UNIVERSITATEA ″ALEXANDRU-IOAN CUZA″ DIN IAȘI

FACULTATEA DE INFORMATICĂ

BACHELOR'S THESIS

Parallel Genetic Algorithm for Travelling salesman problem

Author

Andrei-Alexandru Paunescu

Session: june/july, 2022

Thesis advisor

Lect. Dr. Vidrașcu Cristian

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Parallel Genetic Algorithm for Travelling salesman problem

Andrei-Alexandru Paunescu

Session: june/july, 2022

Thesis advisor

Lect. Dr. Vidrașcu Cristian

Contents

Introduction

Context

In the past, it was believed that the best way to improve computer performance was by creating more efficient and faster processors. Over time, this idea has been changed by the arrival of parallel processing.

Nowadays, parallel computing is becoming very popular, with processors having more and more cores and threads. Among the driving forces that have facilitated this rise in popularity is the rapid improvement in the availability of commodity high performance components for computers and networks.

Parallel algorithms are usually easily scalable, and that has led to their rapid adoption by big companies and projects, which has fueled their increase in popularity even more. Scalable computing clusters have become the standard platform for high-performance and large scale computing. The main attractiveness of these kinds of systems is that they are built using low-cost, affordable hardware and can be tuned to available budget and computational needs.

A genetic algorithm is a heuristic-based method used for finding optimized solutions for search problems based on natural selection and evolutionary biology.

In this thesis, I aim to implement and optimize a parallel genetic algorithm for the traveling salesman problem running on a cluster of computers and analyze the results compared to a sequential algorithm.

Motivation

Because of the ever increasing popularity of parallel computing and the big gains in both performance and efficiency, I feel like there should be a bigger focus on parallel algorithms during the first years of college. There are big advantages to be gained in speed, performance, and time by implementing parallel algorithms for your applications.

I chose to adapt a genetic algorithm to a parallel one because I find the idea of simulating the concept of natural selection fascinating. With this thesis, I aim to improve and optimize a genetic algorithm by implementing a parallel processing algorithm on it and documenting my results and experiences.

Contributions

The project consists of two applications: a server and a client. The server is responsible for the communication between all the clients and provides a graphical interface from which to send commands to all the clients and see the progress of the algorithm. The client connects to the server and receives commands for work and is able to synchronize with all the other clients through the server.
The actions that are available through the server:

- You can visualize the solution on a graph.
- Start\Stop the work on all clients connected.
- Switch the instance of the problem to be solved by the clients.
- Export the solution.

The actions that are available through the client:

- You can select the number of processes to be started.
- Connect and receive work from the server.
- Updating the population of the genetic algorithm from the server.

All of the functionalities from above are used to implement a parallel genetic algorithm for the traveling salesman problem.

Chapter 1

Traveling Salesman Problem

The travelling salesman problem (TSP) is the challenge of finding the shortest possible route that visits every city from a list of cities exactly once and returns to the begining. It is a well-known algorithmic problem in the fields of computer science and operations research.

TSP has gathered so much attention because it's very easy to describe yet very difficult to solve. It is an NP-hard problem in combinatorial optimization. The complexity of calculating the best route will increase when you add more destinations to the problem.

1.1 Input

The problem receives as input a list of cities and their coordinates. The input received by the algorithm will be as a file in the TSPLIB[1] format.

1.2 Output

The solution to the problem consists of an ordered list containing all the cities from the input that describes the shortest path found. The exported output will be as a file in the TSPLIB[1] format.

Chapter 2

Genetic Algorithm

A genetic algorithm is a heuristic-based method used for finding optimized solutions for search problems based on natural selection and evolutionary biology.
Here are some genetic algorithm terms that need to be defined in the context of a traveling salesman problem problem:

- **Gene**: a city represented by it's coordinates (x,y)
- **Individual** (also known as "Chromosome"): a route that visits every city only once.
- **Population**: a collection of individuals (a collection of routes)
- **Parents**: two routes that are combined to create a new route
- **Crossover**: a genetic operator used to combine the genetic information of two parents to generate a new offspring
- **Fitness**: a function that is used to evaluate how good each individual is (total distance of a route)
- **Mutation**: a way to introduce variation in our population by randomly altering the genes of an individual
- **Elitism**: a way to carry the best individuals into the next generation

2.1 Advantages and Disadvantages

Advantages of a genetic algorithm compared to conventional methods:

- Find a good quality solution in a short amount of time.
- A wide range of solutions.
- Parallelism, easily adaptable to various problems.
- Inherently parallel and easily distributed

Disadvantages:

- It might not find the most optimal solution to the defined problem in all cases.
- It cannot guarantee an optimal solution.
- Overfitting (all the individuals have the same genes).
- Optimization Time (It usually takes a while to reach the global optimum).
- It is hard to choose optimal parameters (generations, population size, mutation chance, etc).

2.2 Architecture

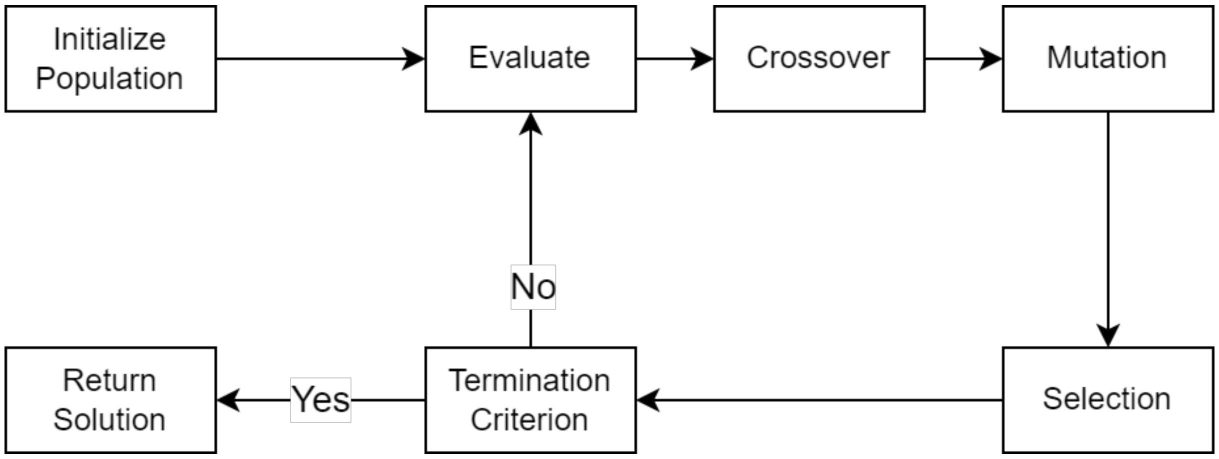


Figure 2.1: Genetic Algorithm architecture without parallelism

In Figure 2.1, we can see the general architecture of the genetic algorithm without parallelism. By doing all of these steps, we simulate natural selection and obtain new individuals each generation. Below, I will explain in detail what every section of the algorithm does.

2.2.1 Initialize Population

Population initialization is the first step in the genetic algorithm process. The initial population is created by generating N number of individuals (N being the size of the population). Each individual is created from a random permutation of genes.

2.2.2 Evaluate

In evaluate we apply the fitness function to the individuals to find out their score. In our case, the fitness calculates the distances between the cities of the individual and returns the sum of the distances.

2.2.3 Crossover

In crossover, we select multiple pairs of individuals with a crossover probability chance. Then we apply an algorithm to combine the genes of both parents to generate a new individual with traits from both parents.

Since we are working with permutations, we can't use the normal two point crossover operator because if we simply switch the genes between two parents, we will have duplicate cities.

To solve this problem, we implemented Partially Mapped Crossover. After choosing two random cut points on parents to build offspring, the portion between cut points, one parent's genes are mapped onto the other parent's genes, and the remaining information is exchanged.

Example:

$P_1 = (3\,4\,8 \,|\, 2\,7\,1 \,|\, 6\,5)$

$P_2 = (4\,2\,5 \,|\, 1\,6\,8 \,|\, 3\,7)$

Firstly, we switch the genes between the crossover points:

$C_1 = (X\ X\ X \,|\, 1\,6\,8 \,|\, X\ X)$

$C_2 = (X\ X\ X \,|\, 2\,7\,1 \,|\, X\ X)$

Then we can fill the genes for those who have no conflict:

$C_1 = (3\,4\,X \,|\, 1\,6\,8 \,|\, X\ 5)$

$C_2 = (4\,X\,5 \,|\, 2\,7\,1 \,|\, 3\,X)$

Then, the first X in the first child is 8, which comes from the first parent, but 8 is already in this child, so we check mapping $1 \leftrightarrow 8$ and see again 1 existing in this child. Again, we check mapping $2 \leftrightarrow 1$, so 2 occupies the first X. Similarly, the second X in the first child is 6, which comes from the first parent, but 6 exists in this child. We check mapping $7 \leftrightarrow 6$ as well, so 7 occupies the second. From these results, child 1 is:

$C_1 = (3\,4\,2 \,|\, 1\,6\,8 \,|\, 7\,5)$

We repeat the process for child 2 and obtain:

$C_2 = (4\,8\,5 \,|\, 2\,7\,1 \,|\, 3\,6)$

2.2.4 Mutation

In mutation, we select individuals with a mutation probability and perform random gene alterations on them. For the algorithm we used a hybridization of two operators: Partial Shuffle Mutation (PSM) and Reverse Sequence Mutation (RSM)[2].

```
input  : Individual $x = [x_1, x_2, ..., x_n]$and $P_m$ is Mutation probability
output: Individual $x = [x_1, x_2, ..., x_n]$
Choose two mutation points a and b such that $1 \leq a \leq b \leq n$;
while $a < b$ do
    Permute($x_a$, $x_b$);
    Choose $p$ a random number between 0 and 1;
    if $p < P_m$ then
        Choose $j$ a random number between 1 and $n$;
        Permute($x_a$, $x_j$);
    end
    $a = a + 1$;
    $b = b - 1$;
end
```

Algorithm 1: Hybridizing PSM and RSM Operator (HPRM)[2]

2.2.5 Selection

The selection stage consists of determining which individuals we keep for the next round of crossover. In our case, we implemented a tournament selection combined with an elitism selection.

In a tournament selection, we select n individuals for the tournament with a selection probability chance. Only the fittest candidate among the selected candidates is chosen and is passed on to the next generation. In this way, many such tournaments take place, and we have our final selection of candidates who move on to the next generation.

The elitism selection consists of selecting some of the best individuals from the previous generation and carrying them over to the next one.

2.2.6 Termination Criterion

The termination condition is usually when a certain number of generations is reached. In our case, because we implemented a parallel genetic algorithm that runs on a network, we keep the clients running until they receive the stop signal from the server or they reach the known optimal solution.

2.3 Parameters

Genetic algorithms include a number of parameters, in our case: population size, mutation probability, crossover probability, elite percentage, selection probability, and migration chance. These parameters usually need to be individually selected for each problem, but I will mention my observations about some of them.

You can choose your population size based on the number of genes in your individual. Usually two times the number of genes is a good start.

A low mutation probability will give better results quickly, but it may lead to premature convergence in a local optimum, while a high probability will lead to a slow convergence speed.

The crossover probability is pretty similar to the mutation probability, but I observed that usually higher values perform better.

For the elite percentage, you don't want values too big because in a few generations your entire population will be formed from the same individuals and the same applies to the migration chance. If you use a larger value, the algorithm will converge to a local optimum faster.

2.4 Parallelism

It is conceptually pretty straight-forward to adapt a genetic algorithm to a parallel processing one. Each process will operate on an individual subpopulation, performing the stages described above. Periodically, a new stage called migration will run that will share the best individuals with the rest of the processes.

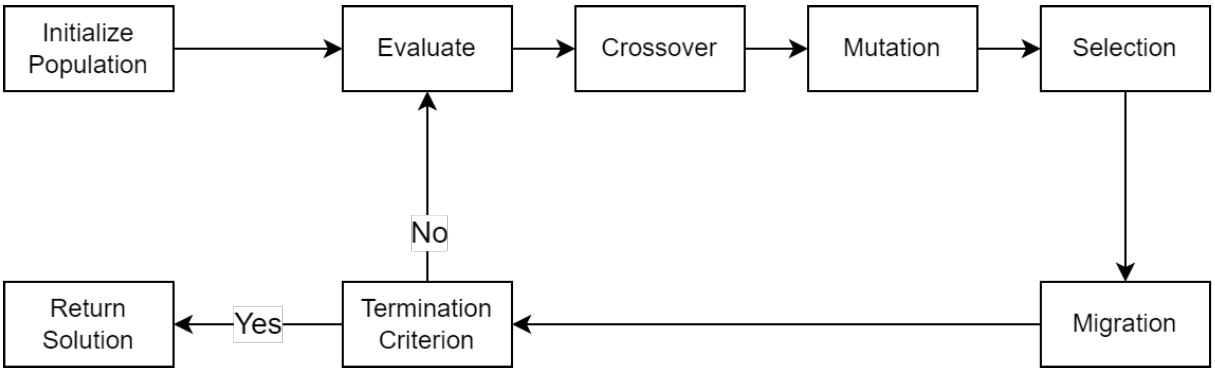


Figure 2.2: Genetic Algorithm architecture with parallelism

Each process generates its own initial population, then each process goes through N generations by performing locally all of the stages described above (Evaluation, Crossover, Mutation, Selection). After these N generations, the processes share their best individuals with the other processes through migration.

2.4.1 Migration Operator

With the implementation of migration, we add another method of affecting the population. The introduction of new individuals will help prevent convergence in a local optimum. During migration, we select some of the best individuals from other processes and replace some of the worst ones from the current process with a migration probability chance. The migration probability chance is critical because it allows us to control the migration's selective pressure. If it is too high, it will lead to faster convergence into a local optimum, and if it is too low, it will lead to fewer individuals migrating and will result in a slower convergence.

2.4.2 Migration Model

For our algorithm, we implemented the Island Model. In this model, processes are allowed to share their individuals with any other process, as illustrated in Figure 2.3. There are no restrictions on where an individual may migrate. This model allows more freedom at the cost of more communication overhead and delay.

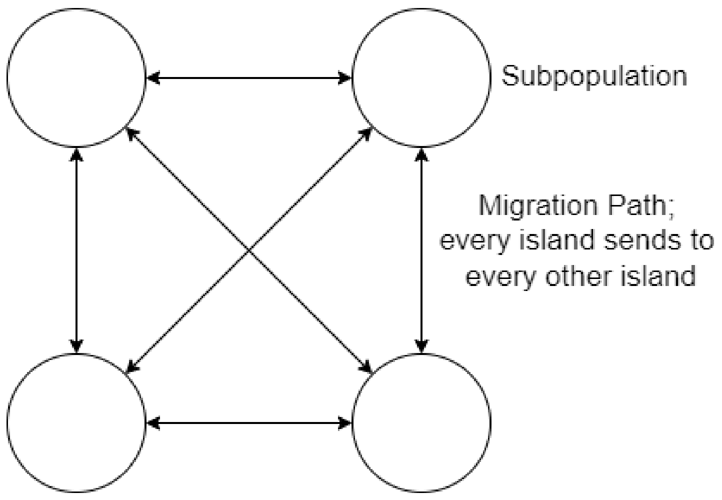


Figure 2.3: Island Model

Chapter 3

Parallel Implementation

The parallel implementation consists of two applications.

A server that is used to facilitate communication between all the clients and to provide the graphical user interface to interact with all the clients.

A client that is used to generate the solution for the traveling salesman problem using a genetic algorithm. The client will also communicate with the server to get information about the other clients results.
The actions that are available through the server:

- You can visualize the solution on a graph.
- Start\Stop the work on all clients connected.
- Switch the instance of the problem to be solved by the clients.
- Export the solution.

The actions that are available through the client:

- You can select the number of processes to be started.
- Connect and receive work from the server.
- Updating the population of the genetic algorithm from the server using migration.

3.1 Technologies used

3.1.1 Python

Python[1] is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

I chose the Python programming language because of its versatility, efficiency, reliability, and extensive support libraries. It is also portable across operating systems.

One of the main concerns and disadvantages of using Python is the Global Interpreter Lock (GIL). GIL is a mutex (or a lock) that allows only one thread to hold control of the Python interpreter. This will make our parallel algorithm redundant when run on a single system. I overcame the limitation of the GIL by using multiple processes instead of threads.

3.1.2 Package Installer for Python (PIP)

Package Installer for Python[2] is a package manager written in Python and is used to install and manage software packages. It hosts thousands of third-party modules for Python.

PIP was used to obtain the library for the graphical user interface and its addons.

3.1.3 PySide2

PySide2 [3] is the official Python module from the Qt for Python project, which provides access to the complete Qt 5.12+ framework.

I chose PySide2 because Qt is a reliable, fast, and easy-to-use graphical user interface framework. It is also cross-platform.

3.1.4 GitHub

GitHub is a provider of Internet hosting for software development and version control using Git. The source code of this project is hosted on Github. I chose Github because it satisfies the requirements of my project.

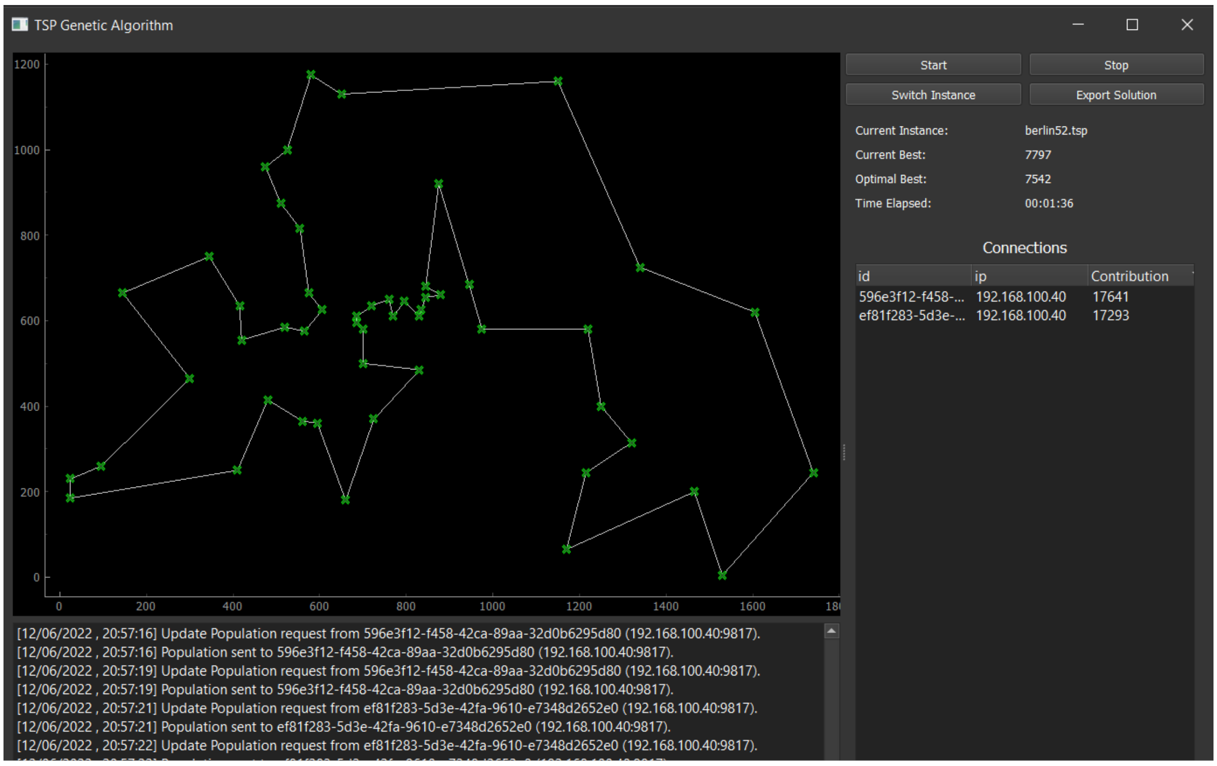
[1]Python `https://www.python.org`

[2]PIP `https://pypi.org/project/pip/`

[3]PySide2 `https://pypi.org/project/PySide2/`

3.2 Server

3.2.1 Graphical user interface



Current Instance:	berlin52.tsp
Current Best:	7797
Optimal Best:	7542
Time Elapsed:	00:01:36

id	ip	Contribution
596e3f12-f458-...	192.168.100.40	17641
ef81f283-5d3e-...	192.168.100.40	17293

```
[12/06/2022 , 20:57:16] Update Population request from 596e3f12-f458-42ca-89aa-32d0b6295d80 (192.168.100.40:9817).
[12/06/2022 , 20:57:16] Population sent to 596e3f12-f458-42ca-89aa-32d0b6295d80 (192.168.100.40:9817).
[12/06/2022 , 20:57:19] Update Population request from 596e3f12-f458-42ca-89aa-32d0b6295d80 (192.168.100.40:9817).
[12/06/2022 , 20:57:19] Population sent to 596e3f12-f458-42ca-89aa-32d0b6295d80 (192.168.100.40:9817).
[12/06/2022 , 20:57:21] Update Population request from ef81f283-5d3e-42fa-9610-e7348d2652e0 (192.168.100.40:9817).
[12/06/2022 , 20:57:21] Population sent to ef81f283-5d3e-42fa-9610-e7348d2652e0 (192.168.100.40:9817).
[12/06/2022 , 20:57:22] Update Population request from ef81f283-5d3e-42fa-9610-e7348d2652e0 (192.168.100.40:9817).
```

Figure 3.1: Graphical user interface

The graphical user interface consists of the following:

1. A window to visualize the current best solution found.
2. A console with the latest logs.
3. A menu that allows the user to control the clients (Start\Stop, Switch instance, Export solution buttons).
4. A section that shows information about the current instance (current instance, current best, optimal best, time elapsed).
5. A list of currently active connections with the server.

3.2.2 Architecture

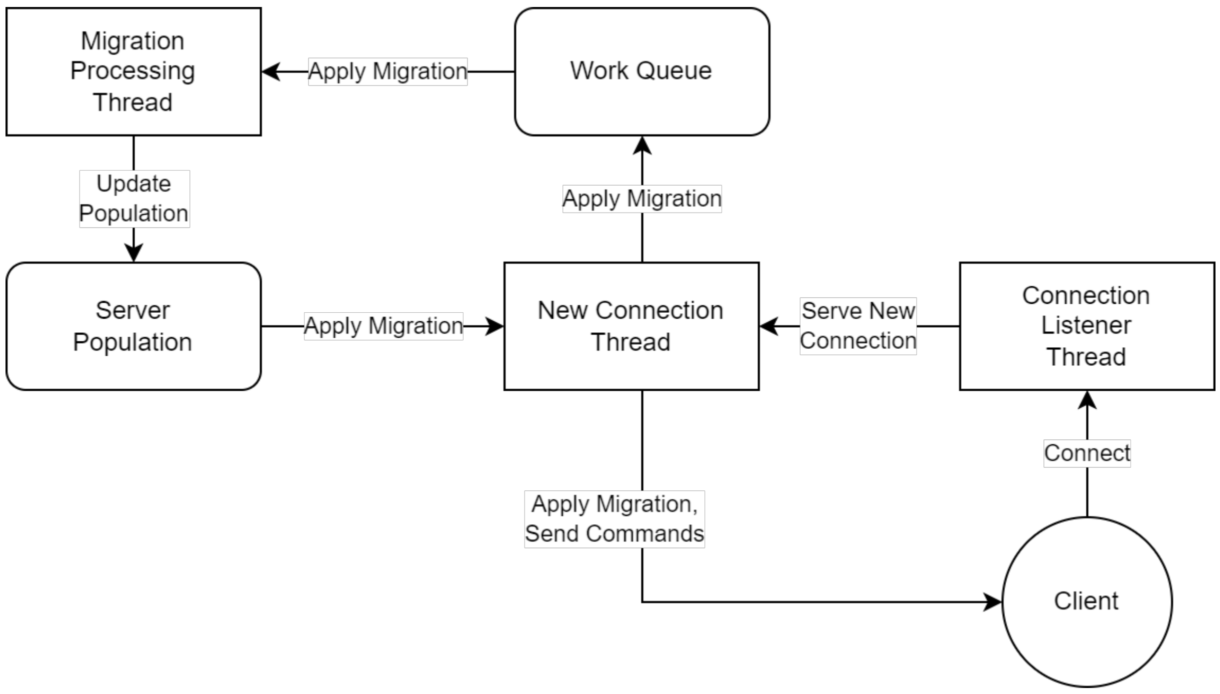


Figure 3.2: Server architecture

The server architecture consists of 3 main threads: the GUI thread, which is used to process the graphical user interface; the connection listener thread, which is used to serve new connections; and the migration processing thread, which is responsible for performing migration with the data received from the clients.

3.2.3 Connection listener thread

The connection listener thread is responsible for the initial communication with the client and the spawning of a new thread to service the new connection.
Possible communication messages are:

- Connect
- CloseConnection
- CloseServer

When a ″Connect″ message is received, it spawns a new thread. The new thread lives while the new connection is active and is used to serve the client with the following communication messages:

- GetWork
- UpdatePopulation
- CloseConnection

When receiving ″GetWork″ from the client, the server will either send the current instance and its parameters to the client or it will send the ″Stop″ message based on its current work state.

When receiving ″UpdatePopulation″ from the client, the server will perform a migration with the client if the instance is the same one. If the instance has changed since the client work has started, the server will send the ″ChangeInstance″ message to update the client to the current instance.

3.2.4 Migration processing thread

The migration processing thread is responsible for managing the population on the server that is used for migration. When a new client sends ″UpdatePopulation″, its current population is put in a work queue to be processed by the migration processing thread, and the server population is sent to perform migration. This work queue is of the Last-In-Last-Out (LILO) type, so that in the event of the thread getting overwhelmed with work, it will process new and improved populations first and be able to skip old and irrelevant populations.

3.3 Client

The client has no graphical user interface and is only available through the command line. You are able to provide it with the number of processes to spawn. Each process spawned has its own connection with the server. By using a multiprocessing approach instead of a multithreaded one, we avoided being CPU-bound by the Python global interpreter lock.

3.3.1 Architecture

The client architecture is the same as the one described in chapter 2.3 except for the added functionality to be able to communicate with the server outside of migration. The main loop of the clients consists of the following:

```
def start(self, ip, port):
    self.initConnection(ip, port)
    while True:
        p, Chromosome_Size, Coordinates = self.getWork()
        if p:
            self.file = p[7]
            self.GA(*p,Chromosome_Size,Coordinates)
        time.sleep(2)
```

At the beginning the client will initialize a connection with the server. This is done by sending the message "Connect" to the server in self.InitConnection(ip, port). Afterwards we keep trying to get work from the server every few seconds until we receive some parameters to start the genetic algorithm.

Then the genetic algorithm will perform migration for every new solution that is better than the last for a faster convergence at the beginning or after every N generations.

When performing migration the client sends to the server the message "UpdatePopulation" and can receive from the server the following:

- Migration
  - If it receives "Migration" from the server that means the migration was accepted and its ok to perform it on the client side as well.

- ChangeInstance
  - If it receives ″ChangeInstance″ from the server it means that the instance has changed on the server and the client is required to do so as well. It will then proceed to restart the genetic algorithm with the new parameters.
- Stop
  - If it receives ″Stop″ it will stop the current genetic algorithm from running.

3.4 Performance

To evaluate the performance of the algorithm, I ran the program on the TSP map *berlin52.tsp* for 60 seconds, 15 times with the same parameters for each number of processes. The optimal solution for this map is 7542.
The results were obtained after running on the following configuration:

- Operating System: Windows 10
- Processor: AMD Ryzen 5 1600 (3.2 GHz)
- Memory: 16 GB

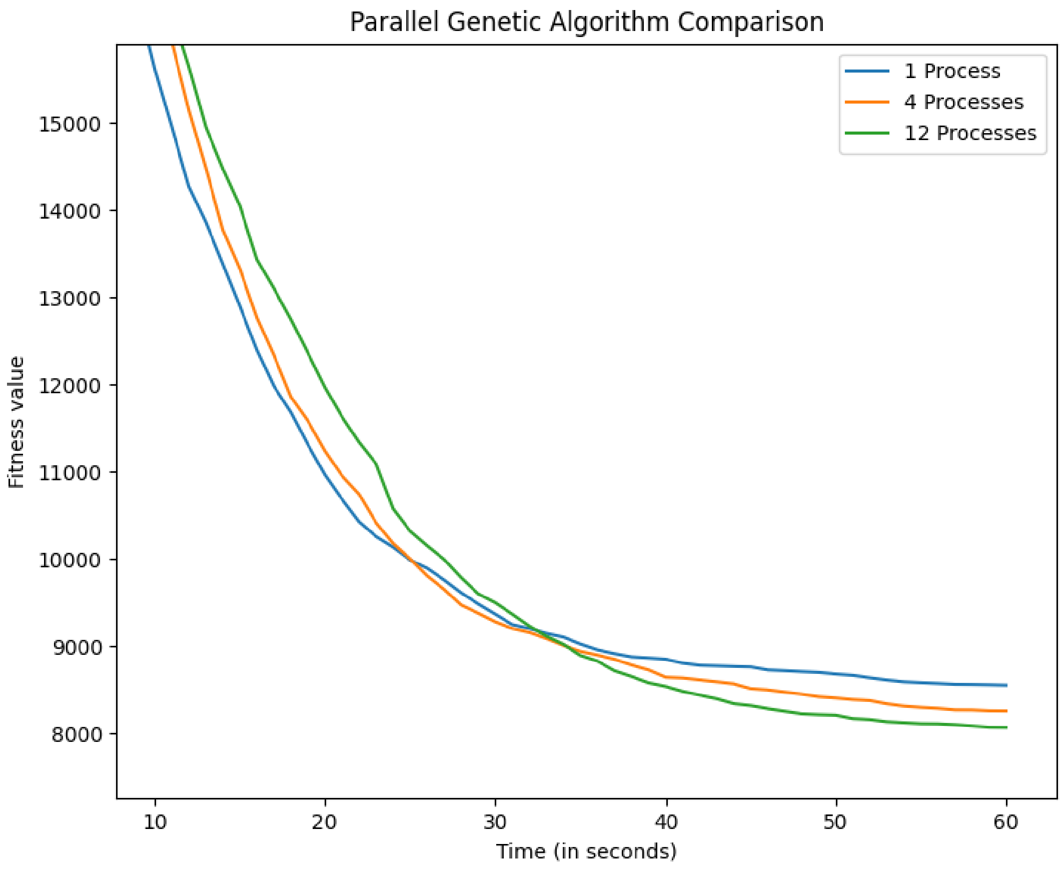


Figure 3.3: Number of processes comparison

From the graph in Figure 3.3 we can observe two things:

- For the first 30 seconds, the fewer the number of processes, the better the results.
- After the first 30 seconds, it's the exact opposite; the higher the number of processes, the better the results.

The first point can be explained by the communication overhead generated by the migration function. We call the migration function every time we find a better solution on a client. This results in a lot of calls to the server, which in turn slows it down until the solutions stop converging so fast.

From the second point it results that there is a relation between the number of processes and the solution found.

Table 3.1: Average of the final values after 60 seconds.

K-Processes	Fitness value
1 Process	8550
4 Processes	8254
12 Processes	8065

Because genetic algorithms are metaheurisitc and use some form of stochastic optimization, they do not guarantee that a globally optimal solution can be found. Because of this, we cannot easily calculate the speedup achieved by a parallel algorithm. But from the results in Table 3.1 and Figure 3.3, we can conclude that there is an increase in performance with the increase in the number of processes.

Conclusions

In this paper, I present my implementation of a solution to the traveling salesman problem using a parallel genetic algorithm. I highlighted the advantages and disadvantages of this type of algorithm and I shared my findings about optimizing such algorithms.

In the end, I achieved better performance by utilizing a parallel algorithm and, despite looking like a small improvement, in the context of genetic algorithms, which are slow and stochastic, it might make the difference between remaining stuck in a local optimum or getting closer to the global one.

Improvements

Although the program is working perfectly fine, i feel like the speed could be further improved. In the future, I would like to make the following improvements:

- Replace some functions of the genetic algorithm with more complex and optimized ones.
- Further improve the communication overhead during migration.
- Improving the graphical user interface and creating one for the client.
- Improve the networking overall between the client and the server.
- Optimizing the parameters to obtain better solutions overall.

Bibliography

[1] Gerhard Reinelt, TSPLIB 95, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf

[2] Otman ABDOUN, Chakir TAJANI and Jaafar ABOUCHABKA, ”Hybridizing PSM and RSM Operator for Solving NP-Complete Problems: Application to Travelling Salesman Problem”. https://doi.org/10.1155/2017/7430125