



Università degli Studi di Salerno

Dipartimento di Scienze Aziendali – Management and Innovation Systems

Dottorato di Ricerca

Management and Information Technology

Curriculum in Informatica, Sistemi Informatici e Tecnologie del Software

XV Ciclo

Tesi di Dottorato

Parallel Genetic Algorithms in the Cloud

Tutor

Prof.ssa Filomena Ferrucci

Coordinatore

Prof. Andrea De Lucia

Candidato

Pasquale Salza

Matr. 8887600010

Anno Accademico 2015/2016

Thesis Reviewers

Prof. Giovanni Acampora, University of Naples 'Federico II', Italy

Prof. Juan Julián Merelo Guervós, University of Granada, Spain

Defense Committee

Prof. Giovanni Acampora, University of Naples 'Federico II', Italy

Prof. Anna Rita Fasolino, University of Naples 'Federico II', Italy

Prof. Filomena Ferrucci, University of Salerno, Italy

Date of defense

April 20th, 2017

Pasquale Salza

Department of Business Science, Management and Innovation Systems

University of Salerno

psalza@unisa.it

Abstract

Genetic Algorithms (GAs) are a metaheuristic search technique belonging to the class of Evolutionary Algorithms (EAs). They have been proven to be effective in addressing several problems in many fields but also suffer from scalability issues that may not let them find a valid application for real world problems. Thus, the aim of providing highly scalable GA-based solutions, together with the reduced costs of parallel architectures, motivate the research on Parallel Genetic Algorithms (PGAs).

Cloud computing may be a valid option for parallelisation, since there is no need of owning the physical hardware, which can be purchased from cloud providers, for the desired time, quantity and quality. There are different employable cloud technologies and approaches for this purpose, but they all introduce communication overhead. Thus, one might wonder if, and possibly when, specific approaches, environments and models show better performance than sequential versions in terms of execution time and resource usage.

This thesis investigates if and when GAs can scale in the cloud using specific approaches. Firstly, Hadoop MapReduce is exploited designing and developing an open source framework, i.e., elephant56, that reduces the effort in developing and speed up GAs using three parallel models. The performance of the framework is then evaluated through an empirical study. Secondly, software containers and message queues are employed to develop, deploy and execute PGAs in the cloud and the devised system is evaluated with an empirical study on a commercial cloud provider. Finally, cloud technologies are also explored for the parallelisation of other EAs, designing and developing cCube, a collaborative microservices architecture for machine learning problems.

Abstract

I Genetic Algorithms (GAs) sono una metaeuristica di ricerca appartenenti alla classe degli Evolutionary Algorithms (EAs). Si sono dimostrati efficaci nel risolvere tanti problemi in svariati campi. Tuttavia, le difficoltà nello scalare spesso evitano che i GAs possano trovare una collocazione efficace per la risoluzione di problemi del mondo reale. Quindi, l'obiettivo di fornire soluzioni basate altamente scalabili, assieme alla riduzione dei costi di architetture parallele, motivano la ricerca sui Parallel Genetic Algorithms (PGAs).

Il cloud computing potrebbe essere una valida opzione per la parallelizzazione, dato che non c'è necessità di possedere hardware fisico che può, invece, essere acquistato dai cloud provider, per il tempo desiderato, quantità e qualità. Esistono differenti tecnologie e approcci cloud impiegabili a tal proposito ma, tutti, introducono overhead di computazione. Quindi, ci si può chiedere se, e possibilmente quando, approcci specifici, ambienti e modelli mostrino migliori performance rispetto alle versioni sequenziali, in termini di tempo di esecuzione e uso di risorse.

Questa tesi indaga se, e quando, i GAs possono scalare nel cloud utilizzando approcci specifici. Prima di tutto, Hadoop MapReduce è sfruttato per modellare e sviluppare un framework open source, i.e., elephant56, che riduce l'effort nello sviluppo e velocizza i GAs usando tre diversi modelli paralleli. Le performance del framework sono poi valutate attraverso uno studio empirico. Successivamente, i software container e le message queue sono impiegati per sviluppare, distribuire e eseguire PGAs e il sistema ideato valutato, attraverso uno studio empirico, su un cloud provider commerciale. Infine, le tecnologie cloud sono esplorate per la parallelizzazione di altri EAs, ideando e sviluppan-

Abstract

do cCube, un'architettura a microservizi collaborativa per risolvere problemi di machine learning.

Acknowledgements

It would not have been possible to write this thesis without the help and support of many people during these three years of research. My sincere apologies to anyone I inadvertently omitted.

First and foremost I want to thank my tutor, Prof. Filomena Ferrucci. I have never met someone so diligent as she is. She has taught me a plenty of things, not only in the research field. I learned to never stop to appearances, always digging into things, and how to work hard and with passion. She has been supportive since the first days I began my Ph.D. entrusted and empowered me in many occasions. Without her precious support, it would not have been possible to conduct this research.

I also have to thank Dr. Una-May O'Reilly and Dr. Erik Hemberg from the ALFA Group, for the wonderful and intense experience I had at the MIT Computer Science and Artificial Intelligence Laboratory.

Thanks to Prof. Andrea De Lucia, Prof. Carmine Gravino, from the University of Salerno, and Dr. Federica Sarro from University College London. I thank my fellow labmates, now friends, of the SESA Lab, Fabio Palomba, Dario Di Nucci and Gemma Catolino, for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun experiences we had in the last three years.

There are many friends I should thank that supported me during the last years. It is a very long list and I will surely miss someone. This list comprises Roberto, Lorenza, Carmelo, Angela, Piergiorgio and Armando.

Last but not least, I would like to thank my family, my parents, my sister and brother, and my aunt Marianna. They supported me in any possible way.

Publications

Part of the ideas, images and data exposed in this thesis have been previously published, or are under review, in the following conferences and journals.

Published to International Conferences

[C1] Filomena Ferrucci, M-Tahar Kechadi, Pasquale Salza and Federica Sarro
A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce

ACM/SIGAPP Symposium on Applied Computing (SAC), 2015

[C2] Pasquale Salza, Filomena Ferrucci and Federica Sarro
elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce

Evolutionary Computation Software Systems Workshop at GECCO (EvoSoft), 2016

[C3] Pasquale Salza, Filomena Ferrucci and Federica Sarro
Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud
Genetic and Evolutionary Computation Conference (GECCO), 2016

[C4] Pasquale Salza, Erik Hemberg, Filomena Ferrucci and Una-May O'Reilly
cCube: A Cloud Microservices Architecture for Evolutionary Machine Learning Classification

Genetic and Evolutionary Computation Conference (GECCO), 2017

Submitted to International Conferences

- [C5] Pasquale Salza, Erik Hemberg, Filomena Ferrucci and Una-May O'Reilly
**Towards Evolutionary Machine Learning Comparison, Competition,
and Collaboration with a Multi-Cloud Platform**
Evolutionary Computation Software Systems Workshop at GECCO (EvoSoft),
2017

Submitted to International Journals

- [J1] Filomena Ferrucci, Pasquale Salza and Federica Sarro
**Using Hadoop MapReduce for Parallel Genetic Algorithms: A Com-
parison of the Global, Grid and Island Models**
- [J2] Pasquale Salza and Filomena Ferrucci
Speed Up Parallel Genetic Algorithms in the Cloud

Others

- [I1] Filomena Ferrucci, M-Tahar Kechadi, Pasquale Salza and Federica Sarro
A Framework for Genetic Algorithms Based on Hadoop
Computing Research Repository (CoRR), arXiv e-prints, 2013
- [I2] Pasquale Salza and Filomena Ferrucci
**An Approach for Parallel Genetic Algorithms in the Cloud using Soft-
ware Containers**
Computing Research Repository (CoRR), arXiv e-prints, 2016

Software Products

The systems proposed in this thesis have been publicly shared, under open source licences, at the following repositories.

elephant56

<https://github.com/pasqualesalza/elephant56>

elephant56 is a Genetic Algorithms (GAs) framework for Hadoop MapReduce with the aim of easing the development of distributed GAs. It provides high level functionalities which can be reused by developers, who no longer need to worry about complex internal structures.

elephant56 is able to:

- run sequential GAs;
- run parallel GAs based on the global, grid and island models;
- report the execution time and population evolution;
- provide sample individual and genetic operator implementations, e.g., number sequence individuals, roulette wheel selection, single point crossover.

The source code is shared under the terms of the *Apache License*, version 2.0¹.

¹<http://www.apache.org/licenses/LICENSE-2.0.html>

AMQPGA

<https://github.com/pasqualesalza/amqpga>

AMQPGA is an implementation in Go of the master/slave parallelisation model for Genetic Algorithms based on Docker and message queues.

The source code is shared under the terms of the *MIT License*².

²<https://opensource.org/licenses/MIT>

cCube

<https://github.com/ccube-uml>

cCube is a cloud microservices architecture for Evolutionary Machine Learning (EML) classification.

It is composed of the following microservices:

- **orchestrator**

<https://github.com/ccube-uml/orchestrator>

It is a client for cCube that creates and provisions the compute units, initiating and directing the symphony of microservices.

- **scheduler**

<https://github.com/ccube-uml/scheduler>

It accepts jobs scheduling through a REST interface.

- **factorizer**

<https://github.com/ccube-uml/factorizer>

It is a REST interface to the storage, currently PostgreSQL.

- **worker**

<https://github.com/ccube-uml/worker>

It provides the code for the learner, filter and fuser microservices as a single container. An EML researcher intended to use cCube has to inject the worker code into its execution environment.

- **gpfunction**

<https://github.com/ccube-uml/gpfunction>

An example of use is also provided: *GP Function*, a multi-objective Genetic Programming (GP) strategy based on NSGA-II.

The source code is shared under the terms of the *MIT License*³.

³<https://opensource.org/licenses/MIT>

Contents

1	Introduction	1
1.1	Motivations	1
1.2	Objectives	2
1.3	Contributions	4
1.4	Thesis Organisation	5
2	Background	9
2.1	Introduction	9
2.2	Genetic Algorithms	10
2.3	Parallel Genetic Algorithms	14
2.4	Cloud Computing	16
3	PGAs Using Hadoop MapReduce	19
3.1	Introduction	19
3.2	Related Work	22
3.3	Background	26
3.4	System Design	29
3.5	Usage	40
3.6	Empirical Study Design	48
3.7	Results	60
3.8	Summary	80
4	PGAs Using Software Containers	81
4.1	Introduction	81
4.2	Related Work	83

Contents

4.3	Background	85
4.4	System Design	89
4.5	Empirical Study Design	94
4.6	Results	100
4.7	Summary	109
5	Exploring Software Containers for Other EAs	111
5.1	Introduction	111
5.2	Related Work	115
5.3	System Design	118
5.4	Demonstration	127
5.5	Summary	131
6	Conclusions and Future Work	135
6.1	Thesis Summary	135
6.2	PGAs Using Hadoop MapReduce	136
6.3	PGAs Using Software Containers	138
6.4	Towards the Parallelisation of other EAs in the Cloud	139
	Acronyms	141
	Bibliography	143

Chapter 1

Introduction

Contents

1.1 Motivations	1
1.2 Objectives	2
1.3 Contributions	4
1.4 Thesis Organisation	5

1.1 Motivations

Genetic Algorithms (GAs) are powerful metaheuristics used to find near-optimal solutions to problems where the search for an optimal solution is too expensive. GAs and other search-based metaheuristics have been proven to be effective in addressing several problems in many fields. Nevertheless, it has been highlighted that attractive solutions in the laboratory may not find a valid application in practice due to scalability issues [27]. Thus, the aim of providing highly scalable GA-based solutions together with the reduced costs of parallel architectures motivate the research on Parallel Genetic Algorithms (PGAs) [38, 58].

Indeed, GAs can be parallelised in different ways [38]. For instance, their population-based characteristics allow evaluating in a parallel way the fitness value of each individual, giving rise to a PGA called ‘global parallelisation

model' or 'master-slave model'. Parallelism can also be exploited to perform genetic operators and thus to generate the next set of solutions. This model is named as 'island model', also called 'distributed model' or 'coarse-grained parallel model'. Furthermore, these two strategies can be combined, giving rise to a third form of parallelisation called 'grid model', also known as 'cellular model' or 'fine-grained parallel model'.

In the literature, different approaches and technologies have been investigated and employed, ranging from multi-core systems to the many-core systems on GPUs and cloud technologies [60, 58, 49, 46]. *Cloud computing* features are very appealing. There is no need of owning the physical hardware since it can be purchased as a service from cloud providers, for the desired time, quantity and quality. In cloud computing, it is required that the distributed application itself can handle scalability, load balancing and fault tolerance by implementing the proper architecture and communication protocols. Based on the fact that PGAs introduce communication overhead, one might wonder if, and possibly when, specific approaches, environments and models show better performance than sequential versions in terms of execution time and resource usage. This thesis investigates if and when GAs can scale in the cloud, exploring several approaches and technologies, e.g., Hadoop MapReduce and software containers.

1.2 Objectives

Even though the main aim of Hadoop MapReduce is rapidly processing vast amounts of data in parallel on large clusters of computing nodes, it represents one of the most mature and employed technologies for parallel algorithms, since it provides a ready to use distributed infrastructure that is scalable, reliable and fault-tolerant. All these factors have made Hadoop very popular both in industry and academia, also for PGAs. For this reason, in this thesis, it is first addressed the problem of realising a framework for Hadoop MapReduce, called 'elephant56', with the aim of easing the development of distributed GAs,

simplifying the interaction with the Hadoop API and implementing the three parallelisation models for PGAs. However, communication overhead could make Hadoop worthless in scaling GAs. Therefore, to understand which model performs better, an empirical study is conducted analysing the execution time, speedup and overhead.

The results of the study showed some limitations of Hadoop. In particular, a critical issue is the imposing presence of overhead due to the communication with the data store, especially for some PGAs models. Moreover, even if in the presence of a simplification through the use of a framework, the development of the GAs is limited to the utilisation of the Java programming language and also specific skills for setup and maintenance activities are expected. To cope with these limitations, in this thesis is described the design and implementation of another solution involving some technologies specially devised for the cloud, such as software containers and message queues. The message queues, adapted to the PGAs models, allows managing the communication and the execution of tasks easily. The software containers, instead, let GAs developers use existing implementations of genetic operators or external tools, without constraints on the adopted programming languages. Indeed, the software containers provide isolated environments where developers can include everything is needed for computation. This is why, in this thesis, a development, deployment and execution workflow is illustrated to suggest how to approach to the PGAs scalability problem, using software containers in the cloud. Moreover, the software containers can be quickly scheduled and replicated in a cloud cluster of multiple nodes and, for this reason, the requirements of scalability and fault-tolerance can be achieved. To understand if, and how much, software containers and message queues are effective in scaling GAs, an empirical study is conducted analysing the setup and execution time, speedup, and overhead when used on cloud clusters of many nodes.

Finally, the approach of software containers and DevOps practices for PGAs used for PGAs, are explored for the use with other Evolutionary Algorithms (EAs). In particular, Evolutionary Machine Learning (EML) is investigated,

where EAs are employed to build and optimise predictive models. To create accurate models, it is required that their training phase is based on a large representative sample of the known data, i.e., the training set, thus methodologies that involve the parallelisation of the algorithms for EML are useful. In this thesis, *cCube* is presented, an open source architecture that helps its users to develop an application that deploys EML algorithms to the cloud. A *cCube* EML application factors data, handles parameter configuration, tasks parallel classifier training with different algorithms, and follows training by filtering and fusing classifier results into a final ensemble model. In *cCube* researchers can run different EML algorithms, their own as well as others', developed in different programming languages, without inserting any code into them to accommodate cloud scaling. Instead of being monolithic, *cCube* has a *microservices* architecture [37], i.e., a suite of small services (microservices), each running its own process and communicating with lightweight protocols, e.g., HTTP resource API and message queues. A *cCube* application is developed and its deployment demonstrated on different clouds, utilising free resources, describing its employment on two cloud providers, using them both separately and together.

1.3 Contributions

The contributions of this thesis are mainly devoted to the research in the fields of EAs, GAs, parallelisation and cloud computing. Interesting issues that affect specific cloud approaches, i.e., Hadoop MapReduce and software containers, have been addressed. These contributions can be summarised as follows:

- **PGAs Using Hadoop MapReduce**
 - Design and develop the *elephant56* framework, an open source project supporting the development and execution of parallel GAs on Hadoop MapReduce.
 - Design the adaptation to MapReduce of the three PGAs models, i.e.,

global, grid and island models.

- Compare the three PGAs and sequential models through an empirical study, involving a cluster of machine on private cloud.

- **PGAs Using Software Containers**

- Design and realise a system to deploy containers of distributed GA applications in cloud environments, by implementing the global parallelisation model and exploiting Advanced Message Queuing Protocol (AMQP).
- Provide a conceptual workflow which describes all the phases of development, deployment and execution of distributed GAs.
- Carry out an empirical study to assess the effectiveness of the model regarding the execution time, speedup, overhead and setup time.

- **Exploring Software Containers for Other EAs**

- Design, develop and demonstrate *cCube*, an open source microservices architecture based on software containers that helps its users develop an application to deploy EML algorithms to the cloud.

A number of scientific articles have been published during the years in which this thesis has been developed that support and validate the impact of these contributions on the scientific community and literature. Furthermore, the majority of the software developed has been publicly shared in the form of open source projects with the hope to trigger further research.

1.4 Thesis Organisation

This thesis document is structured in five chapters. The first chapter provides the common and more general background needed to follow the rest of the thesis. The parallelisation of GAs with regards to Hadoop MapReduce and software containers, is addressed in the second and third chapters, respectively.

The software containers are explored for EML in the fourth chapter. Finally, the last chapter recaps the main conclusions drawn throughout the thesis and propose some future work.

- **Chapter 2. Background**

It introduces the background needed to follow the rest of the thesis. In particular, it provides an overview of GAs, the possible ways given from the literature to parallelise GAs, cloud computing and provision models.

- **Chapter 3. PGAs Using Hadoop MapReduce**

It addresses the problem of GAs parallelisation using the Hadoop MapReduce platform. First, it described the related work and Hadoop MapReduce platform as background. Then, *elephant56* is presented, which is the framework resulting from the design and implementation of the three models to parallelise GAs, also providing an example of use for a simple problem. The empirical study carried out to assess the effectiveness of the PGAs and comparing the three models concludes the chapter.

- **Chapter 4. PGAs Using Software Containers**

It addresses the GA parallelisation using software containers and cloud technologies, i.e., Docker, CoreOS. After a description of relevant related work, the main features of the employed cloud technologies are summarised. Then, the proposed system and conceptual workflow for deployment and execution of GAs in cloud environments are presented. The empirical study carried out to assess the effectiveness of the proposed approach concludes the chapter.

- **Chapter 5. Exploring Software Containers for Other EAs**

It explores software containers and other cloud technologies to address the parallelisation of other EAs, including GAs, to solve machine learning problems, i.e., EML. First, it provides a review of motivations and relevant related. The *cCube* open source platform, resulting from the implementation of the proposed system, is then described and demonstrated on multiple cloud providers.

- **Chapter 6. Conclusions and Future Work**

It contains a global review of the thesis work, revisiting the main conclusions drawn. The thesis objectives are then discussed in view of the results obtained, briefly sketching and discussing possible future lines of research.

Chapter 2

Background

Contents

2.1	Introduction	9
2.2	Genetic Algorithms	10
2.3	Parallel Genetic Algorithms	14
2.4	Cloud Computing	16

2.1 Introduction

This chapter contains the common and more general background needed to follow the rest of the thesis. However, further background sections are also provided in the other chapters, separating contents that are specific to the contexts.

An overview of Genetic Algorithms (GAs) is provided in Section 2.2. The possible ways given from the literature to parallelise GAs are describe in Section 2.3, and Section 2.4 provides an overview of cloud computing and provision models.

2.2 Genetic Algorithms

GAs, first introduced by Goldberg [24], are a metaheuristic search technique belonging to the class of Evolutionary Algorithms (EAs). These, inspired by natural evolution, create consecutive populations of individuals, considered as feasible solutions for a given problem, to search for an optimal solution for the problem under investigation guided by a fitness function. GAs simulate several aspects of the Darwin's 'Theory of Evolution' such as natural 'selection', sexual 'reproduction' and 'mutation'.

A first overview about the 'Search Algorithms' typology, where GAs belong to, is given in Section 2.2.1. Section 2.2.2 illustrates how GAs work.

2.2.1 Search Algorithms

Search Algorithms explore large or infinite spaces in order to look for an approximately optimal solution, according to a cost function, i.e., the 'objective function'. This function assigns a numerical value to solutions and allow to compare them . They can be mostly divided in two subcategories, namely *Local Search Algorithms* and *Global Search Algorithms* [45].

By using a very small amount of memory, Local Search Algorithms start from an initial solution to a better solution, according to the objective function. If the problem is a problem of maximum, the best solution is the one for which the value of the objective function is the highest peak; on the other hand, with a problem of minimum, the best solution is the lowest valley. A possible issue with local search is that it is possible to get stuck into a local maximum (or minimum).

A simple algorithm of local search is the 'Hill Climbing Algorithm' (see Algorithm 1), which starts from an initial solution exploring the local space of the current solution. It chooses the next solution, if it exists, which maximizes the value of the objective function. The algorithm stops when it reaches a local maximum or when it exceeds a predetermined number of executions. It is clear that the choice of the initial solution is determinant and there is no way

Algorithm 1 Hill Climbing Algorithm.

```

1: function HILLCLIMBING(problem)
2:   current ← MAKENODE(problem.initialState)
3:   loop
4:     neighbour ← a highest-valued successor of current
5:     if neighbour.value ≤ current.value then
6:       return current.state
7:     current ← neighbour

```

to escape from a local maximum. For this reason, a mechanism to shake things up is necessary.

If the problem of memory is not so extreme, it is possible to use a parallel version of Hill Climbing called ‘Local Beam Search’. It begins with k randomly generated solutions and, at each step, all the successors of all k solutions are generated and collected in a complete list. Then, the algorithm chooses the best k solutions from the previous and current generated solutions to continue the search. At every generation, some useful information is passed among parallel threads, so that the algorithm abandons unfruitful searches and uses its resources for better solutions. Although execution time is better used, chances of bumping into a local optimum are still as high as for Hill Climbing. A variant called ‘Stochastic Beam Search’ provides the element of randomness to exit from the possible block in a local optimum. Instead of choosing the next best k solutions from candidate ones, it chooses them randomly, where the probability of choosing a giving successor is an increasing function of its value, with respect to the objective function.

Global Search Algorithms do not require an initial solution and their main goal is to find the global optimum for the ‘objective function’. While Local Search Algorithms promote the ‘intensification’ of solutions, Global Search Algorithms promote mostly the ‘diversification’. The former aims at improving the quality, in terms of objective function, of the solutions already explored within the search space. The latter, allows to explore new areas of the solutions space.

GAs can be considered both local and global, since they balance the tradeoff

between intensification and diversification. GAs differ from their more traditional cousins in some points:

- work with a coding of the parameter set, not the parameters themselves;
- search from a population of solutions, not a single point;
- use the objective function information, not auxiliary knowledge;
- use probabilistic transition rules, not deterministic rules.

2.2.2 On the Origin of Genetic Algorithms

The ‘Theory of Evolution’ was developed by Charles Darwin in ‘On the Origin of Species by Means of Natural Selection’ [10]. The central Darwin’s hypothesis is that all organisms derive from one type or few types of simple primitive organisms, having diversified in adapting themselves to different environments. The ‘adaptation’ and the ‘diversification’ are explained with the combination of two phenomena: the pressure applied by natural ‘selection’ and the ‘mutation’ in the context of a certain type of organisms. If a certain group of individuals of the same species change so much the environment in which it lives, it could accelerate the rate of evolution [6]. This means that the natural *selection* rewards the *mutation* that make the individual more suitable to live in its environment. All the features above mentioned are implemented in an original way in GAs.

In GAs the solutions are generated by two parents simulating the so-called ‘sexual reproduction’. Figure 2.1 shows a typical GA execution consisting of the following steps:

- a. GA begins with a set of k randomly generated solutions called ‘population’. Each solution is called ‘individual’ and is represented as a string over a finite alphabet. This string is called ‘chromosome’ and each symbol ‘gene’;
- b. during the ‘fitness evaluation’, every individual is evaluated according to the objective function, i.e., the ‘fitness function’ in this context;

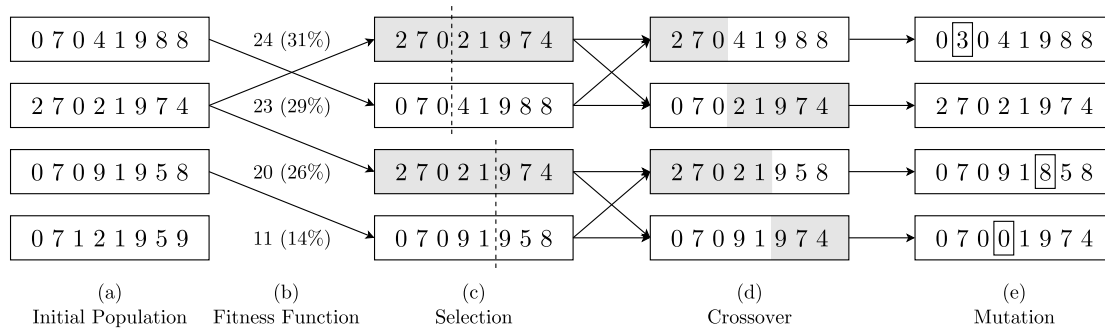


Figure 2.1: The execution of a GA.

- c. with the ‘selection’, the individuals are put in couples according to some criteria based on the fitness values;
- d. the ‘crossover’ mixes individuals and produces two or more children inheriting part of the genes from the parents;
- e. with the ‘mutation’, when the ‘offspring’ is generated, each gene is subjected to a random ‘mutation’ with a certain independent probability.

These steps, defined as ‘genetic operators’, are repeated over time for a certain number of ‘generations’, until some stopping criteria hold. The individual that gives the best solution in the final population is taken to define the best approximation to the optimum for the problem under investigation. The pseudo code of a possible version of GA is showed in Algorithm 2.

Algorithm 2 Genetic Algorithm.

```

1: function GENETICALGORITHM(problem)
2:   population ← GENERATEINITIALPOPULATION(problem)
3:   repeat
4:     EVALUATEFITNESS(population)
5:     population1 ← SELECTPARENTS(population)
6:     population1 ← MAKECROSSOVER(population1)
7:     population1 ← APPLYMUTATIONS(population1)
8:     population ← population1
9:   until TERMINATIONCONDITION( )
10:  return the best solution found

```

The analysis of this process suggests that the following design choices have to be made for tailoring a GA to a given optimisation problem:

1. define the chromosome for representing a solution (i.e., the 'solution encoding') and the number of initial solutions (i.e., the 'population size');
2. choose the criterion (i.e., the 'fitness function') to measure the goodness of a chromosome;
3. define the combination of genetic operators to explore the search space;
4. define the stopping criteria.

2.3 Parallel Genetic Algorithms

The following models have been proposed in literature [38] for Parallel Genetic Algorithms (PGAs):

- 'global model', also called 'master-slave model';
- 'grid model', also called 'cellular model' or 'fine-grained parallel model';
- 'island model', also called 'distributed model' or 'coarse-grained parallel model'.

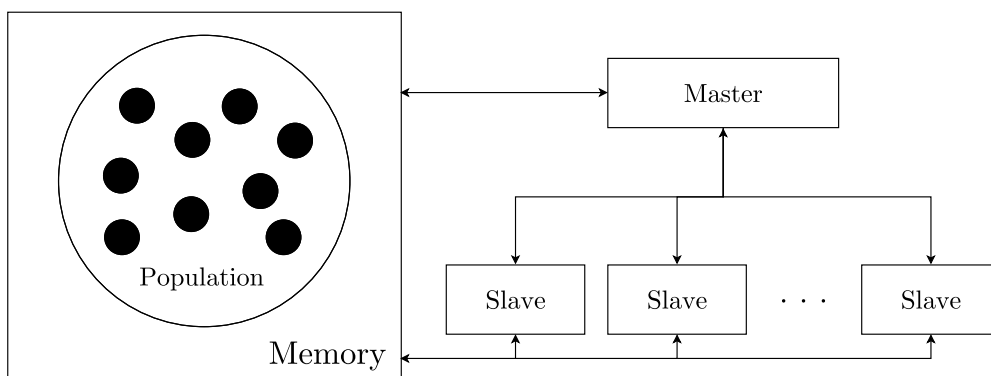


Figure 2.2: The PGA global model.

In the global model (Figure 2.2), there are two primary roles: a master and some slaves. The former is responsible for managing the population (i.e., applying genetic operators) and assigning the individuals to the slaves. The slaves are in charge to evaluate the fitness of each individual. This model does

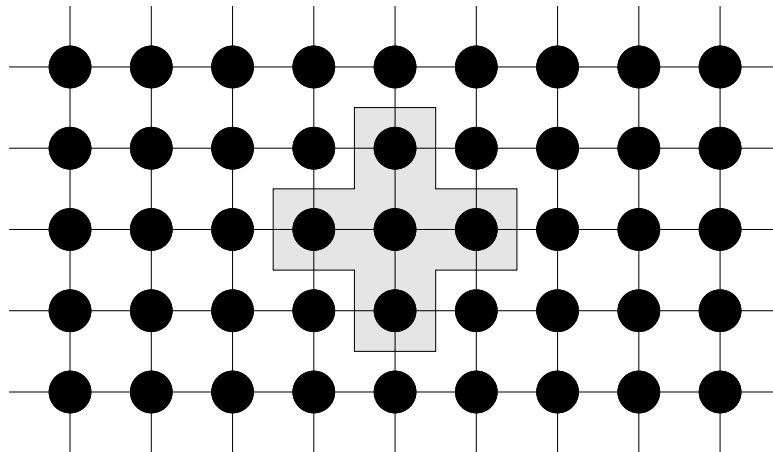


Figure 2.3: The PGA grid model.

not require any changes to the sequential GA since the fitness computation for each individual is independent and thus can be achieved in parallel.

The grid model (Figure 2.3) applies the genetic operators only to portions of the global population (i.e., ‘neighbourhoods’), obtained by assigning each individual to a single node and by performing evolutionary operations also involving some neighbours of a solution. The effect is an improvement of the diversity during the evolutions, further reducing the probability to converge into a local optimum. The drawback is requiring higher network traffic, due to the frequent communications among the nodes.

In the island model (Figure 2.4), the initial population is split into several groups and on each of them, typically referred to as ‘islands’, the GA proceeds independently and periodically exchanges information between islands by ‘migrating’ some individuals from one island to another. The main advantages of this model are that different sub-populations can explore different parts of the search space and migrating individuals among islands enhances the diversity of the chromosomes, thus reducing the probability to converge into a local optimum.

These models show that the parallelisation of GAs is straightforward from a conceptual point of view. However, setting up an actual implementation may be not so trivial due to some common development difficulties that a programmer must tackle in a distributed environment. Probably these limitations have

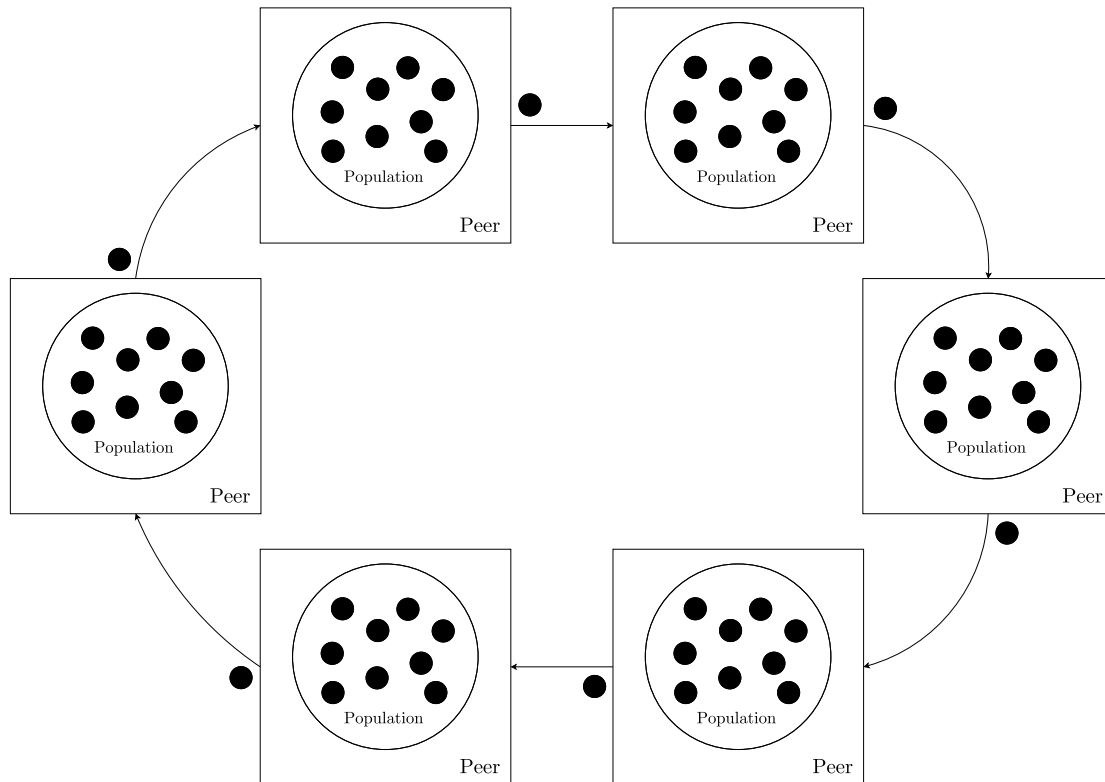


Figure 2.4: The PGA island model.

slowed down the use of parallel GAs for software related tasks.

2.4 Cloud Computing

Cloud computing is a good alternative in the field of parallelisation. There is no need of owning the physical hardware since it can be purchased as a service in the form of virtual instances from cloud providers, for the desired time, quantity and quality. The cloud capability of allocating on demand resources meets many of the requirements of an effective distributed application. It offers the 'scalability', i.e., the capability of enlarging the number of the computational units. It can reduce the execution time by splitting the computational load between multiple nodes. It improves the reliability of the whole system with regards to load balancing of both the requests and problem complexity growth, i.e., load balancing. Furthermore, cloud computing can guarantee fault tolerance in the case of physical or logical failures.

There are three provision models of cloud computing, as shown in Figure 2.5,

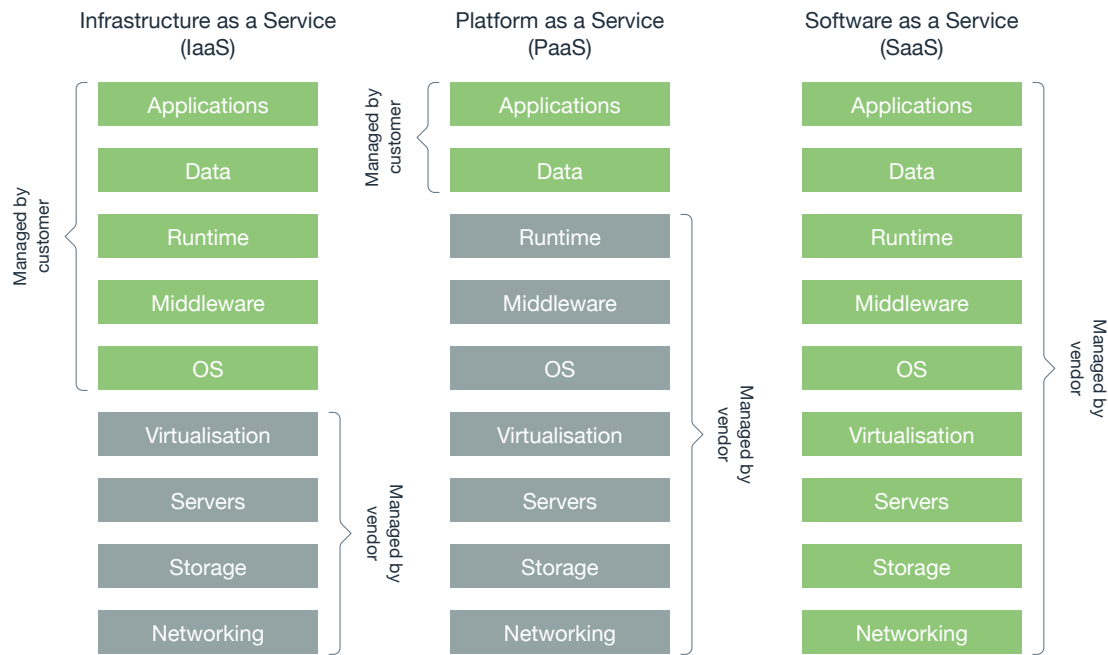


Figure 2.5: The three cloud provision models.

in which the customers and cloud vendors play a different role and have different responsibilities in the management of various aspects: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. In IaaS, the cloud vendor owns the hardware and network and it is responsible for housing, running, and maintenance aspects. The customers are allowed to use virtual infrastructures (i.e., cloud instances), which run on physical resources but that can be created, reconfigured, resized, and removed in a few moments based on the customers' or distributed applications' needs. The cloud instances consist of virtual machines for which the customer has full control of the environment. Before deploying a distributed application, the customer needs to install an operating system and the software stack.

In PaaS, a platform is provided to customers that can run their applications and business in a distributed environment, without having to deal with lower level requirements such as configuration, security, and management aspects. PaaS abstracts away all the aspects related to hardware decisions. Some examples of PaaS are databases, development tools and web server services.

SaaS is the top layer of cloud computing. The cloud vendor supplies software to customers in the form of a service such as e-mail clients, virtual

desktop, and communication services.

Hadoop MapReduce can be run on IaaS if it is installed from scratch. Indeed, it is possible to install Hadoop on a large variety of operating systems. However, it is also possible to get Hadoop in the form of PaaS, meaning that an Hadoop cluster is already installed and provided, and a computation job can be directly run on it. As for software containers, it can be done a similar discourse: containers can be run on Linux virtual machines (i.e., IaaS) and, also, many vendors also provide containers allocation as a service (i.e., PaaS). The specific uses of the cloud and technologies will be better detailed in the following chapters of the thesis.

Chapter 3

Parallel Genetic Algorithms Using Hadoop MapReduce

Contents

3.1	Introduction	19
3.2	Related Work	22
3.3	Background	26
3.4	System Design	29
3.5	Usage	40
3.6	Empirical Study Design	48
3.7	Results	60
3.8	Summary	80

3.1 Introduction

Hadoop MapReduce represents one of the most mature technologies to develop parallel algorithms since it provides a ready to use distributed infrastructure that is scalable, reliable and fault-tolerant [30]. It is able of rapidly processing vast amounts of data in parallel on large clusters of computing nodes. All these factors have made Hadoop very popular both in industry and academia. From

an industry perspective, Hadoop has been widely adopted as an instrument for big data processing, such as data mining, data analytics and search engine [44]. Furthermore, Hadoop has been positively adopted from the research community [44, 30].

Motivated by the success of Hadoop MapReduce in many fields, several researchers have experimented in the last years its use to parallelise Genetic Algorithms (GAs). Nevertheless, it is well known that parallel solutions introduce communication overhead that could make Hadoop be worthless in scaling GAs. One might wonder if, and possibly when, Hadoop shows better performance than sequential versions in terms of execution time. Moreover, a GA developer is interested in understanding which GA parallel model can be more effective. Indeed, GAs can be parallelised in different ways [38] (see Section 2.3). The work described in this chapter aims at analysing and compare of the three models using Hadoop MapReduce.

First, to implement the three Parallel Genetic Algorithms (PGAs), the *elephant56* framework [18, 47] was first developed, which is an open source project supporting the development and execution of parallel GAs. The source code is shared at the address <https://github.com/pasqualesalza/elephant56>, under the terms of the *Apache License*, version 2.0¹. It provides high level functionalities that can be reused by developers, who no longer need to worry about complex internal structures. In particular, it offers the possibility of distributing the GAs computation over a Hadoop MapReduce cluster of multiple computers.

Then, to compare the three PGAs models using MapReduce, an empirical study was carried out by applying them to a challenging software engineering problem that has been already addressed with a sequential GA. In particular, GAs was used to search for a suitable configuration of Support Vector Machines (SVMs) for inter-release fault prediction. Indeed, it has been shown that the estimation accuracy of SVMs, and more in general of machine learning approaches, heavily depends on the selection of a suitable configuration [21, 51,

¹<http://www.apache.org/licenses/LICENSE-2.0.html>

13, 50]. However, a complete search of all possible combinations of parameters values may not be feasible due to the large search space. To this aim, the use of GAs has already been proposed to configure SVMs for software fault prediction [13, 28, 48] but the combination of the two techniques may affect the scalability of the proposed approach when dealing with large software projects. For this reason, the parallelisation of GAs may be exploited to address these scalability issues. The choice of this as a benchmark problem was motivated also by the consideration that it is possible to have different problem instances by varying the size of the input datasets thus allowing to experiment different computational loads.

The three PGA solutions and the sequential version (i.e., Sequential Genetic Algorithm (SGA)) were compared to understand their effectiveness in terms of execution time and speedup. Then, the behaviour of the three parallel models in relation to the overhead produced using Hadoop MapReduce were studied. To give a more machine-independent measure of the algorithms, also the absolute number of fitness evaluations as a measure of the computational effort is provided. The empirical study was carried out executing the experiments simultaneously on a cluster of 150 Hadoop nodes. The experiments were conducted by varying the size of the problems, which consisted of exploiting three datasets with different sizes and by varying the cluster sizes. A total of 30 runs were executed for every single experiment of 300 generations each, with a total running time of about 120 days. Finally, the estimation of costs for the major commercial cloud provider, when executing the same GAs of the proposed experiments, is provided. The study allowed to identify the best model and highlighted some critical aspects.

The chapter is organised as follows. Section 3.2 describes the related work while Section 3.3 first illustrates Hadoop MapReduce platform as background. Section 3.4 presents elephant56 and the employed approach to parallelise GAs by exploiting the Hadoop MapReduce platform. In Section 3.5 is given an example of use of elephant56 for a simple problem. Sections 3.6 and Section 3.7 report, respectively, the design and the results of the empirical study carried

out to assess the effectiveness of the PGAs. Finally, Section 3.8 contains some concluding remarks.

3.2 Related Work

This section reports most relevant work that inspired and guided this study, highlighting similarity and differences. The main interest was about the solutions involving the MapReduce paradigm but also some important work, employing different technologies to run PGAs, is reported.

3.2.1 Parallel Genetic Algorithms Based on MapReduce

In Table summarises 3.1 the work related to the use of MapReduce for PGAs.

Jin et al. [32] were the first to use MapReduce to parallelise GAs. They implemented their specific version of MapReduce on the *.Net* platform and realised a parallel model, which can be considered as a sort of the grid model described in Section 2.3. The mapper nodes compute the fitness function and the selection operator chooses the best individuals on the same machine. A single reducer applies the selection on all the best local individuals received from the parallel nodes. The computation continues on the master node where crossover and mutation operators are applied to the global population. The authors highlighted the worrying presence of overhead and the best efficacy in case of heavy computational fitness work.

The first work exploiting Hadoop as a specific implementation of MapReduce is by Verma et al. [54]. The implemented model is the grid model where the mappers execute the fitness evaluation and the unpaired reducers the other genetic operators for the individuals they receive as input randomly. They studied the scalability factor on a large cluster of Hadoop nodes and they found a clear decrease in performance only when the number of requested nodes surpassed the number of physical CPUs available on the cluster. They confirmed that GAs can scale on multiple nodes, especially with large population size.

Table 3.1: Relevant related work about scaling GAs using MapReduce.

Title	Year	PGA Model			Experimentation		Results
		Global	Grid	Island	Hardware	Problems	
MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms [32]	2008		✓		Private cluster	DLTZ4, DLTZ5 and the Aerodynamic Airfoil Design Simulation	MapReduce suits the GAs parallelisation, demonstrated by experimenting the grid model, but the paradigm needs to be adapted to PGAs models
Scaling Genetic Algorithms Using MapReduce [54]	2009	✓			Private cluster	OneMax	Hadoop MapReduce is able to reduce the execution time of GAs by using a PGA based on the global model on multiple nodes for large populations
Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems Using MapReduce [31]	2010	✓			Academic cloud and Amazon EC2	Job Shop Scheduling	Hadoop is effective when applied to problems with intensive computation work or with large populations using the PGAs based on the global model, due to a very imposing presence of overhead
A Library to Run Evolutionary Algorithms in the Cloud Using MapReduce [17]	2012		✓		Amazon EC2	Genetic Programming Regression problem of dimensionality two	The effort of developing parallel EAs is simplified with the use of libraries/frameworks
A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites [12]	2012	✓			Private cluster	Automatic JUnit Test Suites Generation	A hard real-world problem can be solved by PGAs based on the global model and HDFS is probably the main culprit of the overhead
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud [14]	2013	✓			Google App Engine MapReduce	Automatic Test Data Generation	The use of the cloud can heavily outperform the performances of a local server when using a PGA based on the global model against a sequential GA
Adapting MapReduce Framework for Genetic Algorithm with Large Population [34]	2013			✓	Private cluster	Traveling Salesman Problem	HDFS and Hadoop orchestration operations are the main reasons for overhead when executing a PGA based on the global model
A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce [18]	2015			✓	Amazon EC2	Feature Subset Selection (FSS)	The FSS problem can be effectively solved by using a framework to define and execute PGAs based on the island model on Hadoop MapReduce

Huang and Lin [31] exploited Hadoop MapReduce to implement the global model of GAs to solve the *Job Shop Scheduling* problem on a large private grid of slow machines in order to measure the performance in terms of quality at varying the number of nodes. They also exploited an *Amazon EC2* cluster of faster machines to analyse the performance in terms of execution time. They found very imposing the presence of overhead, especially during the Hadoop job orchestration, and suggested the use of Hadoop MapReduce in GAs parallelisation in the presence of large populations and intensive computation work for the fitness evaluation.

As for an example of application of these methodologies to real-world problems, Di Gironimo et al. [12] were the first to propose a parallel GA for *JUnit* test suite generation based on the global parallelisation model. A preliminary evaluation of the proposed algorithm was carried out aiming at evaluating the speedup with respect to a sequential GA. The obtained results highlighted that using the PGA allowed for saving over the 50% of the time. The algorithm was developed exploiting Hadoop MapReduce and its performance were assessed on a standard cluster. In analysing the overhead time, they considered the Hadoop distributed filesystem (i.e., Hadoop Distributed File System (HDFS)) as the main cause.

Di Martino et al. [14] also investigated how to migrate GAs to the cloud in order to speed up the automatic generation of test data for software projects. They were the first to design the adaptation of three parallelisation models to MapReduce paradigm for automatic test data generation. However, they experimented only the solution based on global model taking advantages of the *Google App Engine* framework. Preliminary results showed that, unless for toy examples, the cloud can heavily outperform the performances of a local server.

Khalid et al. [34] used the island model to solve the *Travelling Salesman Problem* on the Hadoop MapReduce platform. They focused their attention on the scalability factor of the population size. They noticed that the population in a large solutions space, as the one of their problem, can be scaled to multiple

nodes and different sizes. Using a single job for each GA generation and measuring performances, they reckoned the time to orchestrate Hadoop MapReduce jobs and HDFS operations as the main reasons for overhead.

Fazenda et al. [17] were the first to consider the parallelisation of Evolutionary Algorithms (EAs) on the Hadoop MapReduce platform in the general purpose form of a library, in order to simplify the developing effort for parallel EA implementations. The work has been further enhanced by Sherry et al. to produce *FlexGP* [49, 53]. It is probably the first large scale Genetic Programming (GP) system that runs on the cloud implemented over Amazon EC2 with a socket-based client/server architecture.

As can be seen from the Table 3.1, no work realised a comparison of all the three models using Hadoop MapReduce.

3.2.2 Parallel Genetic Algorithms

In the literature many proposals of PGAs using different approaches, methods and technologies can be found. It is important to note that not all the reported work is strictly related to the traditional models compared in this study. However, the following studies describe approaches and results that influenced this work.

Zheng et al. [60] addressed a research question similar to the one of this study, focusing on the global and the island models, using a multi-core (i.e., CPUs) and a many-core (i.e., GPUs) systems. Their parallel algorithm did not use the MapReduce paradigm. However, also in this case, the island model provided better results with respect to the global in terms of quality and execution time. Even though they found the system based on GPUs is faster than the CPUs one, they observed that an architecture with a fixed number of parallel participants and a strict parallelisation schema, such as GPU cores, might perform worse in terms of quality of solutions than another with more parallel nodes and the possibility of communicating (e.g., multi-threading). They stated that a distributed architecture is worth for GAs parallelisation.

As a first attempt of employing cloud technologies, Merelo Guervós et al.

devised *SofEA* [41], a model for Pool-based EAs in the cloud, an evolutionary algorithm mapped to a central *CouchDB* object store. *SofEA* provides an asynchronous and distributed system for individuals evaluations and genetic operators application. Later, they defined and implemented the *EvoSpace Model* [23], consisting of two main components: a repository storing the evolving population and some remote workers, which execute the actual evolutionary process. The study shows how EAs can scale on the cloud and how the cloud can make EAs effective in a real world environment, speeding up the running time and lowering the costs.

3.3 Background

In this Section some background about Hadoop MapReduce in the literature to parallelise GAs is given.

3.3.1 Hadoop MapReduce

MapReduce is a programming paradigm whose origins lie in the old functional programming. It was adapted by Google [11] as a system for building search indexes, distributed computing and large scale databases. It was originally written in C++ language and was made as a framework, in order to simplify the development of its applications. It is expressed in terms of two distinct functions, namely ‘map’ and ‘reduce’, which are combined in a divide-and-conquer way where the map function is responsible for handling the parallelisation while the reduce collects and merges the results. In particular, a master node splits the initial input into several pieces, each one identified by a unique key and distributes them via the map function to several slave nodes (i.e., mappers), which work in parallel and independently from each other performing the same task on a different piece of input. As soon as each mapper finishes its job the output is identified and collected via the reducer function. Each mapper produces a set of intermediate key/value pairs, which are exploited by one or more reducers to group together all the intermediate values associated with

the same key and to compute the list of output results.

Hadoop is one of the most famous products of the *Apache Software Foundation* family. It was created by Doug Cutting and has its origins in *Apache Nuts*, an open source web search engine. In January 2008 Hadoop was made a top-level project at Apache Software Foundation, attracting to itself a large active community, including *Yahoo!*, *Facebook* and *The New York Times*. At present, Hadoop is a solid and valid presence in the world of cloud computing. Hadoop includes an implementation of the *MapReduce* paradigm and the HDFSs, which can be run on large clusters of machines. Currently, Apache introduced a new version of MapReduce (*MapReduce 2.0*), moving the Hadoop platform on a bigger one also known as Yet Another Resource Negotiator (YARN). Not only is it possible to execute distributed MapReduce applications, but YARN is also comprehensive of a large family of other Apache distributed products.

Hadoop provides some interesting features: scalability, reliability and fault-tolerance of computation processes and storage. These characteristics are indispensable when the aim is to deploy an application to a cloud environment. Moreover, Hadoop MapReduce is well supported to work not only on private clusters but also on a cloud platform (e.g., *Amazon Elastic Compute Cloud*) and thus is an ideal candidate for high scalable parallelisation of GAs.

Hadoop MapReduce exploits a distributed file system (an open source implementation of the Google File System), named HDFS, to store data as well as intermediate results for MapReduce jobs. The Hadoop MapReduce interpretation of the distributed file system was conceived to increase large-data availability and fault-tolerance by spreading copies of the data throughout the cluster nodes, to achieve both lower costs (for hardware and RAID disks) and lower data transfer latency between the nodes themselves.

A Hadoop cluster is allowed to accept MapReduce executions, i.e., 'jobs', in a batch fashion. Usually, a job is demanded from a master node, which provides both the data and configuration for the execution on the cluster. A job is intended to process input data and produce output data exploiting HDFS and is composed of the following main phases, also described in Figure 3.1:

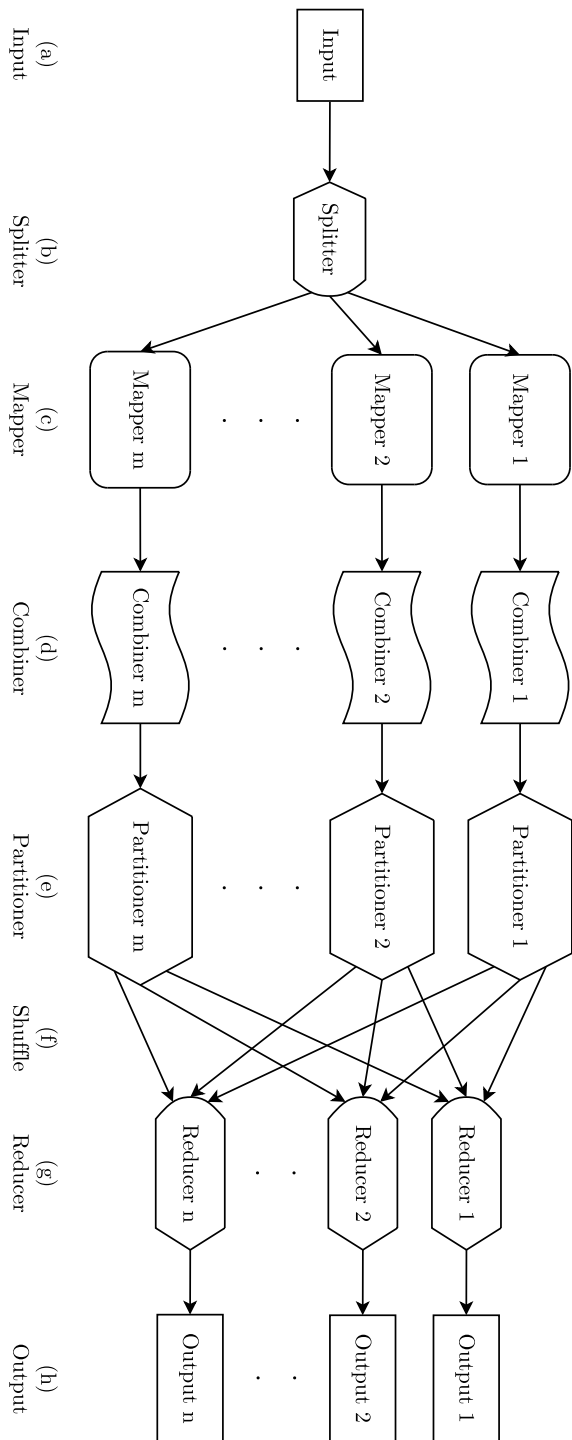


Figure 3.1: The Hadoop MapReduce phases.

Split: the input data (a) is usually in the form of one or more files stored in the HDFS. The splits of key/value pairs called ‘records’ are distributed to the mappers available on the cluster (b). The function, where k_1 and v_1 indicate data types, is described as:

$$input \rightarrow \text{list}(k_1, v_1)_S$$

Map: this phase is distributed on different nodes (c). For each input split, it produces a list of records:

$$(k_1, v_1)_S \rightarrow \text{list}(k_2, v_2)_M$$

Also, a ‘Combine’ phase (d) can be set before the *Partition* for local reducing.

Partition: it is in charge of establishing to which reduce node sending the map output records (e):

$$k_2 \rightarrow \text{reducer}_i$$

Reduce: after a *Shuffle* phase (f) that sort the records, the *Reducer* (g) processes the input for each group of records with the same key and stores the output into the HDFS (h):

$$(k_2, \text{list}(v_2))_M \rightarrow \text{list}(k_3, v_3)_R$$

A developer is expected to extend some particular Java classes to define each job phase.

3.4 System Design

This section describes the design of *elephant56* [18, 47], the framework supporting the development and execution of parallel GAs using Hadoop MapReduce.

Hadoop distributes software applications (i.e., ‘jobs’) on a cluster of nodes by using the Java Virtual Machine (JVM) containers for the computation and the HDFS for the passage of data. A Hadoop cluster is usually composed of

one master node (i.e., 'ResourceManager' for the Hadoop terminology), which manages the work of the other computation slave nodes (i.e., 'NodeManagers'). Hadoop MapReduce is strictly related to the HDFS, which provides scalable and reliable data storage, by replication of data blocks all over the machines involved in the cluster. The file system is managed by the 'NameNode' component controlling the slave 'DataNodes'. elephant56 is in charge of managing all the above aspects, hiding them to the end user.

In the following how the sequential version (SGA) was implemented is first explained and then how MapReduce elements were mapped to the PGAs parallel models. The main challenge during the design phase was to limit the communication and synchronisation overhead of parallel tasks. There was the need of choose where to put the synchronisation barriers, namely the points in which the algorithm needs to wait for the completion of all parallel tasks before continuing with the computation. Generally speaking, the 'Amdhal's law' states that the speedup of a parallel program is limited by the sequential portion of the program, which means it is important to reduce as well as possible the overhead to gain execution time by parallelising. The positions of the synchronisation barriers are deeply bonded to the implemented models and thus it was needed to take them into consideration case-by-case to realise each of the three following parallel model adaptations. Moreover, further details of implementation considered as essential to understand the rest of the work are provided.

3.4.1 Sequential Genetic Algorithm

There are several possible versions of GA execution flows. The parallel adaptations are built on the base of the following SGA implementation, which is composed of a sequence of genetic operators repeated generation by generation, as described in Algorithm 3.

The execution flow starts with an initial population initialised with the `INITIALIZATION` function (1), which can be either a random function or a specifically defined one based on other criteria. Then, at the first generation, the

Algorithm 3 The Sequential Genetic Algorithm (SGA).

```

1: population ← INITIALIZATION(populationSize)
2: for i ← 1, n do
3:   if i = 1 then
4:     for individual ∈ population do
5:       FITNESSEVALUATION(individual)
6:   elitists ← ELITISM(population)
7:   population ← population − elitists
8:   selectedCouples ← PARENTSSELECTION(population)
9:   for (parent1, parent2) ∈ selectedCouples do
10:    (child1, child2) ← CROSSOVER(parent1, parent2)
11:    offspring ← offspring ∪ {child1} ∪ {child2}
12:   for individual ∈ offspring do
13:     MUTATION(individual)
14:   for individual ∈ offspring do
15:     FITNESSEVALUATION(individual)
16:   population ← SURVIVALSELECTION(population, offspring)
17:   population ← population ∪ elitists

```

genetic operator applied is the FITNESSEVALUATION (3–5), which evaluates and assign a fitness value to each individual, letting them be comparable. The ELITISM operator (5–6) allows to add some individuals directly to the next generation (17). The PARENTSSELECTION operator (8) selects the couples of parents for the CROSSOVER phase based on their the fitness values. The mixing of parent couples produces the offspring population (9–11), which is submitted to the MUTATION phase (12–13) in which the genes may be altered. The SURVIVALSELECTION applies a selection between parents and offspring individuals (16) to select the individuals that will take part of the next generations.

It is worth nothing that in each generation a second FITNESSEVALUATION is performed for the offspring individuals (14–15) in order to allow the SURVIVALSELECTION operation. From the second generation on, the individuals in the population will be already evaluated during the previous generations, thus requiring only the FITNESSEVALUATION of the offspring (14–15).

The PGAs described in the following differ from SGA in the way they parallelise the above operators and by adding another new genetic operator in the case of the island model (i.e., the ‘migration’).

3.4.2 Global Model

The implemented PGA for the global model on MapReduce (i.e., PGA_{global}) has the same behaviour of the sequential version, but it resorts to parallelisation for the fitness evaluation. Figure 3.2 shows the flow of the model. The master node, also referred as *Driver*, initialises a random population and writes it into the HDFS. During each generation, it spreads the individuals to the slave nodes in the cluster when:

1. the initial population is evaluated for the first time;
2. the generated offspring needs to be evaluated in order to apply the `SURVIVALSELECTION` to both parents and children.

Following the definition of the Algorithm 3, during the first generation two jobs are required for the parents and offspring populations evaluation. From the second generation on, a job is required for the offspring population only (see Section 3.4.1 for more details). Thus, the total number of jobs required is equal to the number of generations plus one. The *Driver* also executes sequentially the other genetic operators on the entire population that has been evaluated.

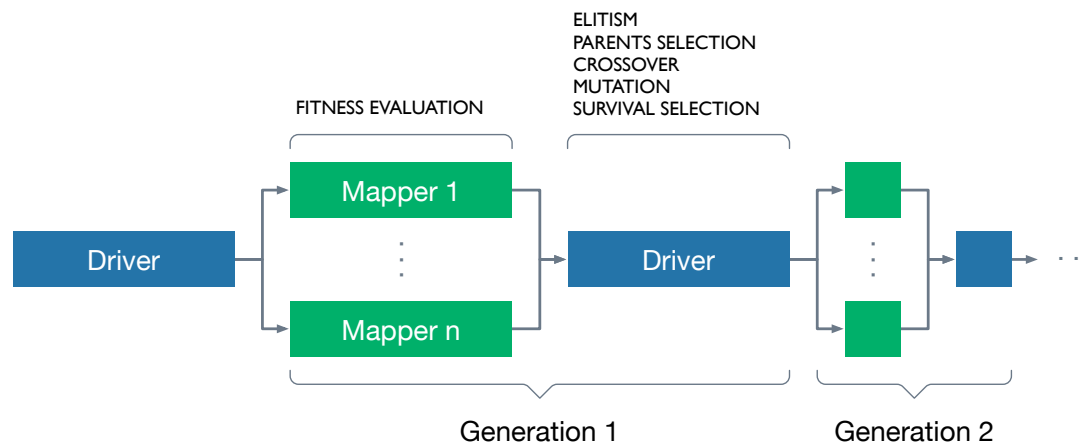


Figure 3.2: The flow of Hadoop MapReduce implementation for PGA_{global} .

More in details, the slave nodes in the cluster perform only the fitness evaluation operator. For all the three models, the mappers receive the records in the form: (individual, destination). The 'destination' field is used only by the other models so that it will be mentioned later. The reduce phase

was deliberately disabled, because there is no need of moving individuals between nodes. After the map phase, the master reads back the individual and continues with the other remaining genetic operators, considering the whole current population.

3.4.3 Grid Model

The implemented PGA for the grid model on MapReduce (i.e., PGA_{grid}) computes the genetic operators only to portions of the population called ‘neighbourhoods’. In the grid model, these portions are chosen randomly during the initialisation (Figure 3.3) and the number of jobs is the same as the number of generations. It is worth noting that the neighbourhoods never share any information with each other. Therefore, the offspring produced during the previous generation will stay in the same neighbourhoods also in the next generation.

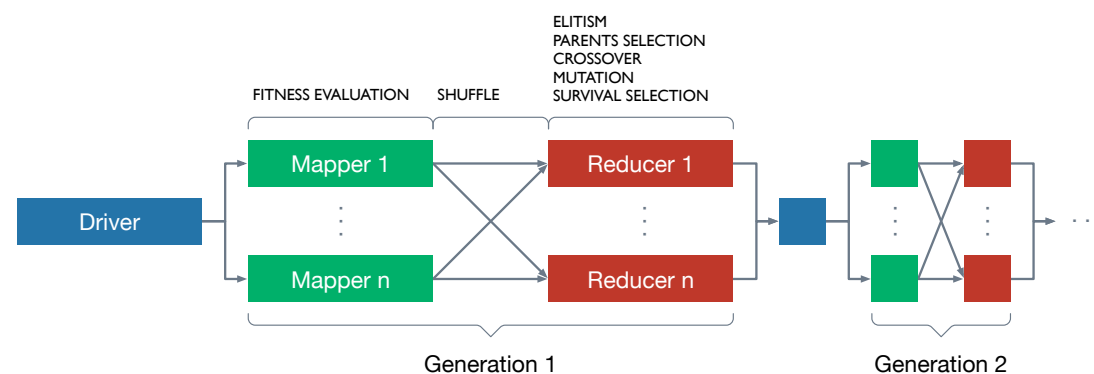


Figure 3.3: The flow of Hadoop MapReduce implementation for PGA_{grid} .

The Driver has the task of randomly generating a sequence of neighbourhoods destinations for the individuals in the current population. These destinations are stored into the record as the value fields so the destinations are known a priori. The parallelisation was exploited in two phases:

1. the mappers initialise a random population during the first generation and compute the fitness evaluation;
2. the partitioner sends the individuals to the correspondent neighbourhood

(i.e., the reducer). The reducers compute the other genetic operators and write the individuals in the HDFS.

In this case, it was decided to fix the number of neighbourhoods to the number of mappers, and so the number of reducers, to study the behaviour of the model regarding the parallelisation through our empirical study.

3.4.4 Island Model

The implemented PGA for the island model on MapReduce (i.e., PGA_{island}) acts similarly to the one for grid model because it operates on portions of the global population called ‘islands’. Each island executes whole periods of generations on its assigned portions, independently from the other islands until a migration occurs (Figure 3.4). It means there is an established migration period, which can be defined as the number of consecutive generations before migration. Since it is possible to run groups of subsequent generations (i.e., ‘periods’) independently, a MapReduce job for each period was exploited.

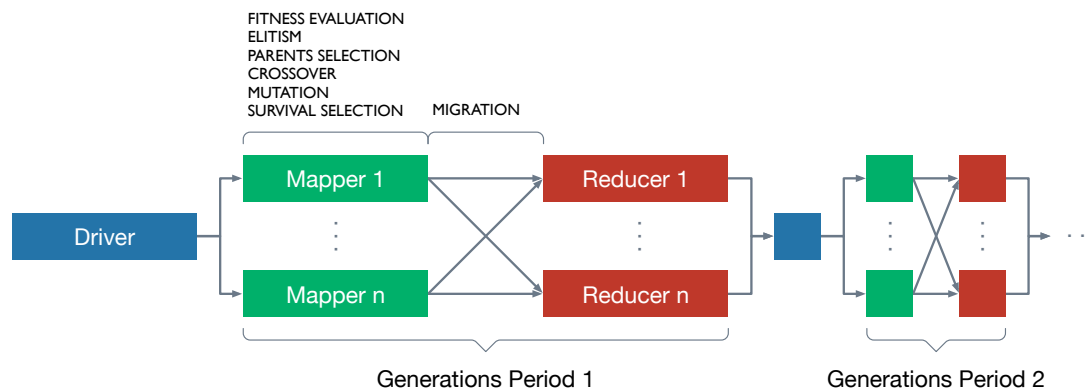


Figure 3.4: The flow of Hadoop MapReduce implementation for PGA_{island} .

In Hadoop, the numbers of mappers and reducers are not strictly correlated, but they were coupled to represent them as islands. The mappers were used to execute the generation periods and at the end of the map phase a function applies the migration criterion with which every individual will have a specific destination island. That is the time in which the second part of the output records is employed. Then the partitioner can establish where to send indi-

viduals and the reducer is used only to write into the HDFS the individuals received for its correspondent island.

Due to the generations groups, the synchronisation barrier is put after every migration and a job is needed for each period.

3.4.5 Data Serialisation

Hadoop is able to move data between nodes through sequences of write and read operations onto HDFS. The raw default serialisation of objects, in our case the individuals, is inefficient if compared to *Avro*², a modern data serialisation system from the same creator of Hadoop *Doug Cutting*. In addition to having a flexible data representation, it is optimised to minimise the disk space and communication through compression. For this reason, the performance of the implementations was tuned by using Avro.

In order to analyse the behaviour of the implemented models during our experiments, a reporter component giving the details of executions was added. Indeed, the interest was about both genetic trend of population and execution time described in detail in a fine-grained manner. The reporter component stores data into the HDFS with a non-invasive and asynchronous working so that the execution time of experiments is never influenced by extra operations.

3.4.6 elephant56 Architecture

In this section the architecture of elephant56 is described. Conceptually, two levels of abstraction were devised:

1. the 'core' level, which manages the communication with the underlying Hadoop MapReduce platform;
2. the 'user' level, which allows the developer to interface with the framework.

²<https://avro.apache.org>

Whereas the core level is immutable for the user, it is through the user level that the developers can extend its functionalities and develop their own GAs.

Core Level

The core level is responsible of dealing with the Hadoop MapReduce platform. This is made by extending the MapReduce classes provided with the Hadoop library. Figure 3.5 shows the structure of the core package, which contains five subpackages, each responsible of a different functionality.

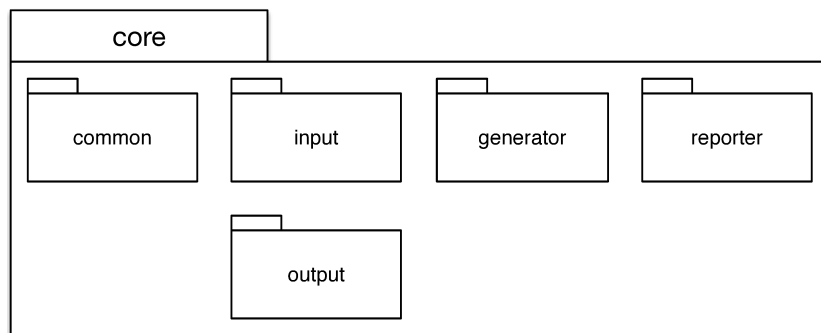


Figure 3.5: The core package class diagram.

The class in charge of the whole execution flow is the Driver class (Figure 3.6). It invokes the components included into the other core packages and it is specialised into different classes, which implement the models shown above. The Driver class is the linking point between core and user layers and the primary interface with the developer.

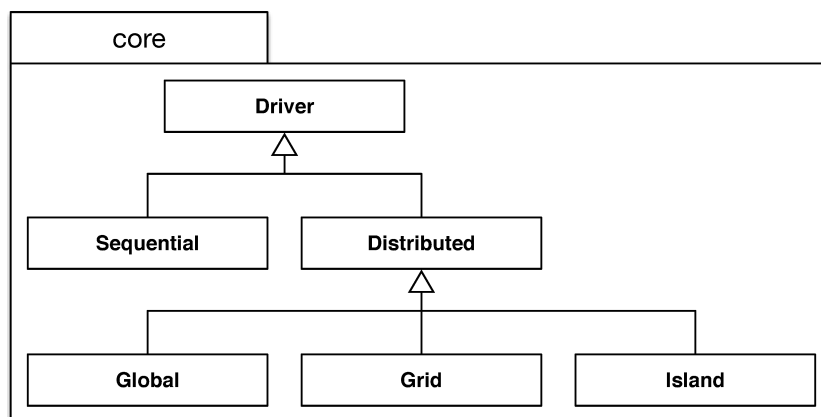


Figure 3.6: The Driver class.

core.common The `core.common` package contains two classes: the `IndividualWrapper` class, a wrapper class for both individual and fitness value objects exploited by `elephant56` to serialisation purposes; the `Properties` class, which allows the distribution of the properties defined by the developer among the different nodes. The `Properties` class involves an XML serialisation process for the distribution and some methods that ease the storage of properties values.

core.input The `core.input` package contains the implementations for the Hadoop input operations. The `NodesInputFormat` assigns a group of individuals (i.e., a MapReduce split) to a specific node. As mentioned above, the serialisation is made by using Avro, thus each split corresponds to a single binary file stored into the HDFS. The information about the split are stored using the `PopulationInputSplit` class and read with the `PopulationRecordReader` class which deserialises the Avro objects into Java objects for the slave nodes.

core.output The `core.output` allows serialising the Java objects produced during the generations and storing them into the HDFS. This is done by using the classes `NodesOutputFormat` and `PopulationRecordWriter`.

core.generator The `core.generator` package (Figure 3.7) is composed of the `GenerationsPeriodExecutor` class that implements the SGA. The specialisation of this class allows the distribution of GAs according to the different parallel models.

The mapper class of Hadoop is exploited by overriding the following methods:

1. `setup()`, which is invoked by Hadoop only at the beginning of the map phase and reads the configuration for the current generation together with the provided user properties;
2. `map()`, which is invoked for each input MapReduce record and stores all the input individuals into a data structure;

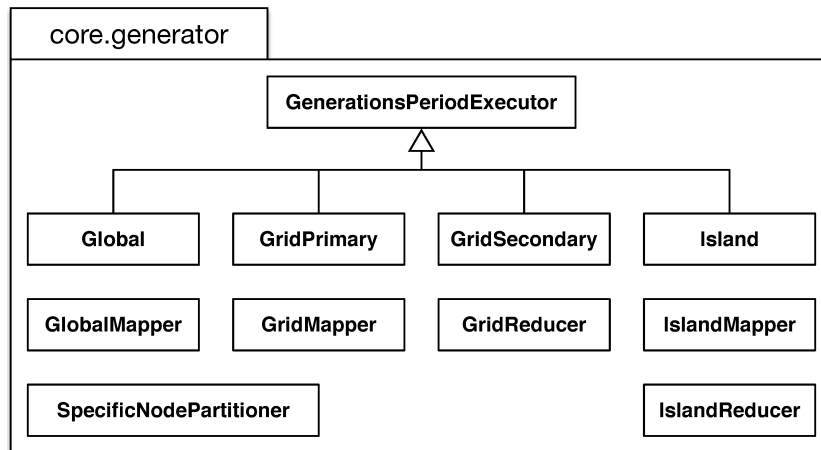


Figure 3.7: The generator package class diagram.

3. `cleanup()`, which is invoked after each input record has been read and starts the evolution process of the individuals.

The global model involves the map phase only and the `GenerationsPeriodExecutor` applies just the fitness evaluation operator on the slave nodes. The other genetic operators are applied by the master node when all the individuals have been evaluated.

The grid model involves three phases as follows: (i) the map, in which the fitness evaluation is performed; (ii) the partition (by using the `SpecificNodePartitioner` class), which assigns each individual to a random neighbourhood; (iii) the reduce, which executes the remaining genetic operators on the neighbours.

The island model also consists of three phases as follows: (i) the map, in which a certain number of generations (i.e., migration period) are executed; (ii) the partition, which assigns each individual to a specific node (i.e., island) established with the migration genetic operator; (iii) the reduce, which only stores the population of a given island into the HDFS.

core.reporter The package `core.reporter` (Figure 3.8) provides some utilities that can be used to produce a report containing some statistics about the GA execution. The output consists of some CSV (Comma Separated Values) files stored in the HDFS.

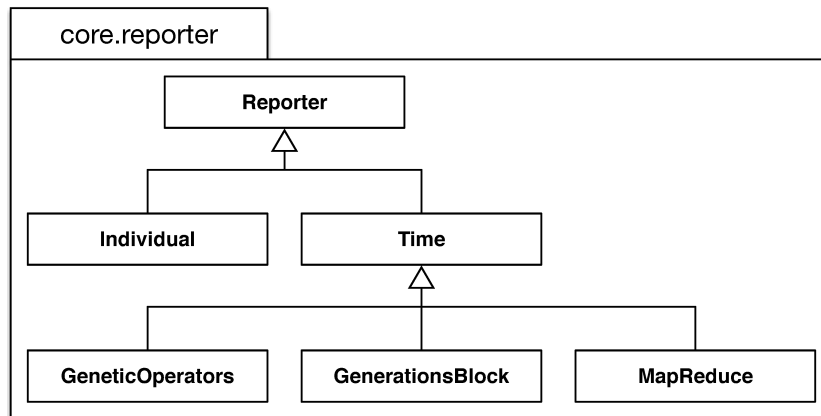


Figure 3.8: The reporter package class diagram.

The Reporter class is specialised into two types: Individual and Time. The former records the chromosome and fitness value of each individual produced during each generation. This allows to keep track and analyse the population evolution. The latter records the execution time (in milliseconds) of: (i) the genetic operators on each node; (ii) the generations; (iii) the MapReduce phases.

All the CSV writing operations are nonblocking so the report functionalities do not influence the computation time of the GA.

User Level

The user package (Figure 3.9) contains all the base classes the developer should extend in order to implement a specific GA, as explained in Section 3.5.

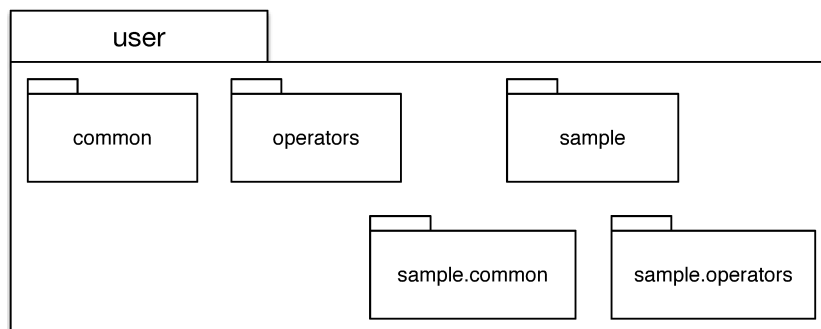


Figure 3.9: The user package class diagram.

In particular, the common package contains the Individual and Fitness-Value classes for the definition of the chromosome and fitness value, while the operators package contains the classes for the definition of specific genetic

operators. Some sample default implementations are included into the `sample` package, such as: sequences of primitive Java types, number fitness values, the random initialisation of sequences, a single point crossover, and the roulette wheel selection.

3.5 Usage

In this section how to use `elephant56` through a running example based on the simple problem of ‘OneMax’ (also known as ‘BitCounting’), which consists in maximising the number of 1 in a bit string, is explained.

The OneMax problem can be formally described as finding a string $\vec{x} = \{x_1, x_2, \dots, x_N\}$, with $x_i \in \{0, 1\}$, that maximises the following equation:

$$F(\vec{x}) = \sum_{i=1}^N x_i \quad (3.1)$$

To solve this problem with GAs, the individuals can be represented as bit strings and the above equation can be used as fitness function. It is also needed to define the genetic operators. `elephant56` allows the developer to define each of these elements by extending the classes of the framework. Because there is an underlying distributed platform (i.e., Hadoop MapReduce), many of the objects are encapsulated into some wrapper objects which ease the serialisation process.

In `elephant56`, an individual can be defined by extending the class `Individual`, which requires at least the implementation of the following serialisation method:

```
1 public abstract class Individual implements Cloneable {
2     public abstract Object clone() throws CloneNotSupportedException;
3     public abstract int hashCode();
4 }
```

For the OneMax example, a bit string chromosome can be defined adapting a list of boolean elements:


```

1 public class BitStringIndividual extends Individual {
2     private List<Boolean> bits;
3
4     public BitStringIndividual(int size) {
5         bits = new ArrayList<>(size);
6     }
7
8     public void set(int index, boolean value) {
9         bits.set(index, value);
10    }
11
12    public boolean get(int index) {
13        return bits.get(index);
14    }
15
16    public int size() {
17        return bits.size();
18    }
19    ...
20 }

```

The second important element to define is the fitness value, namely an object quantifying the result of the fitness function evaluation. This is possible by extending the `FitnessValue` class, which also requires to be comparable:

```

1 public abstract class FitnessValue implements Comparable<
2     ↳ FitnessValue>, Cloneable {
3     public abstract int compareTo(FitnessValue other);
4     public abstract Object clone() throws CloneNotSupportedException;
5     public abstract int hashCode();
6 }

```

For the example, the fitness value is an integer since it is needed to store for each solution how many bits are set to 1:

```

1 public class IntegerFitnessValue extends FitnessValue {
2     private int number;
3 }

```

```
4  public IntegerFitnessValue(int value) {
5      number = value;
6  }
7
8  public int get() {
9      return number;
10 }
11
12 @Override
13 public int compareTo(FitnessValue other) {
14     if (other == null)
15         return 1;
16     Integer otherInteger = ((IntegerFitnessValue) other).get();
17     Integer.compare(number, otherInteger);
18 }
19 ...
20 }
```

Both the `Individual` and `FitnessValue` objects are encapsulated into a wrapper class called `IndividualWrapper`.

Next, there is the need to implement the fitness function by extending the `FitnessEvaluation` class:

```
1  public class FitnessEvaluation<IndividualType extends Individual,
    ↪ FitnessValueType extends FitnessValue> extends GeneticOperator
    ↪ <IndividualType, FitnessValueType> {
2      ...
3      public FitnessValueType evaluate(IndividualWrapper<IndividualType,
    ↪ FitnessValueType> wrapper);
4  }
```

For `OneMax`, the fitness function (Equation 3.1) consists of simply counting the number of bit set to 1 in the bit string:

```
1  public class OneMaxFitnessEvaluation extends FitnessEvaluation<
    ↪ BitStringIndividual, IntegerFitnessValue> {
2      ...
3      @Override
```

```

4  public IntegerFitnessValue evaluate(IndividualWrapper<
    ↪ BitStringIndividual, IntegerFitnessValue> wrapper) {
5  BitStringIndividual individual = wrapper.getIndividual();
6  int count = 0;
7  for (int i = 0; i < individual.size(); i++)
8      if (individual.get(i))
9          count++;
10 return new IntegerFitnessValue(count);
11 }
12 }

```

At this point, it is the moment to define the genetic operators (i.e., crossover, mutation, selection). The procedure is similar to the one used for the fitness function, thus the code for the superclasses extended are omitted.

Before applying the genetic operators, the first step is to create an initial population. This can be done by randomly creates the individuals as follows:

```

1  public class RandomBitStringInitialization extends Initialization<
    ↪ BitStringIndividual, IntegerFitnessValue> {
2  ...
3  private int individualSize;
4  private Random random;
5
6  public RandomBitStringInitilization(..., Properties userProperties,
    ↪ ...) {
7  ...
8  individualSize = userProperties.getInt(INDIVIDUAL_SIZE_PROPERTY)
    ↪ ;
9  random = new Random();
10 }
11
12 @Override
13 public IndividualWrapper<BitStringIndividual, IntegerFitnessValue>
    ↪ generateNextIndividual(int id) {
14 BitStringIndividual individual = new BitStringIndividual(
    ↪ individualSize);
15 }

```

```
16     for (int i = 0; i < individualSize; i++)
17         individual.set(i, random.nextInt(2) == 1);
18
19     return new IndividualWrapper(individual);
20 }
21 }
```

It is worth noting that some properties can be distributed through the `Properties` object, which is filled on the master node and available from the constructor methods of the genetic operators when executed in parallel. In the example, it has been used to read the size of the bit strings.

After the fitness evaluation has happened, elitism and parents selection follow. The choice is to define them by using the `BestIndividualsElitism` and `RouletteWheelParentsSelection` classes, already provided by the framework. Of course, the developer may choose to define other elitism and/or parent selection strategies by extending the classes `Elitism` and `ParentsSelection`, respectively.

The crossover operator needs a specific implementation to manage bit string splits. A single point crossover can be defined as follows:

```
1 public class BitStringSinglePointCrossover extends Crossover<
   ↪ BitStringIndividual, IntegerFitnessValue> {
2     ...
3     private Random random;
4
5     public BitStringSinglePointCrossover(...) {
6         ...
7         random = new Random();
8     }
9
10    @Override
11    public List<IndividualWrapper<BitStringIndividual,
   ↪ IntegerFitnessValue>> cross(IndividualWrapper<
   ↪ BitStringIndividual, IntegerFitnessValue> wrapper1,
   ↪ IndividualWrapper<BitStringIndividual, IntegerFitnessValue>
   ↪ wrapper2, ...) {
```

```

12 BitStringIndividual parent1 = wrapper1.getIndividual();
13 BitStringIndividual parent2 = wrapper2.getIndividual();
14
15 cutPoint = random.nextInt(parent1.size());
16
17 BitStringIndividual child1 = new BitStringIndividual(parent1.
    ↪ size());
18 BitStringIndividual child2 = new BitStringIndividual(parent1.
    ↪ size());
19
20 for (int i = 0; i < cutPoint; i++) {
21     child1.set(i, parent1.get(i));
22     child2.set(i, parent2.get(i));
23 }
24
25 for (int i = cutPoint; i < parent1.size(); i++) {
26     child1.set(i, parent2.get(i));
27     child2.set(i, parent1.get(i));
28 }
29
30 List<IndividualWrapper<BitStringIndividual, IntegerFitnessValue
    ↪ >> children = new ArrayList<>(2);
31
32 children.add(new IndividualWrapper<>(child1));
33 children.add(new IndividualWrapper<>(child2));
34
35 return children;
36 }
37 }

```

The function cross selects a random cut point and builds two new children by mixing the chromosomes of the parents.

Thereafter, a random mutation function is implemented:

```

1 public class BitStringMutation extends Mutation<BitStringIndividual,
    ↪ IntegerFitnessValue> {
2     ...
3     private Random random;

```

```
4
5  public BitStringMutation(...) {
6      ...
7      mutationProbability = userProperties.getDouble(
8          ↪ MUTATION_PROBABILITY_PROPERTY);
9      random = new Random();
10
11     @Override
12     public IndividualWrapper<BitStringIndividual, IntegerFitnessValue>
13         ↪ mutate(IndividualWrapper<BitStringIndividual,
14         ↪ IntegerFitnessValue> wrapper) {
15         BitStringIndividual individual = wrapper.getIndividual();
16
17         for (int i = 0; i < individual.size(); i++)
18             if (random.nextDouble() <= mutationProbability)
19                 individual.set(i, !individual.get(i));
20
21         return wrapper;
22     }
23 }
```

This mutation operator, as defined above, mutates each gene according to a mutation probability that is distributed as a property value.

The survival selection is the last operator to be applied to produce the next offspring. In this example, the Roulette Wheel Selection is used, already implemented by the `RouletteWheelSurvivalSelection` class provided in the framework. Of course, the developer may choose to define his/her own survival selection strategy by extending the class `SurvivalSelection`.

Finally, it is possible to register all the defined classes in the *Driver* as follows:

```
1 public class App {
2     public static void main(String[] args) {
3         ...
4         driver.setIndividualClass(BitStringIndividual.class);
```

```
5     driver.setFitnessValueClass(IntegerFitnessValue.class);
6
7     driver.setInitializationClass(RandomBitStringInitialization.
8         ↪ class);
9     driver.setInitializationPopulationSize(POPULATION_SIZE);
10    userProperties.setInt(INDIVIDUAL_SIZE_PROPERTY, INDIVIDUAL_SIZE)
11        ↪ ;
12
13    driver.setElitismClass(BestIndividualsElitism.class);
14    driver.activateElitism(true);
15    userProperties.setInt(NUMBER_OF_ELITISTS_PROPERTY,
16        ↪ NUMBER_OF_ELITISTS);
17
18    driver.setParentsSelectionClass(RouletteWheelParentsSelection.
19        ↪ class);
20
21    driver.setCrossoverClass(BitStringSinglePointCrossover.class);
22
23    driver.setSurvivalSelectionClass(RouletteWheelSurvivalSelection.
24        ↪ class);
25    driver.activateSurvivalSelection(true);
26
27    driver.setUserProperties(userProperties);
28    ...
29    driver.run();
30    ...
31 }
```

The selection between sequential and parallel models is possible by specifying one of the Driver class specialisations.

Assuming that a Hadoop MapReduce cluster is already set up, the code can be packed in a single JAR file, including the elephant56 dependency, and executed with the standard Hadoop launch method.

3.6 Empirical Study Design

The main aim of this work was to understand if PGAs based on Hadoop MapReduce can be an effective solution to improve the scalability of GAs. Therefore, it was first needed to verify if, and possibly when, PGAs allow to get a better execution time compared to the sequential version (i.e., SGA). Moreover, there was the interest in understanding which PGA model is more effective among the global, grid and island models. Thus, the following research question was defined:

RQ *Is the use of PGAs based on Hadoop MapReduce worth using against SGA and which PGA model performs better?*

To address the **RQ**, a software engineering problem of configuring the SVMs for inter-release fault prediction was considered as a benchmark. The problem takes as input a dataset composed of software project components data, including the information about being faulty or not. The output is a configuration for SVMs optimised for the dataset at hand. The choice of this problem was motivated by the fact that it allows to assess the PGAs scalability considering different problem sizes by simply varying the input datasets size. Furthermore, the problem was already addressed by Di Martino et al. [13, 48] using a sequential approach. To verify the effectiveness of PGAs against SGA, the Di Martino et al.'s solution was employed. Even if the main aim was to exploit GAs for SVMs configuration as a benchmark problem in terms of execution time, it was also wanted to verify that the resulting predictive performance of output SVMs was not negatively affected when the produced by GAs executed in parallel (see Section 3.6.4). Moreover, the costs of the execution of the experimentation on a potential cloud infrastructure were estimated, based on the pricing of the major commercial cloud providers.

Details about the problem and GAs configuration are provided in Section 3.6.1. The datasets employed for the empirical study are described in Section 3.6.2. To understand the effectiveness of PGAs and compare the three parallel models, the applied experimental method described in Section 3.6.3

together with several evaluation criteria, namely the execution time, speedup, overhead, computational effort, predictive performance and cloud costs (see Section 3.6.4). The hardware employed to run the experiments is reported in Section 3.6.5. Finally, Section 3.6.6 analyses some threats to validity that may have affected the experimentation.

3.6.1 Using GAs to configure SVMs for Fault Prediction

The use of machine learning techniques to predict software faults has received increasing attention in the last years [1, 26, 39]. The research is motivated by the need to improve the efficiency of software testing, allowing project managers to better decide how to allocate resources to test the system, thus concentrating their efforts on fault-prone components. Nevertheless, it has been shown that the predictive performance of these techniques heavily depends on the selection of a suitable configuration [13, 50].

The use of GAs has been proposed to configure SVMs for software fault prediction [13, 28, 48, 16, 25]. The idea of exploiting GAs to configure SVMs for fault prediction is based on the observation that such problem can be formulated as an optimisation problem: between the possible configurations, finding the one which leads to the optimal SVMs performance. However, the combination of the two techniques (i.e., GAs and SVMs) may affect the scalability of the proposed approach when dealing with large software projects. This motivated the choice of using this problem as a benchmark for parallelising GAs.

In the experiments, the technique proposed by Di Martino et al. [13] were employed, which has been also adopted in other work [48, 28]. The technique works as follows: a solution to the problem is an SVMs configuration consisting of n parameters (with n determined by the kernel function). As for the kernel function, the widely used *Radial Basis Functions (RBF)* was employed, which has two parameters: C (the soft margin parameter) and γ (the radius of the RBF kernel). The GA chromosome is thus composed by two genes, for C and γ , whose values vary in the ranges 0.000 001 to 0.01 and 8 to 32 000, respectively.

Because the possible values for C and γ are both doubles, the solutions space of the possible SVMs configurations is enlarged remarkably.

To compute the fitness value of a chromosome representing an SVMs configuration, SVMs was executed with such a configuration thus obtaining the fault predictions. Such predictions are then evaluated using *F-measure* [57] as a performance criterion. The F-measure is defined as:

$$F\text{-measure} = 2 \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \quad (3.2)$$

The fitness function operates by applying a 5-fold cross-validation on a common training set and taking the average F-measure value as the final fitness value for each individual. The same setting used in previous work [13, 48, 28] both for the SGA and the PGAs was employed:

1. 200 individuals for the starting population;
2. 300 generations;
3. FITNESSEVALUATION consisting of a 5-fold cross-validation on a common training set and taking the average F-measure value;
4. ELITISM of 1 individual;
5. PARENTSSELECTION, using Roulette Wheel algorithm;
6. single point Crossover, with probability of 0.5;
7. MUTATION, with probability of 0.2;
8. SURVIVALSELECTION, using the Roulette Wheel algorithm.

In the case of the grid model, a number of neighbourhoods equal to the cluster size was used. As for the island model, there was also the need to identify the migration period and the number and selection policy of migrants. The migration period was set to 30 applying a ring topology exchange, whereas the number of migrants to 5% of the best individuals per island.

3.6.2 Input Datasets

The data from the *PROMISE* repository was exploited, which contains several datasets for fault prediction, choosing the software projects with more than two releases. Thus, three datasets were retained for a total of 10 releases: *Log4j* (vv. 1.0, 1.1, 1.2), *Lucene* (vv. 2.0, 2.2, 2.4), *POI* (vv. 1.5, 2.0, 2.5, 3.0). Each release contains a set of components (i.e., Java classes) described in terms of Chidamber and Kemerer (CK) metrics [8], Number of Public Methods (NPM) and Lines of Code (LOC). More details about those software projects and their fault data collection can be found in the work by Jureczko et al. [33].

These three datasets were chosen because they represent three different degrees of computational load for the fitness evaluation when the SVMs are built and validated through cross-validation. Indeed, a preliminary benchmark of the execution time of the fitness evaluation on the average of 30 runs and 300 generations, showed that *Lucene* and *POI* datasets are $2.7\times$ times and $9.5\times$ times slower than *Log4j* respectively. Therefore, this allowed to study the behaviour of PGAs on three problem instances of various size identified as 'small' for *Log4j*, 'medium' for *Lucene* and 'large' for *POI*.

3.6.3 Experimental Method

The **RQ** was addressed by comparing the performance of the PGAs based on each of the three parallel models explained in Section 3.4 and the SGA.

To observe the effectiveness of the considered techniques for inter-release fault prediction, the typical setting where data from the former releases are exploited to build the model to predict faults for a new release was used [43]. In particular, given a software project having n releases, the data collected in the first $n - 1$ releases of the project was used as the training set and the data collected for the last release as the test set. This allowed to simulate the situation that typically arises in real software development contexts, where a project manager can learn some phenomena and/or patterns from previous releases and exploit this knowledge for a more conscious management of the

development of a subsequent version. The fault data for the employed training and test sets are reported in Table 3.2 together with the percentage of faulty and non faulty components.

Table 3.2: Existence of faulty components in the training and test sets, where each component is a Java class and a fault corresponds to the presence of at least one reported bug.

Dataset	Training set classes				Test set classes			
	Faulty		Non faulty		Faulty		Non faulty	
<i>Log4j</i>	71	(29 %)	173	(71 %)	189	(92 %)	16	(8 %)
<i>Lucene</i>	235	(53 %)	207	(47 %)	203	(60 %)	137	(40 %)
<i>POI</i>	426	(46 %)	510	(54 %)	281	(64 %)	161	(36 %)

All the parallel models were executed on three different cluster configurations (i.e., C2, C4, C8) characterised by a different number of nodes (see details in Section 3.6.5). For each combination of model, dataset and cluster configuration, 30 runs were executed. Thus, a total of 900 runs consisting of $3 \cdot 3 \cdot 3 \cdot 30 = 810$ runs for PGAs and $3 \cdot 30 = 90$ runs for SGA were executed.

3.6.4 Evaluation Criteria

To compare the performance of the employed algorithms, the best practice in reporting the results with PGAs, identified by Luque and Alba [38], were followed. Performance was evaluated them both in terms of execution time, speedup, overhead and computational effort, as detailed in the following. The predictive performance was also evaluated in terms of *precision*, *recall*, *accuracy* and *F-measure* to verify that it was not negatively affected by the possible improvement of the execution time. Furthermore, the costs of the same executions on commercial cloud providers infrastructures were estimated. To cope with the stochastic nature of GAs and hardware executions, some statistical tests, described in the following, were performed.

Execution Time

The execution time was measured in milliseconds (ms) using the system clock. As a performance indicator of the whole execution, the execution time achieved by executing all the generations of SGA and PGAs were compared. The partial times were distinguished into computation and overhead times only in a second step, to quantify the time spent for parallel communication.

Speedup

The speedup is defined as the ratio of the sequential execution time to the parallel execution time. There are two types of speedup, i.e., the 'strong' and the 'weak' [38]. The strong speedup compares the parallel run time against the best so-far sequential algorithm. It was not possible to apply it since the intention was to compare the models on the Hadoop MapReduce platform, rather than comparing against different technologies. Moreover, it was not possible to find any implementation providing the same algorithms as the ones proposed.

Instead, the weak speedup compares the parallel algorithm developed by the researchers against their own sequential version. It was calculated by dividing the total amount of time that SGA required by the amount of time required by the PGA. The achieved speedup was compared with respect to the ideal speedup. It is worth noting that the ideal speedup is equal to the number of the employed parallel nodes and corresponds to the situation when the sequential execution time is perfectly split among multiple nodes. The ideal speedup is rarely achieved in practice due to the presence of overhead, but it is usually taken into consideration as an upper limit to compare the performance of parallel algorithms.

According to the best practice by Luque and Alba [38], when using the weak speedup it is important that the evaluated parallel algorithms should compute solutions having 'similar' accuracy as the sequential ones. For this reason, in addition to providing the weak speedup, the predictive performance

of the solutions was computed and reported at the end of the executions.

Overhead

To understand the reasons that prevent the PGAs to have a speedup near to the ideal one on the Hadoop MapReduce platform, the overhead for each execution was quantified. The time of each execution was considered and distinguished between overhead and computation times. In the following, the adopted method to determine these times is described.

The method allowed us to generalise the times of different multiple nodes but related to the same phase (e.g., ‘map computation’), with a proper start and finish time. The aim was to assign to each MapReduce job an initialisation, computation and finalisation time for both map and reduce phases.

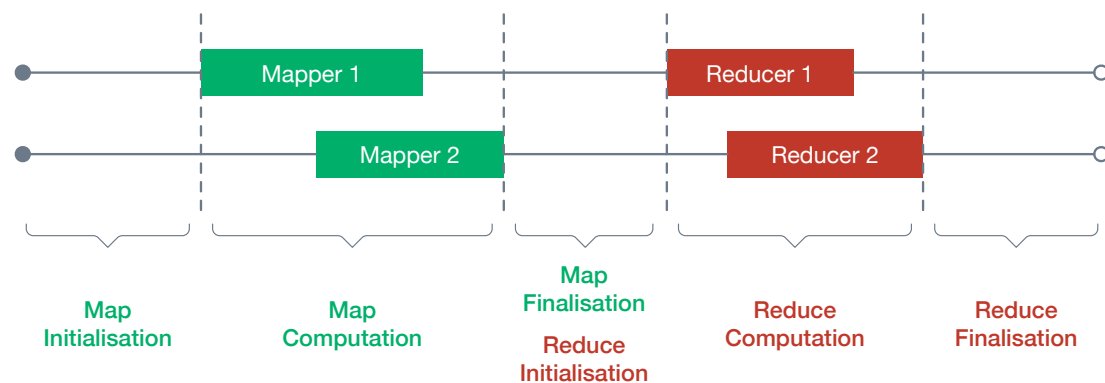


Figure 3.10: The time measurement method for multiple nodes.

Figure 3.10 shows one possible situation of a MapReduce job including a reduce phase. It is the case of the grid and island model, but not of the global model that has only a map phase. In those cases, the ‘map finalisation’ time is measured in the same way as for the ‘reducer finalisation’ time. As can be seen from the figure, the ‘map initialisation’ time is considered as the required time to let the first mapper begin its computation. The ‘map computation’ time is the time between the first mapper start and the last mapper finish time. The time between the last mapper and the first reducer is referred both as ‘map finalisation’ and ‘reduce initialisation’ time. Furthermore, the time after the last ending reducer is referred as the ‘reduce finalisation’ time.

Computational Effort

While the execution time can be exploited to evaluate the actual speed of the computation on a specific infrastructure, the computational effort can give a more machine-independent measure of the algorithms [38]. In the field of metaheuristics, the computational effort is in general measured by the number of evaluations corresponding to the number of points of the solution space visited. To this aim, the absolute number of fitness evaluations was computed and reported on average of the total number of runs. Because the number of generations was fixed, this measure depends only on the characteristics of the used model and cluster size.

Moreover, the eval/s measure for each of the models was also calculated. Even though this measure is strictly dependent on the specific infrastructure and dataset involved, it offers a better view of how a certain PGA can act when solving a problem with a certain computational load, i.e., whose fitness evaluation function requires a certain execution time, and the same degree of parallelisation.

Predictive Performance

To evaluate the predictive performance, four widely used measures (i.e., *precision*, *recall*, *accuracy* and *F-measure* [57]) were employed. These measures leverage on the concepts reported in the confusion matrix of Table 3.3 and are defined as follows.

The *precision* is the ratio between the number of components classified as TP and the number of those classified as TP or FP:

$$precision = \frac{TP}{TP + FP} \quad (3.3)$$

The *recall* is the ratio between the number of components classified as TP and the number of those classified as TP or FN:

$$recall = \frac{TP}{TP + FN} \quad (3.4)$$

The *accuracy* is the ratio between the number of components correctly predicted (i.e., classified as TP and TN) and the total number of components (i.e., the sum of TP, TN, FP, FN):

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.5)$$

The *F-measure* is a measure that provides an indication of a balance between correctness and completeness expressed as the harmonic mean of precision and recall as described above in Equation (3.2).

Table 3.3: The confusion matrix for fault prediction.

		Predicted	
		Faulty	Non faulty
Actual	Faulty	True Positive (TP)	False Negative (FN)
	Non faulty	False Positive (FP)	True Negative (TN)

Cloud Costs Estimation

Even if the experimentation was executed on a private infrastructure, as described in Section 3.6.5, it is also possible to run a Hadoop cluster on any commercial cloud infrastructure. A likely cost for the same execution that was actually performed was estimated, based on the pricing tables of the most used cloud providers. The estimation was based on the selection of cloud instances, i.e., the virtual machines, with a hardware configuration at least equal to the one employed in the experiments. Table 3.4 reports the configurations and pricing of the instances selected for the estimation.

Table 3.4: Commercial cloud configurations and pricing used for the costs estimation.

Provider	Instance	CPUs	RAM (GB)	Storage (GB)	Price (USD/h)
Amazon EC2	t2.medium	2	4	20	0.11
DigitalOcean	2GB	2	2	40	0.03
Microsoft Azure	A2	2	3.5	60	0.07
Google Cloud Platform	n1-standard-2	2	7.5	40	0.10

Statistical Tests

30 runs were executed in order to cope with the inherent randomness of dynamic execution time and GAs, and the average results reported.

Then, the non-parametric inferential statistical test, i.e., the *Wilcoxon Test* [9], was executed as recommended in the literature [3, 29]. The Wilcoxon signed rank test verifies, as the null hypothesis, if two considered populations have identical distributions. It is particularly useful when no assumptions about the normality of the distributions are possible, as for this case. For all the statistical tests, a probability of 5 % of committing a Type-I-Error was accepted.

Furthermore, the Vargha-Delaney \hat{A}_{12} effect size [52] was used. The \hat{A}_{12} test is an estimation of the probability that the algorithms have against each other in obtaining better results regarding the execution time and predictive performance measures. When two algorithms are compared and their results are equivalent, $\hat{A}_{12} = 0.5$. $\hat{A}_{12} > 0.5$ means that, on the average over the 30 runs, the first algorithm obtains better results than the second one with which is compared.

3.6.5 Hardware

In order to execute the experiment, a private *OpenStack* cloud platform was employed, making possible to virtualise the machines needed to compose the Hadoop clusters.

Table 3.5: Virtual machines configuration.

Type	Feature	Value
Hardware	Architecture	64 bit
	CPUs	2
	RAM	2 GB
	Storage	20 GB
Software	Operating System	CentOS 6.6
	Hadoop	Hortonworks 2.2
	Weka	3.7.11
	LibSVM	1.0.6

To run a fair experiment, the same configuration was used for each virtual machine (see Table 3.5). It is worth noting that any hardware component was simulated, thus the virtualisation of OpenStack was only used as a way of equally divide the hardware infrastucture. The partial resource (e.g., CPU cores, RAM) were completely dedicated to the running instances so that they could have fully used them without overlapping with others.

Table 3.6: Cluster configurations exploited by PGAs, where the master node drives the GA execution and the slave nodes compute the genetic operators in parallel.

Name	Master nodes	Slave nodes	Total nodes
C2	1	2	3
C4	1	4	5
C8	1	8	9

In this empirical study, 3 different types of Hadoop clusters were used, summarised in Table 3.6. SGA was executed on a single node, while the clusters C2, C4, and C8 for PGAs . The clusters are organised in 1 master node and a number of slaves equal to the parallelisation degree. There was the need to separate the master node from the slaves because the implementations have a sequential part and it was decided to dedicate slave nodes only to parallel purposes. Moreover, the underneath Hadoop platform requires the execution of many daemons and the resources of just the slaves would not have been enough. Hadoop was installed through the *Hortonworks* distribution, which eased the orchestration and monitoring of the clusters.

Multiple experiments were run simultaneously on a total of 150 OpenStack virtual machines. With 30 runs for every single experiment of 300 generations each, the empirical study took about 120 days for a total of 900 runs.

3.6.6 Threats to Validity

Threats to *construct validity* concern the relationship between the theory behind the experiments and the observations. In order to alleviate possible threats related to measurement, the GAs execution time was quantified using the

system clock, because it represents the speed of a technique to the end-user. In addition, the computational effort as machine-independent measure of the algorithms is also provided.

Threats to *internal validity* concern any confounding factors that could influence the results. A possible threat is related to the randomness due to the use of GAs and variable network/computational load on the nodes at the time of the experiment. Indeed, GAs are intrinsically random and such a threat was mitigated by executing all the experiments 30 times and presenting the average results [3, 29]. Furthermore, both the network and computational nodes may have been biased by the randomness of events and the multiple runs were intended to alleviate these issues as well.

Threats to *external validity* concern the generalisability of the findings outside the scope of this study. An external threat is due to the fact that the three PGAs models were benchmarked for a specific software engineering task, i.e., configuring SVMs using GA for fault prediction. Besides being an example of a real-world application of GAs, this prediction task was chosen because it exhibits scalability issues when dealing with large training dataset, therefore constituting a suitable benchmark for the three different parallel architectures. To this end, three datasets with different sizes and characteristics were investigated. It is worth noting that the configuration chose for the GAs is not exclusive and other possible parameters sets could have used, aiming at improving the predictive performance of models. However, since the main interest was in the comparison of the execution time performance, the most trivial configuration for all the parallelisation models was selected. Moreover, for the grid model it was chosen to use a number of neighbourhoods equal to the cluster size. Although it could have lowered the predictive performance but the execution time. On the one hand, if a minor number had been used, it would not have been possible to exploit the full computational capacity of the parallel nodes. On the other hand, using a major number of neighbourhoods, the population would have always split among the same number of mappers for the fitness evaluation. Then, the parallel reducers would have received more than one

neighbourhood each and processed them sequentially but with less individuals than the other case.

3.7 Results

In this section, the results of the study are presented. The comparison between SGA and PGAs with respect to the execution time is reported in Section 3.7.1. The analyses of the speedup and overhead are reported in Section 3.7.2 and Section 3.7.3, respectively. Section 3.7.4 reports the computational effort whereas the predictive performance is analysed in Section 3.7.5. Section 3.7.6 concludes with the estimation of cloud costs.

3.7.1 Execution Time

Figure 3.11 shows the boxplots of the achieved execution times over 30 runs, while Table 3.7 shows the mean, standard deviation and median values of the same times.

It is possible to observe that the execution for each of the considered clusters of $\text{PGA}_{\text{island}}$ is always faster than each SGA execution for all the considered datasets. Every parallelisation model performs better than SGA for the *POI* dataset, while for the other two datasets $\text{PGA}_{\text{global}}$ and PGA_{grid} are slower than SGA, regardless of the parallel nodes used.

The Wilcoxon test results reported in Table 3.8 confirm the above observations. The execution time of $\text{PGA}_{\text{island}}$ (using C2, C4 and C8 clusters) is significantly lower ($p\text{-value} < 0.05$) than the one of SGA on all the considered datasets, while the execution time of $\text{PGA}_{\text{global}}$ and PGA_{grid} is significantly lower than SGA only on the biggest dataset (i.e., *POI*) and higher on the other two.

The Vargha-Delaney test results (see Table 3.8) confirm ($\hat{A}_{12} = 0$) that $\text{PGA}_{\text{island}}$ achieves better results in terms of execution time than SGA for all the 30 runs, three cluster configurations and datasets. Furthermore, for the *POI* dataset, all the three PGAs performs better than SGA. This can be explained by

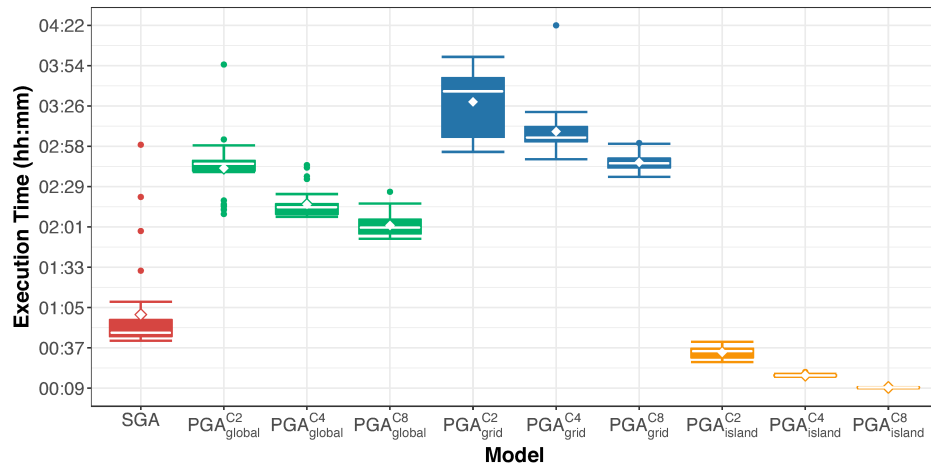
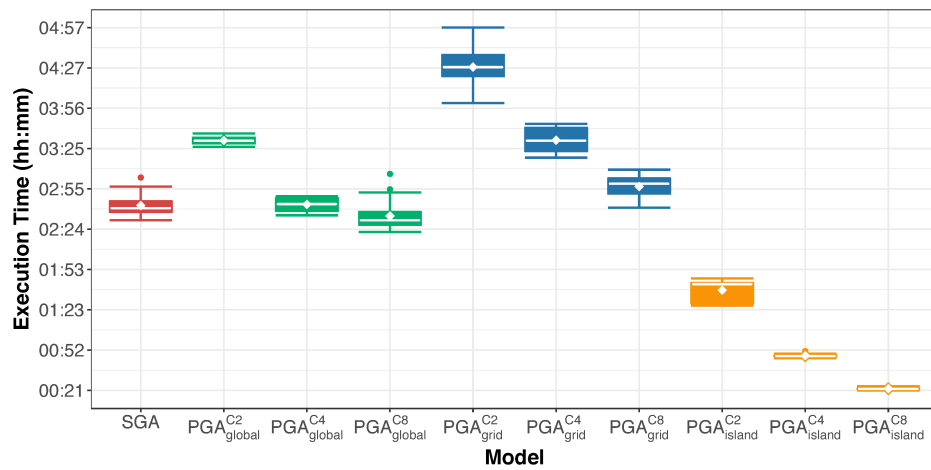
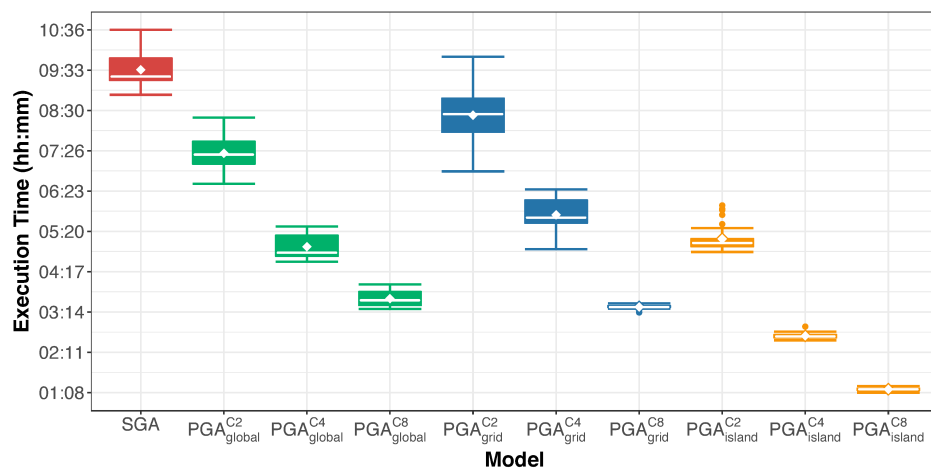
(a) *Log4j*(b) *Lucene*(c) *POI*

Figure 3.11: Execution times achieved by SGA and PGAs on the three clusters for the three datasets.

the fact that for small instances of the problem the overhead due to the data accesses and communication between the nodes is higher than the time needed to compute the fitness function.

As for the comparison between parallel models, the boxplots of Figure 3.11 and the complete Wilcoxon and Vargha-Delaney tests results, reported in Tables 3.9, 3.10 and 3.11, show that the PGA_{grid} model is the slowest model and significantly different from the other two models.

Table 3.7: Execution time (hh:mm) achieved by executing 30 times SGA and PGAs on the considered datasets.

Model	Execution time (hh:mm)								
	<i>Log4j</i>			<i>Lucene</i>			<i>POI</i>		
	Mean	SD	Median	Mean	SD	Median	Mean	SD	Median
SGA	01:00	00:31	00:47	02:42	00:08	02:40	09:33	00:24	09:23
PGA _{global} ^{C2}	02:42	00:20	02:45	03:32	00:02	03:31	07:22	00:27	07:20
PGA _{global} ^{C4}	02:17	00:09	02:15	02:43	00:05	02:43	04:56	00:18	04:47
PGA _{global} ^{C8}	02:02	00:07	02:01	02:34	00:10	02:31	03:35	00:12	03:32
PGA _{grid} ^{C2}	03:29	00:22	03:36	04:27	00:12	04:27	08:22	00:45	08:24
PGA _{grid} ^{C4}	03:08	00:15	03:04	03:32	00:08	03:31	05:46	00:23	05:42
PGA _{grid} ^{C8}	02:46	00:05	02:46	02:57	00:08	02:59	03:22	00:03	03:22
PGA _{island} ^{C2}	00:34	00:04	00:34	01:38	00:07	01:42	05:09	00:20	05:02
PGA _{island} ^{C4}	00:18	00:00	00:18	00:48	00:01	00:48	02:37	00:04	02:36
PGA _{island} ^{C8}	00:09	00:00	00:09	00:23	00:00	00:23	01:13	00:02	01:13

Table 3.8: Wilcoxon test (*p-values*) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the execution time between PGAs and SGA over 30 runs on the considered datasets.

Model	=	<i>Log4j</i>		<i>Lucene</i>		<i>POI</i>	
		<i>p-value</i>	\hat{A}_{12}	<i>p-value</i>	\hat{A}_{12}	<i>p-value</i>	\hat{A}_{12}
PGA _{global} ^{C2}	SGA	<0.001	0.961	<0.001	1.000	<0.001	0.000
PGA _{global} ^{C4}	SGA	<0.001	0.939	0.465	0.556	<0.001	0.000
PGA _{global} ^{C8}	SGA	<0.001	0.921	0.008	0.218	<0.001	0.000
PGA _{grid} ^{C2}	SGA	<0.001	0.992	<0.001	1.000	<0.001	0.083
PGA _{grid} ^{C4}	SGA	<0.001	0.998	<0.001	1.000	<0.001	0.000
PGA _{grid} ^{C8}	SGA	<0.001	0.969	<0.001	0.884	<0.001	0.000
PGA _{island} ^{C2}	SGA	<0.001	0.000	<0.001	0.000	<0.001	0.000
PGA _{island} ^{C4}	SGA	<0.001	0.000	<0.001	0.000	<0.001	0.000
PGA _{island} ^{C8}	SGA	<0.001	0.000	<0.001	0.000	<0.001	0.000

Table 3.9: Wilcoxon test (p -values) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the execution time achieved by SGA and PGAs over 30 runs on the *Log4j* dataset.

Model	SGA		PGA ^{C2} _{global}		PGA ^{C4} _{global}		PGA ^{C8} _{global}		PGA ^{C2} _{grid}		PGA ^{C4} _{grid}		PGA ^{C8} _{grid}		PGA ^{C2} _{island}		PGA ^{C4} _{island}		PGA ^{C8} _{island}	
	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}
SGA	–	0.500	<0.001	0.039	<0.001	0.061	<0.001	0.079	<0.001	0.008	<0.001	0.002	<0.001	0.031	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{global}	<0.001	0.961	–	0.500	<0.001	0.886	<0.001	0.984	<0.001	0.050	<0.001	0.054	0.124	0.441	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{global}	<0.001	0.939	<0.001	0.114	–	0.500	<0.001	0.927	<0.001	0.000	<0.001	0.000	<0.001	0.022	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{global}	<0.001	0.921	<0.001	0.016	<0.001	0.073	–	0.500	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{grid}	<0.001	0.992	<0.001	0.950	<0.001	1.000	<0.001	1.000	–	0.500	<0.001	0.706	<0.001	0.982	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{grid}	<0.001	0.998	<0.001	0.946	<0.001	1.000	<0.001	1.000	<0.001	0.294	–	0.500	<0.001	0.977	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{grid}	<0.001	0.969	0.124	0.559	<0.001	0.978	<0.001	1.000	<0.001	0.018	<0.001	0.023	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000
PGA ^{C8} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500

Table 3.10: Wilcoxon test (p -values) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the execution time achieved by SGA and PGAs over 30 runs on the *Lucene* dataset.

Model	SGA		PGA ^{C2} _{global}		PGA ^{C4} _{global}		PGA ^{C8} _{global}		PGA ^{C2} _{grid}		PGA ^{C4} _{grid}		PGA ^{C8} _{grid}		PGA ^{C2} _{island}		PGA ^{C4} _{island}		PGA ^{C8} _{island}	
	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}
SGA	–	0.500	<0.001	0.000	0.465	0.444	0.008	0.782	<0.001	0.000	<0.001	0.000	<0.001	0.116	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{global}	<0.001	1.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	0.000	0.730	0.517	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{global}	0.465	0.556	<0.001	0.000	–	0.500	0.001	0.801	<0.001	0.000	<0.001	0.000	<0.001	0.101	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{global}	0.008	0.218	<0.001	0.000	0.001	0.199	–	0.500	<0.001	0.000	<0.001	0.000	<0.001	0.063	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{grid}	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{grid}	<0.001	1.000	0.730	0.483	<0.001	1.000	<0.001	1.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{grid}	<0.001	0.884	<0.001	0.000	<0.001	0.899	<0.001	0.937	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000
PGA ^{C8} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500

Table 3.11: Wilcoxon test (p -values) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the execution time achieved by SGA and PGAs over 30 runs on the *POI* dataset.

Model	SGA		PGA ^{C2} _{global}		PGA ^{C4} _{global}		PGA ^{C8} _{global}		PGA ^{C2} _{grid}		PGA ^{C4} _{grid}		PGA ^{C8} _{grid}		PGA ^{C2} _{island}		PGA ^{C4} _{island}		PGA ^{C8} _{island}	
	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}
SGA	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	0.917	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{global}	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	0.130	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{global}	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	0.000	<0.001	0.048	<0.001	1.000	0.043	0.329	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{global}	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	0.000	<0.001	0.000	<0.001	0.826	<0.001	0.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{grid}	<0.001	0.083	<0.001	0.870	<0.001	1.000	<0.001	1.000	–	0.500	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{grid}	<0.001	0.000	<0.001	0.000	<0.001	0.952	<0.001	1.000	<0.001	0.000	–	0.500	<0.001	1.000	<0.001	0.879	<0.001	1.000	<0.001	1.000
PGA ^{C8} _{grid}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.174	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	0.000	<0.001	1.000	<0.001	1.000
PGA ^{C2} _{island}	<0.001	0.000	<0.001	0.000	0.043	0.671	<0.001	1.000	<0.001	0.000	<0.001	0.121	<0.001	1.000	–	0.500	<0.001	1.000	<0.001	1.000
PGA ^{C4} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500	<0.001	1.000
PGA ^{C8} _{island}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	–	0.500

3.7.2 Speedup

Once established the execution times of the SGA and PGAs, the speedup values were calculated.

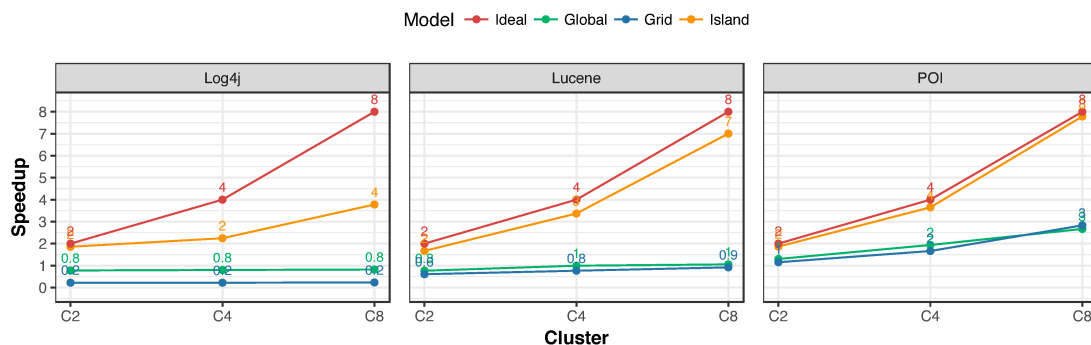


Figure 3.12: Speedup trend per dataset.

Figure 3.12 and Table 3.12 show that the use of parallelisation is worth mainly for the island model, which allowed us to speed up the execution time with respect to SGA of an average over the three datasets of $7.0 \times$ times by exploiting $\text{PGA}_{\text{island}}^{\text{C8}}$, $3.4 \times$ times by exploiting $\text{PGA}_{\text{island}}^{\text{C4}}$ and $1.8 \times$ times by exploiting $\text{PGA}_{\text{island}}^{\text{C2}}$ times. It is clear from Figure 3.12 that $\text{PGA}_{\text{island}}$ tends to the ideal speedup value. On the other hand, $\text{PGA}_{\text{global}}$ and PGA_{grid} improved their speedup only slightly, because of the overhead discussed in Section 3.7.3.

Table 3.12: The speedup values for each PGA model on the three datasets and three clusters.

Model	Speedup								
	<i>Log4j</i>			<i>Lucene</i>			<i>POI</i>		
	Mean	SD	Median	Mean	SD	Median	Mean	SD	Median
$\text{PGA}_{\text{global}}^{\text{C2}}$	0.364	0.142	0.328	0.767	0.040	0.762	1.301	0.114	1.285
$\text{PGA}_{\text{global}}^{\text{C4}}$	0.430	0.177	0.360	0.996	0.053	0.986	1.938	0.121	1.947
$\text{PGA}_{\text{global}}^{\text{C8}}$	0.486	0.223	0.399	1.056	0.097	1.047	2.667	0.125	2.656
$\text{PGA}_{\text{grid}}^{\text{C2}}$	0.296	0.167	0.246	0.608	0.045	0.604	1.153	0.135	1.143
$\text{PGA}_{\text{grid}}^{\text{C4}}$	0.322	0.169	0.265	0.767	0.041	0.760	1.663	0.149	1.656
$\text{PGA}_{\text{grid}}^{\text{C8}}$	0.365	0.200	0.291	0.920	0.050	0.933	2.831	0.114	2.805
$\text{PGA}_{\text{island}}^{\text{C2}}$	1.801	0.959	1.500	1.670	0.197	1.568	1.864	0.150	1.866
$\text{PGA}_{\text{island}}^{\text{C4}}$	3.337	1.775	2.652	3.370	0.174	3.357	3.651	0.188	3.603
$\text{PGA}_{\text{island}}^{\text{C8}}$	6.263	3.317	4.985	7.007	0.344	6.977	7.786	0.376	7.740

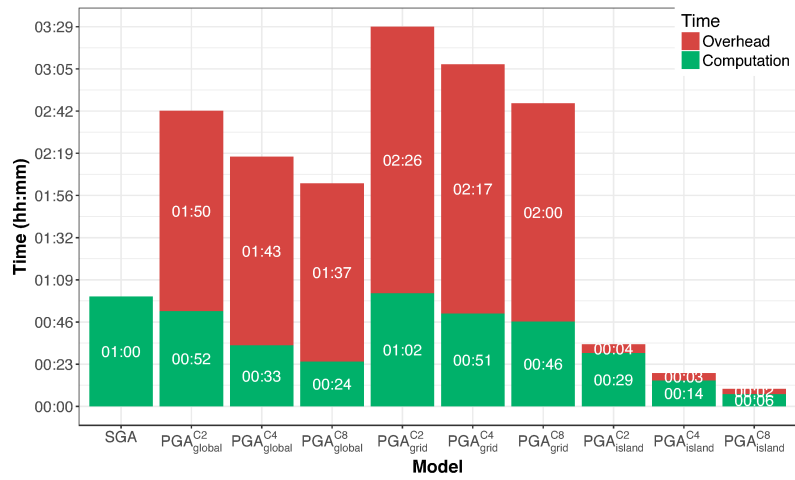
3.7.3 Overhead

To further investigate the behaviour of the parallel implementations on the Hadoop MapReduce platform, the execution time of PGAs was analysed with a more fine-grained scale.

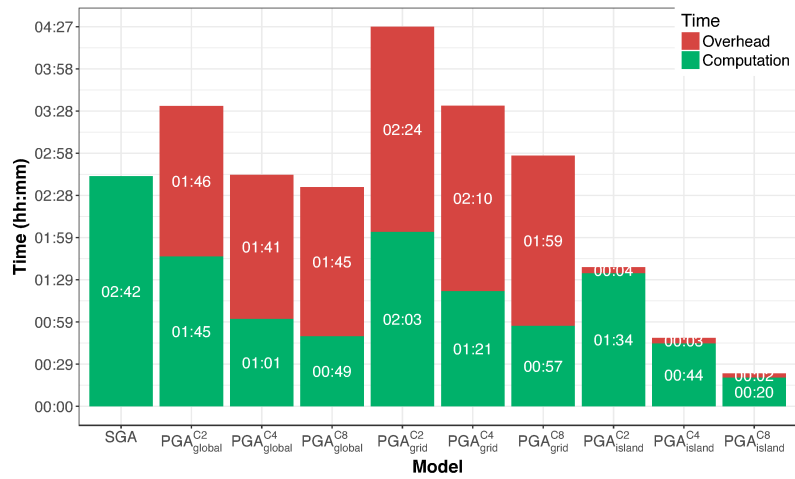
Figure 3.13 shows the computation and overhead times for each PGA and dataset combination, where the overhead is intended as the additional time other than the computational, generally due to communication and distribution platform tasks. The stacked bars represent the mean over 30 runs. It is worth noting that overhead time in Hadoop MapReduce corresponds to the sum of the overhead times of multiple jobs. As it possible to notice from the figure, for the *Log4j* and *Lucene* datasets on global and grid models the overhead time surpasses the computation time. As mentioned above, in the presence of heavy computational work (i.e., *POI* dataset), global and grid models are worth using. The island model is always light in terms of overhead time due to the lower number of jobs involved: where the global and grid models have one job for each generation, the island model executes groups of generations (i.e., periods) in single jobs.

Also, the overhead time per job was analysed. As an example, Figures 3.14, 3.15 and 3.16 report the mean execution time for each job over 30 runs and all the cluster configurations for the three dataset.

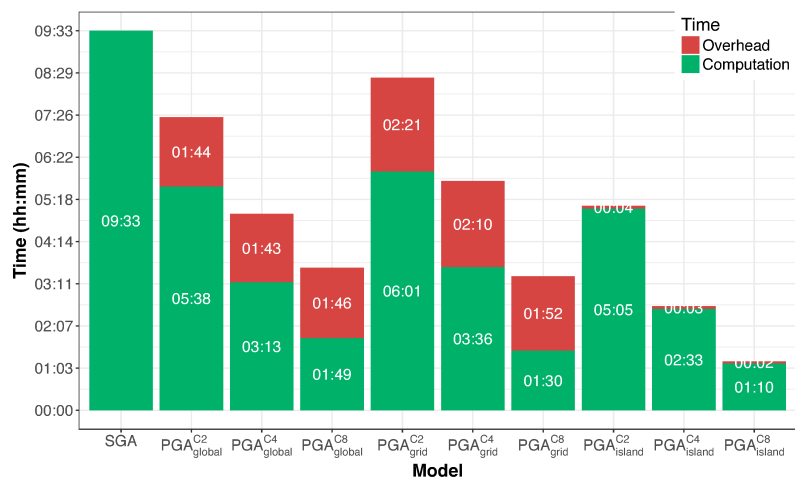
It is possible to observe that the partial overhead times are almost constant (i.e., the standard deviation is very small) over the different jobs. What makes the island model win against other models is basically the fact that it has fewer jobs than others. Moreover, it is worth noting that not all the parallel model implementations have both map and reduce phases and not all of them behave in the same way (see Section 3.4 for details). The map initialisation phase is common to the three parallel models and it takes a similar amount of time: this is due to the fact that each time a new job is requested, Hadoop MapReduce spends some time to orchestrate the cluster. However, the resulting execution time in the case of the reducer communication phases, which is present only in the grid and island models, is much less than the ones for the map phase.



(a) *Log4j*



(b) *Lucene*



(c) *POI*

Figure 3.13: The computation and overhead times for each PGA model.

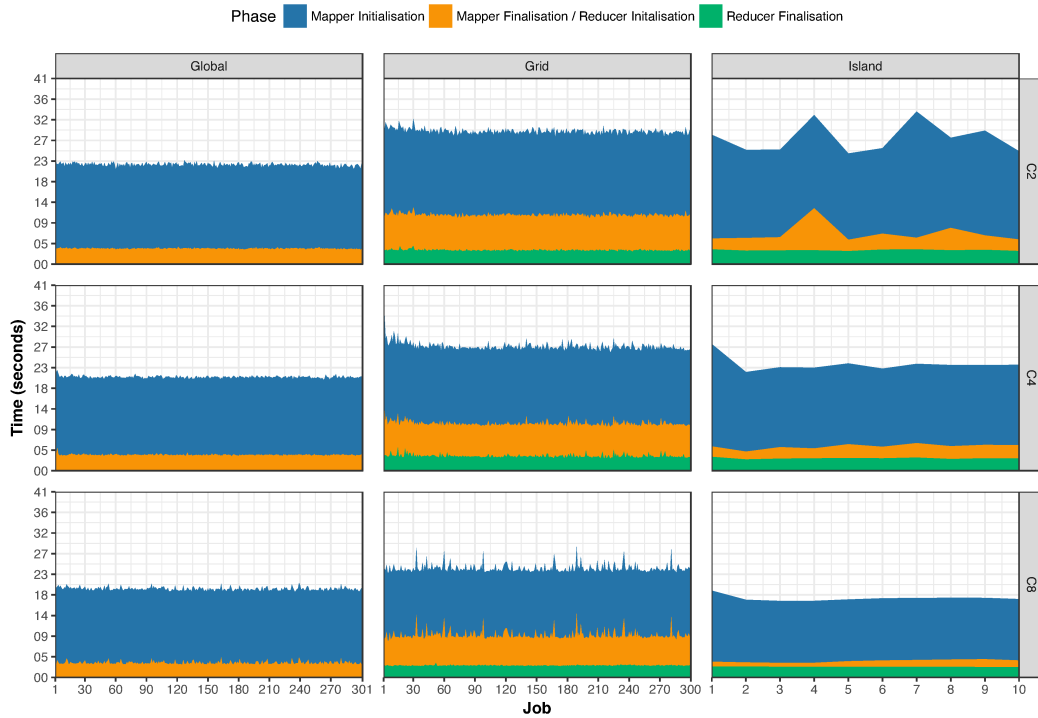


Figure 3.14: The MapReduce mean overhead times per job for each PGA model and cluster configuration on the *Log4j* dataset over 30 runs.

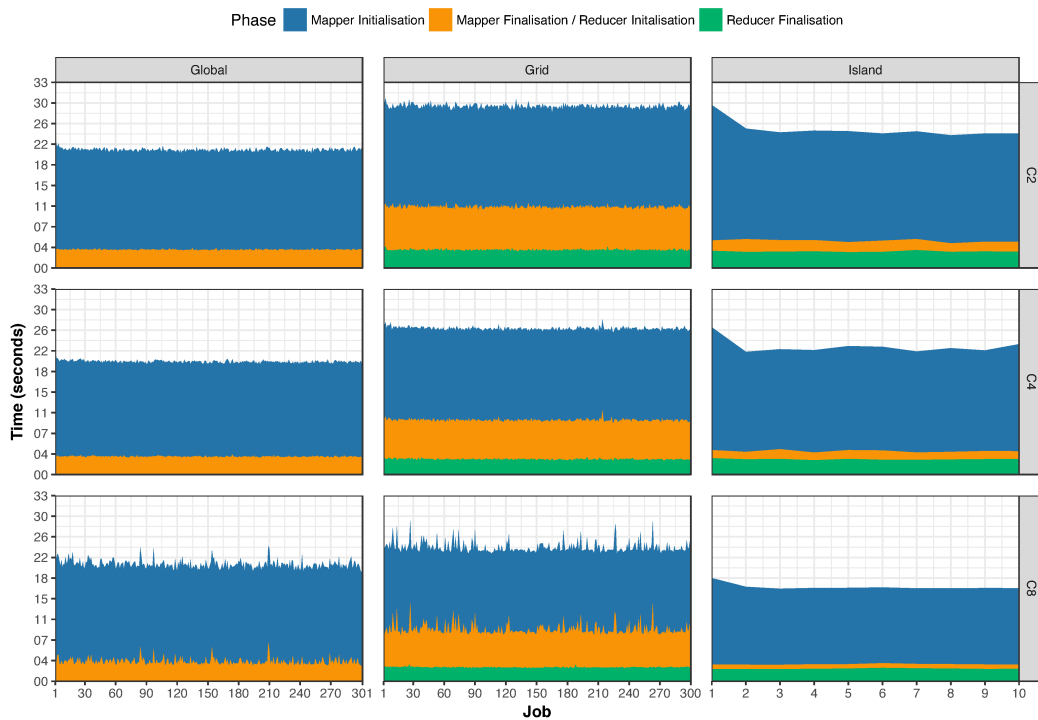


Figure 3.15: The MapReduce mean overhead times per job for each PGA model and cluster configuration on the *Lucene* dataset over 30 runs.

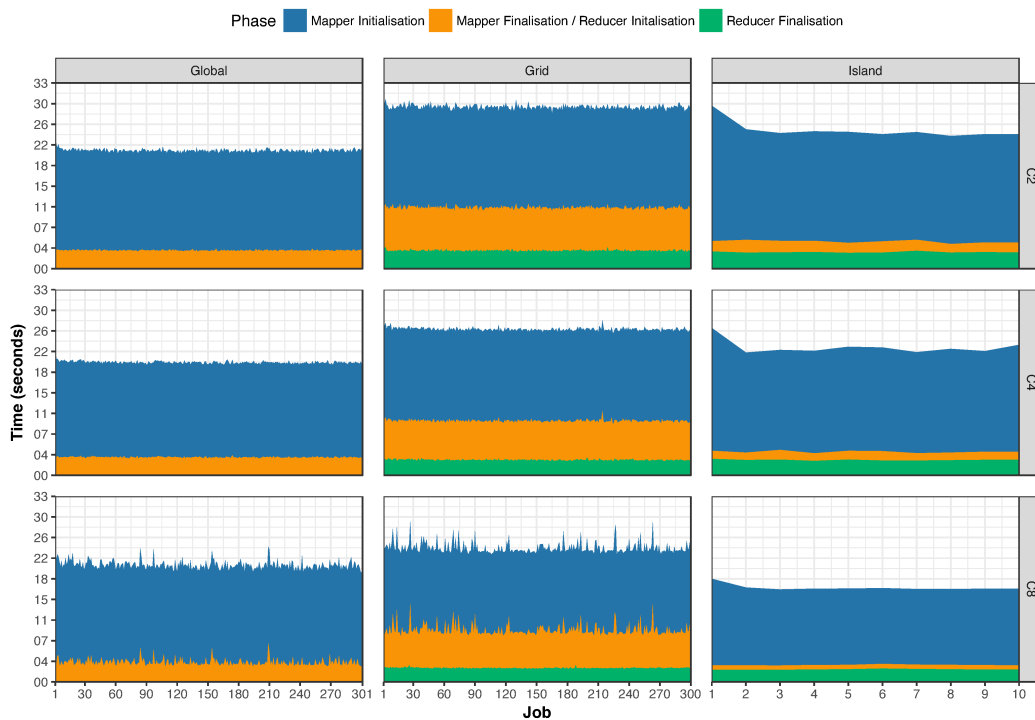


Figure 3.16: The MapReduce mean overhead times per job for each PGA model and cluster configuration on the *POI* dataset over 30 runs.

Because of the Hadoop architecture, the nodes are already prepared to host the reduce phase when the mappers are finishing. Nevertheless, there are some differences in times between the grid and island models. These times are comprehensive of the partitioner component work, which is the Hadoop MapReduce phase responsible for moving individuals to a new neighbourhood and a new island in the grid and island model, respectively. In the grid model, all the individuals can be assigned to a different destination whereas in the island model this is true only for 5% on the number of individuals per island. Moreover, for the grid model there is the need of reading the individuals of the current neighbourhood and deserialise them because the other genetic operators must be executed. The last communication phase of the three PGAs is the same in terms of the execution time and behaviour. Even if PGA_{global} lacks the reduce phase, the last phase of all the three models is responsible of the same task, i.e., writing back the processed individuals to the HDFS.

The correlation between overhead times was tested using the ANOVA test but none was found. The reason may be imputed to the non-deterministic

execution behaviours collected by several layers of abstraction: Hadoop MapReduce is executed on Java Virtual Machines (JVMs) as a shared process on cloud virtual machine instances in a shared OpenStack environment. Although there is not strong statistical evidence, it was observed that, on average, the overhead times seem to be independent of the dataset and cluster size.

3.7.4 Computational Effort

Table 3.14 reports the number of fitness evaluations achieved on the average of 30 runs of each algorithm and dataset. The number of evaluations between the same algorithm for different datasets is the same since the stochastic nature of GA is controlled by using the same random seeds for each dataset.

The number of evaluations for PGA_{global} does not differ from SGA since they perform exact in the same way, regardless of the cluster size. PGA_{grid} is subject to a deterioration of the number of evaluations as the cluster size increases. PGA_{island} is balanced. In general, it can be affirmed that, except for PGA_{grid} , all the models show a number of evaluations similar to the one of SGA.

The Wilcoxon and Vargha-Delaney tests results, reported in Table 3.13, confirmed the above results.

Moreover, Table 3.14 shows the number evaluations per second. It may be used as a predictor for the final execution time of PGAs for problems with the same computational load of the one required for the three datasets, on the same hardware.

Table 3.13: Wilcoxon test (p -values) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the fitness evaluations number achieved by SGA and PGAs over 30 runs on the three datasets and three clusters.

Model	SGA		PGA ^{C2} _{global}		PGA ^{C4} _{global}		PGA ^{C8} _{global}		PGA ^{C2} _{grid}		PGA ^{C4} _{grid}		PGA ^{C8} _{grid}		PGA ^{C2} _{island}		PGA ^{C4} _{island}		PGA ^{C8} _{island}	
	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}	p -value	\hat{A}_{12}
SGA	–	0.500	–	0.500	–	0.500	–	0.500	0.641	0.440	0.010	0.767	<0.001	1.000	0.688	0.531	0.007	0.349	0.113	0.622
PGA ^{C2} _{global}	–	0.500	–	0.500	–	0.500	–	0.500	0.641	0.440	0.010	0.767	<0.001	1.000	0.688	0.531	0.007	0.349	0.113	0.622
PGA ^{C4} _{global}	–	0.500	–	0.500	–	0.500	–	0.500	0.641	0.440	0.010	0.767	<0.001	1.000	0.688	0.531	0.007	0.349	0.113	0.622
PGA ^{C8} _{global}	–	0.500	–	0.500	–	0.500	–	0.500	0.641	0.440	0.010	0.767	<0.001	1.000	0.688	0.531	0.007	0.349	0.113	0.622
PGA ^{C2} _{grid}	0.641	0.560	0.641	0.560	0.641	0.560	0.641	0.560	–	0.500	<0.001	0.684	<0.001	0.881	0.497	0.561	0.902	0.507	0.459	0.566
PGA ^{C4} _{grid}	0.010	0.233	0.010	0.233	0.010	0.233	0.010	0.233	<0.001	0.316	–	0.500	<0.001	0.790	0.011	0.241	0.004	0.222	0.021	0.243
PGA ^{C8} _{grid}	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.000	<0.001	0.119	<0.001	0.210	–	0.500	<0.001	0.000	<0.001	0.000	<0.001	0.000
PGA ^{C2} _{island}	0.688	0.469	0.688	0.469	0.688	0.469	0.688	0.469	0.497	0.439	0.011	0.759	<0.001	1.000	–	0.500	0.016	0.331	0.249	0.589
PGA ^{C4} _{island}	0.007	0.651	0.007	0.651	0.007	0.651	0.007	0.651	0.902	0.493	0.004	0.778	<0.001	1.000	0.016	0.669	–	0.500	<0.001	0.726
PGA ^{C8} _{island}	0.113	0.378	0.113	0.378	0.113	0.378	0.113	0.378	0.459	0.434	0.021	0.757	<0.001	1.000	0.249	0.411	<0.001	0.274	–	0.500

Table 3.14: Average evaluation number and evaluation per second values achieved by executing 30 times SGA and PGAs on the three datasets and three clusters.

Model	<i>Log4j</i>		<i>Lucene</i>		<i>POI</i>	
	eval	eval/s	eval	eval/s	eval	eval/s
SGA	30 214	8.32	30 214	3.10	30 214	0.88
PGA _{global} ^{C2}	30 214	3.09	30 214	2.37	30 214	1.14
PGA _{global} ^{C4}	30 214	3.66	30 214	3.08	30 214	1.70
PGA _{global} ^{C8}	30 214	4.10	30 214	3.26	30 214	2.34
PGA _{grid} ^{C2}	30 500	2.43	30 500	1.90	30 500	1.01
PGA _{grid} ^{C4}	28 840	2.55	28 840	2.26	28 840	1.39
PGA _{grid} ^{C8}	26 260	2.62	26 260	2.47	26 260	2.16
PGA _{island} ^{C2}	30 180	14.74	30 180	5.12	30 180	1.63
PGA _{island} ^{C4}	30 478	27.88	30 478	10.53	30 478	3.23
PGA _{island} ^{C8}	30 083	51.87	30 083	21.60	30 083	6.80

3.7.5 Predictive Performance

As can be observed from Tables 3.15, 3.16, 3.17 reporting the median values of the four employed evaluation criteria, the predictive performance of the parallel models is negatively affected only in the case of the *Log4j* dataset using the PGA_{grid} models. Moreover, the results of the Wilcoxon and Vargha-Delaney tests confirm the above considerations. In Table 3.18 the statistical tests results about the *F-measure* are reported.

Table 3.15: Predictive performance median values achieved by executing 30 times SGA and PGAs on the *Log4j* dataset.

Model	<i>Precision</i>	<i>Recall</i>	<i>Accuracy</i>	<i>F-measure</i>
SGA	0.938	0.259	0.698	0.407
PGA _{global} ^{C2}	0.938	0.259	0.698	0.407
PGA _{global} ^{C4}	0.938	0.259	0.698	0.407
PGA _{global} ^{C8}	0.938	0.259	0.698	0.407
PGA _{grid} ^{C2}	0.889	0.106	0.834	0.190
PGA _{grid} ^{C4}	0.885	0.106	0.834	0.190
PGA _{grid} ^{C8}	0.889	0.106	0.834	0.190
PGA _{island} ^{C2}	0.933	0.251	0.705	0.396
PGA _{island} ^{C4}	0.937	0.291	0.673	0.444
PGA _{island} ^{C8}	0.933	0.259	0.700	0.406

Table 3.16: Predictive performance median values achieved by executing 30 times SGA and PGAs on the *Lucene* dataset.

Model	<i>Precision</i>	<i>Recall</i>	<i>Accuracy</i>	<i>F-measure</i>
SGA	0.617	0.901	0.393	0.733
PGA _{global} ^{C2}	0.617	0.901	0.393	0.733
PGA _{global} ^{C4}	0.617	0.901	0.393	0.733
PGA _{global} ^{C8}	0.617	0.901	0.393	0.733
PGA _{grid} ^{C2}	0.616	0.901	0.394	0.732
PGA _{grid} ^{C4}	0.616	0.901	0.394	0.732
PGA _{grid} ^{C8}	0.616	0.901	0.394	0.732
PGA _{island} ^{C2}	0.616	0.901	0.394	0.732
PGA _{island} ^{C4}	0.616	0.901	0.394	0.732
PGA _{island} ^{C8}	0.616	0.901	0.394	0.732

Table 3.17: Predictive performance median values achieved by executing 30 times SGA and PGAs on the *POI* dataset.

Model	<i>Precision</i>	<i>Recall</i>	<i>Accuracy</i>	<i>F-measure</i>
SGA	0.689	0.861	0.335	0.766
PGA _{global} ^{C2}	0.689	0.861	0.335	0.766
PGA _{global} ^{C4}	0.689	0.861	0.335	0.766
PGA _{global} ^{C8}	0.689	0.861	0.335	0.766
PGA _{grid} ^{C2}	0.689	0.861	0.335	0.766
PGA _{grid} ^{C4}	0.689	0.861	0.335	0.766
PGA _{grid} ^{C8}	0.689	0.861	0.335	0.766
PGA _{island} ^{C2}	0.689	0.861	0.335	0.766
PGA _{island} ^{C4}	0.689	0.861	0.335	0.766
PGA _{island} ^{C8}	0.689	0.861	0.335	0.766

 Table 3.18: Wilcoxon test (*p-values*) and Vargha-Delaney (\hat{A}_{12}) results for the comparison of the *F-measure* values between PGAs and SGA over 30 runs on the considered datasets.

Model	=	<i>Log4j</i>		<i>Lucene</i>		<i>POI</i>	
		<i>p-value</i>	\hat{A}_{12}	<i>p-value</i>	\hat{A}_{12}	<i>p-value</i>	\hat{A}_{12}
PGA _{global} ^{C2}	SGA	–	0.500	–	0.500	–	0.500
PGA _{global} ^{C4}	SGA	–	0.500	–	0.500	–	0.500
PGA _{global} ^{C8}	SGA	–	0.500	–	0.500	–	0.500
PGA _{grid} ^{C2}	SGA	<0.001	0.022	0.782	0.466	0.423	0.518
PGA _{grid} ^{C4}	SGA	<0.001	0.022	0.751	0.479	0.423	0.518
PGA _{grid} ^{C8}	SGA	<0.001	0.022	0.599	0.482	0.584	0.518
PGA _{island} ^{C2}	SGA	0.221	0.448	0.476	0.488	0.361	0.519
PGA _{island} ^{C4}	SGA	0.746	0.559	0.574	0.501	0.361	0.519
PGA _{island} ^{C8}	SGA	0.459	0.484	0.844	0.488	0.584	0.518

3.7.6 Cloud Costs Estimation

Figure 3.17 shows the estimation of costs for the most famous cloud provider for executing the same GAs of the experiments, in relation to the required execution time. Also, the estimation of SGA on a machine purchased on the same cloud provider, having the same configuration of the ones used for the Hadoop cluster, are included. As can be seen from the figure, in the case of PGA_{global} and PGA_{grid} , they achieve to save time against SGA in the case of the POI dataset but needing a greater cost because of the multiple machines. Instead, PGA_{island} requires almost the same cost of a single machine, since it is able to conclude its execution before SGA even if with a greater number of machines. The distance in time is more remarkable for the POI dataset, thus making the use of cloud worth in case of large computational load.

The complete report of cloud costs estimation for all the selected cloud providers is presented in Table 3.19.

Table 3.19: Estimation of cloud costs (USD) on 4 commercial cloud providers of the executions for SGA and PGAs on the three datasets and three clusters.

Model	<i>Log4j</i>				<i>Lucene</i>				<i>POI</i>			
	Amazon	DigitalOcean	Microsoft	Google	Amazon	DigitalOcean	Microsoft	Google	Amazon	DigitalOcean	Microsoft	Google
SGA	0.111	0.030	0.071	0.101	0.298	0.081	0.190	0.271	1.052	0.287	0.669	0.956
PGA ^{C2} _{global}	0.597	0.163	0.380	0.542	0.778	0.212	0.495	0.707	1.624	0.443	1.034	1.477
PGA ^{C4} _{global}	1.008	0.275	0.641	0.916	1.198	0.327	0.762	1.089	2.176	0.594	1.385	1.979
PGA ^{C8} _{global}	1.801	0.491	1.146	1.637	2.268	0.619	1.444	2.062	3.160	0.862	2.011	2.873
PGA ^{C2} _{grid}	0.766	0.209	0.488	0.697	0.983	0.268	0.625	0.893	1.842	0.502	1.172	1.675
PGA ^{C4} _{grid}	1.381	0.377	0.879	1.255	1.557	0.425	0.991	1.415	2.542	0.693	1.618	2.311
PGA ^{C8} _{grid}	2.447	0.667	1.557	2.225	2.596	0.708	1.652	2.360	2.972	0.810	1.891	2.701
PGA ^{C2} _{island}	0.125	0.034	0.080	0.114	0.360	0.098	0.229	0.327	1.133	0.309	0.721	1.030
PGA ^{C4} _{island}	0.134	0.036	0.085	0.121	0.354	0.097	0.225	0.322	1.153	0.315	0.734	1.048
PGA ^{C8} _{island}	0.142	0.039	0.090	0.129	0.340	0.093	0.217	0.309	1.081	0.295	0.688	0.983

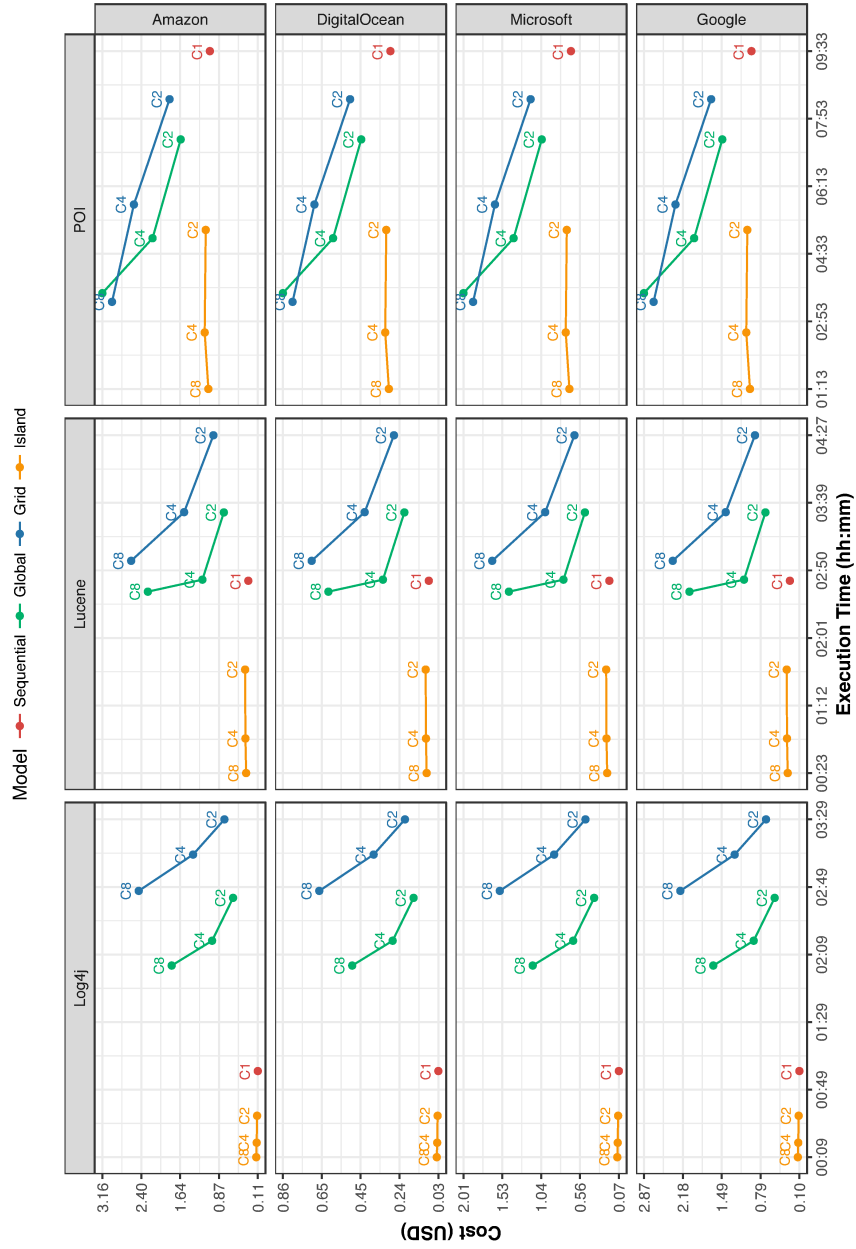


Figure 3.17: The cloud provider execution time and cost values for the execution of SGA and PGAs on the considered datasets and cluster sizes.

3.8 Summary

In this chapter, the parallelisation of GAs on the Hadoop MapReduce platform, based on three models, i.e., the global, grid and island models was addressed . As a benchmark problem, the use of three different PGAs to solve a software engineering problem of configuring the SVMs for inter-release fault prediction was considered. The effectiveness of these models in terms of execution time, speedup, overhead and computational effort was empirically assessed by using three publicly available datasets of real software-faults widely used in fault prediction studies. The three datasets were chosen considering their different size in order to varying the execution time of the GAs.

It emerged that the use of PGA based on the island model outperforms the use of sequential GA and the PGAs based on the global and grid models for all the considered datasets and cluster configurations. The overhead of data store (i.e., HDFS) accesses, communication and latency may impair parallel solutions based on the global and grid model when executed on small problem instances. This is not the case of the island model since it is able to reduce the number of operations performed on the data store, determining a faster execution of tasks and an optimised usage of resources. The use of the island model enabled to speed up the average execution time over the three datasets with respect to SGA of $7.0 \times$, $3.4 \times$ and $1.8 \times$ times by exploiting 8, 4 and 2 nodes, respectively. Moreover, the results of the estimation of the commercial cloud providers costs revealed that the island model is worth using also in term of costs against the execution with a single machine.

In general, a critical aspect in the use of Hadoop MapReduce is the presence of overhead due to the communication with the data store (i.e., HDFS). The distributed nature of the data store introduces an intrinsic communication latency that may drastically worsen the performance if multiple and useless operations are executed. To speed up the execution of tasks, it is useful to reduce data store operations, as it happens with the island model where data store access is limited to the migration phase only.

Chapter 4

Parallel Genetic Algorithms Using Software Containers

Contents

4.1	Introduction	81
4.2	Related Work	83
4.3	Background	85
4.4	System Design	89
4.5	Empirical Study Design	94
4.6	Results	100
4.7	Summary	109

4.1 Introduction

It is argued that a barrier to the wider application of parallel execution has been the high cost of parallel architectures and infrastructures and their management. Genetic Algorithms (GAs) have been effectively parallelised on multi-core (i.e., CPUs) and many-core (i.e., GPUs) systems [60, 59]. However, these solutions are often expensive and may obtain only a certain degree of parallelisation being strictly related to the number of multiple computational units available on the

hardware. On the contrary, technologies based on the network communication may hypothetically be scaled without limits. *Cloud computing* can represent an affordable solution to address the above issues because it breaks the barrier between employed resources and costs: in a short time it is possible to allocate a cluster of the desired size without investing in expensive local hardware and its management.

Even though Hadoop MapReduce, presented in Chapter 3, offers some appealing features, the problem with these solutions is that the data exchange through a distributed file system may slow down the execution of Parallel Genetic Algorithms (PGAs). Moreover, it is required to have dedicate skills for setup and maintenance activities, which often cannot be automatised, to have a fully operational cluster.

In this chapter, *AMQPGA* is presented, an approach to distributing GAs, implementing the global parallelisation model and exploiting technologies especially devised for the cloud (i.e., *Docker*, *CoreOS* and *RabbitMQ*) to fully take advantage of the appealing cloud features of fault-tolerance, scalability and performance optimisation. It also allows GAs developers to use existing implementations of genetic operators or external tools, without constraints on the adopted programming languages. Indeed, ‘software containers’ (i.e., Docker containers) provide isolated environments (i.e., virtual Linux instances) where developers can include everything is needed for the computation. Moreover, to exploit the DevOps (‘development’ and ‘operations’) methodology that determines easy cloud development and deployment processes, a conceptual workflow is proposed to support the development, deployment and execution of distributed GAs, reducing the human effort. The source code is shared at the address <https://github.com/pasqualesalza/amqpga> under the terms of the *MIT License*¹.

The main contributions of this chapter can be summarised as follows:

- design and realise a system to deploy containers of distributed GA applications in cloud environments, by implementing the global parallelisation

¹<https://opensource.org/licenses/MIT>

model and exploiting the Advanced Message Queuing Protocol (AMQP);

- provide a conceptual workflow which describes all the phases of development, deployment and execution of distributed GAs;
- carry out an empirical study to assess the effectiveness of the model regarding the execution time, speedup, overhead and setup time.

The rest of the chapter is organised as follows. Section 4.2 describes some relevant related work. In Section 4.3 the main features of the employed cloud technologies are summarised whereas in Section 4.4 the proposed system and conceptual workflow for deployment and execution of GAs in cloud environments are presented. Section 4.5 and Section 4.6 report, respectively, the design and the results of the empirical study carried out to assess the effectiveness of the proposed approach. Section 4.7 concludes with some final remarks.

4.2 Related Work

A wide range of work is present in the literature about models and technologies for GAs parallelisation [38]. However, this work aims to parallelise GAs on a commercial cloud environment and in a DevOps fashion, so here only the most relevant related work is reported, involving models, technologies, problems and conceptual deployment workflows in the GAs field or the more general Evolutionary Algorithms (EAs) one. This section extends Section 3.2.

Zheng et al. [60] compared the multi-core (i.e., CPUs) and the many-core (i.e., GPUs) systems for GAs parallelisation. Firstly, they found that the system based on GPUs is faster than the CPUs one. However, they observed that an architecture with a fixed number of parallel participants, such as GPU cores, might perform worse in terms of quality of solutions than another with more parallel nodes stating that distributed architectures, e.g., the cloud, are worth for GAs parallelisation.

In addition to the work proposed in Chapter 3, many authors used the

MapReduce paradigm to implement PGAs [32, 14] and some of them with *Hadoop MapReduce* in particular [54, 12]. On the one hand, they claimed that GAs can efficiently scale on multiple Hadoop nodes. On the other hand, it has been highlighted that the overhead is a worrying presence, due to the communication with the data store (i.e., Hadoop Distributed File System (HDFS)). In general, Hadoop MapReduce represents one of the most mature and employed technologies to develop parallel algorithms since it provides a ready to use distributed infrastructure that is scalable, reliable and fault-tolerant [30]. Nevertheless, it requires high performance from the underlying hardware. Moreover, even if using a framework such as *elephant56*, presented in Section 3.4, Hadoop is not suitable for all since specific skills for setup and maintenance activities are needed. Instead, other cloud technologies are affordable, and the scalability and fault-tolerance features can be obtained from the design of the distributed applications themselves.

Another aspect that is strictly connected to this work is the conservation of the metaheuristic nature of GAs, thus allowing the system to be adapted to a wide variety of problems. Even though being related to the EAs in general, a first attempt of generalisation was given from Fazenda et al. [17], who considered the parallelisation of EAs on the Hadoop MapReduce platform in a general purpose form of a library. The work has been further enhanced by Veeramachaneni et al. to produce *FlexGP* [53], which is probably the first large scale Genetic Programming (GP) system that runs in the cloud, implemented over *Amazon EC2* with a socket-based client/server architecture. This aspect is addressed here by using software containers that allow defining any environment for GAs execution.

With cloud computing is possible to satisfy almost any problem need, by mixing a large variety of technologies. Moreover, cloud-specific development methodologies (i.e., DevOps) can ease the production of parallel GAs (see Section 4.4.3). As a first attempt of employing cloud technologies, Merelo Guervós et al. devised *SofEA* [41], a model for Pool-based EAs in the cloud, an evolutionary algorithm mapped to a central *CouchDB* object store. *SofEA*

provides an asynchronous and distributed system for individuals evaluations and genetic operators application. Later, they defined and implemented the *EvoSpace Model* [23], consisting of two main components: a repository storing the evolving population and some remote workers, which execute the actual evolutionary process. It is the first work to involve technologies on the *Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)* level: *Heroku* as PaaS for the population store and *PiCloud* as SaaS for the computing operations. Not only does the work show how EAs can scale on the cloud, but also how the cloud can make EAs effective in a real world environment, speeding up the running time and lowering the costs. The work proposed in this chapter aims at exploiting both IaaS and PaaS, using software containers.

4.3 Background

In this section, some background about the involved technologies and communication protocols is provided. The *container-based virtualisation* and its related most famous utility, i.e., *Docker*, are presented, respectively, in Section 4.3.2 and Section 4.3.1. Section 4.3.3 describes *CoreOS*, the employed technology of containers distributed orchestration whereas Section 4.3.4 illustrates the *Advanced Message Queueing Protocol* used for the system design, for communication together with its most famous implementation, i.e., *RabbitMQ*.

4.3.1 Container-Based Virtualisation

The basic idea behind the classic *hypervisor-based virtualisation* is to emulate the underlying physical hardware, creating a new virtual one and installing a fully working operating system on it. It is the typical model adopted by cloud providers, because of its ability to make hardware shareable and easily maintainable. Even though there are many existent techniques to optimise resources sharing (e.g., the *bare metal virtualisation*), the hypervisor-based virtualisation can be considered as limited in terms of performance. It is true especially when the aim is to execute cloud applications, where several service instances may

require to be created and destroyed in seconds to guarantee the reliability and scalability of the entire system.

While with the hypervisor-based virtualisation everything is performed on the hardware level, the *container-based virtualisation* operates at the operating system level. It provides a lightweight virtual environment, i.e., the *software container*, that groups and isolates a set of processes and hardware resources from the host and any other container. The main difference with the hypervisor-based virtualisation consists in the fact that all containers share the same kernel of the host system, instead of virtualising it, resulting in a high-performance resource utilisation. For this reason, containers are also much smaller and lightweight compared to an entire virtualised operating system. With the isolation, a process inside the container cannot directly see a process or resource outside the container itself, and the network is the only vehicle for communication.

Containers and its features are not such a new technology. Indeed, it was 1979 when, for the first time, it was made possible to create a new root filesystem inside an existing one, using a feature named 'chroot'. This isolation feature was then evolved into the Linux 'namespaces' technology that not only does it offer the isolation of the filesystem, but also of other system resources such as network interfaces. In this way, processes can run in an environment where the resources appear to be dedicated to them. The term 'process container' was first used around late 2006, then renamed to 'control groups' (abbreviated as 'cgroups') in 2007, as a Linux kernel feature available since v2.6.24. While namespaces isolates processes, cgroups lets the user limit the hardware resources for them. The combination of namespaces and cgroups is the basis of the modern *Linux Containers (LXC)* on which Docker was built on² and that Docker simplifies, especially when the aim is to containerise applications.

²Docker stopped running on top of LXC by default since version 0.9, in favour of 'libcontainer', based on namespaces and cgroups as well.

4.3.2 Docker

Docker is an open source container orchestration engine that separates applications from the underlying Linux operating system. With Docker it is possible to manage software containers, which are intended to contain every component of an application. From the application perspective, there is no difference between an execution on a dedicated machine and inside the container: the application is run in a short time in a full isolated Linux environment and can find others only by using the network. This reduces drastically the activities of installation and maintenance of applications: configuration management methodologies can define the environments and the application can be tested during the process from development to actual production execution, in a DevOps fashion (see Section 4.4.3). Docker creates containers from the 'images', i.e., basically read-only templates. Docker also provides an on line registry called 'Docker Hub' where it is possible to push/pull images to/from it. Docker images and registry allow to instantiate containers without repeating installation and build operations. The images can be created through two different operations: by executing operations directly on running containers and saving their state; by executing 'Dockerfiles', a set of instructions which can be maintained in the same way as the source code. Docker is not the only alternative in the field of containers management (e.g., *runC*, *rkt*), but it is currently the most mature product.

4.3.3 CoreOS

If Docker orchestrates containers in a single hosting machine, *CoreOS* can do it on a distributed cluster. CoreOS is an open source lightweight operating system based on a build of *Chrome OS* by *Google*. It allows building large and scalable deployments on varied infrastructure simple to manage, focusing on security, consistency and reliability. CoreOS provides only minimal functionalities required to execute applications inside Docker containers³.

³Currently, CoreOS is developing its own software container technology called 'rkt'.

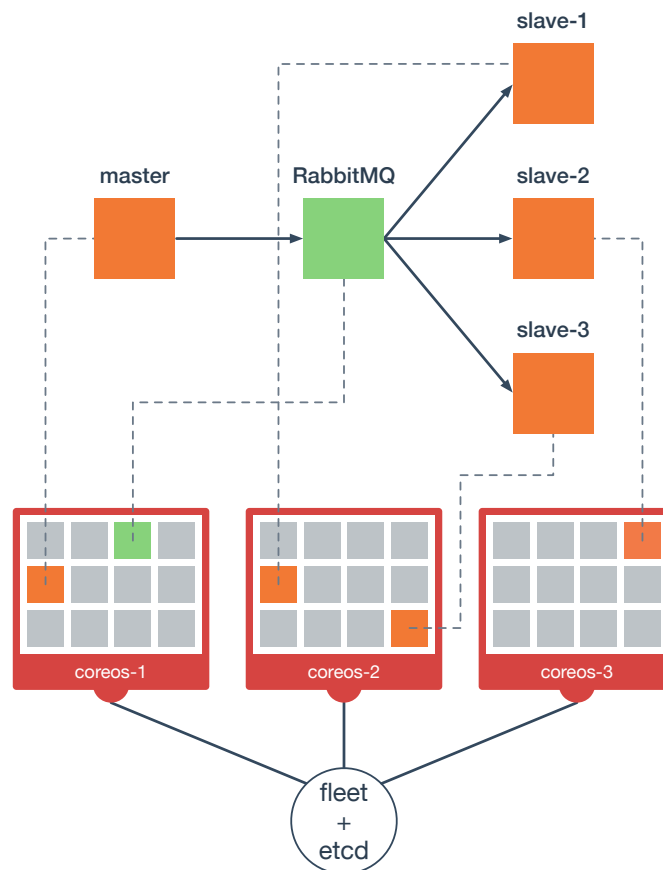


Figure 4.1: CoreOS containers distribution.

Figure 4.1 shows an instance of execution of the proposed system on CoreOS. To manage the cluster, CoreOS exploits a globally distributed key-value store called 'etcd'. Not only does it allow the CoreOS cluster configuration, but also it can be exploited by users as a central point for automatic applications configuration and discovery of other components in the network. The scheduling of containers is managed by a tool called 'fleet' that serves as a cluster-aware init system. It extends on a cluster scale *systemd*, the modern single machine Linux init system. It accepts the requests of containers allocation and schedules assignments to machines in the cluster on an optimisation basis, probing both cluster and applications health.

etcd and fleet were preferred to use over other alternatives (e.g., *Kubernetes*, *Mesos*) because they are, at the same time, lightweight regarding the resource allocation and complete of everything needed to realise the system. Moreover, it is also available as a cloud instance image on the majority of public cloud

providers and thus avoiding the 'lock in' to specific services.

4.3.4 AMQP and RabbitMQ

RabbitMQ is an open source 'message broker' software that implements AMQP. It is written in *Erlang* language and client libraries are available for the majority of programming languages. It is a component able to accept and forward messages, which can consist of plain text or blobs of binary data. Message brokers cover each stage of the exchange setup among participants, namely the 'publishers' and 'consumers'. The publishers produce messages and the consumers pick and process them. It is the job of the message broker to ensure that the messages go from a publisher to the right consumer, based on a chosen scheduling policy. The primary recipient of messages is the 'queue', a potentially unlimited buffer of data, which lives inside RabbitMQ. If the publisher and consumers are connected to a queue, they can communicate with each other without actually knowing each other. It makes RabbitMQ a powerful tool for scalable distribution of tasks since it is possible to add and remove participants without breaking the communication.

RabbitMQ has other contestants regarding the AMQP implementation, but no one has, at the same time, a message broker, High Availability (HA) capabilities, many client and developer tools available for the majority of programming languages, besides being easily deployable as Docker containers. Furthermore, differently from other communication technologies, RabbitMQ can easily sustain a distributed infrastructure without requiring any other discovery technology.

4.4 System Design

In this section the design and implementation of the system to parallelise GAs in the cloud are presented. In Section 4.4.1 the design of the architecture is described. Section 4.4.2 illustrates how the communication protocol to parallelise the master/slave model for GAs was managed. Finally, to better

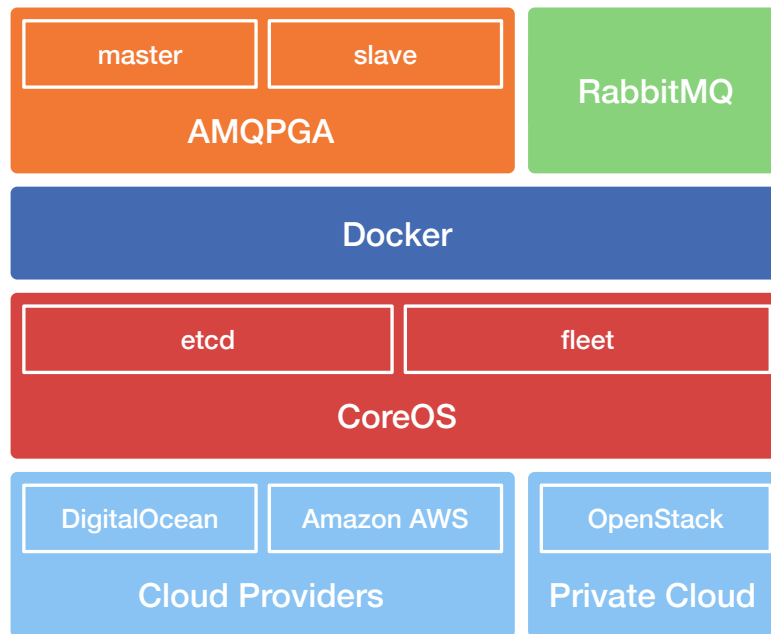


Figure 4.2: The involved architecture layers.

understand how the system can be actually employed in a real world scenario, a conceptual workflow is provided in Section 4.4.3, describing all the phases and participants for development, deployment and final execution of parallel GAs in the cloud.

4.4.1 Architecture Design

Figure 4.2 shows the ensemble of the involved components. The base layer is composed by the cloud infrastructure able of allocating virtual instances of CoreOS, which has been chosen as the cluster manager. With the aim of providing a system as more general and flexible as possible, it allows to consider indifferently commercial cloud providers (e.g., *DigitalOcean*, *Amazon AWS*, *Windows Azure*) and private cloud environments (e.g., *OpenStack*). It is possible since the only requirement for clusterisation is the availability of the CoreOS image, thus breaking the limits in number and resource usage that single providers may impose to the users. Both the main services of CoreOS were employed: *fleet* as the deployment manager and *etcd* as the central configuration point for discovery purposes.

As mentioned in Section 4.3.3 and Section 4.3.2, while CoreOS manages the

machines in the cluster and scheduling aspects, Docker manages the download of containers and their execution on the machines assigned by CoreOS. The powerful feature of Docker of executing an entire environment potentially makes possible the implementation of any genetic operator, in any preferred programming language or by using any external tool.

The two main application services of this proposal exploit the underlying interfaces of CoreOS and Docker. The first is a running container of RabbitMQ whereas the GA cloud implementation, which was named 'AMQPGA' in this work, runs in the form of one master and multiple slave containers, communicating through the RabbitMQ service. Figure 4.1 depicts the above situation on the cluster, where CoreOS schedules all the containers in order to optimise the resources load of the execution.

For the implementation, *Go* was chosen, an open-source programming language by Google. In particular, *Go* simplifies the GA processes and build small containerised environments. It is worth noting that it would be possible to switch the *Go* clients with clients developed with any programming language. The only requirements consist in being able of communicating with RabbitMQ, serialising individuals with the same codification algorithm and respecting the devised communication protocol.

4.4.2 Master/Slave Parallelisation GA with AMQP

The global parallelisation model (also known as the master/slave model) was implemented, where a master node executes the GA generations on the whole population except for the fitness evaluation, which is demanded to distributed slave nodes. This implementation was named 'AMQPGA'. Once the fitness values have been computed, the individuals go back to the master node where the other genetic operators can be applied.

The global model was adapted to the AMQP model, implemented with a combination of *Go* workers and a running RabbitMQ service. The resulting algorithm is an application of the *Remote Procedure Call (RPC)* pattern, depicted in Figure 4.3: 1. the master node publishes the messages (i.e., the individuals)

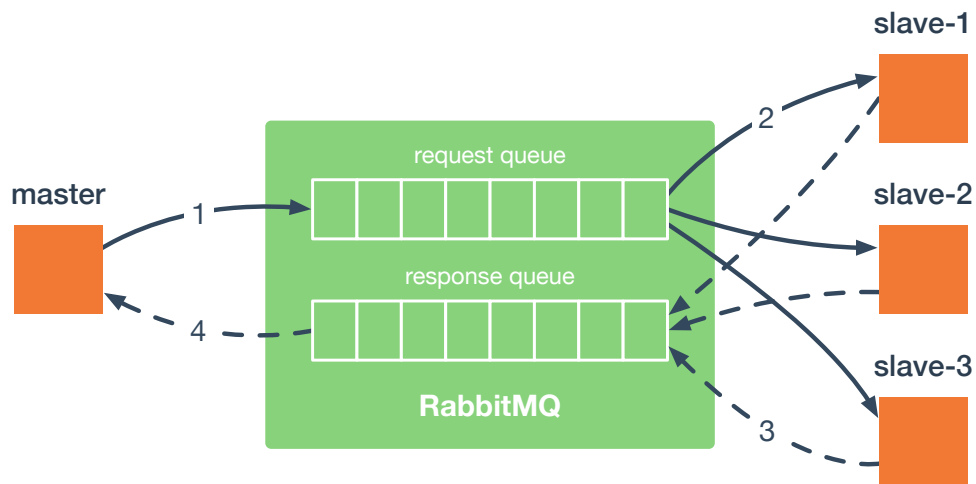


Figure 4.3: The AMQPGA algorithm.

on the request queue; 2. RabbitMQ dispatches the individuals to the subscribed slave nodes, in a round-robin fashion (i.e., assignments are made in equal portions and circular order); 3. the slave nodes process the individuals by computing the fitness function values and publish them on the response queue; 4. the only consumer of the response queue, i.e., the master node, takes back all the individuals and continues the computation until the next generation. Using the message broker as the central point for the computation, it was possible to add any number of further slave nodes to the GA, even at run time, making the system scalable.

4.4.3 Development, Deployment and Execution Workflow

A conceptual workflow for a possible real world scenario is described, in which the development, deployment and execution of distributed GAs are performed in a DevOps fashion. The participation of two different or correspondent actors are expected: the developer and the user.

Figure 4.4 depicts the workflow. From the developer point of view:

1. the developer pushes the source code s/he developed to its public or private *Git* repository. Together with the source code, there will be a *Dockerfile* defining the environment;

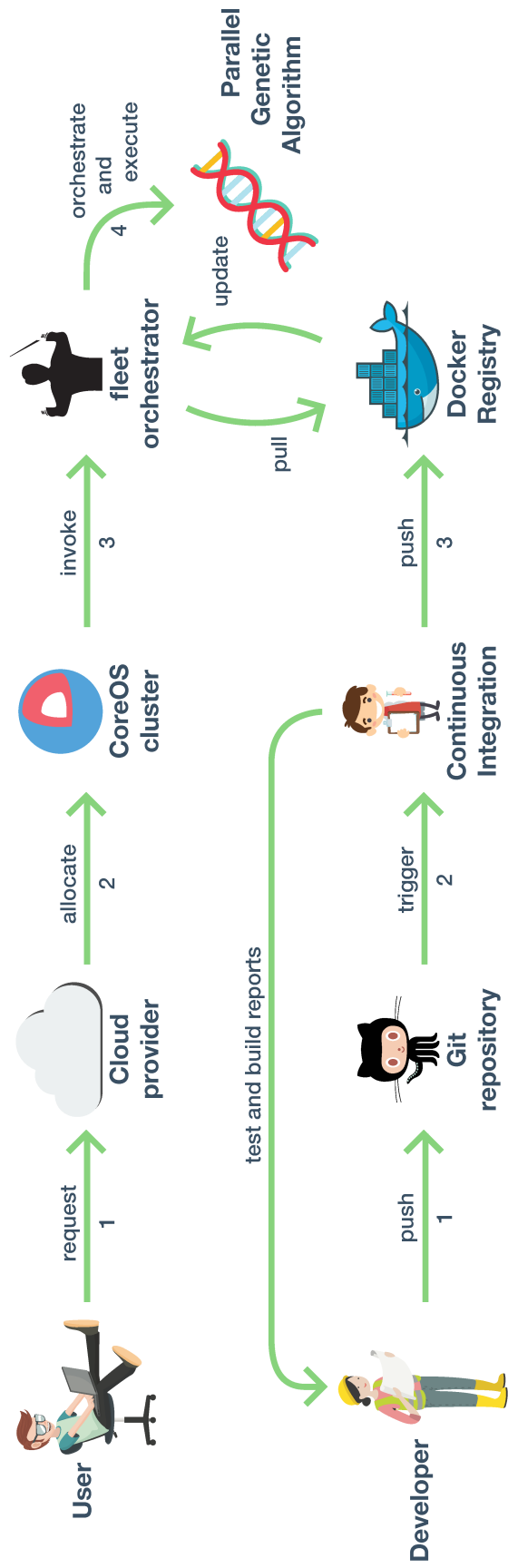


Figure 4.4: GAs development, deployment and execution workflow.

2. with the hook mechanism, the Git repository triggers a continuous integration service that executes, at the same time, both the integration testing of source code and Docker image build;
3. if tests and build have succeeded, a report is sent to the developer and the Docker image pushed in a *Docker Registry*, i.e., a repository for images that can be either the public *Docker Hub* or a private one.

From the user perspective:

1. the user submits a request to a cloud provider and a cluster of the required number of CoreOS nodes is allocated;
2. the user demands to CoreOS for executing the GA with a certain configuration;
3. CoreOS invokes fleet which pulls the Docker image of GA implementation and any other useful service images (e.g., RabbitMQ in this case) from the Docker registry, if there is a newer version available;
4. fleet is ready to orchestrate containers and start the execution of the distributed GA.

4.5 Empirical Study Design

The aim of this work was to understand if the proposed approach can be an effective solution to improve the scalability of GAs. Therefore, it was first needed to verify if GAs parallelised using cloud technologies allow to get a better execution time compared to the sequential version. Moreover, it was also important to quantify the setup time required to have the infrastructure ready to execute the GAs. Thus, the following research question was defined:

RQ *Is the use of the system based on a combination of software containers, message queues and cloud orchestration effective for parallel GAs against the sequential execution?*

Considering that the global parallelisation model is parallelised only during the fitness evaluation, to address the **RQ** a dummy fitness evaluation function was considered as a benchmark, which does nothing except receiving individuals, sleep for a specified time and return a random fitness value to the master [7]. The choice of this dummy function, together with the variation of the network load (i.e., the chromosome size), was motivated by the fact that it allowed to assess the GAs scalability considering different problem sizes by just varying the sleep time. Moreover, the actual time required to have the cloud infrastructure ready to execute the parallel GAs was tested. It is worth noting that in global parallelisation model the populations evolve in the same way as the sequential version. For this reason, any quality results are mentioned.

Details about the problem and GAs configuration are provided in Section 4.5.1. The hardware employed to run the experiments is reported in Section 4.5.2. To understand the effectiveness of the system, the experimental method described in Section 4.5.3 were applied and several evaluation criteria employed, namely the execution time, speedup, overhead and setup time, described in Section 4.5.4. Finally, Section 4.5.5 analyses some threats to validity that may have affected the experimentation.

4.5.1 Experiment Configuration

To understand how the system behaves, the attention was focused only on the fitness evaluation time, since it is the only parallelisation part of the execution of the global parallelisation model [7]. Nevertheless, a full GA was implemented and executed to reproduce a real scenario in which, besides execution time, other system resources are consumed, and the GA needs to fit into provided limits (e.g., the memory). The GA has been configured following other studies parameters [2, 23]. A population of 10 000 individuals was initialised to let the problem be large enough and splittable to multiple slaves. The GA was run for 10 generations and varied the chromosome size according to the experimental method (see Section 4.5.3). As for the other genetic operators, The tournament selection with size 2 was chosen, together with the two point crossover with a

rate of 0.85 and a mutation rate of $\frac{1}{c}$, where c is the chromosome size.

4.5.2 Hardware

As execution bench for the experiments, several instances from the *DigitalOcean* cloud provider were rent . The cloud clusters employing CoreOS was composed using 1 instance dedicated to the master, 1 instance for RabbitMQ and varying the size of instances in 1, 2, 4, 8, 16, 32, 64 and 128 for the slaves to test the scalability of the executions. With the aim of maintaining a low budget, only small instances of virtual machines were selected, which consisted in 1 core processor, 512 MB of memory and 20 GB of SSD disk for \$0.007 h. An exception for the *RabbitMQ* node was made since it requires at least 1 GB of RAM. The configuration of each cloud instance is summarised in Table 4.1.

Table 4.1: The cloud instances configuration.

Type	Feature	Value
Hardware	Architecture	64 bit
	CPUs	1
	RAM	512 MB
	Storage	20 GB
Software	CoreOS	1185.5.0
	Docker	1.12.3
	RabbitMQ	3.6.6
	Go	1.7

4.5.3 Experimental Method

The **RQ** was addressed by comparing the performance of the parallel and sequential GAs with different configurations. The sleep time of the employed dummy fitness function was varied of 0.01 ms, 0.1 ms, 1 ms, 10 ms and 100 ms in order to benchmark different computational times. The network was stressed by varying the individual size (i.e., the chromosome size) in 128, 256, 512, 1024, 2048, 4096, 8192, 16 384, 32 768 and 65 536 genes, where each gene is encoded with 8 bit. It is worth noting that it was not possible to use a larger chromo-

some size, due to the memory limits of the sequential execution on a single machine. All the parallel GAs were executed on different cluster configurations characterised by a different number of nodes (see details in Section 4.5.2). The master node did not participate in any parallel fitness computation since the main interest was in observing the behaviour of peer communication with RabbitMQ. For each combination of sleep time, chromosome size and cluster configuration, 10 runs were executed.

4.5.4 Evaluation Criteria

To compare the performance of the executed experiments, the best practice in reporting the results with PGAs, identified by Luque and Alba [38], were followed. The evaluation criteria, already considered in Section 3.6.4, are briefly repeated in the following. Moreover, the setup times of the cloud infrastructure are also evaluated .

Execution Time

The execution time was measured in milliseconds (ms) using the system clock. As a performance indicator of the whole execution, the execution times achieved by executing all the fitness evaluation phases of sequential and PGAs were compared . The partial times were distinguished into computation and overhead times only in a second step, when the interest was to quantify the time spent for parallel communication.

Speedup

The speedup is defined as:

$$S = \frac{T_S}{T_P}$$

where T_S is the sequential execution time and T_P the parallel execution time. The achieved speedup was compared with respect to the ideal speedup, which is equal to the number of the involved parallel nodes and corresponds to the situation when the sequential execution time is perfectly split among multiple

nodes. The ideal speedup is rarely achieved in practice due to the presence of overhead, but it is usually taken into consideration as an upper limit to compare the performance of parallel algorithms [38].

Overhead

To understand the reasons that prevent the parallel GAs to have a speedup near to the ideal one, the overhead for each execution was quantified. The time of each execution was considered and distinguished into overhead and computation times. The computation time was defined as:

$$T_C = \frac{T_S}{P}$$

where T_S is the sequential time to compute the fitness evaluation function for the whole population and P the number of parallel slaves. Thus, the overhead time is:

$$T_O = T_P - T_C$$

computable if T_P , i.e., the parallel execution time, is given.

Setup Time

One of the points about the proposed system that was taken particularly in consideration is its feasibility in a real world context. To this aim, the setup time required to have the cloud infrastructure ready to execute the GAs was experimented. This automated activity was discriminated into two different times:

- ‘creation’, i.e., the necessary time to acquire the virtual instances from the cloud provider (i.e., DigitalOcean) and let all the machines be recognised as part of the CoreOS cluster;
- ‘deployment’, i.e., the time required to pull the GA and RabbitMQ images from the *Docker Hub* repository, schedule and run the containers on all the participant machines.

Statistical Tests

10 generations and 10 runs were executed for a total of 100 registered times for each experiment configuration, in order to cope with the inherent randomness of dynamic execution time and reported the average results.

To support all the considerations about the obtained results, the non-parametric inferential statistical test was performed, i.e., the *Wilcoxon Test* [9], as recommended in the literature [3, 29, 38]. For all the statistical tests, a probability of 5% of committing a Type-I-Error, i.e., the significance level, was accepted.

Furthermore, the Vargha-Delaney \hat{A}_{12} effect size [52] was used to characterise the magnitude of difference. The magnitude values can be summarised by 4 nominal values, namely the ‘negligible’, ‘small’, ‘medium’ and ‘large’.

4.5.5 Threats to Validity

Threats to *construct validity* concern the relationship between the theory behind the experiments and the observations. In order to alleviate possible threats related to measurement, the GAs execution time was quantified using the system clock, because it represents the speed of a technique to the end-user.

Threats to *internal validity* concern any confounding factors that could influence the results. A possible threat is related to the randomness due to the use of GAs and variable computational/network load on the nodes at the time of the experiment. Indeed, GAs are intrinsically random, and such a threat was mitigated by executing all the experiments 10 times, with 10 generations each, and presenting the average results [3, 29]. Furthermore, the nodes may have been biased by the randomness of system events, and the multiple runs were intended to alleviate these issues as well.

Threats to *external validity* concern the generalisability of the findings outside the scope of this study. An external threat is due to the fact that the PGAs was benchmarked on a particular cloud provider (i.e., DigitalOcean) whose machines performance may differ from other providers. For this reason, the

times concerning the computation were carefully separated from the setup ones. Moreover, the results of this study can be considered as an analysis of the 'execution trends' instead of absolute values, obtained by proportionally varying the configuration parameters of the experiments.

4.6 Results

In this section, the results of this study are presented. The comparison between sequential and parallel GAs, with regarding the execution time, is reported in Section 4.6.1. The analyses of the speedup and overhead are reported in Section 4.6.2 and Section 4.6.3, respectively. The setup time is analysed in Section 4.6.4.

4.6.1 Execution Time

Figure 4.5 shows the boxplots of the achieved execution times of the fitness evaluation phase on the whole population, including the time for the communication. Different combinations of the fitness evaluation time for a single individual (T_f), i.e., the sleep time of the dummy function, the chromosome (c) and cluster sizes (P) were experimented. Moreover, each generation employed a total of 1000 individuals. 10 runs of 10 generations each were performed, registering a total of 100 observations for each combination.

As expected, when increasing the chromosome size, the execution time for the same cluster size and individual fitness time increases proportionally. Focusing on the execution with 1 slave node, for which the communication is not affected by the message broker scheduling policy, a statistical test was carried out. The distributions of consecutive chromosome sizes was iteratively compared looking for the first threshold where the p -value of the two-tailed Wilcoxon signed-ranks test was less than the level of significance of 0.05 (i.e., accepting the alternative hypothesis of equality). To strengthen the differences, the Vargha-Delaney test was also employed considering as different only the couples having a magnitude level equal to *medium* or *large*. It was possible to

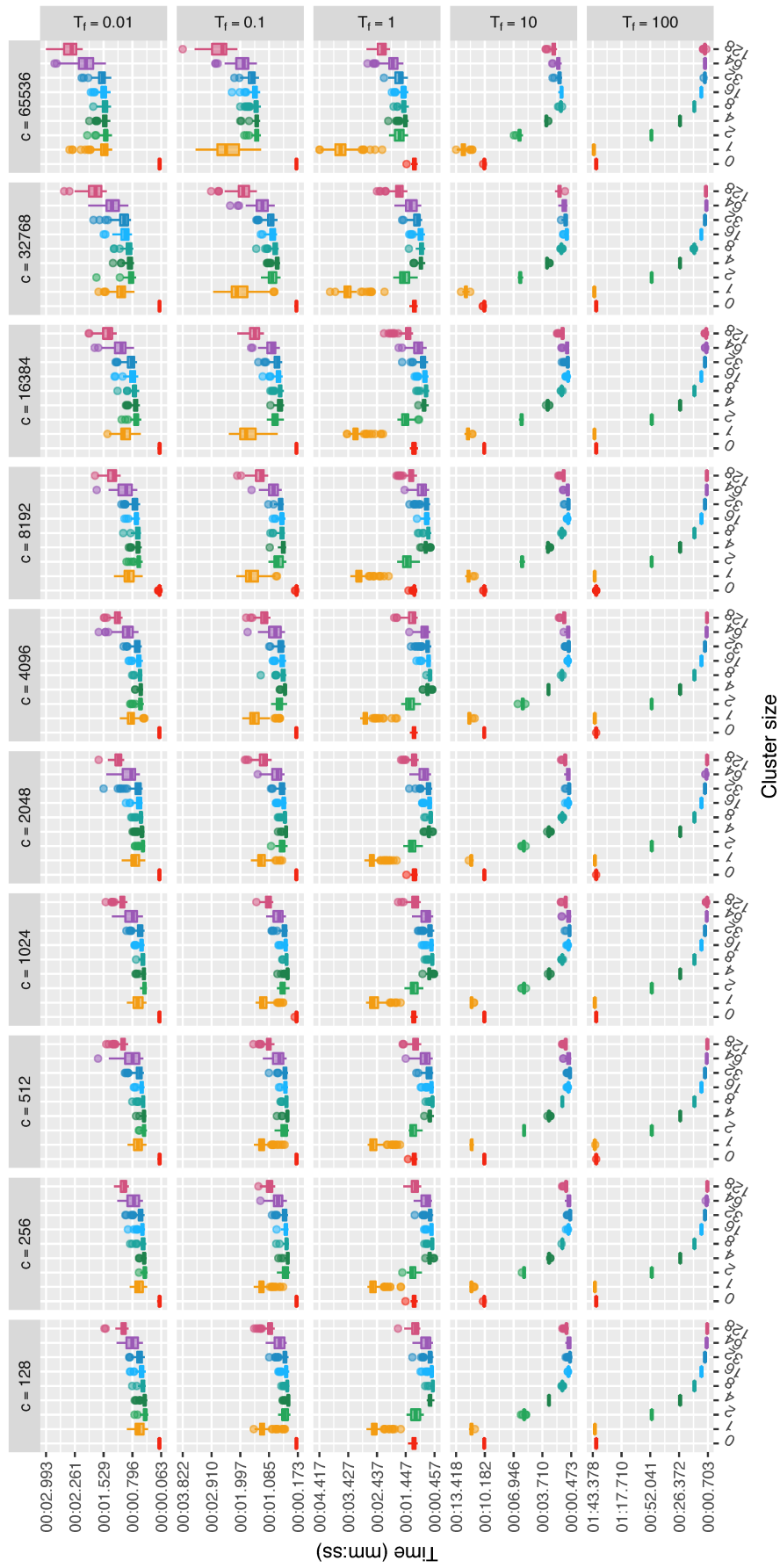


Figure 4.5: The execution times achieved by the sequential and PGAs.

notice that the execution time starts to be greater than smaller chromosome sizes only from 2048 genes on, for all the values of the individual fitness times. Even if the passage through a message queues communication system, i.e., the *RabbitMQ* message broker, adds a certain amount of latency in communication, the chromosome size variation effect is imperceptible if considering that the time is reported in the order of seconds. It can be due to the fact that the network of the employed provider is capable of offering a network speed much higher than what is needed.

Table 4.2: The intervals for which the execution time decreases when increasing the number of parallel nodes.

c	$T_f = 1$		$T_f = 10$		$T_f = 100$	
	P_{min}	P_{max}	P_{min}	P_{max}	P_{min}	P_{max}
128	2	8	2	32	2	128
256	4	8	2	32	2	128
512	4	8	2	32	2	128
1024	4	8	2	32	2	128
2048	4	8	2	32	2	128
4096	4	8	2	32	2	128
8192	4	8	2	32	2	64
16384	4	8	2	16	2	64
32768	4	8	2	16	2	64
65536	—	—	2	16	2	64

As for the scalability on the number of nodes, as easily visible from Figure 4.5, the system begins to scale from $T_f = 1$ on. It is clear that there are some minimum and maximum thresholds of the cluster sizes within the system scales. To support this assumption, a statistical test was performed, whose results are showed in Table 4.2. For each chromosome size and individual fitness time combination, the first cluster size observations group (P_{min}) was considered, whose distribution was significantly greater than the sequential execution one. It was obtained by performing a single-tailed Wilcoxon signed-ranks test setting the level of significance to 0.05. Then, if a minimum threshold was found, the next couples of consecutive cluster size distributions was iteratively compared until the alternative hypothesis by means of the Wilcoxon test

was rejected, meaning that the execution time is not significantly decreasing anymore. Thus, that point was marked as the maximum threshold (P_{max}). As showed in Table 4.2, the system succeeds in scaling for few nodes for $T_f = 1$. It is possible to notice that increasing the T_f helps the system to scale, whereas the chromosome size does not. On the one hand, because of the communication protocol, the parallel nodes must alternate the phase of receiving, computing of fitness evaluation function and sending of individuals. For this reason, the increment of the chromosome size increases the communication time, even forcing the slaves to be idle for a certain time until the next individuals have been made available from the master. On the other hand, the increment of the individual fitness time makes the communication time irrelevant against the computation one thus splitting more linearly the execution times between parallel nodes. Except for the cluster with one node (i.e., cluster size of 1), where the resulting execution time is obviously greater, the clusters with multiple nodes outperform the sequential execution (i.e., cluster size of 0). It means that the execution time is directly proportioned to the individual fitness evaluation time and, as hoped, inversely proportioned to the number of cluster nodes.

4.6.2 Speedup

The speedup characterises the scalability factor when the number of slave nodes is increasing against the sequential execution. The values are shown in Figure 4.6, distinguishing the individual fitness time and the chromosome size. Table 4.3 reports values on average of the 100 observations for the most positive case of $T_f = 10$ and $T_f = 100$ from the above analysis.

As mentioned in Section 4.5, the GAs begins to scale effectively from $T_f = 1$ on. For $T_f = 10$ and $T_f = 100$, the speedup values tend to the linear speedup according to the thresholds observed in Section 4.6.1. The observed values suggest that an employment of a T_f having at least a certain complexity, in terms of execution time, is the only requirement that makes the GA based on the master/slave model effectively scalable on multiple nodes, tending to linear scalability.

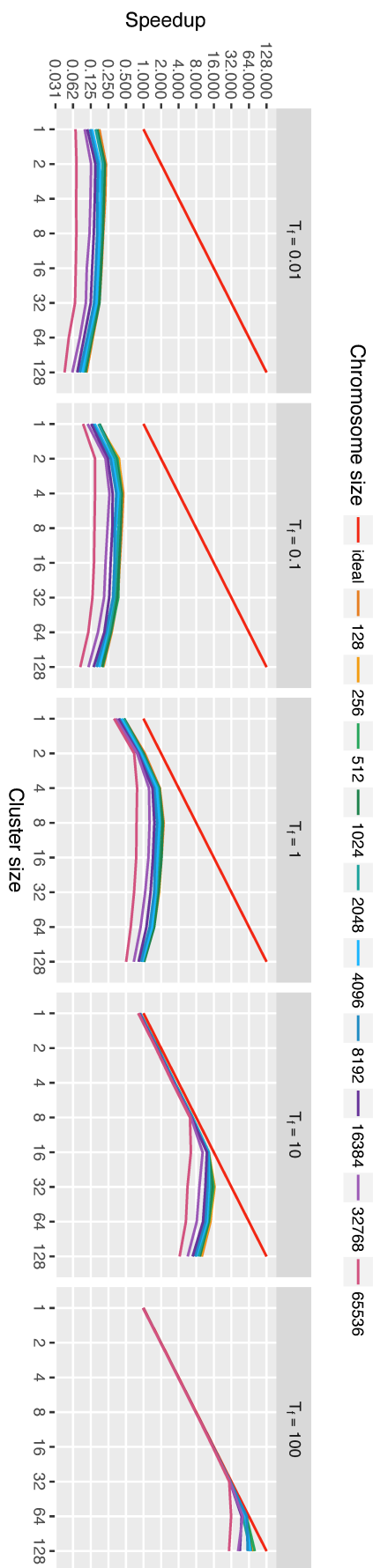


Figure 4.6: The speedup trend.

Table 4.3: The speedup values for $T_f = 10$, $T_f = 100$ and each chromosome size combination.

c	$T_f = 10$								$T_f = 100$							
	1	2	4	8	16	32	64	128	1	2	4	8	16	32	64	128
128	0.877	1.779	3.455	6.957	13.138	16.038	14.058	9.969	0.988	1.975	3.946	7.891	15.628	30.486	56.681	74.939
256	0.877	1.775	3.465	6.958	13.140	16.504	13.831	9.678	0.989	1.975	3.951	7.895	15.585	30.749	52.914	78.033
512	0.879	1.774	3.451	6.967	13.047	15.854	13.215	9.227	0.988	1.977	3.949	7.886	15.616	30.815	57.994	80.539
1024	0.877	1.770	3.463	6.934	13.146	14.994	12.931	9.210	0.988	1.976	3.949	7.885	15.614	30.645	57.135	69.973
2048	0.875	1.765	3.434	6.941	13.118	13.807	12.509	8.741	0.989	1.978	3.951	7.888	15.589	30.321	50.488	67.842
4096	0.861	1.731	3.398	6.777	12.882	13.096	12.135	7.943	0.988	1.973	3.945	7.881	15.553	30.636	55.738	68.073
8192	0.853	1.709	3.335	6.689	12.528	12.467	11.463	7.642	0.986	1.972	3.943	7.868	15.500	30.403	58.775	60.880
16384	0.849	1.694	3.295	6.642	12.171	11.476	10.468	6.957	0.985	1.973	3.940	7.868	15.453	30.121	47.978	44.765
32768	0.832	1.653	3.216	6.501	10.296	9.058	8.170	5.720	0.983	1.971	3.927	7.812	15.382	30.212	49.961	41.780
65536	0.814	1.628	3.136	6.251	6.484	5.666	5.325	4.158	0.979	1.966	3.919	7.825	15.349	29.406	31.672	29.192

4.6.3 Overhead

To further investigate the behaviour of the parallel executions, the execution time was analysed with a more fine-grained scale.

Figure 4.7 shows the computation and overhead based on the individual fitness time and chromosome size combinations, where the overhead is intended as the additional time other than the computational one, generally due to communication and message broker (i.e., RabbitMQ) tasks. The stacked bars represent the mean over 100 observations. As it can be seen from the figure, consistently with the other evaluation criteria, from $T_f = 10$ on the computation time starts to cover the majority of the execution time. Here, it is more evident that the chromosome size only influences the execution time from a certain number of nodes on, a threshold that is shifted accordingly to the individual fitness evaluation time growth.

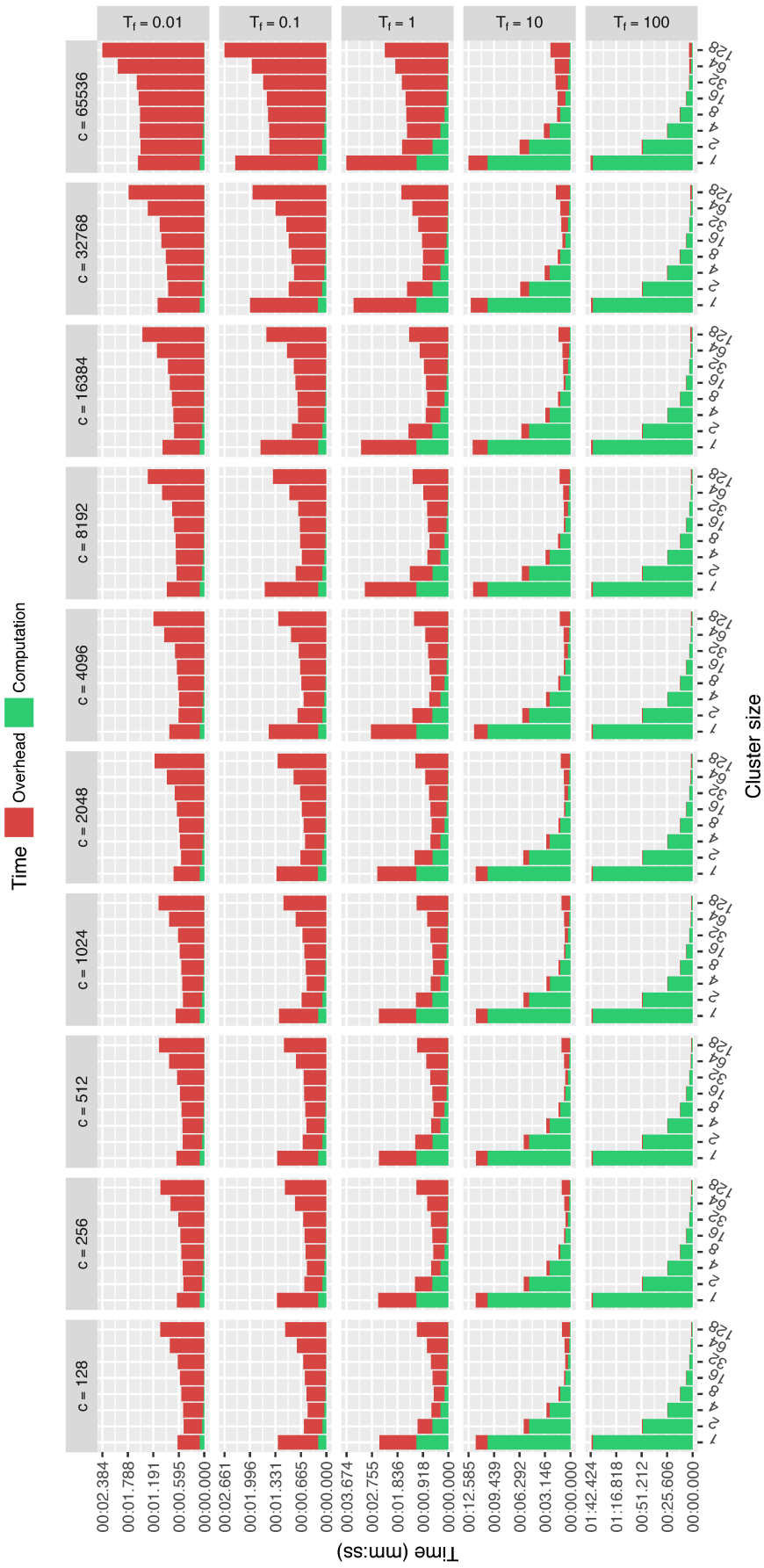


Figure 4.7: The computation and overhead times achieved by the sequential and parallel GAs.

4.6.4 Setup Time

To understand the feasibility of the system in a real world context, the setup time was experimented discriminating into the creation and deployment times. As depicted in Figure 4.8, both the creation and deployment times are proportional to the target number of nodes but in a light way. In the case of the creation time, it is strictly dependent on the specific capability of the cloud provider of instantiating new virtual machines. Also, it is comprehensive of the discovery time to let all the nodes be aware of being part of the same cluster. As for the deployment time, it includes the download of the RabbitMQ and AMQPGA images on all the nodes and the actual scheduling of the containers. The considered times are proportional to different influencing factor that can vary based on the context but, in a general way, they can give an optimistic measure of the setup time that takes only 5 minutes to have the infrastructure ready to start the GAs. Even if with the cloud any consumed time has its cost, this setup time is irrelevant if the GAs is run for many generations and also it is not required to be repeated for any possible future GAs executed on the same cluster.

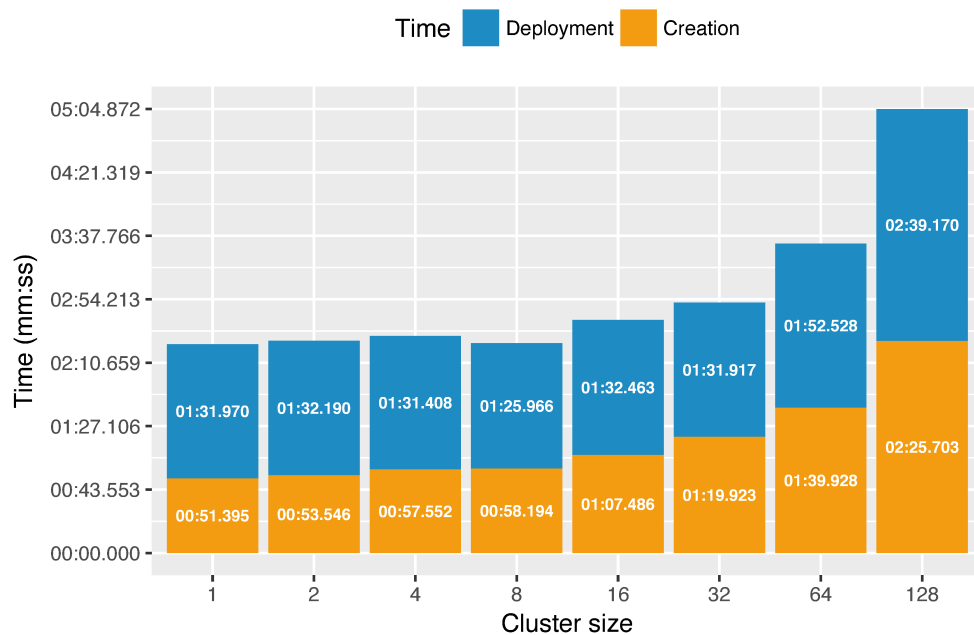


Figure 4.8: The setup times achieved by requesting different cluster sizes.

4.7 Summary

In this chapter, GAs were distributed based on the master/slave model with technologies specifically devised for the cloud, i.e., the software containers, cloud orchestration and message queues. A novel implementation that exploits message queues to schedule parallel GAs tasks was presented. Also the devised system was put in a conceptual workflow for development, deployment and execution activities of distributed GAs. Then, the effectiveness of the system was empirically assessed in terms of execution time, speedup, overhead, using a dummy fitness function as a benchmark problem.

The acceleration of the execution time of the GA application up to a total number of 128 slave nodes was successful. From the results, it emerged that there is a connection between computation load and communication cost. It was also observed that the execution time is directly proportioned to the individual fitness evaluation time and inversely proportioned to the number of cluster nodes. There is an inferior limit for the evaluation time for the fitness function that makes the parallelisation effective. Also, there is also a superior limit regarding the chromosome size that, together with the population size, determines the network load and thus influences the final execution time. Moreover, it was observed that the setup time can be quantified to few minutes even if the request is of many nodes (e.g., 128). It is worth noting that this time is related to a completely automatised activity, which does not require the human presence as in the case of other methodologies, as observed for Hadoop in Chapter 3. The performance and setup times place positively the cloud between other employed technologies for GAs parallelisation, e.g., multi-core systems, GPUs [59, 60] and Hadoop MapReduce.

Chapter 5

Exploring Software Containers for Other Evolutionary Algorithms

Contents

5.1	Introduction	111
5.2	Related Work	115
5.3	System Design	118
5.4	Demonstration	127
5.5	Summary	131

5.1 Introduction

Evolutionary Algorithms (EAs) are metaheuristics for optimisation problems, i.e., high-level problem-independent algorithmic frameworks, based on the concept of populations. They use mechanisms of biological evolution, such as reproduction, mutation, recombination and crossover of individuals. The repetition of these mechanisms creates the concept of ‘evolution’ to improve the fitness of individuals.

There are many EAs in the literature, some of these are:

- Genetic Algorithms (GAs), the most popular of EAs, stressing in particular

the coding of attributes into a set of genes;

- Genetic Programming (GP), consisting in solutions in the form of computer programs, where the aim is that of optimising the ability to solve computational problems;
- Evolution Strategy (ES), mainly focused on the resolution of continuous parameter optimisation problems;
- Differential Evolution (DE), stressing the mutation operator based on vector differences to improve the population.

Moreover, there are some techniques related to EAs:

- Ant Colony Optimisation (ACO), inspired by the behaviour of real ants, solves problems reducing them to finding good paths through graphs;
- Particle Swarm Optimisation (PSO), based on the ideas of animal flocking behaviour.

EAs can be also be specialised to solve specific problems such as ‘machine learning’ problems, giving birth to a specific class of techniques called Evolutionary Machine Learning (EML).

In the previous chapters, GAs were parallelised adapting specific models, i.e., the global, grid, and island ones. To do the same with other EAs using Hadoop MapReduce or software containers, one approach consists in identifying possibly specific parallelisation models, if hopefully already existent in literature, then adapting them to the parallelisation technologies. This was the case of GAs addressed in the previous chapters. Otherwise, it is possible to treat EAs as black boxes, independently executed in parallel nodes, then exchanging data through communication protocols. In this chapter, the parallelisation of EML is investigated using this second way. In particular, the approach of *software containers* and DevOps practices for Parallel Genetic Algorithms (PGAs), used in Chapter 4, are explored to understand if they are effective for the use with EML.

EML techniques are widely employed to build and optimise predictive models. To create accurate models, it is required that their training phase is based on a representative sample of the known data, i.e., the training set. In most of the real world cases, the data volume is considerable as ‘big data’, meaning it is so large that cannot be treated with traditional data management approaches, and using a single computational unit.

For this reason, methodologies that involve the parallelisation of the algorithms for EML, thus reducing the computational load, are useful. The synergistic collaboration of multiple machines, with the aim of producing a final predictive model, first requires creating portions of the original data in a size that is possible to process. With ‘factorization’ of data is possible to split the training data into a certain number of samples, appropriately balanced so that each sample is a reduced version but representative of the whole data. Then, on the basis of these samples, multiple learners can be executed in parallel. They can be completely different EML techniques, implemented in different programming languages. Once the models have been optimised through the use of EAs, it is possible to ‘filter’ them and reject those that do not meet established requirements, based on data withheld from training. At that point, the selected model set is ‘fused’ to produce a final ensemble model, e.g., using the majority voting. The combination of the three operations, i.e., the factorization, filtering and fusion, makes the problem of building a predictive model for a large data implicitly ‘parallelisable’, even combining different learning techniques.

These concepts are part of FCUBE¹ [4], an existing solution that takes advantage of cloud computing toward making disciplined and scalable EML comparison and collaboration more effortless. It introduced the model of ‘*Bring Your Own Learner*’ to allow active collaboration of different EML developers to the same system, with a plug-and-play style interface. The model was devised to face the problem of the research in the field of EML, where new algorithms are always being designed and existent are continuously improved.

¹<https://flexgp.github.io/FCUBE>

The potential learner developers are only required to respect the specifications of the input/output interface, providing the two functions of model building and evaluation. The authors demonstrated FCUBE on a Amazon EC2 cluster of 100 instances, using the *Higgs* dataset with 11 000 000 exemplars, running 5 different learning algorithms. Despite being noteworthy, FCUBE nonetheless is 'locked' into one cloud provider, i.e., *Amazon AWS*, and lacks both automation and robust fault tolerance.

In this chapter, *cCube* (compare, compete, collaborate) is presented, an open source architecture that helps its user develop an application that deploys EML algorithms to the cloud. The source code is shared at the address <https://github.com/ccube-eml>, under the terms of the *MIT License*².

As with FCUBE, *cCube* supports competition and collaboration with filter, factor and fusing. A *cCube* EML application factors data, handles parameter configuration, tasks parallel classifier training with different algorithms, and follows training by filtering and fusing classifier results into a final ensemble model. As such, *cCube* serves 3 types of user: EML algorithm designer, multi-algorithm EML application manager and non-EML literate, i.e., 'black box end users'. *cCube* also supports crowdfunding, i.e., the sharing of costs by a collaborative group that wishes to execute a large multi-learner, factor, filter, and fuse application.

In *cCube* researchers can run different EML algorithms, their own as well as others', developed in different programming languages, without inserting any code into them to accommodate cloud scaling. Instead of being monolithic, *cCube* has a *microservices* architecture [37], i.e., a suite of small services (microservices), each running its own process and communicating with lightweight protocols, e.g., HTTP resource API and message queues. It has one service for each of factorization, scheduling, learning, filtering, and fusion. Each service is independently deployable in a fully automated way making applications easier to scale and more fault tolerant. Collectively *cCube*'s services are minimally centralised and managed by an *orchestrator*. For all of its microservices, *cCube*

²<https://opensource.org/licenses/MIT>

uses lightweight runtime environments in the form of ‘software containers’ (see Section 4.3.1). Moreover, they are designed to make application packaging and execute microservices easily [15]. *cCube* containers can be automatically deployed using *Docker*.

A *cCube* application was developed and its deployment demonstrated on different clouds, utilising free resources, describing its employment on two cloud providers, using them both separately and together.

The chapter is organised as follows. First, a review of the motivations for factoring, filtering and fusion classification and for cloud-scaling and illustrating the relevant related work in Section 5.2. The *cCube* platform is described in Section 5.3. Demonstration is in Section 5.4. Finally, the summary is in Section 5.5.

5.2 Related Work

In the following th most relevant related work concerned with collaborative EML and cloud applications architectures are reviewed.

5.2.1 Collaborative EML

Only one prior project, FCUBE, has addressed the challenge of the EML community collaboratively developing a compendium solution to a noteworthy ‘big data’ classification problem. The project assumes individuals contribute their algorithms, called ‘learners’, each independently written to solve the problem using a smaller sample of the dataset. The software applies a particular general decomposition called ‘factor’, ‘filter’, and ‘fuse’: once contributed learners are collected, it executes them independently and in parallel by *factoring* the entire data and creating splits of the original data into feasible sizes. During training, the classifiers, i.e., models, resulting from all the learners are collected. Then FCUBE executes a step of classifiers filtering and fusion that reduces the collection before creating an ensemble-based solution. This ensemble classifier is the community’s solution to the ‘big data’ problem [4].

FCUBE learners can be completely different EML techniques, implemented in different programming languages. Its 'Bring Your Own Learner' paradigm has a plug-and-play style interface that reduces programming burden on the participants. Algorithm developers are required to respect the specifications of the input/output interface, providing the two functions of classifier training (i.e., model building) and evaluation. *cCube* supports the same *Bring Your Own Learner* paradigm as FCUBE.

5.2.2 Cloud Applications Architectures

Cloud computing exploits a distributed memory resource model. Instead, using a shared memory model, such as GPUs or multi-threaded CPU, requires specific inter-process communication protocols to be embedded within EML software and that force the algorithms to be refactored. Cloud computing removes the inefficiency of owning physical hardware that has been provisioned for infrequent, high workloads and instead offers elastic computation as a service in the form of virtual instances, for whatever time, quantity and quality is required by a specific application.

Significant evolutionary computation work exploits cloud computing while not necessarily solving the explicit collaborative classification challenge. Confining the discussion to those most relevant to *cCube*, besides the aforementioned FCUBE, is one system that uses a synchronous storage service as pool for exchange information among population of solutions [42]. Another, *SPACE* allows the computational resources necessary for running large scale evolutionary experiments to be made available to amateur and professional researchers alike, in a scalable and cost-effective manner, directly from their web browsers [36].

cCube integrates user code into an application that runs on the cloud, in that aspect it is similar to FCUBE. FCUBE runs as a platform on the *Amazon Web Services (AWS)* cloud platform, using the *Amazon Elastic Compute Cloud (EC2)* service, or can equivalently be described as an architecture that creates an application to run on AWS. It has been impressively demonstrated on an EC2 cluster of 100 instances, using the *Higgs* dataset with 11 000 000 exemplars,

running 5 different learning algorithms [4].

However, FCUBE has a number of limitations. For example, FCUBE suffers from a strict dependence on a specific cloud provider, i.e., AWS EC2. Every FCUBE startup process interacts with the Amazon API and its virtual instances are realised and replicated using a technology specifically devised by Amazon, i.e., *Amazon Machine Image (AMI)*. A goal of *cCube* is to provide software independent cloud vendors, e.g., users can take advantage of market prices.

There are also some shortcomings in FCUBE's *Bring Your Own Learner* implementation:

- FCUBE requires manual intervention whenever a new learner is contributed and each new learner triggers a re-build of the FCUBE AMI, since the EML algorithms needs to be explicitly declared;
- should a new EML algorithm execute in a currently unsupported programming language, that learner's execution environment on the virtual instance has to be manually configured. FCUBE's maintenance and extension process is fragile, inflexible and labour intensive;
- FCUBE requires manual intervention in the fusion step when outputs of models are collected;
- the current design does not guarantee broad scalability and true fault-tolerance for every component, since the functionalities are not clearly distinguished. The communication protocols are not reliable, e.g., FCUBE uses *Amazon S3* as a file-based interchange point and relies upon SSH commands.

How *cCube*'s microservices design addresses the shortcomings of FCUBE are elaborated in later sections. In general *cCube* offers enhanced automation, robustness, support for integrated development practices and is independent of a cloud provider.

Cloud computing offers a broad spectrum of technologies from which to compose an evolutionary algorithm or system. Merelo Guervós et al. devised

SofEA, a model for pool-based evolutionary algorithms in the cloud, an evolutionary algorithm mapped to a central *CouchDB* object store [41]. *SofEA* provides an asynchronous and distributed system for individuals' evaluations and genetic operators application. Later, they defined and implemented the *EvoSpace Model* [23], consisting of two main components: a repository storing the evolving population and some remote workers, which execute the actual evolutionary process. It is the first work to involve technologies on the *Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)* level: *Heroku* as PaaS for the population store and *PiCloud* as SaaS for the computing operations. *cCube* takes advantage of *Docker*, *PostgreSQL*, *RabbitMQ*, and *MongoDB*.

Next, in Section 5.3 *cCube* is described and its design choices motivated, which yield cloud and development practices that are principled, systematic and robust.

5.3 System Design

The aim of *cCube* is to facilitate EML comparison, competition and collaboration. A use case that *cCube* handles is factoring, filtering and fusing. There are three possible users of *cCube*:

1. EML researcher, using best practices for software methodology;
2. end user, comparing EML algorithms to gain insight, selecting competing EML algorithms for best performance and collaborating with other end users as a community;
3. *cCube* engineer, that administers, maintains and intervenes manually for the platform.

To build a *cCube* application, the EML developers copy a template from *cCube's* repository and customize configuration, e.g., EML algorithm invocation. The developers then become end users and start the *cCube* client on their machine, provide keys for authorization, thus keeping their sensitive information local and secure. The client, through *Docker*, starts the application after

provisioning resources, running resource discovery and set up. The engineer of *cCube* itself expands and maintains the open source code.

EML Researcher EML developers often use a personal computer to construct an EML algorithm that is accurate and fast enough on a subsample of the available data set. *cCube* is designed to help them integrate best practices into their software development processes. It employs the conceptual work flow shown in Section 4.4.3, that is adapted in Figure 5.1.

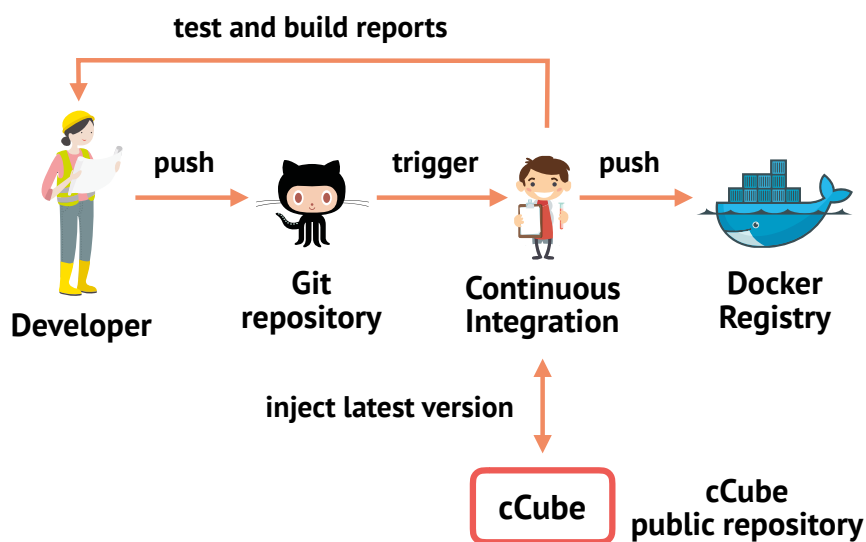


Figure 5.1: *cCube* is consistent with software development best practices, e.g., Continuous Integration.

By practicing *Continuous Integration* [20], the learner developers can maintain their source code in a single repository, and access to automated testing, building and deployment processes. Then, after creating *cCube* as an extension, the developer can execute the learner in the cloud. Using Docker, the development capability was extended to allow the developer to include source code and/or to define the algorithm's execution environment, i.e., every component required for learner execution in any programming language or technology, without requiring a manual development intervention. The interaction interface is kept flexible by defining a wrapping interface. Therefore, the only information required is the path and instruction on how to execute the two phases of EML computation, 1. learning and prediction 2. filtering and fusing

the trained models output path . The developer does not need to make the source code aware of *cCube*'s functionality or parallel computation. Therefore, *any* algorithm can be executed in *cCube*. Once the container is defined, the developer only needs to build and distribute it on a *Docker Registry* repository, to be downloaded, executed and replicated on demand.

***cCube* End User** The end user is interested in executing large-scale EML algorithms, and therefore treats *cCube* as a black box. *cCube* does provisioning and distribution of computational units using cloud accounts. The end user view of *cCube* is shown in Figure 5.2. As it is possible to observe:

- the end user provides configurations for *cCube* tasks, EML algorithms, data set and compute duration;
- the *cCube* client submits requests to cloud providers and a cluster of the required number of nodes is allocated;
- *cCube* is ready to orchestrate containers and start the EML algorithms for factoring, fusion and filtering;
- when a job is submitted a *cCube* cluster pulls the Docker images for the services from the Docker registry and enqueues the tasks for the services.

In another scenario, *cCube* supports end users who collaborate by each contributing some machines they have commissioned from their own account on their cloud provider, in a sort of crowdfunding model. In this case, *cCube* allows each user to keep their cloud credentials private, since the invocation of cloud instances happens local to each of them, i.e., on their own computer. This enables *cCube* to execute securely. The user can leave the cluster at any time.

***cCube* Engineer** The role of the *cCube* engineer is to extend the capabilities for comparison, competition and collaboration provided by *cCube* as well as maintain and administer the architecture. Publicly available source code and licensing are essential for these responsibilities as are minimal manual or labor intensive steps when handling administration and security tasks. A *cCube*

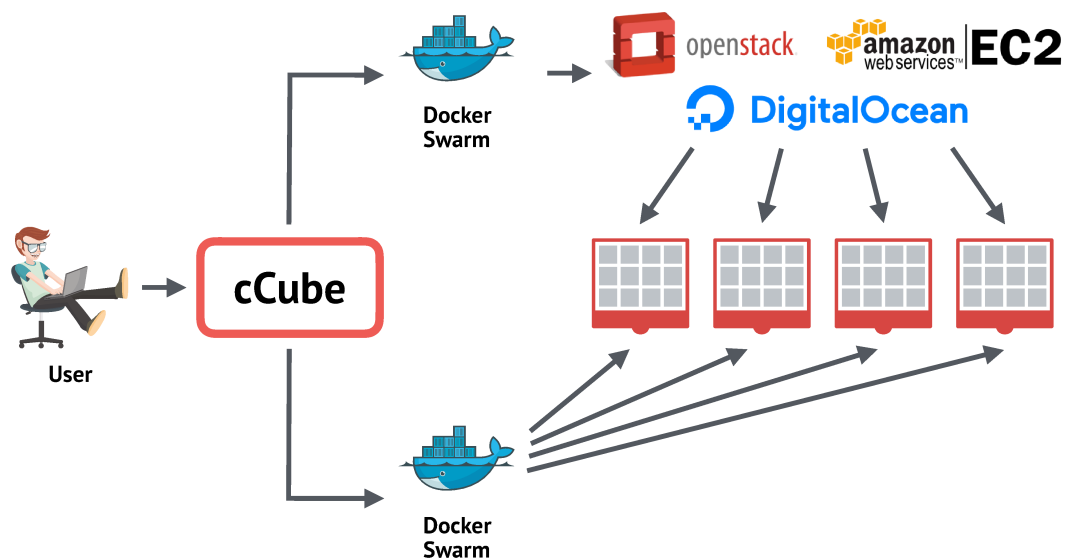


Figure 5.2: *cCube* for the end user perspective, a convenient black box that performs EML at scale using Docker across multiple clouds for compute.

engineer last but not least requires, e.g., expertly designed interfaces and loosely coupled code. For these reason, a microservices design approach was taken in consideration for the *cCube* architecture.

5.3.1 *cCube* Architecture

The demands of the EML researcher, end user and *cCube* engineer are met with microservices and Docker containers with event queues and REST communication [56]. The design goals for *cCube* have been identified as:

- *portability*, in terms of cross-platform and cloud execution;
- *extendability* with open source code;
- *scalability* in distributing the computation;
- *robustness*, i.e., tolerant of failures;
- *maintainability*, e.g., loosely coupled components and explicit assumptions;
- *deployability*, exploiting standard interfaces for deployment;

- *runability*, run anywhere without recompilation as in plug-and-play insertion;
- *support* best practices in development, maintenance and performance.

Instead of being monolithic, *cCube* architecture is based on *microservices*. For each functional component of *cCube*, a single microservice was identified and implemented. These components could be individually developed, using different programming languages (though they were entirely written in Python) and technologies, and the interaction between them occurs through simple communication protocols, i.e., REST API with HTTP and message queues. This allowed *cCube* to have separate provisioning, distribution, communication and EML system execution. An overview of *cCube* is given in Figure 5.3.

For the EML development, microservices and the software containers approach, the *Bring Your Own Learner* model of FCUBE [4] was enhanced. Instead of inserting the learner within the source code and environment of *cCube*, *cCube* was injected inside the container of the EML algorithm, running it as daemon instructed to manage the communication with the other microservices of the system.

5.3.2 *cCube* Implementation

Also, the open source properties of *cCube* were stressed in order to make the microservices fully accessible to the community, e.g., others could learn and extend *cCube*'s code to develop other EML architectures. To facilitate the deployment, as well as development, the traditional use of the hypervisor-based virtualization was abandoned in favor of the container-based one using Docker (see Section 4.3.2 for further details). Thanks to the Docker API, it was relatively easy to develop containers that themselves were easy to build, share and quickly execute in a cloud environment. Moreover, once an image of the container is ready, thus completed the build process, it can be stored into a convenient public registry that Docker provides, i.e., *Docker Hub*.

cCube achieves independence from cloud providers by employing *Docker*

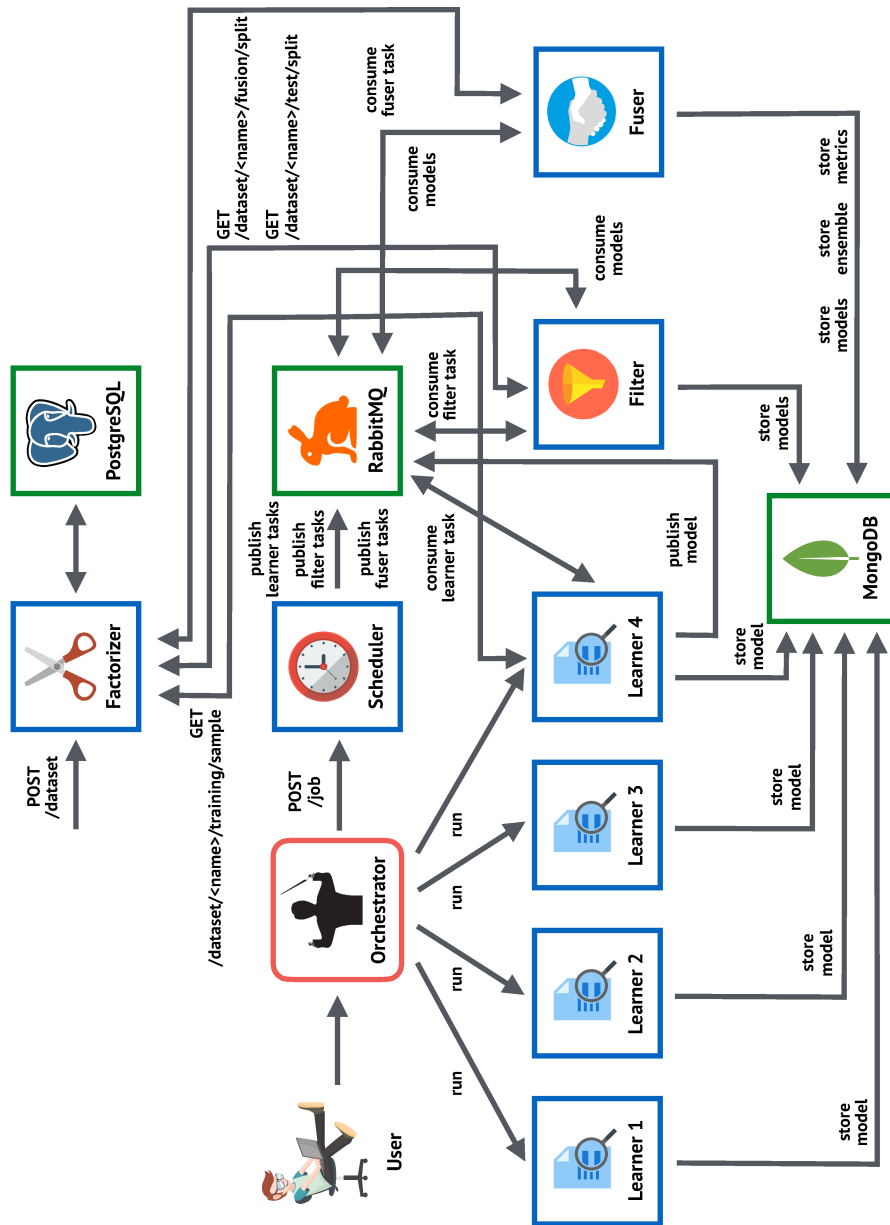


Figure 5.3: Under the hood of the *cCube* black box there is a microservices and containers architecture using REST API and message queues for extensible, automated, reliable and scalable EML.

Swarm. In this work, Docker Swarm was employed instead of CoreOS (see Section 4.3.3) since Swarm was made publicly available at the time of the demonstration. This is a native technology that composes a cluster of Docker platforms between machines running the Docker engine. In addition, by using Docker, *cCube* was implicitly made flexible against the limitations of the quantity of machines the providers usually impose upon their users, through an infrastructure definable as ‘multi-cloud’, i.e., based on the allocation of instances by different cloud providers but participating in the same system [19].

Microservices

An Orchestrator creates and provisions the compute units, initiating and directing the symphony of microservices. It uses a bridge pattern [55] to interface with different cloud providers, e.g., *OpenStack* and *Amazon EC2*.

The Learner, Filter and Fuser microservices were designed as part of one single container, i.e., the Worker, thus stored by the developer as a single image on a Docker registry. The EML developer needs to inject the *cCube* Worker component into the target EML algorithm execution environment, i.e., a Docker container. This component will act as daemon inside the container and be in charge of communication with the *cCube* cluster. Thus, it will execute the EML algorithm for learning or prediction, using environment variables defined by the EML developer. By the means of a parameter given during the orchestration, the container is able to detect which ‘role’ to play.

The following microservices, whose overview is shown in Figure 5.3, were designed and implemented:

Factorizer: uses the bridge pattern [55] to interface with the storage. The storage is currently *PostgreSQL*³, but it is possible to add other technologies by means of the definition of other driver classes. The Factorizer exposes its services through a REST interface thus the communication happens using HTTP. This allows the data set upload via a `POST /dataset` request and then the data set can be split into separate parts on demand and following the

³<https://www.postgresql.org>

components needs, e.g., to use for training, fusion, and testing.

Scheduler: it accepts jobs through a REST interface. A job is considered as a chain of processes executed on a *cCube* cluster, involving all the components, i.e., microservices. Once a new job is requested, the Scheduler creates the tasks for the Learners, Filter and Fuser and publish them on different message queues in JSON format. First of all, a task represents a placeholder to let other microservices carry out their duties by consuming them. Exploiting a convenient feature of Advanced Message Queueing Protocol (AMQP), in case of a microservice failed, e.g., cloud provider hardware fault, the task would be put again into the queue and that computation run again by another container. If the EML algorithm concludes the computation, then the task is acknowledged, i.e., definitively removed from the queue. However, in the case the fault is due to an EML algorithm failure, *cCube* avoids the repetition of the same task since, with the same configuration, it would fail again. Thus, the daemon recognises it, acknowledges the task in the queue and sends just a failure placeholder message to the Filter. Moreover, the tasks contain all the relevant information needed for running an activity, e.g., the target dataset name, the parameters of separation for training, fusion and testing, features to include/exclude, duration. In the case of the Learners, the tasks would be in the number equal to the degree of parallelization expected by the user. A producer/consumer pattern is controlled by a 'queue manager', implemented by using the *RabbitMQ* (see Section 4.3.4) message broker service. It is worth noting that multiple EML algorithms can participate in the same job and, also, multiple jobs can run on the same cluster.

Learner: each of the multiple learners consumes a task, training data sample and parameters, which are possibly generated at random when the task is generated by the Scheduler, allowing also a parameters factorization. First, a data sample for training is given by the Factorizer, using the GET /dataset/<name>/training/sample request. Once received the data, the Learner executes the EML algorithm according to a given duration, i.e., one of the parameters included in the task. Then, the computed model is compressed and

stored in a message and sent to a queue for outputs. Moreover, a copy of each produced model is stored into a *MongoDB*⁴ database.

Filter: EML algorithm output is filtered according to some ‘filter policy’. When the Filter task has been consumed, it knows exactly the number of expected outputs. Therefore, it consumes outputs until the last expected message arrives. Then, the models are executed on a split of the data that has been set aside, as a result of the GET /dataset/<name>/fusion request, according to a filter policy. It is worth noting that the split is named ‘fusion’ since it is also used during the the fusion phase to produce the ensemble. In the demonstration, the EML algorithm predictive performance is compared to a majority class classifier and if it is above some threshold the model is accepted. After the set of filtered models is done, it is published as a single message to another queue for fusing. The result of the filtering phase is also stored into a MongoDB database.

Fuser: filtered models are eventually fused together to collaborate in an ensemble model. A fusion task is consumed and the message provides the needed information. Data for fusion comes from the Factorizer through a GET /dataset/<name>/fusion request, i.e., the same split used during the filter phase. Then, another data split, as a result of the GET /dataset/<name>/test request, is used to test the ensemble and computing some predictive performance metrics. The ‘fusion policy’ may require the models to be executed both for the fusion and testing phases. In the demonstration, an ensemble based on majority of votes was composed, thus the models were executed only for the test phase. Finally, the ensemble, computed metrics and all the models are stored using the MongoDB service.

***cCube* Injection**

As mentioned above, the EML researchers were allowed to inject the *cCube* code in their containers. This piece of code is maintained by the *cCube* engineers and publicly distributed in the form of a executable script, that could be

⁴<https://www.mongodb.com>

for instance make downloadable from a commodity URL (i.e., `https://raw.githubusercontent.com/ccube-eml/worker/master/install.sh`).

The EML developer is asked to use a *cCube* configuration file that, in the current version, corresponds to a *Dockerfile*, i.e., a Docker container environment definition file. The user is allowed to define the environment using all the features given by Docker, e.g., the inheritance from other public container images already providing the set of tools required for the execution. The only requirement for the connection with *cCube* is the definition of some predefined environment variables, e.g., `$CCUBE_LEARN_COMMAND`, `$CCUBE_PREDICT_COMMAND`, and the inclusion of few download and execution lines, e.g., `ENTRYPOINT ["/ccube"]`.

In the following demonstration, the *GP Function* EML algorithm executable [4] was encapsulated into a container without changing the source code. The environment variables define how the *cCube* daemon can interact with the EML algorithm and its outcome for both the learning and prediction phases.

5.4 Demonstration

In this section how to use *cCube* to run the *GP Function* EML algorithm is demonstrated with a split of the *Higgs* dataset [5]. A private *OpenStack* cloud was used and then a commercial provider, i.e., *Amazon EC2*. It is worth noting that only instances eligible for the *Amazon AWS Free Tier* were used, thus all the computation was free.

5.4.1 Environment Preparation

First, the current implementation of *GP Function* needed to be wrapped in a *Dockerfile* showed in Listing 5.1. Even though not being the original developers, it was enough to know only a few input parameters for *GP Function* to run it in *cCube*:

- where to find the data used for learning;

- where to find the algorithm output, i.e., the models;
- how to execute the model to predict data;
- where to collect the ensemble model.

Listing 5.1: The `.ccube` file for the GP Function algorithm.

```
1 FROM openjdk
2 LABEL maintainer "John Doe john.doe@ccube-eml"
3
4 # cCube injection.
5 RUN curl -sSL https://raw.githubusercontent.com/ccube-eml/worker/
   ↪ master/install.sh | sh
6 WORKDIR /ccube
7 ENTRYPOINT ["python3", "-m", "worker"]
8
9 # Environment preparation.
10 COPY gpfunction.jar /gpfunction/gpfunction.jar
11
12 # cCube configuration.
13 ENV CCUBE_LEARN_COMMAND "java -jar /gpfunction/gpfunction.jar -train
   ↪ \${CCUBE_LEARN_DATASET_FILE} -minutes \${
   ↪ CCUBE_LEARN_DURATION_MINUTES} -properties \${
   ↪ CCUBE_LEARN_PARAMETERS_PROPERTIES_FILE}"
14 ENV CCUBE_LEARN_WORKING_DIRECTORY "/gpfunction"
15 ENV CCUBE_LEARN_OUTPUT_FILES "\${CCUBE_LEARN_WORKING_DIRECTORY}/
   ↪ mostAccurate.txt"
16 ENV CCUBE_PREDICT_COMMAND "java -jar /gpfunction/gpfunction.jar -
   ↪ predict \${CCUBE_PREDICT_DATASET_FILE} -model \${
   ↪ CCUBE_PREDICT_INPUT_FILES}/mostAccurate.txt -o predictions.csv
   ↪ "
17 ENV CCUBE_PREDICT_WORKING_DIRECTORY "/gpfunction"
18 ENV CCUBE_PREDICT_PREDICTIONS_FILE "\${
   ↪ CCUBE_PREDICT_WORKING_DIRECTORY}/predictions.csv"
```

The first section in Listing 5.1 is used to prepare and set up the environment for the execution. With help of the inheritance capacity of Docker, it was possible to start with an environment that had Java already installed. This was made on line (1) by picking the `openjdk` container that is publicly available on *Docker Hub*, i.e., the official and public Docker registry. Lines (5–7) were used to download the *cCube* code (5), install the dependencies, and define the starting point for the container (6–7), i.e., the *cCube* daemon. With line (10) we copied the GP Function JAR executable into the environment. The interface between the EML developer and the *cCube* system are on lines (13–18). The

environment variables were defined, and gave only the relevant information *cCube* would take advantage of during the execution. Moreover, *cCube* provides some predefined environment variables that it will fill in at run time, e.g., the `$$CCUBE_LEARN_DATASET_FILE` will contain the dynamic path *cCube* assigns to the training data samples.

```

1. ~ -- fish (fish)
> orchestrator node create --number 64
--provider openstack
--configuration openstack-configuration.yml

~
> orchestrator cluster init ccube-jah7d6sa-00
--provider openstack
--configuration openstack-configuration.yml

~
> orchestrator cluster join ccube-jah7d6sa-01
--manager ccube-jah7d6sa-00
--token SWMTKN-1-7dh736iwdlfsjdk
--provider openstack
--configuration openstack-configuration.yml

~
> orchestrator service create gpfunction
--replicas 64 --manager ccube-jah7d6sa-00
--provider openstack
--configuration openstack-configuration.yml

2. ~ -- fish (fish)
> orchestrator node create --number 64
--provider amazon
--configuration amazon-configuration.yml

~
> orchestrator cluster init ccube-fc7dc0mh-00
--provider amazon
--configuration amazon-configuration.yml

~
> orchestrator cluster join ccube-fc7dc0mh-01
--manager ccube-fc7dc0mh-00
--token SWMTKN-1-4usrLxwltxlyybu3
--provider amazon
--configuration amazon-configuration.yml

~
> orchestrator service create gpfunction
--replicas 64 --manager ccube-fc7dc0mh-00
--provider amazon
--configuration amazon-configuration.yml

```

(a) OpenStack

(b) Amazon

Figure 5.4: The terminal execution of the orchestrator for the OpenStack and Amazon providers.

Finally, the Docker build process was started and pushed the resulting image to the public Docker Registry.

5.4.2 Cloud Swarm Setup

The cluster for *cCube* was composed in three steps:

1. run it on an OpenStack private cloud and created a cluster with 64 nodes using the *cCube* orchestrator;
2. compose a separate cluster on Amazon EC2 with 64 other nodes on which the system was tried again;
3. finally, just disassemble the two clusters and joined them into a single hybrid cluster with a total of 128 nodes.

Only small and cheap instance types were launched:

1. Amazon EC2, 't2.micro' with 1 CPU and 1 GB RAM;
2. OpenStack private cloud, instances with 1 CPU and 1 GB RAM.

To do this, the *cCube* Orchestrator was extended to interface with both the cloud providers. The *cCube* Orchestrator runs a number of threads equal to the cluster size, in order to ask for new allocations in parallel. To avoid stressing the API interface of the providers the start of the threads was delayed of 1 s each.

Two different times to measure were identified:

- 'creation', i.e., the necessary time to acquire the virtual machine and be able to communicate with it through an SSH connection;
- 'provisioning', i.e., the time required to install Docker on the machine and add it to the Docker Swarm.

Figure 5.4 shows the commands that were executed in the terminal to run the Orchestrator. Most of the output is intentionally hidden, since it was mostly debugging logs. It is worth noting that, even if the cloud providers are different, the commands are quite similar. Given the cloud provider and a configuration file containing the account credentials, the *cCube* Orchestrator is able to:

1. create the 64 nodes on the provider, and provision them with the latest version of Docker (`node create`);
2. initialise the cluster, i.e., the Docker Swarm, by electing one of the nodes (e.g., `ccube-jah7d6sa-00`) as a manager (`cluster init`);
3. let all other nodes join the cluster as workers, by using the secret token given by the previous command and the name of the manager (`cluster join`).

From this point, the infrastructure is ready to run any Docker service, as *cCube* in our case. The setup time was measured, by running each of command

10 times, getting 64 nodes in 15 min on average for OpenStack and 9 min for Amazon. It took only a couple of minutes to provision the two clusters and let the nodes join a single Docker Swarm.

5.4.3 *cCube* Execution

Once the cluster was ready, *cCube* was executed with the GP Function classifier on a split of the Higgs dataset [5]. This was done by running the `service create` command for each of the microservices. The template of the command is shown in Figure 5.4 where the execution of the Learner microservices is specified. By specifying the `-replicas` parameter, a single command could run it in parallel on 64 Learners.

Overview		Messages		
Name	Features	Ready	Unacked	Total
gpfunction@filter.outputs	D	0	0	0
gpfunction@filter.tasks	D	0	1	1
gpfunction@fuser.tasks	D	0	1	1
gpfunction@learner.outputs	D	14	0	14
gpfunction@learner.tasks	D	0	50	50

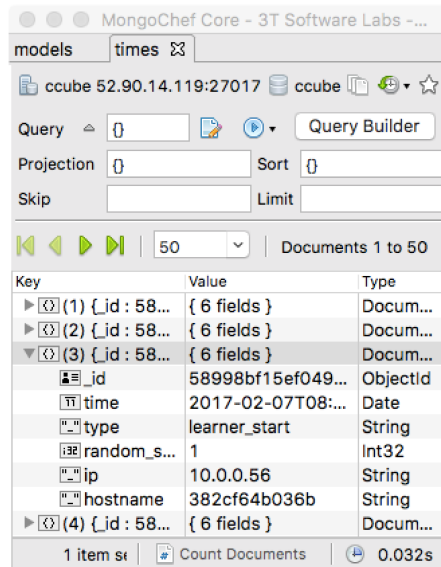
Figure 5.5: The message queues status during the *cCube* execution.

Figure 5.5 shows a screenshot of the RabbitMQ administration web page displaying the status of *cCube* messages at a point during the computation. During that moment, 14 Learners had completed their work and the other 50 Learners were still running. In the meantime, the Fuser was waiting for all the 64 Learner outputs.

Finally, Figure 5.6 shows the output from the Fuser that was collected in the MongoDB database, storing both the models and the times.

5.5 Summary

In this chapter, the approach of *software containers* and DevOps practices for PGAs were explored to understand if they are effective for the use with other



Key	Value	Type
▶ (1) { _id : 58...	{ 6 fields }	Docum...
▶ (2) { _id : 58...	{ 6 fields }	Docum...
▼ (3) { _id : 58...	{ 6 fields }	Docum...
_id	58998bf15ef049...	ObjectId
time	2017-02-07T08:...	Date
type	learner_start	String
random_s...	1	Int32
ip	10.0.0.56	String
hostname	382cf64b036b	String
▶ (4) { _id : 58...	{ 6 fields }	Docum...

Figure 5.6: The output of the *cCube* execution on the MongoDB database.

EAs than GAs. In particular, *cCube* was presented, an open source architecture for the comparison, competition and multi-party collaboration of EML algorithms and users. It was used as a first attempt. The goal of *cCube* is to provide an instrument for EML developers to run their algorithms on a large scale using parallel cloud machines without changing code. By using the ‘Bring Your Own Learner’ model, the collaboration of different EML algorithms was made possible, to learn on the same dataset and combining the results into an ensemble after filtering and fusion.

A microservices architecture simplifies the development and the maintenance processes, and facilitates extension from the open source community. Moreover, the microservices were implemented in the form of software containers using Docker and the execution of *cCube* demonstrated. The main contributions are:

- let EML developers define the environment for the execution of their algorithms and easily inject and run *cCube*'s code;
- package the microservices in distributable images and make them available through a public repository;
- using the capability of Docker Swarm, a flexible infrastructure was real-

ised, avoiding the ‘lock’ in from specific cloud providers and possibly runnable on multi-cloud;

- allow users collaboration, in the form of crowdfunding.

Currently, the proposed architecture allows to fuse models resulting from different learners only if the EML algorithms are available on the same container. For instance, the GP Function in Java can fuse with another Genetic Programming learner written in Python. Therefore, the use of specific algorithms is possible by specifying it using a task parameter. It is in the future agenda to enhance the architecture to better support multiple EML algorithms running and their models fusion, adding a prior ‘executor’ microservice whose purpose is to free the Fuser from the strict bond with the execution environments.

Also, future work will involve extending *cCube* further for comparison and competition, as well as testing its usability, testing the large scale capacity over more cloud providers and more nodes on big data challenges.

Considering the current implementation, the approach of software containers to run parallel EA as black boxes demonstrated to be already effective.

Chapter 6

Conclusions and Future Work

Contents

6.1 Thesis Summary	135
6.2 PGAs Using Hadoop MapReduce	136
6.3 PGAs Using Software Containers	138
6.4 Towards the Parallelisation of other EAs in the Cloud	139

6.1 Thesis Summary

This thesis studied the problem of parallelising Genetic Algorithms (GAs) in cloud environments, using different approaches.

Hadoop MapReduce has been demonstrated to be effectively employable as a parallelisation platform for GAs, especially when the problem fitness function requires high computational load. From the results of the study emerged that the island model, in particular, gives the best performance in the majority of the cases. Thanks to the use of the elephant56 framework, which is described in this thesis, it is also possible to reduce the effort in developing Parallel Genetic Algorithms (PGAs), letting the end user focus only on the genetic operators implementation.

Nevertheless, Hadoop MapReduce results to be particularly suitable when a cluster of machines is already available. Indeed, even if using commercial

cloud machines is possible, it requires specific skills for setup and maintenance activities. Instead, using specific cloud technologies, as the ones employed in the second work of this thesis, may simplify the setup of the parallel workers, the deployment and execution of PGAs. From the results emerged that software containers and message queues can be employed effectively also for a model, i.e., the global model, where Hadoop MapReduce showed low performance.

The software containers, together with the other technologies and approaches employed for GAs parallelisation, demonstrated to be suitable also when used for other Evolutionary Algorithms (EAs). In particular, they allowed to design and implement *cCube*, an open source architecture that helps its users to develop an application that deploys Evolutionary Machine Learning (EML) algorithms to the cloud.

In this chapter final detailed remarks and future work are provided for every aspect addressed in the thesis.

6.2 PGAs Using Hadoop MapReduce

In Chapter 3, the parallelisation of GAs was addressed using the Hadoop MapReduce platform, based on three models, i.e., the global, grid and island models. As a benchmark problem, a software engineering problem of configuring the Support Vector Machines (SVMs) for inter-release fault prediction was considered. The effectiveness of these models in terms of execution time, speedup, overhead and computational effort was empirically assessed by using three publicly available datasets, which were chosen considering their different size in order to varying the execution time of the GAs. It emerged that the use of PGA based on the island model outperforms the use of sequential GA and the PGAs based on the global and grid models for all the considered datasets and cluster configurations since it is able to reduce the number of operations performed on the data store, determining a faster execution of tasks and an optimised usage of resources. Moreover, the results of the estimation of the commercial cloud providers costs revealed that the island model is worth using

also in term of costs against the execution with a single machine.

In general, a critical aspect in the use of Hadoop MapReduce is the presence of overhead due to the communication with the data store (i.e., Hadoop Distributed File System (HDFS)). The distributed nature of the data store introduces an intrinsic communication latency that may drastically worsen the performance if multiple and useless operations are executed. To speed up the execution of tasks, it is useful to reduce data store operations, as it happens with the island model where data store access is limited to the migration phase only. One avenue for future work is to evaluate the improvements in performance when tuning the island model specific configuration parameters, such as the migration intervals [40], using Hadoop MapReduce. Also, aiming at comparing the achieved results with the theoretical models for GAs parallelisation proposed by the literature [7, 35], adapting them to consider the effects of Hadoop MapReduce.

Regarding the global model, one way to improve the execution time could be the reduction of the data transmitted during the distribution of computational load. For instance, by providing the driver of an individuals registration capability, once the fitness evaluation had completed, the output could be reduced to the fitness values only instead of transferring the complete chromosomes. Unfortunately, the serial nature of Hadoop MapReduce requires the use of synchronisation barrier, i.e., waiting for other parallel work completion, thus not allowing the global model to take full advantage of Hadoop MapReduce except for the case of intensive fitness evaluation work. As for the grid model, a synchronisation barrier was placed in the driver, as suggested by Di Martino et al. [14]. A substantial simplification of the communication could be applied by making the execution of neighbourhoods evolution entirely independent throughout all the generations. This option was not exploited because the main interest was in providing a flexible solution, allowing to define and use different strategies for the resolution of GAs in the form of a framework. It is on the future agenda to implement and study these improvements, also for the global and grid models.

6.3 PGAs Using Software Containers

In Chapter 4, GAs were distributed based on the master/slave model with technologies specifically devised for the cloud, i.e., the software containers, cloud orchestration and message queues. Also the devised system was put in a conceptual workflow for development, deployment and execution activities of distributed GAs. The effectiveness of the system was empirically assessed in terms of execution time, speedup, overhead, using a dummy fitness function as a benchmark problem.

The execution time of the GA application was accelerated up to a total number of 128 slave nodes. It emerged that there is a connection between computation load and communication cost and that the execution time is directly proportioned to the individual fitness evaluation time and inversely proportioned to the number of cluster nodes. Moreover, it was observed that the setup time can be quantified to few minutes even if the request is of many nodes (e.g., 128). The performance and setup times place positively the cloud between other employed technologies for GAs parallelisation, e.g., multi-core systems, GPUs [59, 60] and Hadoop MapReduce, especially considering that these times are related to a completely automatised activity, which does not require the human presence as in the case of other methodologies.

One avenue for future work is to evaluate other parallel models for GAs such as the cellular and island model [38]. The empirical study should be replicated with other fitness functions, and it is planned to put into operation the structure by solving real world optimisation problems, such as Test Suite generation [12, 14]. Furthermore, to make the system more flexible and easy to use, it is also planned to abstract the concepts further and propose it in the form of a framework, similarly to the work proposed in Chapter 5. In this way, the developer would have to deal exclusively with the activity of genetic operators programming. Also, it is in future plan to study the effects of the population size parameter tuning in presence of heterogeneous environments [2, 22], e.g., the cloud, and using other parallelisation models, e.g., Pool-based EAs [23], to

reduce the need of synchronisation but hopefully improve the performance.

6.4 Towards the Parallelisation of other EAs in the Cloud

The main aim of Chapter 5 was to explore the approaches of software containers, described in Chapter 4, for the use with other EAs than GAs, such as Genetic Programming (GP), Evolution Strategy (ES), Differential Evolution (DE) and some related techniques, e.g., Ant Colony Optimisation (ACO), Particle Swarm Optimisation (PSO). There are mainly two ways to parallelise EAs:

1. identify specific parallelisation models and adapt them to target technologies;
2. using EAs as black boxes, independently executed in parallel, exchanging results through communication protocols.

The former is the approach addressed for GAs in Chapter 3 and Chapter 4. In this case, the parallel models are considered as distinct EAs themselves since they alter the normal execution given by the sequential version, e.g., the grid and island models. Therefore, the first challenge is to identify a possible parallel model for the target EA and devise it if not already present in the literature. Then, it is required to adapt the identified model to specific technologies, e.g., the MapReduce paradigm. For instance, even if not strictly related to the field of EAs, the parallelisation of Tabu Search (TS) was addressed during the period of work on this thesis. TS is a search algorithm (see Section 2.2.1) that enhances the performance of local search prohibiting, for this reason the term 'tabu', the search to previously visited solutions. Software containers were used to implement a novel parallel version of TS. Parallel nodes run different TS algorithms, exchanging information through a central database service, avoiding to re-calculate fitness, i.e., one of the most time consuming operators [38], and undertake areas of previously explored solutions space.

The second possible approach treats EAs as programs that runs in parallel. The main effort consists in finding a way to coordinate the exchange of and treat the outcomes. Chapter 5 explored this approach for a class of EAs, solving machine learning classification problems in particular, i.e., EML. *cCube* was presented as an open source architecture for the comparison, competition and multi-party collaboration of EML algorithms and users. The goal of *cCube* is to provide an instrument for EML developers to run their algorithms on a large scale using parallel cloud machines without changing code. By using the 'Bring Your Own Learner' model, the collaboration of different EML algorithms was made possible, to learn on the same dataset and combining the results into an ensemble after filtering and fusion. A microservices architecture, implemented in the form of software containers using Docker, simplified the development and the maintenance processes, and facilitates extension from the open source community. Moreover, the *cCube* was demonstrated on different clouds.

The positive results from these preliminary explorations allowed planning the adaptation of cloud technologies for other EAs, using both the possible approaches, i.e., devise and adapt specific parallel models and black boxes. Moreover, distributed and sequential EAs will be compared in terms of quality and execution time, and possibly mixed as an ensemble of algorithms working together to solve the same problem, in the same way of *cCube*.

Acronyms

AMQP Advanced Message Queueing Protocol.

EA Evolutionary Algorithm.

EML Evolutionary Machine Learning.

GA Genetic Algorithm.

GP Genetic Programming.

HDFS Hadoop Distributed File System.

JVM Java Virtual Machine.

PGA Parallel Genetic Algorithm.

SGA Sequential Genetic Algorithm.

SVM Support Vector Machine.

YARN Yet Another Resource Negotiator.

Bibliography

- [1] Wasif Afzal and Richard Torkar. On the Application of Genetic Programming for Software Engineering Predictive Modeling: A Systematic Review. *Expert Systems with Applications*, 38(9):11984–11997, September 2011.
- [2] Enrique Alba, Antonio J. Nebro, and José M. Troya. Heterogeneous Computing and Parallel Genetic Algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362–1385, September 2002.
- [3] Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.
- [4] Ignacio Arnaldo, Kalyan Veeramachaneni, Andrew Song, and Una-May O’Reilly. Bring Your Own Learner: A Cloud-Based, Data-Parallel Commons for Machine Learning. *IEEE Computational Intelligence Magazine*, 10(1):20–32, February 2015.
- [5] P. Baldi, P. Sadowski, and D. Whiteson. Searching for Exotic Particles in High-Energy Physics with Deep Learning. *Nature Communications*, 5, February 2014.
- [6] J. Mark Baldwin. A New Factor in Evolution. *The American Naturalist*, 30(354):441–451, 1896.
- [7] Erick Cantú-Paz and David E. Goldberg. On the Scalability of Parallel Genetic Algorithms. *Evolutionary Computation*, 7(4):429–449, 1999.

- [8] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [9] William J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, 3 edition, 1999.
- [10] Charles Darwin. *On the Origin of Species by Means of Natural Selection*. Murray, London, 1859.
- [11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [12] Linda Di Geronimo, Filomena Ferrucci, Alfonso Murolo, and Federica Sarro. A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 785–793, 2012.
- [13] Sergio Di Martino, Filomena Ferrucci, Carmine Gravino, and Federica Sarro. A Genetic Algorithm to Configure Support Vector Machines for Predicting Fault-Prone Components. In *International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 247–261, 2011.
- [14] Sergio Di Martino, Filomena Ferrucci, Valerio Maggio, and Federica Sarro. Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, pages 113–135. IGI Global, 2013.
- [15] Erhan Ekici. Microservices Architecture, Containers and Docker, December 2014. URL: https://www.ibm.com/developerworks/community/blogs/1ba56fe3-efad-432f-a1ab-58ba3910b073/entry/microservices_architecture_containers_and_docker.

-
- [16] Karim O. Elish and Mahmoud O. Elish. Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems and Software*, 81(5):649–660, May 2008.
- [17] Pedro Fazenda, James McDermott, and Una-May O’Reilly. A Library to Run Evolutionary Algorithms in the Cloud using MapReduce. In *European Conference on Applications of Evolutionary Computation (EvoApplications)*, pages 416–425, 2012.
- [18] Filomena Ferrucci, Pasquale Salza, M-Tahar Kechadi, and Federica Sarro. A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1664–1667, 2015.
- [19] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-Cloud Systems. In *IEEE International Conference on Cloud Computing (CLOUD)*, pages 887–894, 2014.
- [20] Martin Fowler. Continuous Integration, May 2006. URL: <https://www.martinfowler.com/articles/continuousIntegration.html>.
- [21] Wei Fu, Tim Menzies, and Xipeng Shen. Tuning for Software Analytics: Is It Really Necessary? *Information and Software Technology*, 76:135–146, 2016.
- [22] Pablo García-Sánchez, Gustavo Romero, Jesús González, Antonio Miguel Mora, Maribel García Arenas, Pedro Ángel Castillo, Carlos Fernandes, and Juan Julián Merelo. Studying the Effect of Population Size in Distributed Evolutionary Algorithms on Heterogeneous Clusters. *Applied Soft Computing*, 38:530–547, January 2016.
- [23] Mario García-Valdez, Leonardo Trujillo, Juan Julián Merelo Guervós, Francisco Fernandez de Vega, and Gustavo Olague. The EvoSpace Model for Pool-Based Evolutionary Algorithms. *Journal of Grid Computing*, 13(3):329–349, September 2015.

- [24] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
- [25] Iker Gondra. Applying Machine Learning to Software Fault-Proneness Prediction. *Journal of Systems and Software*, 81(2):186–195, February 2008.
- [26] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, November 2012.
- [27] Mark Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.
- [28] Mark Harman, Syed Islam, Yue Jia, Leandro L. Minku, Federica Sarro, and Komsan Srivisut. Less is More: Temporal Fault Predictive Performance over Multiple Hadoop Releases. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 240–246. Springer, 2014.
- [29] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification*, volume 7007, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [30] Ibrahim Abaker Targio Hashem, Nor Badrul Anuar, Abdullah Gani, Ibrar Yaqoob, Feng Xia, and Samee Ullah Khan. MapReduce: Review and Open Challenges. *Scientometrics*, 109(1):389–422, October 2016.
- [31] Di-Wei Huang and Jimmy Lin. Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems Using MapReduce. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 780–785, 2010.

-
- [32] Chao Jin, Christian Vecchiola, and Rajkumar Buyya. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In *IEEE International Conference on E-Science (e-Science)*, pages 214–221, 2008.
- [33] Marian Jureczko and Lech Madeyski. Towards Identifying Software Project Clusters with Regard to Defect Prediction. In *International Conference on Predictive Models in Software Engineering (PROMISE)*, pages 1–10, 2010.
- [34] Noor Elaiza Abd Khalid, Ahmad Firdaus Ahmad Fadzil, and Mazani Manaf. Adapting MapReduce Framework for Genetic Algorithm with Large Population. In *IEEE Conference on Systems, Process & Control (ICSPC)*, pages 36–41, 2013.
- [35] Jörg Lässig and Dirk Sudholt. General Upper Bounds on the Runtime of Parallel Evolutionary Algorithms*. *Evolutionary Computation*, 22(3):405–437, September 2014.
- [36] Guillaume Leclerc, Joshua E. Auerbach, Giovanni Iacca, and Dario Floreano. The Seamless Peer and Cloud Evolution Framework. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 821–828, 2016.
- [37] James Lewis and Martin Fowler. Microservices, a Definition of This New Architectural Term, March 2014. URL: <https://martinfowler.com/articles/microservices.html>.
- [38] Gabriel Luque and Enrique Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Number 367 in Studies in Computational Intelligence. Springer, 2011.
- [39] Ruchika Malhotra. A Systematic Review of Machine Learning Techniques for Software Fault Prediction. *Applied Soft Computing*, 27:504–518, February 2015.
- [40] Andrea Mambrini and Dirk Sudholt. Design and Analysis of Schemes for Adapting Migration Intervals in Parallel Evolutionary Algorithms. *Evolutionary Computation*, 23(4):559–582, December 2015.

- [41] Juan Julián Merelo Guervós, Antonio Miguel Mora García, Carlos M. Fernandes, and Anna Isabel Esparcia-Alcázar. SofEA, a Pool-Based Framework for Evolutionary Algorithms using CouchDB. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 109–116, 2012.
- [42] K. Meri, M. G. Arenas, A. M. Mora, J. J. Merelo, P. A. Castillo, P. García-Sánchez, and J. L. J. Laredo. Cloud-Based Evolutionary Algorithms: An Algorithmic Study. *Natural Computing*, 12(2):135–147, June 2013.
- [43] Thomas J. Ostrand and Elaine J. Weyuker. How to Measure Success of Fault Prediction Models. In *International Workshop on Software Quality Assurance (SOQUA)*, pages 25–30. ACM, 2007.
- [44] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman, and Fabio Kon. A Comprehensive View of Hadoop Research: A Systematic Literature Review. *Journal of Network and Computer Applications*, 46:1–25, November 2014.
- [45] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
- [46] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. Develop, Deploy and Execute Parallel Genetic Algorithms in the Cloud. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 121–122, 2016.
- [47] Pasquale Salza, Filomena Ferrucci, and Federica Sarro. Elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce. In *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1315–1322, 2016.
- [48] Federica Sarro, Sergio Di Martino, Filomena Ferrucci, and Carmine Gravino. A Further Analysis on the Use of Genetic Algorithm to Configure Support Vector Machines for Inter-Release Fault Prediction. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1215–1220. ACM, 2012.

-
- [49] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May O'Reilly. Flex-GP: Genetic Programming on the Cloud. In *European Conference on Applications of Evolutionary Computation (EvoApplications)*, volume 7248, pages 477–486, 2012.
- [50] Liyan Song, Leandro L. Minku, and Xin Yao. The Impact of Parameter Tuning on Software Effort Estimation using Learning Machines. In *International Conference on Predictive Models in Software Engineering (PROMISE)*, page 9, 2013.
- [51] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 321–332, 2016.
- [52] András Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
- [53] Kalyan Veeramachaneni, Ignacio Arnaldo, Owen Derby, and Una-May O'Reilly. FlexGP: Cloud-Based Ensemble Learning with Genetic Programming for Large Regression Problems. *Journal of Grid Computing*, 13(3):391–407, September 2015.
- [54] Abhishek Verma, Xavier Llorà, David E. Goldberg, and Roy H. Campbell. Scaling Genetic Algorithms Using MapReduce. In *International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 13–18, 2009.
- [55] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [56] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, 2010.

- [57] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.
- [58] Shin Yoo, Mark Harman, and Shmuel Ur. Highly Scalable Multi Objective Test Suite Minimisation Using Graphics Card. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 219–236, 2011.
- [59] Shin Yoo, Mark Harman, and Shmuel Ur. GPGPU Test Suite Minimisation: Search Based Software Engineering Performance Improvement Using Graphics Cards. *Empirical Software Engineering*, 18(3):550–593, June 2013.
- [60] Long Zheng, Yanchao Lu, Mengwei Ding, Yao Shen, Minyi Guoz, and Song Guo. Architecture-Based Performance Evaluation of Genetic Algorithms on Multi/Many-core Systems. In *IEEE International Conference on Computational Science and Engineering (CSE)*, pages 321–334, 2011.