A Parallel Genetic Algorithms Framework based on Hadoop MapReduce

Filomena Ferrucci, Pasquale Salza
University of Salerno, Italy
{fferrucci,psalza}@unisa.it

M-Tahar Kechadi
University College Dublin, Ireland
tahar.kechadi@ucd.ie

Federica Sarro
University College London, United Kingdom
f.sarro@cs.ucl.ac.uk

ABSTRACT

This paper describes a framework for developing parallel *Genetic Algorithms* (*GAs*) on the *Hadoop* platform, following the paradigm of *MapReduce*. The framework allows developers to focus on the aspects of *GA* that are specific to the problem to be addressed. Using the framework a *GA* application has been devised to address the *Feature Subset Selection* problem. A preliminary performance analysis showed promising results.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Algorithms, Experimentation, Performance

Keywords

Hadoop, MapReduce, Parallel Genetic Algorithms

1. INTRODUCTION

Genetic Algorithms (*GAs*) are powerful metaheuristic techniques used to address many software engineering problems [7] that involve finding a suitable balance between competing and potentially conflicting goals, such as looking for the set of requirements that balance software development cost and customer satisfaction or searching for the best allocation of resources to a software development project.

GAs simulate several aspects of the "Darwin's Evolution Theory" to converge towards optimal or near-optimal solutions [5]. Starting from an initial population of individuals, each represented as a string (*chromosome*) over a finite alphabet ("genes"), every iteration of the algorithm generates a new population executing genetic operations on selected individuals, such as *crossover* and *mutation*. Along the chain of generations, the population tends to improve its individuals by keeping the strongest individuals (characterised by the best fitness value) and rejecting the weakest ones. The generations continue until a specified *stopping condition* is verified. The problem solution is given by the individual with the best fitness value in the last population.

GAs are usually executed on single machines as sequential programs, so scalability issues prevent that they are effectively applied to real-world problems. However, these algorithms are naturally parallelisable, as it is possible to use more than one population and execute the operators in parallel. Parallel systems are becoming commonplace mainly due to the increasing popularity of *Cloud Systems* and the availability of distributed platforms, such as *Apache Hadoop* [12], characterized by easy scalability, reliability and fault-tolerance. It is characterized by the use of the *MapReduce* algorithm and the distributed file system *HDFS*, which run on large clusters of commodity machines.

In this paper, it is presented 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 aspects of *Hadoop* and allow users to focus on the definition of their problems in terms of *GAs*. An example of use of the framework for the "Feature Subset Selection" problem is also provided together with preliminary performance results.

2. RELATED WORK

As described by Di Martino et al. [3], there exist tree possible main grain parallelism implementations of *GAs* by exploiting the *MapReduce* paradigm: *Fitness Evaluation Level* (*Global Parallelisation Model*); *Population Level* (*Coarse-grained Parallelisation Model* or *Island Model*); *Individual Level* (*Fine-grain Parallelisation Model* or *Grid Model*) [3]. The *Global Parallelisation Model* was implemented by Di Martino et al. [3] with three different *Google App Engine MapReduce* platform configurations to address automatic test data generation.

For the same purpose, Di Geronimo et al. [2] developed *Coarse-grained Parallelisation Model* on the *Hadoop MapReduce* platform.

"MRPGA" is a parallel *GAs* framework based on *.Net* and "Aneka", a platform which simplifies the creation and communication of nodes in a *Grid* environment [8]. It exploits the *MapReduce* paradigm, but with a second *reduce* phase. A master coordinator manages the parallel *GA* execution among nodes, following the *islands parallelisation model*, in which each node computes the *GA* operators for a portion

ACM SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/10.1145/2695664.2696060 ...$15.00.

of the population only. During the first step, every *mapper* node reads a portion of individuals and computes the *fitness* values. Then, in the first *reduce* phase, each node applies the *selection* operator for the input individuals, giving a local optimum relative to each island. The eventual single *reducer* computes the global optimum and applies the remaining *GA* operators. Comparing the results from *MRPGA* with a sequential version of the same algorithm, they have had an interesting degree of scalability *overhead* factor but also a high overhead.

The approach by Verma et al. [11] was developed on *Hadoop*. The number of *Mapper* and *Reducer* nodes are unrelated. The main difference with *MRPGA* lies in the fact a sort of "migration" is performed among the individuals as it randomly sends the outcome of the *Mapper* nodes to different *Reducer* nodes. Results showed the potential of parallel *GA* approach for the solution of big computation problems.

3. THE PROPOSED FRAMEWORK

The proposed framework aims to reduce the effort for implementing parallel *GAs*, exploiting the features of *Hadoop* in terms of scalability, reliability and fault tolerance, thus allowing developers do not worry about the infrastructure for a distributed computation. One of the problems with the execution of *GAs* in a distributed system is the *overhead* that might be reduced if the network movements are reduced. So it is important to minimise the number of phases, whenever they are associated to an execution which passes from a machine to a new one. For this reason, the framework implements the *islands* model (mostly following the design provided from Di Martino et al. [3]), forcing the machines to work always with the same portion of population and using just one *Reducer* phase.

In the following it is described the design of the involved components, starting from the *Driver*, which is the fulcrum of the framework.

3.1 The Driver

It manages the interaction with the user and launch jobs on the cluster. It is executed on a machine even separated from the distributed cluster and it also computes some functions in order to manage the received results, generation by generation.

The elements involved into the *Driver* process are:

Initialiser: the user can define how to generate the first individuals for each island, but the default implementation 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*;

Generator: it executes one generation on the cluster and produces the new population, storing each individual again in *HDFS* with the fitness values and a flag indicating if an individual satisfies the *stopping condition* defined by the user;

Terminator: at the end of each generation, the *Terminator* component checks the *stopping conditions*: the standard one of "if the maximum number of generations has been reached" or a defined by user one, by checking the presence of the flag in the *HDFS*. Once terminated, the population is directly submitted to the *SolutionsFilter* job;

Migrator: this job allows moving individuals from an island to another, according to the criteria defined by the user such as the period of migration, number and destinations of migrants and the selected method for choosing migrants. This phase is optional and it can occur by setting a "migration period" parameter;

SolutionsFilter: when the job terminates, all the individuals of the last generation are filtered according to those that satisfy the *stopping condition* and those which do not. Then, the results of the whole process is stored in *HDFS*.

3.2 The Generator and the other components

Each *generator* job makes the population evolve. In order to develop the complex structure described below, it was 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 rather than the raw serialisation used by *Hadoop* as default method. Thus, using *Avro* [1] it is easier to store objects and to save space on the disk, also allowing any external treatment of them and a quick exchange of 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*.

The generation work is distributed on nodes of the *Cloud* as follows:

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

$$\text{split} : \mathit{individuals} \rightarrow \text{list}\,(\mathit{individual}, \text{NULL})$$

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 according to J islands, the J *Mappers* compute the *fitness values* for each individual of its corresponding island:

$$\text{map} : (\mathit{individual}, \text{NULL}) \rightarrow (\mathit{individual}_{\text{F}}, \text{NULL})$$

The user defines how to evaluate the *fitness* 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* acts in a chain. It checks if the current individuals satisfies the *stopping condition*. 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 *stopping condition* has not been satisfied yet, this 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:

$$\text{map} : (\mathit{individual}_{\text{F}}, \text{NULL}) \rightarrow (\mathit{couple\_information}, \mathit{individual}_{\text{F}})$$

If an individual has been chosen more than one time, it is replicated. Then all the individuals, including those not

chosen if the *elitism* is active, leave the current machine and go to the correspondent *Reducer* for the next step;

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

$$\text{reduce} : (\textit{couple\_information}, \text{list}\,(\textit{individual}_{\text{F}})) \rightarrow (\textit{individual}, \text{TRUE})$$

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

Mutation: during this phase, the chained *Mappers* manage to make the mutation of the genes defined by the user. Only the *offspring* can be mutated:

$$\text{map} : (\textit{individual}, \text{TRUE}) \rightarrow (\textit{individual}_{\text{M}}, \text{NULL})$$

Elitism: in the last phase. If the user chooses to use the optional elitism, the definitive population is chosen among the individuals of the offspring and the previous population:

$$\text{map} : (\textit{individual}, \text{NULL}) \rightarrow (\textit{individual}, \text{NULL})$$

At this point, the islands are ready to be written into *HDFS*.

The architecture of the generator component provides two levels of abstraction:

- 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, thus the user can use them for most cases.

The *Terminator* component plays two roles in different times: 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, i.e. the count of the maximum number of generations has been reached or at least one individual has been marked of satisfying the *stopping condition* during the most recent generation phase. 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.

The *Initialiser* computes an initial population according to the definition of the user.

Another optional component is the *Migrator* which shuffles individuals among the islands according to the definition of the user. It is at the same time a local component and a job executed on the cluster. It is started by the *Driver* according to a period counter.

The component *SolutionsFilter* is invoked only when at least one individual has provoked the termination by satisfying the *stopping condition*. It simply filters the individuals of the last population by dividing those that satisfy the condition from those which do not. More details about the proposed framework can be found in [4].

4. PRELIMINARY ANALYSIS

Many software engineering problems involve prediction and classification tasks (e.g. effort estimation [9] and and fault prediction [10]). Classification consists of learning from example data, called "training dataset", with the purpose of properly classifying any new input data. The *training dataset* includes a list of records, also called "instances", consisting of a list of "attributes" (*features*). Several classifiers exist, among them *C4.5* builds a *Decision Tree*, which is able to give a likely class of membership for every record in a new dataset. The effectiveness of the classifier can be measured by the *accuracy* of the classification of the new data:

$$\textit{accuracy} = \frac{\text{correct classifications}}{\text{total of classifications}}$$

"Feature Subset Selection" (*FSS*) helps to improve prediction/classification accuracy by reducing noise in the *training dataset* [6], selecting the optimal subset of features. Nevertheless the problem is NP-hard. *GAs* can be used to identify a near-optimal subset of features in a training dataset. The framework was exploited in order to develop a *GA* to address the problem. For the development of the application of *FSS*, only 535 lines of code were written against the 3908 of the framework infrastructure. In terms of the framework, the *Driver* is the main part of the algorithm and is executed on one machine. It needs some additional information before execution: the parameters for the *GA* environment, such as the number of individuals in the initial population, the maximum number of generations, etc.; the *training dataset*; the *test dataset*.

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). During each generation, every subset is evaluated by computing the *accuracy* value and all the *GAs* operations are applied until target *accuracy* is achieved or maximum number of generations is reached, according to the following steps:

Fitness: for each subset of attributes, the *training dataset* is reduced with respect of the attributes in the subset. The fitness value is computed by applying the steps:

1. Build the *Decision Tree* through the *C4.5* algorithm and computing the *accuracy* by submitting the current dataset;
2. The operations are repeated according to the *cross-validation folds* parameter;
3. The mean of *accuracies* is returned.

4.1 Subject

The "Chicago Crime" dataset (from the *UCI Machine Learning Repository* was used. The dataset has 13 features

and 10000 instances. It was divided into two parts: the first 60 % as *training dataset* and the last 40 % as *test dataset*.

The test bench was composed by three versions:

- **Sequential**: a single machine executes a *GA* version similar to the framework one by using the *Weka Java* library, for the *University of Waikato*;
- **Pseudo-distributed**: here the framework version of the algorithm is used. 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. Of course, this is the object of interest.

All the versions run on the same remote *Amazon EC2* cluster and all the machines involved had the same configuration, in order to give a strong factor of fairness. This configuration is named by *Amazon* as "m1.large", a machine with 2 CPUs, 7.5 GB of RAM and 840 GB for storage.

The *sequential* version was executed on a single machine and also the *pseudo-distributed*, but over a *Hadoop* installation. Instead, the *distributed* version needed a full *Hadoop* cluster composed by 1 master and 4 computing slaves.

The chosen configuration for the test consisted of 10 generations, a migration period of 5 with 10 migrants. It was specified as a *stopping condition* only the "maximum number of generations".

The final *accuracy* is computed on *test dataset* using the classifier built with the best individual identified.

4.2 Results

As for the *accuracy*, the results were 89.55 % for *sequential*, 89.40 % for *pseudo-distributed* and 89.45 % for *distributed* version. This confirms that the application achieves its objective, by giving a subset that has both a reduced number of attributes and a suitable *accuracy* value.

As for the *running time*, the *distributed* version won against other versions, with 1384 s against 2993 s for *pseudo-distributed* and 2179 s for *sequential*. It is clear that the distributed version must face some considerable intrinsic *Hadoop* bottlenecks such as *map* initialisation, data reading, copying and memorisation etc. and it has an additional phase of *Migration* not present in the *sequential* version. The proposed framework does not consider every single aspect of *Hadoop* and this suggest that a further optimisation might increase the performance of the parallel version compared with the sequential one. It is predictable that the *distributed* version performs better when the amount of required computation is large [4]. Since the *pseudo-distributed* is second to the *distributed*, this suggests that is worth splitting the work among multiple nodes. More details about the analysis carried out can be found in [4].

5. CONCLUSIONS AND FUTURE WORK

This work proposed a framework for *Genetic Algorithms* with the aim of simplifying the development of *Genetic Algorithms* on *Cloud*. The obtained results showed an interesting effort ratio between the number of *lines of code* for the application against the ones for the framework. The preliminary tests about the performance confirmed the natural tendency of *GAs* to parallelisation but the threat of *overhead* is still around the corner. Indeed, the use of parallelisation seems to be worth only in presence of computationally intensive and not trivial problems [4] but scalabity issues need to be deeper addressed. Because of the use of specific components (e.g. *MapTasks* and *ReduceTasks* chains), it does not seem unreasonable to suggest that giving an "ad-hoc" optimisation of *Hadoop* might improve the performance and there is also a possibility that the change of model of parallelisation [3] could give better results.

6. REFERENCES

[1] D. Cutting. Data interoperability with apache avro. Cloudera Blog, 2011.

[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 *Software Testing, Verification and Validation (ICST), 2012 IEEE 5th International Conference on*, 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*, chapter 6, pages 113–135. IGI Global, 2012.

[4] F. Ferrucci, M.-T. Kechadi, P. Salza, and F. Sarro. A framework for genetic algorithms based on hadoop. *arXiv preprint arXiv:1312.0086*, 2013.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[6] M. A. Hall. *Correlation-based Feature Selection for Machine Learning*. PhD thesis, The University of Waikato, 1999.

[7] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[8] C. Jin, C. Vecchiola, and R. Buyya. Mrpga: An extension of mapreduce for parallelizing genetic algorithms. In *E-Science (e-Science), 2008 IEEE 4th International Conference on*, pages 214–221. IEEE, 2008.

[9] F. Sarro, S. Di Martino, F. Ferrucci, and C. Gravino. A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In *Applied Computing (SAC), 2012 ACM 27th Symposium on*, pages 1215–1220. ACM, 2012.

[10] F. Sarro, F. Ferrucci, and C. Gravino. Single and multi objective genetic programming for software development effort estimation. In *Applied Computing (SAC), 2012 ACM 27th Symposium on*, pages 1221–1226. ACM, 2012.

[11] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *Intelligent Systems Design and Applications (ISDA), 2009 9th International Conference on*, pages 13–18. IEEE, 2009.

[12] T. White. *Hadoop: The Definitive Guide*. O'Reiily, third edition, 2012.