Using Hadoop MapReduce for Parallel Genetic Algorithms: A Comparison of the Global, Grid and Island Models

Filomena Ferrucci fferrucci@unisa.it
Department of Computer Science, University of Salerno, Italy

Pasquale Salza psalza@unisa.it
Department of Computer Science, University of Salerno, Italy

Federica Sarro f.sarro@ucl.ac.uk
Department of Computer Science, University College London, United Kingdom

doi:10.1162/EVCO_a_00213

Abstract

The need to improve the scalability of Genetic Algorithms (GAs) has motivated the research on Parallel Genetic Algorithms (PGAs), and different technologies and approaches have been used. Hadoop MapReduce represents one of the most mature technologies to develop parallel algorithms. Based on the fact that parallel algorithms introduce communication overhead, the aim of the present work is to understand if, and possibly when, the parallel GAs solutions using Hadoop MapReduce show better performance than sequential versions in terms of execution time. Moreover, we are interested in understanding which PGA model can be most effective among the global, grid, and island models. We empirically assessed the performance of these three parallel models with respect to a sequential GA on a software engineering problem, evaluating the execution time and the achieved speedup. We also analysed the behaviour of the parallel models in relation to the overhead produced by the use of Hadoop MapReduce and the GAs' computational effort, which gives a more machine-independent measure of these algorithms. We exploited three problem instances to differentiate the computation load and three cluster configurations based on 2, 4, and 8 parallel nodes. Moreover, we estimated the costs of the execution of the experimentation on a potential cloud infrastructure, based on the pricing of the major commercial cloud providers. The empirical study revealed that the use of PGA based on the island model outperforms the other parallel models and the sequential GA for all the considered instances and clusters. Using 2, 4, and 8 nodes, the island model achieves an average speedup over the three datasets of 1.8, 3.4, and 7.0 times, respectively. Hadoop MapReduce has a set of different constraints that need to be considered during the design and the implementation of parallel algorithms. The overhead of data store (i.e., HDFS) accesses, communication, and latency requires solutions that reduce data store operations. For this reason, the island model is more suitable for PGAs than the global and grid model, also in terms of costs when executed on a commercial cloud provider.

Keywords

Genetic algorithms, parallel genetic algorithms, Hadoop MapReduce, global model, grid model, island model, fault prediction.

Manuscript received: 7 November 2016; revised: 1 May 2017; accepted: 23 May 2017.

1 Introduction

Genetic Algorithms (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. 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) (Luque and Alba, 2011; Yoo et al., 2011). Different approaches and technologies have been investigated and employed, ranging from multi-core systems on CPUs to many-core systems on GPUs and cloud technologies (Zheng et al., 2011; Yoo et al., 2011; Sherry et al., 2012; Salza et al., 2016a).

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 (Hashem et al., 2016). It is capable 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 (Polato et al., 2014). Furthermore, Hadoop has been positively adopted from the research community (Polato et al., 2014; Hashem et al., 2016). Motivated by the success of Hadoop MapReduce in many fields, several researchers have experimented in the last years its use to parallelise 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 (Luque and Alba, 2011). 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 "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." To the best of our knowledge, in the literature there does not exist a study providing an analysis and a comparison of the three models using Hadoop MapReduce. The main aim of this work is to fill this gap.

To this end, we carried out an empirical study by applying the three parallel models to a challenging software engineering problem that has been already addressed with a sequential GA, that is, Sequential Genetic Algorithm (SGA). In particular, we used GAs to search for a suitable configuration of Support Vector Machines (SVMs) to be used for inter-release fault prediction. Indeed, it has been shown that the performance of SVMs, and more generally of machine learning approaches, heavily depends on the selection of a suitable configuration for different software engineering prediction tasks (Corazza et al., 2010, 2013; Sarro et al., 2012; Song et al., 2013; Tantithamthavorn et al., 2016; Fu et al., 2016). Since a complete search of all possible combinations of parameters values may not be feasible due to the large search space, the use of GAs has been successfully exploited to automatically configure SVMs for software fault prediction (Di Martino et al., 2011; Sarro et al., 2012; Harman et al., 2014); however, the combined use of these techniques may affect the scalability of the approach when dealing with large software

projects. This motivated us to investigate the use of parallel GAss to configure SVM for software fault prediction. The choice of the fault prediction problem as a benchmark problem was also motivated by the fact that real-world problem instances of various sizes are publicly available (Menzies et al., 2016), thus allowing us to experiment PGAs with different computational loads.

We compared the three PGA solutions and the sequential version (i.e., the SGA) to understand their effectiveness in terms of execution time and speedup, and we studied the behaviour of the three parallel models in relation to the overhead produced using Hadoop MapReduce. To give a more machine-independent measure of the algorithms, we also counted the absolute number of fitness evaluations as a measure of the computational effort. 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. We executed a total of 30 runs for every single experiment of 300 generations each, with a total running time of about 120 days. Finally, we provide the estimation of costs for the major commercial cloud provider when executing the same GAs of our experiments. The study allowed us to identify the best model and highlighted some critical aspects.

The rest of the article is organised as follows. In Section 2, we first describe Hadoop MapReduce platform and the three models of GAs parallelisation. Section 3 presents the approach we employed to parallelise GAs by exploiting the Hadoop MapReduce platform. Sections 4 and 5 report, respectively, the design and the results of the empirical study we carried out to assess the effectiveness of the PGAs. Section 6 describes related work, while Section 7 contains some final remarks and future work.

2 Background

In this section, we give some background about Hadoop MapReduce and the strategies proposed in the literature to parallelise GAs.

2.1 Hadoop MapReduce

MapReduce is a programming paradigm whose origins lie in old functional programming. It was adapted by Google (Dean and Ghemawat, 2008) 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 Hadoop Distributed File System (HDFS), 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 PGAs.

Hadoop MapReduce exploits a distributed file system (an open source implementation of the Google File System), that is, 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, that is, "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, which a developer is expected to define as an extension for specific Java classes:

Split: the input data 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. 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. For each input split, it produces a list of records:

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

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

$$k_2 \rightarrow reducer_i$$

Reduce: it processes the input for each group of records with the same key and stores the output into the HDFS:

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

2.2 Three Parallel Models for Genetic Algorithms

The following models have been proposed in the literature (Luque and Alba, 2011) to parallelise the execution of GAs:

- "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."

In the global model, 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 of evaluating the fitness of each individual. This model does not require any changes to the SGA since the fitness computation for each individual is independent and thus can be achieved in parallel.

The grid model 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, 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 subpopulations 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 slowed down the use of PGAs for software related tasks.

3 The Parallel GAs Based on MapReduce

To realise the PGAs explained in Section 2, we exploited the *elephant56* framework (Ferrucci et al., 2015; Salza et al., 2016b), which is an open source project supporting the development and execution of PGAs.[1]

Hadoop distributes software applications ("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 (ResourceManager for the Hadoop terminology), which manages the work of the other computation slave nodes (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.

In the following, we first explain how we implemented the sequential version (i.e., SGA) and then how we mapped the MapReduce elements to the PGA models. The main challenge during the design phase was to limit the communication and synchronisation overhead of parallel tasks. We had to 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, "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

[1] elephant56 is freely available at https://github.com/pasqualesalza/elephant56

Algorithm 1 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 (parent_1, parent_2) ∈ selectedCouples do
10:         (child_1, child_2) ← CROSSOVER(parent_1, parent_2)
11:         offspring ← offspring ∪ {child_1} ∪ {child_2}
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
```

execution time by parallelising. The positions of the synchronisation barriers are deeply bonded to the implemented models and thus we had to take them into consideration case-by-case to realise each of the following three parallel model adaptations. Moreover, we supply further details of implementation that we consider essential to understand the rest of the work.

3.1 Sequential Genetic Algorithm (SGA)

There are several possible versions of GAs 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 1.

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 genetic operator applied is the FITNESSEVALUATION (3–5), which evaluates and assigns 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 noting 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").

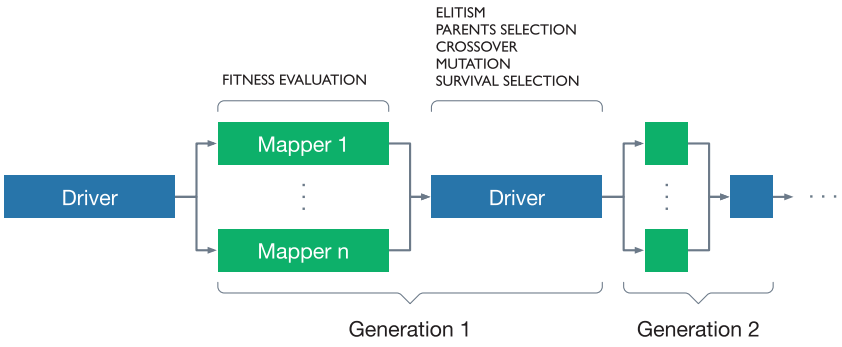


Figure 1: The flow of Hadoop MapReduce implementation for PGA_{global}.

3.2 PGA Based on the Global Model

The PGA we implemented for the global model on MapReduce (i.e., PGA_{global}) has the same behaviour as the sequential version, but it resorts to parallelisation for the fitness evaluation. Figure 1 shows the flow of the model. The master node, also referred to 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 Algorithm 1, 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 3.1 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.

In more detail, the slave nodes in the cluster perform only the fitness evaluation operator. For all three models, the mappers receive the records in the form: (individual, destination). The "destination" field is used only by the other models and it will be mentioned later. We deliberately disabled the reduce phase because there is no need to move 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.

3.3 PGA Based on the Grid Model

The PGA we implemented 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 (see Figure 2) 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 neighbourhood also in the next generation.

The `Driver` has the task of randomly generating a sequence of neighbourhood destinations for the individuals in the current population. These destinations are stored

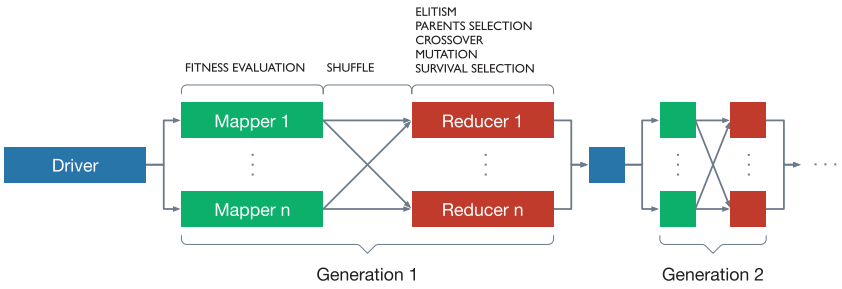


Figure 2: The flow of Hadoop MapReduce implementation for PGA_{grid}.

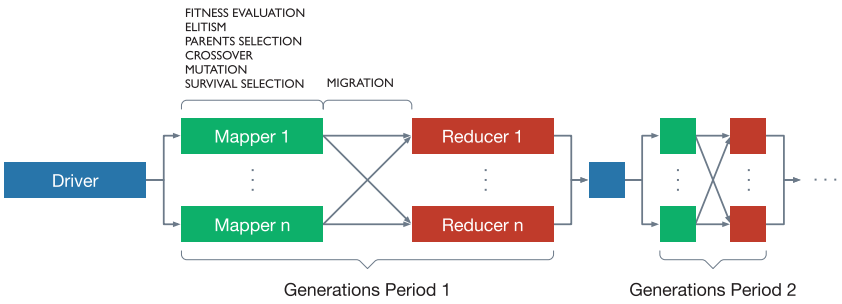


Figure 3: The flow of Hadoop MapReduce implementation for PGA_{island}.

into the record as the value fields so the destinations are known a priori. We exploited the parallelisation 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, we 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 PGA Based on the Island Model

The PGA we implemented 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 (see Figure 3). 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, we exploited a job for each period.

In Hadoop, the numbers of mappers and reducers are not strictly correlated, but we coupled them to represent them as islands. We used the mappers 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 individuals 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.

3.5 Implementation Details

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, we tuned the performance of our implementations by using Avro.

In order to analyse the behaviour of the implemented models during our experiments, we added a reporter component that gave us the details of executions. We were interested in both the genetic trend of population and the execution time described in detail in a fine-grained manner. Our reporter component stores data into the HDFS with a noninvasive and asynchronous working so that the execution time of experiments is never influenced by extra operations.

4 Empirical Study Design

Our aim was to understand if PGAs based on Hadoop MapReduce can be an effective solution to improve the scalability of GAs. Therefore, we had to verify if, and possibly when, PGAs allow us to get a better execution time compared to the sequential version (i.e., SGA). Moreover, we were interested in understanding which PGA model is most effective among the global, grid, and island models. Thus, we sought to answer the following research question:

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

To address the RQ we considered as a benchmark the software engineering problem of configuring the SVMs by using GAs for inter-release fault prediction. 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 allowed us to assess the PGAs scalability on different real-world problem instances of various sizes. Furthermore, the problem was already addressed in previous work (Di Martino et al., 2011; Sarro et al., 2012) by using a sequential approach. To verify the effectiveness of PGAs against SGA, we exploited the sequential approach proposed by Di Martino et al. (2011). Even if our main aim was to exploit GAs for SVMs configuration as a benchmark problem in terms of execution time, we also assessed whether the predictive performance of SVMs was affected by executing the GAs in parallel (see Section 4.4.5). Moreover, we estimated the costs of the execution of the experimentation on a potential cloud infrastructure, based on the pricing of the major commercial cloud providers.

Details about the problem and GAs configuration are provided in Section 4.1. The datasets employed for the empirical study are described in Section 4.2. To understand

[2] https://avro.apache.org

the effectiveness of PGAs and compare the three parallel models, we applied the experimental method described in Section 4.3 and employed several evaluation criteria, namely the execution time, speedup, overhead, computational effort, predictive performance, and cloud costs (see Section 4.4). The hardware employed to run the experiments is reported in Section 4.5. Finally, Section 4.6 analyses some threats to validity that may have affected our experimentation.

4.1 Using GAs to Configure SVMs for Software Fault Prediction

The use of machine-learning-techniques to predict software faults has received increasing attention in recent years (Hall et al., 2012; Malhotra, 2015). 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. A predictive model for software fault usually classifies a target software module as faulty or non-faulty based on the information available from previously released software components. In the software fault prediction context, the dependent variable is represented by the faults contained in a software component while the independent variables may vary from project to project and can be related to different aspects of the software project such as code size and complexity metrics, software process metrics, software testing metrics (Fenton and Neil, 1999; D'Ambros et al., 2012; Bowes et al., 2016). In order to build fault prediction models several machine learners, such as Decision Trees, Support Vector Machines, and Naive Bayes, have been widely used. Nevertheless, it has been shown that the predictive performance of machine-learning techniques for fault prediction strongly depends on the selection of a suitable configuration (Di Martino et al., 2011; Sarro et al., 2012; Hall and Bowes, 2012; Tantithamthavorn et al., 2016; Fu et al., 2016).

GAs have been proposed to configure *Support Vector Machines (SVMs)* for software fault prediction (Di Martino et al., 2011; Harman et al., 2014; Sarro et al., 2012; Gondra, 2008). The idea is based on the observation that such a problem can be formulated as an optimisation problem: between the possible configurations, we have to identify 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 our choice of using this problem as a benchmark for PGAs.

In our experiments, we employed the technique proposed by Di Martino et al. (2011), which has been also adopted in other work (Sarro et al., 2012; Harman et al., 2014). 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, we employed the widely used *Radial Basis Functions* (*RBF*), 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 0.000001 to 0.01 and 8 to 32000, 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, we executed SVMs with such a configuration thus obtaining the fault predictions. Such predictions are then evaluated using *F-measure* (Witten and Frank, 2005) as a performance criterion. The F-measure is defined as:

$$F - measure = 2\frac{precision * recall}{precision + recall} \quad (1)$$

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. We employed the same setting used in previous work (Di Martino et al., 2011; Sarro et al., 2012; Harman et al., 2014) both for the SGA and PGAs:

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, we used a number of neighbourhoods equal to the cluster size. As for the island model, we also needed to identify the migration period and the number and selection policy of migrants. We set the migration period to 30 applying a ring topology exchange, whereas the number of migrants to 5% of the best individuals per island.

4.2 Datasets

We exploited data from the *PROMISE* repository (Menzies et al., 2016), which contains several datasets for fault prediction and we chose the software projects with more than two releases. Thus, we retained three datasets for a total of 10 releases: *Log4j* (vv. 1.0, 1.1, and 1.2), *Lucene* (vv. 2.0, 2.2, and 2.4), and *POI* (vv. 1.5, 2.0, 2.5, and 3.0). Each release contains a set of components (i.e., Java classes) described in terms of Chidamber and Kemerer (CK) metrics (Chidamber and Kemerer, 1994), 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 and Madeyski (2010).

We chose these three datasets 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 and 9.5 times slower than *Log4j*, respectively. Therefore, this allowed us to study the behaviour of PGAs on three problem instances of various size that we identified as "small" for *Log4j*, "medium" for *Lucene*, and "large" for *POI*.

4.3 Experimental Method

We addressed the RQ by comparing the performance of the PGAs based on each of the three parallel models explained in Section 3 and SGA.

To observe the effectiveness of the considered techniques for inter-release fault prediction, we used the typical setting where data from the former releases are exploited to build the model to predict faults for a new release (Ostrand and Weyuker, 2007). In particular, given a software project having n releases, we used the data collected in the first $n - 1$ releases of the project as the training set and the data collected for the last release as the test set. This allowed us to simulate the situation that typically arises in real

Table 1: 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.

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

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 training and test sets we employed is reported in Table 1 together with the percentage of faulty and non faulty components.

We executed all the parallel models on three different cluster configurations (i.e., C2, C4, and C8) characterised by a different number of nodes (see details in Section 4.5). For each combination of model, dataset, and cluster configuration, we executed 30 runs. Thus, we executed 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.

4.4 Evaluation Criteria

To compare the performance of the employed algorithms, we mostly followed the best practices in reporting the results with PGAs, identified by Luque and Alba (2011). We evaluated them both in terms of execution time, speedup, overhead, and computational effort, as detailed in the following. We also evaluated the predictive performance 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, we estimated the costs of the same executions on commercial cloud providers infrastructures. To cope with the stochastic nature of GAs and hardware executions, we performed multiple executions and assessed the results by using the statistical tests described in Section 4.4.7.

4.4.1 Execution Time

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

4.4.2 Speedup

The speedup is defined as the ratio of the sequential execution time to the parallel execution time. There are two types of speedup, that is, the "strong" and the "weak" (Luque and Alba, 2011). The strong speedup compares the parallel run time against the best-so-far sequential algorithm. We could not apply it since our intention was to compare the models on the Hadoop MapReduce platform, rather than comparing against different technologies. Moreover, we could not find any implementation providing the same algorithms as we do.

Instead, the weak speedup compares the parallel algorithm developed by the researchers against their own sequential version. We calculated it by dividing the total

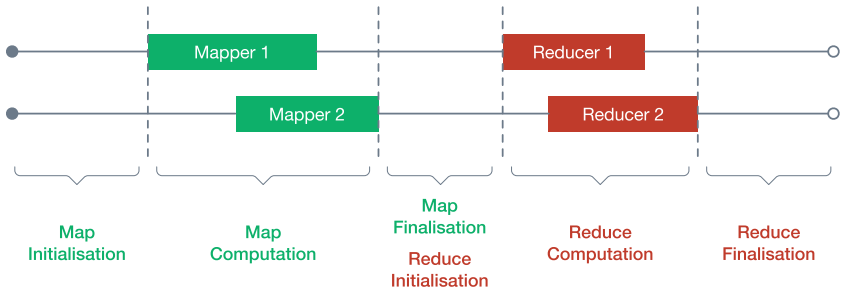


Figure 4: The time measurement method for multiple nodes.

amount of time that SGA required by the amount of time required by the PGA. We compared the achieved speedup with respect to the ideal speedup. It is worth noting that the ideal speedup is equal to the number of 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 (2011), 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, we computed and reported the predictive performance of the solutions at the end of the executions.

4.4.3 Overhead

To understand the reasons that prevent the PGAs to have a speedup near the ideal one on the Hadoop MapReduce platform, we quantified the overhead for each execution. We distinguished between overhead and computation times. In the following, we describe the method we adopted to determine these times.

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 4 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 Figure 4, we consider the "map initialisation" time 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.

4.4.4 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 (Luque and Alba, 2011). In the field of metaheuristics, the computational effort is generally measured by the number of

Table 2: 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

evaluations corresponding to the number of visited points of the solution space. We counted the absolute number of computed fitness evaluations, reporting them on average of the total number of runs. Because we fixed the number of generations, this measure depends only on the characteristics of the used model and cluster size.

Moreover, we calculated also the eval/s measure for each of the models. 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, that is, whose fitness evaluation function requires a certain execution time, and the same degree of parallelisation.

4.4.5 Predictive Performance

To evaluate the predictive performance, we employed four widely used measures (i.e., *precision*, *recall*, *accuracy*, and *F-measure* (Witten and Frank, 2005)). The *precision* is the ratio between the number of components classified as TP and the number of those classified as TP or FP:

$$precision = \frac{TP}{TP + FP} \tag{2}$$

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

$$recall = \frac{TP}{TP + FN} \tag{3}$$

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, and FN):

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

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 in Equation (1).

4.4.6 Cloud Costs Estimation

Even though we executed the experimentation on a private infrastructure, as described in Section 4.5, it is also possible to run a Hadoop cluster on any commercial cloud infrastructure. We estimated a likely cost for the same execution that we actually performed, based on the pricing tables of the most used cloud providers. The estimation was based on the selection of cloud instances, that is, the virtual machines, with a hardware configuration at least equal to the one we employed in our experiments. Table 2 reports the configurations and pricing of the instances we selected for the estimation.

Table 3: Virtual machine configurations.

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

Table 4: 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

4.4.7 Statistical Tests

We executed 30 runs in order to cope with the inherent randomness of dynamic execution time and the GAs and reported the average results. Then, we used the nonparametric inferential statistical test, that is, the *Wilcoxon Test* (Conover, 1999), as recommended in the literature (Arcuri and Briand, 2011). The Wilcoxon signed rank test verifies, as the null hypothesis, if two populations have identical distributions. It is particularly useful when no assumptions about the normality of the distributions are possible, as for our case. For all the statistical tests, we accepted a probability of 5% of committing a Type-I-Error. Furthermore, we used the Vargha-Delaney $\hat{A}_{12}$ effect size to estimate the probability that two algorithms have against each other in obtaining better results regarding the execution time and predictive performance measures (Arcuri and Briand, 2011). 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.

4.5 Hardware

To execute the experiment, we employed a private OpenStack cloud platform where we virtualised the machines to compose the Hadoop clusters. To run a fair experiment, we used the same configuration for each virtual machine (see Table 3). It is worth noting that we did not simulate any hardware component, thus exploiting the virtualisation of OpenStack only as a way to equally divide the hardware infrastucture. We completely dedicated the partial atomic resource (e.g., CPU cores, RAM GB) to the running instances so that they could have fully used them without overlapping with others.

In our empirical study, we used three different types of Hadoop clusters, summarised in Table 4. SGA was executed on a single node, while for PGAs we exploited the clusters C2, C4, and C8. The clusters are organised in one master node and a number of slaves equal to the parallelisation degree. We needed to separate the master node from

the slaves because our implementations have a sequential part and we 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. We installed Hadoop through the Hortonworks distribution, which eased the orchestration and monitoring of the clusters.

We ran multiple experiments simultaneously on a total of 150 OpenStack virtual machines. The empirical study took about 120 days for a total of 900 runs when executing 30 runs for each single experiment of 300 generations each.

4.6 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, we also provided the computational effort as machine-independent measure of the algorithms.

Threats to *internal validity* concern any confounding factors that could influence our 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 we mitigated such a threat by executing all the experiments 30 times and presenting the average results (Arcuri and Briand, 2011). Moreover, 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.

Threats to *external validity* concern the generalisability of our findings outside the scope of our study. An external threat is due to the fact that we benchmarked the three PGAs for a specific software engineering task, that is, 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 datasets, thus constituting a suitable benchmark for the three different parallel architectures. To this end, we investigated three datasets with different sizes and characteristics. It is worth noting that the configuration we chose for the GAs is not exclusive and we could have used other possible parameters sets aiming at improving the predictive performance of models. However, since we were interested mostly in the comparison of the execution time performance, we selected the most trivial configuration for all the PGAs. Moreover, for the grid model we chose to use a number of neighbourhoods equal to the cluster size. On the one hand, if we had used a minor number, we could not have exploited 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 fewer individuals than the other case.

5 Results

In this section, we present the results of our study. Let us recall that for each considered dataset (i.e., *Log4j*, *Lucene*, and *POI*), we executed 30 times the SGA and PGAs. The comparison between SGA and PGAs with respect to the execution time is in Section 5.1. The analyses of the speedup and overhead are in Sections 5.2 and 5.3, respectively. Section 5.4 reports the computational effort and the predictive performance is analysed in Section 5.5. Section 5.6 concludes with the estimation of cloud costs. Some of the results are reported in the online appendix (Ferrucci et al., 2016).

Table 5: Execution time achieved by executing 30 times SGA and PGAs on the datasets.

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

5.1 Execution Time

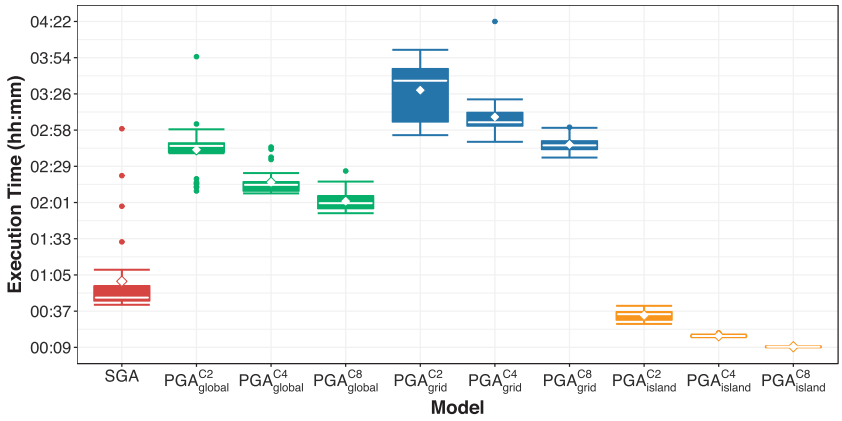
Table 5 and Figure 5 show the achieved execution times obtained over 30 runs. We can observe that the execution for each of the considered clusters of PGA_{island} is always faster than each SGA execution for all the considered datasets. Every PGA performs better than SGA for the *POI* dataset, while for the other two datasets PGA_{global} and PGA_{grid} are slower than SGA, regardless of the number of parallel nodes used.

The Wilcoxon test results (see Table 6) confirm the above observations. The execution time of PGA_{island}, using C2, C4, and C8 clusters, is significantly lower (i.e., *p-value* < 0.05) than the one of SGA on all the considered datasets, while the execution time of PGA_{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 6) confirm (i.e., $\hat{A}_{12} = 0$) that PGA_{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 perform better than SGA. It can be explained by the fact that, for small instances of the problem, the overhead due to data accesses and communication between the nodes is higher than the time for the fitness function.

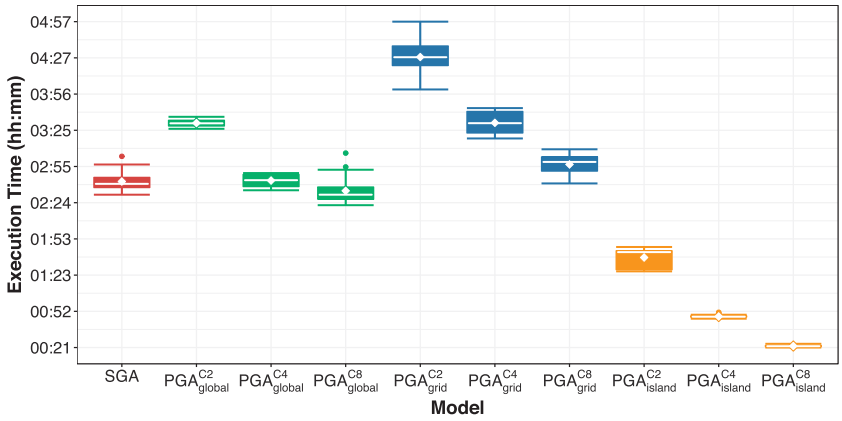
As for the comparison between PGAs, the boxplots of Figure 5 and the complete Wilcoxon and Vargha-Delaney tests results (Ferrucci et al., 2016), show that the PGA_{grid} model is the slowest model and significantly different from the other two models.

5.2 Speedup

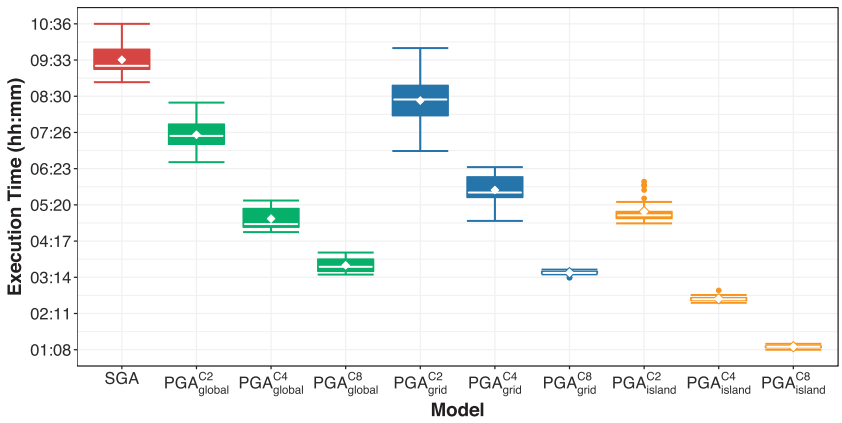
Once we established execution times of the SGA and PGAs, we calculated the speedup values. Figure 6 shows that the use of parallelisation is worth while mainly for the island model, which allowed us to speed up the execution time with respect to SGA of an average over the three datasets of 7.0 times by exploiting PGA^{C8}_{island}, 3.4 times by exploiting PGA^{C4}_{island} and 1.8 times by exploiting PGA^{C2}_{island} times. It is clear from Figure 6 that



(a) *Log4j*



(b) *Lucene*



(c) *POI*

Figure 5: Execution times achieved by SGA and PGAs on the three clusters for the three databases.

Table 6: 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 datasets.

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

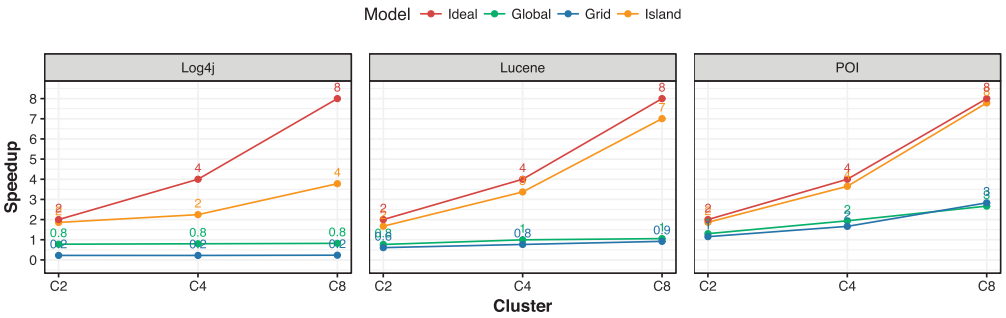
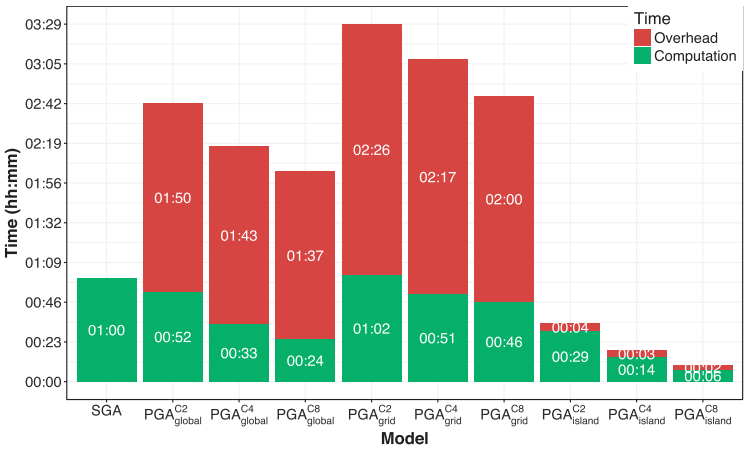


Figure 6: Speedup trend per dataset.

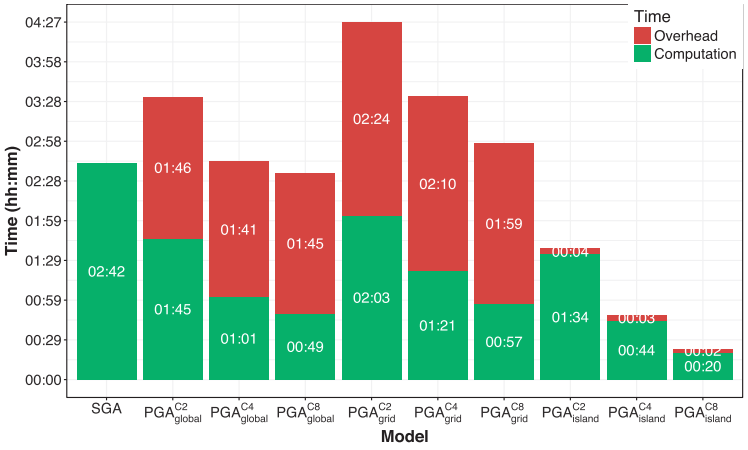
PGA_{island} tends to the ideal speedup value. On the other hand, PGA_{global} and PGA_{grid} improved their speedup only slightly, because of the overhead discussed in Section 5.3. The complete speedup values are reported in the online appendix (Ferrucci et al., 2016).

5.3 Overhead

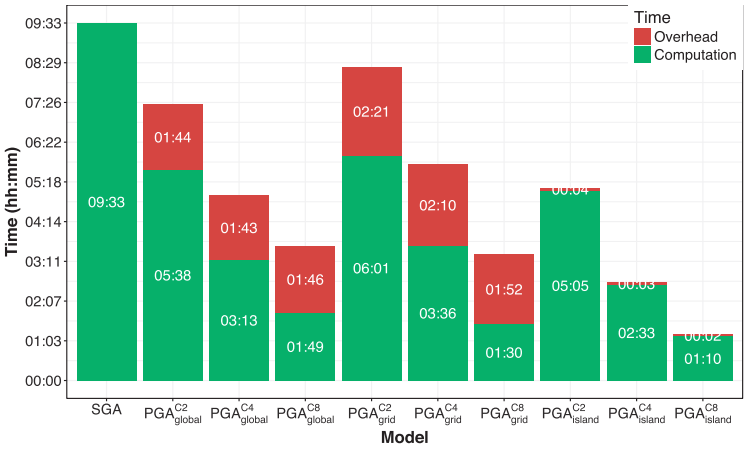
To further investigate the behaviour of the parallel implementations on the Hadoop MapReduce platform, we analysed the execution time of PGAs with a more fine-grained scale. Figure 7 shows the computation and overhead times for each PGA and dataset combination, where the overhead is intended as the additional time other than the computation, generally due to communication and Hadoop environment 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 we can see from Figure 7, for the *Log4j* and *Lucene* datasets on global and grid models the overhead time surpasses the computation time. As we mentioned above, in the presence of



(a) *Log4j*



(b) *Lucene*



(c) *POI*

Figure 7: The computation and overhead times for each PGA model.

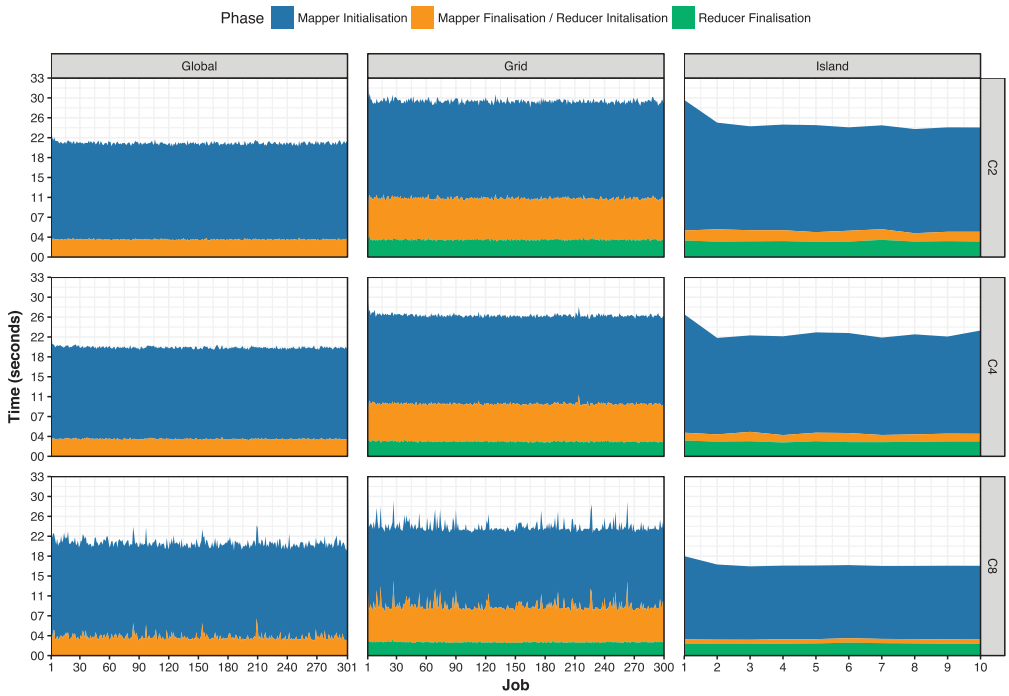


Figure 8: The MapReduce mean overhead times per job for each PGA model and cluster configuration on the *Lucene* dataset over 30 runs.

heavy computational work (i.e., *POI* the dataset) the 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.

We also analysed the overhead time per job. As an example, we report in Figure 8 the mean execution time for each job over 30 runs and all the cluster configurations for the *Lucene* dataset. However, similar observations can be done for the other datasets and configurations and are shown in the online appendix (Ferrucci et al., 2016). We can 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 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). The map initialisation phase is common to the three parallel models and it takes a similar amount of time: every time a new job is requested, Hadoop 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 those for the map phase. 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: they are comprehensive of the partitioner component work, which is the Hadoop 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

Table 7: Average evaluation number and evaluation per second values achieved by executing 30 times SGA and PGAs on the three datasets and three clusters.

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

individuals can be assigned to a different destination whereas in the island model this is true only for 5% of 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, that is, writing back the processed individuals to the HDFS.

We tested the correlation between overhead times using the ANOVA test but we found none. We impute the reason to the nondeterminism of several layers of abstraction: Hadoop MapReduce is executed on JVMs as a shared process on cloud virtual machine instances in a shared OpenStack environment. Although there is not strong statistical evidence, we observed that, on average, the overhead times seem to be independent of the dataset and cluster size.

5.4 Computational Effort

Table 7 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 are equivalent 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 exactly 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 whereas PGA_{island} is balanced. In general, we can affirm 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 the online appendix (Ferrucci et al., 2016), confirmed it. Moreover, Table 7 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.

5.5 Predictive Performance

As we can observe from Table 8 reporting the median values of the four evaluation criteria we employed, 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 we performed confirm the above considerations. In Table 9 we report the statistical tests results about the *F-measure*.

5.6 Cloud Costs Estimation

Figure 9 shows the estimation of costs for the Amazon cloud provider for executing the same GAs of our experiments, in relation to the required execution time. The complete report of cloud costs estimation for all the selected cloud providers is presented in the online appendix (Ferrucci et al., 2016). We also include the estimation of SGA on a machine purchased on the same cloud provider, having the same configuration as the ones used for the Hadoop clusters. As can be seen from Figure 9, in the case of PGA_{global} and PGA_{grid}, they save time against SGA in the case of the POI dataset but impose 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 with a greater number of machines. The distance in time is more remarkable for the POI dataset, thus making the use of the cloud worthwhile in the case of large computational load.

6 Related Work

In this section, we report the most relevant work that inspired and guided our study, highlighting similarities and differences of them. We were mainly interested in the solutions involving the MapReduce paradigm but we are also reporting some important work employing different technologies to run PGAs.

6.1 PGAs Based on MapReduce

Table 10 is a summary of the work related to the use of MapReduce for PGAs.

Jin et al. (2008) 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 2.2. 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. (2009). 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 8: Predictive performance median values achieved by executing 30 times SGA and PGAs on the three dataset.

(a) *Log4j*

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

(b) *Lucene*

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

(c) *POI*

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

Table 9: 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 datasets.

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

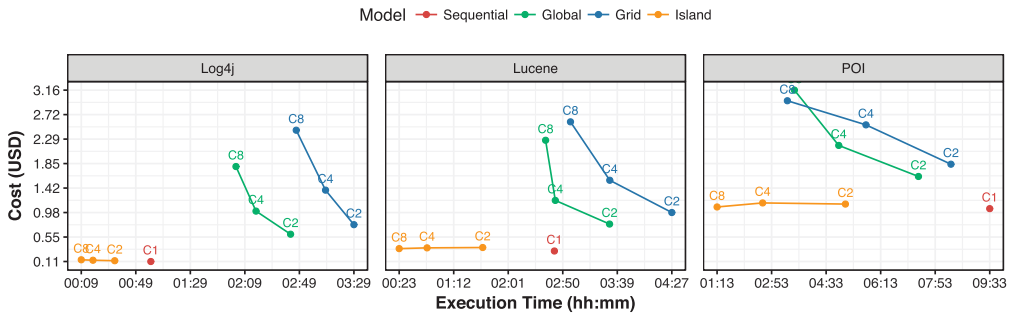


Figure 9: The Amazon cloud provider execution time and cost values for the execution of SGA and PGAs on the datasets and cluster sizes.

Huang and Lin (2010) 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 Geronimo et al. (2012) 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 parallel genetic algorithm allowed

Table 10: 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 (Jin et al., 2008)	2008		✓		Private cluster	DLTZ4, DLTZ5 and the Aerodynamic Airfoil Design Simulation	MapReduce suits the GAs parallelisation, demonstrated by experimenting with the grid model, but the paradigm needs to be adapted to PGA models
Scaling Genetic Algorithms Using MapReduce (Verma et al., 2009)	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 (Huang and Lin, 2010)	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 (Fazenda et al., 2012)	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 (Di Geronimo et al., 2012)	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

Table 10: Continued.

Title	Year	PGA Model			Experimentation		Results
		Global	Grid	Island	Hardware	Problems	
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud (Di Martino et al., 2013)	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 (Khalid et al., 2013)	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 (Ferrucci et al., 2015)	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

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., HDFS) as the main cause.

Di Martino et al. (2013) 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 with 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. (2013) 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 in 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. (2012) were the first to consider the parallelisation of Evolutionary Algorithm (EAs) on Hadoop MapReduce platform in the general purpose form of a library, in order to simplify the developing effort for parallel EAs implementations. The work has been further enhanced by Sherry et al. to produce *FlexGP* (Sherry et al., 2012; Veeramachaneni et al., 2015). 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.

To the same aim, Ferrucci et al. (Ferrucci et al., 2013, 2015; Salza et al., 2016b) implemented a framework for PGAs development, deployment, and execution on Hadoop MapReduce platform, based on the island model. They described the design of the framework and how a developer could interact in defining his/her genetic operators or using some provided samples included with the framework. They also assessed the framework with a preliminary experiment on the problem of *Feature Subset Selection*.

As can be seen from Table 10, no previous work has compared all three models using Hadoop MapReduce.

6.2 Parallel Genetic Algorithms

In the literature we can find many proposals of both parallel GAs and EAs using different approaches, methods, and technologies (Knysh and Kureichik, 2010; Luque and Alba, 2011; Johar et al., 2013; Sudholt et al., 2015). In this section, we focus on those studies that describe approaches and results that influenced our work even if they are not always strictly related to the parallel models compared in our study. Zheng et al. (2011) addressed a research question similar to ours by comparing the global and the island models, using a multi-core (i.e., CPUs) and a many-core (i.e., GPUs) systems. The main difference with respect to our work is that 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 one based on CPUs, 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.,

multithreading). They stated that a distributed architecture is worthwhile for GAs parallelisation. As a first attempt of employing cloud technologies, Merelo Guervós et al. (2012) devised *SofEA*, a model for Pool-based EAs in the cloud, an EA 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* (García-Valdez et al., 2015), 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 runtime and lowering the costs.

7 Conclusions and Future Work

In this article, we faced the parallelisation of Genetic Algorithms (GAs) on the Hadoop MapReduce platform, based on three models, that is, the global, grid, and island models. As a benchmark problem, we considered the use of three different Parallel Genetic Algorithms (PGAs) to solve a software engineering problem of configuring the Support Vector Machines (SVMs) for inter-release fault prediction. We empirically assessed the effectiveness of these models in terms of execution time, speedup, overhead, and computational effort by using three publicly available datasets of real software faults, widely used in fault prediction studies. The three datasets were chosen considering their different sizes in order to vary the execution time of the GAs.

We found that the use of PGA based on the island model outperforms the use of Sequential Genetic Algorithm (SGA) 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., Hadoop Distributed File Systems (HDFSs)) accesses, communication, and latency may impair parallel solutions based on the global and grid models when executed on small problem instances. This is not the case for 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. We also observed that the use of more nodes allowed us to further reduce the execution time. The use of the island model enabled us to speed up the average execution time over the three datasets with respect to SGA of 7.0, 3.4, and 1.8 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. 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 (Mambrini and Sudholt, 2015), using Hadoop MapReduce. We also aim to compare our results with the theoretical models for GAs parallelisation proposed in the literature (Cantú-Paz and Goldberg, 1999; Lässig and Sudholt, 2014), adapting them to consider the effects of Hadoop MapReduce.

Regarding the global model, a 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 individual's 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 syncronisation barrier, that is, 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, we placed a syncronisation barrier in the driver, as suggested by Di Martino et al. (2013). A substantial simplification of the communication could be applied by making the execution of neighbourhoods evolution entirely independent throughout all the generations. We did not exploit this option because we were interested in providing a flexible solution, allowing us to define and use different strategies for the resolution of GAs in the form of a framework. It is on our future agenda to implement and study these improvements also for the global and grid models.

References

Arcuri, A., and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering*, pp. 1–10.

Bowes, D., Hall, T., Harman, M., Jia, Y., Sarro, F., and Wu, F. (2016). Mutation-aware fault prediction. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 330–341.

Cantú-Paz, E., and Goldberg, D. E. (1999). On the scalability of parallel genetic algorithms. *Evolutionary Computation*, 7(4):429–449.

Chidamber, S. R., and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

Conover, W. J. (1999). *Practical nonparametric statistics*. 3rd ed. New York: John Wiley & Sons.

Corazza, A., Di Martino, S., Ferrucci, F., Gravino, C., Sarro, F., and Mendes, E. (2010). How effective is Tabu search to configure support vector regression for effort estimation? In *International Conference on Predictive Models in Software Engineering*, Article No. 4.

Corazza, A., Di Martino, S., Ferrucci, F., Gravino, C., Sarro, F., and Mendes, E. (2013). Using Tabu search to configure support vector regression for effort estimation. *Empirical Software Engineering*, 18(3):506–546.

D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4):531–577.

Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Di Geronimo, L., Ferrucci, F., Murolo, A., and Sarro, F. (2012). 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*, pp. 785–793.

Di Martino, S., Ferrucci, F., Gravino, C., and Sarro, F. (2011). A genetic algorithm to configure support vector machines for predicting fault-prone components. In *International Conference on Product-Focused Software Process Improvement*, pp. 247–261.

Di Martino, S., Ferrucci, F., Maggio, V., and Sarro, F. (2013). Towards migrating genetic algorithms for test data generation to the cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, pp. 113–135.

Fazenda, P., McDermott, J., and O'Reilly, U.-M. (2012). A library to run evolutionary algorithms in the cloud using MapReduce. In *European Conference on Applications of Evolutionary Computation*, pp. 416–425.

Fenton, N. E., and Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689.

Ferrucci, F., Kechadi, M.-T., Salza, P., and Sarro, F. (2013). A framework for genetic algorithms based on Hadoop. *Computing Research Repository*. abs/1312.0086.

Ferrucci, F., Salza, P., Kechadi, M.-T., and Sarro, F. (2015). A parallel genetic algorithms framework based on Hadoop MapReduce. In *ACM/SIGAPP Symposium on Applied Computing*, pp. 1664–1667.

Ferrucci, F., Salza, P., and Sarro, F. (2016). Using Hadoop MapReduce for parallel genetic algorithms: A comparison of the global, grid and island models—Appendix. Retrieved from https://doi.org/10.6084/m9.figshare.5091898

Fu, W., Menzies, T., and Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135–146.

García-Valdez, M., Trujillo, L., Merelo Guervós, J. J., Fernandez de Vega, F., and Olague, G. (2015). The EvoSpace model for pool-based evolutionary algorithms. *Journal of Grid Computing*, 13(3):329–349.

Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195.

Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304.

Hall, T., and Bowes, D. (2012). The state of machine learning methodology in software fault prediction. In *IEEE International Conference on Machine Learning and Applications*, pp. 308–313.

Harman, M., Islam, S., Jia, Y., Minku, L., Sarro, F., and Srivisut, K. (2014). Less is more: Temporal fault predictive performance over multiple Hadoop releases. In *International Symposium on Search Based Software Engineering*, pp. 240–246.

Hashem, I. A. T., Anuar, N. B., Gani, A., Yaqoob, I., Xia, F., and Khan, S. U. (2016). MapReduce: Review and open challenges. *Scientometrics*, 109(1):389–422.

Huang, D.-W., and Lin, J. (2010). Scaling populations of a genetic algorithm for job shop scheduling problems using MapReduce. In *IEEE International Conference on Cloud Computing Technology and Science*, pp. 780–785.

Jin, C., Vecchiola, C., and Buyya, R. (2008). MRPGA: An extension of MapReduce for parallelizing genetic algorithms. In *IEEE International Conference on E-Science*, pp. 214–221.

Johar, F. M., Azmin, F. A., Suaidi, M. K., Shibghatullah, A. S., Ahmad, B. H., Salleh, S. N., Aziz, M. Z. A. A., and Shukor, M. M. (2013). A review of genetic algorithms and parallel genetic algorithms on graphics processing unit (GPU). In *IEEE International Conference on Control System, Computing and Engineering*, pp. 264–269.

Jureczko, M., and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *International Conference on Predictive Models in Software Engineering*, pp. 1–10.

Khalid, N. E. A., Fadzil, A. F. A., and Manaf, M. (2013). Adapting MapReduce framework for genetic algorithm with large population. In *IEEE Conference on Systems, Process & Control*, pp. 36–41.

Knysh, D. S., and Kureichik, V. M. (2010). Parallel genetic algorithms: A survey and problem state of the art. *Journal of Computer and Systems Sciences International*, 49(4):579–589.

Lässig, J., and Sudholt, D. (2014). General upper bounds on the runtime of parallel evolutionary algorithms. *Evolutionary Computation*, 22(3):405–437.

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

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing*, 27:504–518.

Mambrini, A., and Sudholt, D. (2015). Design and analysis of schemes for adapting migration intervals in parallel evolutionary algorithms. *Evolutionary Computation*, 23(4):559–582.

Menzies, T., Krishna, R., and Pryor, D. (2016). The Promise Repository of Empirical Software Engineering Data. Retrieved from http://openscience.us/repo

Merelo Guervós, J. J., Mora García, A. M., Fernandes, C. M., and Esparcia-Alcázar, A. I. (2012). SofEA, a pool-based framework for evolutionary algorithms using CouchDB. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 109–116.

Ostrand, T. J., and Weyuker, E. J. (2007). How to measure success of fault prediction models. In *International Workshop on Software Quality Assurance*, pp. 25–30.

Polato, I., Ré, R., Goldman, A., and Kon, F. (2014). A comprehensive view of Hadoop research: A systematic literature review. *Journal of Network and Computer Applications*, 46:1–25.

Salza, P., Ferrucci, F., and Sarro, F. (2016a). Develop, deploy and execute parallel genetic algorithms in the cloud. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 121–122.

Salza, P., Ferrucci, F., and Sarro, F. (2016b). Elephant56: Design and implementation of a parallel genetic algorithms framework on Hadoop MapReduce. In *Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1315–1322.

Sarro, F., Di Martino, S., Ferrucci, F., and Gravino, C. (2012). 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*, pp. 1215–1220.

Sherry, D., Veeramachaneni, K., McDermott, J., and O'Reilly, U.-M. (2012). Flex-GP: Genetic programming on the cloud. In *European Conference on Applications of Evolutionary Computation*, pp. 477–486.

Song, L., Minku, L. L., and Yao, X. (2013). The impact of parameter tuning on software effort estimation using learning machines. In *International Conference on Predictive Models in Software Engineering*, p. 9.

Sudholt, D., Kacprzyk, J., and Pedrycz, W. (2015). Parallel evolutionary algorithms. In *Springer handbook of computational intelligence*, pp. 929–959. New York: Springer.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2016). Automated parameter optimization of classification techniques for defect prediction models. *ACM/IEEE International Conference on Software Engineering*, pp. 321–332.

Veeramachaneni, K., Arnaldo, I., Derby, O., and O'Reilly, U.-M. (2015). FlexGP: Cloud-based ensemble learning with genetic programming for large regression problems. *Journal of Grid Computing*, 13(3):391–407.

Verma, A., Llorà, X., Goldberg, D. E., and Campbell, R. H. (2009). Scaling genetic algorithms using MapReduce. In *International Conference on Intelligent Systems Design and Applications*, pp. 13–18.

Witten, I. H., and Frank, E. (2005). *Data mining: Practical machine learning tools and techniques*. Burlington, MA: Morgan Kaufmann.

Yoo, S., Harman, M., and Ur, S. (2011). Highly scalable multi objective test suite minimisation using graphics card. In *International Symposium on Search Based Software Engineering*, pp. 219–236.

Zheng, L., Lu, Y., Ding, M., Shen, Y., Guoz, M., and Guo, S. (2011). Architecture-based performance evaluation of genetic algorithms on multi/many-core systems. *IEEE International Conference on Computational Science and Engineering*, 321–334.