial solutions to go around them.

Temperature calculation

It is not necessary to recalculate lengths at every timestep. Only when the temperature changes there is a need to recalculate probabilities. Using logarithm it is possible to map probabilities to energies and it is not necessary to recalculate lengths.

Printing and results

Root process will handle output and give state of the program to the user. At the end each process will give the root process its best route and its length.

Running the program

After compiling the program with included Makefile (paths might differ) running the program is fairly straightforward.

createcities.py can be used to create random cities into a square of size 1×1. Its syntax is:

```
python createcities.py {number of cities} {outpufile}
```

Other configuration values can be changed by modifying this file or the ensuing configuration .cities-file.

Trivial cities can be created with

```
python trivial.py {number of cities}
```

This will create a square and a circle with given number of cities.

Running the code can be done with:

```
mpirun -n {ntasks} traveller.exe {cityfile}
```

or by running any of the compare_{property}.sh that create 10 routes that differ in the value of {property}. compare_ntasks.sh has additional internal multipliers that can be tweaked. Each file uses cities from corresponding .cities-file.

Plotting a single route can be done with

```
python plotroute.py {cityfile} {routefile}
```

Plotting a comparison series can be done with

```
python analyze_{property}.py {cityfile} {folder that has the
    routes}
```

4 Accuracy and initial values

For trivial city placements and small city values results look very promising.

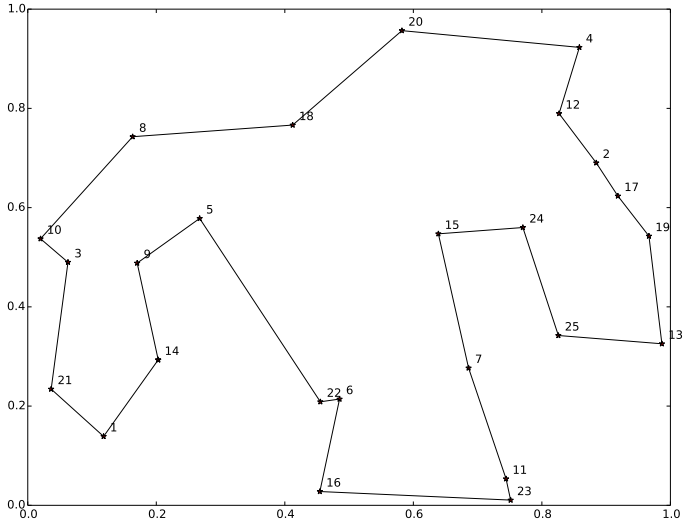

Figure 1: Example solution of a random placement of 25 cities

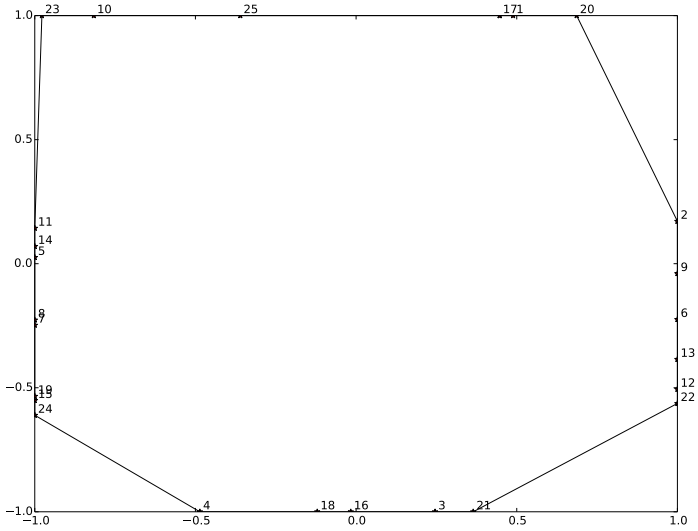

Figure 2: Example solution of a square with 25 cities

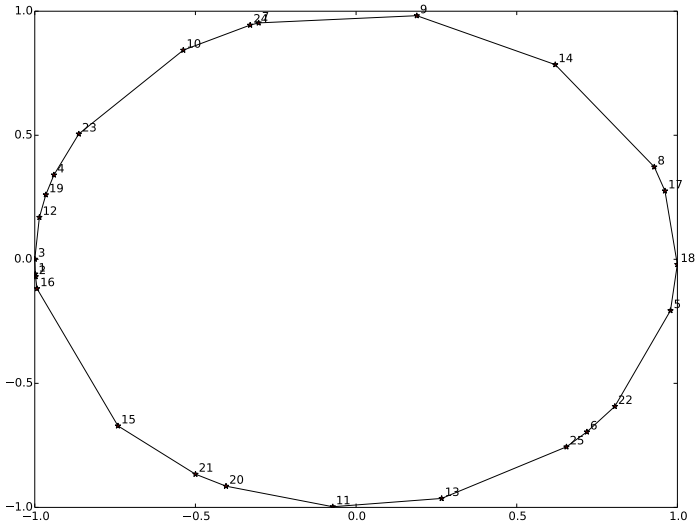

Figure 3: Example solution of a circle with 25 cities

Difficulties start to appear when the number of cities grow. Optimization

problems like the travelling salesman are tricky because the only measure that can be used to measure accuracy is how low the quantity that was to be minimized is at the end. Whether the answer is near the global minimum or at a local minimum, no-one can tell. Examples of this are shown below. As the number of cities rise, the solutions do not necessarily converge.

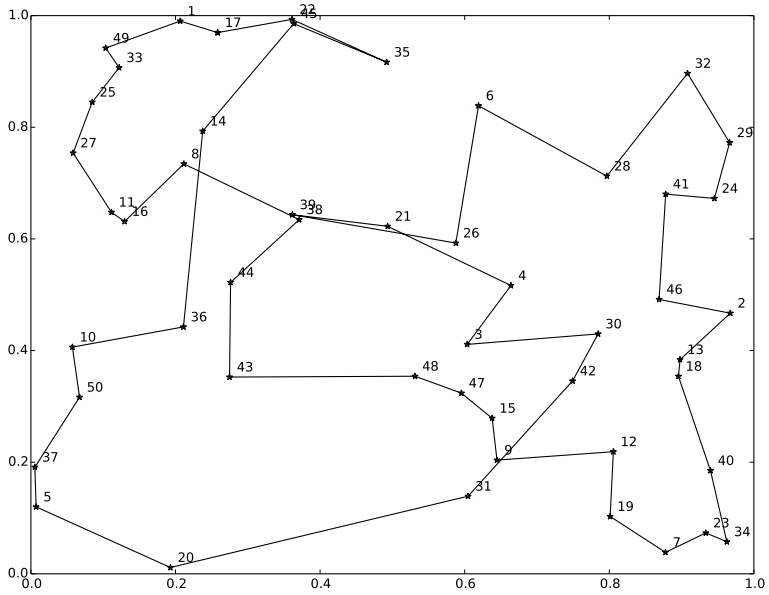

Figure 4: Example solution of a bigger problem

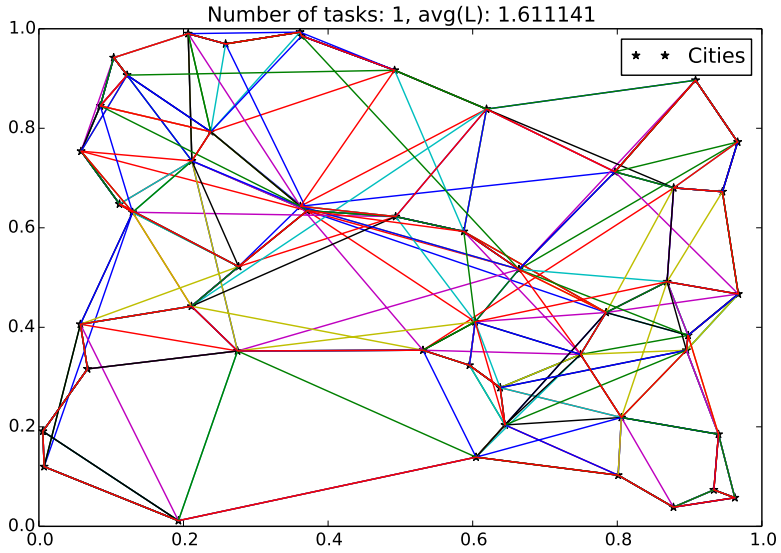

Figure 5: Solutions do not necessarily converge

Choosing the best initial values for a problem of a certain size can also be hard, but measuring the dependecy of accuracy versus different input parameters will help.

Measured parameters are:

1. Number of parallel tasks
2. Population size
3. Migration rate

Problem solved was for 50 randomly generated cities inside a square. A sample of 10 different seeds was used for each measurement point.

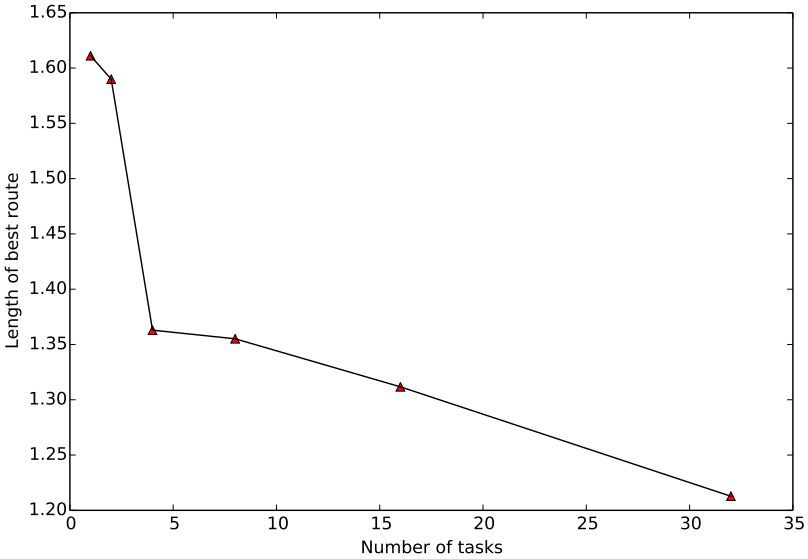

Figure 6: Number of tasks vs. accuracy

It is clear that the solution will benefit from the number of tasks. As the number of tasks grows, so does the overall population and overall timesteps of the program. But it's not certain that the increase in accuracy would be reached by a single task with a larger population and runtime. Genetic algorithm can possibly benefit from isolated populations that will optimize themselves. In addition the runtime for multiple tasks isn't much larger for multicore systems when each tasks gets a dedicated processor.

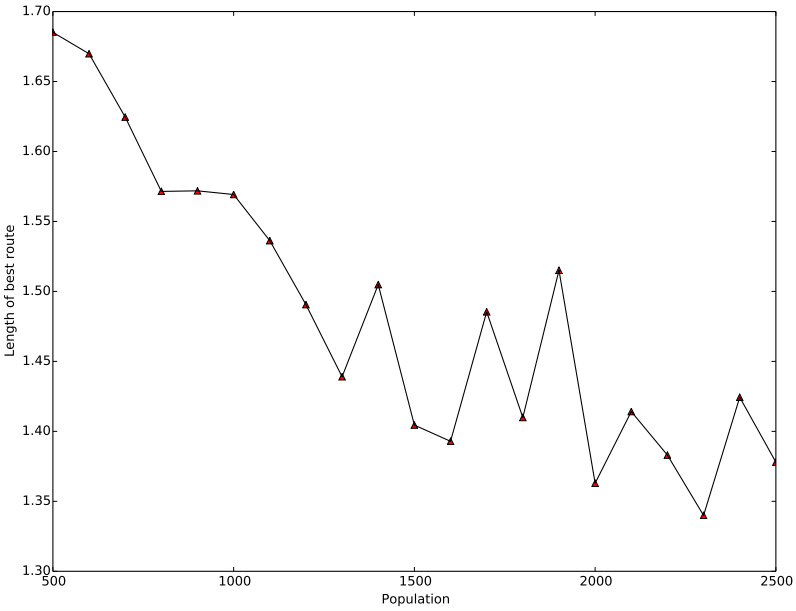

Figure 7: Population vs. accuracy

Increasing the population will also result in increased accuracy. There is a caveat, however, as larger populations need larger runtimes in order to converge. Without larger runtime, the larger population will not give benefits. Runtime increases linearly when population rises. This behaviour is plotted in the following picture.

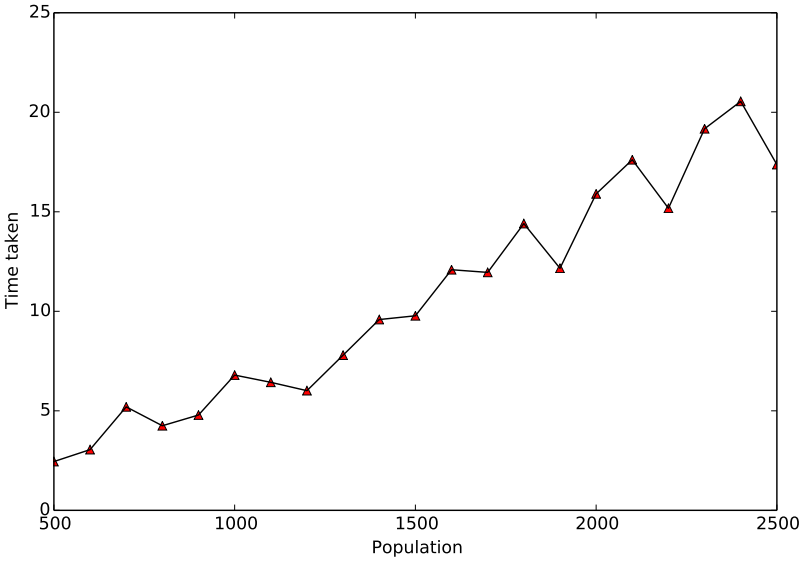

Figure 8: Population vs. time taken

Interestingly migration rate does not have a that much of an effect to the accuracy. The measurement was done with quite a high population number so the effect might be smaller than it would be with smaller population.

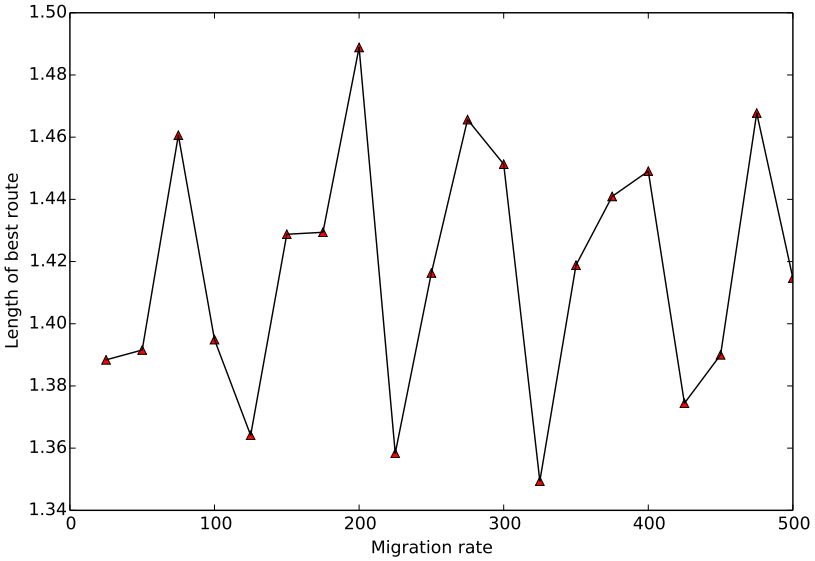

Figure 9: Migration rate vs. accuracy

5 Parallelization efficiency

The following figure shows how the accuracy of the code improves when number of tasks increases in a situation where overall runtime is kept constant by lowering the number of steps. Running a single task for a longer time will not give as good of a result as running multiple populations for smaller time. This is because the population saturates and reaches a local minimum.

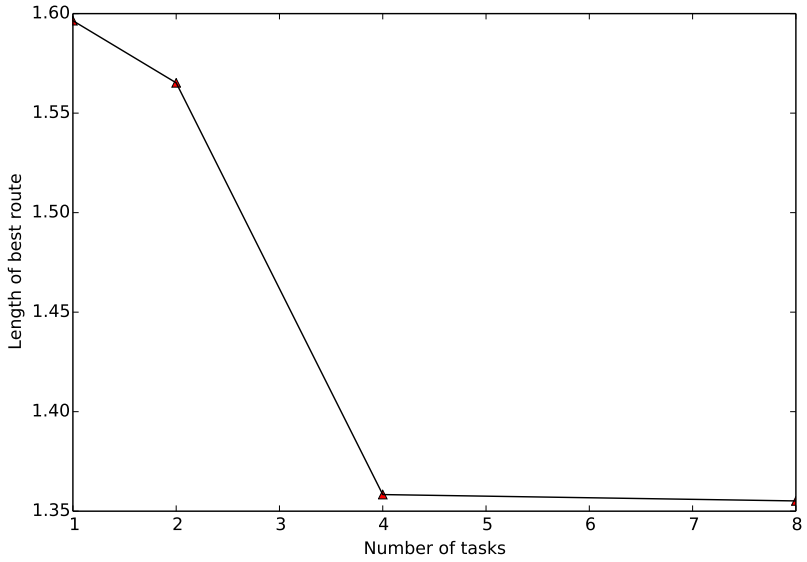

Figure 10: Number of tasks vs. accuracy (same real runtime, number of steps differ)

Next figures will dispel the idea that this could be fully countered by increasing population size. Runtime of the code is heavily depedendent on the population size and thus a single task with a large population will not achieve the same efficiency or accuracy as the same population distributed among many tasks.

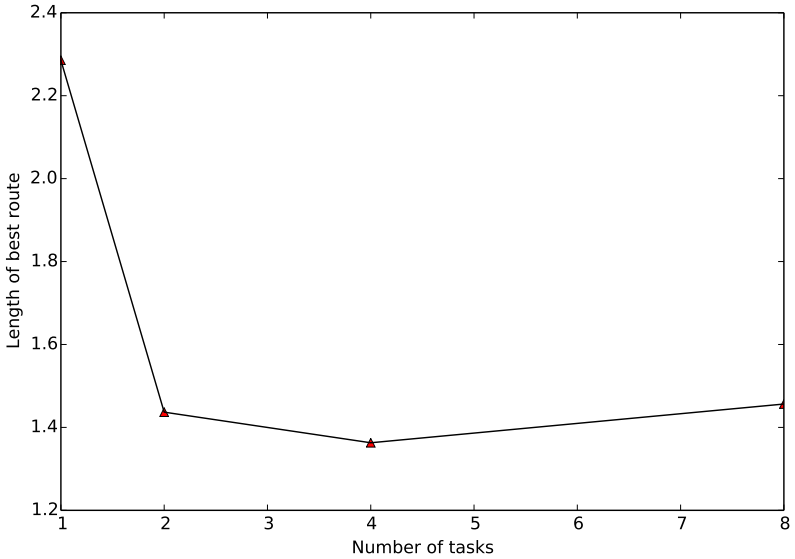

Figure 11: Number of tasks vs. accuracy (same number of steps, same overall population)

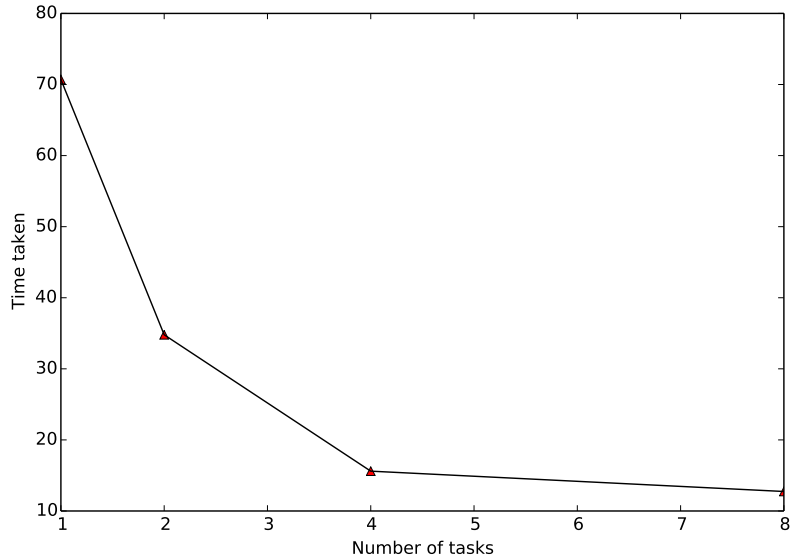

Figure 12: Number of tasks vs. used time (same number of steps, same overall population)

When using multiple tasks, the amount communication between them should be kept at the minimum. Following picture shows the running time plotted as a function of migration rate. With too low of a rate, there will be a significant slowdown.

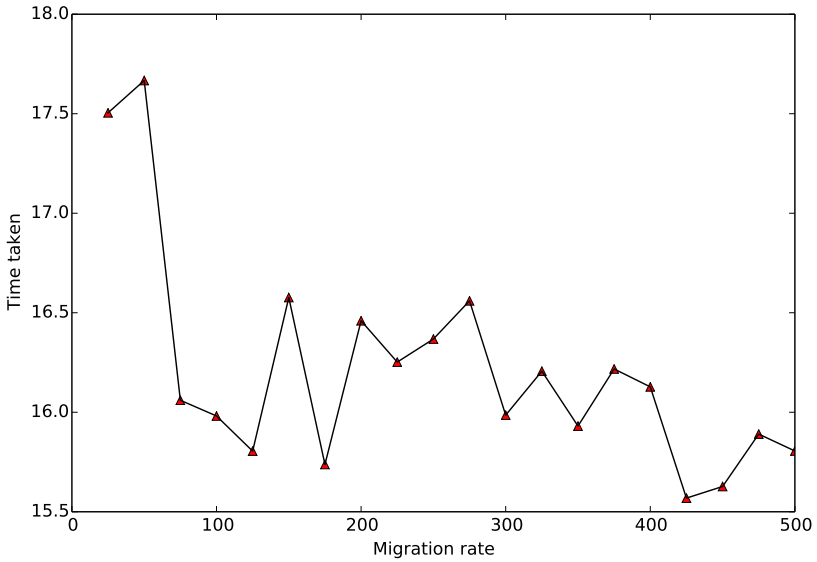

Figure 13: Migration rate vs. used time (ntasks = 4)

6 Conclusions

In this report I have presented an implementation of a genetic algorithm for travelling salesman-optimization problem. The results show that it can efficiently find solutions for problems of considerable size. Its accuracy can be improved by increasing parameter scales: population size, runtime and number of workers. In a sense, the program *needs* to be run in parallel to be significant. Thus I consider the parallelization successful.