

A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites

Linda Di Geronimo, Filomena Ferrucci, Alfonso Murolo, Federica Sarro

University of Salerno

Via Ponte don Melillo, 84084 Fisciano, Italy

e-mail: {fferrucci, fsarro}@unisa.it

Abstract— Software testing represents one of the most explored fields of application of Search-Based techniques and a range of testing problems have been successfully addressed using Genetic Algorithms. Nevertheless, to date Search-Based Software Testing (SBST) has found limited application in industry. As in other fields of Search-Based Software Engineering, this is principally due to the fact that when applied to large problems, Search-Based approaches may require too much computational efforts. In this scenario, parallelization may be a suitable way to improve the performance especially due to the fact that many of these techniques are “naturally parallelizable”. Nevertheless, very few attempts have been provided for SBST parallelization. In this paper, we present a Parallel Genetic Algorithm for the automatic generation of test suites. The solution is based on Hadoop MapReduce since it is well supported to work also in the cloud and on graphic cards, thus being an ideal candidate for high scalable parallelization of Genetic Algorithms. A preliminary analysis of the proposal was carried out aiming to evaluate the speed-up with respect to the sequential execution. The analysis was based on a real world open source library.

Keywords- *Parallel Genetic Algorithm, Hadoop MapReduce, Search Based Software Testing.*

I. INTRODUCTION

Software testing represents one of the most explored fields of application of Search-Based techniques. Indeed, a range of testing problems has been addressed, such as structural testing (both static and dynamic), functional testing, non-functional testing, mutation testing, test case prioritization, and so on (see [27] for a survey). Despite all such efforts, to date, Search-Based Software Testing (SBST) has found limited application in industry [7][24]. As in other fields of Search-Based Software Engineering (SBSE), this is principally due to the fact that in general when applied to large problems, Search-Based approaches may require too much computational efforts [24][40]. Indeed “*many approaches that are attractive and elegant in the laboratory, turn out to be inapplicable in the field, because they lack scalability*” [20], i.e., the ability to be efficient when applied to large real-world software systems.

In this scenario, parallelization may be a suitable way to improve the performance, both in terms of computational time and effectiveness in the exploration of the search space. Moreover, many of these techniques are “naturally parallelizable” [20]. As a matter of fact, the *population based* characteristic of Genetic Algorithms (GA) allows the fitness function of each individual to be computed in parallel. Nevertheless, very few attempts have been provided for

SBST parallelization. As suggested in [40] “*one possible historical barrier to wider application of parallel execution has been the high cost of parallel execution architectures and infrastructure. ...While commodity PCs have significantly reduced the cost of such clusters, their management can still be a non-trivial task, restricting the potential availability for developers.*”

The emerging use of Cloud Computing and of General Purpose computing on Graphical Processing Unit (GPGPU) can represent an affordable solution to address the above issues. This can provide a significant impulse in SBST parallelization allowing for more scalable research proposals suitable to be transferred in software industry. Indeed, GPGPU exploits a parallelism (originally designed for graphics) for non graphical tasks using a single hardware component (GPU) that is less expensive than multiple PCs and has smaller management costs [40]. Cloud Computing solutions offer a parallel distributed computational environment together with an on demand resource handling and allocation to easily scale up. As a consequence, companies that plan to use this kind of solutions as an alternative to traditional cluster-based platforms could employ virtually unlimited computational resources without caring about the management and the maintenance of the overall IT infrastructure. Furthermore, the on-demand resource allocation mechanism allows companies to reduce costs consistently since they have to pay only for the computational resources actually used.

Based on these considerations in this paper we present a Genetic Algorithm for the automatic test suite generation for Object Oriented software and show how it can be parallelized using Hadoop MapReduce [4]. Hadoop MapReduce is a framework for developing applications that rapidly process vast amounts of data in parallel on large clusters of computing nodes. This choice was motivated by the fact that it is becoming the de-facto standard MapReduce implementation and it has been used also in industry [37]. Moreover, it is well supported to work not only on clusters, but also on the cloud [3] and on graphic cards [19], thus being an ideal candidate for high scalable parallelization of GAs.

The proposed Parallel Genetic Algorithm (PGA) takes as input the software to be tested and an initial population of random solutions (i.e., a set of test cases) that are evolved according to a given coverage criterion (i.e., branch coverage) following a global parallelization model. This means that at each iteration the individual fitness evaluation, which is the most time consuming GA task in the considered domain, is carried in a parallel way exploiting the

MapReduce programming model. At the end of the evolution process a JUnit test suite, optimized to cover as much branches as possible of the software under test, is given in output to the user.

A preliminary analysis of the proposed PGA was realized on standard cluster exploiting a real word open source library. The results were evaluated in terms of the time needed to generate the test suites and compared with the ones achieved executing the same GA in the traditional sequential way.

Although the problem of parallelizing Search-Based techniques is not new (see section V for related work), to the best of our knowledge this work is the first that proposes a PGA based on Hadoop MapReduce for the automatic generation of JUnit test suite, showing also a preliminary evaluation of its use.

The rest of the paper is organized as follows. In section II we describe the design of a Genetic Algorithm for the automatic generation of JUnit test suite. Section III presents the approach we proposed to parallelize the genetic algorithm exploiting HadoopMapReduce. Section IV reports the design and the results of the analysis we carried out to assess the effectiveness of the proposed approach. Section V describes related work while some final remarks and future work conclude the paper.

II. A GENETIC ALGORITHM FOR THE AUTOMATIC GENERATION OF TEST SUITE

Genetic Algorithms (GA) [17] are evolutionary algorithms that, inspired by the theory of natural evolution, simulate the evolution of species emphasizing the law of survival of the strongest to solve, or approximately solve, optimization problems. To this end, a fitness function is used to evaluate the goodness (i.e., fitness) of the solutions represented by the individuals (i.e., chromosomes) and genetic operators based on selection and reproduction are employed to produce new offspring.

The elementary evolutionary process of GA is composed by the following steps:

1. An initial population is usually randomly generated;
2. A fitness function is used to assign a fitness value to each individual;
3. According to their fitness value some individuals are selected as parents (i.e., reproduction selection) and new individuals (offspring) are created by applying reproduction operators (i.e., crossover and mutation) and evaluated using the fitness function;
4. To determine the individuals that will be included in the next generation a selection based on individual's fitness value is applied (i.e., survival selection);
5. Steps 2, 3, and 4 are repeated until stopping criteria hold.

In the following, we describe the design choices we made for tailoring GA to automatically produce test suites for the unit testing of classes of a given software system.

In such a scenario, a chromosome represents a test suite (i.e., a set of n test cases) for a given Java class belonging to the software under test (SUT). We employ a representation similar to [16] for each test case that contains a sequence of

statements of length l for object creation, state change, and method invocation. In particular, each statement can be one of these types: primitive, constructor, field, and method [16].

The initial population is randomly generated. Then the classical evolutionary scheme is applied aiming at evolving towards populations containing chromosomes with a high fitness value.

The fitness of each individual is measured in terms of branch coverage by exploiting the Cobertura coverage tool [8]. Given a test suite and a SUT, this tool calculates the percentage of branches covered by the test suite when executed with JUnit on the instrumented bytecode of the SUT. In particular, Cobertura considers a branch as covered if it is exercised on the true and false outcomes, at least once. Moreover, if a same branch is covered more than one time by different test cases within the same test suite (i.e., chromosome) the corresponding branch is counted just once during the fitness evaluation.

Concerning the reproduction operators we employ a single point crossover which generates offspring O1 and O2 from two parents (i.e., two test suites) P1 and P2, selecting a random point of cut in the parent chromosomes and swapping their test cases with a rate of 0.5. As for the employed mutation operator, it changes with probability 0.25 each gene in a chromosome replacing the current test case with a new one obtained by using a feedback-directed random generation which adds method arguments in an incremental way (i.e., building, executing, and checking arguments against a set of contracts to determine whether an input is redundant, illegal, contract-violating, or useful for generating more inputs for a given method). These genetic operators working on an entire test case allowed us to avoid the difficulties due to dependencies between statements of a test case [36].

As for the selection operators we employed Tournament and Linear Ranking selectors for reproduction and survival selection, respectively.

The search process is stopped after performing a fixed number of generations or whether the best solution doesn't change for a certain number of generations.

III. A PARALLEL GENETIC ALGORITHM BASED ON HADOOP MAP REDUCE

In the following subsections we first give some background, reporting on the strategies proposed in the literature to parallelize Genetic Algorithms and recalling the main aspects of MapReduce and Hadoop MapReduce. Then, we present the design of the proposed parallel Genetic Algorithm using Hadoop MapReduce.

A. Parallelization Strategies

Several GA parallelization strategies exist depending on the grain of parallelization to achieve. Basically, three levels of parallelization can be exploited:

- fitness evaluation level (i.e., *global parallelization model*);
- population level (i.e., *coarse-grained parallelization or island model*);

- individual level (i.e., *fine-grained parallelization or grid model*) [34].

In the global parallelization model, a node acting as a master, manages the population (i.e., applying genetic and selection operators) and distributes the individuals among slave nodes which compute only the fitness values of the individuals. The main advantage of using such a model is that it does not require any change to the design of traditional GA since the individual fitness evaluation is independent from the rest of the population.

In the island model the population is subdivided in several subpopulations of relatively large size which are located in several islands (i.e., nodes). Thus, a Genetic Algorithm is executed on each subpopulation and such subpopulations exchange information by allowing some individuals to migrate from one island to another according to a given temporal criteria. The main expected advantages of this model are: (i) different subpopulations could explore different portions of the search-space; (ii) migrating individuals injects diversity into the converging population.

Finally, in the grid model each individual is placed on a grid (i.e., each individual is assigned to a node) and all GA operations are performed in parallel evaluating simultaneously the fitness and applying locally selection and genetic operations to a small neighboring. A drawback of this approach is the overhead due to the frequent communications between grid nodes.

There exist proposals that fall in the so called hybrid models which combine different levels of parallelization.

In the following we define a way to exploit the first parallelization model using Map Reduce.

B. MapReduce

MapReduce is an elegant and flexible paradigm which enables to develop large-scale distributed applications [12]. It is expressed in terms of 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 parallelization while the Reduce collects and merges the results. In particular, a master node splits the initial input in several pieces, each one identified by a unique *key*, and distributes them via the Map function to several slave nodes (i.e., *Mappers*) which work in parallel and independently from each other performing the same task on a different piece of input. As soon as each Mapper finishes its own job the output is identified and collected via the Reducer function. In particular, each *Mapper* produces a set of *intermediate* key/value pairs which are exploited by one or more *Reducers* to group together all the *intermediate* values associated to the same key and to compute the list of output results. It is worth mentioning that the different *intermediate* keys emitted by the *Mapper* functions affect the way the model distributes the computation of each *Reducer* on different machines. Thus, the program automatically invokes and allocates a number of distinct *Reducers* that correspond to the number of distinct *intermediate* keys.

C. HadoopMapReduce

Several different implementations of MapReduce have been proposed. The most famous one is the AppEngine-MapReduce [18], built on the top of the distributed Google File System, and Hadoop MapReduce [4].

Hadoop MapReduce is an open-source project of the Apache Software Foundation aiming at supporting developers in realizing applications that rapidly process vast amounts of data in parallel on large clusters of computing nodes. Its popularity is increasing rapidly as well as its adoption by large companies such as IBM and Yahoo.

With respect to its “big brother”, the AppEngine-MapReduce [18], Hadoop MapReduce supports both Map and Reduce phases, thus avoiding a programmer to manage its own Reducer. Using Hadoop also lets us avoid some limitations imposed by GoogleApp Engine (e.g., it is not allowed to use files or executing external threads and processes). Moreover, Hadoop MapReduce is well supported to work not only on clusters, but also on the cloud [3] and on graphic cards [19], thus being an ideal candidate for high scalable parallelization of GA.

Hadoop MapReduce exploits a distributed file system (an open source implementation of Google File System), named the Hadoop Distributed File System (HDFS), to store data as well as intermediate results and uses the MapReduce programming model for data processing. The Hadoop MapReduce interpretation of the Distributed File System was conceived to increase large-data availability and fault-tolerance by spreading copies of the data throughout the cluster nodes, in order to achieve both lower costs (for hardware and RAID disks) and lower data transfer latency between the nodes themselves. Hadoop offers also a database named HBase that was created to compete with Google’s DataStore in order to speed up the data retrieval for billions of small information.

In the next section we detailed how we exploited the Hadoop MapReduce programming model to parallelize the proposed Genetic Algorithm for the automatic creation of test suites.

D. The proposed Parallel Genetic Algorithm based on Hadoop MapReduce

The underlying idea in using MapReduce to parallelize the Genetic Algorithm described in section II is to encapsulate each iteration of the GA as a separate MapReduce job and parallelize the chromosome fitness evaluation assigning such task to several *Mappers*, while a single *Reducer* is responsible to collect the results and to perform the genetic operations (i.e., parents selection, crossover and mutation, and survival selection) needed to produce a new generation following a global parallelization model.

Figure 1 shows the proposed architecture based on HadoopMapReduce and composed by the following main components: a *Parallel Genetic Algorithm*, a *Master*, a number of *Mappers* and a *Reducer*, together with two other units, namely *InputFormat* and *OutputFormat*, which are responsible to split the data for the *Mappers* and to store the

Reducer output into the Hadoop Distributed File System (HDFS), respectively.

Let us note that these components communicate with each other exploiting the HDFS distributed file system provided by Hadoop, while the communications within the Hadoop framework (i.e., those between the master and slave nodes) are carried out via socket using SSH (Secure Shell).

The *Parallel Genetic Algorithm* module lets the user (i.e., a tester) specify the Software Under Test (SUT) and manages the overall execution of GA. Once the GA is terminated it returns to the user a test suite for the software under test.

Figure 2 reports the pseudocode for the *Parallel Genetic Algorithm* component. First of all a PreExecutionPhase (line 1) is needed to instrument the SUT bytecode in order to get information about the coverage in a program run (lines 8-9) and to generate an initial random population (lines 10-11). Then, it starts the evolutionary process by creating and executing the MapReduceJob (lines 3-4) and checks at each iteration if a termination criterion holds (line 2). Finally, a postExecution phase is needed to clean the local file system and the distributed one.

The MapReduceJob is the core of the *Parallel Genetic Algorithm* module since it allows us to parallelize fitness evaluations and distribute the computation over the nodes (see Fig. 1). In particular, a MapReduceJob consists of three phases (i.e., **Split**, **Map**, and **Reduce**) in which each component performs its proper task as detailed in the following.

Split. In this phase the *InputFormat* module gets the current population (i.e., the test suites composing the current population) from the HDFS and processes it in order to split it in crumch of data (i.e., input split) to be distributed among the *Mapper* modules.

The number of input splits is dynamically computed on the basis of the number of available *Mappers*.

The *Master* module is responsible to coordinate and supervise the assignment of resources and the computations taking care also of the load balancing aspects. In more details, once the *InputFormat* begins to emit the $\langle key, value \rangle$ pairs - exploiting the RecordReader component of Hadoop - the underlying Hadoop framework is automatically notified and the *Master* component is invoked to assign the input split produced by the *InputFormat* to the available *Mappers*.

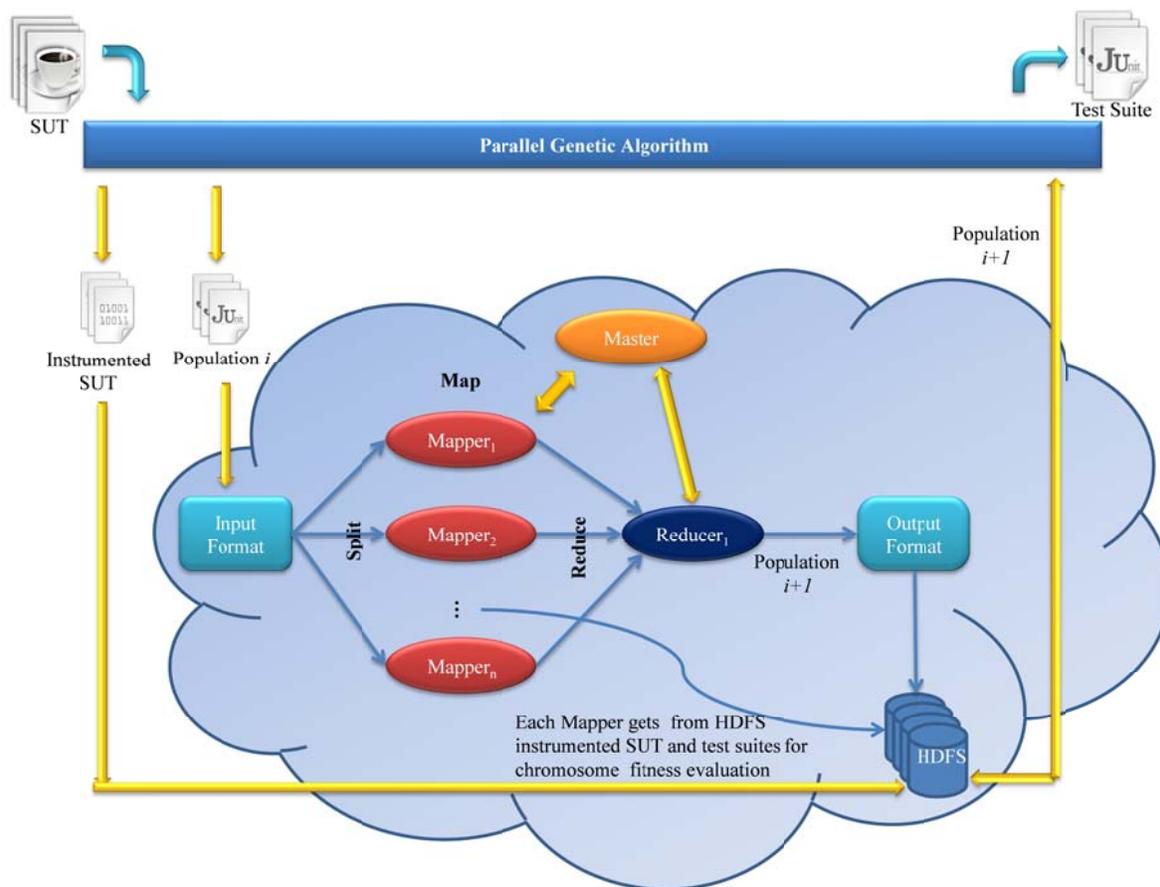


Figure 1. Architecture of the proposed PGA for test suite generation.

```

1 population = preExecution(SUT);
2 while(!stopCriteria ()) {
3   mpJob= createMapReduceJob();
4   population= mpJob.execute(population);
5 }
6 postExecution();

7 preExecution (SUT):population {
8   instrumentedCode = instrument (SUT);
9   move(instrumented Code, HDFS);
10  population = createRandomPopulation();
11  return population;

```

Figure 2. Parallel Genetic Algorithm pseudocode.

Map. In this phase each *Mapper* carries out its task on the received input split in a parallel and independent way. In particular, each *Mapper* is responsible to exercise a class of the SUT with the test suites associated to the received chromosomes and to observe the software behavior in order to evaluate the corresponding fitness value (i.e., branch coverage).

Once such evaluation is completed, each *Mapper* generates a new pair $\langle key, value \rangle$, where *value* is a pair $\langle chromosome, fitness\ value \rangle$, while the new *key* will be used by the Master module to properly assign the *Reducers*. Since our architecture does not require any other parallel computation, the generated key will be the same for all the chromosomes, in order to have only a single *Reducer*.

The *Master* module in this phase is responsible to collect the outputs produced by the *Mappers* that will constitute the input for *Reducer*.

Reduce. As soon as a *Mapper* evaluates a chromosome the corresponding data (i.e., key, chromosome, and fitness value) is sent to the *Reducer*. Once the entire population has been available to the *Reducer* it can perform the survival selection and apply on the new generation the crossover and mutation operators to produce a new offspring to be evaluated in the next MapReduceJob. Note that to obtain the entire population the *Reducer* should wait until all *Mappers* have replied. However each *Mapper* evaluates approximately the same number of chromosomes and if they are executed in parallel they require approximately the same time. Finally, the data concerning with the new offspring is saved by the *OutputFormat* - using the *RecordWriter* - into the HDFS, allowing the *Parallel Genetic Algorithm* module to verify whether the stopping criteria hold.

The *Master* module in this phase is responsible to notify *Parallel Genetic Algorithm* to restart the computation invoking the MapReduceJob for the new offspring created by *Reducer*.

IV. PRELIMINARY EVALUATION

In this section we report the results achieved in a preliminary analysis carried out to assess the speed-up achieved by using the Parallel Genetic Algorithm described in section III with respect to the sequential Genetic Algorithm described in section II (denoted in the following as SGA).

TABLE I. APACHE COMMONS PRIMITIVES PACKAGES

Package	Description
org.apache.commons.collections.primitives	Collections of primitive values.
org.apache.commons.collections.primitives.adapters	Adapters for converting between the primitive and object based versions of the collections framework.
org.apache.commons.collections.primitives.adapters.io	Adapters for converting between primitive collections and Java I/O classes.
org.apache.commons.collections.primitives.decorators	Decorators of primitive collections.

TABLE II. THE CASE STUDY SUBJECT.

Subject	# Classes	# Branches	LOC
Commons Primitives (CP)	<i>Total</i>		
	259	1446	4605
	<i>Testable</i>		
	89	1024	2826

A. Subject

The proposed approach was empirically evaluated by applying it to an open source software library, namely Apache Commons Primitives [5], which contains collections and utilities specially designed for use with primitive types in Java (i.e., boolean, byte, char, double, float, int, long and short). Indeed, despite a lot of functionality is provided by the JDK for Object types, relatively little functionality is provided for primitive types. Thus, Apache Commons Primitives library addresses this by providing a set of utility and collection classes for primitives (e.g., ArrayByteList, ArrayShortList, ArrayCharList, ByteIterator). In particular, it is organized in four packages containing 259 classes, for a total of 4605 Lines of Code (LOC) and 1446 branches (see Table I). As done in previous work [16], we tested all the classes which were not interface, abstract, or private (see Table II).

B. Evaluation Criteria

To compare the performance of the employed algorithms we evaluated them both in terms of execution time and branch coverage. In particular, the execution time was measured using the system clock and the TotalTime is composed of the following parts:

- *InitTime*: total time required by PGA for initializing a Map with the information (i.e., SUT instrumented bytecode, JUnit, test cases) needed to perform the fitness evaluation in each generation;
- *EvalTime*: total time spent to evaluate the fitness of chromosomes;
- *RemainTime*: time given by the difference between TotalTime and (InitTime+EvalTime); it comprises the time required to create the initial random population, perform selection, crossover, and mutation operators and, in case of PGA also the time spent for the communication among nodes.

The speed-up is then calculated by dividing the total amount of time that SGA required by the amount of time required by PGA.

It is worth noting that we executed ten runs in order to cope with the inherent randomness of dynamic execution time and of the GA, and reported the average results.

Moreover, as sanity check we compared the branch coverage achieved by SGA and PGA with the one obtained by randomly generating the same number of solutions (i.e., test suites) evaluated by SGA and PGA.

Experimental Setup

To set up SGA we exploited different settings for population size and generation number analyzing whether exploring a wider search space allowed us to achieve more accurate coverage results. In particular, we started experimenting with small setting values (i.e., 25 chromosomes and 10 iterations) and increasing them until we found a setting which allowed SGA to cover at least 75% of branches. The resulting setting employed a population of 225 chromosomes, each containing 6 test cases with a number of statements ranging from 2 to 20. The search process was stopped after performing 100 generations or whether the best solution did not change for 10 generations. The same setting was employed for PGA to allow for a fair comparison.

As for the employed hardware, SGA was executed on a node equipped with the configuration reported in Table II, while for PGA we exploited nine Maps and one Reducer distributed in a small cluster of three nodes for a total of six cores. The employed version of Hadoop was 0.21 and we also exploited Cygwin 1.7.1 in order to run Hadoop on Windows Operating System.

C. Results

Figure 3 reports on the execution time for both SGA and PGA. As we can see, the total execution time (i.e., TotalTime) is highly reduced by using PGA, thus allowing us to achieve a speed-up of 57% with respect to the use of SGA. Moreover, we can observe that the time required to initialize each Map (i.e., InitTime) is much less than the RemainTime, whose main part consists of the overhead due to data management during the reduce phase (i.e., using HDFS to move at each iteration the new population to the Master node). This suggested us that trying to diminish such an overhead by means of cloud computing or graphic cards could allow us to further reduce the total execution time. Regarding the achieved branch coverage, as we expected using PGA with the same setting of SGA allowed us to generate test suites able to cover in average about the same percentage of branches with respect the ones provided by SGA (i.e., 78%). Moreover, both algorithms achieved a better branch coverage with respect to the test suite randomly generated (i.e., 28 %).

Exploiting the speed-up provided by PGA in the future we could try to improve the coverage acting on the setting for population size and generation number.

TABLE III. EQUIPMENT OF THE EMPLOYED NODES

HARDWARE	CPU	Intel Core i3 2100
	RAM	4GB
	Hard Disk	SATA 500GB 5200RPM
	Connectivity	10/100/1000 Ethernet LAN
SOFTWARE	Operating System	Windows 7 Home Premium SP1 64bit
	Java Virtual Machine	Java SE Runtime v.1.6.
	JUnit	v. 4.10
	Hadoop	v. 0.21
	Cygwin	v. 1.7.1

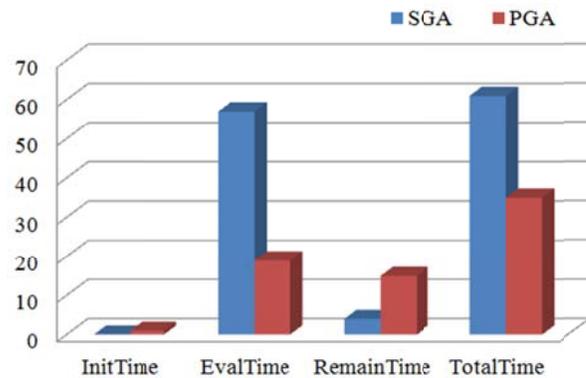


Figure 3. Execution time performance of SGA and PGA.

D. Threats to Validity

The focus of this paper was to propose the use of a parallel/distributed GA for automatic test suite generation and to conduct a preliminary investigation on its effectiveness. Thus, several improvements can be done in future work to mitigate the construct, internal, and external validity threats of this study. As for the construct validity the metrics employed to evaluate and compare the performance of the testing techniques were based on coverage criteria and execution time since the main goal of the study was to analyze the speed-up due to the use of PGA. However, they can be enriched exploiting for example other criteria to assess the quality of the generated test suite, or providing a fine-grained information on the RemainTime. As for the threats related to the internal validity, they can be due to the bias introduced by the intrinsic randomness of GA and to the employed GA setting. We mitigate these threats by using average results obtained from ten executions. As for the external validity, it can be affected by the fact that the subject of the study was an open-source software containing about 250 Java classes, thus it could be interesting taking into account further software projects of different sizes to confirm the achieved results.

V. RELATED WORK

In this section, we present some related works. First of all, we discuss previous works on the design of parallel Genetic Algorithms (not applied in the context of software engineering) based on MapReduce, showing similarity and differences with our proposal. Then, we describe solutions based on parallel search-based approaches employed for automated testing and for other software engineering problems. Finally, we report on several studies proposing different frameworks for the parallel execution of test cases.

A. Parallel Genetic Algorithms based on MapReduce

To the best of our knowledge, in the literature two proposals [22][37] have been reported proposing a parallel GA based on MapReduce to solve classical optimization problems. Nevertheless, such proposals did not take into account the specific context of software testing and differ from the one proposed in this paper under several aspects.

In [22] an extension of MapReduce, named MRPGA, was proposed. In that work the authors claimed that due to its iterative nature, the GA process cannot follow the two phases pattern of MapReduce. Thus, they added to the pattern a further reduction phase to perform a global selection at the end of each iteration of parallel GA. In particular, the architecture of the run time system consists of one master and multiple mappers and reducers. The master has the responsibility of scheduling the execution of parallel tasks, while the mapper workers have to execute the map functions (i.e., the evaluation of chromosome fitness defined by the user) and the reducer workers are responsible to execute reduce functions (i.e., the selection operation on chromosomes to choose local optimum individuals). The difference with the standard implementation of MapReduce concerns with the support provided for a second reduce phase that is conceived to use just a reducer responsible of the selection of the global optimum individuals. Moreover, a coordinator client is introduced to coordinate the executions of the parallel GA iterations. MRPGA was implemented on .NET platform using C# language and was applied to solve the DLTZ4 and DLTZ5 problems [14].

Our solution differs from the one of [22] since it does not employ the additional reduction phase, indeed, as suggested also by Verma *et al.*, this phase is not necessary since the local reduce can be implemented within a Combiner as shown in [13]. Further, using “default_key” and 1 as values produced by the mapper, reducer, and final reducer functions, MRPGA does not employ any characteristics of the MapReduce model (i.e., the grouping by keys and the shuffling). Moreover, in their proposal a huge amount of work regarding mutation, crossover, and evaluation of the convergence criteria is made by a single coordinator affecting the scalability of their approach [37]. To avoid this problem, in our solution we split these jobs among the master node (performing evaluation of the convergence criteria) and the reducer (performing mutation and crossover).

Differently from [22], Verma *et al.* [37] designed a Parallel GA based on the traditional MapReduce model and realized it by exploiting Hadoop MapReduce [4]. The

proposed PGA was employed to solve the ONEMAX problem [32] showing that larger problems could be resolved without any changes in the implementation of the algorithms by adding more resources. Our solution is similar to [32], except for the use of multiple reducers. Indeed, they followed a coarse-grained parallelization model performing a local survival selection on multiple reducers to speed-up the overall execution time, however as pointed out in [31] a local selection can lead to a reduction in selection pressure affecting the time taken to converge. Thus, since in the domain we considered the computation of the fitness function is the most time consuming task we preferred employing a global parallelization level using only one reducer. However, if a higher level of parallelization is required, the architecture we proposed can be extended, basically acting on the number of the involved reducers and the intermediate keys, to parallelize also the execution of selection and genetic operators (i.e., coarse-grained parallelization) or to enhance the selection pressure by means of migrations among groups of chromosomes (i.e., fine-grained parallelization) as described in [15].

B. Parallel Genetic Algorithms for Automated Testing

Search-based algorithms have been used to automate a variety of software testing activities, such as test data generation (e.g., [2][21][24][36]), test case generation and selection (e.g., [16][39]), test case prioritization (e.g., [38]), and so on. For sake of space we remind interested readers to [1] and [27] for a review of these works and we discuss in the following the parallel Search-based approaches proposed for Software Testing.

In [15] the use of MapReduce has been recently suggested for addressing the computational issues related to SBST. In particular, the authors described three architectures based on MapReduce to achieve different levels of parallelization of Genetic Algorithms (i.e., global, island, and grid models). Moreover, they illustrated with an example how to design a Parallel Genetic Algorithm based on Google AppEngine-MapReduce [18] for test data generation. As they stated, the proposal needs to be validated in practice carrying out an empirical evaluation of the solutions; this is necessary to highlight on the field the strength or weakness of the different architectures, as well as to assess their actual scalability. Thus, in this paper we designed a Parallel Genetic Algorithm conceived for JUnit test suite generation based on the global parallelization model proposed in [15] and realized it exploiting the Hadoop MapReduce framework reporting also a preliminary evaluation of its use on standard cluster. With respect to AppEngine-MapReduce [18], Hadoop MapReduce provides several advantages as described in section III-C.

A different approach to address parallelization of SBST has been recently proposed by Yoo *et al.* in [40], where a parallel multi-objective Genetic Programming for test suite minimization was devised for exploiting the computational power of modern graphic cards. The obtained results showed that for their problem the speed-up achieved by using GPU was logarithmic correlated to the problem size (i.e., SUT and

test suite size) and ranged from 1x to 25x with respect to the use of a single-threaded version of the same GP.

C. Parallel Search-Based Approaches in Software Engineering

As for the use of parallel solutions in other areas of SBSE, Mitchell *et al.* [28] suggested the exploitation of a distributed architecture to parallelize modularization through the application of search-based clustering.

To the same end, Mahdavi *et al.* [26] developed a parallel hill climbing algorithm exploiting a cluster of standard PCs.

More recently, Asadi *et al.* [6] compared different distributed architectures to parallelize a GA for the concept location problem.

D. Parallel Execution of Testing

Since software testing is one of the most expensive phases of the software development process, in the last decades a huge amount of research efforts has been devoted to speed-up testing activities. Nevertheless, despite the general advantages of parallelizing software testing [33], very little work has been made to distribute software testing over multiple computers.

In the case study presented by Lastovetsky and Alexey [25] it was shown that parallelizing regression testing for a distributed programming system resulted in a speed up of up to 7.7 on two 4-processor workstations.

These promising results encouraged also the development of parallel regression testing tools based on JUnit, such as Joshua [23] and GridUnit [9][10][11]. The main idea underlying these tools was that a master node distributes test cases for execution across slave machines to speed-up the testing process and then collects the results. To this end Joshua exploited Jini for distributing the regression test suite execution over different CPUs, while GridUnit exploited a computational Grid.

In [30] a distributed execution framework for JUnit test cases, named HadoopUnit, was proposed and preliminary results using a 150-node cluster suggest that HadoopUnit can reduce test execution time significantly (about 30x improvement). Recently high interest has been attracting the use of cloud computing for addressing the significant computing resources and the lengthy execution times requested by software testing. Paper [35] takes into account the characteristics of an application under test and the types of testing to perform to decide when migrating software testing to the Cloud. A proposal for distributing the execution of test cases in the Cloud was presented in [29]. In particular, they described a framework for the distributed execution of the York Extensible Testing Infrastructure (YETI), a language agnostic random testing tool [29]. They employed the MapReduce primitives. Before the execution, the needed files (i.e., the test cases and the employed testing tool) are uploaded to the distributed file system file to be later read by the Mapper nodes. Then, the Master node launches a Mapper for each test case and each Mapper reads its data and executes the corresponding test case. Finally, the Reducer collects the test case results from each Mapper and outputs them to a file on the DFS. The preliminary results

reported in [29] were promising, showing that exploiting the proposed framework on the Amazon Elastic Computing Cloud (EC2) [3] the performances of YETI improved reducing the testing time.

VI. CONCLUSION AND FUTURE WORK

In this paper we proposed the use of a Parallel Genetic Algorithm (PGA) for test suite generation exploiting Hadoop MapReduce and showed a preliminary evaluation of its use on a small cluster. The obtained results highlighted that using PGA allowed us to save over the 50% of time.

Since the use of parallel SBST approaches is still in its early phases, several directions can be prospected as future work. First of all a deeper empirical evaluation of the proposed approach is needed to assess on other subjects its strength or weakness, as well as to assess its actual scalability employing different GA settings, numbers of maps, and larger clusters. Also the use of Hadoop MapReduce should be assessed running it not only on standard clusters, but also exploiting cloud computing and graphic cards. Moreover, it can be interesting to realize and compare higher levels of parallelization, such as by parallelizing the genetic operations other than the fitness evaluation.

It would be also interesting to verify how other approaches for test suite generation (e.g., EvoSuite [16]) could be take advantages of parallel/distributed computation as the one described in this paper.

Finally, as a long-term research goal, it will be desirable to integrate these SBST approaches within a whole Validation-as-a-Service platform, available in the Cloud, to support the entire software testing process.

ACKNOWLEDGMENT

The research has been carried out exploiting the computer systems funded by University of Salerno's Finanziamento Medie e Grandi Attrezzature (2010).

REFERENCES

- [1] S. Ali, L. C Briand, H. Hemmati, R. K., Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," IEEE Transactions on Software Engineering, 36(6) 2010, pp. 742-762.
- [2] S. Ali, M. Z. Z. Iqbal, A. Arcuri, L. C. Briand, "A Search-Based OCL Constraint Solver for Model-Based Test Data Generation," in Proceedings of the 11th International Conference On Quality Software, 2011, pp. 41-50.
- [3] Amazon Elastic Compute Cloud, <http://aws.amazon.com/ec2>
- [4] Apache Hadoop Map Reduce, <http://hadoop.apache.org/mapreduce/>
- [5] Apache Commons Primitives, <http://commons.apache.org/primitives/>
- [6] F. Asadi, G. Antoniol, Y. Gueheneuc, "Concept locations with genetic algorithms: A comparison of four distributed architectures," in Proceedings of the 2nd International Symposium on Search Based Software Engineering, 2010, pp. 153 - 162.
- [7] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in Proceedings of Future of Software Engineering, 2007, pp. 85-103.
- [8] Cobertura, Coverage Tool, <http://cobertura.sourceforge.net/>

- [9] A. Duarte, W. Cirne, F. Brasileiro, P. Machado, "Gridunit: software testing on the grid," in Proceedings of the 28th International Conference on Software engineering, 2006, pp. 779-782.
- [10] A. Duarte, G. Wagner, F. Brasileiro, W. Cirne, "Multienvironment software testing on the grid," in Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging, 2006, pp. 61-68.
- [11] R. N. Duarte, W. Cirne, F. Brasileiro, P. Duarte, and D. L. Machado, "Using the computational grid to speed up software testing," in Proceedings of 19th Brazilian Symposium on Software Engineering, 2005.
- [12] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters," in Proceedings of the 6th Symposium on Operating System Design and Implementation, 2004, pp.137-150.
- [13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", *Communications of ACM*, 51(1):107-113, 2008.
- [14] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, "Scalable multiobjective optimization test problems," in Proceedings of Congress on Evolutionary Computation, 2002, pp. 825-830.
- [15] S. Di Martino, F. Ferrucci, C. Gravino, V. Maggio, F. Sarro, "Using MapReduce in the Cloud to enhance effectiveness and scalability of genetic algorithms for test data generation," available at <http://www.dmi.unisa.it/people/sarro/www/research/SBSTCloud.html>
- [16] G. Fraser, A. Arcuri, "Evolutionary generation of whole test suite," in Proceedings of the 11th International Conference On Quality Software, 2011, pp. 31-40.
- [17] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley, 1989.
- [18] GoogleAppEngine, <http://code.google.com/appengine/>
- [19] CUDA on Hadoop MapReduce, <http://wiki.apache.org/hadoop/CUDA%20On%20Hadoop>
- [20] M. Harman, "The current state and future of search-based software engineering," in Proceedings of Future of Software Engineering 2007, pp. 342-357.
- [21] M. Harman, P. McMinn, "A theoretical and empirical study of search based testing: local, global and hybrid search," *IEEE Transactions on Software Engineering*, 36(2), 2010, pp. 226-247.
- [22] C. Jin, C. Vecchiola, R. Buyya: "MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms," in Proceedings of IEEE Fourth International Conference on eScience, 2008, pp. 214-221.
- [23] G. M. Kapfhammer, "Automatically and transparently distributing the execution of regression test suites," in Proceedings of the 18th International Conference on Testing Computer Software, 2000, pp. 149-174.
- [24] K. Lakhota, P. McMinn, M. Harman, "Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?," in Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009, pp. 95-104.
- [25] A. Lastovetsky, "Parallel testing of distributed software," *Information Software Technology*, 47(10) 2005, pp. 657-662.
- [26] K. Mahdavi, M. Harman, R. M. Hierons, "A multiple hill climbing approach to software module clustering," in Proceedings of International Conference on Software Maintenance, 2003, pp. 315-324.
- [27] P. McMinn, "Search-based software test data generation: a survey," *Journal of Software Testing, Verification & Reliability* 14(2), 2004, pp. 105-156.
- [28] B. S. Mitchell, M. Traverso, and S. Mancoridis, "An architecture for distributing the computation of software clustering algorithms," in Proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 181-190.
- [29] M. Oriol, F. Ullah, "YETI on the Cloud", in Proceedings of the Third International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 434-437.
- [30] T. Parveen, S. Tilley, N. Daley, P. Morales, "Towards a distributed execution framework for JUnit test cases," in Proceedings of IEEE International Conference on Software Maintenance, 2009, pp. 425-428.
- [31] A. J. Sarma, J. Sarma, and K. D. Jong, "Selection pressure and performance in spatially distributed evolutionary," in Proceedings of the World Congress on Computational Intelligence, 1998, pp 553-557. IEEE Press.
- [32] J. Schaffer, L. Eshelman, "On Crossover as an Evolutionary Viable Strategy," in Proceedings of the 4th International Conference on Genetic Algorithms, Morgan Kaufmann, 1991, pp. 61-68.
- [33] E. Starkloff, "Designing a parallel, distributed test system," *Aerospace and Electronic Syst Systems Magazine, IEEE*, 16(6), 2001, pp. 3-6.
- [34] J. Stender, "Parallel Genetic Algorithms: Theory and Applications," *Frontiers in Artificial Intelligence*, 1993, vol.14.
- [35] S. R. Tilley, T. Parveen, "When to Migrate Software Testing to the Cloud?," in Proceedings of Third International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 424-427.
- [36] P. Tonella, "Evolutionary testing of classes," in Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis 2004, pp. 119-128.
- [37] A. Verma, X. Llorà, D.E. Goldberg, R.H. Campbell, "Scaling Genetic Algorithms Using MapReduce," in Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications, 2009, pp.13-18.
- [38] K.R., Walcott, M.L., Soffa, G.M. Kapfhammer, R.S. Roos, "Time aware test suite prioritization," in Proceedings of the International Symposium on Software Testing and Analysis, 2006, pp. 1-12.
- [39] S. Yoo, M., Harman, "Pareto efficient multi-objective test case selection". In Proceedings of International Symposium on Software Testing and Analysis, 2007, pp, 140-150.
- [40] S. Yoo, M. Harman, S. Ur "Highly scalable multi objective test suite minimisation using graphics cards," in Proceedings of Third International Symposium on Search Based Software Engineering, 2011, pp. 219-236.