UAlg
UNIVERSIDADE DO ALGARVE

A Study on the Deployment of GA in a Grid Computing Framework

Sérgio Manuel Correia Baltazar

Dissertação para obtenção do grau de Mestre em Engenharia Informática

Trabalho efetuado sob a orientação de:

Prof. Doutor Helder Aniceto Amadeu de Sousa Daniel

Prof. Doutor José Valente de Oliveira

2015

A Study on the Deployment of GA in a Grid Computing Framework

Declaração de Autoria de Trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

_____

Acknowledgements

I would like to gratefully and sincerely thank my supervisors Helder Daniel and José Valente de Oliveira for the useful comments, remarks and total engagement through the building process of this master thesis. Furthermore I would like to thank the University of Algarve and the Faculty of Sciences and Technology which provided me the physical resources to perform the experiments. I would like to thank my wife Sylvie and my daughter Rita for their unconditional support, patience and love.

Abstract

Genetic algorithms (GA) play a very noticeable role for solving optimization problems, including many scientific, economic and socially relevant ones. GAs, along with genetic programming (GP), evolutionary programming (EP), and evolution strategies, are the major classes of evolutionary algorithms (EAs), i.e. algorithms that simulate natural evolution. In real-world applications the runtime of GAs can be computationally demanding mainly due to the requirements on the population size. This issue can be mitigated using parallelization, which can lead to faster and better performing GAs. Although most of the existing implementations of Parallel Genetic Algorithms (PGAs) use either clusters or massively parallel processing (MPP), *grid* computing is economically relevant (can be built with out-of-date computers) and has some advantages over clusters such as no centralized control, security and access to distributed heterogeneous resources in dynamic worldwide virtual organizations. This research uses the real-world Travelling Salesman Problem (TSP) as a benchmark for the parallelization of GAs in a grid computing framework. TSP is a well-known NP-hard combinatorial optimization problem that can be formally described as the problem of finding the shortest Hamiltonian cycle in a graph. In fact, many routing, production and scheduling problems found in engineering, industry and business, can be shown to be equivalent to TSP, thus its interest. Informally, the problem can be described as follows. A salesman has a large number of cities to visit and he needs to find the shortest path to visit all the cities, without revisiting any of them. The main difficulty in finding optimal solutions to TSP is the large number of possible tours; (n-1)!/2 for symmetric n cities tour. As the number of cities in the problem increases, the number of possible tours also increases, in a factorial way. TSP is therefore computationally intractable, thus fully justifying the employment of a stochastic optimization method such as GA. However even stochastic optimization algorithms may take too much time to compute, as the problem size increases. In a GA for large populations the time required to solve the problem may be excessively long. One way of speeding up such algorithms is to use additional resources such as additional processing elements running in parallel and collaborating to find the solution. This leads to concurrent implementations of GAs, suitable to be deployed on parallel and/or distributed collaborating resources. Parallel evolutionary algorithms (PEAs) are intended to implement faster and better performing algorithms, using structured populations, i.e., spatial distributions of individuals.

A possible way of decentralizing a population is distributing it by a set of processing nodes (islands) which periodically exchange (migrate) candidate solutions; the so-called Island Model. The island model allows for a considerable number of migration topologies and, to the best of our knowledge, there is a lack of research work concerning the comparison of those migration topologies, when implementing PEAs in grid computing frameworks. In fact, the comparison of migration topologies using a grid computing framework, as proposed in this work, does not seem to be present in the literature. This comparison aims at providing a technically sound question to the research question:

What is the fastest Island Model topology for solving TSP instances using an order-based genetic algorithm, in a distributed heterogeneous grid computing environment, without losing significant fitness comparatively to the correspondent sequential panmictic implementation of the same algorithm?

A *hypothesis* aiming at answering the research question can be stated as follows:

For solving TSP instances using an order-based genetic algorithm, in a distributed heterogeneous grid computing environment, without losing significant fitness comparatively to the correspondent sequential panmictic implementation of the same algorithm, choose a coordinated Island Model topology, from any of the tested topologies (star, cartwheel, tree, fully connected multilayered, rooted tree-ring, ring), with as many nodes as possible (even slow ones) and select the migration frequency that optimizes the execution time for the chosen topology.

The research methodology is primarily experimental, observing and analysing the behaviour of the algorithm while changing the properties of the island model.

Results show that the GA is speeded up when deployed in a grid environment, while maintaining the quality of the results obtained in the sequential version. Furthermore, even obsolete computers can be used as nodes contributing to speed up the execution time of the algorithm. This work also argues the suitability of an asynchronous approach for deploying GA in a grid computing environment.

Keywords

Genetic Algorithms; Island Model; Asynchronous Genetic Algorithm; Grid Computing; Globus Toolkit; Travelling Salesman Problem;

Resumo

Os algoritmos genéticos (AG) desempenham um papel importante na resolução de muitos problemas de otimização, incluindo científicos, económicos e socialmente relevantes. Os AGs, conjuntamente com a programação genética (PG), a programação evolutiva (PE), e as estratégias de evolução, são as principais classes de algoritmos evolutivos (AEs), ou seja, algoritmos que simulam a evolução natural. Em aplicações do mundo real o tempo de execução dos AGs pode ser computacionalmente exigente, devido, principalmente, aos requerimentos relacionados com o tamanho da população. Este problema pode ser atenuado através da paralelização, que pode levar a GAs mais rápidos e com melhor desempenho. Embora a maioria das implementações existentes de Algoritmos Genéticos Paralelos (AGPs) utilize *clusters* ou processamento massivamente paralelo (PMP), a computação em *grid* é economicamente relevante (uma *grid* pode ser construída utilizando computadores obsoletos) e tem algumas vantagens sobre os *clusters*, como por exemplo a não existência de controlo centralizado, segurança e acesso a recursos heterogéneos distribuídos em organizações virtuais dinâmicas em todo o mundo. Esta investigação utiliza o problema do mundo real denominado de Problema do Caixeiro Viajante (PCV) como referência (*benchmark*) para a paralelização de AGs numa infraestrutura de computação em *grid*. O PCV é um problema NP-difícil de otimização combinatória, bem conhecido, que pode ser formalmente descrito como o problema de encontrar, num grafo, o ciclo hamiltoniano mais curto. De facto, muitos problemas de roteamento, produção e escalonamento encontrados na engenharia, na indústria e outros tipos de negócio, podem ser equiparados ao PCV, daí a sua importância. Informalmente, o problema pode ser descrito da seguinte forma: Um vendedor tem um grande número de cidades para visitar e precisa encontrar o caminho mais curto para visitar todas as cidades, sem revisitar nenhuma delas. A principal dificuldade em encontrar as melhores soluções para o PCV é o grande número de caminhos possíveis; *(n-1)! / 2* para um caminho de *n* cidades simétricas. À medida que o número de cidades aumenta, o número de caminhos possíveis também aumenta de uma forma fatorial. O PCV é, portanto, computacionalmente intratável, justificando plenamente a utilização de um método de otimização estocástica, como os AGs. No entanto, mesmo um algoritmo de otimização estocástica pode demorar demasiado tempo para calcular, à medida que o tamanho do problema aumenta. Num AG para grandes populações, o tempo necessário para resolver o problema pode até ser excessivamente longo.

Uma forma de acelerar tais algoritmos é usar recursos adicionais, tais como elementos adicionais de processamento funcionando em paralelo e colaborando para encontrar a solução. Isto leva a implementações simultâneas de AGs, adequadas para a implementação em recursos colaborando em paralelo e/ou de forma distribuída. Os Algoritmos evolutivos paralelos (AEPs) destinam-se a implementar algoritmos mais rápidos e com melhor desempenho, usando populações estruturadas, ou seja, distribuições espaciais dos indivíduos. Uma das maneiras possíveis de descentralizar a população é distribuí-la por um conjunto de nós de processamento (ilhas) que trocam periodicamente (migram) potenciais soluções; o chamado modelo de ilhas. O modelo de ilhas permite um número considerável de topologias de migração e, pela Informação que foi possível apurar, há uma carência de trabalhos de investigação sobre a comparação dessas topologias de migração, ao implementar AEPs em infraestruturas de computação em *grid*. De facto, a comparação de topologias de migração, utilizando uma infraestrutura de computação em *grid*, como proposto neste trabalho, parece não estar disponível na literatura. Esta comparação tem como objetivo fornecer uma resposta tecnicamente sólida para a questão de investigação:

Qual é a topologia, de modelo de ilhas, mais rápida para resolver instâncias do PCV usando um algoritmo genético baseado em ordem, num ambiente de computação em grid, heterogéneo e distribuído, sem uma perda significativa de fitness, comparativamente com a implementação sequencial e panmítica do mesmo algoritmo?

Uma hipótese para responder à questão de investigação pode ser expressa da seguinte forma:

*Para resolver instâncias TSP, usando um algoritmo genético baseado em ordem, num ambiente de computação em grid, heterogéneo e distribuído, sem uma perda significativa de fitness, comparativamente com a implementação sequencial e panmítica do mesmo algoritmo, escolha qualquer uma das topologias coordenadas do modelo de ilhas, de entre as topologias testadas (estrela, roda, árvore, matriz totalmente conectada**,** árvore-anel**,** anel) com o maior número de nós possível (mesmo os mais lentos) e selecione a frequência de migração g que otimiza o tempo de execução para a topologia escolhida.*

A metodologia de investigação é essencialmente experimental, observando e analisando o comportamento do algoritmo ao alterar as propriedades do modelo de ilhas. Os resultados mostram que o AG é acelerado quando implementado num ambiente *grid*, mantendo a qualidade dos resultados obtidos na versão sequencial. Além disso, mesmo os computadores obsoletos podem ser usados como nós contribuindo para acelerar o tempo de execução do algoritmo. Este trabalho também discute a adequação de uma abordagem assíncrona para a implementação do AG num ambiente de computação em *grid*.

Termos Chave

Genetic Algorithms; Island Model; Asynchronous Genetic Algorithm; Grid Computing; Globus Toolkit; Travelling Salesman Problem;

Index

CHAPTER 1: Introduction

Genetic algorithms (GA) play a very noticeable role for solving optimization problems, including many scientific, economic and socially relevant ones. GAs, along with genetic programming (GP), evolutionary programming (EP), and evolution strategies, are the major classes of evolutionary algorithms (EAs), i.e. algorithms that simulate natural evolution. A GA operates on a population of individuals each one of them being a candidate solution to the considered optimization problem. In the classical form, each individual is composed by one or more chromosomes which in turn are viewed as a set of genes. In real-world applications the runtime of GAs can be computationally demanding mainly due to the requirements on the population size. Another potential issue is the premature convergence of the population. It is well-known that both of these issues can be mitigated using parallel genetic algorithm (PGA) (Alba and Troya, 1999; Cantú-Paz, 1999; Cantú-Paz, 2000; Alba and Tomassini, 2002). Unfortunately, relatively to GA, PGA increases the number of parameters the practitioner has to deal with. Currently, there is a wealth of work on PGA, e.g., see (Alba, 2005; Knysh and Kureichik, 2010; Umbarkar and Joshi, 2013; Luque and Alba, 2011; Johar et al., 2013; Alba, Luque, and Nesmachnow, 2013) for some recent reviews.

Classical taxonomies of PGA distinguish between single population (panmictic) models and multi-population or structured models (Cantú-Paz, 2000; Nowostawski and Poli, 1999). The former can be viewed as a master-slave architecture where the master runs a panmictic GA and is the evaluation of individuals that is performed in parallel by slave processors. This model is also known as the parallel panmictic model has it does not change the behavior of the panmictic GA. In the multi-population (or deme) PGA the two main models are i) the cellular or fine-grained GA, and ii) the coarse-grained, distributed, or island model (IM). In the fine-grained the population is partitioned into a high number of small sub-populations typically one individual per processing unit (crossover is restricted to a small overlapping neighborhood. Although hybrid and hierarchal models are also common the IM is by far the more popular model, and is also adopted in this work.

The island model has its roots in a model used to describe natural sub-populations (or demes) that evolve in a semi-isolated way as it happens in islands (Cohoon et al., 1987). Following this metaphor, in an IM each deme is processed by an island, i.e., a GA in its own processing unit. Occasionally, demes exchange some of their individuals (migrants) with other demes. The details of the exchange include the migration frequency (how often migration occurs), the number of migrants, the policy of migrants (which individuals migrate and which are replaced at the receiving deme), and the topology of migrations. Topology establishes the communication pathways between islands. In another words, a topology defines the flow of migrants from one subpopulation to another. The topology is arguably the less studied of the above aspects, especially in the asynchronous heterogeneous case. Migration can be synchronous or asynchronous. Synchronous migration is typically easier to analyze as it occurs periodically at the same time instant for all islands. Asynchronous migration is harder to analyze as it depends on the events occurring in each island and, thus each island migrates at its own time independently of the others, as found often in nature. Islands can be homogeneous or heterogeneous depending on whether they are based on the same GA with the same parameters and deployed over similar processing elements, or not. The deployment can be either in hardware, or simulated in software.

Although most of the existing implementations of Parallel Genetic Algorithms (PGAs) use either clusters or massively parallel processing (MPP), *grid* computing is economically relevant and has some advantages over clusters such as no centralized control, security and access to distributed heterogeneous resources in dynamic worldwide virtual organizations. In fact, a grid computing framework can be constructed using commodity computing, i.e. normal, open-standards and often outdated hardware for parallel computing, in the antipodes of the high-performance and high-cost supercomputing. Given the past and current pace of replacement of computing hardware in teaching, development, and research labs, more and more processing power is becoming available for useful computing at low cost, if any. This has the potential to allow the study larger instances of hard problems under modest budgets, e.g., (Brightwell et al., 2000; Myers and Cummings, 2003; Plaza et al., 2006; Bernabe and Plaza, 2011).

A grid can be viewed as a form of distributed computing where a virtual super computer is built out of many networked, loosely coupled, and geographically dispersed computing nodes (Foster, Kesselman, and Tuecke, 2001). Each node (desktop pc, laptop, workstation, transputer, cluster, etc.) may be physically connected using conventional network hardware, thus being an effective way of sharing commodity computing resources. However, the characteristics of such computing environment rises some interesting challenges. For instance: it is expected a high level of heterogeneity among nodes (different processing speeds and different storage). Also, as nodes can be geographically separated and connected over the internet, communication times among the units are random variables with high variability. Consequently, parallel synchronous computing (where nodes synchronize at specific times) does not serve the purpose of shortest completion times without some kind of load balancing. Asynchronous parallel approaches are therefore preferred. Moreover, as each subpopulation of the PGA is assigned to a processing node, and as processing nodes exhibit different processing characteristics, each subpopulation has its own evolutionary pace. In other words, different islands evolve at different rates.

In the past, several studies used grid-enabled computing for parallelizing genetic and evolutionary algorithms. These make use of the master-slave model, e.g., (Durillo et al., 2008; Nebro et al., 2008); the cellular model, e.g., (Dorronsoro et al., 2007; Luque, Alba, and Dorronsoro, 2009); the island model, e.g., (Limmer and Fey, 2010; Luna et al., 2008; Melab, Cahon, and Talbi, 2006; Talbi, Cahon, and Melab, 2007); and hierarchical and hybrid models, e.g. (Lim et al., 2007; Tantar et al., 2007). However, none of these works evaluates the influence of the topologies. Typically, the ring topology is used. If computing nodes can dynamically enter or leave the grid, ring topologies are among the more vulnerable ones. Nevertheless, the island model allows for a considerable number of migration topologies but, to the best of our knowledge, there is a lack of research on the impact of topologies on the performance of the whole optimization process, when deployed over a grid computing framework. Also, studies on island models based on heterogenous islands, using different GA and different GA settings, are available, e.g., (Cantú-Paz, 2000; Alba, Nebro, and Troya, 2002; Baugh and Kumar, 2003; Jakobović, Golub and Čupić, 2014). Unfortunately, the migration topology issue is not discussed in the mentioned studies neither.

This research focus on the comparison of topologies for parallelizing order-based genetic algorithms when distributed over heterogeneous computing environments. To the best of our knowledge, no study either theoretical or experimental is available for such scenario. In particular, the study focus on the analysis of the influence of the migration topologies and their relationship with migration frequency. Given heterogeneity, all the analyzed topologies can be viewed as coordinated topologies. In such type of topologies there is one island (the coordinator) that is responsible for generating the solution, and for stopping the evolution of all other (coordinated) islands once the solution is found. Notice that this is the sole function of the coordinator; no synchronism is implied. We study three notable cases: i) the case where no migration takes place (isolated island), ii) the case where migration flows only in the direction of the coordinator (star, cartwheel, rooted tree-ring, trees, fully connected multi-layered), as well as iii) the case where migration flows to, and from, the coordinator (rings).

This work uses the real-world Travelling Salesman Problem (TSP) as a benchmark for the parallelization of GA in a grid computing framework. The Travelling Salesman Problem (TSP) is a well-known, and widely studied, NP-hard (Garey and Johnson, 1979) combinatorial optimization problem. Many routing, production and scheduling problems found in engineering, industry and business can be shown to be equivalent to TSP, thus its interest. One may expect that the techniques and findings observed for TSP could be also applied to other combinatory problems (Applegate et al., 2007). Moreover, there are several real-world benchmark instance and their optimal solutions freely available (Reinelt, 1991). TSP seems to show a good trade-off between complexity and the type of computation resources we are interested in this study. Many studies on (P)GA also adopt the same problem, not necessarily using the same instances, e.g., (Grefenstette et al., 1985; Braun, 1990; Potvin, 1996; Sena, Megherbi, and Isern, 2001; Wang et al., 2005; Weise et al., 2014). Informally, the problem can be described as follows. A salesman has a large number of cities to visit and he needs to find the shortest visiting path to all cities, without revisiting any of them (Lawler, 1985). More formally, TSP can be described as the problem of finding the shortest Hamiltonian cycle in a graph. Figure *1.1* presents the TSPLIB (Reinelt, 1991) data set for *13509* cities in the USA (*USA13509*). TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types. Furthermore, TSPLIB also provides the best known solutions for those sample instances.

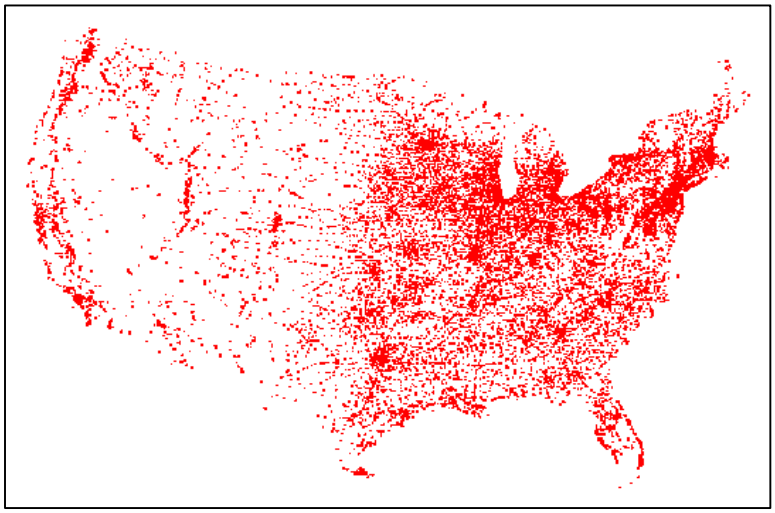


Figure 1.1: TSPLIB *USA13509* data set.

The main difficulty in finding optimal solutions to TSP is the large number of possible tours; (*n*-1)!/2 for symmetric *n* cities tour. As the number of cities in the problem increases, the number of possible tours also increases, in a factorial way. TSP is therefore computationally intractable, thus justifying the employment of a stochastic optimization method such as GA. For illustration purposes, figure *1.2* presents the optimal tour for the *USA13509* data set, found by (Applegate et al., 1998).

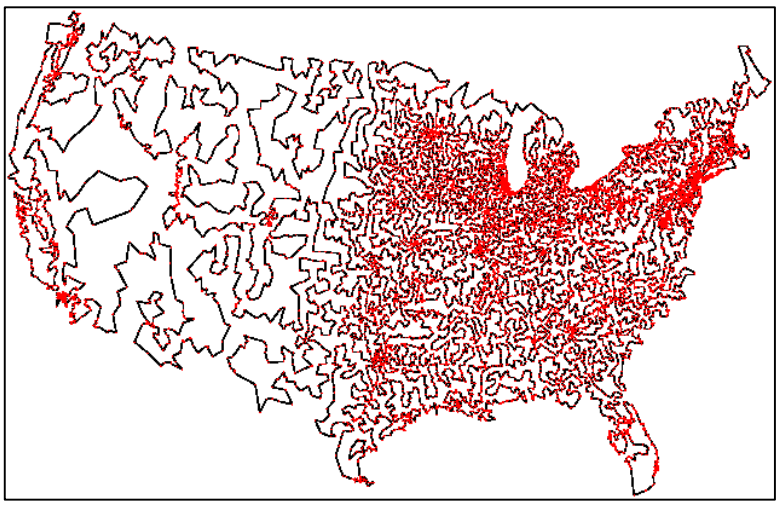


Figure 1.2: TSPLIB *USA13509* optimal tour.

1.1. Thesis Contributions

The main contribution of this research is to provide a technically sound answer to the following question:

What is the fastest Island Model topology for solving TSP instances using an order-based genetic algorithm, in a distributed heterogeneous grid computing environment, without losing significant fitness comparatively to the correspondent sequential panmictic implementation of the same algorithm?

In order to answer this question, a comparison of migration topologies, using a grid computing framework, was performed. To the best of our knowledge, this comparison, in a distributed heterogeneous environment such as proposed in this work, cannot be found in the literature. The research methodology is primarily experimental, observing and analysing the behaviour of the algorithm while changing the properties of the island model.

1.1. Thesis Organization

The organization of the text is as follows: Chapter *2* introduces Evolutionary Algorithms, points their application in real-world optimization problems and identifies Genetic Algorithms as one of their major classes. Evolutionary algorithms models are presented and the Travelling Salesman Problem is introduced as well as the elected baseline Genetic Algorithm to solve this optimization problem. Chapter *3* introduces Parallel Evolutionary Algorithms as well as a taxonomy of these algorithms. The parallelization of the baseline GA is presented, arguing the suitability of coordinated island model topologies for deployment of GAs in distributed heterogeneous environments such as grid. In chapter *4*, a comparison of the selected island model topologies is presented. Some of the most important results are presented and discussed. Finally, chapter *5* presents the most noticeable conclusions of this work, providing an answer to the research question. The thesis ends with some future work considerations pointing further possible investigation directions.

CHAPTER 2: Evolutionary Algorithms

Evolutionary algorithms (EAs) are stochastic optimization algorithms which simulate natural evolution, i.e. they are based on Darwin's evolution theory of natural selection and "survival of the fittest". First proposed by (Rechenberg, 1965), evolutionary algorithms have been widely applied to combinatorial optimization problems in several different domains such as biology, chemistry, computer aided design, cryptanalysis, medicine, microelectronics, pattern recognition, production planning, robotics, telecommunications, etc. The major classes of EAs are the genetic algorithms (GAs) (Holland, 1975), genetic programming (GP) (Koza, 1992), evolutionary programming (EP) (Fogel, 1962), and evolution strategies (ESs) (Rechenberg, 1965).

In GAs, the search space of a problem is represented by a collection, i.e., a *population*, of *individuals*, often referred to as *chromosomes*. Each individual, i.e., chromosome, is, in fact, a candidate solution for the problem. The goal of using a GA is to find the individual in the population which has the best "genetic material", i.e., that represents the best solution for the problem been solved. The quality of an individual is calculated using an evaluation function, also known as *fitness* function. The process of natural evolution is simulated by using genetic operators, namely *selection*, *crossover* and *mutation*. A simple genetic algorithm works as follows (Goldberg, 1989):

1) The initial population is randomly generated;
2) In every iteration of the algorithm, i.e., generation:
    a. Individuals are evaluated using the fitness function;
    b. Parents are selected from the population;
    c. Parents produce children by crossover;
    d. Mutation is performed over the newly created children;
    e. Some individuals, selected according to a given criteria, are removed from the population, assuring that the initial population size is maintained;
3) When stopping criteria is achieved the algorithm stops, otherwise it proceeds to next generation (step 2).

2.1. Evolutionary Algorithm Models

There are two main subclasses of Evolutionary Algorithms (EAs): panmictic and structured EAs (Alba and Tomassini, 2002). In *panmictic* or *global* EAs, selection operation is executed globally, i.e., any individual can compete or mate with any other and can be replaced by a new one. In the case of structured EAs, individuals are arranged spatially and thus the population is divided in several subpopulations.

Panmictic EAs can be classified as generational or steady-state algorithms. Generational models assume that at each step, i.e., generation, the whole population of N individuals is replaced by a new one. On the other hand, in a steady-state EA, one single individual is replaced at each step. Steady-state and generational can be considered as the two extremes of generation gap algorithms where a given number of the individuals M (mortality) are replaced with new ones (Jakobovic et al., 2013). $M=1$ for steady-state EAs and $M=N$ for generational EAs.

As for structured EAs, island models and cellular (cEA) algorithms are very popular optimization procedures (Alba and Troya, 1999). Island models are also known as coarse-grained as they deal with isolated subpopulations which exchange individuals between them, i.e., every g generations, n individuals migrate from one subpopulation to another. On the other hand, in cEAs (fine-grained EAs), an individual has its own pool of potential mates defined by neighbouring individuals, as shown in Figure *2.1.1*.

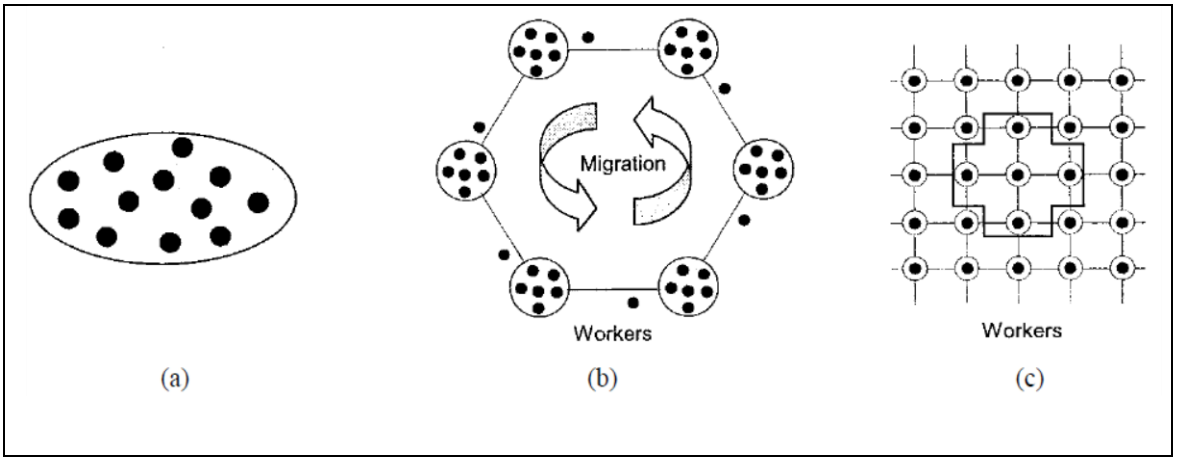


Figure 2.1.1: Panmictic EA (a), Island Model (b), Cellular EA (c), in (Alba and Tomassini, 2002).

Distributed EAs and Cellular EAs provide a better sampling of the search space and improve the behaviour of the basic algorithm (Gordon and Whitley, 1993). A distributed model is usually faster than a panmictic EA, as the run time and the number of evaluations are potentially reduced thanks to its separate search in several regions from the problem space, featuring high diversity and species formation (Alba and Tomassini, 2002).

So far, all the presented models assume that the genetic material, as well as the evolutionary conditions, e.g., selection and recombination methods, are the same for all individuals and all populations of a structured EA. According to (Alba and Tomassini, 2002), these algorithm types are called *uniform*. If different subpopulations are allowed to evolve with different parameters and/or with different individual representations for the same problem, then the distributed algorithm is called *nonuniform.* The first to study the use of different mutation and crossover rates in different populations was (Tanese, 1987). An example of nonuniform dEAs was provided by (Lin et al., 1994) with their *injection island GA* (iiGA), which considers multiple populations encoding the same problem but using a different representation size in different islands. In iiGA, the migration of individuals is one-way only, i.e., from a low-resolution node to a high-resolution node. Similar topology approaches have been used, such as (Herrera et al., 1998) or (Sefrioui and Périaux, 2000).

2.2. The Baseline Genetic Algorithm to Solve TSP

An important decision in this work was the selection of the baseline genetic algorithm for solving TSP, to be first implemented in its sequential version. In order to reduce the enormous number of possibilities, the programming language was the first filter to be applied. Some authors argue that the features of *Java* make it a candidate language for grid computing (Getov et al., 2001). In the past years, *C/C++* and *Java* have been the most commonly used languages to develop parallel metaheuristics implementations (Parejo et al., 2012). Assuming *Java* as the chosen programming language, the next step was the choice of a known efficient, and fast (for challenging reasons), a genetic algorithm implemented in that programming language. The *Java* implementation proposed by (Saiko, 2005), has been reported as very efficient and has been referred in recent scientific publications such as (Rzeźniczak, 2012) and (Saračević et al., 2012). Saiko's implementation is inspired on the original algorithm proposed in *A Fast TSP Solver Using GA on Java* (Sengoku and Yoshihara, 1998) that obtained quite good results while increased the execution speed of the algorithm. The chosen algorithm combines:

a) *Greedy Crossover* (Grefenstette et al., 1985) mating, based on the crossover engine from JGAP (Meffert and Rotstan, 2014);
b) Mutation, by random permutation;
c) Path optimization, provided by the *2-opt* method (Croes, 1958).

The *Greedy Crossover* operator selects the first city of one parent, compares the cities next to that city in both parents, and chooses the closer one to extend the tour. If one city has already appeared in the tour, it chooses another city. If both cities have already appeared, it randomly selects another city. Mutation is based on random permutation, i.e. swapping two randomly selected cities from the chromosome. The *2-opt* optimization method is a simple local search algorithm that allows eliminating path crossings in a fast and efficient way, as illustrated on figure *2.2.1*.

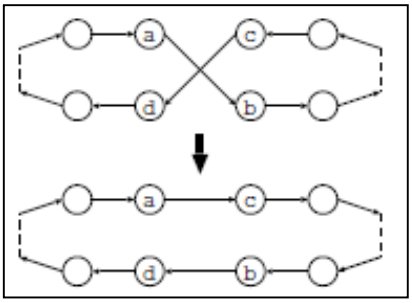


Figure 2.2.1: The *2-opt* optimization method, in (Sengoku and Yoshihara, 1998).

Figure *2.2.2* presents the pseudo code for the sequential panmictic version of the chosen algorithm. Given a dataset of *N* cities, each one placed at (*x*,*y*) coordinates, each chromosome, i.e., each individual of the population, is represented by an ordered list of cities, corresponding to a possible tour of the salesman. The goal of the algorithm is to find the optimal tour, corresponding to the minimal distance travelled to visit all the cities. So, the fitness of each chromosome is determined by the tour length and the main goal is to minimize that value, i.e., minimize the total distance travelled by the salesman.

```
Set the population size N, the number of migrants k, and the migration frequency g;
Randomly initialize population;
Sort population by fitness (from smaller distance to larger);
repeat
    for j := 1 to N/2 do
        Randomly select parents p1 and p2 from the best half of the population;
        child1 := Permutation (Clone (p1));
        child2 := Permutation (Clone (p2));
        child3 := Crossover (p1; p2);
        child4 := Crossover (p2; p1);
        child5 := Permutation (Clone (child3));
        child6 := Permutation (Clone (child4));
        Apply 2opt method to all children;
        Add all children to the population;
    end
    Sort population by fitness (from smaller distance to larger);
    Remove the last (worst) individuals beyond the initial population size;
until no change in the best individual for a LIMIT (s);
```

Figure 2.2.2: Pseudo-code for the baseline sequential panmictic version of the GA.

The initial population is randomly generated, assuring that there is no chromosome duplication, and then ordered through fitness evaluation. Although the original size of the population, as proposed by the author, was *N=1000*, one could conclude that, as far as this research goes, there is no fitness improvement when the population size goes beyond the number of cities of the data set we want to run.

Looking at figure *2.2.3.*, one can observe that the best tour length decreases as population size increases and stabilizes when the population size equals the number of cities on the data set. These results were obtained for the N=280 instance of the TSP problem. From this point, even duplicating the population size (*N=560*), the algorithm found the same best tour length. Similar results are observed for other instances of TSP. In each generation, the best solution is compared with the best solution from the previous generation. The algorithm stops when the best solution remains unchanged for *s* consecutive generations. The original value proposed by the author was *s=100* generations but one could experimentally conclude that *s=52* was enough and thus this was the parameter value used for all the performed experiments on this research.

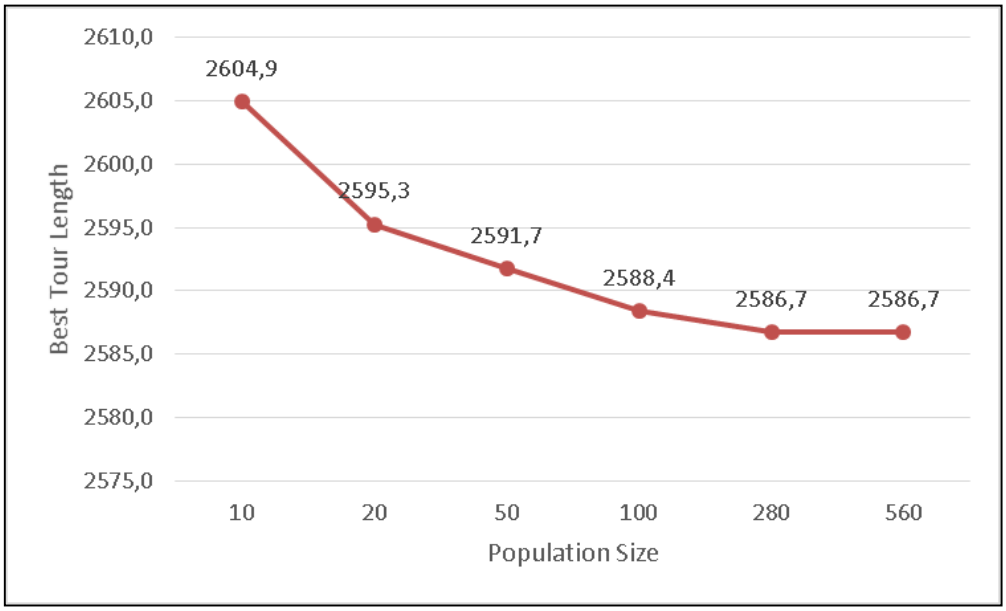


Figure 2.2.3: Average Best Tour Length (280 cities data set).

Every generation, after checking the best solution age, the second half, i.e., the worst half of the population is removed. Then for as many iterations as half the initial population size, the following process is repeated:

a) Two parents are randomly selected;
b) Four children (children 1, 2, 5 and 6) are obtained by cloning both parents and two children (children 3 and 4) by mating using *GreedyCrossover*;
c) Children 1, 2, 5 and 6 are then mutated, with a probability of mutation *Pm=0.25*, by random permutation;
d) All new children are optimized by the *2-opt* heuristic method and finally added to the existing population.

After this set of iterations, the resulting population is once again sorted by fitness evaluation (from smaller distance to larger) and the worst chromosomes, that exceed the initial population size, are removed from the population in order to maintain the initial population size.

CHAPTER 3: Parallel Evolutionary Algorithms

Considering that genetic algorithms have an inherent property of implicit parallelism (Holland, 1975), they are appropriate candidates for parallelization. Such parallel implementations enable the evolution, in parallel, of distinct populations and allow scalability to larger ones. A parallel genetic algorithm (PGA) is usually faster, less prone to finding only sub-optimal solutions, and able to cooperate with other search techniques in parallel (Alba and Troya, 1999). Commonly, a parallel program divides a task into chunks which are simultaneous processed using multiple processors. If these chunks are ran in different computers the implementation of the program is also addressed as distributed.

According to (Hart et al., 1996), PGAs provide the following benefits:

- Reduce the time to locate a solution;
- Reduce the number of function evaluations;
- Explore the large populations size over the parallel platforms used for running the algorithms;
- Improve the quality of the solutions;
- Solves large scale, large dimensions problems with more efficacy and efficiency.

The parallelization method can use a single population while parallelizing the most time consuming tasks, or divide the population into several subpopulations. Although the fact that genetic algorithms are natural candidates for parallelization, in some cases the parallel implementation of an algorithm may produce worst results than its *sequential* version, e.g., in cases where the communication delay between processing elements is not negligible. This may occur when the problem to solve has a lower level of complexity and therefore the execution of the algorithm is not especially time-consuming. The most recent studies on PEAs still adopt island models as a privileged approach for implementation of PEAs. (Andalon-Garcia and Chavoya, 2012) present a comparison of different topologies of the Island model, (Jakobović et al., 2014) propose the design and the application of asynchronous models of parallel evolutionary algorithms, (Lopes et al., 2013, 2014) study the dynamic selection of migration flows and the configuration of migratory flows, (Lässig and Sudholt, 2013) present a rigorous analysis of migration in PEAs and (Märtens and Izzo, 2013) propose an asynchronous island model for multi-objective evolutionary optimization on heterogeneous and large-scale computing platforms.

Notice that a sequential algorithm may also divide the global population in subpopulations. Thus, hereafter, *serial* stands for the sequential implementation of the *panmictic* version of the algorithm.

3.1. Parallel Evolutionary Algorithm Models

Parallel evolutionary algorithms (PEAs) can be classified as i) master-slave or global, ii) distributed (coarse grained or fine grained), and iii) hybrid (Jakobović et al., 2014). Master-slave or global PEA has a single population and usually only parallelizes the fitness evaluation operation. Each individual may compete for selection and mate with any other individual, as for the panmictic version of the EA. Assuming that fitness evaluation operation is the most time consuming genetic operation, several authors have been used global PEAs, applying parallel methods exclusively to fitness evaluation, such as (Cantú-Paz, 1997; Borovska, 2006; Cantú-Paz, 2007).

Distributed EAs (DEAs) are also called multiple-deme parallel EAs, island EAs or coarse grained EAs. DEAs have a relatively small number of demes, i.e., subpopulations, and individuals occasionally migrate between demes. The migration mechanism requires the definition of several additional parameters, e.g., communication topology, migration condition, number of migrants, migrant selection and integration method. Considering that demes may overlap, the same set of individuals may belong to more than one deme (Nowostawski and Poli, 1999). DEAs have been extensively used, taking advantage of the availability of numerous computing nodes in order to achieve high execution speedups (Park et al., 2008; He et al., 2007; Melab et al., 2006; Nowostawski and Poli, 1999; Alba et al., 2002; Alba et al., 2004). On the other hand, a small number of cellular PEAs implementations is reported, such as (Eklund, 2004), as they often need a specialized hardware platform for their implementation.

Hybrid parallel EAs can implement both global and distributed methods in a single algorithm, e.g., multiple-deme models with master-slave algorithms run on each deme (Jakobović et al., 2014). This kind of approach was presented by (Acampora et al., 2011), providing a hybrid solution for the e-learning experience binding problem, and by (Iturriaga et al., 2013) presenting a hybrid PEA for the optimization of broker virtual machines subletting in cloud systems.

Regarding synchronicity, an EA can be synchronous or asynchronous. In a synchronous algorithm, when an individual is being accessed by a processing element, e.g., its fitness is being evaluated or its genetic material changed, this individual cannot be changed by any other processing element, e.g., the master node waits for all the worker nodes to finish evaluating the individuals. A synchronous master-slave EA has therefore the same properties as a serial EA, while reducing the execution time of the algorithm. According to one of most recent works on parallel evolutionary algorithms (Jakobović et al., 2014), the asynchronous PEA models have not been extensively investigated nor used in practice and are shown to be a viable alternative to traditional models. Also, asynchronous implementation of the island model is better suited for large-scale and heterogeneous computing architectures (Märtens and Izzo, 2013), e.g., computational grids such as the one used on this research. Therefore, the proposed model, presented on section 4.2, is an asynchronous one, unlike synchronous approaches that use migration as a synchronization point so that an island has to wait until all other islands have finished their generations to migrate and proceed.

Migration can be defined as the process by which the islands are able to exchange information. Migration topology is defined by the graph of inter-island links, i.e., the means of communication between the islands. Standard migration assumes exchanging *n* individuals (migration size) every *g* generations (migration interval or frequency). According to (Skolicki and De Jong, 2005), migration size and interval are arguably the two most important migration parameters, and (Tanese, 1989) performed some of the first studies about their impact.

Another important parameter is migration policy, which defines how to select migrators from the source island and individuals that will be replaced in the target island. A common migration policy consists in replacing the worst individuals in the target population by the best individuals from the source population (*best-worst policy*), which is the migration policy used in this work.

However, other migration policies may be used, such as *random-random,* i.e., random individuals are selected from the source island to replace random ones at the target island, and any other possible combination of worst, best or random individuals selection either at source or target island. As shown by (Cantú-Paz, 2001), random-random policy should not increase the selection intensity, and thus seems appropriate to be used when the investigation focus on other migration parameters than policy.

The migration topology describes the flow of individuals between the islands. Considering graph theory, an island PEA can be represented as a graph in which vertices correspond to islands and edges represent the migration flows of individuals. From this analogy to graph theory, one can design many possible topologies. The process of migration, by introducing new characteristics through the exchange of individuals between the islands, promotes the evolution of the subpopulations (Lopes et al., 2014). (Ruciński et al., 2010) presents a detailed study on the impact of the migration topology on the island model, comparing fourteen different migration topologies. However, Ruciński and colleagues' experiments were ran on an 8-core Apple OSX 64-bit server, using multiple threads to implement islands model, preventing us to reproduce their results since in this study a grid environment is used. Up-to-date literature on the Island Model and parallel optimization, commonly refer to static topologies as *Ring* (Figure *3.1.1*), *Torus*, *Cartwheel* and *Lattice* (Figure *3.1.2*), *Hypercube*, *Broadcast* and *Fully Connected* (Figure *3.1.3*) and Barabási-Albert (Figure *3.1.4*).

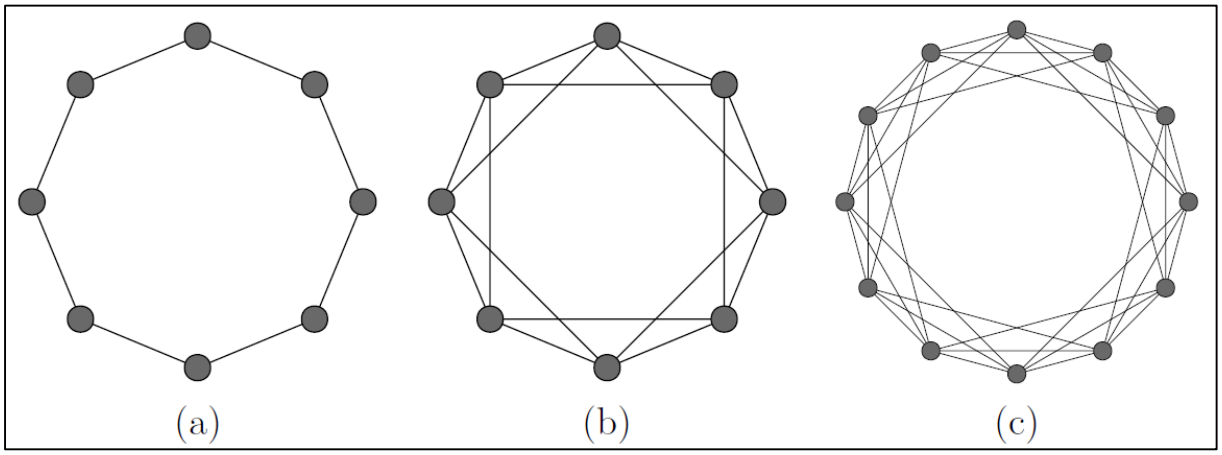


Figure 3.1.1: Ring (a), Ring+1+2 (b) and Ring+1+2+3 (c) topologies, in (Ruciński et al., 2010).

In the *Ring* topology (Figure *3.1.1a*) communication is allowed only between neighbour islands and may be either unidirectional or bidirectional. Extending the *Ring* topology, one can add edges that connect every second island (Figure *3.1.1b*), every third island (Figure *3.1.1c*) and so on. In our experiments, we only consider the unidirectional case for the basic Ring topology (See figure *3.1.1a*).

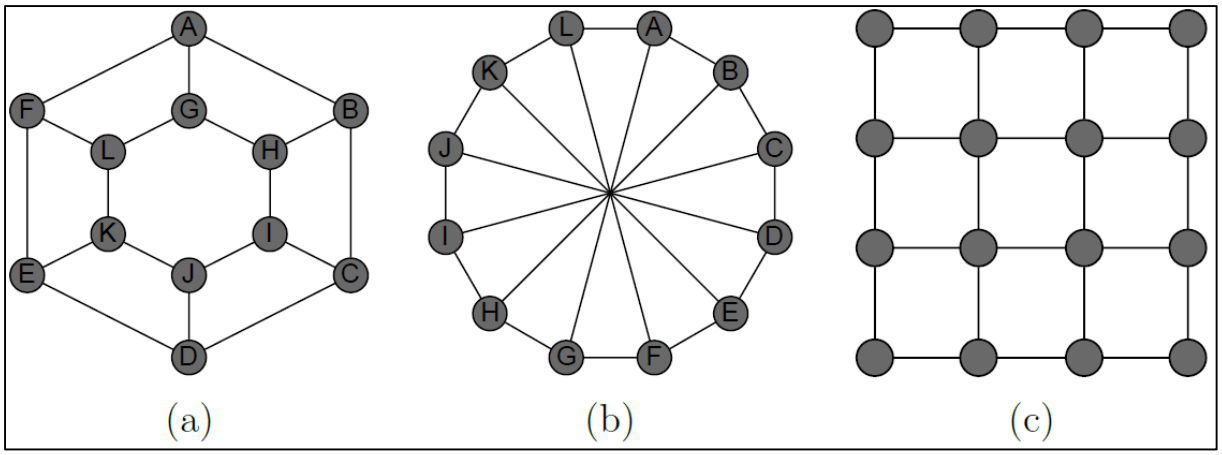


Figure 3.1.2: Torus (a), Cartwheel (b) and Lattice (c) topologies, in (Ruciński et al., 2010).

The *Torus* topology may appear in many variants, depending on the number and arrangement of "rings" and on the allowed paths of communication where some edges may be unidirectional while others bidirectional. Figure *3.1.2a* presents a generic variant of *Torus* topology, i.e. the one in which islands are distributed in two parallel rings with corresponding islands connected and with all the communication links bidirectional.

The *Cartwheel* topology (Figure *3.1.2b*) is a ring with additional edges connecting all pairs of islands laying on opposite "ends" of the basic structure. *Lattice* is a matrix like topology (Figure *3.1.2c*) where nodes are connected to the upper, lower, left and right neighbours, which is very popular in the context of fine-grained parallelization studies.

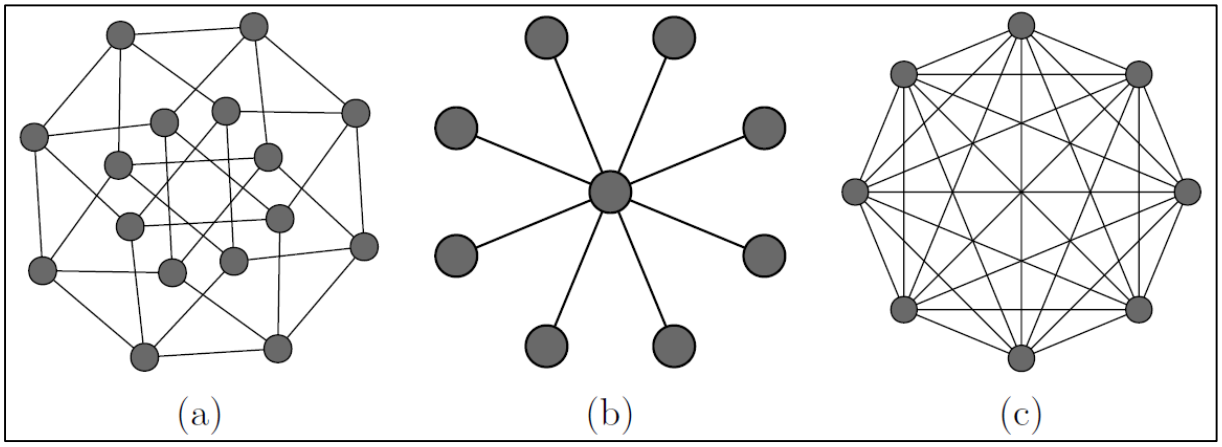


Figure 3.1.3: Hypercube (a), Broadcast (b) and Fully Connected (c) topologies,
in (Ruciński et al., 2010).

Hypercube topology (Figure *3.1.3a*) offers the best diameter (the maximum shortest distance between any pair of nodes) to number of edges ratio, with very low and homogeneous degree of connectivity (the number of edges incident to the node) what eliminates potential bottlenecks (Ruciński et al., 2010). This topology is obtained by putting islands in vertexes of a hypercube and routing connections according to the edges of this geometrical structure, implying that the number of islands must be powers of *2*. *Broadcast* topology (Figure *3.1.3b*), also known as *Star* topology, is rather associated with the Master-Slave model. The biggest problem in this kind of topology is that the central node may become either communicational or processing bottleneck. In the *Fully Connected* topology (Figure *3.1.3c*), all pairs of nodes are directly connected. While it presents the lowest possible diameter, its drawback is the quickly increasing number of connections, becoming an issue in applications where the number of nodes is high.

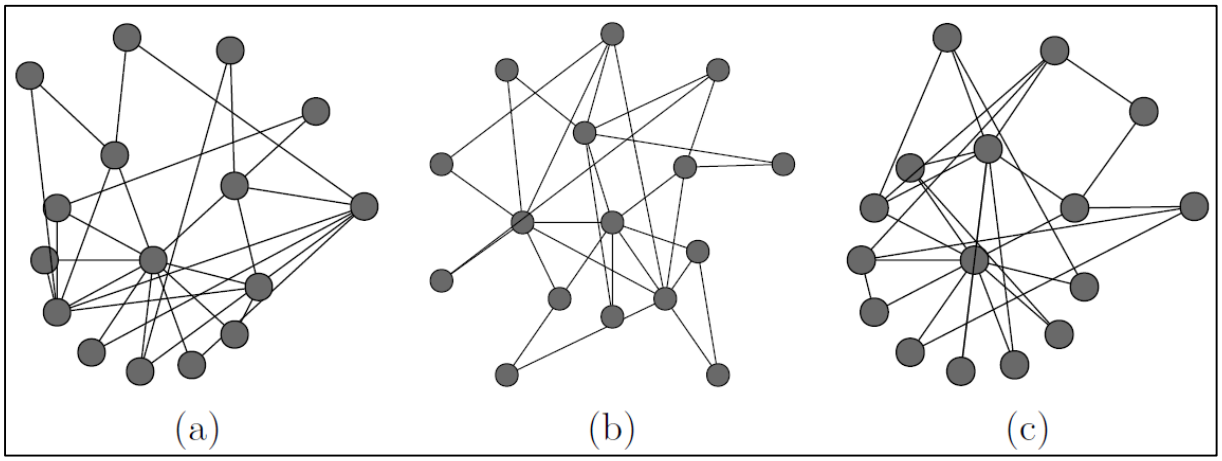


Figure 3.1.4: Three examples of Barabási-Albert topologies with 16 islands, in (Ruciński et al., 2010).

The *Barabási-Albert* (BA) model is an algorithm for generating random *scale-free networks* topologies are based on *scale-free networks*, i.e. networks where the number of nodes having a specified degree of connectivity follows a power plan, a property that is found in many natural phenomena such as the network of protein interaction in cells and the network of hyperlinks in the World Wide Web. In order to obtain a scale-free network, the BA model incorporates the features of incremental growth and preferential attachment. The algorithm that constructs the network has two parameters: initial cluster size m_0 and the number of links added at each step *m.* Figure *3.1.4* presents three example networks obtained using the BA algorithm with the same parameter values: $m_0 = 3$, $m = 2$.

All these topologies have a static configuration, i.e., the same pre-defined topology is maintained during the execution of the algorithm. (Lässig and Sudholt, 2013), (Lopes et al., 2013) and (Lopes et al., 2014) introduced random topologies and probabilistic methods of selection of migration flows, e.g., roulette wheel and tournament selection, into the comparison set of topologies. Concerning computing platforms used to run the experiments, (Tang et al,. 2004) and (Andalon-Garcia and Chavoya, 2012) compared the model performance using different migration topologies for a PGA implementation on a cluster platform, (Ruciński and Biscani, 2010) studied the impact of the migration topology on the island model running the experiments on an 8-core (Intel Xeon processors) Apple OSX 64-bit server. (Lässig and Sudholt, 2013), (Lopes et al., 2013) and (Lopes et al., 2014) do not specify the platforms that were used to run the experiments. To the best of our knowledge, the comparison of migration topologies using a grid computing framework, as proposed in this work, cannot be found in the literature.

3.2. A Review of Studies on Comparison of Island Model Topologies

Studies on PGA first appeared in the 1980's, cf. (Gordon and Whitley, 1993; Cantú-Paz and Meja-Olvera, 1994; Alba, 2005). Currently there is a vast number of works on PGA and in particular on IM. See (Knysh and Kureichik, 2010; Umbarkar and Joshi, 2010; Luque and Alba, 2011; Johar et al., 2013; Alba, Luque, and Nesmachnow, 2013) for some recent reviews on these works.

There are relatively less studies on the impact of the migration topology on the performance of PGA. In the own words of (Cantú-Paz, 1998): "A traditionally neglected aspect of parallel GAs has been the topology of the interconnection between demes". Actually, one of the earliest and more influential works on PGA suggests that "the choice of topology is not considered to be important. The network should have high connectivity and a small diameter to ensure adequate "mixing" as time progresses." (Cohoon et al., 1987).

This section focus on the studies on the comparisons of IM migration topologies. Migration topologies can be either static or dynamic. This work concerns only static topologies, i.e., topologies that are defined *a priori* and do not change during the whole optimization process. The reader interested in dynamic topologies, which are adjusted during the evolution process, is referred to (Candan et al., 2012; Lopes et al., 2013; Lopes et al., 2014).

Currently there are theoretical and experimental comparative studies on migration topologies. It seems consensual to credit the first theoretical study on PGA to (Pettey and Leuze, 1989) where a generalization of the schema theorem is derived to PGA where migrants are uniformly sampled from arbitrary neighbor islands at every generation. The first theoretical study on migration topologies is due to (Cantú-Paz, 1999) and is particularly relevant. The assumptions include islands running until convergence to a unique solution, exchanging migrants at that moment, and restarting afterwards. The main conclusion being that the quality of the results are independent of the topology as long as all islands use the same migration rates and the same number of neighbors. Unfortunately, some popular topologies do not meet the last criteria. For instance, that is the case of star and the cartwheel topologies.

Also, letting the islands run until convergence before allowing migration can be a strong assumption when the goal is the minimization of the completion time. The work of (Skolicki and De Jong, 2007) adopts the intra- and inter-island perspective to analyze the dynamics of the Island model. More recently (Lässig and Sudholt, 2013) contributed with a runtime analysis for a problem where a parallel (crossover less) evolutionary strategy requires polynomial time while the corresponding panmictic algorithm requires exponential time.

(Cantú-Paz and Meja-Olvera, 1994) seems to be the first of experimental comparative studies, in which it is shown that dense topologies (fully connected, 4-D hypercubes, toroidal mesh) are better than sparsely connected ones (unidirectional and bidirectional rings) in the sense that the former finds the global solution with fewer function evaluations. (Wang et al., 1998; Wang et al., 2005) use an order-based GA as the base GA for solving TSP instances in each one of the following five scenarios: i) Isolated islands with an equally divided population among the islands. Isolated island run in parallel without any communication. The optimization process stops when all islands stop. This is equivalent to run several time the base GA with different small size populations; ii) IM with migration. In this case an iterated ring is used where a copy of the local best individuals in island I are send to the island $(I + i) \bmod v$ (i refers to the iteration number and v to number of islands; iii) partitioned IM. The difference from i) is that now, the population in each island is disjoint from the population of any other island. Thus, each island focus on a small region of the search space; iv) IM with tour segmentation and recombination, i.e., each island evolves part of a TSP tour, and v) IM with both segmentation and migration. The studied topologies are homogeneous as all the islands are similar.

The main conclusion is that the scenarios involving migration operate in the shortest execution time. (He, Sýkora, and Salagean, 2006) resort to a parallel machine and to the standard Message Passing Interface (MPI) library to compare three types of topologies (linear array, Cartesian lattice, and random graphs) for each main type of PGA (master-slave, fine-grained and coarse-grained). In the last case (IM), periodical synchronization was considered. (Sekaj, 2004) presents a comparison of six topologies, each one with nine islands. Both unidirectional (star, ring, tree, and fully connected layers structure) and bidirectional (Cartesian lattices) were considered. Although both the homogeneous and the heterogeneous cases (different parametrization of the GA for different islands) were considered, no reference is made to synchronism or to deployment.

(Tang et al., 2004) studied two topologies: the unidirectional coordinated (rooted) ring, and the coordinated random topology, where at each migration instant an island receives migrants from one random island. The study concludes that the ring topology outperforms the random one in the tested instances of a quadratic assignment problem (QAP). (Andalon-Garcia and Chavoya, 2012) compared the star, the coordinated unidirectional and bidirectional ring topologies running in cluster platform through standard MPI, over a set of benchmark functions. The obtained results clearly show that the completion times decrease with an increase of the number of islands, up to a certain limit, and that it is more likely for the star to be the faster topology. (Guan and Szeto, 2013) propose a very interesting study on topology comparison. The study compares a continuous of topologies starting from a fully connected one and, and by systematically removing links, go over more and more sparse topologies until that the traditional ring is reached. The study concludes that for a set of four benchmark functions the optimal performance of the IM is obtained when the number of links ranges in 40-70% of the maximum number of links, i.e., the number of links in a fully connected topology. The study assumes that all islands are equally important leaving out coordinated topologies such as stars or trees. (Lopes et al., 2014) studied migrations topologies using Differential Evolution as the base evolutionary algorithm. The study compares two classic topologies, ring and star, with five stochastically generated dynamic ones. Unfortunately, no reference is made to synchronism or to deployment.

From the above review it is apparent that ring topology is, by far, the most studied one, both in PGA in general, and in topology studies in particular. It is also clear that no study either theoretical or experimental seems to be available that fully satisfy the premises of this work.

3.3. Deployment of PGAs in Grid Computing Frameworks

Concerning hardware and computer architecture, most of the existing parallel implementations of PGAs use either cluster or massively parallel processing (MPP) (Umbarkar and Joshi, 2013). According to the same authors, grid has many advantages over clusters such as no centralized control, security, access to distributed heterogeneous resources, and easy and reliable access to remote data sources and service to any available application. Using a grid (Kesselman and Foster, 1999) is both economically relevant and a technically challenging topic considering the potential heterogeneity of computer systems that compose a grid and the additional communication issues on this kind of distributed frameworks.

The need for High-performance computing (HPC) at a lower cost, led to the replacement of supercomputers by clusters. A cluster is a collection of parallel or distributed computers which are interconnected using high-speed networks such as Ethernet (Sadashiv and Kumar, 2011). These computers work together in the execution of intensive computing tasks, providing high availability, enabling a standby node when other node fails, and load-balancing by sharing the computational workload as a single virtual computer.

A grid is a form of distributed computing whereby a *super virtual computer* is composed of many networked, loosely coupled, and geographically dispersed computers acting together to perform large tasks. As proposed by (Chetty and Buyya, 2002), a computing grid may be analogously compared to the electrical power grid. In a grid, each computing node (Desktop and laptop computers, clusters, supercomputers, etc.) may be physically connected to any organization using conventional network hardware, thus being an effective way of sharing resources and solving problems in dynamic worldwide virtual organizations.

BOINC (Anderson, 2004), which stands for Berkeley Open Infrastructure for Network Computing, is a very good example of grid computing application. This open source middleware system was originally developed to support the *SETI@home* (Anderson et al., 2002) project but, at this moment, is used for other distributed applications in areas like mathematics, medicine, molecular, biology, and astrophysics. The main goal of *BOINC* is to make it possible for researchers to tap into the enormous processing power of personal computers from volunteer PC owners all around the world.

The challenges faced in grid computing (Sadashiv and Kumar, 2011) include:

a) *Dynamicity*: Resources in grid are owned and managed by more than one organization which may enter and leave the grid at any time causing burden on the grid;
b) *Administration*: To form a unified resource pool, a heavy system administration burden is raised along with other maintenance work to coordinate local administration policies with global ones;
c) *Development*: Problems are concerned with ways of writing software to run on grid computing platforms, which includes to decompose and distribute to processing elements, and then assembling solutions;
d) *Accounting*: Finding ways to support different accounting infrastructure, economic model and application models that can cope well with tasks that communicate frequently and are interdependent;
e) *Heterogeneity*: Finding ways to create a wide area data intensive programming and scheduling framework in heterogeneous set of resources;
f) *Programming*: The low-coupling between nodes and the distributed nature of processing increase the complexity of programming applications over grids when compared to a sequential programming. However, have less complexity then tightly-coupled parallel applications that can be ran on a cluster.

In parallel computing, both hardware and software issues must be considered. Hardware issues are directly related to parallel architectures, while software issues have to do with parallel programming models. Multiprocessor computers require shared-memory programming to run parallel programs, unlike distributed systems that use distributed-memory programming. PGAs can be implemented using low-level inter-process communications, e.g., *sockets* or *pthreads*.

At a higher level of abstraction, the most commonly used libraries for parallel implementations include MPI (Gropp et al., 1994) and MPI-2 (Gropp et al., 1999) for distributed memory platforms and OpenMP (Chapman et al., 2008) for shared memory architectures. When developing PGAs in *Java*, the most commonly used method to implement the communication and synchronization is the built-in remote method invocation (RMI) (Grosso, 2001) which enables the *Java* program to export an object that will be accessible across the network through a TCP port. Knowing that a grid is composed of loosely coupled, heterogeneous and geographically dispersed computers, this kind of environment is suitable for a loosely coupled model of parallelism, requiring little or no communication of results between tasks, e.g., communication of intermediate results.

Globus (Foster and Kesselman, 1997) is often referred as the *de facto* standard grid technology (Develder et al., 2012), providing support for security, information discovery, resource management, data management, communication, fault detection and portability (Alba, 2005).

Aiming the exploitation of large-scale availability of computing resources on grid computing platforms, a few parallel implementations of EAs, adapted to this environment, have been proposed. Concerning PEA models, these implementations include:

a) Master-slave model (Nebro et al., 2008; Durillo et al., 2008);
b) Distributed subpopulation model, i.e., island model (Melab et al., 2006; Talbi et al., 2007; Luna et al., 2008; Limmer and Fey, 2010) ;
c) Cellular model (Dorronsoro et al., 2007; Luque et al., 2009);
d) Hierarchical parallel models (Lim et al., 2007);
e) Parallel hybrid multi-objective evolutionary algorithms (MOEAs) (Tantar et al., 2007).

The main trend in the last years is using *Parallel Metaheuristics Frameworks*, instead of developing ad-hoc implementations of parallel metaheuristics. Due to its versatility for solving problems in several application domains, EAs have been the parallel metaheuristics of preference and therefore implemented in most of the parallel metaheuristics frameworks (Alba et al., 2013). Although about twenty of these frameworks have been proposed, the most relevant, which allow execution in grid infrastructures, are as follows:

a) *ParadisEO-CMW* (Cahon et al., 2005): An object-oriented framework, developed in C++, portable on Windows, Unix and MacOS, which provides support for parallel and distributed architectures using MPI and for grid computing systems such as *Globus*. However, its use is only intended for grids using *Condor* (Liu et.al., 2009) scheduling system and only implements *ring* migration topology for island EAs.

b) *Java Grid-enabled Genetic Algorithm (JG^{2}A)* (Bernal et al., 2009): An object-oriented framework, developed in *Java*, which provides support for the parallel execution of GAs in Globus-based grids. Like *ParadisEO-CMW* , this framework also imposes the use of *Condor* as the Local Resource Manager (LRM) (Thain et al., 2003) in both server and slave nodes. Concerning PEA models, this framework does not implement island models, being limited to a master-slave approach with a single population and a parallelization strategy based on the distribution of the fitness evaluations.

c) *pALS* (Bernal and Castro, 2010): acronym for *Parallel Adaptive Learning Search* is an object-oriented framework for the development of parallel and cooperative metaheuristics, developed in *Java*, which includes a module for execution of PGAs in Globus-based grids. Although *pALS* allows the implementation of island models, one could only find references to ring topology. Like the previous frameworks, *pALS* also imposes *Condor* as the resource manager.

d) (Limmer and Fey, 2010): The authors propose a framework which is presented as the first one for distributed EAs in generic *Globus* based computational grids. Unlike *pALS*, *JG^{2}A* and *ParadisEO-CMW*, this framework does not impose the use of *Condor* LRM, thus providing a more flexible solution. The parallelization strategies available on this framework are the distribution of all the fitness evaluations in a master-slave approach and a *ring* migration topology for island EAs, as shown in figure *3.3.1*.

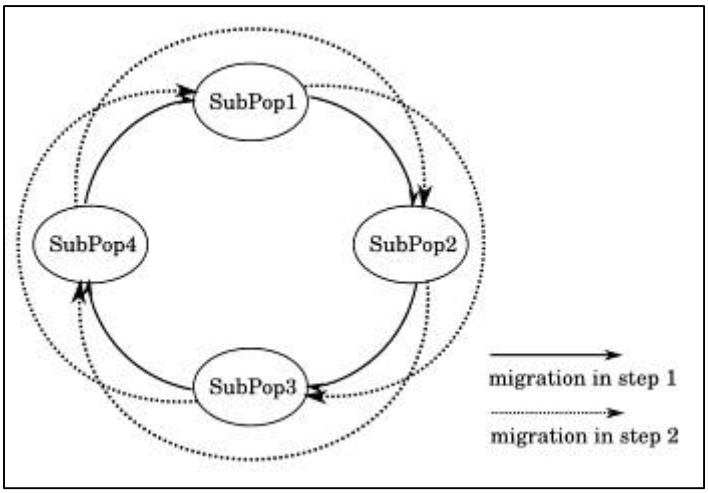


Figure 3.3.1: Ring migration topology, in (Limmer and Fey, 2010).

Considering the available literature, most of the existing frameworks which provide the implementation of distributed versions of EAs, i.e., island EAs, focus on ring topology. (Limmer and Fey, 2010) state that ring topology can be seen as the most suitable parallelization approach for the application in grids, although without presenting any comparison to other topologies.

To sum up, to the best of our knowledge there is a lack of investigation work concerning the comparison of different migration topologies, when implementing PEAs in heterogeneous asynchronous grid computing frameworks, such as *Globus*. Therefore, the current work presents a comparison of several migration topologies, using TSP as an application benchmark, without using any of the above frameworks.

3.4. A concurrent implementation of the Baseline Genetic Algorithm

There are two main questions that have driven a concurrent implementation, parallel or distributed, of the serial base algorithm, presented in section *2.2*. The first question refers to the synchronicity of the concurrent algorithm. The choice of an asynchronous approach is justified by the fact that a grid is composed of loosely coupled, heterogeneous and geographically dispersed computers. In this kind of computing environment, a typical synchronous approach, where a computing node waits for the previous one before proceeding its computation, is hardly suitable when the goal of the implementation is to speed up processing time. As the nodes in the grid are heterogeneous, with potentially significant different performances, the execution time of the algorithm would not be reduced and, in some cases, could even increase.

Furthermore, the computing nodes of a grid may belong to different worldwide institutions and, for that reason, can eventually add an additional communication overhead to the global processing time. If the synchronous case was to be considered, load balancing would be necessary for optimizing processing time. A periodically online process would be necessary to check the grid nodes availability and rate their performances. Based on that rate, the algorithm would distribute the computing effort by the grid nodes, according to their performance rate. Although this may be a viable technical solution to assure load-balancing, this method may fail because of the inexistence of a centralized control of the grid nodes, i.e. the performance rate of a grid node, at a given moment, is not necessarily maintained in time as the node's local users may launch any other processes that impact on the system performance. For these reasons, the implementation of the algorithm was definitively driven to an asynchronous approach.

The second question is related to the choice of migration topology when implementing island model. As reviewed in section *3.3*, most of the existing frameworks which provide the implementation of distributed versions of EAs, i.e., island EAs, focus on ring topology. Moreover, some authors, such as (Limmer and Fey, 2010), argue that ring topology can be seen as the most suitable approach for the application in grids, although without presenting a comparison to other topologies. Considering a ring topology, as presented in figure *3.3.1*, how would the algorithm perform if one, or more, computing nodes fails or if it has a considerable lower performance when compared to the other nodes? This work argues that, in that case, the performance of the algorithm could be negatively affected, namely in terms of the quality of the results, if the algorithm is asynchronous, as the grid node will not benefit from the neighbor's migrant chromosomes. If the synchronous case is considered, the execution time can also increase as a node may be waiting for a slow neighbor before proceeding its computation. A coordinated topology may thus provide a better fault-tolerance to this kind of situations. In coordinated island models, as proposed in this work, the grid nodes periodically share their best solutions with their parent and/or sibling nodes, providing population diversity and preventing local optima. On the other hand, child nodes are expected to provide quality solutions for their parents which will need less generations to reach their best solution to the problem and, thus, reducing the processing time of the algorithm. Moreover, the obtained experimental results, discussed in chapter *4*, show that adding more nodes to the coordinated topology, allows obtaining significant gains in terms of processing time.

The concurrent implementation proposed in this work may be classified as an asynchronous coordinated genetic algorithm, implementing an island model based genetic algorithm. The concurrent algorithm is distributed over a set of grid computing nodes. (Marin et al., 1994) proposed a centralized master-slave scheme in which slave processors execute a GA on their population and periodically send their best partial results to the master. Then, the master chooses the fittest individuals, found by any processor, and sends them to the slaves. (Cantú-Paz, 1999) proposed a coordinated topology for GAs with multiple populations, introducing a tree representation of the extended neighbourhood of root deme (*0*), as shown in figure *3.4.1*.

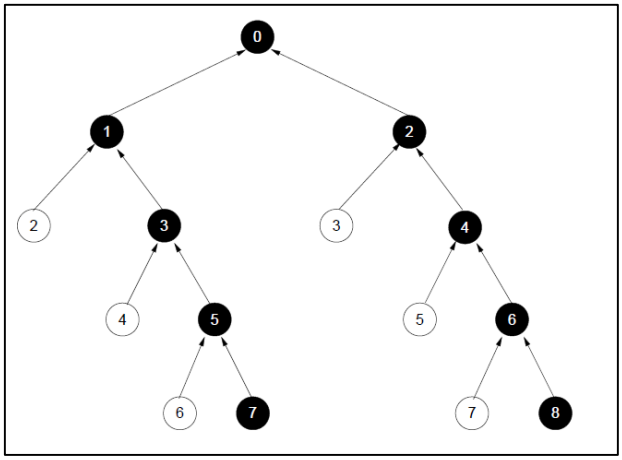


Figure 3.4.1: Tree representation of the extended neighbourhood of root deme, in (Cantú-Paz, 1999).

Some years later, (Sekaj, 2004) proposed a 3-level migration topology where each third-level node shares its results with all the second-level nodes, as shown in figure *3.4.2*.

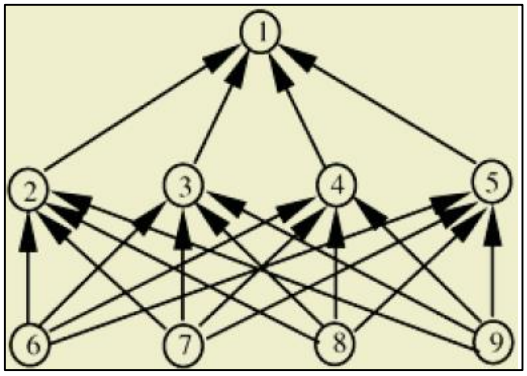


Figure 3.4.2: *3*-level PEA topology, in (Sekaj, 2004).

More recently, (Filipowicz et al, 2011) presented a study using a similar topology, with four levels. However, their work was published in Polish, in their university journal, and an English version could not be found, except for the abstract of the document. The analysis of Filipowicz and colleagues' work could be very interesting and some of its results could eventually be compared with the ones produced in this thesis, as they both use the Travelling Salesman Problem (TSP) as real-world application of PGAs. However, as far as one can understand, using available online translation tools, the work of Filipowicz and colleagues does not specify implementation details such as the used hardware infrastructure preventing its reproduction and consequent fair comparison to the current investigation work. On the other hand, it is clearly possible to understand that the author only ran the experiments five times for each pair of TSPLIB dataset/migration topology, for small instance, i.e., up to 159 cites (TSPLIB *u159* data set), and did not provide any statistical validation of the results. Therefore, although one could not ignore that work, knowing that the experiments should be ran at least 30-100 times (Alba et al., 2013) and a statistical assessment of the results should be performed, e.g., by an analysis of variance (ANOVA) test, no significant conclusion can be drawn from it.

According to the EAs' models presented on section 2.1, the tested topologies in this research can be classified as structured EAs, more precisely uniform distributed EAs, as they deal with isolated subpopulations. Each computing node runs the same EA, as well as the evolutionary conditions, e.g., selection and recombination methods, are the same for all individuals and all subpopulations. Each node loads its initial subpopulation randomly from the global population and the subpopulation size is determined by the position of the computing node on the tree and by its number of child nodes.

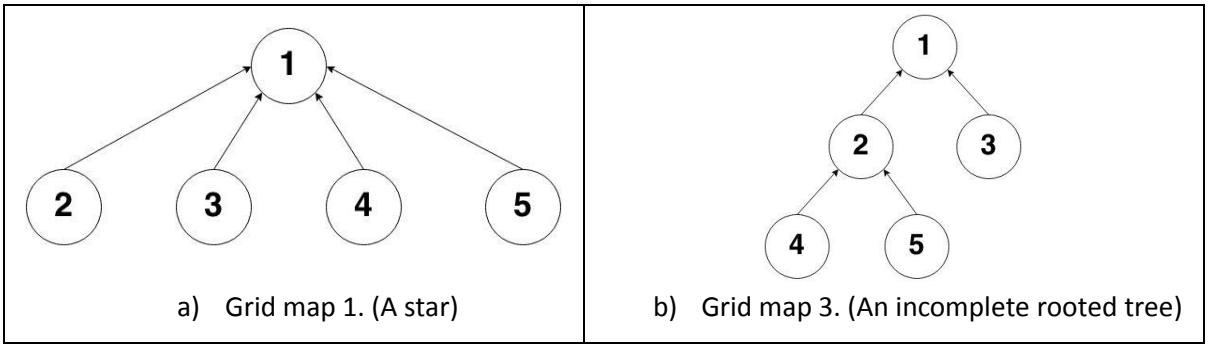
To the best of our knowledge, studies on coordinated topologies are limited to the above mentioned ones.

In principle, the depth of the tree levels in a coordinated topology is limited only by the number of available nodes and by the size of the population. Currently, the work is limited to *7* processing nodes. In future work considerations, presented on section *5*, a new topology is proposed, combining a coordinated topology and multithreading, that can virtually provide an unlimited number of computing nodes.

In order to clarify some concepts, let us analyse the topologies that were experimentally tested in this research. These topologies, presented in tables *3.4.1* and *3.4.2*, were called *grid maps.* Each grid map (GM) specifies a different island topology used to run the algorithm and is physically defined by a comma-separated-value (CSV) file. The original algorithm was thus adapted to dynamically change its behaviour, assuring automation and minimizing the user interaction in the process of producing experimental results. For this purpose, a set of scripts were developed to automate the execution of the experiments. For instance, the command line "*./launch_tsp_sharing a280 2 False*" launches the experiments for the *TSPLIB* data set with *280* cities (*a280*), telling the algorithm to migrate periodically 2 chromosomes not randomly chosen, i.e. choose the best 2 chromosomes to migrate. If the last parameter is "*True*", then the migrators will be randomly chosen. Tested TSPLIB data sets were *eil51* (*51* cities), *st70* (*70* cities), *eil76* (*76* cities), *eil101* (*101* cities), *ts225* (*225* cities) and *a280* (*280* cities).

The algorithm runs *100* times each possible configuration. Programmatically, possible run configurations are defined by a set of nested loops: a first loop for the grid maps, a second one for the number of generations before migration (*epoch*) and a last loop for the population distribution technique.

Initially, the grid computing test bed only had *5* nodes, as shown in figure *3.5.1*. The first experiments were performed on that test bed, for *GM1* and *GM3* which are presented in table *3.4.1*.

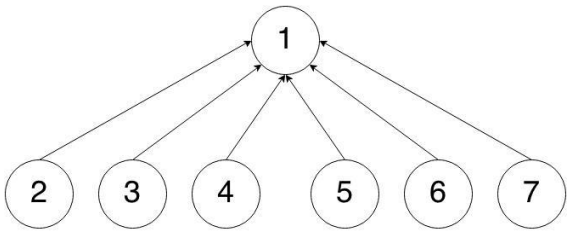


a) Grid map 1. (A star)

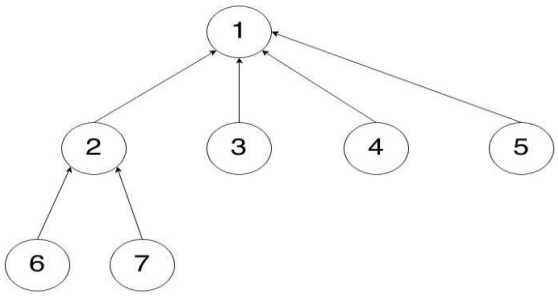
b) Grid map 3. (An incomplete rooted tree)

Table 3.4.1: Tested topologies (*grid maps*) on the *5* nodes test bed.

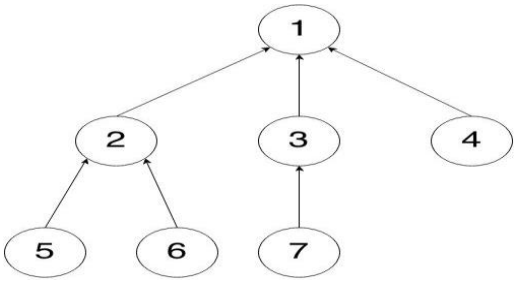
For the *7* nodes test bed, grid maps vary from *1* to *9*, as presented in table *3.3.4*, possible epoch values are *g* = {*1*,*2*,*3*,*4*,*5*,*10*,*20*,*50*,*100*,*1000*} and population distribution (*pd*) may be *Equally-Distributed (E)* or *Tree-Distributed (T)*. In the case of an equally-distributed population, each grid node will deal with an initial subpopulation of size *N/7* where *N* is the global population size. For tree-distributed population, considering the example of grid map *2*, nodes {*1*, *3*, *4*, *5*} will load an initial subpopulation of size *N/5* and nodes {*2*, *6*, *7*} will start with a population of size *(N/5) / 3*, i.e., *N/15*. Notice that for *GM1*, *E* technique results in the same distribution as *T*. In fact, these techniques produce different population distributions only for trees with more than 2 levels such as *GM2*. Therefore, a possible run configuration is a combination of a grid map, an epoch and the population distribution technique, e.g., *GM = 1* / *g = 2* / *pd = E*.



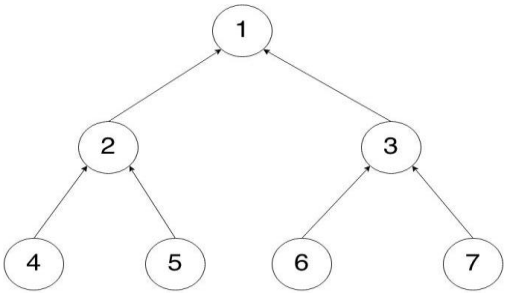
a) Grid map 1. (A star)



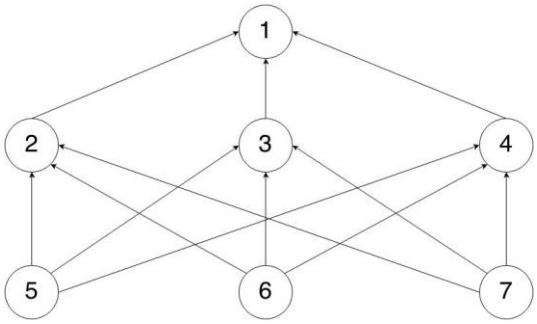
b) Grid map 2. (An incomplete rooted tree)

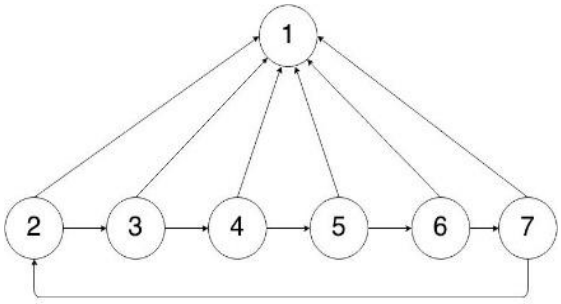


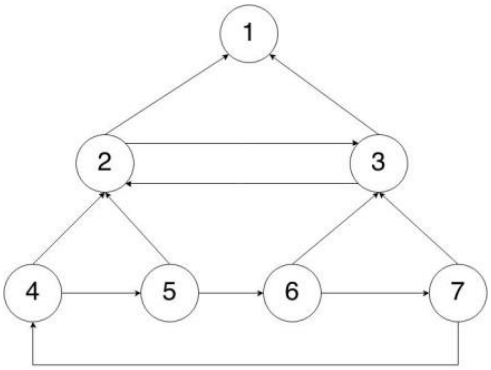
c) Grid map 3. (An incomplete rooted tree)

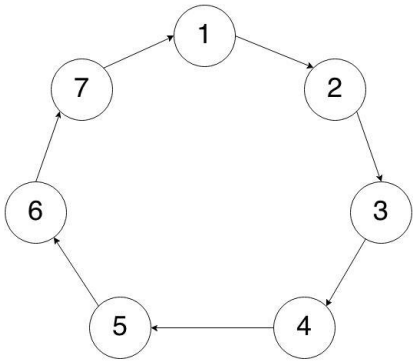


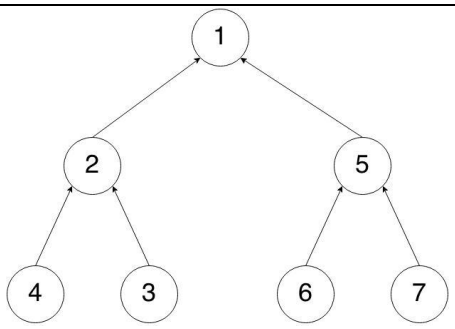
d) Grid map 4. (A complete rooted binary tree)

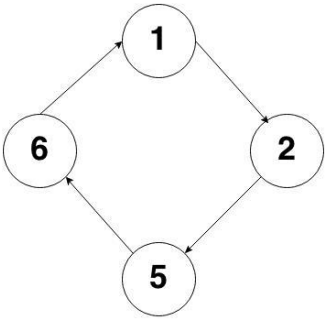
<table>
<tr>
<td> 

e) Grid map 5. (Three-fully connected layers)</td>
<td> 

f) Grid map 6.(A Cartwheel)</td>
</tr>
<tr>
<td> 

g) Grid map 7. (A rooted tree-ring)</td>
<td> 

h) Grid map 8. (A ring)</td>
</tr>
<tr>
<td> 

i) Grid map 9. (A complete rooted binary tree)</td>
<td> 

j) Grid map 10. (A ring, only with the fastest nodes)</td>
</tr>
</table>

Table 3.4.2: Tested topologies (*grid maps*) on the *7* nodes test bed.

In grid map *1*, at every epoch, the root node will benefit from the solutions shared by its child nodes, i.e., all remaining nodes. In the case of tree-distributed population, all computing nodes load an initial subpopulation of size *N/7* where *N* is the global population size.

As for grid map *2*, the root node will benefit from the solutions shared by nodes *2, 3, 4* and *5*. In the same way, node *2* will receive the shared results from its child nodes, i.e., nodes *6* and *7*. In this case, node *2* acts simultaneously as a parent and child node. For tree-distributed population, nodes *1, 3, 4* and *5* will load an initial subpopulation of size *N/5*. Nodes *2, 6* and *7* will start with a population of size *(N/5) / 3*, i.e., *N/15*.

In the case of grid map *3*, the root node will benefit from the solutions shared by nodes *2, 3,* and *4*. In the same way, node *2* will receive the shared results from its child nodes, i.e., nodes *5* and *6*. At last, node *3* will benefit from the solutions shared by node *7*. If population is tree-distributed, nodes *1* and *4* will load an initial subpopulation of size *N/4*. Nodes *2, 5* and *6* will start with a population of size *(N/4) / 3*, i.e., *N/12*. Finally, nodes 3 and 7 will have an initial subpopulation of size *(N/4) / 2*, i.e., *N/8*.

Grid map *4* represents a complete binary tree topology, where all nodes, except the root one, have parent nodes. Root node will benefit from the solutions shared from nodes *2* and *3*. Node *2* will receive the shared results from nodes *4* and *5* and node *3* will benefit from the solutions shared by nodes *6* and *7*. In the case of tree-distributed population, the root node will load an initial subpopulation of size *N/3* and all remaining nodes will start with an initial subpopulation of size *(N/3) / 3*, i.e., *N/9*.

In grid map *5*, each mid-level node (*2, 3* and *4)* receives, every epoch, the migrators from nodes *5, 6* and *7*. The root node receives the migrators from the nodes *2, 3* and *4*. For tree-distributed population, root node has a population of size *N/4*, nodes *2, 3* and *4* have a population of size *(N/16)* and nodes *6, 7* and *8* have a population of size *(N/16) * 3*.

Grid map *6* implements a ring topology, although limited to the bottom level of the tree. In terms of distribution of the population is identical to grid map *1*.

In the same way, grid map *7* is similar to grid map *4*, however ring topology is implemented in both middle and bottom levels of the tree.

Grid map *8* corresponds to a typical ring topology where migration between islands occurs to the immediately clockwise neighbour.

Finally, grid map *9* is similar to grid map *4*, except for the position of the worker nodes in the complete binary tree, i.e., the fastest nodes appear in the higher levels of the tree. Root node (sbgrid1), which is the fastest grid node, maintains its isolated top level position. The intermediate level is occupied by nodes *2* and *6* and the bottom-level by nodes *5, 3, 7 and 8*.

Grid map *10* corresponds to a typical ring topology as for *GM8,* but only using the fastest nodes from our test bed. The detailed system specifications of each grid node are presented in section *3.5*.

In order to insure fair comparisons with the serial version of the algorithm, *each one of the tested topologies* was developed using the baseline GA, tested on the same set of traveling salesman problem instances, and started running with the same set of initial populations, as (Wang et al., 1998) did on his research which compared several PGAs using TSP as a benchmark function. This means that, for each of the *100* runs of the serial algorithm, for each TSPLIB data set, the initial population is stored to be later used by the corresponding iteration of the concurrent run. For instance, every different configuration for the 10^{th} run of for the *a280* data set, will use the same initial population as the 10^{th} run of the serial version of the algorithm did, for the same data set (*a280*).

The concurrent implementation uses mostly the same parameters as the serial algorithm does. In each generation, the best solution is compared with the best solution from the previous generation. The algorithm stops when the best solution remains unchanged for *s* consecutive generations. Every generation, after checking the best solution age, the second half, i.e., the worst half of the population is removed. Then for as many iterations as half the initial population size, the following process is repeated:

a) Two parents are randomly selected;
b) Four children (children 1, 2, 5 and 6) are obtained by cloning both parents and two children (children 3 and 4) by mating using *GreedyCrossover*;
c) Children 1, 2, 5 and 6 are then mutated, with a probability of mutation *Pm=0.25*, by random permutation;
d) All new children are optimized by the *2-opt* heuristic method and finally added to the existing population.

After this set of iterations, the algorithm checks for migration files from the other nodes. When available, migrant individuals are added to the current population. The resulting population is once again sorted by fitness evaluation (from smaller distance to larger) and the worst chromosomes, that exceed the initial population size, are removed from the population in order to maintain the initial population size. Finally, if the current generation corresponds to the defined migration epoch, the fittest individuals are sent to the parent node(s), if any.

```
Set the population size N, the number of migrants k, and the migration frequency g;
Randomly initialize population;
Sort population by fitness (from smaller distance to larger);
i := 0;
repeat
    for j := 1 to N/2 do
        Randomly select parents p1 and p2 from the best half of the population;
        child1 := Permutation (Clone (p1));
        child2 := Permutation (Clone (p2));
        child3 := Crossover (p1; p2);
        child4 := Crossover (p2; p1);
        child5 := Permutation (Clone (child3));
        child6 := Permutation (Clone (child4));
        Apply 2opt method to all children;
        Add all children to the population;
    end
    [When available] Add received migrants to the current population;
    Sort population by fitness (from smaller distance to larger);
    Remove the last (worst) individuals beyond the initial population size;
    i := i + 1;
    if (i mod g == 0) then
        Send the k first (best) individuals to parent node(s) [if any];
    end
until no change in the best individual for a LIMIT (s);
```

Figure 3.4.3: Pseudo-code for the concurrent version of the GA.

As one may observe in figure *3.4.3*, the main differences between the sequential and concurrent versions of the algorithm are the inclusion of the features that allow chromosomes migration and their integration on the target subpopulation. Considering a tree topology, the root node is the top-level one and only has the migrant chromosomes integration feature. The remaining nodes (worker nodes) may have both migration and integration features, depending on the topology. Concerning migration process, additional parameters are required, namely migration frequency and migration size. Migration frequency defines how frequently chromosomes migrate, i.e. the number of generations before migration (epoch). Tested epoch values were {*1,2,3,4,5,10,20,50,100,1000*}. Migration size specifies the number of migrant chromosomes, i.e. the number of chromosomes to be migrated every epoch. (Starkweather et al., 1991) and (Limmer and Fey, 2010) argue that migration size should be rather small.

Every epoch, each worker node creates a CSV file, containing their migrant chromosomes, and sends it to the corresponding parent node(s), which include those chromosomes in their population. The file communication is performed using *globus-url-copy* command, a Globus scriptable command line tool. At each generation, the grid nodes check the working directory for the existence of migrant chromosomes files. If exists, they will integrate those chromosomes in their population and, if not, they proceed their computation without any waiting time, regarding the asynchronous characteristics of the concurrent algorithm.

3.5. Grid Computing Test Beds

Grids are often constructed using general-purpose grid middleware software. Globus® Toolkit (Foster and Kesselman, 1997) and Glite (Sipkova et al., 2006) are two of the most used open source middleware solutions. Unlike Glite, Globus® Toolkit is compatible with several Linux distributions, having a specific package for Debian Linux, the selected operating system for this project. For these facts, Globus® Toolkit was the selected middleware for this project grid construction. For the purpose of this project, two test beds were fully installed and set up, with *5* and *7* nodes respectively. All grid nodes, on both test beds, were formatted and installed with the current latest versions of *Debian Linux (7.4)* operating system and Globus® Toolkit *(5.2.5)*.

The first test bed, presented in table *3.5.1*, contains a total of *5* grid nodes, i.e., the root node and *4* worker nodes.

Name	Type	Processor Model	CPU (MHz)	Memory (kB)
sbgrid1	Root	Intel(R) Pentium(R) 4	2800	1.034.224
sbgrid2	Worker	Intel (R) Celeron (R)	2400	709.108
sbgrid3	Worker	Pentium III (Coppermine)	997	384.344
sbgrid4	Worker	Pentium III (Coppermine)	650	384.412
sbgrid5	Worker	AMD Athlon(TM) XP 2200+	1800	774.132

Table 3.5.1: *5* nodes test bed based on Globus® Toolkit Middleware.

The second test bed, presented in table *3.5.2*, contains a total of *7* grid nodes, i.e. the root node and *6* worker nodes.

Name	Type	Processor Model	CPU (MHz)	Memory (kB)
sbgrid1	Root	Intel(R) Pentium(R) 4	2800	1.034.224
sbgrid2	Worker	Intel (R) Celeron (R)	2400	709.108
sbgrid3	Worker	Intel Pentium III (Coppermine)	997	384.344
sbgrid4	Worker	Intel Pentium III (Coppermine)	650	384.412
sbgrid5	Worker	AMD Athlon(TM) XP 2200+	1800	774.132
sbgrid6	Worker	Intel(R) Pentium(R) 4	1700	254.472
sbgrid7	Worker	Intel Pentium III (Katmai)	500	514.460

Table 3.5.2: *7* nodes test bed based on Globus® Toolkit Middleware.

All the nodes are connected to the same switching equipment of our local area network (LAN), using conventional 10/100 Mbit/s Ethernet network cards. The proposed test beds demonstrate why grid computing is an economically attractive solution when compared to Cluster and MPP. As shown at image *3.5.1*, one can build a grid computing framework using out-of-date computers that, when working together, may provide a relevant processing power to run PGAs.

Image 3.5.1: A picture of our grid computing framework, built with out-of-date computers.

CHAPTER 4: An Experimental Study on Selected IM Topologies

This chapter presents, an experimental comparative study between serial and distributed versions of the genetic algorithm for a carefully selection of island model topologies. In order to assure a fair comparison, the serial runs of the algorithm were performed on the fastest grid node, i.e., node *1* (root node). The appendix *A* contains all the figures corresponding to the complete set of performed analysis. Besides the data being analyzed (execution time, fitness or number of generations), each figure also contains the *p* and *F* values from the analysis of variance (ANOVA) test and the *p* value from the Kruskal-Wallis test, assuring the statistical validation of the results. The default value for *p* was set to *0.05*, i.e. providing a confidence interval of 95%. If $p < 0.05$ then the differences in the results are statistically significant, otherwise those differences are not statistically significant. When the differences in the results are not statistically significant, it briefly means that there is no significant difference between the data series being analyzed.

4.1. GA Tuning – Experimental choices

As previously explained in section *3.3*, each possible configuration, i.e., combination of grid map, number of generations before migration and population distribution technique, was run *100* times. Grid maps vary from *GM1* to *GM9*, possible epoch (*g*) values are {*1,2,3,4,5,10,20,50,100,1000*} and population distribution technique (*pd*) may be *Equally-Distributed (E)* or *Tree-Distributed (T)*. Notice that, as far as this investigation goes, testing *g=1000* is equivalent to test the algorithm with no migration topology, knowing that even in the worst cases, the total number of generations never reaches that value and thus migration of chromosomes will never be performed in *g=1000* configurations.

As stated in section *3.4*, migration size specifies the number of migrant chromosomes, i.e. the number of chromosomes to be migrated every epoch. Some authors, such as (Starkweather et al., 1991) and (Limmer and Fey, 2010) argue that migration size should be rather small. In fact, one could experimentally conclude that, as far as this investigation goes, *n=2* is a suitable value.

4.2. GA Performance Measure

Table *4.2.1* presents the comparison of the experimental results (serial and concurrent versions) with *TSPLIB* known optimal solutions. As one can observe, the average fitness, obtained by both the serial and concurrent versions of the GA, is near the best known solutions provided by *TSPLIB*. Also, the concurrent version of the algorithm allows to significantly reduce the execution time of the algorithm, while maintaining the quality of the results.

	Serial GA		***Concurrent GA***			
Data set	*Avg. Time (s)*	*Avg. Min. Fitness*	*Configuration*	*Avg. Time (s)*	*Avg. Min. Fitness*	*Best known fitness (TSPLIB)*
a280	793	2587	GM1-g4	139	2588,6	2579
ts225	407	126645	GM1-g4	61,4	126649	126643
eil101	35,4	641,18	GM4-g4	4,68	643,2	629
eil76	12	544,09	GM1-g4	1,17	544,89	538
st70	7,9	677	GM1-g5	1	677,5	675
eil51	3,09	428,3	GM2-g3	0	428,81	426

Table 4.2.1 Comparison of the experimental results with *TSPLIB* known optimal solutions.

Before starting discussing the experimental results, it is important to state that GAs' performance is often measured by the number of times the evaluation (fitness) function is invoked by the algorithm. Knowing that, in our case, the evaluation function is called in every generation, the performance of the algorithm should be measured by the product of the total number of performed generations and population size. However, one could observe that, in a few cases, there is no direct relationship between the execution time and the total number of generations. An example of this fact can be observed in figures *4.2.1* and *4.2.2*, for the *ts225 TSPLIB* data set (*225* cities). Considering this fact, comparison of topologies is based on execution time of the algorithm, in order to answer the initial research question. As for fitness, the goal of the algorithm is to minimize the fitness of the solution, in order to find the shortest path to the travelling salesman.

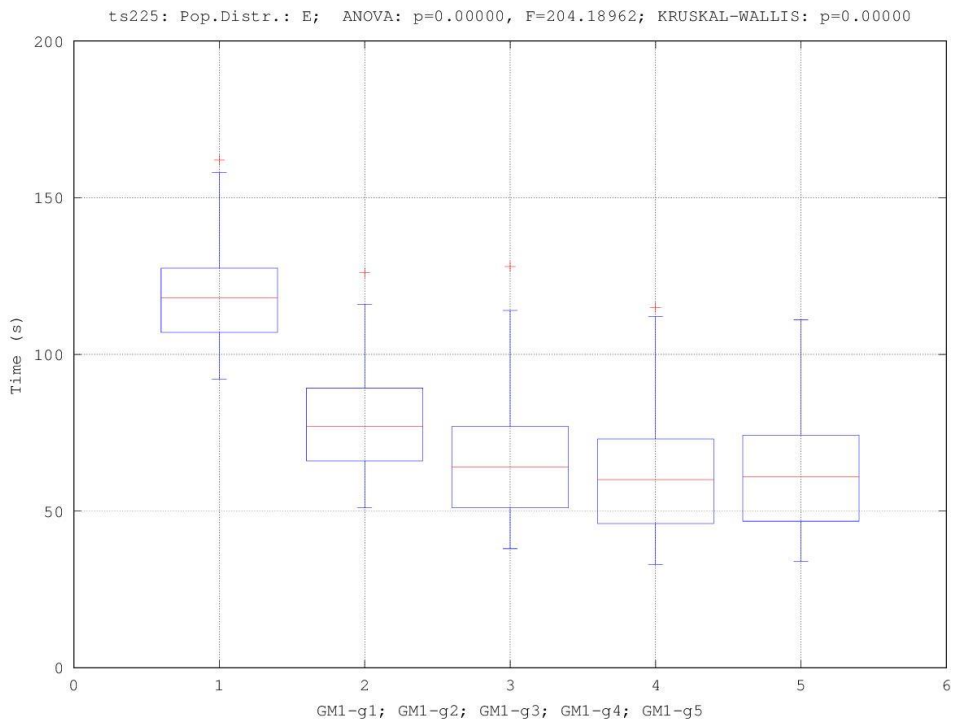


Figure 4.2.1: Execution time of *GM1*, for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

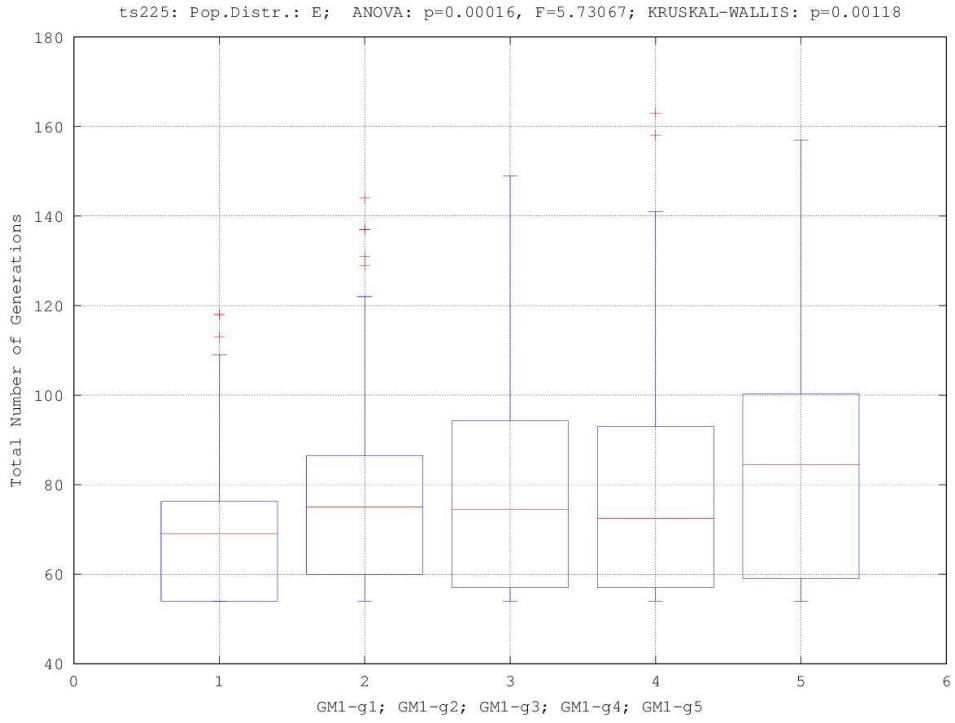


Figure 4.2.2: Total number of generations of *GM1*, for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

4.2.1 Speedup

An issue related to topology is the concept of speedup. In deterministic parallel computing, speedup is defined as $S = Ts/Tp$ where Ts is the completion time of the best sequential algorithm and Tp is completion time of the parallel algorithm when ran over p processors. This definition is not directly applicable to the island model. On one hand, PGA are stochastic optimization algorithms meaning that completion times can be different for different runs of exactly the same algorithm. Also, it is not acceptable to compare times for algorithms whose produced solutions have different fitness, as it can happen in successive runs of the same algorithm. On the other hand, due to migration, an island model based PGA is intrinsically different from a GA. Several different variant definitions have been proposed for PGA, e.g. (Cantú-Paz, 2000; Alba and Tomassini, 2002; Alba, Nebro, and Troya, 2002; Lim et al., 2007; Luque and Alba, 2011; Jakobović, Golub and Čupić, 2014).

Given the types of topologies analyzed (asynchronous with heterogeneous processors) and the stochastic optimization technique employed, the following is proposed:

Definition 1. The expected root speedup (ERS) is *ERS = E(Tpanmictic) / E(Tparallel)* where *E(Tpanmictic)* is the average execution time as measured in the fastest node with the total population of size *N*; and *E(Tparallel)* is the average execution time as measured in the same node acting as root in a topology where *N* is equally divided among the *v* nodes, given that:

a) All other experimental conditions are equal (e.g., same problem instance, same parameters and same stop criteria);
b) There is no statistically significant difference in the fitness of the solutions obtained in the panmictic and parallel cases.

According to (Alba, 2005), this definition corresponds to a w*eak speedup* which compares the parallel algorithm developed by a researcher against his own serial version.

One remark is in order here. In a sense, the proposed definition is a competitor of the definition proposed in (Jakobović, Golub and Čupić, 2014). In that work, the definition involve a certain acceptable quality level (fitness) that the obtained solutions are supposed to reach before times can be measured. This is an extra user-defined, problem specific, parameter that the analyst need to worry about. In this respect, the proposed definition is independent of the studied problem or its instances. Table *4.2.1.1* presents the ERS results for the *star* topology, ran on the 7 nodes test bed. No statistically significant difference was found for the other studied topologies.

Instance (Reinelt, 1991)	E(Tpanmictic)	E(Tparallel) – Star topology	ERS
eil51	3.1	0	-----
st70	7.9	1.0	7.9
eil76	12	1.17	10.26
eil101	35.4	4.68	7.56
ts225	407	61.4	6.63
a280	793	139	5.71

Table 4.2.1.1: Expected Root Speedup (ERS) for the *star* topology with 7 nodes.

Speedup can be *sub-linear* ($ERS < v$), *linear* ($ERS = v$), and s*uper-linear* ($ERS > v$). In short, the sources for super-linear speedup, observed in the *st70*, *eil76* and *eil101* data sets, are (Alba, 2002):

- The higher chances of finding an optimum by using more processors, due to the intrinsically heuristic multipoint nature of PGAs;
- Splitting the global large population into smaller subpopulations that fit into the processor caches provides faster algorithms than using a single main memory;
- The operators work on much smaller data structures, and they do so in parallel, which is an additional source of speedup.

The minimum fitness (distance), obtained with both the panmictic and the PGA versions for the star topology with optimized migration frequency, is near the optimal solution. The minimum average fitness obtained by the sequential algorithm has a systematic difference which is less than 2% from the optimal solution for each one of the tested instances. The times of the PGA are significantly lower than the corresponding panmictic ones. Also the expected root speedup decreases with the size of the problem (number of cities).

4.3. Comparing the 5 nodes and 7 Nodes Test Beds

A first important statement is that, for both 5 and 7 nodes test beds, concurrent versions run much faster than the serial version of the algorithm, independently of the tested topology and the *TSPLIB* data set. This fact can be observed in Figures *4.3.1* and *4.3.2* for the *5* nodes and *7* nodes test beds, respectively.

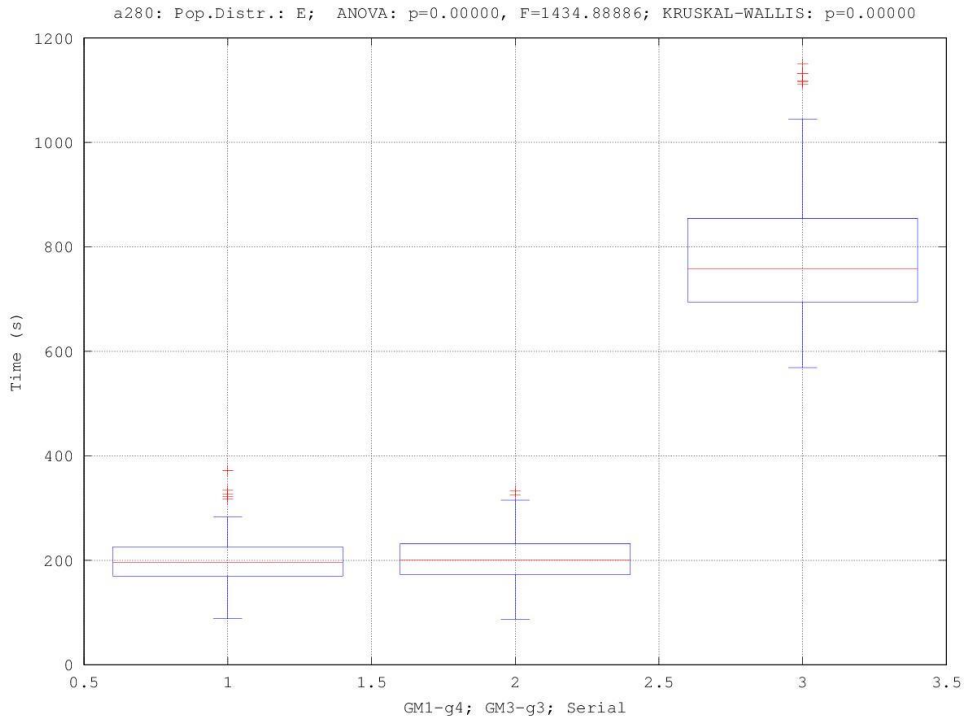


Figure 4.3.1: Execution time of best configurations on the 5 nodes test bed, for the *a280* data set (Nodes: 5; Pop.: 56 per node).

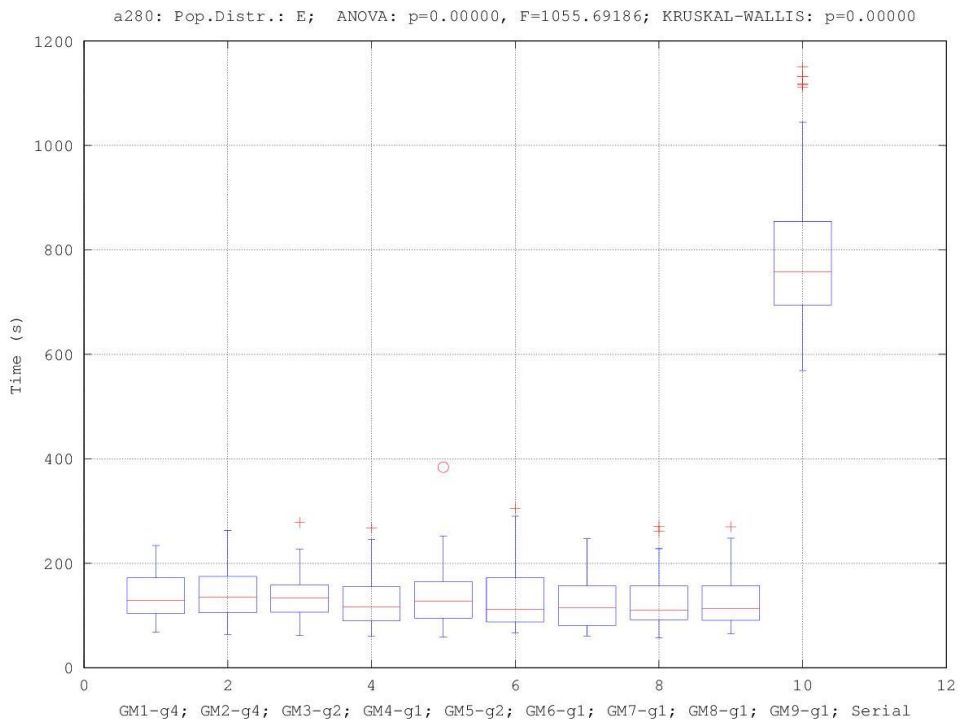


Figure 4.3.2: Execution time of best configurations on the 7 nodes test bed, for the *a280* data set (Nodes: 7; Pop.: 40 per node).

The first experiments were performed on the *5* nodes test bed, presented in table *3.5.1* and were limited to *GM1* and *GM3* for the a280 *TSPLIB* data set (*280* cities). Inserting two additional nodes to the grid, originated the *7* nodes test bed, presented in table *3.5.2*, in which several additional grid maps were tested. The grid map sets for the *5* nodes and *7* nodes test beds are presented in tables *3.4.1* and *3.4.2*, respectively.

Studying the impact of the number of grid nodes running a given topology, it was possible to observe that, for the *a280* data set, adding *2* nodes to the grid test bed allows to obtain significant gains in terms of processing time either in *GM1* (*star* topology) or in *GM3* (incomplete rooted tree topology). Figure *4.3.3* shows statistically significant differences in the results ($p < 0.05$) when comparing execution time of the best configuration of GM1 for both *5* and *7* nodes. Comparing fitness results, showed in figure *4.3.4*, one can observe that the differences in the results are also statistically significant, with a value of *p* very close to *0.05* (*0.04*). This means that additional nodes allowed to significantly reduce the execution time, without a relevant degradation of fitness.

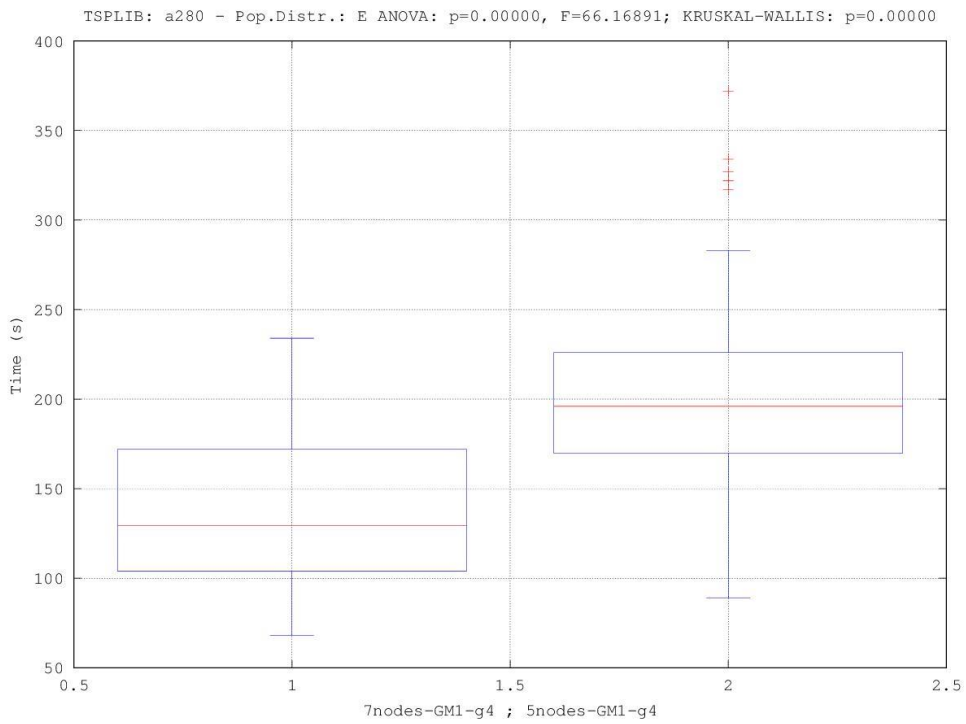


Figure 4.3.3: Execution Time of best *GM1* configuration for *5* and *7* nodes test beds (Nodes: 5; Pop.: 56 per node / Nodes: 7; Pop.: 40 per node).

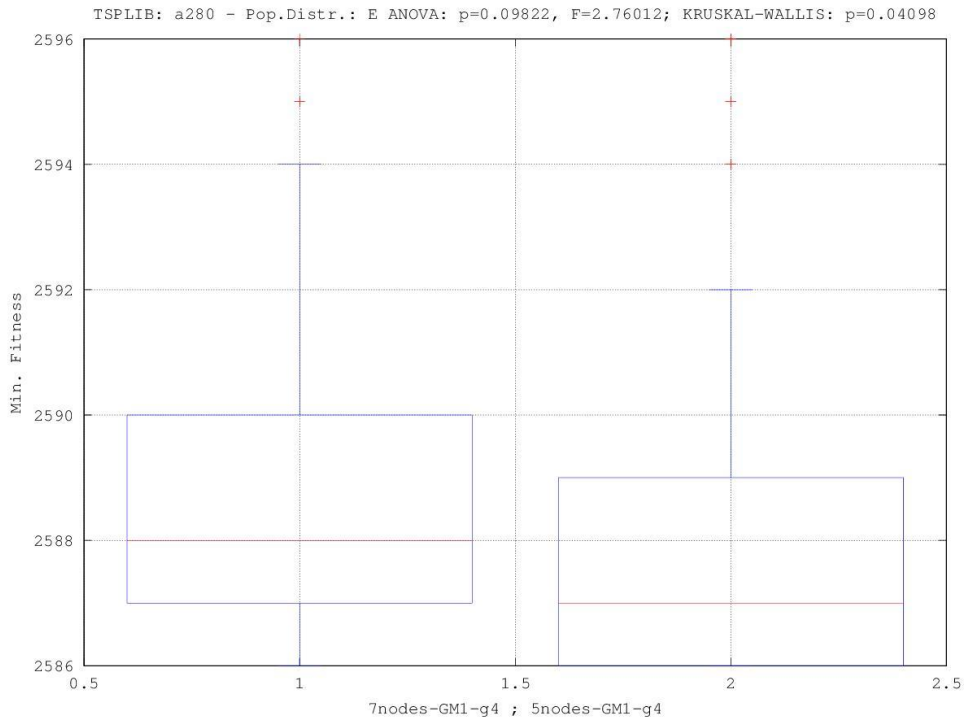


Figure 4.3.4: Minimum fitness of best *GM1* configuration for *5* and *7* nodes test beds (Nodes: 5; Pop.: 56 per node / Nodes: 7; Pop.: 40 per node).

Repeating the same analysis for *GM3*, figure *4.3.5* shows statistically significant differences in the results ($p < 0.05$) when comparing execution time of the best configuration of *GM3* for both *5* and *7* nodes. On the other hand, comparing fitness results, showed in figure *4.3.6*, one can observe that the differences in the results are not statistically significant ($p >= 0.05$). This means that additional nodes promoted a significant reduction on the execution time, without a significant degradation of fitness.

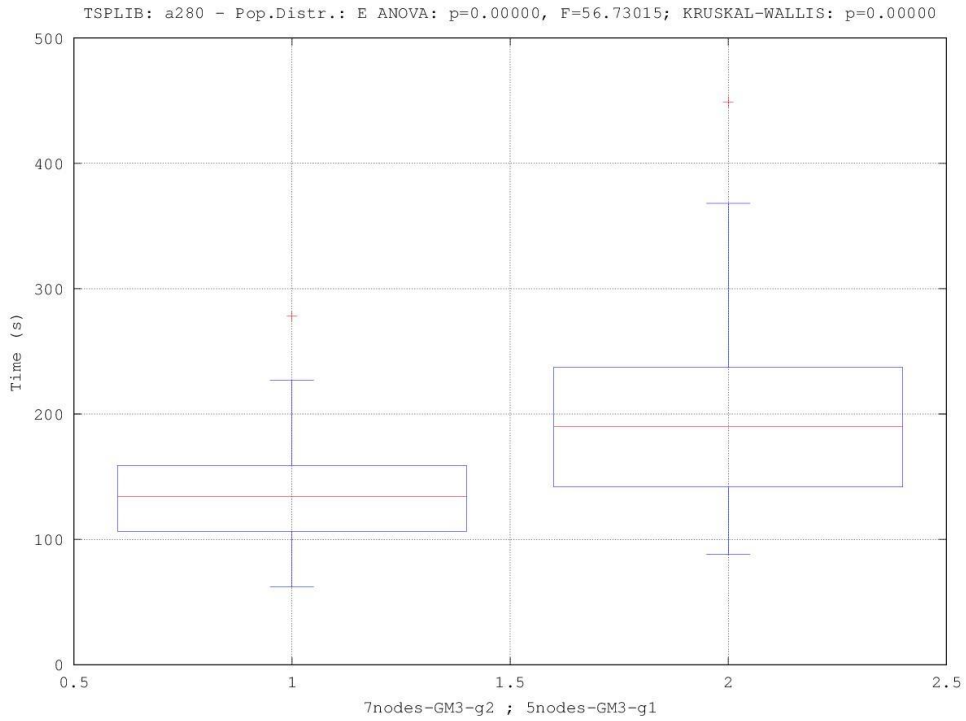


Figure 4.3.5: Execution Time of best *GM3* configuration for *5* and *7* nodes test beds (Nodes: 5; Pop.: 56 per node / Nodes: 7; Pop.: 40 per node).

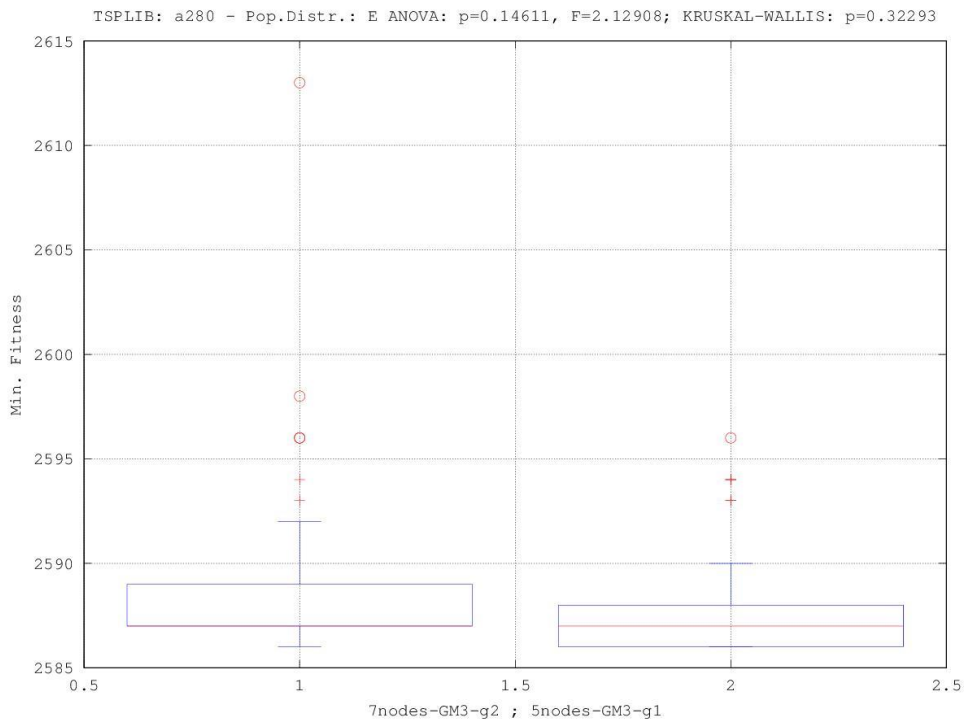


Figure 4.3.6: Minimum Fitness of best *GM3* configuration for *5* and *7* nodes test beds (Nodes: 5; Pop.: 56 per node / Nodes: 7; Pop.: 40 per node).

The observation of figures *4.3.3* to *4.3.6* allows one to conclude that additional grid nodes promoted a significant reduction on the execution time, without a significant degradation of fitness for both *GM1* (*star* topology) and *GM3* (incomplete rooted tree topology). Considering the utility of the additional grid nodes, all further results, and correspondent discussion, will therefore refer to the *7* nodes test bed.

4.4. Population Distribution Technique

Using the *7* nodes test bed, an additional variable was introduced, related with the population distribution technique. The goal was to test if different population distribution techniques would impact on the performance of the algorithm. Figures *4.4.1* and *4.4.2* show an example, for the eil101 *TSPLIB* data set (101 cities), that demonstrates that, in fact, the way the population is distributed through the islands does impact on the performance of the algorithm. According to the population distribution techniques presented on section 3.3, each node will process a population of *N/7* individuals, in the case of the Equally-distribution technique (E). As for the Tree-distribution technique (T), considering GM4 (a complete rooted binary tree), the root node will have a population of *N/3* individuals and all other nodes will have a population of *N/9* individuals.

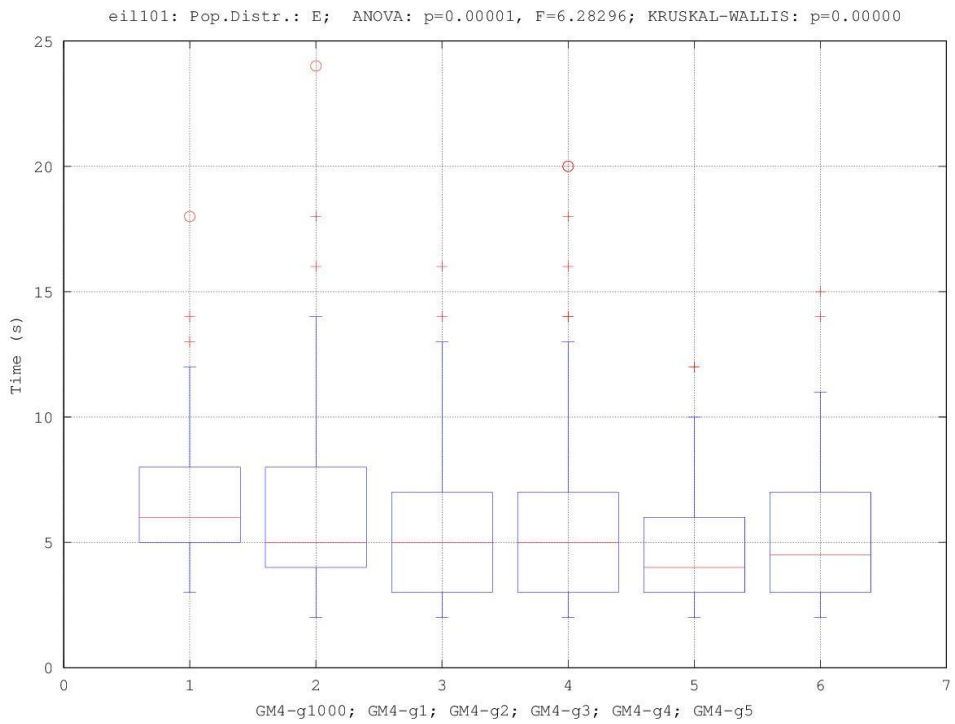


Figure 4.4.1: Execution time for *eil101* data set (*GM4*), using Equally-Distributed Population (E) (Nodes: 7; Pop.: 14 per node).

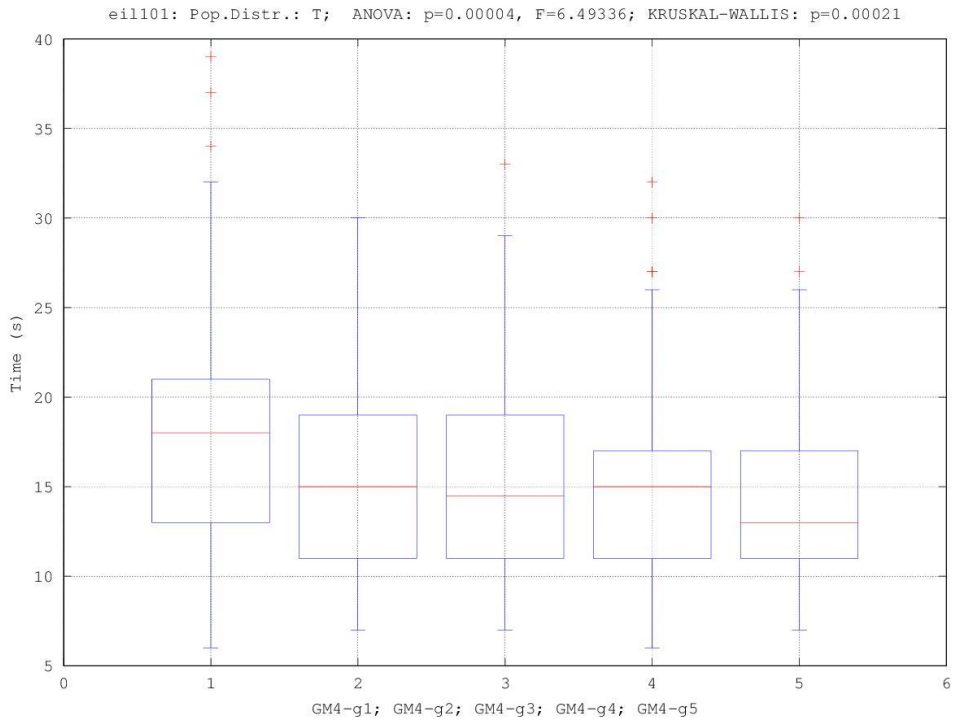


Figure 4.4.2: Execution time for *eil101* data set (*GM4*), using Tree-Distributed Population (T) (Nodes: 7; Pop.: 33 for node 1 / 11 for nodes 2, 3, 4, 5, 6 and 7).

For all the tested data sets (*eil51*, *st70*, *eil76*, *eil101*, *ts225* and *a280*), a similar behaviour was repeatedly verified for *GM1* to *GM5* and *ep* = {*1*, *2*, *3*, *4*, *5*} for each *GM*. One could observe that the algorithm performed systematically faster in the cases where the population is equally-distributed (E), rather than tree-distributed (T). Possible explanations to this fact are that in the case of tree-distributed population:

a) Root node has a larger population;
b) Worker nodes have a smaller population and, for that reason, produce lower quality solutions;
c) A combination of both a) and b).

Keeping in mind the initial research question, the results presented and discussed in this chapter will thus refer to equally-distributed population (E).

4.5. Serial vs Grid Results

It is also important to present the comparison between the serial (or sequential) version of the GA and its concurrent version. In order to do so, the serial version was compared with the fastest configurations for each grid map (GM). These configurations, presented in table *4.5.1*, were selected using the following criteria:

1) Configuration with the lowest execution time;

2) Configuration with the lowest fitness, in case of tie in 1);

3) Any configuration, in case of tie in 2).

Data set	***GM1***	***GM2***	***GM3***	***GM4***	***GM5***	***GM6***	***GM7***	***GM8***	***GM9***	***GM10***
a280	g=4	g=4	g=2	g=1	g=2	g=1	g=1	g=1	g=1	g=5
ts225	g=4	g=4	g=2	g=1	g=2					
eil101	g=5	g=5	g=5	g=4	g=5					
eil76	g=4	g=3	g=2	g=2	g=5					
st70	g=5	g=3	g=5	g=2	g=5					
eil51	g=3	g=3	g=3	g=3	g=5					

Table 4.5.1: Fastest configurations for the tested topologies, using 7 nodes.

As an example of how this table is constructed, consider *GM1* and dataset *a280*, and fig. *4.5.1* below. From this figure, it is clear that for the *a280* data set, g=4 yields the best results.

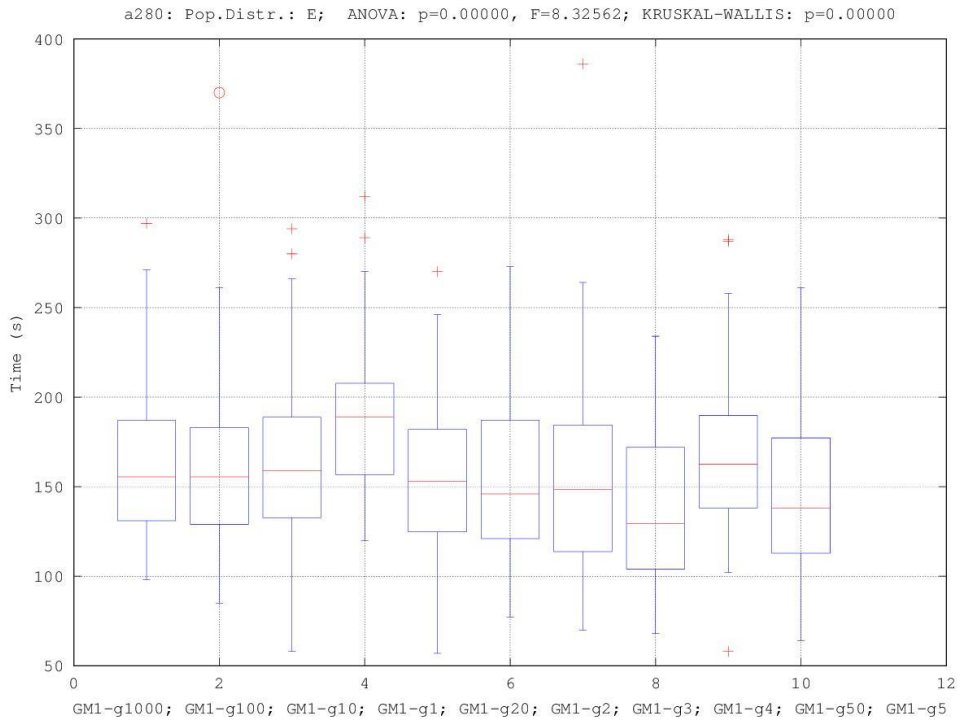


Figure 4.5.1: Execution time of GM1, for the *a280* data set (Nodes: 7; Pop.: 40 per node).

Figures *4.5.2* to *4.5.7* demonstrate the usefulness of deployment on a grid environment, showing significant gains in terms of execution time, for every TSPLIB data set that was tested. In all these figures, the x-axis has the fastest grid map/epoch combination, i.e., the configuration with better performance for the specified data set, as detailed in table *4.5.1*. For instance, in figure *4.5.2*, *GM1-g3* corresponds to the *grid map 1/epoch 3* configuration (*3* generations before migration), which is the fastest configuration for the *eil51* data set.

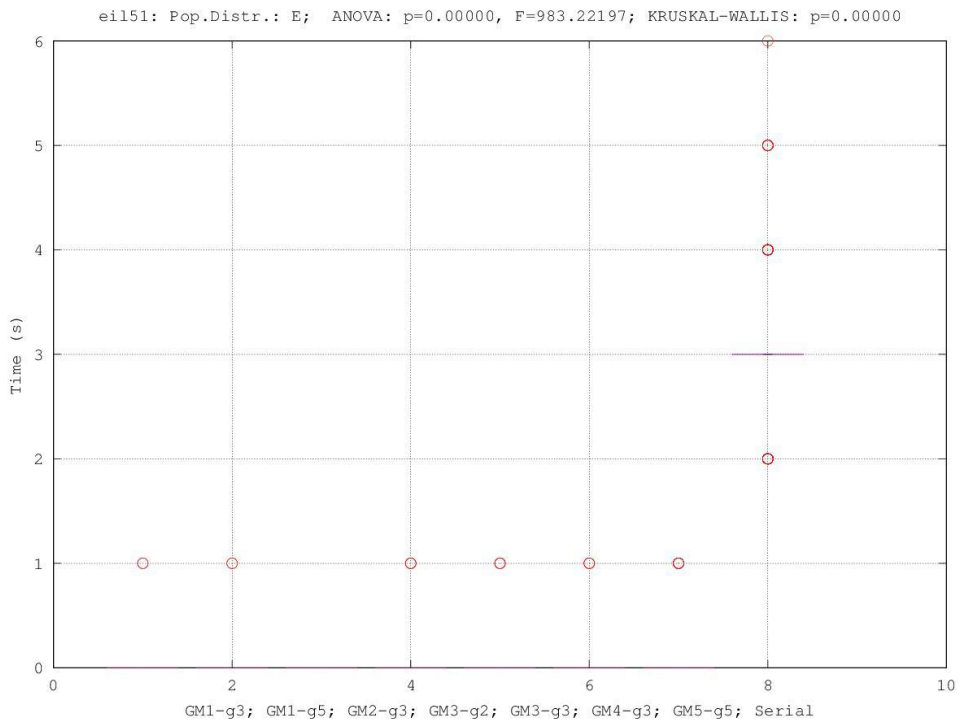


Figure 4.5.2: Execution time for the *eil51* data set (Nodes: 7; Pop.: 7 per node).

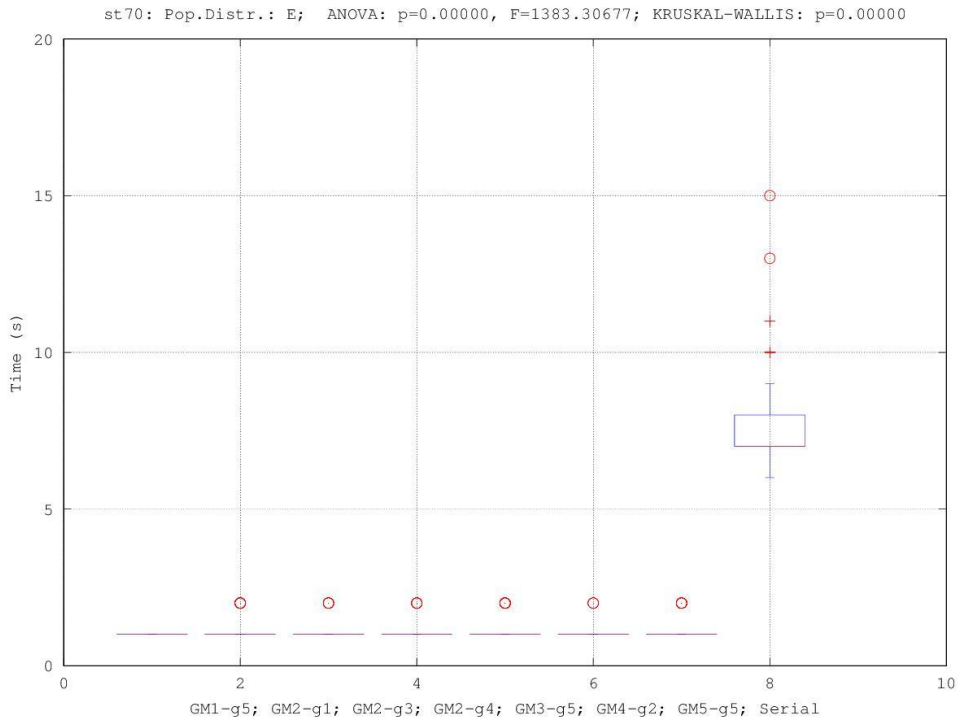


Figure 4.5.3: Execution time for the *st70* data set (Nodes: 7; Pop.: 10 per node).

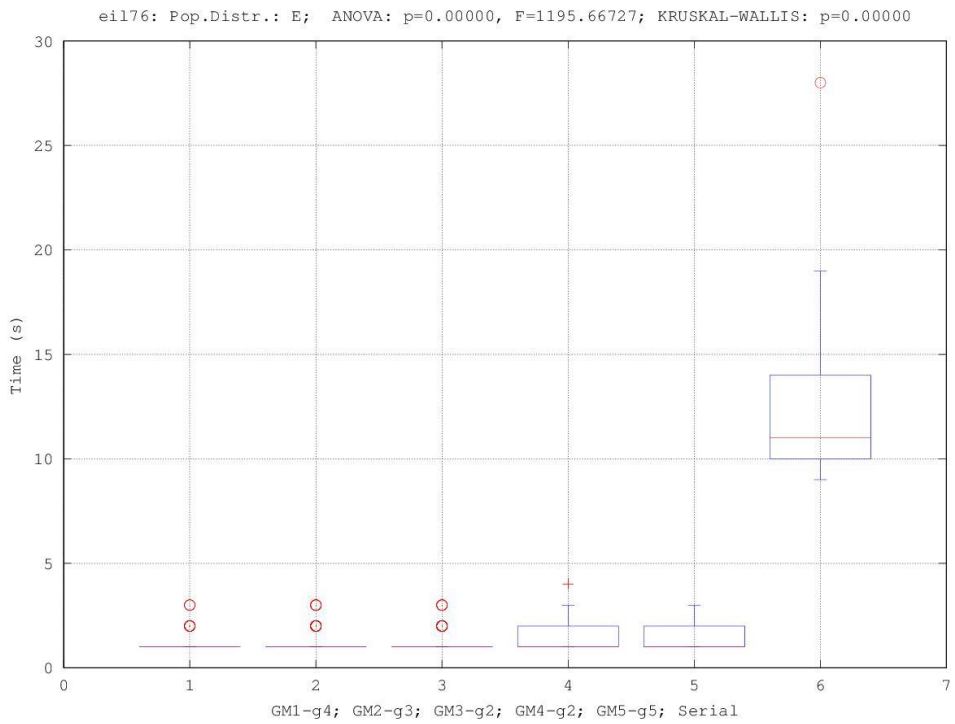


Figure 4.5.4: Execution time for the *eil76* data set (Nodes: 7; Pop.: 10 per node).

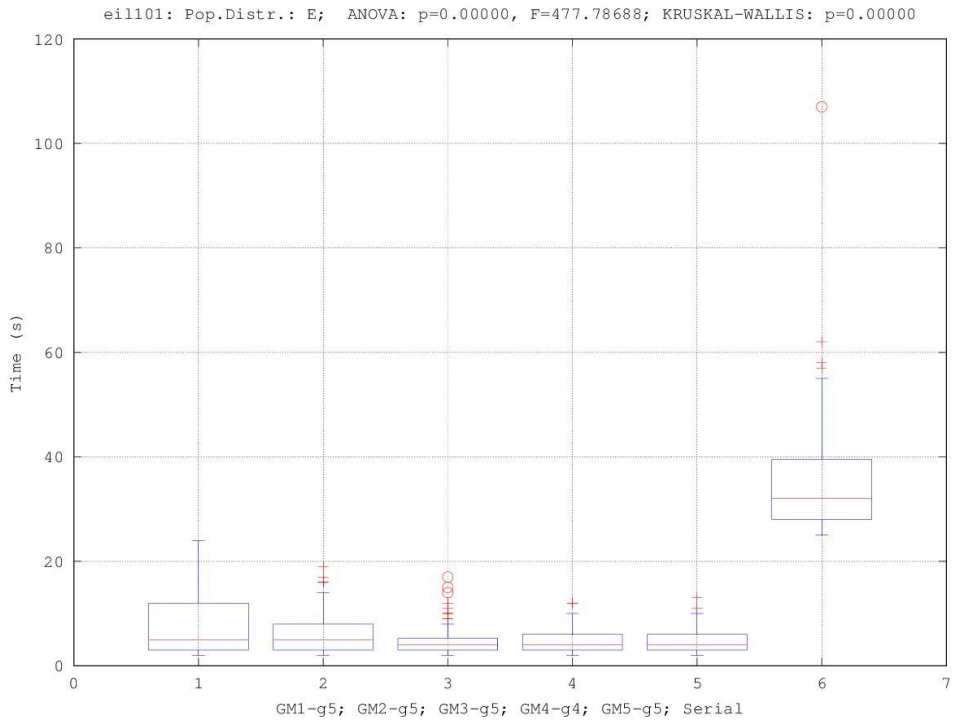


Figure 4.5.5: Execution time for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

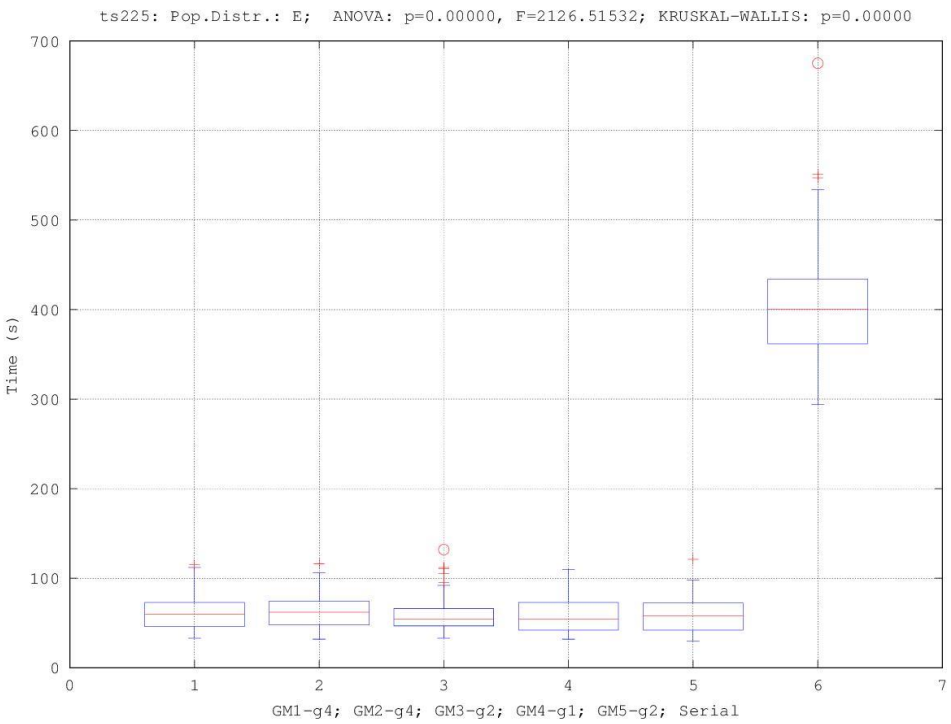


Figure 4.5.6: Execution time for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

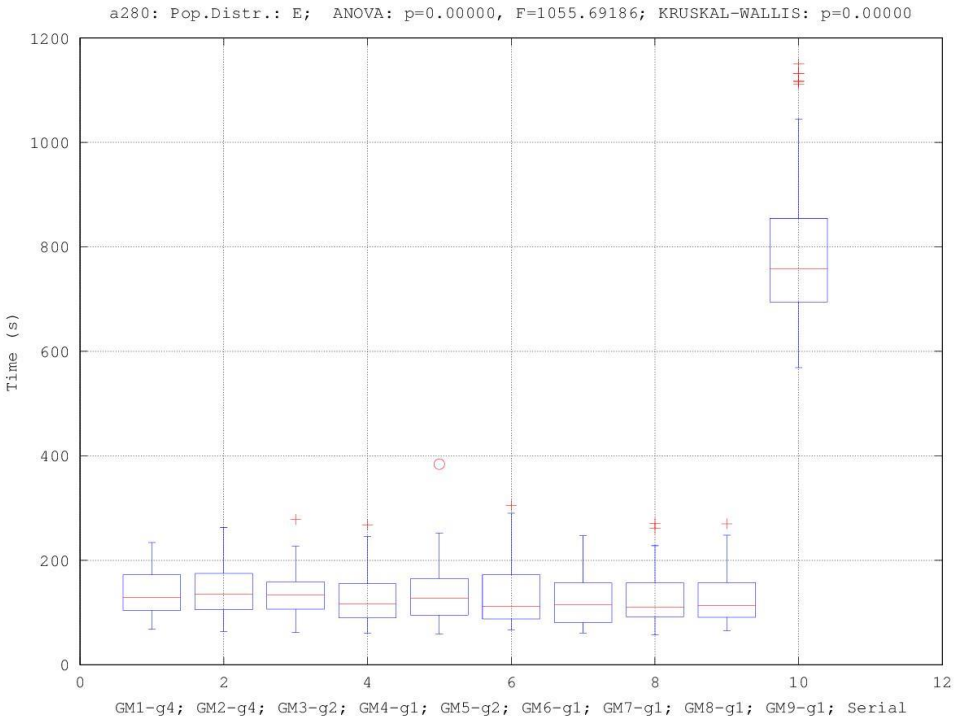


Figure 4.5.7: Execution time for the *a280* data set (Nodes: 7; Pop.: 40 per node).

Observing figures *4.5.2* to *4.5.7*, one can conclude that the differences between execution times, for different topologies, get more accentuated with the size of the problem, i.e., as the number of cities increases. Another conclusion is that distributing the computing effort by several grid nodes, the performance is much better than the serial implementation, no matter the chosen topology.

As for fitness, the goal of the algorithm is to minimize the fitness of the solution, in order to find the shortest path to the travelling salesman. Figures *4.5.8* to *4.5.13* present a similar analysis for fitness values, for the same TSP instances presented before.

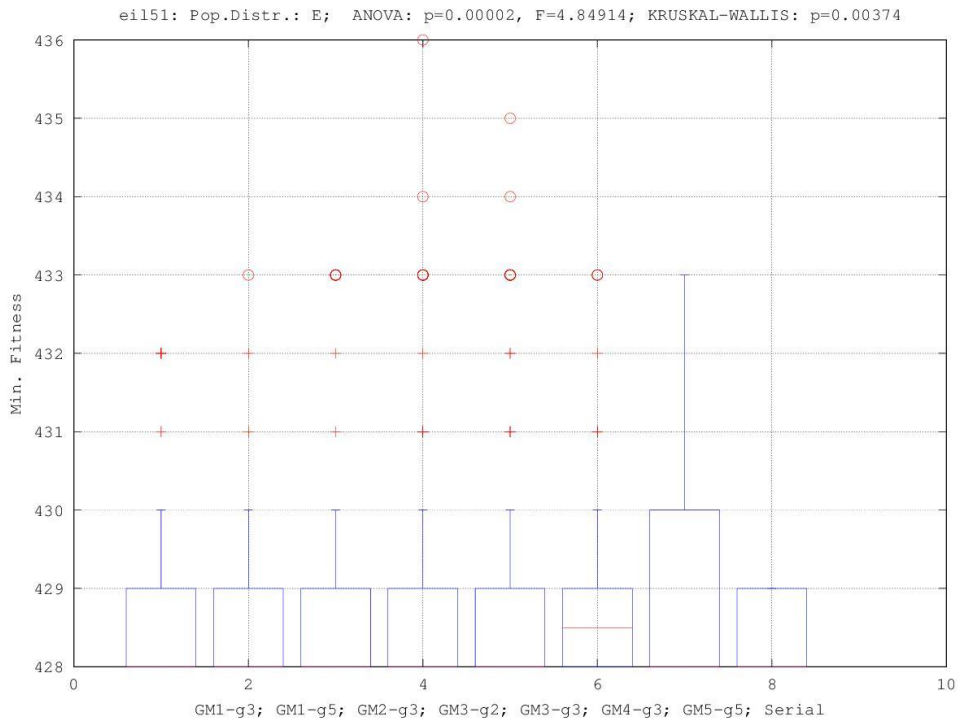


Figure 4.5.8: Minimum fitness for the *eil51* data set (Nodes: 7; Pop.: 7 per node).

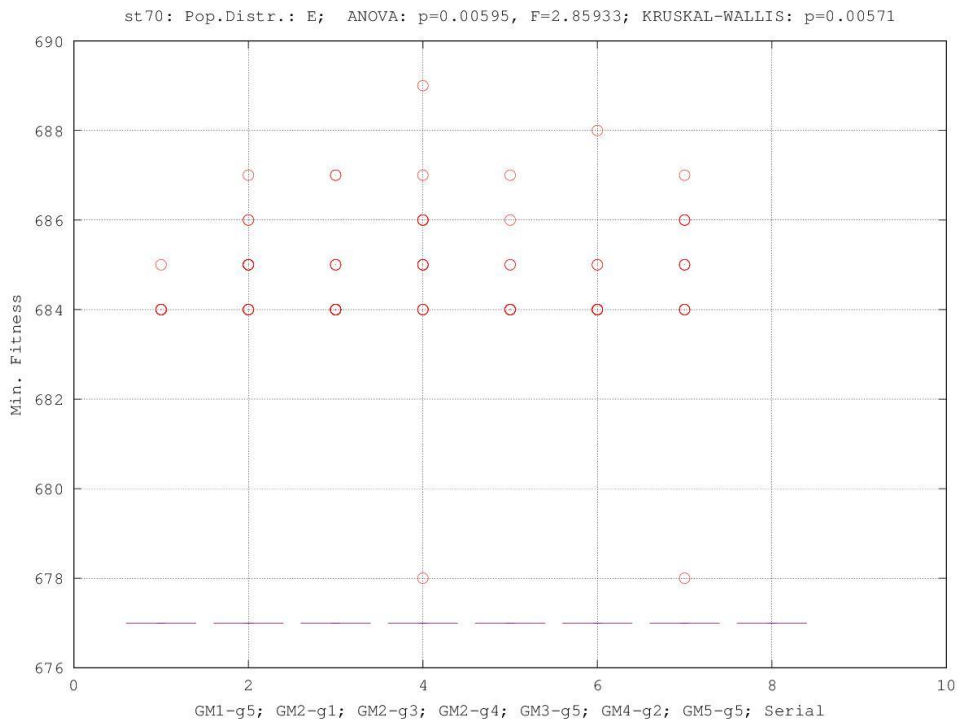


Figure 4.5.9: Minimum fitness for the *st70* data set (Nodes: 7; Pop.: 10 per node).

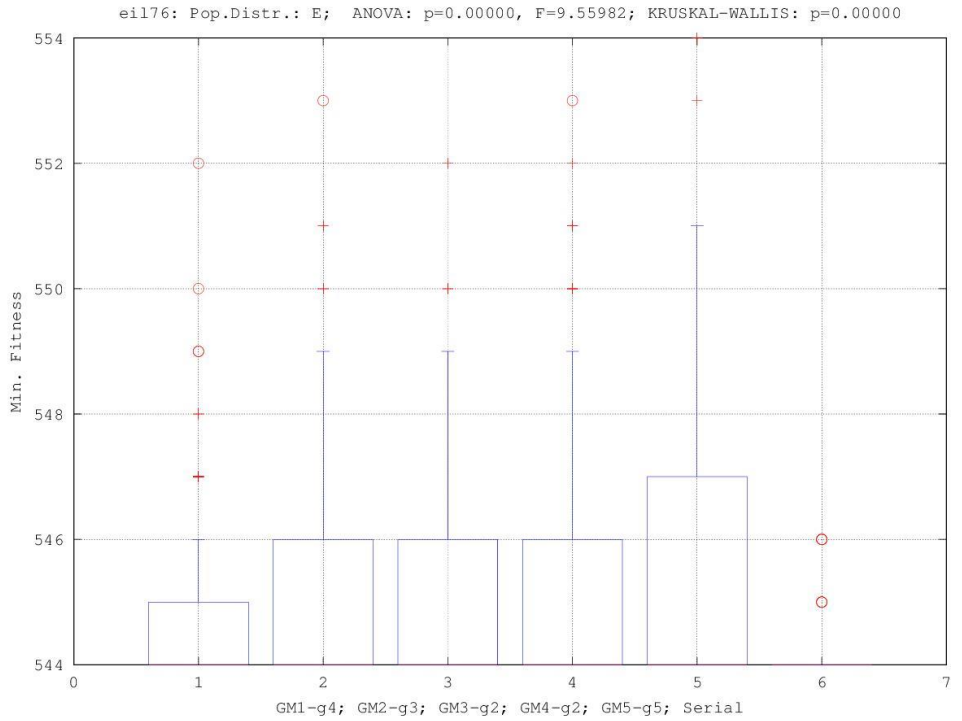


Figure 4.5.10: Minimum fitness for the *eil76* data set (Nodes: 7; Pop.: 10 per node).

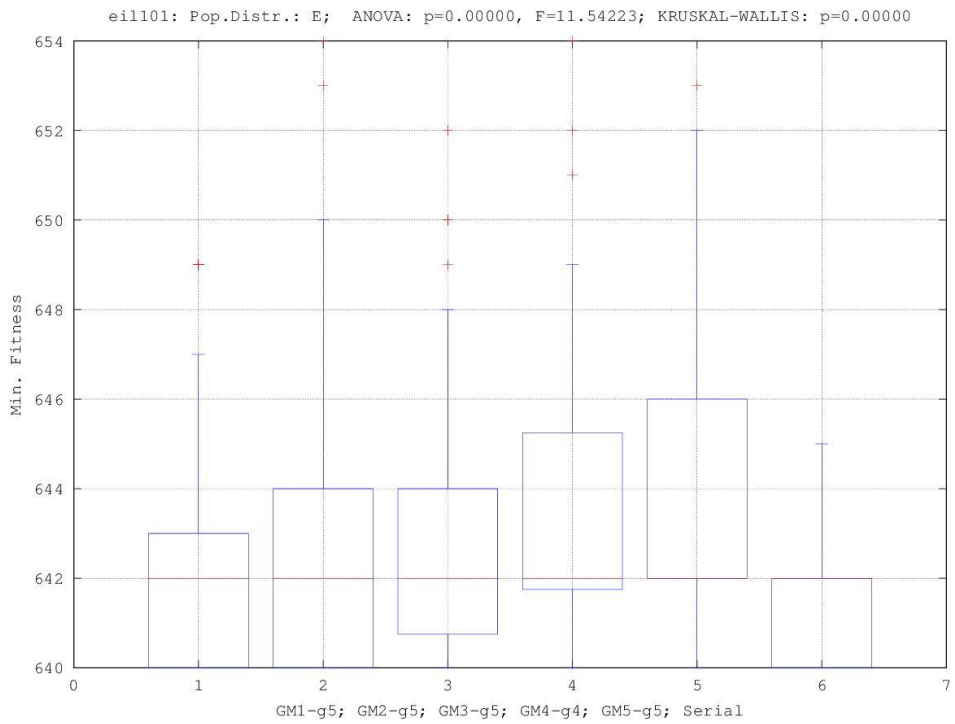


Figure 4.5.11: Minimum fitness for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

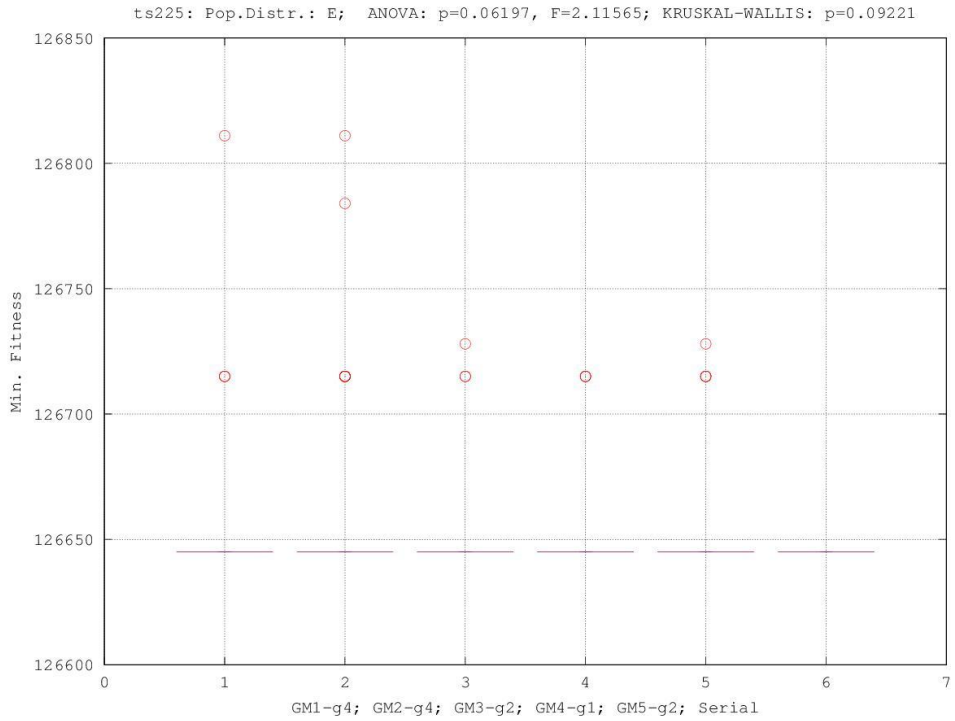


Figure 4.5.12: Minimum fitness for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

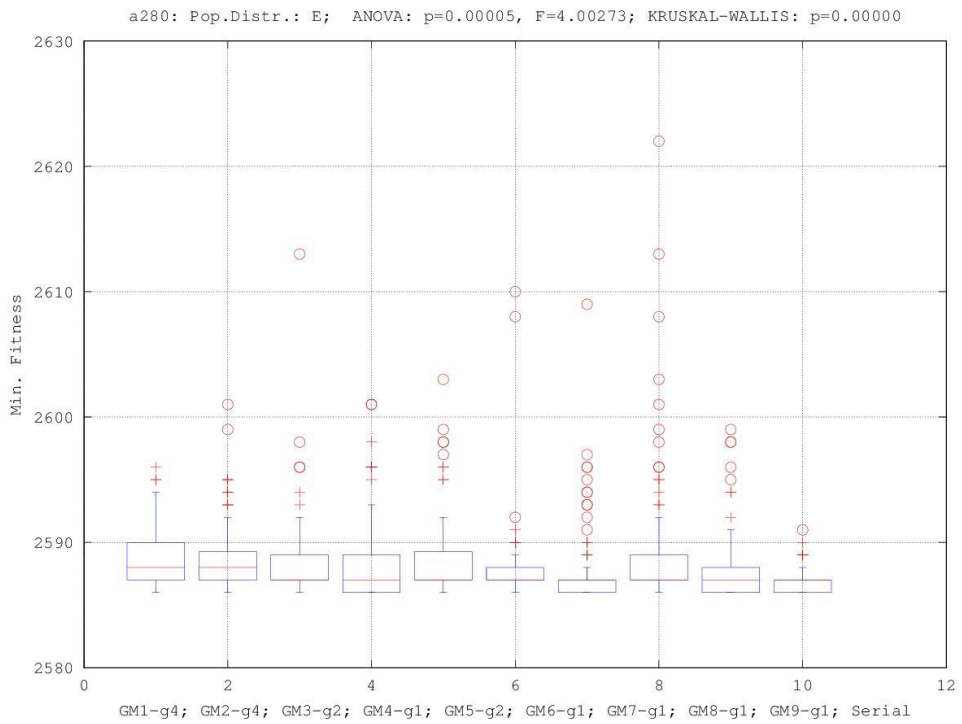


Figure 4.5.13: Minimum fitness for the *a280* data set (Nodes: 7; Pop.: 40 per node).

Observing figures *4.5.8* to *4.5.13*, one can observe that the deployment of the algorithm on a grid test bed does not considerably degrades the quality of the results (fitness) obtained with the serial implementation. Furthermore, for the *ts225* data set, the differences in the fitness results are not statistically significant ($p >= 0.05$), meaning that there is no significant difference between the data series which are being analysed.

Looking in more detail at the results for each data set, one can observe, for the *eil51* data set, that *g=1* promotes a delay in the execution time in all the GMs, with a slight improvement of the fitness for GM1. However, in most of the cases, *g=1000* (no migration at all) has similar time results to the remaining configurations ($g = 2, 3, 4$ *or* 5). This can be interpreted as follows: eil51 is a small enough data set that can be solved by the root node along (even with a small population equals to T/7). In this case, the grid is to a great extend useless. However, it is interesting to note for such small population a share of good quality individuals can improve the fitness. Obviously, this sharing has to be fast enough to be used by the root, thus g=1 is the only value that affects the results; all the other *g* being equally equivalent to a no-sharing setup. Figures *4.5.14* and *4.5.15* illustrate this fact, in terms of execution time and fitness, respectively.

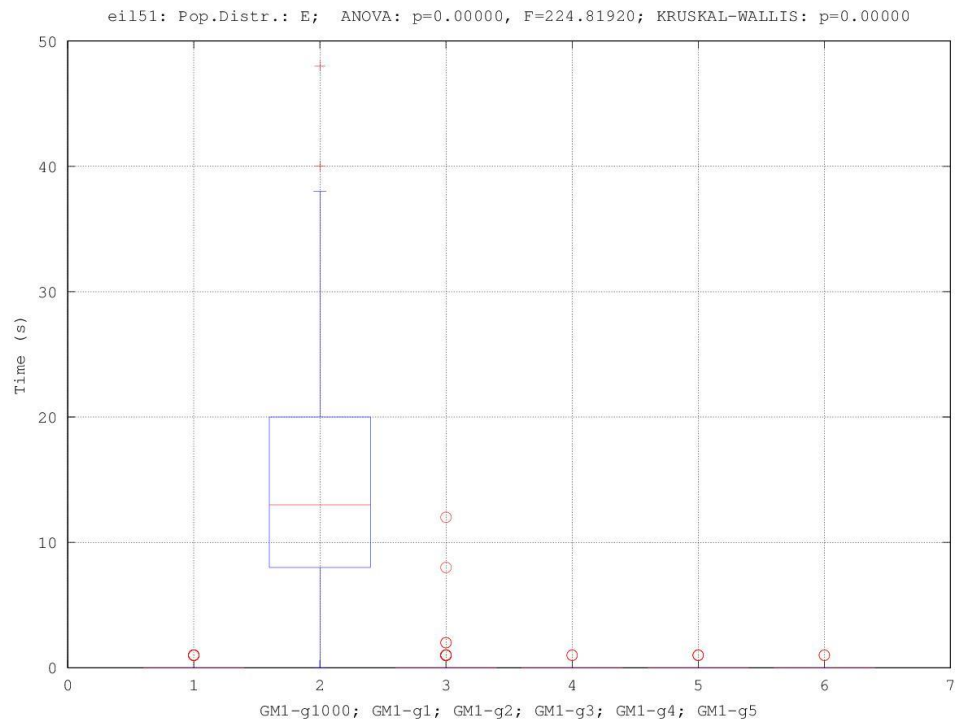


Figure 4.5.14: Execution time of *GM1* for the *eil51* data set (Nodes: 7; Pop.: 7 per node).

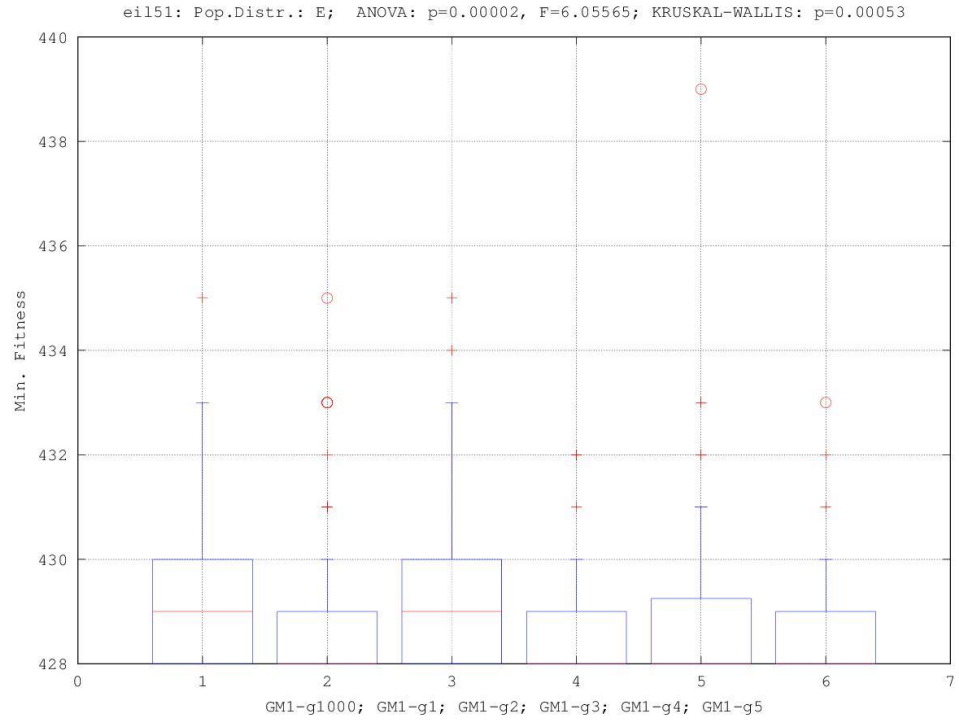


Figure 4.5.15: Minimum fitness of *GM1* for the *eil51* data set (Nodes: 7; Pop.: 7 per node).

Regarding the *st70* data set, in most of the cases, *g=1000* has similar results to the remaining configurations (*g = 1, 2, 3, 4 or 5*). However, there is an exception for *GM5*. Looking at figure *4.5.16*, one can observe that *g=1000* configuration is the slowest one, especially when compared to *g=5*. As for the fitness, presented in figure *4.5.17*, the differences in the results are not statistically significant, which, no matter the *g* value

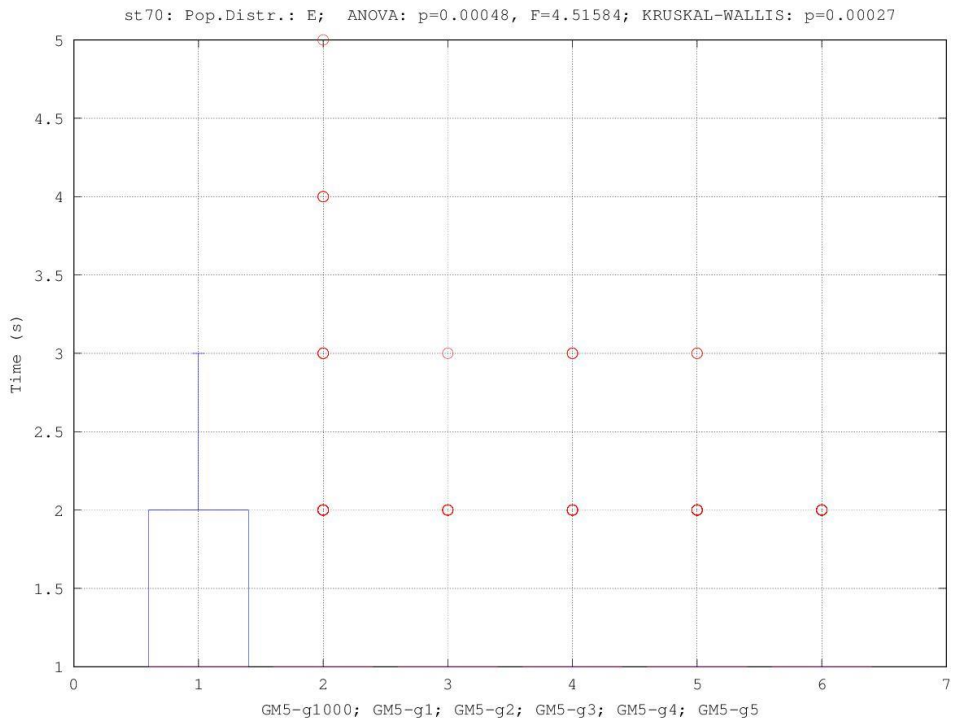


Figure 4.5.16: Execution time of *GM5* for the *st70* data set (Nodes: 7; Pop.: 10 per node).

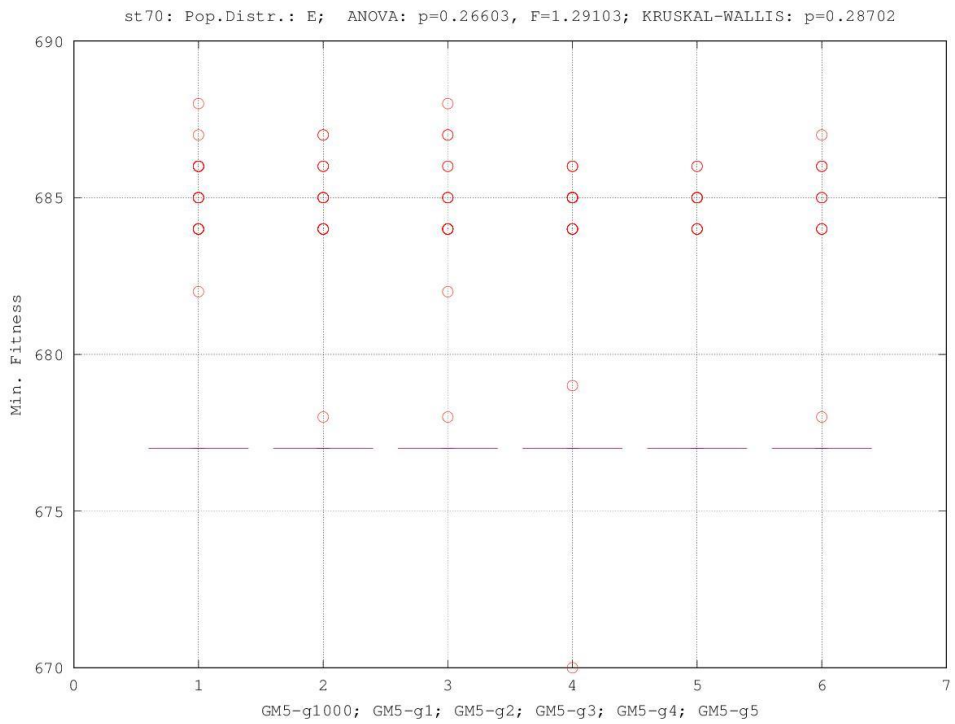


Figure 4.5.17: Minimum fitness of *GM5* for the *st70* data set (Nodes: 7; Pop.: 10 per node).

For the *eil76* data set, *g=1000* has similar results to the remaining configurations (*g = 1, 2, 3, 4 or 5*), except for *GM3* and *GM5*. In the case of GM5, only execution time has statistically significant differences in the results, where *g=1000* increases the execution time of the algorithm. In its turn, GM3 presents both execution time and fitness with statistically significant differences in the results, as shown in figures *4.5.18* and *4.5.19.*

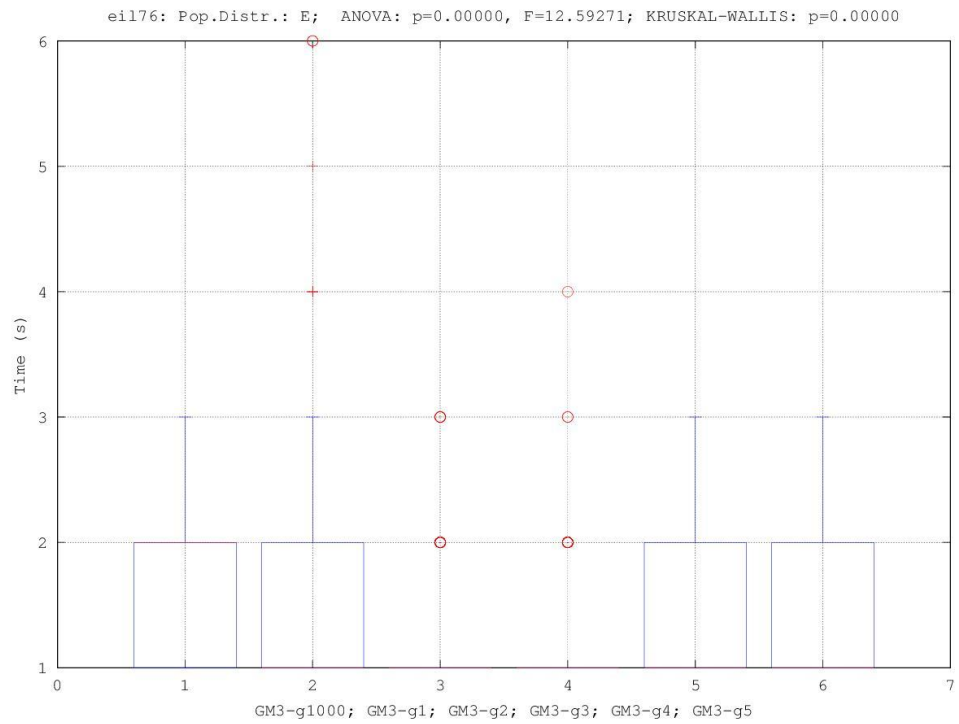


Figure 4.5.18: Execution time of *GM3* for the *eil76* data set (Nodes: 7; Pop.: 10 per node).

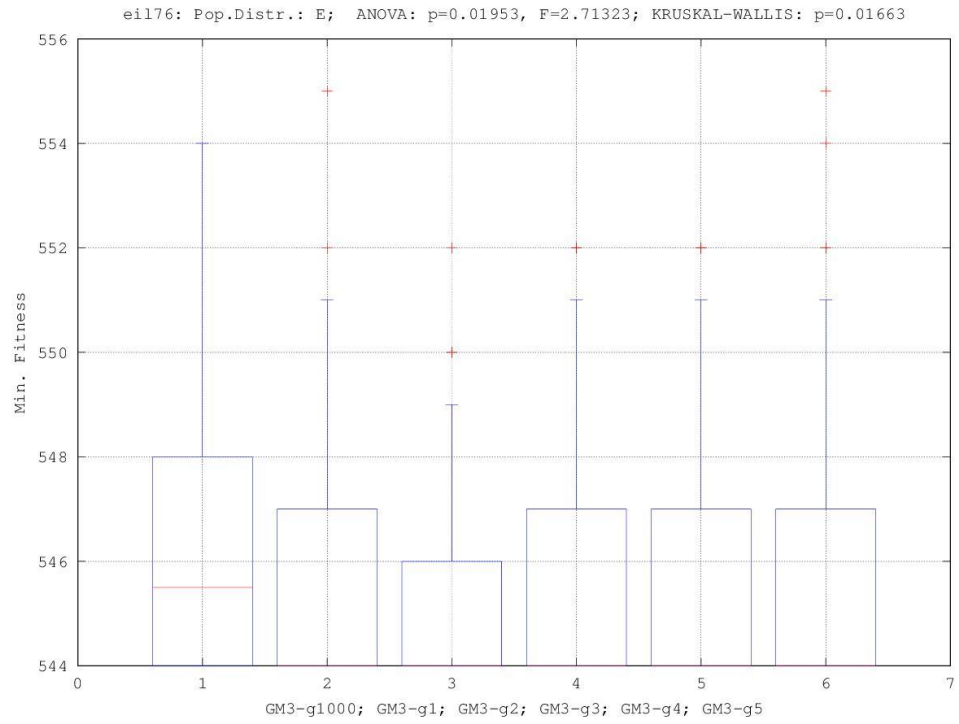


Figure 4.5.19: Minimum fitness of *GM3* for the *eil76* data set (Nodes: 7; Pop.: 10 per node).

As for the *eil101* data set, it is interesting to verify that the benefits of using the grid become much clearer. Looking at figures *4.5.20* to *4.5.24* (*GM1* to *GM5*), one can observe that execution times for $g = \{1, 2, 3, 4, 5\}$ are above *g=1000*. Figure *4.5.20* shows that, for *GM1* (a star), the execution time for *g=1000*, i.e. without migrations, is lower than for any other value of *g*.

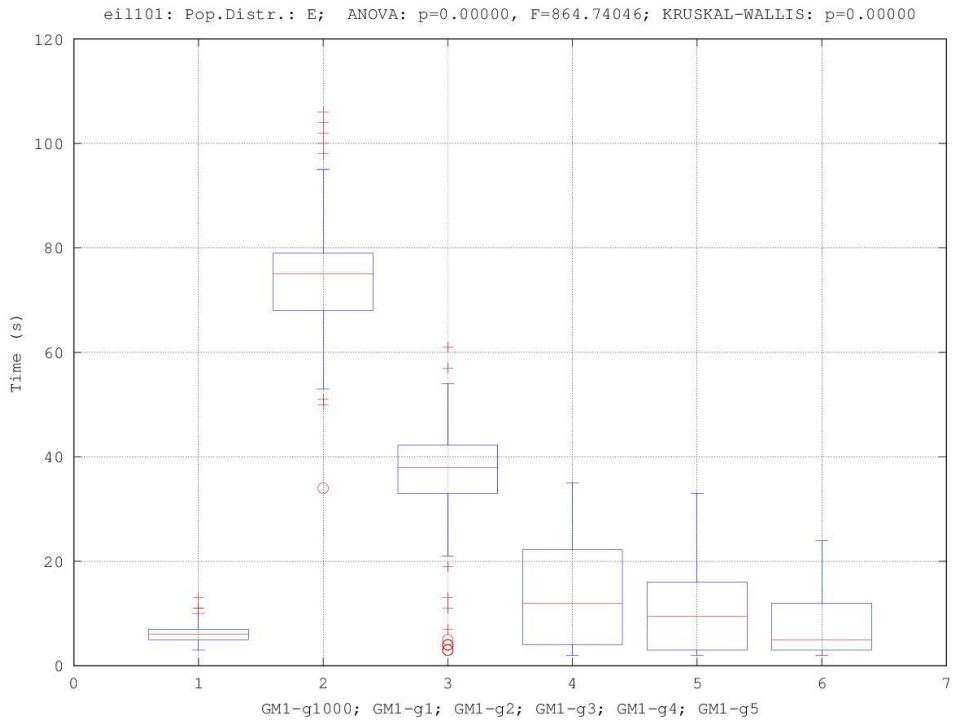


Figure 4.5.20: Execution time of *GM1* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

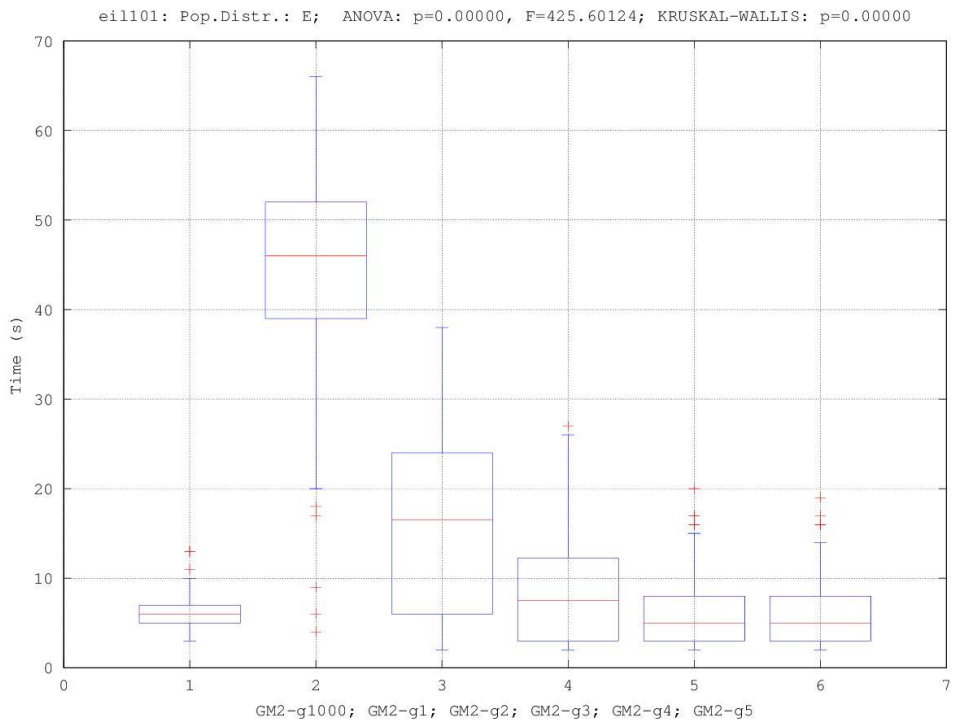


Figure 4.5.21: Execution time of *GM2* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

In the case of *GM2* (an incomplete rooted tree), the *g=4* and *g=5* configurations are faster than *g=1000*, as shown in figure *4.5.21*. This tendency continues with *GM3*, where only *g=1* performs slower than *g=1000*, as one can observe in figure *4.5.22*.

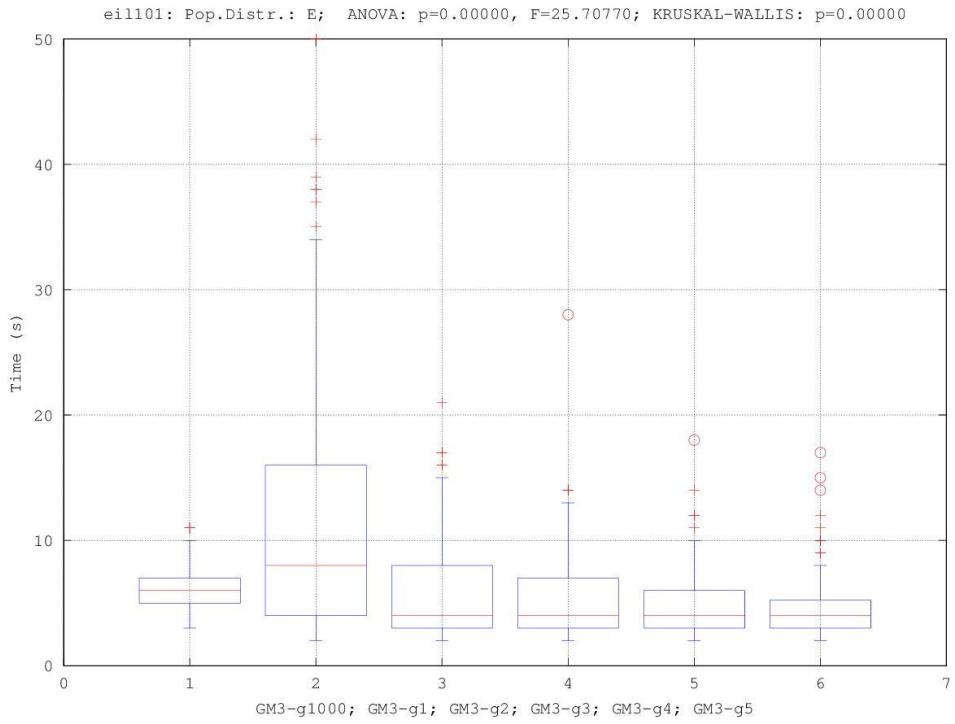


Figure 4.5.22: Execution time of *GM3* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

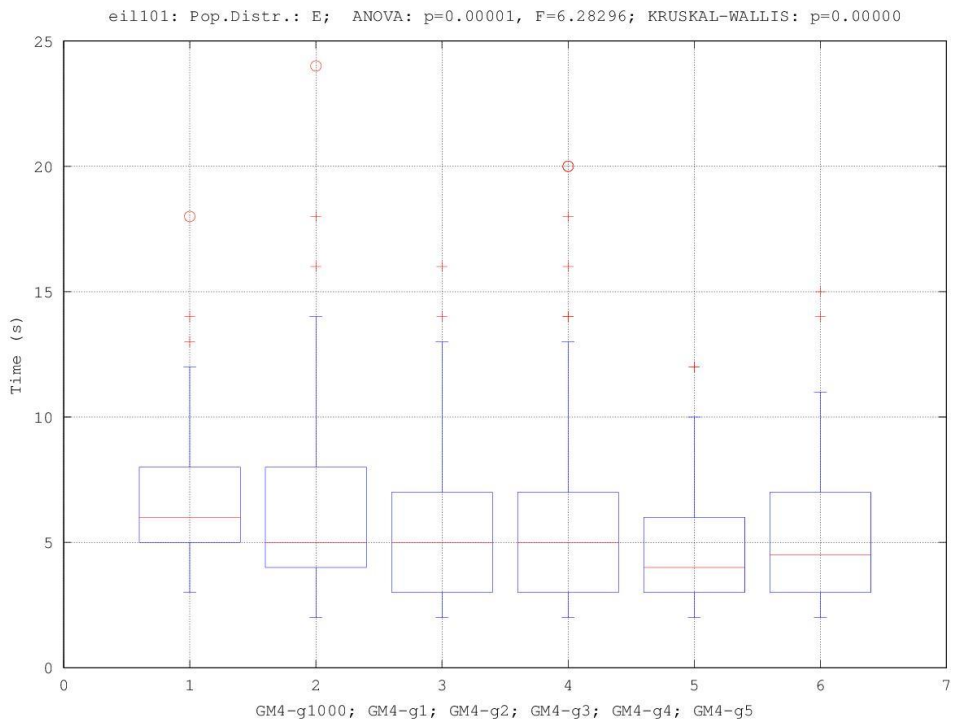


Figure 4.5.23: Execution time of *GM4* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

With *GM4*, all configurations *g = {1, 2, 3, 4, 5}* perform faster than g=1000 (see figure *4.5.23*) and g=4 has the lowest execution time.

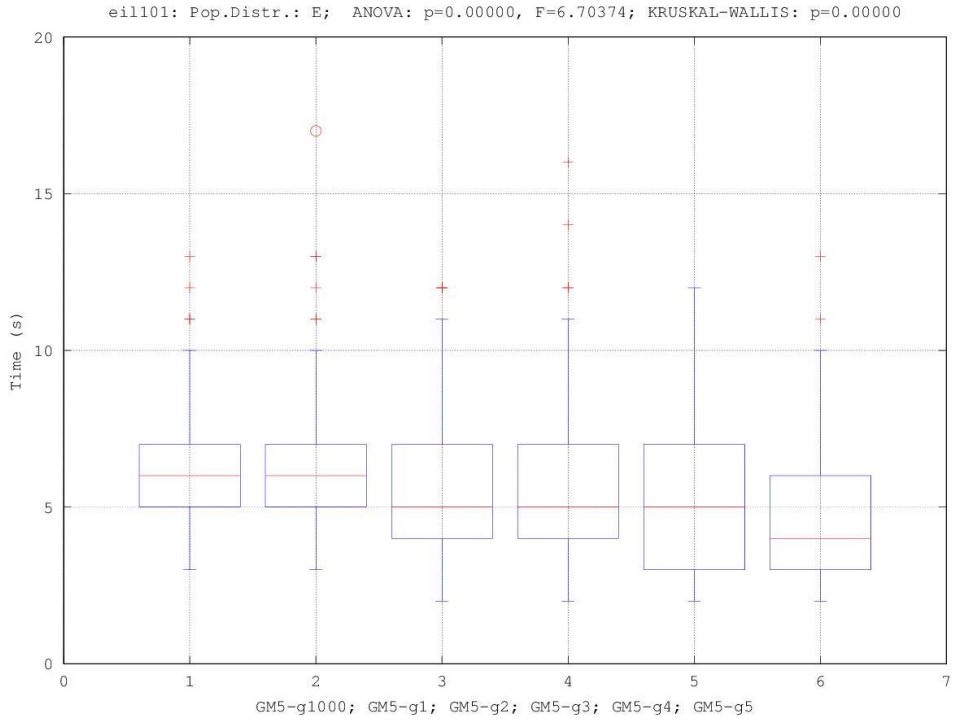


Figure 4.5.24: Execution time of *GM5* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

Using *GM5*, the results are identical to *GM4*. The configurations *g = {2, 3, 4, 5}* perform faster than *g=1000* while *g=1* is very similar to *g=1000*, as one can observe in *4.5.24*.

In terms of fitness, one can observe that g=4 and g=5 configurations were able to obtain better results, from G*M1* to *GM4* (see figures *4.5.25* to *4.5.29*).

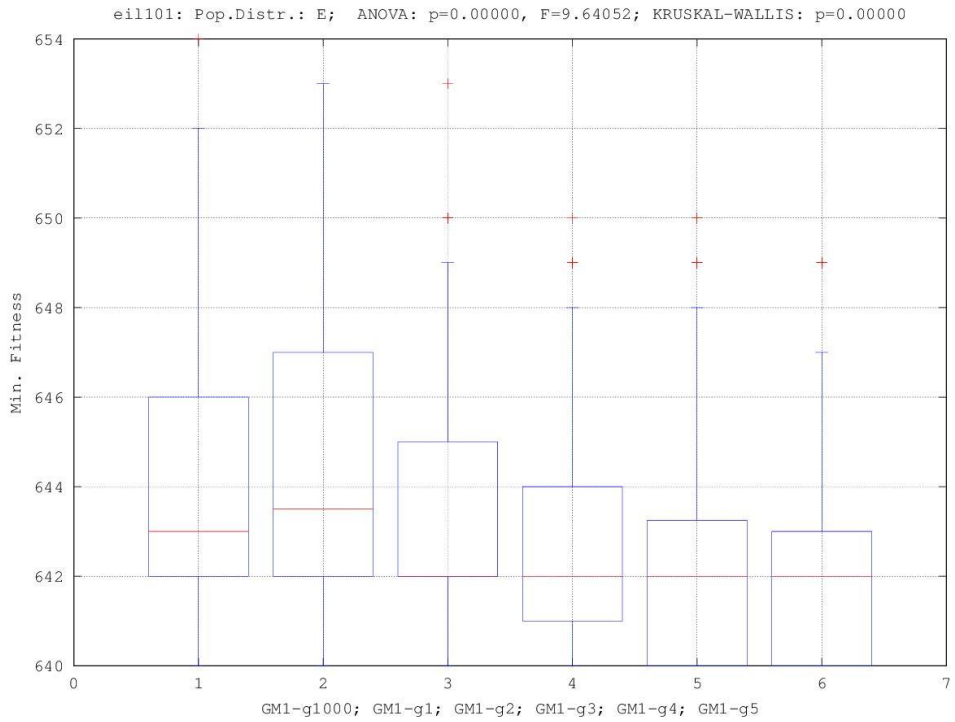


Figure 4.5.25: Minimum fitness of *GM1* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

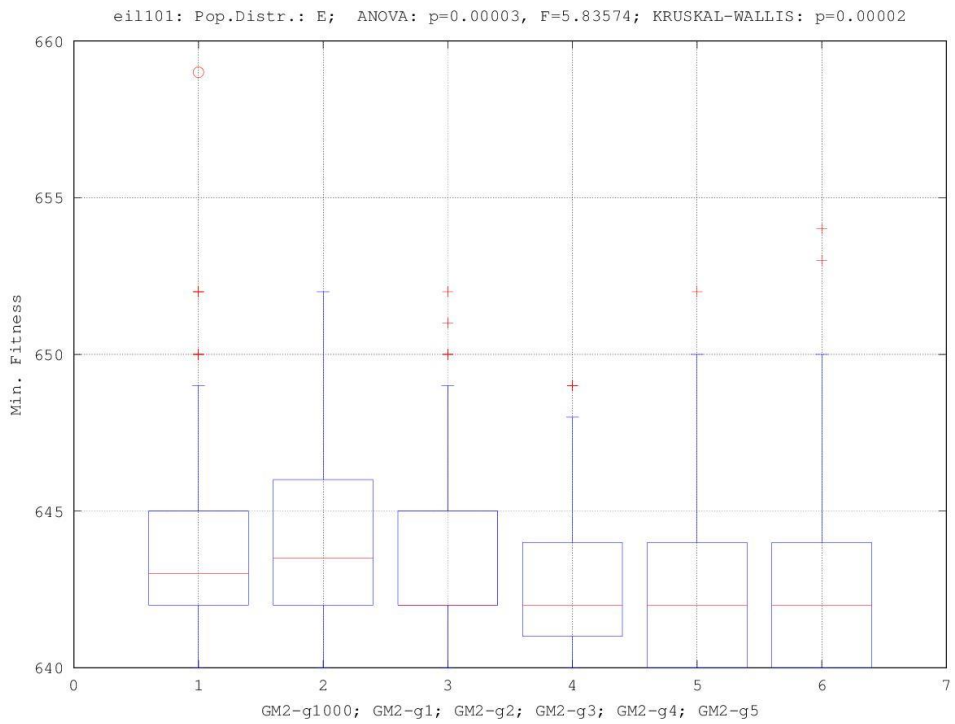


Figure 4.5.26: Minimum fitness of *GM2* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

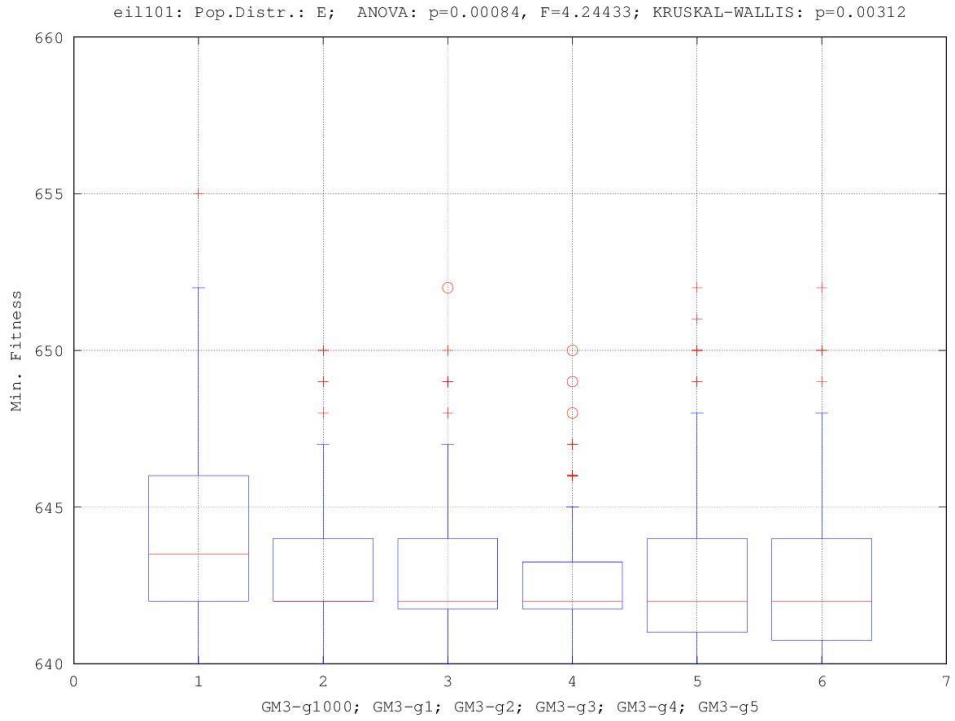


Figure 4.5.27: Minimum fitness of *GM3* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

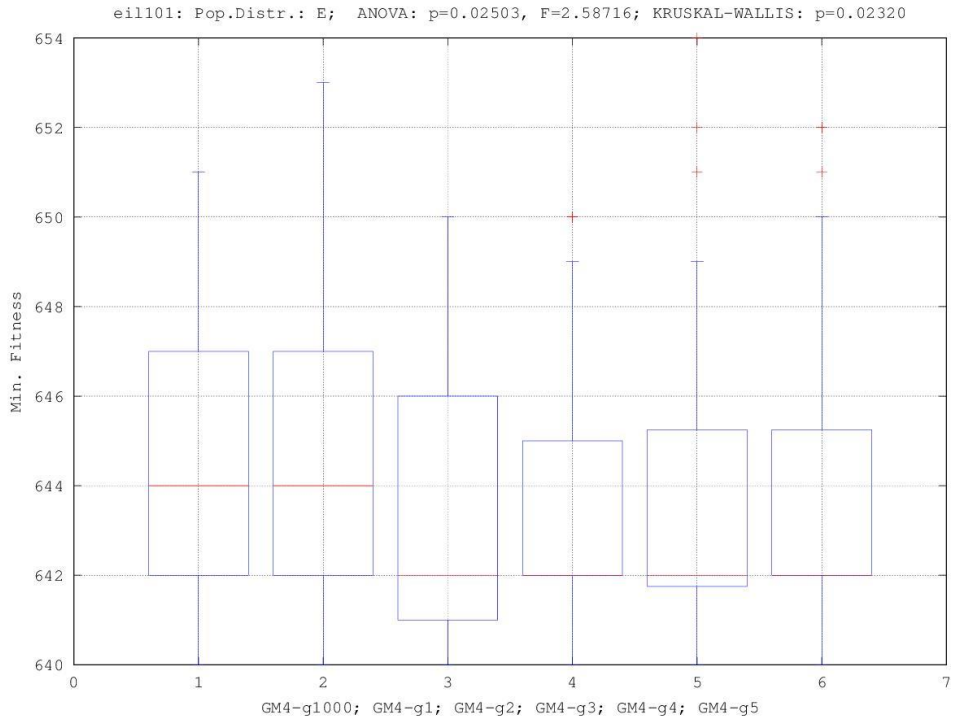


Figure 4.5.28: Minimum fitness of *GM4* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

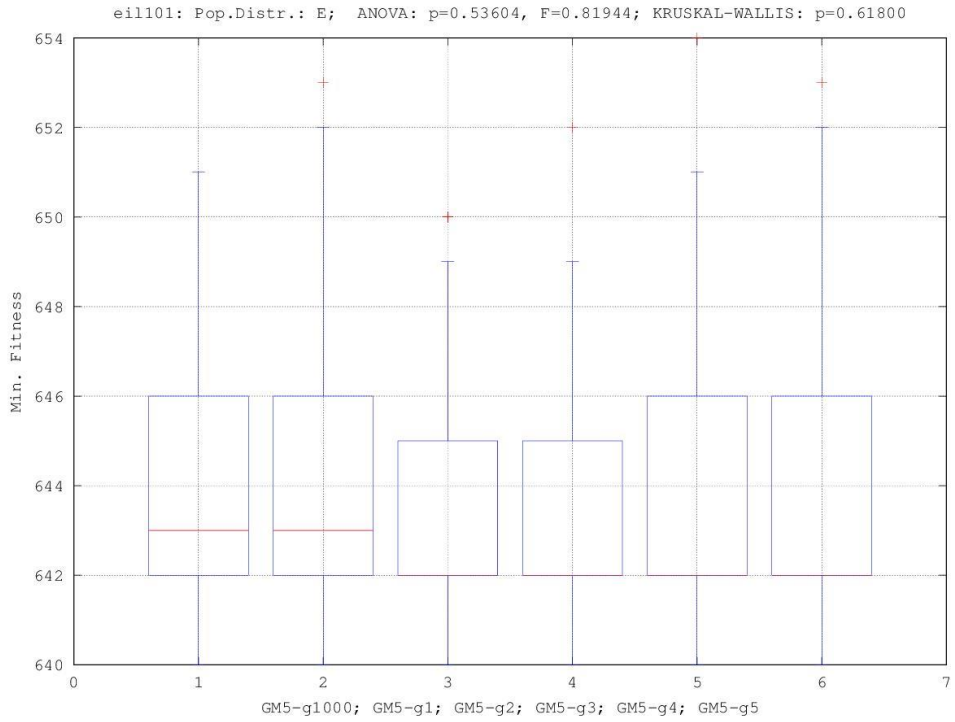


Figure 4.5.29: Minimum fitness of *GM5* for the *eil101* data set (Nodes: 7; Pop.: 14 per node).

In the case of *GM5*, figure *4.5.29* shows that there are no significant differences between the different values of *g*, i.e., regarding fitness one can choose any of the configurations for GM5 when applied to the *TSPLIB eil101* data set (101 cities).

Regarding the *ts225* data set, let us look at figures *4.5.30* and *4.5.31* which are similar to figures *4.5.6* and *4.5.12* respectively, but without the *Serial* values. Figure *4.5.30* presents the execution time of best *g* for each GM. The differences in these results are not statistically significant and the same applies to fitness, as shown in figure *4.5.31*. The fact that the differences in the results, for both time and fitness, are not statistically significant means that, no matter the chosen GM, the algorithm takes about the same time while maintaining the quality of the results.

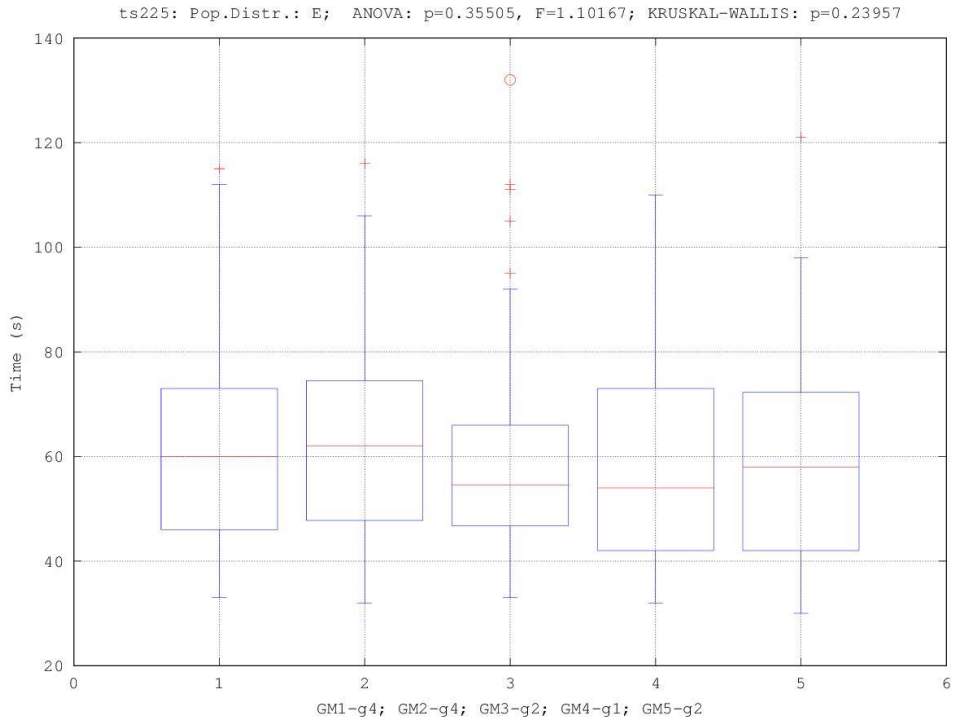


Figure 4.5.30: Execution time for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

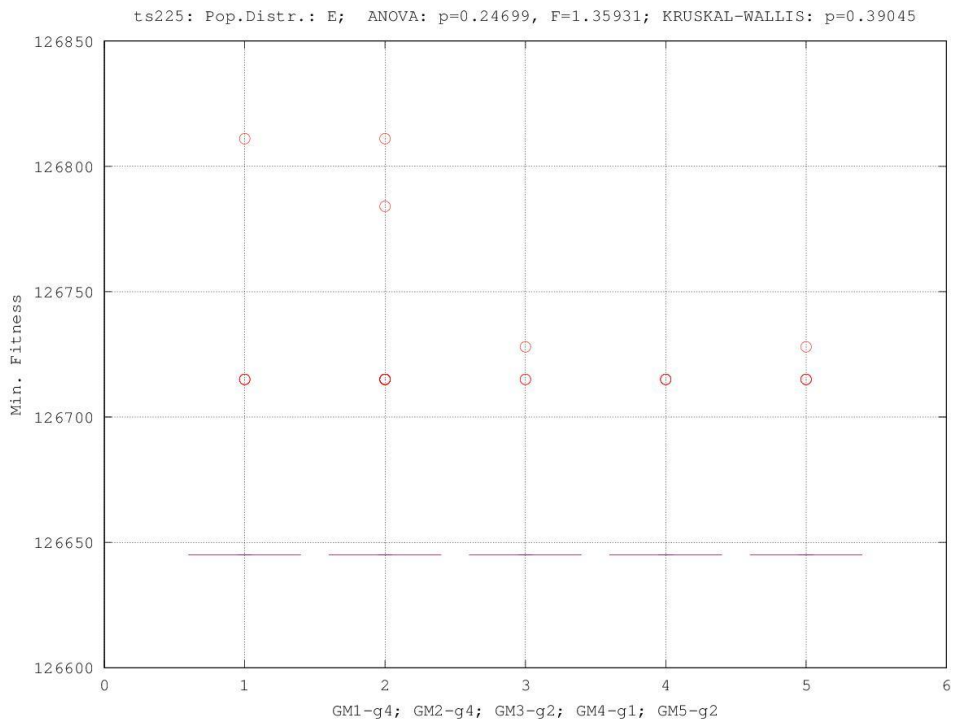


Figure 4.5.31: Minimum fitness for the *ts225* data set (Nodes: 7; Pop.: 32 per node).

4.6. Topologies Comparison - 280 Cities data set

The largest tested data set was the *TSPLIB a280* (280 cities). On this data set, additional g values and GMs were tested, namely g = {*10*, *20*, *50*, *100*} and *GM6*, *GM7*, *GM8* and *GM9*. For simplicity reasons, these data sets' analysis will focus on the comparison of best configuration for each GM. From this point, all presented figures will refer to this data set assuming topologies with *7* grid nodes with a population of *N/7* (40) individuals each.

The first noticeable fact, for this data set, is that the differences in the execution time results are not statistically significant, when comparing the best configuration for each grid map, as shown in figure *4.6.1*.

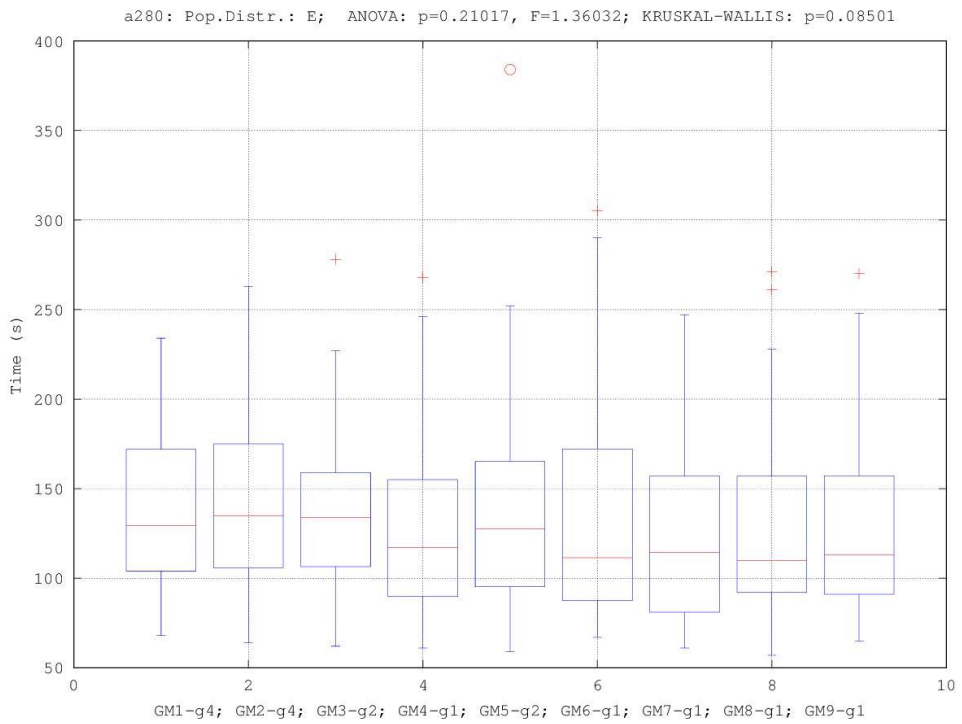


Figure 4.6.1: Execution time for the *a280* data set.

In terms of fitness, for the *a280* data set, the differences in the results were also not statistically significant until *GM6*, as shown in figure *4.6.2*.

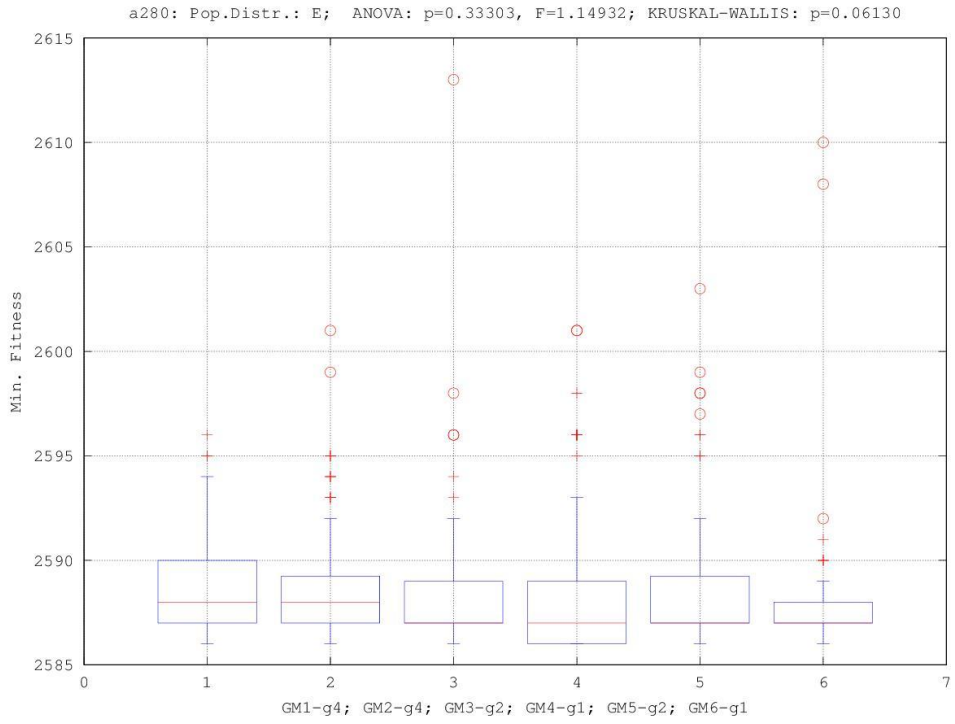


Figure 4.6.2: Minimum fitness for the *a280* data set, for the best *g* from *GM1* to *GM6*.

Regarding fitness, the results became statistically significant with the addition of *GM7* (see figure *4.6.3*), considering the results from the Kruskal-Wallis test.

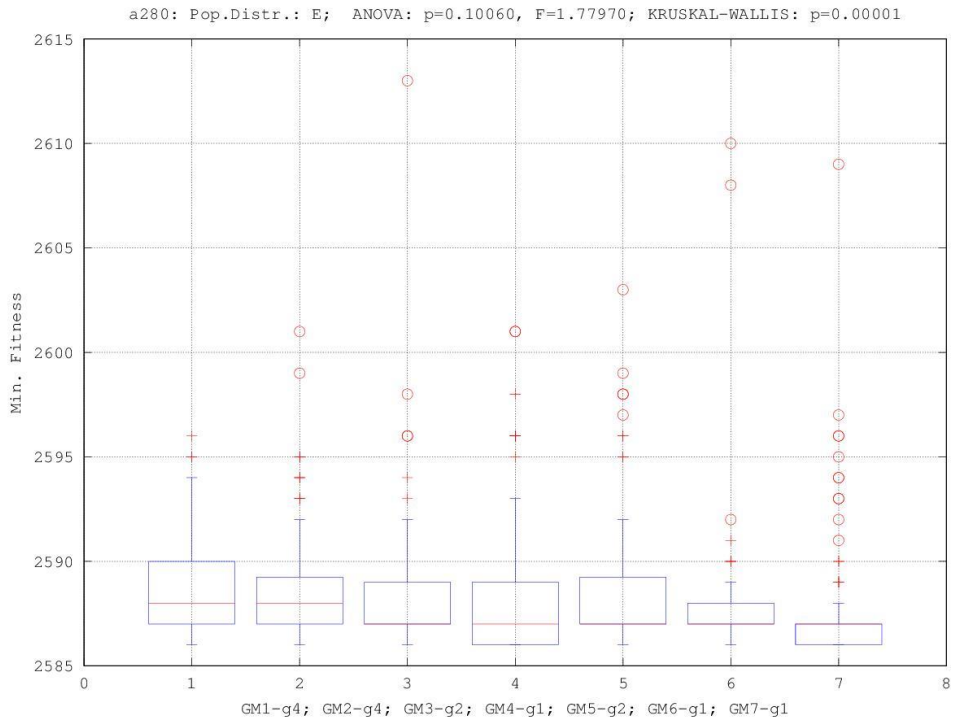


Figure 4.6.3: Minimum fitness for the *a280* data set, from *GM1* to *GM7*.

Notice that figure *4.6.3* also presents a difference between the results provided by ANOVA and Kruskal-Wallis tests. According to ANOVA, the differences in the results are not statistically significant. In the opposite, Kruskal-Wallis test concludes that the differences in the results are statistically significant. Considering these facts, one can conclude that GM7 provides the best fitness results, if there is an effective difference in these results.

Replacing *GM7* with *GM8*, the differences in the results became, once again, not statistically significant, as shown in figure *4.6.4*.

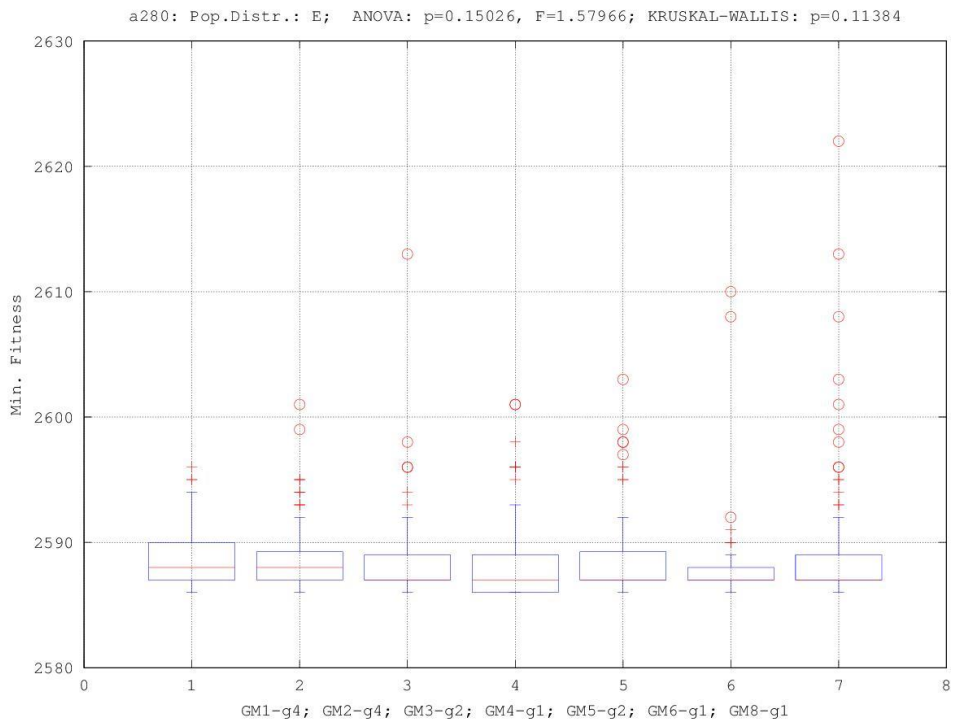


Figure 4.6.4: Minimum fitness for the *a280* data set, with *GM1* to *GM6* and *GM8*.

Observing figures *4.6.2* to *4.6.4*, one can conclude that, as far as this investigation goes, *GM7* (rooted tree-ring topology) promoted a statistically significant improvement of the fitness, unlike the other tested topologies.

Notice that *GM9* was obtained by maintaining the complete binary tree topology in *GM4*, while changing the nodes distribution, placing the fastest ones on the top levels of the tree. Looking at the comparison between those two grid maps, presented in figures *4.6.5* and *4.6.6*, one can observe that the differences in the results are not statistically significant. This means that placing the fastest nodes in the top levels of the tree has no significant impact on the performance of the algorithm, in both execution time and fitness.

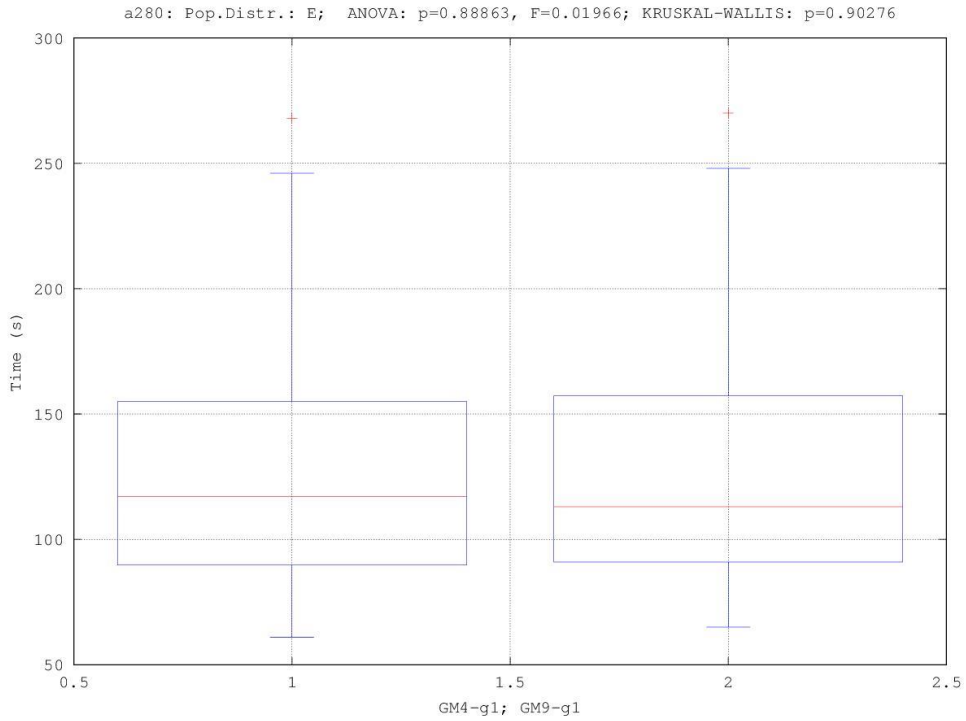


Figure 4.6.5: Execution time comparison for the *a280* data set.

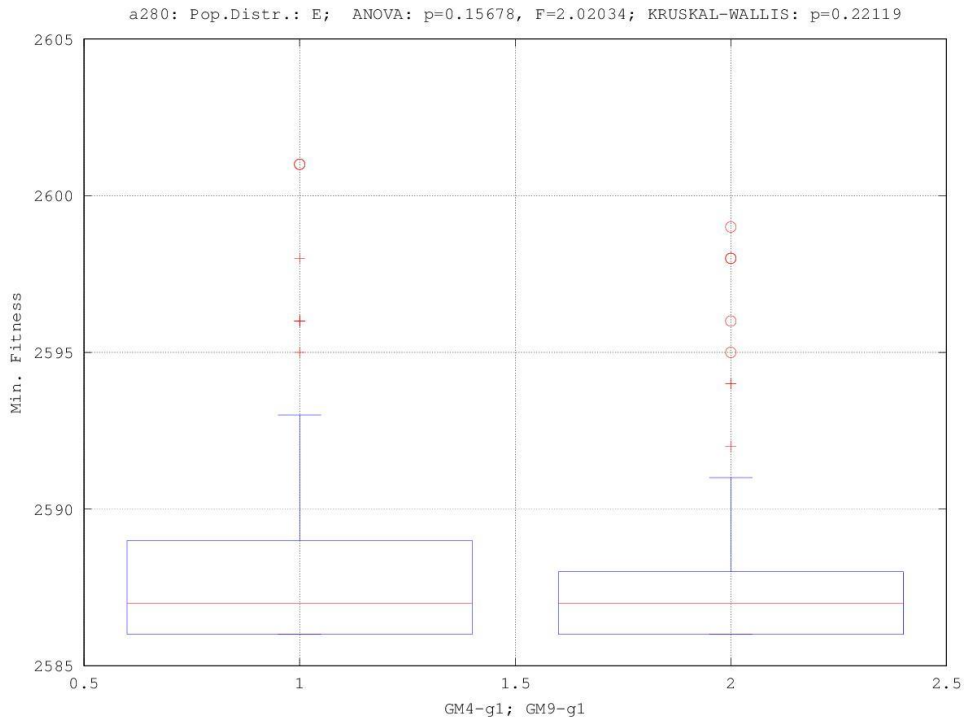


Figure 4.6.6: Fitness comparison for the *a280* data set.

4.7. The Contribution of Slower Grid Nodes

Considering the heterogeneity of the test beds and the asynchronous properties of the GA, an important issue to address was verifying the effective contribution of all computing nodes, namely the slowest ones. *GM10* was specifically designed for this purpose, by selecting *GM8* (one could select any other topology, knowing that the differences are not statistically significant in terms of execution time, for the *a280* data set) and removing the slowest nodes (*sbgrid3, sbgrid4 and sbgrid7*). If these nodes provide an effective contribution, their removal would promote a degradation of the execution time of the algorithm. In fact, this working hypothesis was verified to the *a280* data set.

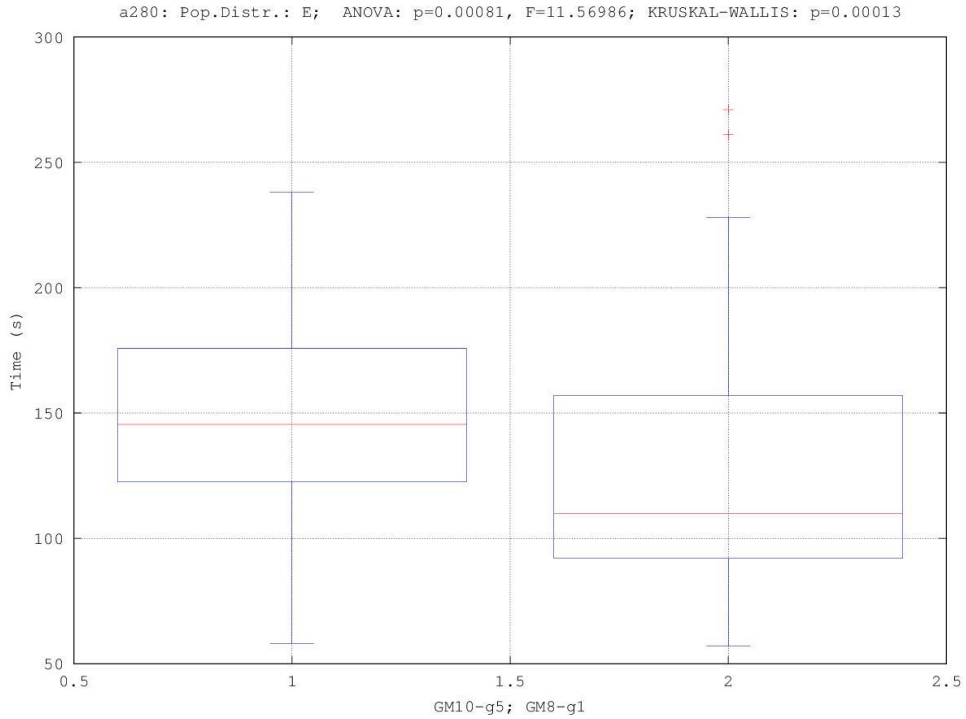


Figure 4.7.1: Execution time comparison for the *a280* data set.

Comparing the best configurations of *GM10* and *GM8*, as shown in figure *4.7.1*, one can observe that *GM10* did promote a statistically significant degradation of the execution time, when compared to *GM8*.

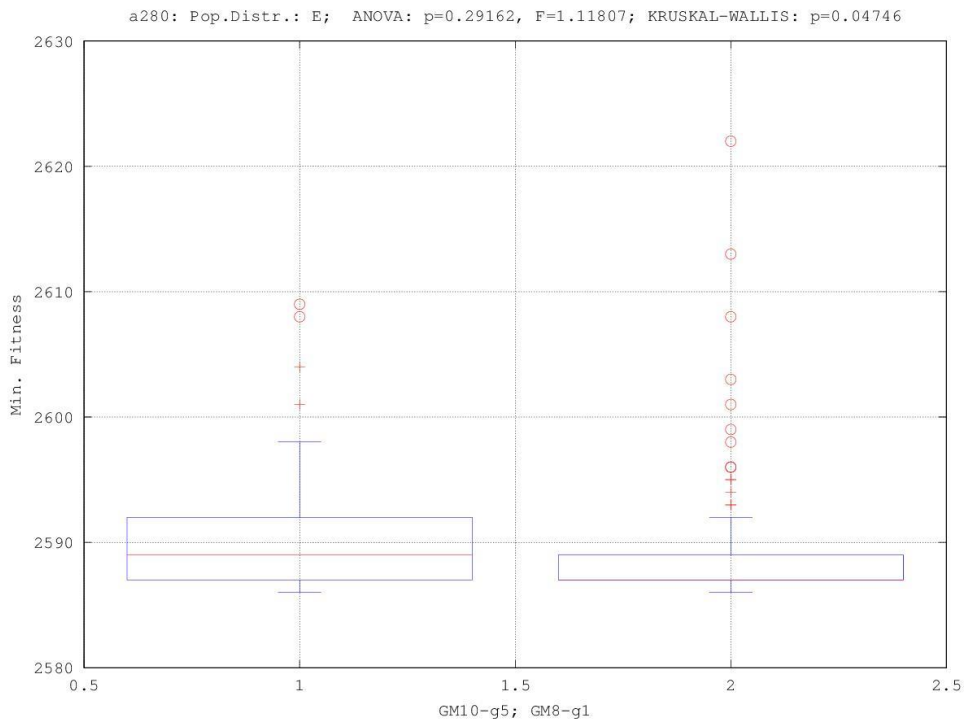


Figure 4.7.2: Fitness comparison for the *a280* data set.

In fact, besides promoting a significant degradation of the execution time, *GM10* also promoted a significant degradation of the fitness results, as shown in figure *4.7.2*. Therefore, the working hypothesis was verified to the *a280* data set, i.e., the slowest nodes provide an effective contribution as their removal of the test bed promote a statistically significant degradation of the results, for both execution time and fitness.

4.8. Statistical Validation

ANOVA and Kruskal-Wallis default values for *p* were set to *0.05*, i.e. providing a confidence interval of 95%. ANOVA test assumes that samples are drawn from normally distributed populations and homoscedasticity (homogeneity of variance), i.e. the variance of data in groups should be the same. The homogeneity of variances can be tested with Levene´s test (Levene, 1960), by checking the null hypothesis that the population variances are equal. If the resulting *p* value of Levene´s test is less than the chosen value of significance (typically *0.05*) the null hypothesis of equal variances is rejected and one can conclude that there is a difference between the variances in the population.

Levene´s original test relies on group mean values, but (Brown and Forsythe, 1974) extended it to median and trimmed mean values. The studies performed by these authors indicated that the trimmed mean version performed best when the underlying data followed a heavy-tailed distribution, and the median version performed best when the underlying data followed a heavily skewed distribution. The mean value provided the best performance for symmetric, moderate-tailed, distributions. Although the optimal choice of the Levene's test version depends on the underlying distribution, the median version provides good robustness against several types of non-normality, while retaining good statistical power. Table *4.8.1* presents the results of the Levene's test, applied to the data from section 4.6, concerning the *a280* data set.

Levene's Test	
Type	**p-value**
Mean	0,00258031
Median	**0,103325446**
Trimmed mean	0,010252713

Table 4.8.1: Results of the Levene's test, applied to the data from the a280 data set.

Applying the median version of the Levene's test to the source data of figure *4.6.1* (the execution time results for the a280 data set), one could obtain $p = 0.103$ and conclude that there is no significant difference between the variances. The other ANOVA assumption, i.e., the normally distribution of the population, was verified by the Shapiro-Wilk test (Shapiro and Wilk, 1965). In this test, if the resulting p value is less than 0.05, we reject the null hypothesis that the data is normally distributed. Applying the Shapiro-Wilk test to the same source data, one could obtain $p = 0$ and conclude that samples are not drawn from normally distributed populations. This fact can increase the chance of false positive results when analyzing data with a test that assumes normality, such as ANOVA. Although some authors argue that the false positive rate is not very affected by the violation of the normality assumption (Glass et al., 1972; Harwell et al., 1992; Lix et al., 1996), we decided to provide an additional statistical validation using the Kruskal-Wallis test (Kruskal and Wallis, 1952), a non-parametric test where no assumptions are made about the distribution of data. The null hypothesis of the Kruskal-Wallis test is that the samples come from populations with the same distribution. Applying the Kruskal-Wallis test to the same source data, one could obtain $p = 0.085$ and conclude that the null hypothesis cannot be rejected, i.e. there are no significant differences between the results, which is exactly the same conclusion obtained with the ANOVA test.

Furthermore, all figures presented in chapter 4 included the results of both ANOVA and Kruskal-Wallis tests, in order to clarify any eventual doubts related to the suitability of the ANOVA test when applied to non-normal distributions.

CHAPTER 5: Conclusions and Future Work

In order to provide a technically sound answer to the research question:

What is the fastest Island Model topology for solving TSP instances using an order-based genetic algorithm, in a distributed heterogeneous grid computing environment, without losing significant fitness comparatively to the correspondent sequential panmictic implementation of the same algorithm?

a comparison of migration topologies, using a grid computing framework, was performed. To the best of our knowledge, this comparison, in a distributed heterogeneous environment such as proposed in this work, could not be found in the literature. The research methodology was primarily experimental, observing and analysing the behaviour of the algorithm while changing the properties of the island model, using the Travelling Salesman Problem (TSP) as a benchmark for the parallelization of GA in a grid computing framework. The main difficulty in finding optimal solutions to TSP is the large number of possible tours; $(n\text{-}1)!/2$ for symmetric n cities tour. As the number of cities in the problem increases, the number of possible tours also increases, in a factorial way. TSP is therefore computationally intractable, thus fully justifying the employment of a stochastic optimization method such as GA.

This work has verified the benefits of using parallel evolutionary algorithms (PEAs), i.e., a faster and better performing algorithm, due to the use of a structured population, i.e. a spatial distribution of individuals (Alba and Tomassini, 2002). Population decentralization was achieved by distributing the population by a set of processing nodes (islands) which periodically exchange (migrate) candidate solutions. For all the tested data sets, no matter the chosen topology, the concurrent versions of the algorithm performed much faster than the serial version. Furthermore, this work argues on the suitability of an asynchronous approach, considering the dynamic and potentially heterogeneous characteristics of the grid. Furthermore, coordinated topologies are presented as the most suitable approach when deploying GA in a grid computing environment. Besides providing a better fault-tolerance to slow or faulty nodes, the periodically share of best solutions with parent nodes does provide population diversity and prevents getting trapped in local optima. On the other hand, child nodes are expected to provide quality solutions to their parents who will need less generations to reach their best solution to the problem, reducing the processing time of the algorithm.

As far as this investigation went, one could conclude that parallelization allows to significantly reduce the execution time of GAs, without affecting significantly the quality (fitness) of the results obtained in the serial version.

The influence of the number of nodes in a given topology was also addressed. It was possible to observe that, for the *a280* data set, adding *2* nodes to the grid test bed allowed to obtain significant gains in terms of processing time, without a significant degradation of fitness.

Two different techniques for population distribution were addressed, concluding that the algorithm performed systematically faster in the cases where the population is equally-distributed (E), rather than tree-distributed (T). Possible explanations to this fact were also presented.

It was possible to observe that the differences between execution times, when comparing with the serial version, get more accentuated with the increasing size of the problem, i.e., as the number of cities increases. The benefits of using the grid become much clearer at *eil101* data set (*101* cities) and beyond, without significant fitness degradation. Furthermore, for the largest tested data sets (*ts225* and *a280*), the differences, for both execution time and fitness, were not statistically significant, meaning that, no matter the chosen topology, the algorithm took about the same time to perform while maintaining the quality of the results. In fact, *GM7* (rooted tree-ring topology) promoted a small, but statistically significant, improvement of the fitness, when compared with the other tested topologies.

Another noticeable fact was that *GM9* (complete binary tree with the fastest nodes on the top levels of the tree) did not produce statistically significant differences in the results when compared to *GM4*, i.e., a complete binary tree but with one of the fastest nodes swapped to the bottom of the tree. This means that, at least for this dataset, placing the fastest nodes in the top levels of the tree has no significant impact on the performance of the algorithm, in both execution time and fitness.

Answering our research question:

For the largest tested problems, all tested topologies, with a carefully chosen *g*, produced the same results in terms of execution time of the algorithm. Furthermore, one could observe that the rooted tree-ring topology (*GM7*) promoted a statistically significant improvement of the fitness, unlike the other tested topologies. This suggests that one can choose any of the tested topologies, only considering the execution time of the algorithm. Considering both execution time and fitness, one should choose the rooted tree-ring topology (*GM7*). On the other hand, adding more nodes to our grid, allowed obtaining significant gains in terms of processing time, without a significant degradation of fitness. To sum up, a *hypothesis* aiming at answering the research question can be stated as follows:

For solving TSP instances using an order-based genetic algorithm, in a distributed heterogeneous grid computing environment, without losing significant fitness comparatively to the correspondent sequential panmictic implementation of the same algorithm, choose a coordinated Island Model topology, from any of the tested topologies (star, cartwheel, tree, fully connected multilayered, rooted tree-ring, ring), with as many nodes as possible (even slow ones) and select the migration frequency that optimizes the execution time for the chosen topology.

Considering that even slow nodes can contribute to speed up the processing time, in future work, it would be interesting to measure the contribution of each node to the running algorithm, determining a contribution rate directly related to the processing power of each computing node.

Although this work addressed several different *TSPLIB* data sets (from *51* to *280* cities), it would be very interesting to analyse the results for bigger data sets, such as *att532* (*532* cities), *pr1002* (*1002* cities) or even *pla85900* (*85.900* cities) which is the largest data set available from *TSPLIB*.

All the presented models can also be implemented using multithreading technology on multi-core machines, becoming widely available nowadays, which could reduce the potential communication delays associated to grid frameworks. Moreover, considering that each node of a computing grid can run a multithread PEA, one can conceive the execution of a PEA where each island applies, in turn, a PEA to its subpopulation using multithreading technology. Figure *5.4* shows an example of a possible island topology for a coordinated multithread PEA.

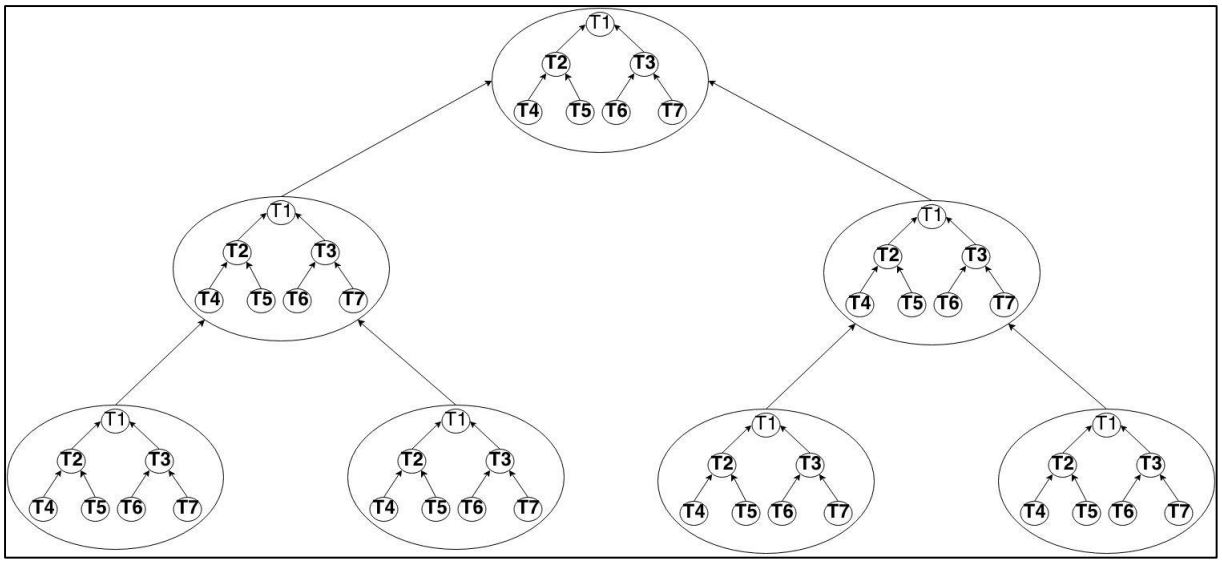


Figure 5.4: Coordinated multithread PEA example.

Considering the inexistence of a PEA framework for generic Globus-based grids providing different migration topologies for the island model, future work can also address the development of such a framework, based on the current work. That framework should be flexible enough to provide the execution of other PEAs than the presented in this work.

Future research work is intended to be a part of an international cooperation project between the Moroccan *Centre National pour la Recherche Scientifique et Technique* (CNRST) and the Portuguese *Fundação para a Ciência e a Tecnologia* (FCT), currently being evaluated by these institutions in both countries. This cooperation project aims:

a) The development of a software framework aiming at delivering support for deployment of a classical GA in parallel and distributed infrastructures (grid computing, clusters and symmetric multiprocessing). This implementation will be based on the knowledge produced by the current research;

b) The application of the proposed framework to analysis of raw data received continuously from the satellite *Eumetsat* (www.eumetsat.int) for early warning and mitigation of natural disasters.

References

Acampora, G., Gaeta, M., and Loia, V. (2011). Hierarchical optimization of personalized experiences for e-Learning systems through evolutionary models. *Neural Computing and Applications*, *20*(5), pp. 641-657.

Alba, E., and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *Complexity*, *4*(4), pp. 31-52.

Alba, E., and Troya, J. M. (2001). Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, *17*(4), pp. 451-465.

Alba, E., Nebro, A. J., and Troya, J. M. (2002). Heterogeneous computing and parallel genetic algorithms. *Journal of Parallel and Distributed Computing*, *62*(9), pp. 1362-1385.

Alba, E., Luna, F., Nebro, A. J., and Troya, J. M. (2004). Parallel heterogeneous genetic algorithms for continuous optimization. *Parallel Computing*, *30*(5), pp. 699-719.

Alba, E., and Tomassini, M. (2002). Parallelism and evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, *6*(5), pp. 443-462.

Alba, E., Luque, G., and Nesmachnow, S. (2013). Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, *20*(1), pp. 1-48.

Alba, E. (2002). Parallel evolutionary algorithms can achieve super-linear performance. *Inform. Process. Lett.* vol. 82, no. 1, pp. 7–13.

Alba, E. (2005). *Parallel metaheuristics: a new class of algorithms*. John Wiley & Sons.

Andalon-Garcia, I. R., and Chavoya, A. (2012). Performance comparison of three topologies of the island model of a parallel genetic algorithm implementation on a cluster platform. In *Electrical Communications and Computers (CONIELECOMP), 2012 22nd International Conference on*, pp. 1-6.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@ home: an experiment in public-resource computing. *Communications of the ACM*, *45*(11), pp. 56-61.

Anderson, D. P. (2004). Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pp. 4-10.

Applegate, D., Bixby, R., Chvatal, V. and Cook, W. (1998). On the Solution of the Traveling Salesman Problems. *Documenta Mathematica – Extra Volume ICM*, chapter 3, pp. 645-656.

Applegate, D., Bixby, R., Chvatal, V., and Cook, W. (2007). The Traveling Salesman Problem: A Computational Study. *Princeton Series in Applied Mathematics*. Princeton University Press.

Baugh, J. W. Jr., and Kumar, S.V. (2003). Asynchronous Genetic Algorithms for Heterogeneous Networks Using Coarse-Grained Dataflow. *Genetic and Evolutionary Computation GECCO 2003.* Ed. by Erick Cantú-Paz et al. Vol. 2723, pp. 730-741.

Bernabe, S. and Plaza, A. (2011). Commodity Cluster-Based Parallel Implementation of an Automatic Target Generation Process for Hyperspectral Image Analysis. *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 1038-1043.

Bernal, A., and Castro, H. (2010). pALS: an object-oriented framework for developing parallel cooperative metaheuristics. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on,* pp. 1-8.

Bernal, A., Ramirez, M. A., Castro, H., Walteros, J. L., and Medaglia, A. L. (2009). JG 2 A: A Grid-enabled object-oriented framework for developing genetic algorithms. In *Systems and Information Engineering Design Symposium, 2009. SIEDS'09,* pp. 67-72.

Borovska, P. (2006). Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster. In *Int. Conf. on Computer Systems and Technologies*, pp. 1-6.

Bremermann, H. J., Rogson, M., and Salaff, S. (1965). Search by evolution. *Biophysics and Cybernetic Systems*, pp. 157-167.

Brightwell, R., Fisk, L.A., Greenberg, D.S., Hudson, T., Levenhagen, M., Maccabe, A.B., and Riesen, R. (2000). Massively parallel computing using commodity components. *Parallel Computing* 26.2-3, pp. 243-266.

Brown, M. B., and Forsythe, A. B. (1974). Robust tests for the equality of variances. *Journal of the American Statistical Association*, *69* (346), pp. 364-367.

Cahon, S., Melab, N., and Talbi, E. G. (2005). An enabling framework for parallel optimization on the computational grid. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on* (Vol. 2), pp. 702-709.

Candan, C., Goëffon, A., Lardeux, F., and Saubion, F. (2012). A dynamic island model for adaptive operator selection. *Genetic and Evolutionary Computation Conference, GECCO '12*, Philadelphia, PA, USA, July 7-11, 2012, pp. 1253-1260.

Cantú-Paz, E. (1997). Designing efficient master-slave parallel genetic algorithms. Illinois Genetic Algorithms Laboratory Technical Report No. 97004.

Cantú-Paz, E. (1998). A Survey of Parallel Genetic Algorithms". Calculateurs Paralleles, *Reseaux et Systems Repartis 10*, pp. 141-171.

Cantú-Paz, E. (1999). Topologies, migration rates, and multi-population parallel genetic algorithms, in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, pp. 91–98.

Cantú-Paz, E. (2000). *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers.

Cantú-Paz, E. (2001). Migration policies, selection pressure, and parallel evolutionary algorithms. *Journal of heuristics*, *7*(4), pp. 311-334.

Cantú-Paz, E. (2007). Parameter setting in parallel genetic algorithms. In *Parameter setting in evolutionary algorithms*, pp. 259-276.

Cantú-Paz, E., and Meja-Olvera, M. (1994). Experimental results in distributed genetic algorithms. *International Symposium on Applied Corporate Computing*, pp. 99-108.

Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*. MIT press.

Chetty, M., and Buyya, R. (2002). Weaving computational Grids: How analogous are they with electrical Grids? *Computing in Science & Engineering*, *4*(4), pp. 61-71.

Cohoon, J. P., Hegde, S. U., Martin, W. N., and Richards, D. (1987). Punctuated Equilibria: A Parallel Genetic Algorithm. *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pp. 148-154.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, *6*(6), pp. 791-812.

Develder, C., De Leenheer, M., Dhoedt, B., Pickavet, M., Colle, D., De Turck, F., and Demeester, P. (2012). Optical networks for grid and cloud computing applications. *Proceedings of the IEEE*, *100*(5), pp. 1149-1167.

Dorronsoro, B., Arias, D., Luna, F., Nebro, A. J., and Alba, E. (2007). A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP. In *High Performance Computing & Simulation Conference (HPCS)*, pp. 759-765.

Durillo, J. J., Nebro, A. J., Luna, F., and Alba, E. (2008). A study of master-slave approaches to parallelize NSGA-II. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1-8.

Eklund, S. E. (2004). A massively parallel architecture for distributed genetic algorithms. *Parallel Computing*, *30*(5), pp. 647-676.

Filipowicz, B., Chmiel, W., Dudek, M., and Kadłuczka, P. (2011). Efektywność wielopopulacyjnego algorytmu ewolucyjnego dla zagadnień permutacyjnych. *Automatyka/Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie*, *15*, pp. 147-158.

Fogel, L. J. (1962). Autonomous automata. *Industrial Research*, *4*(2), pp. 14-19.

Foster, I., and Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, *11*(2), pp. 115-128.

Foster, I., Kesselman, C., and Tuecke, S. (2001) The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.* 15.3, pp. 200-222.

Glass, G.V., Peckham, P.D. and Sanders, J.R. (1972). Consequences of failure to meet assumptions underlying fixed effects analyses of variance and covariance. Rev. Educ. Res. 42, pp. 237-288.

Golberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. *Addison-Wesley*.

Gordon, V. S., Whitley, D. (1993). "Serial and parallel genetic algorithms as function optimizers," in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., pp. 177–183.

Grefenstette, J., Gopal, R., Rosmaita, B., and Van Gucht, D. (1985). Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pp. 160-168.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*. MIT press.

Gropp, W., Lusk, E., and Thakur, R. (1999). *Using MPI-2: Advanced features of the message-passing interface*. MIT press.

Guan, W., and Szeto, K.Y. (2013). Topological Effects on the Performance of Island Model of Parallel Genetic Algorithm. *Advances in Computational Intelligence.* Ed. By Ignacio Rojas, Gonzalo Joya, and Joan Cabestany. Vol. 7903, pp. 11-19.

Hart, W. E., Baden, S., Belew, R. K., and Kohn, S. (1996). Analysis of the numerical effects of parallelism on a parallel genetic algorithm. In *Parallel Processing Symposium, 1996, Proceedings of IPPS'96, The 10th International*, pp. 606-612.

Harwell, M.R., Rubinstein, E.N., Hayes, W.S. and Olds, C.C. (1992). Summarizing Monte Carlo results in methodological research: the one- and two-factor fixed effects ANOVA cases. J. Educ. Stat. 17, pp. 315-339.

He, H., Sýkora, O., Salagean, A., and Mäkinen, E. (2007). Parallelisation of genetic algorithms for the 2-page crossing number problem. *Journal of Parallel and Distributed Computing*, *67*(2), pp. 229-241.

He, H., Sýkora, O., and Salagean, A. (2006). Various Island-based Parallel Genetic Algorithms for the 2-Page Drawing Problem. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, as part of the 24th IASTED International Multi-Conference on Applied Informatics, February 14-16 2006, Innsbruck, Austria*, pp. 316-323.

Herrera, F., Lozano, M., and Moraga, C. (1998). Hybrid distributed real-coded genetic algorithms. In *Parallel Problem Solving from Nature—PPSN V*, pp. 603-612.

Holland, J. H. (1975). Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. U. Michigan Press.

Iturriaga, S., Nesmachnow, S., Dorronsoro, B., Talbi, E. G., and Bouvry, P. (2013). A parallel hybrid evolutionary algorithm for the optimization of broker virtual machines subletting in cloud systems. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pp. 594-599.

Jakobović, D., Golub, M., and Čupić, M. (2014). Asynchronous and implicitly parallel evolutionary computation models. *Soft Computing*, *18*(6), pp. 1225-1236.

Johar, F.M., Azmin, F.A., Suaidi, M.K., Shibghatullah, A.S., Ahmad, B.H., Salleh, S.N., Aziz, M.Z.A.A., and Md Shukor, M. (2013). A review of Genetic Algorithms and Parallel Genetic Algorithms on Graphics Processing Unit (GPU)". *Control System, Computing and Engineering (ICCSCE), 2013 IEEE International Conference on*, pp. 264-269.

Kesselman, C., and Foster, I. (1999). The grid: blueprint for a future computing infrastructure. *Chapter2, Morgan Kaufmann Publication*.

Knysh D.S. and Kureichik V.M. (2010). *Parallel genetic algorithms: a survey and problem state of the art*. Journal of Computer and Systems Sciences International 49.4, pp. 579-589.

Kruskal, W. H., and Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, *47*(260), pp. 583-621.

Lässig, J., and Sudholt, D. (2013). Design and analysis of migration in parallel evolutionary algorithms. *Soft Computing*, *17*(7), pp. 1121-1144.

Lawler, E. L., Lenstra, J. K., Kan, A. R., and Shmoys, D. B. (1985). The Traveling Salesman Problem: a guided tour of combinatorial optimization, volume 3.
Levene, H. (1960). Robust tests for equality of variances. *Contributions to probability and statistics: Essays in honor of Harold Hotelling*, *2*, pp. 278-292.

Lim, D., Ong, Y. S., Jin, Y., Sendhoff, B., and Lee, B. S. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, *23*(4), pp. 658-670.

Limmer, S., and Fey, D. (2010). Framework for distributed evolutionary algorithms in computational grids. In *Advances in Computation and Intelligence*, pp. 170-180.

Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal, The*, *44*(10), pp. 2245-2269.

Lin, S. C., Punch III, W. F., and Goodman, E. D. (1994). Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Parallel and Distributed Processing, 1994. Proceedings. Sixth IEEE Symposium on*, pp. 28-37.

Liu, C., Zhao, Z., and Liu, F. (2009). An Insight into the architecture of Condor-a Distributed Scheduler. In *Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on*, pp. 1-4.

Lix, L.M., Keselman, J.C. and Keselman, H.J. (1996). Consequences of assumption violations revisited: A quantitative review of alternatives to the one-way analysis of variance F test. Rev. Educ. Res. 66, pp. 579-619.

Lopes, R. A., Silva, R. C. P., Campelo, F., and Guimarães, F. G. (2013). Dynamic selection of migration flows in island model differential evolution. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pp. 173-174.

Lopes, R. A., Pedrosa Silva, R. C., Freitas, A. R., Campelo, F., and Guimarães, F. G. (2014). A study on the configuration of migratory flows in island model differential evolution. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, pp. 1015-1022.

Luna, F., Nebro, A. J., Alba, E., and Durillo, J. J. (2008). Solving large-scale real-world telecommunication problems using a grid-based genetic algorithm. *Engineering Optimization*, *40*(11), pp. 1067-1084.

Luque, G. and Alba, E. (2011). *Parallel genetic algorithms: Theory and real world applications.* Studies in Computational Intelligence 367. Berlin: Springer.

Luque, G., Alba, E., and Dorronsoro, B. (2009). An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 1395-1402.

Marin, F. J., Trelles-Salazar, O., and Sandoval, F. (1994). Genetic algorithms on lan-message passing architectures using PVM: Application to the routing problem. In *Parallel Problem Solving from Nature—PPSN III*, pp. 534-543.

Märtens, M., and Izzo, D. (2013). The asynchronous island model and NSGA-II: study of a new migration operator and its performance. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pp. 1173-1180.

Meffert, K., Meseguer, J., D Martı, E., Meskauskas, A., and Vos Neil Rotstan, J. (2011). JGAP—Java Genetic Algorithms Package. *URL http://jgap.sourceforge.net/. Last accessed: May 3rd 2014.*

Melab, N., Cahon, S., and Talbi, E. G. (2006). Grid computing for parallel bioinspired algorithms. *Journal of parallel and Distributed Computing*, *66*(8), pp. 1052-1061.

Melab, N., Mezmaz, M., and Talbi, E. G. (2006). Parallel cooperative meta-heuristics on the computational grid: A case study: the bi-objective flow-shop problem. *Parallel computing*, *32*(9), pp. 643-659.

Garey, M.R., and Johnson, D.S. (1979). Computers and intractability: a guide to the theory of NP-completeness. WH Freeman & Co., San Francisco.

Myers, D.S. and Cummings, M.P. (2003). Necessity is the mother of invention: a simple grid computing system using commodity tools". *Journal of Parallel and Distributed Computing 63.5. Special Issue on Computational Grids*, pp. 578-589.

Nebro, A. J., Luque, G., Luna, F., and Alba, E. (2008). DNA fragment assembly using a grid-based genetic algorithm. *Computers & Operations Research*, *35*(9), pp. 2776-2790.

Nowostawski, M., and Poli, R. (1999). Parallel genetic algorithm taxonomy. *Knowledge-Based Intelligent Information Engineering Systems, 1999. Third International Conference*, pp. 88-92.

Park, H. H., Grings, A., dos Santos, M. V., and Soares, A. S. (2008). Parallel hybrid evolutionary computation: automatic tuning of parameters for parallel gene expression programming. *Applied Mathematics and Computation*, *201*(1), pp. 108-120.

Pettey, C.C. and Leuze, M.R. (1989). A Theoretical Investigation of a Parallel Genetic Algorithm. *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 398–405.

Plaza, A., Valencia, D., Plaza, J., and Martinez, P. (2006). Commodity clusterbased parallel processing of hyperspectral imagery". *Journal of Parallel and Distributed Computing* 66.3, pp. 345-358.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. *Library Translation no. 1122, Ministry of Aviation, Royal Aircraft Establishment*, Farnborough, Hants, UK.

Rechenberg, I. (1973). "Evolutionsstrategie: Optimierung Technischer Systeme Nach Prinzipien der Biologischen Information". Frommann-Holzboog, Stuttgart.

Reinelt, G. (1991). TSPLIB—A traveling salesman problem library. *ORSA journal on computing*, *3*(4), 376-384.

Ruciński, M., Izzo, D., and Biscani, F. (2010). On the impact of the migration topology on the island model. *Parallel Computing*, *36*(10), pp. 555-571.

Rzeźniczak, T. (2012). Implementation aspects of data visualization based on map of attributes. *Journal of Theoretical and Applied Computer Science*, *6*(4), pp. 24-36.

Sadashiv, N., and Kumar, S. D. (2011). Cluster, grid and cloud computing: A detailed comparison. In *Computer Science & Education (ICCSE), 2011 6th International Conference on*, pp. 477-482.

Saiko, D. (2005). Traveling Salesman Problem – Java Genetic Algorithm Solution. *URL https://github.com/dsaiko/tsp. Last accessed: June 23rd 2014.*

Saračević, M., Mašović, S., and Plojović, Š. (2012). UML modeling for traveling salesman problem based on genetic algorithms. *SouthEast Europe Journal of Soft Computing*, *1*(2).

Sefrioui, M., and Périaux, J. (2000). A hierarchical genetic algorithm using multiple models for optimization. In *Parallel Problem Solving from Nature PPSN VI*, pp. 879-888.

Sekaj, I. (2004). Robust parallel genetic algorithms with re-initialization. In *Parallel Problem Solving from Nature-PPSN VIII*, pp. 411-419.

Sengoku, H., and Yoshihara, I. (1998). A fast TSP solver using GA on JAVA. In *Third International Symposium on Artificial Life, and Robotics (AROB III'98)*, pp. 283-288.

Shapiro, S. S., and Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). *Biometrika*, pp. 591-611.

Sipkova, V., and Tran, V. (2006). Glite Lightweight Middleware for Grid Computing. In *The 2nd International Workshop on Grid Computing for Complex Problems (GCCP'2006)*, pp. 228-241.

Skolicki, Z., and De Jong, K. (2005). The influence of migration sizes and intervals on island models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pp. 1295-1302.

Skolicki, Z., and De Jong, K. (2007). The importance of a two-level perspective for island model design. *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pp. 4623-4630.

Starkweather, T., Whitley, D., and Mathias, K. (1991). *Optimization using distributed genetic algorithms*, pp. 176-185.

Talbi, E. G., Cahon, S., and Melab, N. (2007). Designing cellular networks using a parallel hybrid metaheuristic on the computational grid. *Computer Communications*, *30*(4), pp. 698-713.

Tanese. R. (1987) “Parallel genetic algorithms for a hypercube,” in *Proc. 2nd Int. Conf. Genetic Algorithms*, J. J. Grefenstette, Ed., p. 177.

Tanese, R. (1989, December). Distributed genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pp. 434-439.

Tang, J., Lim, M. H., Ong, Y. S., and Er, M. J. (2004). Study of migration topology in island model parallel hybrid-GA for large scale quadratic assignment problems. In *Control, Automation, Robotics and Vision Conference, 2004. ICARCV 2004 8th* (Vol. 3), pp. 2286-2291.

Tantar, A. A., Melab, N., Talbi, E. G., Parent, B., and Horvath, D. (2007). A parallel hybrid genetic algorithm for protein structure prediction on the computational grid. *Future Generation Computer Systems*, *23*(3), pp. 398-409.

Thain, D., Tannenbaum, T., and Livny, M. (2003). Condor and the Grid. *Grid Computing: Making the Global Infrastructure a Reality*. pp. 299–335.

Umbarkar, A. J., and Joshi, M. S. (2013). Review of parallel genetic algorithm based on computing paradigm and diversity in search space. *ICTACT Journal on Soft Computing*, vol. *3*, pp. 615-622.

Getov, V., Von Laszewski, G., Philippsen, M., and Foster, I. (2001). Multiparadigm communications in Java for grid computing. *Communications of the ACM*, *44*(10), pp. 118-125.

Wagner, S., and Affenzeller, M. (2005). *Heuristiclab: A generic and extensible optimization environment*, pp. 538-541.

Wang, L., Maciejewski, A.A., Siegel, H.J., and Roychowdhury, V.P. (1998). A comparative study of five parallel genetic algorithms using the traveling salesman problem. *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pp. 345-349.

Wang, L., Maciejewski, A. A., Siegel, H. J., Roychowdhury, V. P., and Eldridge, B. D. (2005). A Study of Five Parallel Approaches to a Genetic Algorithm for the Traveling Salesman Problem. *Intelligent Automation & Soft Computing* 11.4, pp. 217-234.

Appendix A – Experimental Results

This work was supported by a wide set of experiments which generated a considerable amount of results. Those experimental results were the source to several hundreds of files, mostly graphical representations of the results. The most relevant figures are included in the text of this document. To access the full set of results, and their graphical representation, please refer to the "Experimental Results" folder in the digital support (CD-ROM).