Chapter 6
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

Sergio Di Martino
University of Naples Federico II, Italy

Filomena Ferrucci
University of Salerno, Italy

Valerio Maggio
University of Naples Federico II, Italy

Federica Sarro
University of Salerno, Italy

ABSTRACT
Search-Based Software Testing is a well-established research area, whose goal is to apply meta-heuristic approaches, like Genetic Algorithms, to address optimization problems in the testing domain. Even if many interesting results have been achieved in this field, the heavy computational resources required by these approaches are limiting their practical application in the industrial domain. In this chapter, the authors propose the migration of Search-Based Software Testing techniques to the Cloud aiming to improve their performance and scalability. Moreover, they show how the use of the MapReduce paradigm can support the parallelization of Genetic Algorithms for test data generation and their migration in the Cloud, thus relieving software company from the management and maintenance of the overall IT infrastructure and developers from handling the communication and synchronization of parallel tasks. Some preliminary results are reported, gathered by a proof-of-concept developed on the Google’s Cloud Infrastructure.

DOI: 10.4018/978-1-4666-2536-5.ch006
INTRODUCTION

The software testing encompasses a range of different activities that are critical for software quality assurance. For each activity, depending on the testing objective, specific test cases need to be devised to check the system (Bertolino, 2007). Since an exhaustive enumeration of software inputs is unfeasible for any reasonable-sized system (McMinn, 2004), a careful selection of test data must be performed to obtain a high testing effectiveness. This task is often difficult, time-consuming, and error-prone. Thus, the need to decrease time and costs of software testing, while increasing its effectiveness has motivated the research for advanced techniques able to automatically generate test data. This is an active research area and a number of different approaches has been proposed and investigated in the literature (e.g., (Ali, 2010; Bertolino, 2007; De Millo, 1991; Miller, 1976)).

Among them, Search-Based techniques (Harman, 2007) are promising approaches to increase testing quality, by automatically generating relevant test data (Harman, 2001). Search-Based techniques include a variety of meta-heuristics, such as Local Search (i.e., Hill Climbing, Tabu Search, Simulated Annealing, etc.), Evolutionary Algorithms (i.e., Genetic Algorithms, Evolution Strategies, Genetic Programming, etc...), Ant Colony Optimization, or Particle Swarm Optimization. All these meta-heuristics search for a suitable solution in a typically large input space guided by a fitness function which expresses the goals and leads the exploration into potentially promising areas of the search space. Thus, using these approaches, test data generation is treated as a search or optimization problem whose goal is to find the most appropriate input data conforming to some adequacy criteria (i.e., test goals/objectives), such as maximizing the code coverage. Thus, moving from conventional manual test data definition to Search-Based test data generation essentially consists in defining a suitable fitness function to determine how good a test input is.

The generic nature of these metaheuristics let them to be fruitful for different testing goals and issues, simply by redefining the solution representation and the fitness function. Thus, in the last decades there has been an explosion of researches on the use of Search-Based techniques for software testing that have addressed a range of testing problems, giving rise to a very active research field, known as Search-Based Software Testing (SBST). These techniques have been used for structural testing (both static and dynamic) functional testing (both for generating test data and testing the conformance of the implementation to its specification), non-functional testing (e.g., testing for robustness, stress testing, security testing, gray-box testing (e.g., assertion testing and exception condition testing), state–based testing, regression testing, interaction testing, integration testing, test case prioritization, and so on (McMinn, 2004; Ali, 2010). Despite all such efforts, so far these investigations have produced limited impact in industry (Bertolino, 2007). Maybe the main reason for that lies in the fact that few attempts have been made to improve performance of these techniques and make them more scalable. On the other hand, while several empirical studies have been carried out showing that Search-Based testing can outperform other automated testing approaches (e.g., random testing), little attention has been deserved to the scalability and effectiveness issues for real world applications (Ali, 2010). One of the few experimentation with complete open source applications reported that there are still many challenges to be addressed to make existing Search-Based test data generation tools robust and with a significant level of code coverage that can be useful for industrial use (Lakhota, 2009).

The use of Cloud Computing can provide a significant impulse in this direction. Indeed, this model is based on the provisioning of configurable computing resources in a distributed environment allowing for an on-demand resource allocation from a virtual unlimited resources and infra-
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

structure functionality. Thus, with respect to the use of traditional cluster-based platform, Cloud Computing allows for easy scalability in a cost effective way since it eliminates unnecessary purchases, allowing one to pay only for the resources actually used, and does not require management and maintenance of the overall IT infrastructure.

As a consequence, the migration of SBST techniques to the Cloud leads to two main advantages for software companies:

1. Achieving a greater efficiency and scalability, thus reinforcing the cost-effectiveness of these testing approaches.
2. Allowing for the exploration of solution spaces larger than those considered by canonical sequential Search-Based techniques, by means of an easy-to-use parallelization mechanism.

In this chapter, we focus on the problem of test data generation and the use of Genetic Algorithms. We start by providing background on Genetic Algorithms and its application to software testing. Then, we describe three possible architectures for migrating Genetic Algorithms to the Cloud exploiting the MapReduce paradigm (Dean, 2004). Moreover, we provide a proof-of-concept, with some initial experimental results, based on Google’s Cloud infrastructure, namely App Engine (Google, 2012). Some indication on future research directions and final remarks close the chapter.

BACKGROUND: GENETIC ALGORITHM AND ITS USE FOR TEST DATA GENERATION

Genetic Algorithms (GAs) are meta-heuristics that simulate the evolution of natural systems, emphasizing the principles of survival of the strongest to solve, or approximately solve, optimization problems. Thus, these algorithms create consecutive populations of individuals, considered as feasible solutions for a given problem, to search for a solution that provides a good approximation of the optimum for the problem under investigation. In the literature several studies have reported that this approach consistently outperforms traditional optimization methods for many classes of problems (Goldberg, 1989; Harman, 2001).

In the implementation of a GA, there are several key aspects that have to be considered for its application to any given optimization problem (Harman, 2001). A crucial role is played by the way a solution is represented for the given domain. In general, such a solution is encoded by a fixed length binary string, called chromosome (in analogy with the biological equivalent), that embeds values for the variables involved in the problem. Each chromosome is evaluated by means of a properly defined fitness function that gives an indication of its goodness in solving the problem. Clearly, the choice of a good function plays a central role in the definition of a suitable GA. Consequently, in general, the elementary process of each GA is the following:

1. A random initial population, i.e. a family of chromosomes, is generated.
2. A new population (i.e. a generation), intended as a set of new chromosomes, is created, starting from the previous one, by applying genetic operators (e.g., crossover, mutation) to a subset of properly chosen chromosomes, in order to explore other solutions in the search space.
3. The second step is repeated until either (1) the fitness of the best solution has converged or (2) a maximum number of generations has been reached. The chromosome with the best solution in the last generation is taken, giving the best approximation to the optimum for the problem under investigation.
Summarizing, in the application of GAs, the following issues have to be addressed (Harman, 2001):

- Defining the way a chromosome encodes a solution and choosing the number of considered chromosomes (i.e., the population size).
- Defining the fitness function to measure the goodness of a chromosome.
- Defining how to apply genetic operators to generate new offspring.
- Defining the halting criteria.

For test data generation, the search space consists in the set of the possible inputs to the program (Harman, 2001; Srivastava, 2009). Consequently, each chromosome represents a specific set of values for the involved parameters and has to be encoded in a way that allows for mutation and crossover operations. The goodness of each chromosome is measured as the conformance to a given adequacy criterion, defined by the tester, and depends on the test problem under investigation. As an example, in the case of white box testing, fitness functions based on structural testing measures can be exploited to get indications on the goodness of chromosomes. The most employed code coverage measures are (Glenford, 2004):

- **Statement Coverage**: Measures the ratio of covered statements with respect to the total number of statements. A statement is covered if it is exercised at least once.
- **Decision (or Branch) Coverage**: Measures the ratio of covered decisions with respect to the total number of decisions. A decision is covered if it is exercised on the true and false outcomes, at least once.
- **Condition Coverage**: Measures the ratio of covered conditions with respect to the total number of conditions. A condition is covered if each of its decisions takes on all possible outcomes at least once.

A general schema related to the use of GAs as test data generators is shown in Figure 1. Once the tester has defined the specific GA identifying the above aspects (e.g., the evaluation criterion, encoded within a fitness function), the GA generates test data to exercise the SUT, leading to an evaluation of the fitness value. It is worth noting that for white box testing, this observation can be done by properly utilizing the code (i.e., the

---

**Figure 1. A generic GA scheme for test data generation**

![Figure 1. A generic GA scheme for test data generation](image)
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

software execution must be traced), while for black box testing it is necessary to look at the obtained results. Once the evolutionary process of GA is terminated, the chromosome giving the best fitness value for the defined criterion is returned in output to the tester as the best set of test data.

MIGRATING GENETIC ALGORITHMS FOR SBST TO THE CLOUD

In this section we start by discussing how to parallelize GAs together with the main technical difficulties that must be faced. Then, we present different ways to exploit the Cloud Computing and the MapReduce model to achieve distributed GAs for software test data generation.

Parallelizing GAs for SBST

To date, SBST solutions have found limited application in industry (Bertolino, 2007). As in other fields of Search-Based Software Engineering, this is principally due to the fact that in general when applied to large problems, Search-Based approaches (including GA) may require too much computational efforts (Harman, 2007) and “many approaches that are attractive and elegant in the laboratory, turn out to be inapplicable in the field, because they lack scalability” (Harman, 2007).

In this scenario, parallelization may be a suitable way to improve SBST performance, both in terms of computational time and effectiveness in the exploration of the search space. Let us note that GAs are “naturally parallelizable” (Harman, 2007) since their population based characteristics allows to evaluate in a parallel way the fitness of each individual, with a minimal overhead.

When dealing with GAs applied in the field of SBST, the benefits of the parallelism can be twofold. From one hand, it can be used to compute in a concurrent fashion how much each chromosome exercises the SUT. On the other hand, it can be used to perform genetic operators and thus to generate the next set of test data. Furthermore, these two strategies can be combined, giving rise to the following three main grains of parallelization:

• 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)

In the global parallelization model there are two main roles: a Master and some Slaves. The former is responsible to manage the population (i.e., apply evolutional operators) and to assign the individuals to the slaves. 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 can be achieved in parallel.

In the island model the initial population is split in several groups and on each of them, typically referred to as “island,” the GA proceeds independently and periodically exchanges information among islands by “migrating” some individuals from one island to another. The main advantages of this model are that (1) different subpopulations can explore different parts of the search space, and (2) migrating individuals among islands enhances diversity of the chromosomes, thus reducing the probability to converge into a local optimum.

The grid model extends the island one by enhancing the selection pressure by means of migrations among groups of chromosomes. This is obtained by assigning each individual to a single node and by performing evolutionary operations that involve also some neighbors of a solution. 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. The consequence of these models is that to date the
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

Parallelization of GAs is straightforward from a conceptual point of view. However, setting up an actual implementation may be not so trivial due to some common development difficulties that a programmer must tackle in a distributed environment. Probably these limitations have slowed the use of parallel GAs in software testing. In fact, to the best of our knowledge, there are in the literature few attempts to exploit parallel GAs for SBST (Alba, 2008; Harman, 2011; Di Gironimo, 2012).

We can distinguish two main classes of problems in the development and use of parallel GAs:

1. The need to devote extensive efforts to manage and maintain the overall IT infrastructure.
2. The need to handle the communication and/or synchronization among different modules.

The use of Cloud Computing may be a means of addressing the first issue since the Cloud infrastructure lets an on-demand resource handling and allocation, relieving the developers from the management of the distributed IT infrastructure.

As for the second issue, the use of the MapReduce paradigm can be a solution, as it allows for disregarding of taking care the coordination and execution of parallel tasks.

In the next section, we provide a proposal to exploit GAs for SBST in a Cloud-based environment using the MapReduce model.

Cloud Computing Technologies

Cloud Computing has been defined as “[…] a model for enabling convenient, on-demand network access to a shared pool of configurable resource (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” (Mell, 2009). Outsourcing IT commodities to a Cloud Computing provider can bring many benefits for a business. Among them (and apart from economical aspects) there is a better availability of resources, due to the high redundancy of computational power in the data centers and an excellent scalability. Moreover, the Cloud is usually based on a pay-per-use concept where additional computational resources can be allocated on demand. Also from an ecological point of view, the Cloud solution can bring many benefits, since with this paradigm, different enterprises share computing power as well as data storage, with a drastic improvement of energy efficiency.

Today there are many Cloud providers, offering different kinds of services, ranging from the Infrastructure-as-a-Service (IaaS) up to the Software-as-a-Service (SaaS) solutions (Mell, 2009). Some of them support the development of Cloud-based applications delivering a computing platform, such as Amazon Elastic MapReduce (Amazon, 2012), Google App Engine (Google, 2012), or Microsoft Azure (Microsoft, 2012). These typically include operating system, programming language execution environment, database, and web server so that developers can build and run their software solutions on a Cloud platform without managing the underlying hardware and software layers. Moreover, they do not have to allocate resources manually for satisfying application demand since the compute and storage resources can scale automatically. This allows developers to focus mainly on their application features, without taking care of extensive deployment and/or configuration effort, e.g. setting up a server or replacing a failed hardware component.

In the following we briefly describe Google App Engine (GAE from now on) since it is the platform we employed in our study. GAE enables developer to build web applications by means of a well-documented SDK available in Java, Python, and Go programming languages (Google, 2012). GAE is also responsible to handle tasks such as load balancing and fault tolerance; it provides to user’s applications the same large-scale services that power Google’s applications for data storage, caching and network access. The main components of GAE are: the runtime environment, which is
The MapReduce Model

The most straightforward solution to integrate GAs into the Cloud could employ the existing implementations of parallel GAs, usually based on the Message Passing Interface (MPI). However, MPI-based solutions are not supported by all the Cloud platforms (e.g., Microsoft Azure) and have many drawbacks, mainly due to the fact that they do not scale well on multiple clusters (Verma, 2009) and do not provide any mechanism to support fault tolerance. As a consequence, if a computation fails during execution, the whole program needs to be restarted. Moreover, as these solutions are not supposed to work in a Cloud-based environment, they are not able to exploit the benefits provided by the infrastructure, such as an on-demand resource usage allocation. On the other hand, several Cloud providers support the use of the MapReduce model, which is an elegant, flexible, and highly scalable paradigm proposed to enable users to develop large-scale distributed applications by parallelizing computations and resource usage (Dean, 2004). Moreover, the MapReduce implementations available in the Cloud exploit the underlying infrastructure that is in charge of managing load balancing, resource allocation, fault tolerance, job startup, etc. Thus, this model and the corresponding implementations can represent a viable solution to integrate GAs into the Cloud.

In the following, we report the main aspects of this paradigm and the main implementations supported in the Cloud.

MapReduce relies on the definition of only two distinct functions, namely \textit{Map} and \textit{Reduce} (Dean, 2004). These functions are combined together in a divide-and-conquer way, so that the former is responsible to handle the parallelization while the latter collects and merges the results.

It is worth noting that, even if these two functions represent the main core of the model, their implementations are strongly related to the particular problem at the hand. As a consequence, while

---

Table 1. Instance classes provided by Google App Engine (GoogleDeveloper, 2012)

<table>
<thead>
<tr>
<th>Class</th>
<th>Memory</th>
<th>CPU</th>
<th>Cost per Hour per Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>128MB</td>
<td>600MHz</td>
<td>$0.08</td>
</tr>
<tr>
<td>F2</td>
<td>256MB</td>
<td>1.2GHz</td>
<td>$0.16</td>
</tr>
<tr>
<td>F4</td>
<td>512MB</td>
<td>2.4GHz</td>
<td>$0.32</td>
</tr>
</tbody>
</table>
the MapReduce model supplies the abstraction of the overall parallel computational infrastructure, each problem must be integrated into the model by implementing these two functions, namely by expressing its execution in terms of these two functions. A typical computation in the MapReduce model encompasses the following three steps:

1. The program splits the input data into a set of partitions which are identified by a unique key. This aspect is crucial since each single split constitutes the input for the next step and consequently determines how different Map function invocations are distributed across multiple machines.

2. When all the data has been collected and split, the program assigns each group and its corresponding key to a single Map task. Each Mapper is executed independently and produces a set of intermediate key/value pairs. The way these keys are created determines the distribution of the work among Reducers on different machines since the MapReduce model automatically allocates a new Reducer for each unique key. Moreover, Mappers typically use re-execution as technique to support fault tolerance (Verma, 2009).

3. Once all the Mappers terminate, the program is notified and it gets in charge to invoke the Reduce tasks. Each Reducer is responsible to group together all the intermediate values associated to the same key and to compute the list of output results.

The overall computation of the MapReduce model, as presented in the seminal paper (Dean, 2004), is depicted in Figure 2.

There are some well-known implementations of MapReduce supported in the Cloud, such as the App Engine MapReduce (GoogleMR, 2012) and the Hadoop MapReduce (HadoopMR, 2012). App Engine MapReduce is an open-source library for building applications compliant to the MapReduce model for Google App Engine.

Figure 2. The MapReduce overall computation (Dean, 2004)
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud (GoogleMR, 2012). It makes available an implementation of the Mapper’s functionality, while it is required to write the code for the Reducers. Moreover, it offers also an implementation of an InputReader and an OutputReader, two modules able to respectively read from and write to the DataStore provided by GAE. The 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 rapidly increasing due to its adoption by large companies such as IBM and Yahoo. With respect to App Engine MapReduce, Hadoop MapReduce provides both the Map and the Reduce modules, thus avoiding a developer to manage its own Reducer, and exploits a distributed file system, named Hadoop Distributed File System, to store data. It has been originally designed to work on Linux clusters, but it can now also run in the cloud thanks to services such as Amazon Elastic MapReduce (Amazon, 2012) and Windows Azure (Microsoft, 2011). Recently Microsoft has released to the research community an iterative MapReduce runtime, named Daytona (Daytona, 2011). It is designed to work on Windows Azure and provides a high-performance system for data analysis and machine learning algorithms. Similarly to App Engine MapReduce and Hadoop MapReduce, Daytona MapReduce uses Cloud storage services (i.e., Azure Blob Storage) for the dynamic data partitioning and Azure infrastructure services for robustness and dynamic scalability. In Table 2 we summarized the main characteristics of these MapReduce implementations supported in the Cloud.

Exploiting MapReduce and Cloud Infrastructure for SBST

In this section, we describe how to exploit MapReduce to support the parallelization of GAs for Search-Based Software Testing and its use in the Cloud. We propose three models which implements the levels of parallelization we described in the previous section (i.e., global, coarse-grained, and fine-grained). Let us note that in the following we refer to the App Engine-MapReduce since we employed it in our study (see next Section), but the proposal could be easily adapted to work also with other MapReduce implementations, such as Hadoop MapReduce.

The architecture for the global level of parallelization (i.e., parallelization of the fitness evaluation) using the MapReduce model is depicted

<table>
<thead>
<tr>
<th>Data Handling</th>
<th>Google App Engine MapReduce</th>
<th>Apache Hadoop MapReduce</th>
<th>Daytona MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduling</td>
<td>Node/Rack Aware. Dynamic task scheduling through global queue</td>
<td>Data Locality. Rack aware. Dynamic task scheduling through global queue</td>
<td>Dynamic task scheduling through global queue</td>
</tr>
<tr>
<td>Failure Handling</td>
<td>Re-execution of failed tasks</td>
<td>Re-execution of failed tasks</td>
<td>Re-execution of failed tasks</td>
</tr>
<tr>
<td>Environment</td>
<td>Google App Engine</td>
<td>Linux Clusters Amazon Elastic MapReduce on EC2 Compatible with Windows Azure</td>
<td>Windows Azure Compute, Windows Azure Local Development Fabric</td>
</tr>
<tr>
<td>Intermediate Data Transfer</td>
<td>Http</td>
<td>File, Http</td>
<td>File, TCP</td>
</tr>
</tbody>
</table>

Table 2. Comparison among MapReduce implementations supported in the Cloud
in Figure 3. The underlying idea is to delegate to each Mapper the burden of evaluating the fitness for each chromosome in parallel, while a single Reducer is responsible to collect the results and to perform the evolving operations, namely crossover and mutation, to generate the new offspring. In this architecture, we can highlight four main modules, i.e., the User Program, the Master, the Mappers and the Reducer, together with two other units, responsible to handle the I/O.

To give a deeper insight into the proposed architecture, in the following we describe the responsibility of each component depicted in Figure 3:

- The User Program is responsible to create an initial random population and to start the evolutionary process by invoking the InputReader. Moreover, it is also responsible to stop the process according to some termination criteria, such as population convergence or a max number of performed executions.
- The InputReader module is in charge to read the (possibly instrumented) source code of the SUT from the DataStore and to assign the chromosomes of the current generation to a bunch of Mappers. To parallelize the computation of the fitness function for each chromosome, we have to assign each of them to a different Mapper. This is achieved by generating a unique key for each chromosome, which will be calculated as a function of the triple \(<\text{chromosome}, \text{current generation}, \text{specific unit of the SUT}>\). It is worth noting that we suggest considering also the unit, in this way the model supports the application of multiple instances of GA to different units composing the SUT.
- The Mapper is responsible to evaluate each received chromosome exercising the

![Figure 3. The proposed architecture for the global parallelization approach](image-url)
unit of the SUT with the corresponding test data and observing the software behavior to compute chromosome fitness value (e.g., branch coverage). Then, the Mapper generates an intermediate pair <key, value>, where the key is needed to properly assign the Reducers, while value is the pair <chromosome, fitness value>. By assigning the same key to all intermediate pairs we can realize a global parallelization model, this forces the model to use exactly one Reducer.

- The Reducer module is invoked once all the Mappers terminate their execution. Once collected all the pairs, it applies the selection on the entire population and then the evolution operators, to obtain a new offspring.
- At the end of each iteration, the OutputWriter saves the data relative to the new offspring into the DataStore.
- The Master module manages the overall computation together with the resource assignment. As the InputReader begins to emit the <key, value> pairs the Master module assigns chromosomes to Mapper. Similarly, when Mappers emit intermediate <key, value> pairs the Master module is responsible to collect them to properly assign the input to the Reducer module. Finally, once the Reduced outputs the new offspring, the Master module notifies the User Program to check according to the stopping criterion if the computation should be terminated or restarted by invoking the InputReader on the new offspring.

Let us observe that the above architecture allows us to obtain a parallelization only of the fitness evaluation. This is not a limitation, since in the considered domain, the computation of the fitness function is usually the most costly task (Michael, 2001) requiring to execute the target program and to evaluate the fitness for each new chromosome created by the genetic algorithm.

If a higher level of parallelization is required, this architecture can be extended to achieve a coarse-grained level (i.e., to parallelize also the execution of selection and genetic operators). In this case, as shown in Figure 4, some modules of the above architecture remain unchanged (i.e., User Program, Reducer, OutputWriter, and Master), while others have to be modified, i.e., InputReader and Mapper. In particular, we have to define a partition strategy able to assign each island of chromosomes to a different Reducer. This can be obtained by modifying the algorithm necessary to generate the keys in the InputReader. Now the key is calculated considering also the island that a chromosome belongs to. As for the Mapper, it has to emit a pair <key, value> where value is computed as in the previously described architecture, while key is now function also of the island containing the chromosome. As a result, each chromosome belonging to the same island, in the same generation and exercising the same code, will be fed to the same Reducer. Also in this architecture, the Reducers are responsible to apply selection and evolution operators for the new offspring, making it available to the OutputWriter. Finally, since the coarse-grained model strongly relies on the concept of migrating chromosomes among islands, this duty will be in charge of the Mapper module that randomly changes the island used in the computation of the key value, with the consequence that the involved chromosomes will be assigned to a different Reducer.

As final level of parallelization, a fine-grained model can be achieved with a slight variation in the last proposal. In this model, each chromosome is assigned to a single Mapper and then to a randomly selected Reducer. This can be obtained by acting on the way the Mapper module produces key values, which now are pseudo-randomly generated. It is worth noting that in the MapReduce model on the Cloud we have no knowledge on
the actual node containing a chromosome, so the pseudo-random function is a way to implement the neighboring required by the fine-grained model.

PROOF OF CONCEPT

To provide a proof of concept, in the present section we describe how we parallelized an existing GA for test data generation (Michael, 2001) exploiting GAE MapReduce and the global parallelization model detailed in the previous section. To this aim, we carried out a preliminary analysis on the effectiveness of applying the proposed approach on two different target functions, by executing the parallel algorithm on a local server machine and on three different configurations of the Google’s Cloud. The first function implements a simple numeric program already applied in other related works, while the second one provides a computational-intensive example, useful for a qualitative evaluation of the approach in terms of computational time and usage of resources.

In the following, we first recall the main aspects of the employed algorithm, then describe considered configurations and subjects, and finally discuss the obtained results.

The Employed Genetic Algorithm for Test Data Generation

The GA proposed by Michael et al. (Michael, 2001) is based on the Condition-Decision coverage so that for each condition in the code two requirements (true and false) must be taken at least by one input. Moreover, it follows the dynamic test data generation paradigm (Miller, 1976) using the
data collected during the execution as feedback to determine which input tests satisfy the requirements (Michael, 2001). To this end, the algorithm exploits a coverage table to track the requirements to be reached and those which have been satisfied. More details about the Michael’s algorithm can be found in (Michael, 2001). In the following we describe the main aspects of the parallel version we designed.

Starting from the first test requirement, an initial random population of n=100 chromosomes (each representing a test input) is generated. Then, the evolutionary process is started by invoking the InputReader, which splits the population in several crunches of data to be distributed among Mappers by the Master node. Each Mapper executes the instrumented code of the program under test to calculate the fitness value of the assigned chromosomes exploiting a fitness function based on the Korel function (Korel, 1990). Table 3 shows how the fitness is calculated for some typical conditions. A small fitness value indicates that the test input is close to satisfying the target requirement. If the program’s execution fails to reach the target, then the fitness function takes its worst possible value (i.e., MAXVALUE).

Once all Mappers terminate their execution, the Reducer starts the reproduction phase by applying on the received population traditional single point crossover and mutation operators. The crossover operator generates a new offspring from two parent chromosomes, selecting a random point of cut in the parents and swapping the corresponding test genes (i.e., a test input) with a rate of 0.5. As for the mutation operator, it replaces with probability 0.25 each gene in the chromosome with a new random test input. Let us note that if the test inputs are integer, these will be encoded in binary code by using the GrayCode before applying the genetic operators, otherwise if the input is real the standard binary encoding will be employed. Previous experimental results showed that the use of GrayCode allowed for finding suitable inputs more quickly (Michael, 2001). Once genetic operators are applied, the next generation is formed selecting the first n/2 best individuals, while the other chromosomes will be taken randomly from the offspring and parents.

At the end of this phase, data related to the new generation is saved through the OutputWriter into the DataStore, where it is retrieved by the Master module to check if the algorithm should be terminated or restarted by invoking the InputReader on the new offspring. The algorithm is stopped if it has found one chromosome that satisfies the test requirement or if it has performed a maximum number of generations (i.e., 30). In both cases, the next requirement to be satisfied is recovered from the coverage table and a new random population of n=100 chromosomes is generated to restart the search for an input that covers this requirement. This process continues until all requirements have been examined.

### Table 3. Computation of the fitness function

<table>
<thead>
<tr>
<th>Condition</th>
<th>Fitness Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = y</td>
<td>[</td>
</tr>
</tbody>
</table>
| x \(!=\) y | \[MAXVALUE if x = y \]
|           | 0 otherwise                                                                  |
| x > y     | \[1/MAXVALUE if x = y \]                                                      |
|           | \[y - x if x < y \]                                                          |
|           | 0 otherwise                                                                  |
| x ≥ y     | \[x - y if x < y \]                                                          |
|           | 0 otherwise                                                                  |
| x < y     | \[1/MAXVALUE if x = y \]                                                      |
|           | \[y - x if x > y \]                                                          |
|           | 0 otherwise                                                                  |
| x ≤ y     | \[y - x if x > y \]                                                          |
|           | 0 otherwise                                                                  |
| flag      | \[MAXVALUE if flag is true \]                                                 |
|           | 0 otherwise                                                                  |
Subject

The assessment has been conducted considering two different functions, namely quad and compareSumAndProduct. Figure 5 reports the pseudocode of both programs.

Quad solves quadratic equations of the form \( ax^2 + bx + c \). It takes three double input variables corresponding to the three different coefficients, namely \( a \), \( b \), and \( c \), and derives the nature of the solutions for the corresponding equation depending on the relationship defined by the discriminant:

\[ D = b^2 - 4ac. \]

CompareSumAndProduct takes three integer input variables, namely \( n \), \( j \), and \( k \), and evaluates if the sum of the first \( n \) natural numbers is greater, equal, or less than the value corresponding to the product of other two parameters, namely \( j \times k \).

Experimental Setting

We executed the parallel GA both on a local server and in the Cloud. The configuration of the server is reported in Table 4. As for the Cloud, we employed three different configurations characterized by different resources, namely F1, F2, and F4, and reported in Table 5. Moreover, for each different configuration we employed 30 multiple shards, i.e., the total number of Mappers running concurrently.
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

**Evaluation Criteria**

The performance of the GA has been assessed in terms of execution time, measured using the system clock, and of condition coverage, given by the percentage of satisfied test requirements. It is worth noting that we executed ten runs for each configuration in order to cope with the inherent randomness of the search-based approaches, and reported the average results together with the standard deviation.

**Results**

Tables 6 and 7 contain the results obtained running the algorithm both on local server and in the Cloud with the functions quad and compareSumAndProduct, respectively.

As for the quad function, we can notice that the achieved coverage was almost the same for all the cases, but the local server was by far faster than all the considered configurations in the Cloud.

This is due to the fact that the activation of each instance in the Cloud requires some time, especially when using services such as the MemCache or DataStore that usually take many resources just to start-up, making the Cloud solution not convenient for applications requiring a limited amount of computational power. It is also worth noting that there is no significant difference in running times among the three different configurations provided by Google.

On the other hand, with the more computationally intensive compareSumAndProduct function, the benefits of the Cloud become more evident. Indeed, while using the server took approximately 81 minutes, in the Cloud the execution required only 36 minutes using F4. In this latter scenario, the differences among the three Cloud configurations become more relevant: F4 takes less than half the time of F1 to complete the execution. The conclusion we can draw is that the Cloud platform provided by GAE has an initial overhead.

---

### Table 4. Configuration of the employed local server

<table>
<thead>
<tr>
<th>Hardware</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5645 2.4 GHz (6-core)</td>
</tr>
<tr>
<td>RAM</td>
<td>24 GB</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>2 x HD SATA 300GB 10K rpm</td>
</tr>
<tr>
<td>Connectivity</td>
<td>2 x Gbit LAN on board</td>
</tr>
<tr>
<td>Software</td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows Server 2007</td>
</tr>
<tr>
<td>Java Virtual Machine</td>
<td>Java SE Runtime v.1.7.</td>
</tr>
<tr>
<td>Google App Engine</td>
<td>v. 1.6.1</td>
</tr>
</tbody>
</table>

### Table 5. Google app engine configurations (05/2012)

<table>
<thead>
<tr>
<th>Frontend Class</th>
<th>Memory Limit</th>
<th>CPU Limit</th>
<th>Cost per Hour per Instance (05/2012)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1 (default)</td>
<td>128MB</td>
<td>600MHz</td>
<td>$0.08</td>
</tr>
<tr>
<td>F2</td>
<td>256MB</td>
<td>1.2GHz</td>
<td>$0.16</td>
</tr>
<tr>
<td>F4</td>
<td>512MB</td>
<td>2.4GHz</td>
<td>$0.32</td>
</tr>
</tbody>
</table>

### Table 6. Results for quad

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Coverage</th>
<th>Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Server</td>
<td>75.00%</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>1’29”</td>
<td>0’04”</td>
</tr>
<tr>
<td>F1</td>
<td>78.33%</td>
<td>0.08</td>
</tr>
<tr>
<td>F2</td>
<td>71.67%</td>
<td>0.08</td>
</tr>
<tr>
<td>F4</td>
<td>73.34%</td>
<td>0.08</td>
</tr>
</tbody>
</table>

### Table 7. Results for compareSumAndProduct

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Coverage</th>
<th>Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Dev.</td>
</tr>
<tr>
<td>Server</td>
<td>83.34%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>81’25”</td>
<td>19’11”</td>
</tr>
<tr>
<td>F1</td>
<td>83.34%</td>
<td>0</td>
</tr>
<tr>
<td>F2</td>
<td>83.34%</td>
<td>0</td>
</tr>
<tr>
<td>F4</td>
<td>83.34%</td>
<td>0</td>
</tr>
</tbody>
</table>

---

"IGI GLOBAL PROOF"
to start the computation, but once everything is running, the benefits of the parallelization clearly outperform a local machine.

Consequently, Cloud solutions are particularly useful in presence of non-trivial and computational-intensive problems.

As for the required resources, in Table 8 we report values of the percentage usage of free available quotas, expressed in terms of DataStore Write and Read operations and of Task Queue API Calls. Since these results are only influenced by the different executions of the genetic algorithm, we report mean values and standard deviation for usage of quotas calculated on the thirty experimental trials performed (i.e., ten per each CPU configuration).

The quotas of Datastore usage immediately remark the computational-intensive nature of the compareSumAndProduct function. The amount of resources used by this function is by far higher than the one used by the quad function. Moreover, while Write and Read quotas are slightly the same for the quad function, compareSumAndProduct performs a lot of Read operations, thus requiring a lot of resources. This is due to the fact that GA performs a lot of iterations before reaching the final coverage.

Finally, the obtained results suggest the effectiveness of some optimizations we designed to limit the usage of DataStore resources. Indeed we exploited only the freely available ones, without requiring the activation of the billing system. Those optimizations are illustrated in the next section.

### Lesson Learned on the Problems to Migrate GAs to the Cloud

Even if the migration of applications to the Cloud provides an easy-to-use solution to leverage the benefits of a scalable and distributed environment, it imposes a set of different constraints that have to be considered during the design and the implementation of algorithms. In general, these constraints may be classified in two distinct categories, namely software restrictions and infrastructure restrictions, respectively referred to limitations on software libraries and components running into the Cloud and to the specific features of the provided computational platform.

However, in practice, these limitations are different for each Cloud provider, such as Amazon or Google, and strongly depend on the specific kind of service offered, e.g., Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS).

In this section we describe some limitations and issues we faced for the migration of the considered GA to GAE, the Google PaaS Cloud solution, using the Java SDK. Moreover we report the details of some additional improvements made to the source code in order to leverage the benefits of the parallelization provided by the MapReduce model and to optimize the corresponding resource usage.

The first important software-related restriction to consider concerns the set of classes in the Java standard library (the Java runtime environment, or JRE) that can be accessed by applications running on GAE. The full white list of supported classes may be found in (GoogleJavaWhitelist, 2012).

### Table 8. Resource usage and total number of Fitness Evaluation

<table>
<thead>
<tr>
<th>Function</th>
<th>DataStore Write Ops</th>
<th>DataStore Read Ops.</th>
<th>Task API Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>quad</td>
<td>19.67%</td>
<td>18.37%</td>
<td>3.97%</td>
</tr>
<tr>
<td>compareSumAndProduct</td>
<td>28.93%</td>
<td>62.90%</td>
<td>6.63%</td>
</tr>
</tbody>
</table>

Mean | Std. Dev. | Mean | Std. Dev. | Mean | Std. Dev. |
15.75 | 14.62 | 3.44 | 1.19
Most of these limitations are imposed for security reasons in order to limit potential vulnerabilities and malicious behaviors derived by manipulating the byte-code executed on the JVM running in the Google Cloud. As a consequence, this list does not include packages such as java.lang.reflection. This is an important restriction since this package is mostly used by many Java-based code instrumentation tools, such as JPF (JPF, 2012) or CodeCover (CodeCover, 2012), exploited by many test-input data generation systems to evaluate the coverage of the code. Thus it was not possible to use those tools to evaluate coverage and the proposed GA exploited an instrumentation strategy specifically suited for GAE. However such an implementation does not yet support an automatic code instrumentation technique, requiring that instrumentation probes are manually injected into the code. Other software restrictions concern limitations on spawning multiple threads and on writing to files. In fact, due to the distributed nature of the application, writing to local files is not supported and the storage of data is supported only through the distributed storage system provided by Google, i.e., the distributed datastore.

In addition there are some infrastructure-related features of the platform that are worth considering to properly adapt existing algorithms or to design new ones. Such limitations mainly concern the specific GAE protocols to control the executions and the data storage.

Process execution on GAE is controlled by Tasks: each process is associated to a single GAE task, and all the tasks are organized in different Task Queues. These queues are used to manage different load balancing configurations (e.g., setting the maximum number of tasks in each queue) and to control the level of parallelization among different tasks. For instance, in our study the execution of tasks relied on two different queues: a high rate queue, called genetic-task-queue, devoted to run parallel and multiple Map-tasks for the evaluation of the fitness function, and a lower rate one, the process-queue, performing one task at a time corresponding to a single step in the GA process (e.g., fitness evaluation, generation of new population, and so on) in order to control the correct execution of the overall workflow. However GAE imposes that the execution of each task cannot take more than 10 minutes, which is an important constraint to consider when designing the execution of tasks. Moreover, it is worth mentioning that GAE currently lacks of fault tolerance strategies for tasks: if an error occurs during the execution, it may potentially freeze the overall application, since it does not provide any execution recovery system or a mechanism to exclude faulty tasks from queues.

Another key aspect concerns issues related to the communication with the Google Datastore. These issues are very important for the design of an application running into the Cloud, since the distributed nature of the datastore introduces an intrinsic communication latency that may drastically worsen the performance if multiple and useless operations are executed. Reducing Datastore operations determines a faster execution of tasks and an optimized usage of resources.

To this aim, we designed several optimizations and realized a package of classes based on the Java JDO APIs (GoogleJDO, 2012), which constitutes a wrapper layer between our application and the underlying Google Storage APIs.

All of these optimizations for the Datastore are mainly devoted to reduce as much as possible the usage of the free limited quotas imposed by Google, especially write operations. We optimized the code of the MapReduce library to avoid the creation of Datastore indexes for different entities attributes, drastically reducing the consumption of the “Datastore Write operation quota” from the 34% to the 4%. Moreover, we also performed writing operations to a properly defined “batch writing process,” i.e., a single write operation to save multiple entities at a time. Thus we avoided single write operations, reducing time and quotas necessary to save information into the datastore.
RELATED WORK

Parallel Genetic Algorithms Based on MapReduce

In the literature, we can find two proposals to use the MapReduce model to parallelize GAs (Gin, 2008) and (Verma, 2009). However, such proposals did not take into account the specific context of software testing. In (Gin, 2008) an extension of MapReduce, named MRPGA, has been proposed and motivated by the observation that a parallel GA cannot follow the two phases of MapReduce due to its iterative nature. The extension is based on a further phase for the global selection performed at the end of any iteration of parallel GAs. Moreover, a coordinator client is introduced to coordinate the executions of the parallel GA iterations. The proposed architecture consists of one Master node and multiple Mapper and Reducer nodes (Gin, 2008). The difference with the classical MapReduce is the introduction of a second reduce phase finalized to select the global optimum individuals. In order to simplify fault handling, the Master replicates the optimum individuals that the proposed extension of MapReduce has selected for each round of evolutionary operations. In case a node (e.g., a computer) resulted to be not available during the execution, the proposed algorithm just restarts the execution from the last round. This fault tolerance mechanism does not need a complex distributed file system for reliability purpose. The proposed extension of MapReduce for parallel GAs has been implemented using C# language on .NET platform and the developer runtime system has been evaluated in several student laboratories at the University of Melbourne. All the architectures we proposed in this work to parallelize GAs by means of MapReduce differ from the one of (Gin, 2008), none of them employ the additional reduction phase which is not necessary as suggested also by Verma et al. (Verma, 2009). Moreover, in their proposal a single coordinator affecting the scalability of their approach makes a huge amount of work regarding mutation, crossover, and evaluation of the convergence criteria. To avoid this problem, in our first architecture we split these jobs among the master node (performing evaluation of the convergence criteria) and the reducer (performing mutation and crossover), while the other two architectures do not suffer of this problem as these operations are carried out in a parallel fashion.

(Verma, 2009) provided another approach for using MapReduce for parallel GAs and highlighted several concerns and limitations of the proposal described in (Gin, 2008). The main concern was about scalability of parallel GAs. The previous proposal does not scale above 32 nodes, due to the inherent serial organization imposed by the use of a single coordinator, which performs mutation, crossover, and evaluation of the convergence criteria. Furthermore, they highlighted that the extension proposed in (Gin, 2008) could easily be implemented in the traditional MapReduce model using a Combiner instead of the proposed local reduce, as shown in (Dean, 2008). Finally they pointed out that using the same value (i.e., 1) as keys produced by the mapper, the reducer and the local reducer, MRPGA does not employ any characteristics of MapReduce model (i.e., the grouping by keys and the shuffling). Differently, Verma et al. (Verma, 2009) exploited the traditional MapReduce model by implementing GA in Hadoop to solve the ONEMAX problem. They investigated scalability and convergence of the proposed GA and found that larger problems could be resolved by adding more computational resources without changing algorithm implementation. Our second architecture is similar to (Verma, 2009) since the authors 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 (Verma, 2009) a local selection can lead to a reduction in selection pressure affecting the time taken to converge. This together with the fact that in the domain we considered the com-
putation of the fitness function is the most time consuming task (Michael, 2001) motivated us to investigate other architectures proposed in this work (i.e., GAs following global parallelization and grid parallelization models).

Parallel Genetic Algorithms for Automated Testing

In Di Gironimo (2012), a parallel Genetic Algorithm conceived for JUnit test suite generation and based on the global parallelization model proposed in this work has been realized. A preliminary evaluation of the proposed algorithm was carried out aiming to evaluate the speed-up with respect to the sequential execution. The obtained results highlighted that using the parallel genetic algorithm allowed for saving over the 50% of time. Differently from this work, the algorithm was conceived for test suite generation (and not for test data generation), realized exploiting Hadoop MapReduce, and its performance was assessed only on standard cluster.

Another approach to address parallelization of SBST has been recently proposed in Yoo (2011), 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.

Migrating Software Testing to the Cloud

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. In Tilley (2010) authors took into account the characteristics of an application under test and the types of testing to decide if migrate software testing to the Cloud. A proposal for distributing the execution of test cases in the Cloud was presented in Oriol (2010) where the authors described a framework for the distributed execution of the York Extensible Testing Infrastructure (YETI), a language agnostic random testing tool (Oriol, 2010). 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 Oriol (2010) were promising, showing that exploiting the proposed framework on the Amazon Elastic Computing Cloud (Amazon, 2012) the performances of YETI improved reducing the testing time.

FUTURE RESEARCH DIRECTIONS

The migration of SBST approaches in the Cloud is still in its early phases, so several research directions can be prospected.

First of all, it is important to carry out an extensive empirical evaluation of the solutions we proposed. This is necessary to highlight on the field their strength and weakness, as well as to assess their actual scalability. Furthermore, it will be very interesting to investigate how and if other Search-Based techniques previously employed in SBST, such as the Tabu Search or the Hill Climbing, can be migrated to the Cloud. Once this migration will be implemented, it will be very interesting to compare these different meta-heuristics, both in terms of performances and effectiveness, for each specific testing problem and strategy. Moreover, in the solutions proposed in this chapter, we have only considered the problem of the test data generation. Other types of SBST
Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud

may present specific issues, thus probably requiring other types of architectures.

Another interesting direction might be to assess the effort versus cost benefit trade-offs in using PaaS rather than an IaaS, which is typically less expensive than PaaS.

Finally, as a long-term research goal, SBST approaches can be integrated in a Validation-as-a-Service Cloud platform able to support companies in the entire software testing process.

CONCLUSION

In this chapter, we have proposed some solutions to migrate Genetic Algorithms to the Cloud, aiming at obtaining an automatic generation of test data. This migration is motivated by the need to achieve a greater efficiency and scalability of this Search-Based technique, thus possibly reinforcing its effectiveness thanks to the available computational power of the Cloud.

To simplify such a migration, we have suggested the use of the MapReduce paradigm, relieving programmers by the most of low level issues in managing a distributed computation. This is also motivated by the fact that MapReduce is natively supported by several Cloud infrastructures, such as Google, Amazon, and Microsoft. We have suggested several models implementing three different levels of parallelization (i.e., global, coarse-grained, and fine grained), with the aim to automatically generate test data and discussed issues and concerns about them. Finally we have provided a proof of concept, implementing the proposed solution taking advantages of the Google App Engine framework. Preliminary results show that, unless for toy examples, the Cloud can heavily outperform the performances of a local server.

REFERENCES


Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud


ADDITIONAL READING


Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud


