A Framework for Genetic Algorithms Based on Hadoop

Filomena Ferrucci*, M-Tahar Kechadi†, Pasquale Salza*, Federica Sarro‡

* Department of Management and Information Technology, DISTRA (MIT)
Università degli Studi di Salerno, Italy
e-mail: fferrucci@unisa.it

† School of Computer Science and Informatics
University College Dublin, Ireland
e-mail: tahar.kechadi@ucd.ie

‡ Department of Computer Science, CREST
University College London, United Kingdom
e-mail: f.sarro@cs.ucl.ac.uk

Abstract—Genetic Algorithms (GAs) are powerful metaheuristic techniques mostly used in many real-world applications. The sequential execution of GAs requires considerable computational power both in time and resources. Nevertheless, GAs are naturally parallel and accessing a parallel platform such as Cloud is easy and cheap. Apache Hadoop is one of the common services that can be used for parallel applications. However, using Hadoop to develop a parallel version of GAs is not simple without facing its inner workings. Even though some sequential frameworks for GAs already exist, there is no framework supporting the development of GA applications that can be executed in parallel. In this paper is described a framework for parallel GAs on the Hadoop platform, following the paradigm of MapReduce. The main purpose of this framework is to allow the user to focus on the aspects of GA that are specific to the problem to be addressed, being sure that this task is going to be correctly executed on the Cloud with a good performance. The framework has been also exploited to develop an application for Feature Subset Selection problem. A preliminary analysis of the performance of the developed GA application has been performed using three datasets and shown very promising performance.

Keywords—Parallel Genetic Algorithms, Hadoop, MapReduce, Metaheuristics

I. INTRODUCTION

For problems with non polynomial complexity the search for an optimum solution is considered to be a mission impossible, as it involves huge resources and execution time. Metaheuristic techniques, such as Genetic Algorithms (GAs), constitute the best alternative to find near-optimal solutions for such problems within a reasonable execution time and limited resources.

GAs approach mimics the biological process of reproduction. It starts with an initial population of individuals, each of which is represented by a chromosome. The GA iterates by executing some common operations on the selected individuals, which are crossover and mutation. The population tends to improve its individuals by keeping the strongest individuals and rejecting the weakest ones. However, one of the main drawbacks of the technique is how to model the real-world problem into a genetic one. The model, usually called coding, is far from being straightforward. In order to model an original optimisation problem into a genetic one, some main functions of GAs need to be defined properly. The fitness function should correspond to objective function of the original problem. This function will be used to evaluate the individuals. Usually the individuals with good fitness will be selected for next generation.

Often, GAs are executed on single machines as sequential programs. However, the main principle behind these algorithms is not really sequential, as it is possible to select more than two individuals for reproduction, use more than one population, and execute the operators in parallel. Parallel systems are becoming commonplace mainly with the increasing popularity of the Cloud Systems, GAs can be executed in parallel without changing their main principle. Currently, one available distributed platform is Apache Hadoop and its easy installation and maintainability are two key aspects that contributed to its great popularity. Nowadays it is common for an industry to rent a cluster on-line in order to request the execution of their applications as services.

All these elements add to the motivations described in this paper. elephant56 is a framework for developing GAs that can be executed on the Hadoop platform, following the paradigm of MapReduce. The main purpose of the framework is to completely hide the inner workings of Hadoop and allow users to focus on the main aspects of GA of their applications, being sure that their task will be correctly executed and with a good performance. In this way, the only concerns of users are the GA model and the settings of the key inputs, parameters and functions used in GAs (such as fitness, selection, initial population, etc...).

The intended goals of this paper are:

- Develop a complete framework that allows the user to develop and execute full applications;
- Provide some frequent ready “on-the-shelf” functions,

†The name “elephant56” combines two ideas: “elephant” resembles the Hadoop platform; “56” is the number of chromosomes of elephants citing the world of Genetics.
such as common criteria of selection and individuals representations that can be useful in most of the possible GAs implementations;

- Develop testing strategies to evaluate the performance of elephant56. For this reason a complete example of use of the framework is included in this paper.

The rest of paper is organised as follows. Section II shows the basis needed to understand the contents of this paper, with a quick overview of the existing literature about the subject. In section III the design and development of the framework are explained, by starting with the Driver component and finishing with the description of all components under two main profiles: the “core” and “user”. In section IV the framework is tested by developing a complete example of use called “Feature Selection Subset”, by explaining how the problem was adapted and showing the results and performance. Section V gives a final view of the achieved results suggesting possible future work to improve the framework.

II. BACKGROUND AND RELATED WORK

In the following it is briefly introduced the background of the theory and technologies behind the framework. Furthermore, it is described the developments and ideas of the existing literature in the same area of research.

A. Genetic Algorithms

As mentioned above Genetic Algorithms, described in [1] as a metaheuristic technique, can find reasonable solutions within reasonable time, for which exact techniques are unsuitable. GAs simulate several aspects of the “Darwin’s Evolution Theory” in order to enable them to converge towards optimal or near-optimal solutions. The main concept behind these metaheuristic algorithms is robustness, the balance between efficiency and efficacy necessary for survival in many different environments. Unfortunately, in real-world applications the search is fraught with discontinuities and vast multimodal, in such a way that the space of search is noisy as shown in figure 1. In order to move carefully through the space of search, heuristics are needed. Some widely accepted search procedures simply lack of this quality but this does not imply that these algorithms are not useful. Indeed, they have been used successfully in many applications but where the domain of search is limited.

- Work with a coding of the parameter set, not the parameters themselves;
- Search from a population of states, not a single point;
- Use the objective function (fitness function) information, not auxiliary knowledge;
- Use probabilistic transition rules, not deterministic rules.

In GAs the new individuals are generated by two parents selected among the current population individuals for so-called sexual reproduction.

![Figure 2. The execution of a Genetic Algorithm (Adapted from [2])](image)

As it can be seen in figure (2 (a)), GAs begin with a set of \( k \) randomly generated states called “population”. Each state is called “individual” and is represented as a string over a finite alphabet. This string is called “chromosome” and each symbol “gene”. Every iteration of the algorithm generates a new population and consists of the following steps:

- **Selection**: Each state is evaluated by the fitness function (b). Each value influences the random choice among the successors for the next step. Once chosen the \( k \) successors are grouped into couples (c);
- **Crossover**: The algorithm chooses for each individual a division point, which is called crossover point. At this point the sexual reproduction (d) in which two children are created begins: the first takes the first part of the first parent and the second of the second parent; the other takes the second part of the first parent and first part of the second parent;
- **Mutation**: When the offsprings are generated, each gene is subjected to a random mutation with a small independent probability (e).

Depending of the code used to represent the feasible solutions of a given problem, the representation of the genetic code can condition the crossover and mutation steps. For example, if the chromosome is composed of concatenated values, a simple division may not make any sense. The mathematical explanation of why GAs work, was given by the “Holland’s Schema Theorem” in [3], which says:

*Short, low order, above average schemata receive exponentially increasing trials in subsequent generations of a Genetic Algorithm.*

The concept of schema is a string (chromosome) in which some of the positions (genes) can be left unspecified. For example, with a binary alphabet representation, the schema 01 * 0 describes all chromosomes (instances of the schema) in which the first, the third and the fourth genes are fixed,
which are 0100 and 0110. Holland showed that if the average fitness value of the instances of a schema is above the average, the number of instances of the schema within the population will grow over time. This justifies the choice of selecting with more probability individuals that have higher fitness value and the need to use the mutation to shake things up. Moreover, it is shown that since each individual owns a large number of different schemas, during each generation the number of schemas implicitly processed are in the order of \( k^3 \), where \( k \) is the number of individuals. Understanding if GAs provide near-optimal solutions is the object of study of the “Building-Block Hypothesis” which has to be confirmed yet.

### B. Hadoop MapReduce

The term Hadoop [4] comprises a family of many related projects with the same infrastructure for distributed computing and large-scale data processing. It is better known for the MapReduce algorithm, shown below, and its distributed file system HDFS, which runs on large clusters of commodity machines. Hadoop 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, 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.

MapReduce is a programming model whose origins lie in the old functional programming. It was adapted by Google [5] as a system for building search indexes, distributed computing and database communities. It was written in C++ language and was made as a framework, in order to simply develop its applications. In Hadoop programs are mainly in Java language but it is also possible, through a mechanism called “streaming”, to develop programs in any language that supports the standard I/O. MapReduce is a batch query processor and the entire dataset is processed for each query. It is a linearly scalable programming model where users programs at least two functions: the “map” function and “reduction” functions. These functions process the data in terms of key/value pairs which are unaware of the size of the data or the cloud that they are operating on, so they can be used unchanged either for a small dataset or for a massive one.

![Hadoop MapReduce Diagram](image)

**Figure 3. Hadoop MapReduce**

A typical MapReduce program (figure 3) starts on a single machine using a piece of code called “driver”. This launches and manages the execution called “job” of the entire distributed program on the cluster. Then several components at different stages operate:

- **Splitter**: The input (a) is often in the form of a simple text, which consists of one or more files stored in the distributed file system HDFS. It undergoes a first treatment from the Splitter (b). Depending on the criteria used, it first creates the key/value pairs called “records” and sends them directly to an available Mapper on the cluster. The function, where \( k_1 \) and \( v_1 \) means data types, is described as:

  \[
  \text{split} : \text{input} \rightarrow \text{list} (k_1, v_1)_S
  \]

- **Mapper**: It is the first function of the core of Hadoop. A Mapper (c) runs on a machine of the cluster a process called “MapTask”. Once the input has been given, it produces a list of records according to the algorithm described by the programmer:

  \[
  \text{map} : (k_1, v_1)_S \rightarrow \text{list} (k_2, v_2)_M
  \]

- **Combiner**: It is also called “Local Reducer” and it is an optional component. The Combiner function does not replace the reduce function but it can help cutting down the amount of data exchanged between Mappers and Reducers. It (d) runs on the same machine that made the MapTask and it computes new pairs having the same key:

  \[
  \text{combiner} : (k_2, \text{list (v_2)})_M \rightarrow (k_2, v_2)_M'
  \]

- **Partitioner**: It (e) establishes the criteria by which records are assigned to a Reducer. This is also called the “shuffle operation” (f) and ensures that records with the same key will be assigned to the same Reducer. If not directly specified, the default Partitioner acts like a hash function on keys:

  \[
  \text{partition} : k_2 \rightarrow \text{hash} (k_2)
  \]

- **Reducer**: Finally, the Reducer (g) concludes the job. If a Partitioner with hash on keys has been used, it can process all the records with the same key created by the whole cluster:

  \[
  \text{reduce} : (k_2, \text{list (v_2)})_M \rightarrow (k_3, v_3)_R
  \]

Once the job has been completed, the last records are written one for each Reducer to the output files in HDFS (h).

### C. Related Work

The existent literature proposes some parallel version of GAs using the MapReduce paradigm. The first is an extension, by adding a second Reducer, of MapReduce named “MRPGA” [6] based on .Net. In this implementation a coordinator client manages the executions of the parallel GA iterations. The chosen model is the island model in which each participating node computes GAs operations for a portion of the entire population. In the first phase, each Mapper node receives its own portion of population and computes the fitness value for each of its individuals. The Reducer nodes of the first reduce phase receive the individuals of the correspondent island and apply the selection function. The final Reducer computes the global selection and the other following GAs functions.

Another approach, this time developed on Hadoop, is presented by [7]. The number of Mapper nodes and the one of the Reducer nodes are unrelated. The Mapper nodes computes the fitness function and the Reducer a local selection followed by the other GAs functions. The substantial difference with
MRPGA lies in the fact that the partitioner supplies a sort of “migration” among the individuals as it randomly sends the outcome of the Mapper nodes to different Reducer nodes.

In [8] the three main grain parallelism are described and implemented by exploiting the MapReduce paradigm:

1. **Fitness Evaluation Level** (Global Parallelisation Model);
2. **Population Level** (Coarse-grained Parallelisation Model or Island Model);
3. **Individual Level** (Fine-grain Parallelisation Model or Grid Model).

In the Global Parallelisation Model (figure 4) a master node manages the population and computes for them all the GAs functions but the fitness evaluation is computed by the slave nodes. The model is adapted to MapReduce by delegating some Mappers the task of evaluating the fitness value for each individual in parallel. Then, the single Reducer collects the results and performs the other GA operations. One generation corresponds to one MapReduce execution, so that the whole computation is a sequence of MapReduce executions.

In the Coarse-grained Parallelisation Model (figure 5) the population is subdivided into “islands” and the GA is independently run on each of them. Periodically the islands exchange information by “migrating” some individuals. Here the number of Reducers is higher than in the previous model. After having computed the fitness values in the Mappers, a Partitioner assigns each island to a different Reducer in order to compute the other GAs functions in parallel.

In the Fine-grain Parallelisation Model (figure 6) each individual is placed on a grid and the GA operations are performed in parallel by evaluating simultaneously the fitness value and applying the selection limited only to the small adjacent neighbourhood. This model is slightly adapted by modifying the previous one: the Partitioner uses a pseudo-random function in such a way that the described local neighbourhood is developed.

The Global Parallelisation Model has been implemented by [8] in order to solve a problem of Automatic Test Data Generation. It has been developed on the Google App Engine MapReduce platform. Also serving the same purpose, [10] has developed Coarse-grained Parallelisation Model on the Hadoop MapReduce platform.

The framework presented in this paper mainly exploits the Coarse-grained Parallelisation Model, without differing so much from the implementation proposed by [8]. It introduces some new modifies that better suit its intrinsic nature of framework.

III. DESIGN AND DEVELOPMENT

Here the design and development of the single involved components are explained. It starts from the last level of abstraction, which is the Driver component, and goes into depth as far as the Hadoop development of the components.

A. Driver

The Driver is the main part of the framework and it represents the only interface between the two involved parts in each work, managing the two main aspects:

- The interaction with the user;
- The launch of the jobs on the cluster.

The Driver is executed on a machine even separated from the cluster and it also computes some functions as soon as the first results are received from the cluster.

Figure 7 describes the elements of the whole process every time it is executed, each one managed by the Driver process:

- **Initialiser**: This component (a) can be optional if there already exists a population. The user can define how to generate the first individuals for each island. The default component makes it randomly. The Initialiser is a MapReduce job which produces the individuals in the form of a sequence of files, stored in a serialised form directly in HDFS (b);
- **Generator**: It is the heart of the framework (c). It executes one generation on the cluster and produces
the new population, storing each individual again in HDFS within the fitness values and a flag indicating if an individual satisfies the termination criterion.

- **Terminator**: At the end of each generation, the Terminator component (d), which does not work as a job, checks the stopping conditions (e.g., if the maximum number of generations has been reached). Once terminated, the population is directly submitted to the SolutionsFilter job (f).

- **Migrator**: This optional job (e) allows moving individuals from an island to another, according to the criteria defined by the user such as the frequency of migration, number and destinations of migrants and the selection method for choosing migrants.

- **SolutionsFilter**: When the job terminates, all the individuals of the last generation are filtered according to those that satisfy the termination criterion and those which do not. Then, the results of the whole process is stored in HDFS (g).

### B. Generator

Each generator job makes the population evolve. In order to develop the complex structure described below, it is needed to use multiple MapTasks and ReduceTasks. This is possible using a particular version of ChainMapper and ChainReducer classes of Hadoop, slightly modified in order to treat Avro objects. Hadoop manages the exchange of information among the tasks with a raw method of serialisation. Using Avro, it is easy to store objects and to save space on the disk, allowing an external treatment of them. It also permits a quick exchange data among the parties involved in the MapReduce communication.

A chain allows to manage the tasks in the form described by the pattern:

\[(MAP)^+ (REDUCE) (MAP)^*\]

which means one or more MapTasks, followed by one ReduceTask and other possible MapTasks.

Figure 8 describes how the generation work is distributed on the cloud:

- **Splitter**: The Splitter (b) takes as input the population (a), deserialises it and splits the individuals into J groups (islands) (c), according to the order of individuals. Each split contains a list of records, one per each individual:

  \[\text{split} : \text{individuals} \rightarrow \text{list } (\text{individual}, \text{NULL})\]

  During the deserialisation, the splitter adds some fields to the objects which will be useful for the next steps of the computation, such as the fitness function.

- **Fitness**: Here (d) according to J islands, the J Map- pers compute the fitness values for each individual within its corresponding island for which it has not been calculated yet:

  \[\text{map} : (\text{individual}, \text{NULL}) \rightarrow (\text{individual}, \text{NULL})\]

  The user defines how to the fitness is evaluated and the values are stored inside the corresponding field of the objects.

- **TerminationCheck**: Without leaving the same machine of the previous map, the second map (e) acts in a chain. It checks if the current individuals satisfy the termination criterion. This is useful, for instance, when a lower limit target of the fitness value is known. If at least one individual gives a positive answer to the test, the event is notified to the other phases and islands by a flag stored in HDFS. This avoids the executions of the next phases and generations.

- **Selection**: If the termination criterion has not been satisfied yet, this (f) is the moment to choose the individuals that will be the parents during the crossover for the next iteration. The users can define this phase in their own algorithms. The couples which have been selected are all stored in the key:

  \[\text{map} : (\text{individual}, \text{NULL}) \rightarrow (\text{individual}, \text{NULL})\]
not chosen if the elitism is active, leave the current worker and go to the correspondent Reducer for the next step (g).

- **Crossover:** In this phase (h), the individuals are grouped by the couples established during the selection. Then each Reducer applies the criteria defined by the user and makes the crossover:

\[
\text{reduce} : (\text{couple\_information, list (individuals)}) \rightarrow (\text{individual, TRUE})
\]

This produces the offspring (marked with the value TRUE in the value field) that is read, together with the previous population, during the next step.

- **Mutation:** During the mutation (i), the chained Mappers manage to make the mutation of the genes defined by the user. Only the offspring can be mutated:

\[
\text{map} : (\text{individual, TRUE}) \rightarrow (\text{individual, NULL})
\]

- **Elitism:** In the last phase (j), if the user chooses to use elitism, the definitive population is chosen among the individuals of the offspring and the previous population:

\[
\text{map} : (\text{individual, NULL}) \rightarrow (\text{individual, NULL})
\]

At this point (k), the islands are ready to be written in HDFS (l).

The architecture of the generator component provides two levels of abstraction (see figure 9):

- The first one, which is called the “core” level, allows the whole job to work;
- The second one, which is called the “user” level, allows the user to develop his own Genetic Algorithm and to execute it on the cloud.

The core is the base of the framework with which the end-user does not need to interact. Indeed, the fact that a MapReduce job is executed is totally invisible to the user. It consists of everything that is needed to run an application on Hadoop. The final user can develop his own GA simply by implementing the relative classes, without having to deal with map or reduce details. If the user does not extend these classes, a simple behaviour is implemented by doing nothing else than forwarding the input as output. The framework also makes some default classes available so that the user can use them for most cases.

C. **Terminator**

The Terminator component (figure 10) plays two roles:

- After the execution of every generation job, by calling the methods of the Terminator class on the same machine where the Driver is running;
- During the generator job, through the use of the TerminationCheckMapper.

It checks if the stopping conditions have occurred:

- the count of the maximum number of generations has been reached;
- at least one individual has been marked of satisfying the termination criterion during the most recent generation phase.

Figure 8. The Generator component

![Diagram of the Generator component](image)

Figure 10. The two levels of abstraction of the Terminator component

![Diagram of the Terminator component](image)
The count is maintained by storing a local counter variable that is updated after the end of each generation. The check for the presence of marked individuals is done by looking for possible flags in HDFS. If it terminates, the execution of the SolutionsFilter component will eventually follow.

**D. Initialiser**

The Initialiser (see figure 11) computes an initial population. This is an optional component, because the entire work can start even if the population data are already present inside HDFS.

The class InitialiserMapper generates the next individual according to the user’s definition.

**E. Migrator**

The optional Migrator component (figure 12) shuffles individuals among the islands according to the user’s definition. It is at the same time a local component and a job executed on the cloud.

It is started by the Driver and on the frequency counter, chooses if it is time to perform a migration.

**F. SolutionsFilter**

The component SolutionsFilter (figure 13) is invoked only when at least one individual has provoked the termination by satisfying the termination criterion. It simply filters the individuals of the last population by dividing those that satisfy the criterion from those which do not.

**IV. Preliminary Analysis**

In order to test the performance of the framework, a real-world problem has been used, recording results and performance. The problem of classification in Machine Learning consists of learning from well-known example data, called “training dataset”, with the purpose of being able to properly classify any new input data. The existing algorithms of Machine Learning act like predictors on the new data. Therefore it is important to have a good level of “accuracy” of the prediction. One way of improving its performance is to find an optimal dataset, resulting from the original training dataset. This problem is known as “Feature Subset Selection”. Unfortunately, the search for the optimum is a NP-Hard problem. Genetic Algorithms have been used to model and look for near-optimal classes for the problem.

**A. Feature Subset Selection**

The training dataset includes a list of records, also called “instances”. Each record is a list of “attributes” (features), in which one attribute is the class attribute. Given a training dataset, the next step is to build a classifier. For instance, the C4.5 algorithm can be used in order to build a Decision Tree, which is able to give a likely class of ownership for every record in the new dataset. The effectiveness of the classifier is measured by the accuracy of the classification of the new data:

\[
\text{accuracy} = \frac{\text{correct classifications}}{\text{total of classifications}}
\]
Sometimes, it is possible to have the same accuracy or even a better one with a classifier that considers a lower number of attributes rather than one with more, as explained in [12].

The search for an optimal feature subset is not only important for building the classifier quickly and with very good response time, but also in terms of real costs. For example, let the problem to be solved be the identification of the presence of a certain disease. The training dataset is a collection of real medical records, where some features are cost-free information about the patient, whereas others are expensive test results such as blood test, DNA test, etc. In such a context, it is clear that saving the number of collected features means saving money.

Once outlined the advantages of reducing the number of features, describing the problem under a mathematical point of view is needed. Having a training dataset with \( m \) different attributes, except for the class attribute which has to be always present, the number of possible subsets is \( 2^m \). Let \( C \) be the execution time in order to build a classifier, which has received a dataset, and let \( A \) be the execution time to compute the accuracy of the resulting classifier, the resulting running time to find the best subset would be:

\[
O(2^m)AC
\]

As a consequence, the use of a GA may simplify the search within the space of solutions and give the proper way to look for good near-optimal solutions.

The resulting model is an adaptation of the application will be executed by the framework where each part is modelled based on the Feature Subset Selection. The driver is the main part of the algorithm and is executed on one machine. Moreover, who is going to use the algorithm must specify for the execution the following information:

- The generic arguments for the GA execution, such as how many individuals initially to generate, the maximum number of generations, etc.;
- The training dataset;
- The test dataset.

There is also the possibility of specifying the accuracy target in order to terminate the process before reaching the maximum number of generations, if this lower limit has been reached during the computation.

Since computing the accuracy for all the possible subsets needs exponential time, the idea is to submit a randomly generated initial group of attribute subsets, making the initial population, to the algorithm generations. During each generation, every subset is evaluated by computing the accuracy value and all the GAs functions are applied until target accuracy is achieved or maximum number of generations is reached. The control of the satisfying the termination criterion is controlled during the termination phase. At the end, the last population is ready to be tested with the test dataset.

By giving the training dataset as input, the first applied operation is the initialisation. The algorithm generates the \( r \) random attribute subsets which will be the initial population of individuals. Every individual (subset) is encoded as an array of \( m \) bit, where each bit shows if the corresponding enumerated attribute is present into the subset (value 1) or not (value 0).

Since the records in the training dataset are never altered during the whole algorithm, it is not necessary to encode them. They will be available when needed for the generations.

Every generation phase processes all the individuals within the population, according to the following steps:

- **Fitness**: For each subset of attributes, the training dataset is filtered to respect the attributes in the subset. In such a way, the fitness value is computed by applying the steps:
  1) Select the current portion of the dataset that is going to act as training dataset and the portion as test dataset;
  2) Build the Decision Tree through the C4.5 algorithm and computing the accuracy by submitting the current dataset;
  3) The operations are repeated according to the folding parameter, following the technique of the “Crossing Folding”;
  4) The best accuracy is returned.

- **Selection**: It chooses which individuals will be the parents during the crossover. It is important to give the individuals with the best accuracy the best probability to be chosen. The algorithm uses the method of the “roulette-wheel selection”: it builds a wheel according to the fitness values (accuracy) of each individual, after which it will turn for every new couple in such a way as to choose who will form it.

- **Crossover**: At this step, the new offspring is produced splitting the parents of each couple into two parts, according to a random crossover point, and then mixing the parts obtaining two new children, which have one part of the mother and one of the father.

- **Mutation** According to a probability to mutate, during this step each subset may change the attributes into itself.

- **Elitism** Optionally enabled, this step allows to choose the best individuals among the ones in the new offspring and in the previous generation, in such a way as to guarantee the growth of the accuracy target after every generation.

Since the algorithm is executed on different islands, it will be important to give a variability factor for each of them. The Migration manages the passage of a certain number of randomly selected individuals from an island to another.

### B. Subject

Three example dataset were given, referring to real problems. They are all coming from the UCI Machine Learning Repository\(^2\). Everything was submitted with little differences of parameters in each specific case, but all with two options of elitism active and not. Since these experiments aim is to analyse the behaviour of the algorithm during a full computation, the target accuracy was not specified to let the algorithm be executed until the maximum number of generations be reached.

\(^2\)UCI Machine Learning Repository
http://archive.ics.uci.edu/ml/
Each dataset was divided into two parts:

- The first 60% as training dataset;
- The last 40% as test dataset, eventually used to compute the obtained best accuracy.

The test bench was composed by three versions:

- **Sequential**: A single machine executes a GA similar to the framework one by using the Weka\(^3\) Java library.
- **Pseudo-distributed**: Here the framework version of the algorithm make the scene. It is executed on a single machine again, setting the number of islands to one. It requires Hadoop in order to be executed.
- **Distributed**: The framework is executed on a cluster of computers on Hadoop platform. This is the version of most interest.

All the versions execute on a remote Amazon EC2 cluster. This was chosen in order to give a factor of fairness to all the solutions.

The single machine of the sequential version was:

| Type     | Arch  | CPUs | RAM (GB) | Storage (GB) |
|----------|-------|------|----------|--------------|
| m1.large | 64-bit| 2    | 7.5      | 2 x 420      |

Although it run a sequential algorithm, the pseudo-distributed version needed a Hadoop installation:

| Hadoop Role     | Type     | Arch  | CPUs | RAM (GB) | Storage (GB) |
|-----------------|----------|-------|------|----------|--------------|
| All Hadoop roles | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |

On the other hand, the distributed version needed a full Hadoop cluster:

| Hadoop Role                  | Type     | Arch  | CPUs | RAM (GB) | Storage (GB) |
|------------------------------|----------|-------|------|----------|--------------|
| JobTracker, NameNode         | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |
| TaskTracker, DataNode        | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |
| TaskTracker, DataNode        | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |
| TaskTracker, DataNode        | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |
| TaskTracker, DataNode        | m1.large | 64-bit| 2    | 7.5      | 2 x 420      |

### C. Results

It is interesting analysing some aspects from the results. These consist of a measure of the effort in developing the application for Feature Subset Selection with the framework and other aspects regarding the performance.

1) Developing Effort: A total of 5811 lines of code were written during the development of the whole framework where 1903 include the test classes, as shown in figure 14.

For the development of the application (figure 15) of Feature Subset Selection were written 535 lines of code against the 3908 of the framework infrastructure.

3\(^3\)Weka is a collection of Machine Learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualisation. It is also well-suited for developing new Machine Learning schemes.

http://www.cs.waikato.ac.nz/ml/weka/

Figure 14 shows that all the versions act more or less in the same way. It is more important to consider the next parameter.

2) Number of attributes: It is the number of attributes of the best individual after the final generation. This parameter is directly referred to the specific problem, because it was one of the target to achieve. As described before, the lower the value is, the better it is.

Figure 16 shows that all the versions act more or less in the same way. It is more important to consider the next parameter.

3) Accuracy: The accuracy is the value that was computed just after the execution of each algorithm. It was obtained by submitting a common test dataset to the resultant new training dataset filtered through the subset of attributes found at the end.

By looking at figure 17, again the three versions do not have substantial differences. The upside is that the framework achieves its objective, by giving a subset that has both a reduced number of attributes and an accuracy that still suits the initial one.

4) Running time: While the number of attributes and accuracy give a measure of efficacy of the distributed version,
the running time weighs against the different versions of the algorithms at the time of choosing which one to use.

V. CONCLUSIONS AND FUTURE WORK

The test was preliminary for some reasons:

- Defeating the sequential version of Weka, which is specialised in Machine Learning algorithms, is a challenge itself;
- The probability factor of Genetic Algorithms needs an average of a greater number of executions of the test in order to be more reliable.

The results obtained in the example of use of the framework show that writing GAs with the framework is effective because of the good ratio of number of lines of code for developing the application to the ones of the framework itself.

Even though the running time is not always on its side, some clues suggest that an “ad hoc” optimisation of Hadoop might improve considerably the performance. The case in which the sequential version is defeated suggests that for large size instances it is worth parallelising.

Nowadays it is possible to rent a cluster for little money without investing on an expensive and dedicate hardware. In a few minutes everything is completely operative so as to run any type of distributed applications. This fact together with the saving of money, the easiness with which it is possible to develop the algorithms and its flexibility, give this framework a relevant meaning. For all these reasons, it can be worth improving it in the future. It can be advantageous to convert the project into an open-source project. The current complexity of the framework needs a specific care in the discovery of possible bugs and the potential interest covers the prospective of the existence of a dedicate community.

Even though many of the most common basic functions are already implemented, it could be useful to add some other implementations to better cover the possible needed cases. Moreover, some tools to treat the data read and produced by the framework could help make it more compatible with external applications.

The most important improvement needed is analysing the possible bottlenecks caused by the inner structure of Hadoop. As the features of Hadoop that can treat data in a more efficient way are many, it could be worth considering them in a future development.

REFERENCES

[1] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc., 1989.
[2] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Third, Prentice Hall, 2010.
[3] J. H. Holland, Adaptation in Natural and Artificial Systems. The University of Michigan Press, 1995.
[4] T. White, Hadoop: The Definitive Guide, Third. O’Reily, 2012.
[5] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters”, OSDI, 2004.
[6] C. Jin, C. Vecchiola, and R. Buyya, “Mrpga: an extension of mapreduce for parallelizing genetic algorithms”, in eScience, 2008. eScience ’08. IEEE Fourth International Conference on, IEEE, 2008, pp. 214–221.
[7] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell, “Scaling genetic algorithms using mapreduce”, in *Intelligent Systems Design and Applications, 2009. ISDA’09. Ninth International Conference on*, IEEE, 2009, pp. 13–18.

[8] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro, “Towards migrating genetic algorithms for test data generation to the cloud”, in *IGI Global*, 2012, ch. 6, pp. 113–135.

[9] G. Luque and E. Alba, *Parallel Genetic Algorithms, Theory and Real World Applications*. Springer, 2011.

[10] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, “A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites”, in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, IEEE, 2012, pp. 785–793.

[11] D. Cutting, *Data interoperability with apache avro*, Cloudera Blog, 2011. [Online]. Available: http://blog.cloudera.com/blog/2011/07/avro-data-interop.

[12] M. A. Hall, “Correlation-based feature selection for machine learning”, PhD thesis, The University of Waikato, 1999.