CMPSC240A – FINAL PROJECT

A Parallel Genetic Algorithm for the Travelling Salesperson Problem

MARCH 20, 2014

BY JAKOB OFTEBRO, EIRIK AASVED HOLST AND ULRIK SAGEN

Table of Contents

Abstract

Since the beginning of the 20th century the Travelling-Salesman-Problem has been one of the most intensively studied problems in optimization. By many considered the most famous NP-Hard problem, it is often used as a benchmark for other optimization methods. The problem is not solvable in polynomial time, but there exists several methods of estimation that efficiently will approximate the problem within an error of 1%. This report describes a method utilizing a Genetic Algorithm to estimate the TSP problem on high-performance multi-core computers. It will go through our approach, problems that occurred as well as graphs and statistical analysis of the algorithm in its complete state.

Our approach to the problem

As we ventured into the task of estimating the Travelling Salesman Problem we were all fairly novice when it came to the implementation and execution of estimation algorithms. The course CS165A had recently introduced us to several Artificially Intelligent algorithms. One of these was the search heuristic that mimics the process of natural selection; The Genetic Algorithm. We had a fairly decent grasp on how we wanted to test and rate our algorithm, but when it came to the modeling of the algorithm we knew little.

We knew we wanted to use a Genetic Algorithm to approach the problem, but beyond that we were unsure. To gain further insight we decided to look for other solutions on the web. Our first plan was to use a sequential code by Marek Obitko. The code in question was incredibly efficient, but after working on deciphering the methods used and all the different heuristics we realized that it was a bit too advanced for our liking, and that we'd rather build our own code from scratch than to try and parallelize Obitkos code.

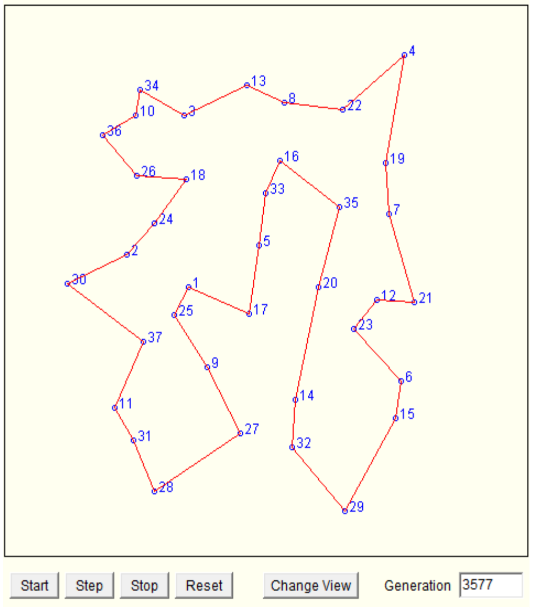


Figure 1: Screenshot of Marek Obitkos genetic TSP solver

When it came to programming language we wanted to use Java, as that was the language in which we had most experience. While searching the web for different implementations of parallel extensions in Java we found the open-source library MPJ-Express. This is a library built on the MPI libraries in C, and it is distributed under an MIT license. It took us a while to make it work, but after finally getting our "Hello Back" from both our computer and the Triton machine we were ready to begin writing our code.

We started off by creating an environment for the cities and the euclidean distances between them. We created the classes Country and City containing all necessary methods for this, these classes are more thoroughly described in our code-description later. They contained methods to create a countryMatrix with size NxN and a certain chance of a city on each spot, and a distanceMatrix that contained the distance between every city. After a swift testing phase we established that the first classes were working and thus we could start on the sequential methods to actually solve the problem.

We decided to write our sequential code using a simple heuristic fitness-function that basically counted the fitness of a path as the sum of the Euclidean distance between the cities in a given order.

In the end we created a paint class to help us monitor how our code actually worked, to see if it followed paths legal inside the world of the TSP, and later to see how it approached problems that we knew the solution to. During parallelization we had to make several alterations to the original code. We also implemented methods for the code to support existing TSP problems, an important feature when it comes to testing as described later. The parallelization of this code is thoroughly described in our code-description paragraph.

Problems we encountered

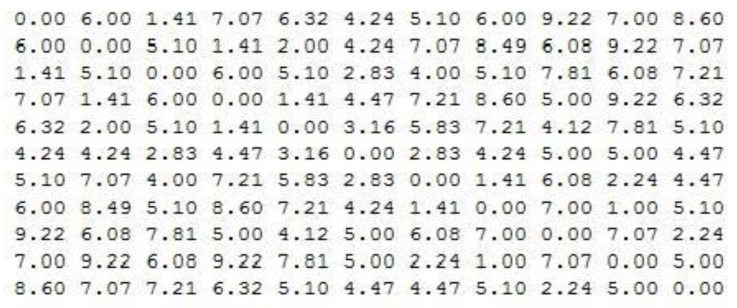
It did not take long before we encountered minor difficulties. Our first apparent problem was to find a way to check the validity of our solution. Our answer to this was to find already solved problems, and compare results from our code to them. We did this by hardcoding the solved problems into our code, and compared the distance our code generated, to the optimal distance. Because our code is not guaranteed to find the optimal solution, we decided to put a threshold at 3%. If the generated fitness is within 3% of the optimal solution, the program ends.

Another problem we encountered was how to send our self-made objects from one processor to another. This turned out to be an impossible task, as MPJ-Express does not support sending of objects at all. We solved this by only sending the important parts of the objects as arrays of primitive variable types. This has both negative and positive effects. The negative being that it takes time to extract the information from our objects, and putting them into arrays. The positive effect is that less information is sent between the processors, speeding up the sending and receiving of information.

We tried to implement a cross-over mutation(Combining two chromosomes by taking the first half of the first chromosome and the second half of the second chromosome, and visa versa) in the evolving phase. Unfortunately we found this to be very challenging and decided against it. When implementing a cross-over mutation in TSP you face one major problem. When crossing two chromosomes you need the part of the chromosome being mutated together to contain exactly the same cities. If this is not true you will receive chromosomes that both visit the same city twice, and that forget to visit cities, something which obviously is unacceptable for a TSP solution. We found that the logic and computation-time needed for such a feature to be implemented would outweigh the possible converging-speedup that it would yield.

How the code works

Our code was built from the bottom up starting with the basic environment for the TSP problem. It contains eight classes (see fig **3**) that help with the logic and genetic-methods, as well as a main loop that takes care of the parallelism and problem logistics.



```
0.00 6.00 1.41 7.07 6.32 4.24 5.10 6.00 9.22 7.00 8.60
6.00 0.00 5.10 1.41 2.00 4.24 7.07 8.49 6.08 9.22 7.07
1.41 5.10 0.00 6.00 5.10 2.83 4.00 5.10 7.81 6.08 7.21
7.07 1.41 6.00 0.00 1.41 4.47 7.21 8.60 5.00 9.22 6.32
6.32 2.00 5.10 1.41 0.00 3.16 5.83 7.21 4.12 7.81 5.10
4.24 4.24 2.83 4.47 3.16 0.00 2.83 4.24 5.00 5.00 4.47
5.10 7.07 4.00 7.21 5.83 2.83 0.00 1.41 6.08 2.24 4.47
6.00 8.49 5.10 8.60 7.21 4.24 1.41 0.00 7.00 1.00 5.10
9.22 6.08 7.81 5.00 4.12 5.00 6.08 7.00 0.00 7.07 2.24
7.00 9.22 6.08 9.22 7.81 5.00 2.24 1.00 7.07 0.00 5.00
8.60 7.07 7.21 6.32 5.10 4.47 4.47 5.10 2.24 5.00 0.00
```

Figure 2: Distance matrix. Shows all the distances of a country with 11 cities. Position i,j holds the distance from city i to city j, and thus it is a mirrored matrix with a zero diagonal.

Our method starts by creating a Country object (see fig **3** (B)). This class takes an input argument that decides if it will use a TSP instance from a text file or if it will generate a random problem with its makeCities() method. The distances are then calculated and stored in the 2D-array distanceMatrix (see fig **2**) for easy use during the fitness-calculation.

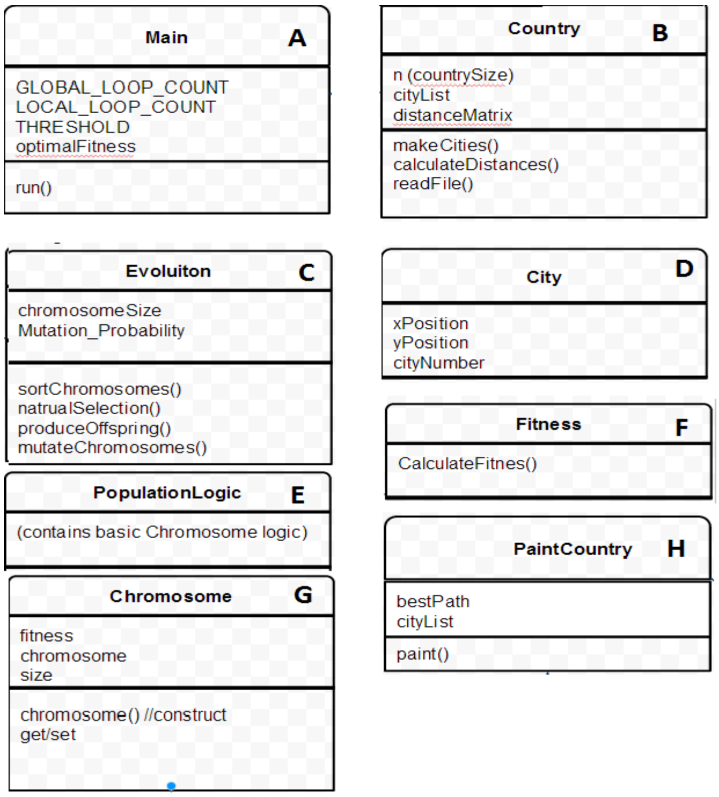


Figure 3: The eight classes and the most important methods/fields in each class. (A) The Main class containing the run() method, as well as the fields deciding the different loop counts. TRESHOLD decides how close to optimal solution the code will terminate. (B) CountryClass, (C) Evolution class, (D) City class, (E) PopulationLogic class, (F) Fitness class, (G) Chromosome class. (H) PaintCountry class takes an array of unuque numbers from 1 to n, paints the cities with their corresponding coordinates and the path among them.

When the program starts, an initial population of randomly generated chromosomes is created and distributed to all processors, so that each processor has its own subpopulation. Every processor will then evolve and mutate their subpopulation a set amount of times (LOCAL_LOOP_COUNT). Then each processor sends its subpopulation to the root processor. The root will mix all the chromosomes from the subpopulations, and send them back out to the other processors (GLOBAL_LOOP_COUNT).

The chromosomes are randomly distributed on each processor. This ensures a healthy mix of chromosomes, and makes the probability of being stuck (see fig **4**) lower.

Every chromosome is a list of numbers, each number represents a city/node, and the order of numbers represents when each city/node is visited. Numbers can only appear once in every chromosome, and all numbers (from 1 to the number of cities) must be included.

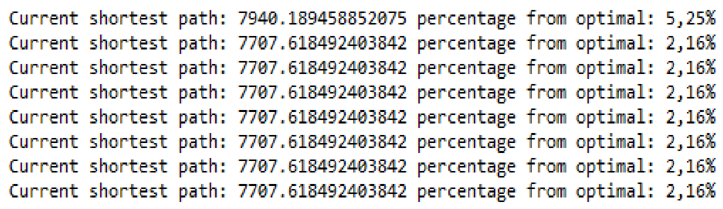

```
Current shortest path: 7940.189458852075 percentage from optimal: 5,25%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
Current shortest path: 7707.618492403842 percentage from optimal: 2,16%
```

Figure 4: There is a possibility of getting stuck for a while at the same solution. The code will eventually find a better path, but that depends on chance (the new chromosomes in the population)

The evolution is done by sorting all chromosomes based on their fitness level, using the calculateFitness() method. The best ¼ of the chromosomes is selected to carry on the next generation. If a chromosome is selected, it will be copied 4 times and every copy is given a random mutation. For every new generation, ¼ of the chromosomes will also be new, randomly generated chromosomes. The chromosomes is now sent through something called “random only improving mutation”. This means that every chromosome is first given a random point-mutation(Swap two numbers), and then we test every single possible point-mutation until we find one that improves its fitness level. This might seem inefficient, but we found that this lets us find better solutions after fewer generations.

Graphical preview of evolution

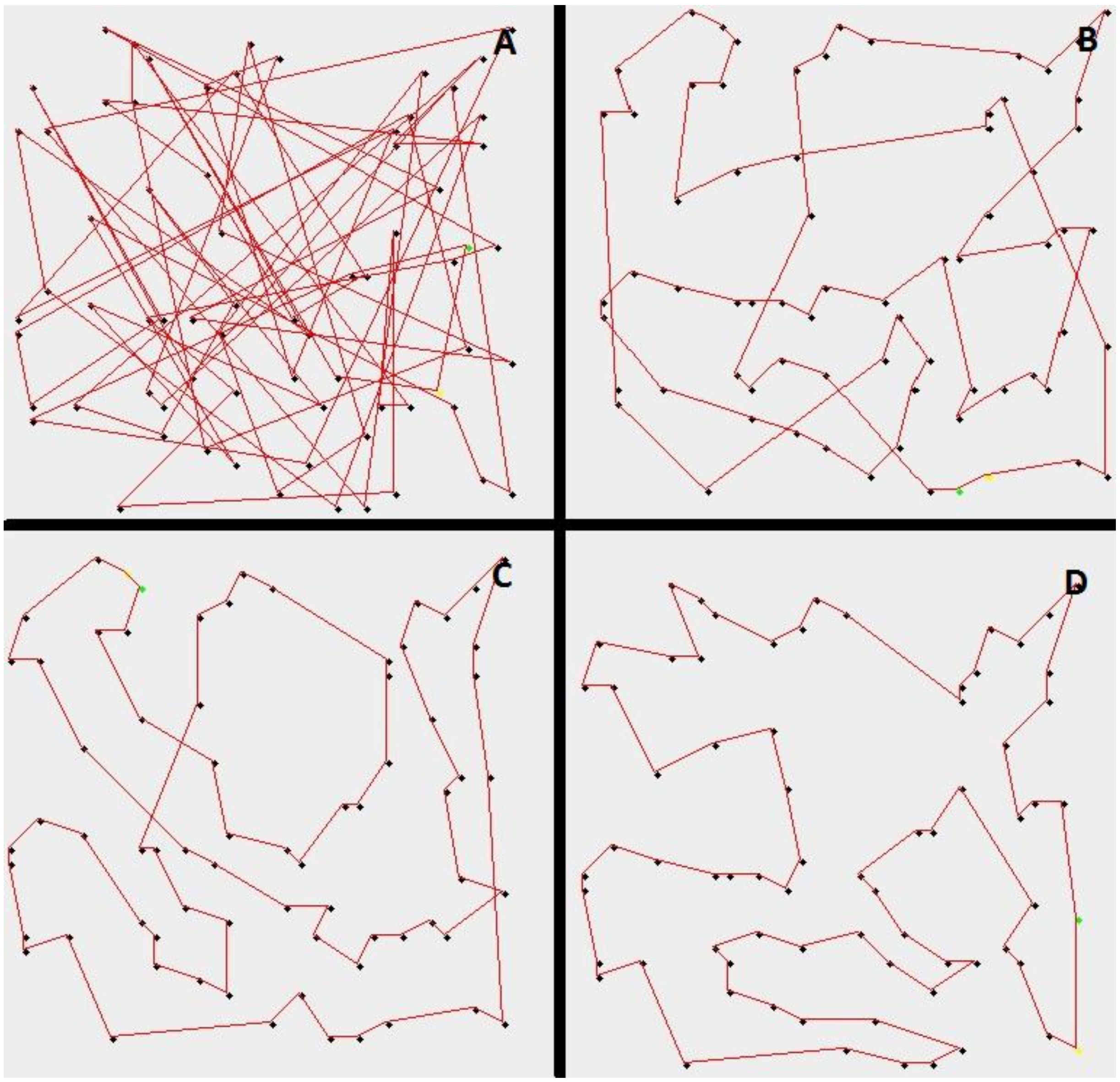


Figure 5: Graphical evolution on a randomized Country with 72 cities. (A) Shows how one of the random chromosomes looked before any evolution or fitness has been calculated. B, C and D shows the best chromosome in the evolution at a certain time.

Performance tuning

A challenge with our code was to figure out what combination of different values produced the best performance. The numbers in question is LOCAL_LOOP_COUNT, MUTATION_PROBABILITY and subPopulationSize.

The mutation probability was the easiest to set. Our code has a good mutation method, as all mutations will make the chromosome better. Because of this, every chromosome is better of being mutated than not, thus we set the mutation probability to 100%.

Inner loop count was a bit harder to set a specific value for. We found that for different types of tests and problems, different values yielded different results. On small problems, a lower local loop count (<10) was more efficient because an answer was found within a small amount of iterations. On bigger problems, more inner loops gave better improvement from iteration to iteration and found optimal solutions faster. A problem with many inner loop iterations is that even though the optimal solution has been found, we will not know before all iterations is done. This means that the lowest running time we will get is the time it takes for the code to finish all inner loop iterations. The probability for this happening on big problems is low, and it is not a big issue.

The population size and sub population size was not that hard to set. We tested different values, all exponents of 2. After a while, we found that this number was best to make dependent on the number of processors we ran on. A sub population size of 128 gave good results all over, on both small and large problems. The population size is thus set to 128 * "number of processors".

Testing of the code

As stated earlier, the testing of the code was a slight concern for us as we began working on the project. We knew that we wanted to test how our code worked on existing problems, but the fact that we had no way other than brute force to find the real optimal path gave us some trouble. Luckily, our previous professor Teofilo Gonzales showed us an online TSP library with over 80 solved TSP instances, as well as their solution.

This library became a core part of our testing-procedures as we finally had a way to confirm whether our program found solutions close to or equal to the optimal path. We used these libraries to see how many iterations, or how much time, was needed to reach a certain THRESHOLD value, and thus we had a valid benchmark for our algorithms performance.

Strong scaling analysis

The queue for requesting nodes on Triton was extremely long when we were trying to test it there, so the code was primarily tested on our multi core laptops. We ran the problem on a known problem and measured the time it took to reach a threshold of 7% from the optimal solution. We did this on one, two, four and eight processors to get an idea of the parallel efficiency and scaling of the problem. We ran the code 16 times on each number of processors and calculated the average: Here are the plots:

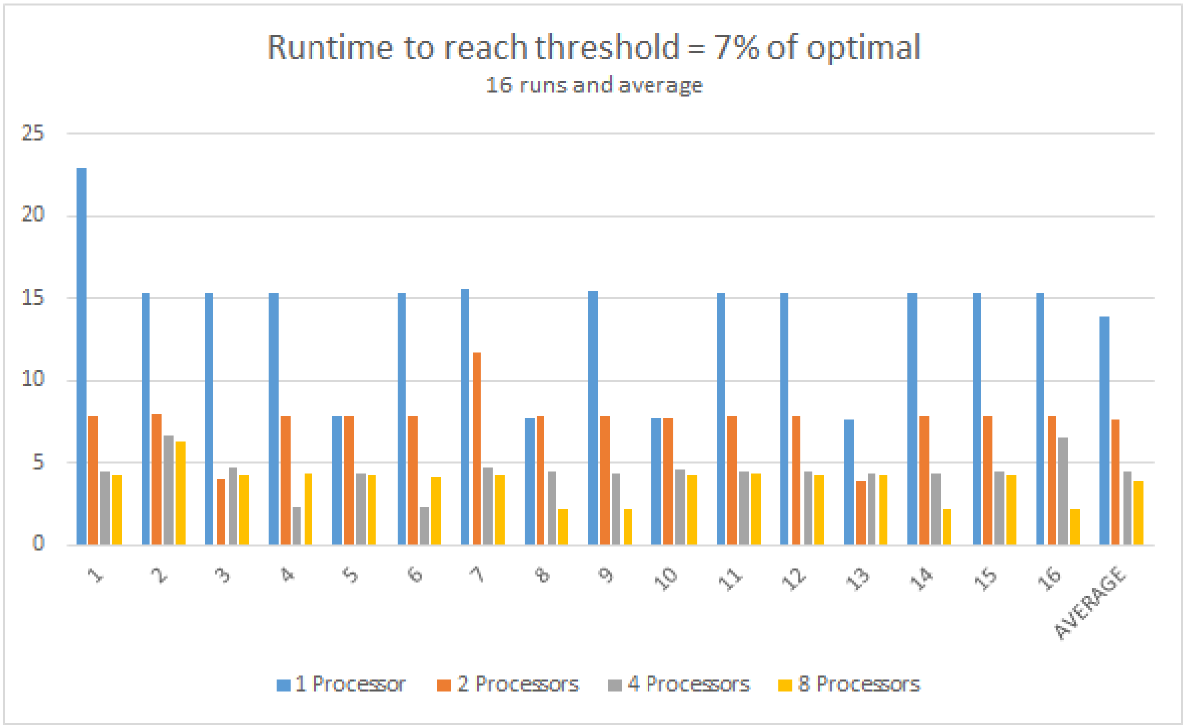


Figure 6: Runtime in seconds to reach a threshold of 7% from the optimal solution. The code ran 16 times on one, two, four and eight processors

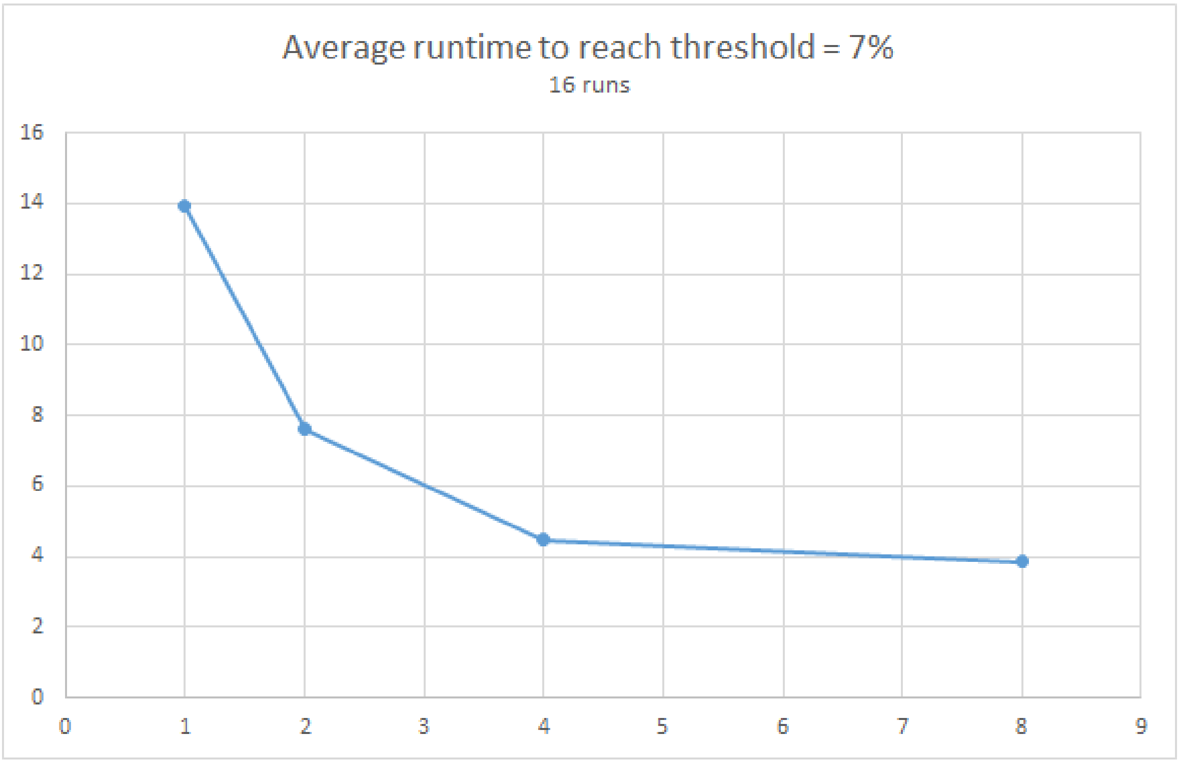


Figure 7: Average runtime in seconds to reach a threshold of 7% from the optimal solution on one, two, four and eight processors

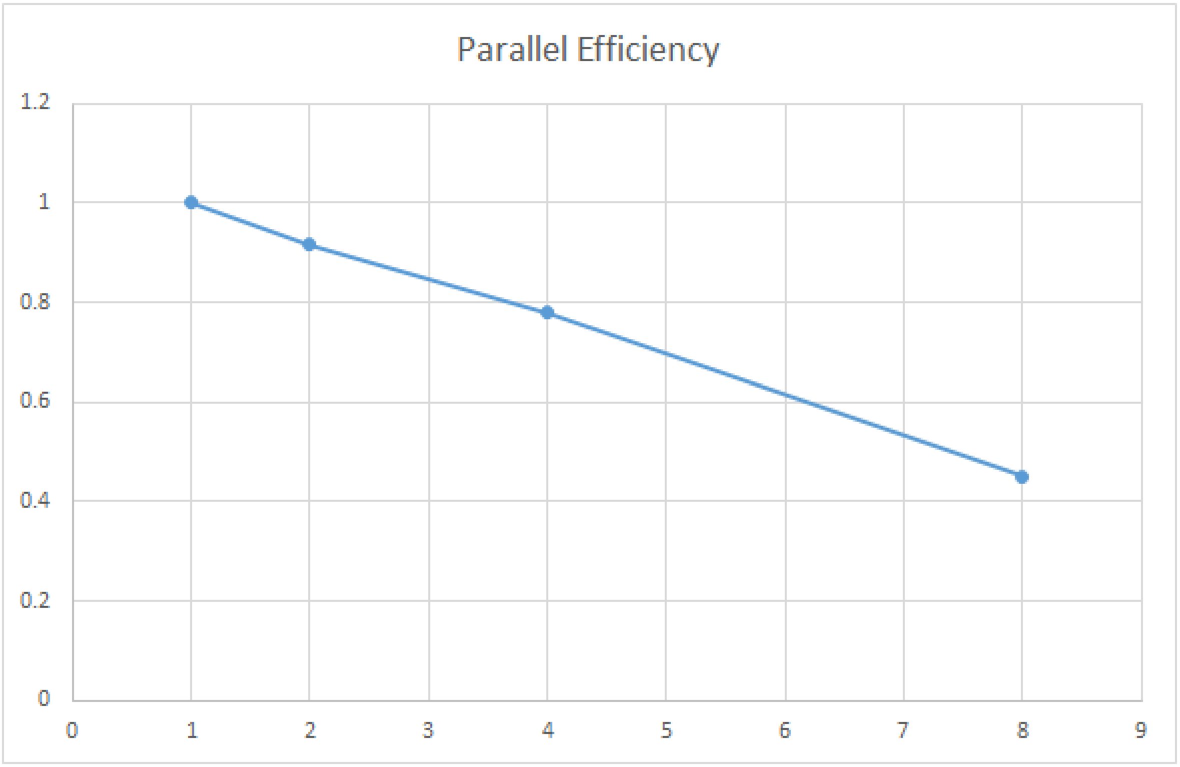


Figure 8: Parallel Efficinecy on a known problem of n = 52 on one, two, four and eight processors

The code was also ran for a fixed time of t = 16 seconds and then returned how close it was to the known optimal solution. Here are the plots:

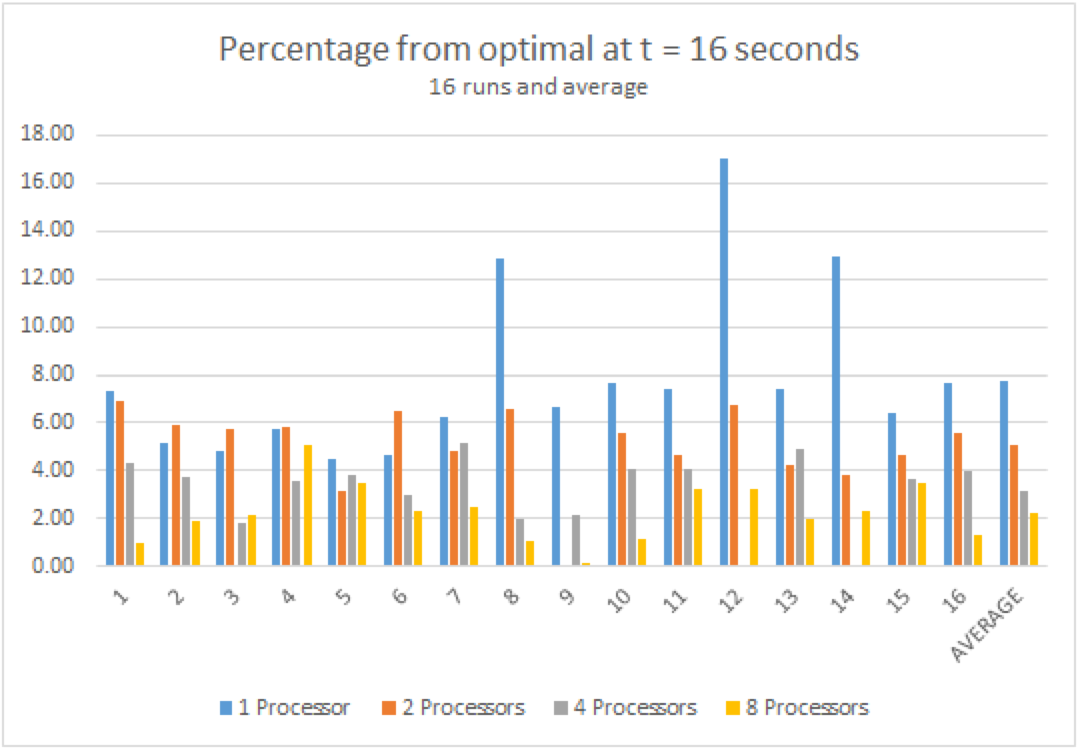


Figure 9: Percentage from optimal after a fixed runtime of t = 16 seconds. The code ran on one, two, four and eight processors

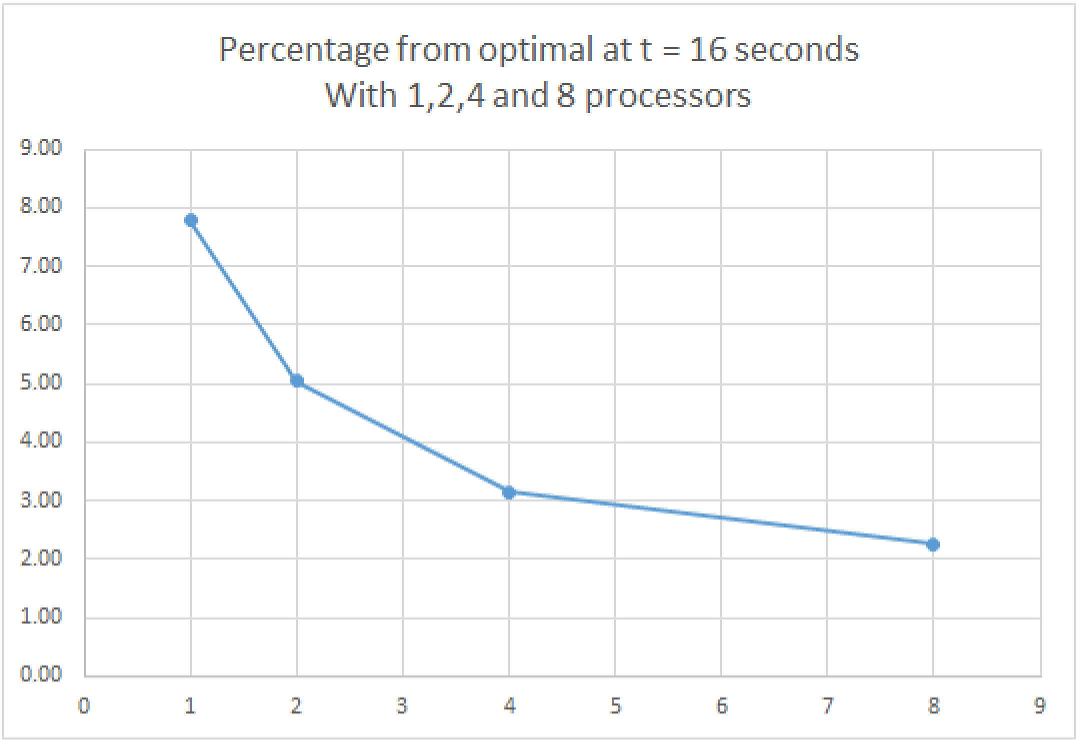


Figure 10: Average percentage from optimal after t = 16 seconds on one, two, four and eight processors

After hours of waiting for nodes on Triton we finally got some assigned, and started testing there. We ran the problem of 64 cities ($possible\ solutions \approx 1.27 * 10^{89}$) with fixed number of iterations and plotted runtime and parallel efficiency for 1, 2, 4, 8, 16, 32, 64 and 128 processors. As shown, there is an almost linear scale up when the code runs on just one node (up to 16 processors), but when more nodes are used, the amount of message passing between the processors is so vast that the scale up is roughly constant, i.e. doubling the number of processors don't change the runtime. As mentioned, this is not because of a low potential parallelism, but because there is just too much communication. In the outer loop, all processors send all their chromosomes to the root processor and gets a random mixed subpopulation in return. This part takes up the majority of runtime when the number of processor gets big (>32). To have a large number of inner loops and a rather small number of outer loops may seem like a solution to this, but it tend to get stuck at a local maximum more often when that is the case. The subpopulations needs a mix every now and then to boost the evolution.

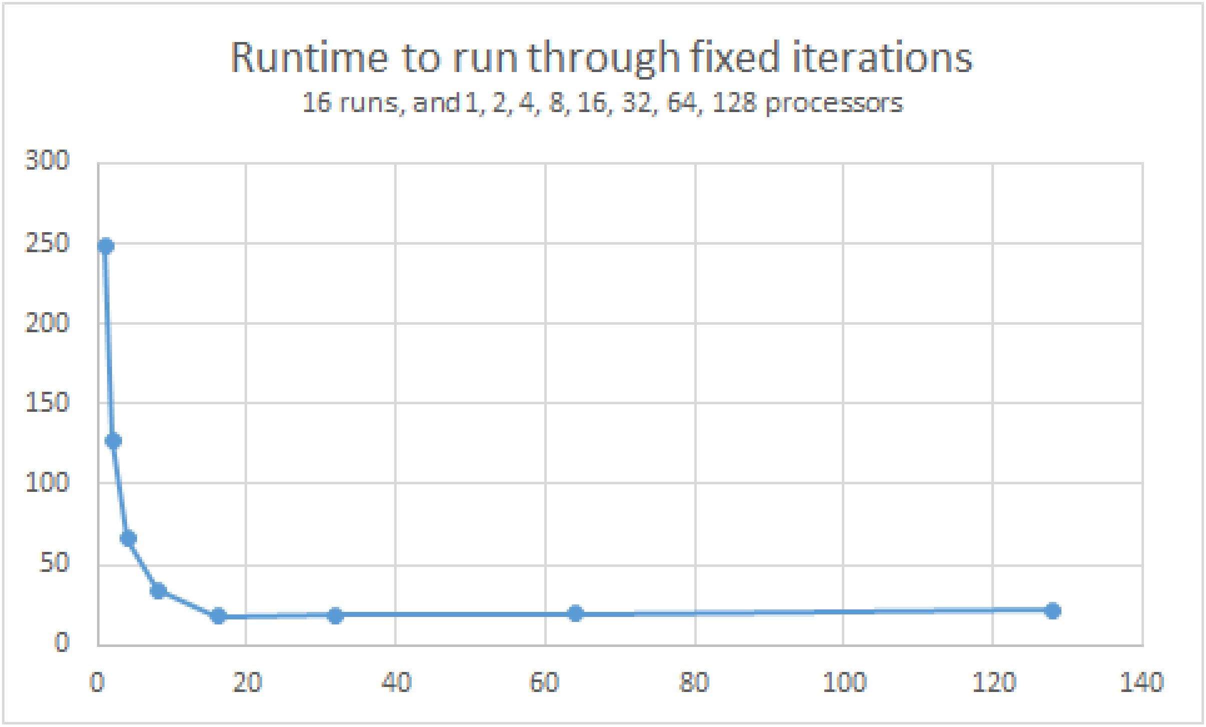


Figure 11: Runetime to finish a fixed number of nested loops on 1, 2, 4, 8, 16, 32, 64 and 128 processors

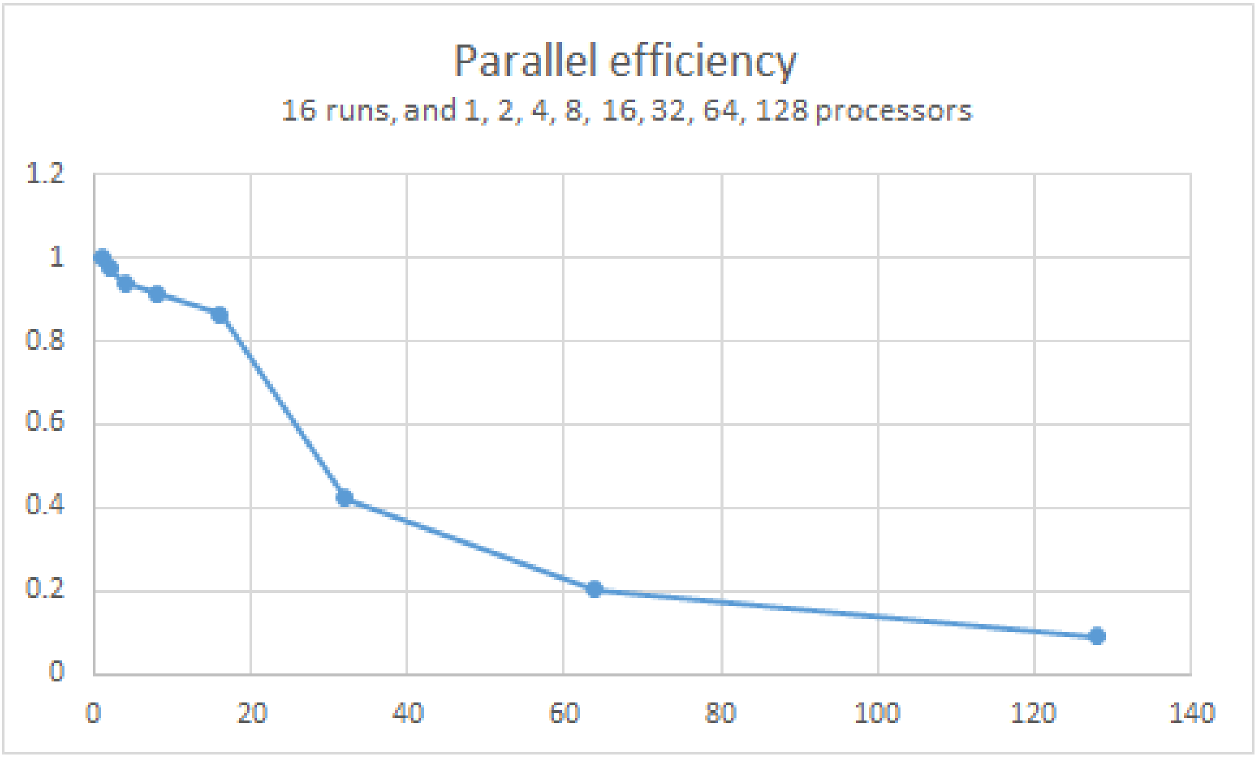


Figure 12: Parallel efficiency for a fixed number of nested loops. The drop after 16 processors is when the inifiband is utilized. This causes a major time consumer on data exchange and message passing

Weak scaling analysis

The algorithm runs eight outer loops and eight inner loops and returns the time it takes to complete. The problem size varies almost proportional to the square root of the number of processors.

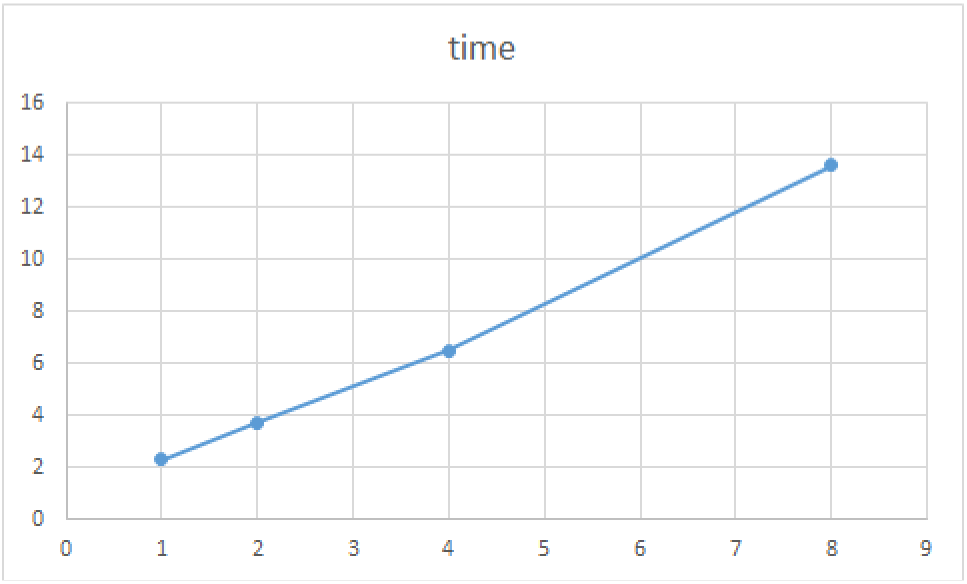


Figure 13: Weak scaling on a fixed number of nested loops. The problem size increases proportionally to the square root of the number of processors

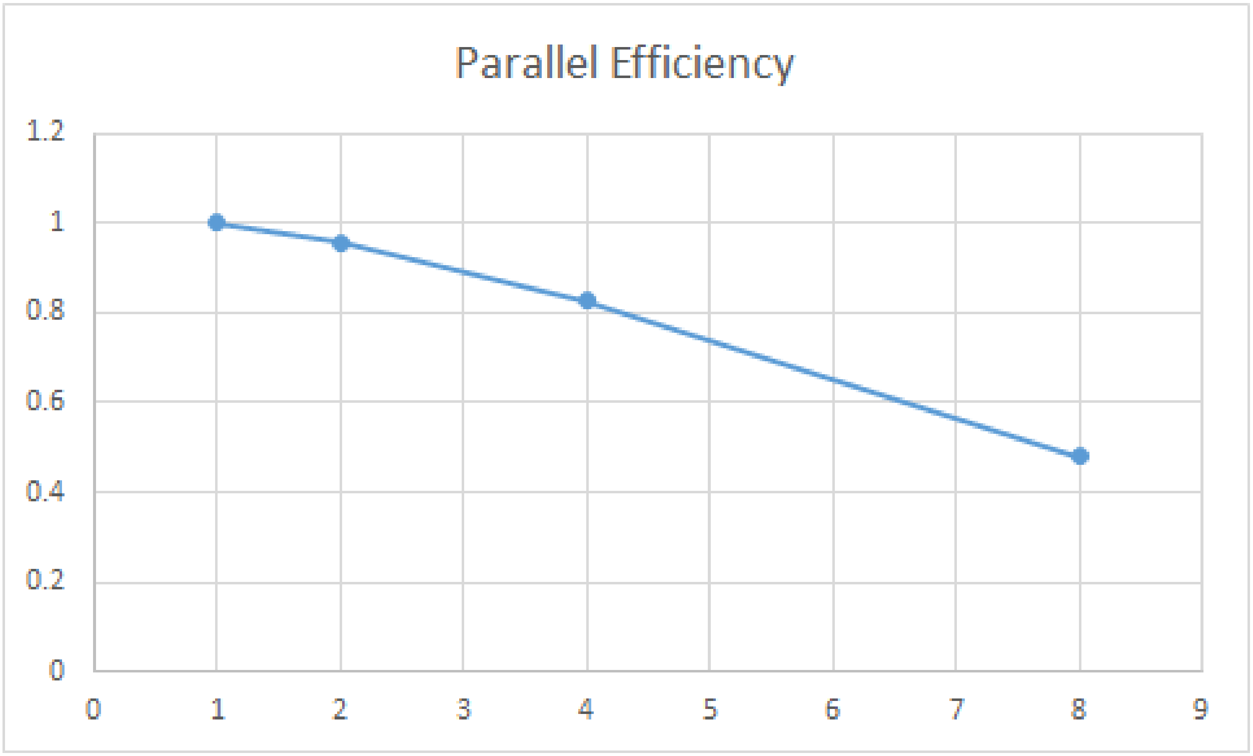


Figure 14: Parallel efficiency in the form of weak scaling

Improvements:

We would like to make our code prevent itself from being stuck. This can be done by making the code “reset”, or replace all chromosomes with new ones, when it has not improved over a set amount of iterations. When this is done. The best chromosomes will be stored, to prevent the code to potentially delete the best solution we can find. If we can implement this, it means that our code will, eventually, find the optimal solution every time.

Concluding thoughts

After the completion of this project, we are left with a lot of knowledge about how performance tuning, parallel computing and genetic algorithms work. When it comes to the genetic algorithm we found that this seems like a really good way of finding good or optimal paths in the TPS problem. Our code found solutions much faster than we anticipated. But in the future we would look at methods to decrease the computation needed, to reduce the running time on larger problems.

After looking over our test results it became apparent that the time spent on sending and receiving information became too large as the number of processors and nodes increased. This threw light on a big problem in our code; We send a lot of information between processors. After more testing, we found a solution at the cost of performance: increase the local loop count. This was a compromise we didn’t like because it meant that our code had a higher risk of getting stuck. We will look into better ways of handling the amount of communication between processors, but this turns out to be a big setback when it comes to solving TSP with parallelized genetic algorithms, because the chromosomes must include an amount of information that is proportional to the problem size.