elephant56: Design and Implementation of a Parallel Genetic Algorithms Framework on Hadoop MapReduce

Pasquale Salza[1], Filomena Ferrucci[1], Federica Sarro[2]
[1]University of Salerno, Italy – [2]University College London, United Kingdom
psalza@unisa.it, fferrucci@unisa.it, f.sarro@ucl.ac.uk

ABSTRACT

elephant56 is an open source framework for the development and execution of single and parallel Genetic Algorithms (GAs). It provides high level functionalities that can be reused by developers, who no longer need to worry about complex internal structures. In particular, it offers the possibility of distributing the GAs computation over a Hadoop MapReduce cluster of multiple computers. In this paper we describe the design and the implementation details of the framework that supports three different models for parallel GAs, namely the global model, the grid model and the island model. Moreover, we provide a complete example of use.

CCS Concepts

•**Computing methodologies** → **Genetic algorithms;** *Massively parallel algorithms;* •**Networks** → **Cloud computing;**

Keywords

Parallel Genetic Algorithms; Hadoop MapReduce; Cloud Computing

1. INTRODUCTION

Genetic Algorithms (GAs) are a powerful technique used in problems where the search for an optimal solution is expensive and we aim to find at least a near-optimal solution. Although attractive and elegant in laboratory, GAs are usually executed as sequential programs and therefore scalability issues may prevent their effective application to real-world problems. Nevertheless, parallelisation is a suitable way to improve the performance in terms of computational time because GAs are 'naturally parallelisable' [7]. For instance, their population based characteristics allow evaluating in a parallel way the fitness of each individual (i.e., global parallelisation model). Parallelism can also be exploited to perform genetic operators and thus to generate the next set of solutions (i.e., island model). Furthermore, these two strategies can be combined, giving rise to a third form of parallelisation (i.e., grid model). All these algorithms are known in literature as Parallel Genetic Algorithms (PGAs).

It is argued that a barrier to wider application of parallel execution has been the high cost of parallel architectures and infrastructures and their management. Cloud computing can represent an affordable solution to address the above issues because it breaks the barrier between employed resources and costs: In few seconds it is possible to allocate a cluster of the desired size without investing in expensive local hardware and its management. At present, Hadoop MapReduce is a solid and valid presence in the world of cloud computing. MapReduce is a programming paradigm whose origins lie in the old functional programming. It was adapted by Google [1] as a system for building search indexes, distributed computing and large scale databases and later implemented in the Hadoop platform as part of the Apache Software Foundation family. Hadoop was created by Doug Cutting and has its origins in Apache Nuts, an open source web search engine. In January 2008 it became a top-level Apache project, attracting to itself a large active community, including Yahoo!, Facebook and The New York Times. In combination with the Hadoop Distributed File System (HDFS), Hadoop MapReduce can run on large clusters of machines with some interesting features such as 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 cloud platforms (e.g., Amazon Elastic Compute Cloud) and thus is an ideal candidate for high scalable parallelisation of GAs.

In this paper we present elephant56[1], an open source framework[2] supporting the development and execution of parallel GAs. Our aim is to provide a framework to encourage users to develop their own GAs, explicitly controlling the data types and genetic operators involved, and executing them on a distributed Hadoop cluster, without dealing with the parallelisation part of GAs (i.e. dealing with Hadoop MapReduce) thus allowing developers to focus on the GAs definition only. The framework allows also the execution of a sequential GA on a single commodity machine.

elephant56 is the extension of the proposal by Ferrucci

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931722

[1]The name 'elephant56' combines two ideas: 'elephant' like the Hadoop mascot and '56', which is the number of chromosomes of the elephant.

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

et al. [5, 6], which in its first version provided support only for the island model and the source code was not publicly available. To the best of our knowledge, elephant56 is the first publicly available framework based on Hadoop MapReduce.

The rest of the paper is organised as follows. Section 2 describes related work and Section 3 provides an overview of the technologies and concepts involved. Section 4 presents the algorithms we devised to implement the sequential and parallel GAs. Next, in Section 5 we illustrate a complete example of use of the framework by implementing the OneMax problem. In Section 6, the architecture of the framework is described whereas Section 7 contains some final remarks and future work.

2. RELATED WORK

In this section we describe some of the relevant work that inspired and guided our study, highlighting the difference with them.

In the literature GAs have been parallelised with different approaches, methods and technologies. Jin et al. [9] were the first to use MapReduce paradigm to this aim. They implemented their own version of MapReduce on .Net platform and realised a parallel model which can be considered as a hybrid of models described in Section 3.2: The mapper nodes compute the fitness function and a selection chooses the best individuals on the same machine; the single reducer applies the selection on all the best local individuals received from parallel nodes; the computation continues on the master node where crossover and mutation operators are applied on the global population.

The first work on Hadoop MapReduce is from Verma et al. [11]. The implemented model is the grid model: The mappers execute the fitness evaluation and the unpaired reducers execute the other genetic operators on the individuals they randomly receive as input. This work is important mainly because the authors studied the scalability factor on a large cluster of Hadoop nodes finding a clear decrease in performance only when the number of requested nodes surpassed the number of physical CPU available on the cluster. They confirmed that GAs can scale on multiple nodes. Huang and Lin [8] exploited Hadoop MapReduce on a private large grid of slow machines to measure the performance in terms of fitness value of the solutions at scaling the number of nodes. They also exploited a real Amazon EC2 cluster of faster machines in order to analyse the execution time factor. They suggested the use of Hadoop MapReduce in GA parallelisation in presence of large populations and intensive computation work for fitness evaluation.

Zheng et al. [12] compared the multi-core system with the many-core system on GPUs in parallelising GAs. They implemented both the global and island models, finding that the island model was preferable in terms of quality and execution time because it reduced the parallel communication. Even though they found the system based on GPU is faster than the many-core one, they observed that an architecture with a fixed number of parallel participants, such as GPU cores, might perform worse in terms of quality of solutions than another with more parallel nodes. They explicitly affirmed that a distributed architecture is worth for GAs parallelisation.

Di Gironimo et al. [2] were the first to propose a parallel GA for automatic generation of JUnit test suite based on the global parallelisation model. A preliminary evaluation of the proposed algorithm was carried out aiming to evaluate the speedup with respect to the sequential execution. The obtained results highlighted that using the parallel genetic algorithm allowed for saving over the 50 % of time. The algorithm was developed exploiting Hadoop MapReduce and its performance were assessed on a standard cluster. Subsequently, Di Martino et al. [3] proposed some solutions to migrate GAs to the cloud, aiming at obtaining automatic test data generation. This migration is motivated by the need to achieve a greater efficiency and scalability of this Search-Based technique for automated software testing, thus possibly reinforcing its effectiveness thanks to the available computational power of the cloud. The authors suggested the use of the MapReduce paradigm, relieving programmers from most of the low level issues in managing a distributed computation. This is also motivated by the fact that MapReduce is natively supported by several cloud infrastructures. In particular, they suggested how to adapt the three parallelisation models to MapReduce paradigm with the aim to automatically generate test data and discussed issues and concerns about them. They also implemented a solution based on the 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.

Pushed by the motivation of solving a problem of Symbolic Regression by using Genetic Programming, Fazenda et al. [4] were the first to consider the parallelisation of Evolutionary Algorithms (EAs) on Hadoop MapReduce platform in a general purpose form of a library, in order to simplify the developing effort for parallel EA implementations. With the same aim, Ferrucci et al. [5, 6] implemented a framework for PGAs development, deployment and execution on Hadoop MapReduce platform, based on island model. They described the design of the framework and how a developer could interact in defining his/her own 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. In this work we extended this framework by implementing also the global and grid models and making it available to the community as an open source project. Moreover, we improved the implementation of the island model because the previous one executed a job for each generation whereas we introduced the concept of periods of generations in order to speedup the PGA based on this model.

3. BACKGROUND

In this section we give some background about the involved technologies and the concepts needed to follow the rest of the paper.

3.1 Hadoop MapReduce

The MapReduce paradigm is based on two distinct functions, namely 'map' and 'reduce', which are combined together in a divide-and-conquer way where the map function is responsible to handle the parallelisation while the reduce collects and merges the results.

A Hadoop cluster is allowed to accept MapReduce executions (i.e., 'jobs') in a batch fashion. Usually, a job is demanded from a master node which provides both the data and configuration for the execution on the cluster. A job is intended to process input data and produce output data

exploiting a distributed file system (an open source implementation of Google File System), named 'HDFS'. HDFS is also used for intermediate data. A job is composed of the following main phases:

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:

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

A developer is expected to extend some specific Java classes in order to define each job phase.

Hadoop is able to move data between nodes through sequences of write and read operations to/from the HDFS. The default raw serialisation of objects is inefficient if compared to 'Avro'[3]. It is a modern data serialisation system from the same creator of Hadoop, Doug Cutting. In addition to have a flexible data representation, it is optimised to minimise the disk space and communication through data compression.

3.2 Parallel Models for Genetic Algorithms

The following models have been proposed in literature [10] to parallelise the execution of Genetic Algorithms:

- 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 roles: a 'master' node and one or more 'slave' nodes. The former is responsible to manage the population (i.e., apply genetic operators) and to assign the individuals to the slave nodes. The latter are in charge to evaluate the fitness for each individual. This model does not require any changes to the sequential GA, since the fitness computation for each individual is independent and thus achievable in parallel.

The grid model applies the genetic operators only to portions of the global population. This is obtained by assigning each individual to a single node and by performing evolutionary operations that involve also some neighbours of a solution, according to a defined criterion of neighbourhood. The effect is an improvement of the diversity during the evolutions, further reducing the probability to converge into a local optimum, with the drawback of requiring higher network traffic due to the frequent communications among the nodes.

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

4. ALGORITHMS

In the following we first explain the design of a Sequential Genetic Algorithm (SGA) and then we illustrate how proposal to map the MapReduce elements for each of three GA parallel models.

4.1 Sequential Genetic Algorithm (SGA)

There are several possible versions of GA execution flows. The parallel adaptations are built on the base of the following SGA implementation which is composed of a sequence of genetic operators repeated generation by generation, as described in Algorithm 1.

Algorithm 1 Sequential Genetic Algorithm (SGA)

```
1:  population ← Initialization(populationSize)
2:  for i ← 1, n do
3:    for individual ∈ population do
4:      FitnessEvaluation(individual)
5:    if elitism is active then
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:   if survival selection is active then
15:     for individual ∈ offspring do
16:       FitnessEvaluation(individual)
17:     population ← SurvivalSelection(population, offspring)
18:   if elitism is active then
19:     population ← population + elitists
```

The execution flow starts with an initial population initialised with the Initialization function (1), which can be a random function or a specific one based on other criteria. Then, at each generation the first genetic operator applied is the FitnessEvaluation (2-4) which evaluate and assign a fitness value to each individuals letting them to be comparable. The Elitism operator (5-7) is optional and it allows to add some individuals directly to the next generation (18-19). TheParentsSelection 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, which is optional, applies a selection between parents and offspring individuals (14-17) to select

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

the individuals that will take part of the next generations. The PGAs proposed 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).

4.2 PGA for the Global Model

The PGA that implements the global model on MapReduce has the same behaviour of the sequential version, but it resorts to parallelisation for the fitness evaluation. Figure 1 shows the workflow of the model. The master node, also referred as `Driver`, initialises a random population and writes it in the HDFS. During each generation, it spreads the individuals to the slave nodes in the cluster when: (i) the initial population is evaluated for the first time; (ii) the generated offspring needs to be evaluated in order to apply the survival selection to both parents and children. This means that a single job is needed at each generation. The `Driver` also executes sequentially the other genetic operators on the entire population that has been evaluated.

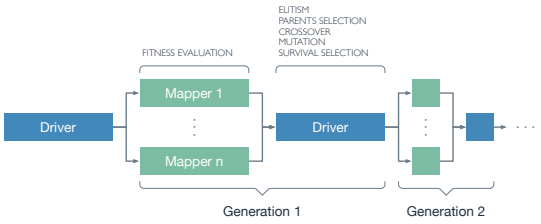

Figure 1: The workflow for the global model.

More in details, the slave nodes in the cluster perform only the fitness evaluation operator. The mappers receive the records in the form (individual, destination). The 'destination' field is used only by the other models, so it will be mentioned later. We deliberately disabled the reduce phase, because there is no need of moving individuals between nodes. After the map phase, the master reads back the individual and continues with the other remaining genetic operators, considering the whole current population.

4.3 PGA for the Grid Model

The PGA that implements the grid model applies the genetic operators only to portions of the population called 'neighbourhoods'. In the grid model these portions are chosen randomly at each generation (Figure 2). The number of jobs is the same as the number of generations.

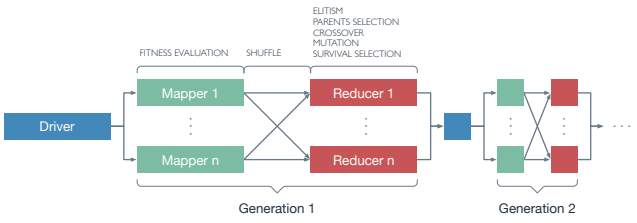

Figure 2: The workflow for the grid model.

The `Driver` has the task of randomly generate a sequence of neighbourhoods destinations for the individuals in the current population. These destinations are stored into the record as the value fields so the destinations are known a priori. We exploited the parallelisation in two phases: (i) the mappers initialise a random population during the first generation and computes the fitness evaluation; (ii) 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.

4.4 PGA for the Island Model

The PGA for the island model acts similarly to the one for the grid model because it operates on portions of the global population called 'islands'. Each island executes whole periods of generations on its assigned portions, independently from the other islands until a migration occurs (Figure 3). This means there is an established migration period, which can be defined as the number of consecutive generations before a migration. Since it is possible to run groups of subsequent generations (i.e., periods) independently, we exploited a MapReduce job for each period.

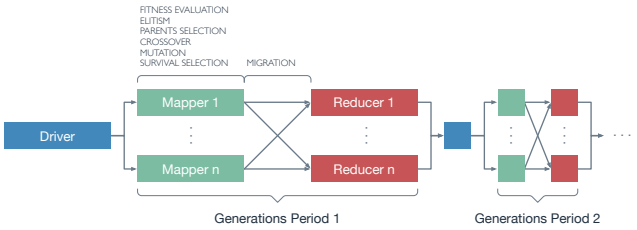

Figure 3: The workflow for the island model.

In Hadoop, the numbers of mappers and reducers are not strictly correlated, but we coupled them in order 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 individuals will have a specific destination island: This 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.

5. USING ELEPHANT56

In this section we explain how to use elephant56 through a running example based on the simple problem of 'OneMax' (also know as 'BitCounting'), which consists in maximising the number of 1 in a bit string. Even though some sample implementations are provided with the framework already (e.g., individuals and genetic operators), we are going to be intentionally redundant here for explanatory purposes. To the same aim, we are going to deliberately illustrate some solutions that may be inefficient. Moreover, in order to improve the code readability we do not strictly follow the Java programming language and postpone some details to Section 6 .

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

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

To solve this problem with GAs, the individuals can be

represented as bit strings and the above equation can be used as fitness function. We also need to define the genetic operators. elephant56 allows the developer to define each of these elements by extending the classes of the framework. Because there exists an underlying distributed platform (i.e., Hadoop MapReduce), many of the objects are encapsulated into some wrapper objects which ease the serialisation process.

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

```
public abstract class Individual implements Cloneable {
  public abstract Object clone() throws
      ↪ CloneNotSupportedException;
  public abstract int hashCode();
}
```

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

```
public class BitStringIndividual extends Individual {
  private List<Boolean> bits;

  public BitStringIndividual(int size) {
    bits = new ArrayList<>(size);
  }

  public void set(int index, boolean value) {
    bits.set(index, value);
  }

  public boolean get(int index) {
    return bits.get(index);
  }

  public int size() {
    return bits.size();
  }
  ...
}
```

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

```
public abstract class FitnessValue implements
    ↪ Comparable<FitnessValue>, Cloneable {
  public abstract int compareTo(FitnessValue other);
  public abstract Object clone() throws
      ↪ CloneNotSupportedException;
  public abstract int hashCode();
}
```

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

```
public class IntegerFitnessValue extends FitnessValue {
  private int number;

  public IntegerFitnessValue(int value) {
    number = value;
  }

  public int get() {
    return number;
  }

  @Override
  public int compareTo(FitnessValue other) {
    if (other == null)
      return 1;
    Integer otherInteger = ((IntegerFitnessValue) other
        ↪ ).get();
    Integer.compare(number, otherInteger);
  }
  ...
}
```

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

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

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

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

```
public class OneMaxFitnessEvaluation extends
    ↪ FitnessEvaluation<BitStringIndividual,
    ↪ IntegerFitnessValue> {
  ...
  @Override
  public IntegerFitnessValue evaluate(IndividualWrapper
      ↪ <BitStringIndividual, IntegerFitnessValue>
      ↪ wrapper) {
    BitStringIndividual individual = wrapper.
        ↪ getIndividual();
    int count = 0;
    for (int i = 0; i < individual.size(); i++)
      if (individual.get(i))
        count++;
    return new IntegerFitnessValue(count);
  }
}
```

At this point, we need to define the genetic operators (i.e., crossover, mutation, selection). Given that standard operators are already provided by the framework, in the following we will describe in details only the definition of those genetic operators that require a specific implementation in order to manage the classes we have defined so far. The procedure to define a genetic operator is similar to the one used for the fitness function, thus we will omit the code for the superclasses extended.

Before applying the genetic operators we need to create an initial population, this can be done by randomly creates the individuals as follows:

```
public class RandomBitStringInitialization extends
    ↪ Initialization<BitStringIndividual,
    ↪ IntegerFitnessValue> {
  ...
  private int individualSize;
  private Random random;

  public RandomBitStringInitilization(..., Properties
      ↪ userProperties, ...) {
    ...
    individualSize = userProperties.getInt(
        ↪ INDIVIDUAL_SIZE_PROPERTY);
    random = new Random();
  }

  @Override
  public IndividualWrapper<BitStringIndividual,
      ↪ IntegerFitnessValue> generateNextIndividual(
      ↪ int id) {
    BitStringIndividual individual = new
        ↪ BitStringIndividual(individualSize);

    for (int i = 0; i < individualSize; i++)
      individual.set(i, random.nextInt(2) == 1);

    return new IndividualWrapper(individual);
  }
}
```

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

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

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

```
public class BitStringSinglePointCrossover extends Crossover<BitStringIndividual, IntegerFitnessValue> {
  ...
  private Random random;

  public BitStringSinglePointCrossover(...) {
    ...
    random = new Random();
  }

  @Override
  public List<IndividualWrapper<BitStringIndividual, IntegerFitnessValue>> cross(IndividualWrapper<BitStringIndividual, IntegerFitnessValue> wrapper1, IndividualWrapper<BitStringIndividual, IntegerFitnessValue> wrapper2, ...) {
    BitStringIndividual parent1 = wrapper1.getIndividual();
    BitStringIndividual parent2 = wrapper2.getIndividual();

    cutPoint = random.nextInt(parent1.size());

    BitStringIndividual child1 = new BitStringIndividual(parent1.size());
    BitStringIndividual child2 = new BitStringIndividual(parent1.size());

    for (int i = 0; i < cutPoint; i++) {
      child1.set(i, parent1.get(i));
      child2.set(i, parent2.get(i));
    }

    for (int i = cutPoint; i < parent1.size(); i++) {
      child1.set(i, parent2.get(i));
      child2.set(i, parent1.get(i));
    }

    List<IndividualWrapper<BitStringIndividual, IntegerFitnessValue>> children = new ArrayList<>(2);

    children.add(new IndividualWrapper<>(child1));
    children.add(new IndividualWrapper<>(child2));

    return children;
  }
}
```

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

Thereafter, a random mutation function is implemented:

```
public class BitStringMutation extends Mutation<BitStringIndividual, IntegerFitnessValue> {
  ...
  private Random random;

  public BitStringMutation(...) {
    ...
    mutationProbability = userProperties.getDouble(MUTATION_PROBABILITY_PROPERTY);
    random = new Random();
  }

  @Override
  public IndividualWrapper<BitStringIndividual, IntegerFitnessValue> mutate(IndividualWrapper<BitStringIndividual, IntegerFitnessValue> wrapper) {
    BitStringIndividual individual = wrapper.getIndividual();

    for (int i = 0; i < individual.size(); i++)
      if (random.nextDouble() <= mutationProbability)
        individual.set(i, !individual.get(i));

    return wrapper;
  }
}
```

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

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

Finally, we need to register with the *Driver* all the classes we defined as follows:

```
public class App {
  public static void main(String[] args) {
    ...
    driver.setIndividualClass(BitStringIndividual.class);
    driver.setFitnessValueClass(IntegerFitnessValue.class);

    driver.setInitializationClass(RandomBitStringInitialization.class);
    driver.setInitializationPopulationSize(POPULATION_SIZE);
    userProperties.setInt(INDIVIDUAL_SIZE_PROPERTY, INDIVIDUAL_SIZE);

    driver.setElitismClass(BestIndividualsElitism.class);
    driver.activateElitism(true);
    userProperties.setInt(NUMBER_OF_ELITISTS_PROPERTY, NUMBER_OF_ELITISTS);

    driver.setParentsSelectionClass(RouletteWheelParentsSelection.class);

    driver.setCrossoverClass(BitStringSinglePointCrossover.class);

    driver.setSurvivalSelectionClass(RouletteWheelSurvivalSelection.class);
    driver.activateSurvivalSelection(true);

    driver.setUserProperties(userProperties);
    ...
    driver.run();
    ...
  }
}
```

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

Assuming that a Hadoop MapReduce cluster is already set up, we can pack the code in a single JAR file, including the elephant56 dependency, and execute it with the standard Hadoop launch method.

6. ARCHITECTURE

In this section we describe the architecture of elephant56. We conceptually divided it into two levels of abstraction:

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

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

6.1 Core Level

The core level is responsible of dealing with the Hadoop MapReduce platform. This is made by extending the MapReduce classes provided with the Hadoop library. Figure 4 shows the structure of the `core` package, which contains five subpackages, each responsible of a different functionality.

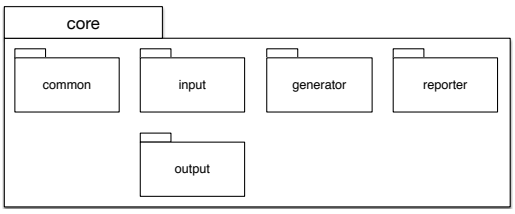

Figure 4: The `core` package class diagram.

The class in charge of the whole execution flow is the `Driver` class (Figure 5). It invokes the components included into the other core packages and it is specialised into different classes, which implement the models shown in Section 4. The `Driver` class is the linking point between `core` and `user` layers and the primary interface with the developer.

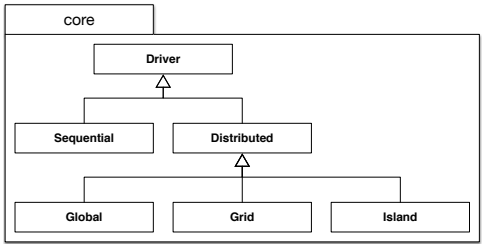

Figure 5: The `Driver` class.

6.1.1 core.common

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

6.1.2 core.input

The `core.input` package contains the implementations for the Hadoop input operations. The `NodesInputFormat` assigns a group of individuals (i.e., a MapReduce split) to a specific node. As mentioned in Section 3, the serialisation is made by using Avro, thus each split corresponds to a single binary file stored into the HDFS. The information about the split are stored using the `PopulationInputSplit` class and read with the `PopulationRecordReader` class which deserialises the Avro objects into Java objects for the slave nodes.

6.1.3 core.output

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

6.1.4 core.generator

The `core.generator` package (Figure 6) is composed of the `GenerationsPeriodExecutor` class that implements the SGA described in Section 4.1. The specialisation of this class allows the distribution of GAs according to the different parallel models (see Section 4).

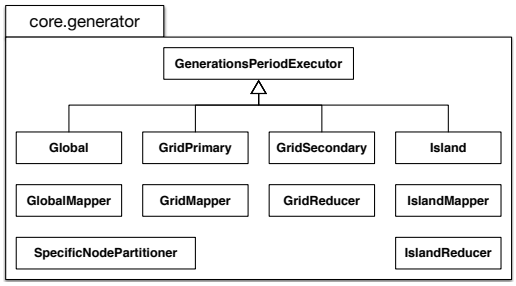

Figure 6: The `generator` package class diagram.

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

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

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

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

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

6.1.5 core.reporter

The package `core.reporter` (Figure 7) provides some utilities that can be used to produce a report containing some statistics about the GA execution. The output consists of some CSV (Comma Separated Values) files stored in the HDFS.

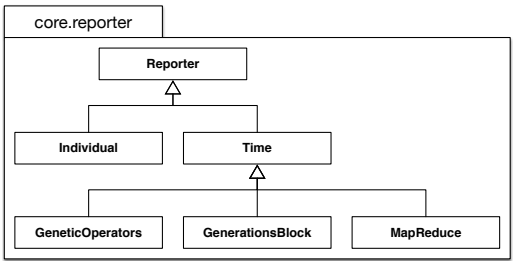

Figure 7: The `reporter` package class diagram.

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

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

6.2 User Level

The `user` package (Figure 8) contains all the base classes the developer should extend in order to implement a specific GA, as explained in Section 5.

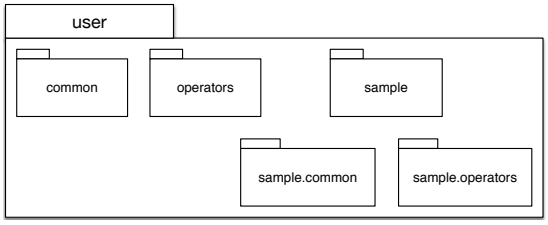

Figure 8: The `user` package class diagram.

In particular, the `common` package contains the `Individual` and `FitnessValue` classes for the definition of the chromosome and fitness value, while the `operators` package contains the classes for the definition of specific genetic operators. Some sample default implementations are included into the `sample` package, such as: sequences of primitive Java types, number fitness values, the random initialisation of sequences, a single point crossover, and the roulette wheel selection.

7. CONCLUSIONS AND FUTURE WORK

This paper described elephant56, a novel framework that allows developers to implement their own GAs in a distributed way on a Hadoop MapReduce cluster of computers.

As future work we plan to empirically evaluate the framework in order to assess its scalability with all the three models (i.e., the global, grid and island model).

Furthermore, it is our intention to enrich the framework with a full documentation for the usage of the framework together with some example problem implementations.

8. REFERENCES

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] 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 *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 785–793. IEEE, 2012.

[3] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro. Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, pages 113–135. IGI Global, 2013.

[4] P. Fazenda, J. McDermott, and U.-M. O'Reilly. A Library to Run Evolutionary Algorithms in the Cloud using MapReduce. In *European conference on Applications of Evolutionary Computation (EvoApplications)*, pages 416–425. Springer, 2012.

[5] F. Ferrucci, M.-T. Kechadi, P. Salza, and F. Sarro. A Framework for Genetic Algorithms Based on Hadoop. *arXiv preprint arXiv:1312.0086*, Nov. 2013.

[6] F. Ferrucci, P. Salza, M.-T. Kechadi, and F. Sarro. A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce. In *ACM/SIGAPP Symposium on Applied Computing (SAC)*, pages 1664–1667. ACM Press, 2015.

[7] M. Harman. The Current State and Future of Search Based Software Engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[8] D.-W. Huang and J. Lin. Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems Using MapReduce. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 780–785. IEEE, 2010.

[9] C. Jin, C. Vecchiola, and R. Buyya. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In *IEEE International Conference on E-Science (e-Science)*, pages 214–221. IEEE, 2008.

[10] G. Luque and E. Alba. *Parallel Genetic Algorithms: Theory and Real World Applications*. Number 367 in Studies in Computational Intelligence. Springer, Berlin, 2011.

[11] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling Genetic Algorithms Using MapReduce. In *International Conference on Intelligent Systems Design and Applications (ISDA)*, pages 13–18. IEEE, 2009.

[12] L. Zheng, Y. Lu, M. Ding, Y. Shen, M. Guoz, and S. Guo. Architecture-Based Performance Evaluation of Genetic Algorithms on Multi/Many-core Systems. In *IEEE International Conference on Computational Science and Engineering (CSE)*, pages 321–334. IEEE, 2011.