A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation

Ciprian Paduraru
University of Bucharest and
Electronic Arts
Bucharest, Romania

Marius-Constantin Melemciuc
University of Bucharest
Bucharest, Romania

Alin Stefanescu
University of Bucharest
Bucharest, Romania

ABSTRACT

This paper presents a distributed implementation for a genetic algorithm, using Apache Spark, a fast and popular data processing framework. Our approach is rather general, but in this paper the parallelized genetic algorithm is used for test data generation for executable programs. The viability of the approach is demonstrated on two examples.

KEYWORDS

distributed implementation, Apache Spark, genetic algorithms, test data generation, test coverage

ACM Reference format:
Ciprian Paduraru, Marius-Constantin Melemciuc, and Alin Stefanescu. 2017.
A distributed implementation using Apache Spark of
a genetic algorithm applied to test data generation. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 7 pages.
DOI: http://dx.doi.org/10.1145/3067695.3084219

1 INTRODUCTION

The importance of cybersecurity has increased year over year recently, given the wide interconnectivity between various software systems. Both academia and industry are investing many resources to find ways to detect critical security bugs in software components. An efficient technique for that is fuzz testing [8], which is a type of testing where many random input data are generated and the program under test is executing them with the goal of exhibiting security bugs or abnormal behaviour.

Since random testing may need too much time to find interesting paths in the program, several methods have been proposed to better guide the test data generation towards uncovered area of the program using, e.g., genetic algorithms or search-based techniques [11] or symbolic execution [4].

In this paper, we will investigate the use of genetic algorithms for test data generation, making use of the recent advancements in the distributed computing area. In particular, we implemented a parallel version of a genetic algorithm in Apache Spark [20]. We then applied this to the area of test data generation for x86 binary/executable programs.

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3084219

The main contributions of the paper are the following:

- the parallel implementation of a genetic algorithm in Apache Spark. To the best of our knowledge this is the first investigation of this type. The majority of the existing approaches for parallelizing genetic algorithms use a master-slave architecture [12] and not the map-reduce paradigm underlying Apache Spark. We also provide some valuable lessons learned while optimizing our solution on Spark.
- our approach is rather generic, so it has potential to be adapted to other types of genetic algorithms.
- we propose a fitness function based on some "probabilities" that certain branch conditions occur in some order and use them to guide the tests towards areas not yet explored. This is different from existing "goal-oriented" approaches, which attempt to find test data for a given path in the control-flow graph.

The paper is structured as follows. Next section presents the problem we want to solve and a solution based on genetic algorithms. Section 3 provides details about the parallel implementation of our approach. Section 4 offers an evaluation on a couple of examples, while the last section is reserved to future work.

2 USE CASE: A GENETIC ALGORITHM FOR TEST DATA GENERATION

The main motivation for our research stems from some concrete problems in security testing. To provide some context, we are part of an international industrial research project on software metrics called MEASURE[1], within which we have a close collaboration with a big software company providing security solutions called Bitdefender. They want to improve their security testing techniques and tools using state-of-the-art research. As mentioned in the previous section, fuzz testing is a popular technique which can be improved in several ways [4, 11]. We chose to investigate the use of genetic algorithms with a slightly different type of a fitness function (see Subsection 2.3). Moreover, since we have experience with Apache Spark we wanted to check the suitability of a map-reduce architecture for parallelism.

We tried to keep the implementation as generic as possible, such that if a user would like to adapt it to its problem, s/he would only have to fill the custom functionality for initialisation, selection, fitness, and genetic operations, leaving the entire parallelisation duty to the framework. In our opinion, the map-reduce pattern used by Apache Spark fits very well as a generalized pattern for the evaluation of genetic algorithms on a distributed system.

[1] http://measure.softeam-rd.eu

2.1 The problem: test data generation for executable (x86) programs

Since the above mentioned software security company must analyze various types of files for which there is no source code available, a requirement of our use case is that the system under test is an x86 binary program. This complicates various aspects due to the lower level of assembly code. We have started to collaborate to develop an in-house framework for dynamic analysis of binary files, which involves several methods including fuzz testing and symbolic execution.

Our use case of genetic algorithms addressed in this paper is to automatically create testing data for checking binary programs. The program is then executed against these data and monitored for issues such as crashes, assertions, or memory leaks. Such an approach for testing software programs is fuzz testing, which generates the test data more or less randomly. The most used metric for the quality of the testing data set is to have as many executed instructions of the program as possible. We recently started investigating genetic algorithms in order to improve the quality of the automated test data generation by guiding the search towards paths that are less probable than others using a dynamic fitness evaluation.

Our implementation uses a tool, called Tracer that can run a program P against the input test data and produce a trace, i.e., an ordered list of branch instructions $B_0, \ldots, B_n$ that a program encountered while executing with the given input *test*: Tracer(P, *test*) = $B_0 B_1 \ldots B_n$. Because a program can make calls to other libraries or system executables, each branch is a pair of the module name and offset where the branch instruction occurred: B_i = (*module*, *offset*). Note that we divide our program in basic blocks, which are sequences of x86 instructions that contain exactly one branch instruction at its end. As a tracer, we used an internal tool developed by the Bitdefender company, but with minor modifications we may use also an open-source tracer such as Bintrace[2].

2.2 Genetic representation

At the core of our approach we keep a population of individuals, where each one represents an input test data for the evaluated program P. Using the classical genetic algorithms parlance, each individual is identified by its "chromosome" which is a sequence of "genes". In our case, a gene is represented in memory as 1 byte and therefore an individual is an ordered list of bytes which will be the input for P. For instance, if P needs an input composed of a character variable, an integer, and a string, a possible individual could be a list of bytes $[C_0, I_0, \ldots, I_3, S_0, \ldots, S_N]$, with variable lengths for N. The user can also specify ranges for variables (e.g., integers such as $I_0, \ldots, I_3$ can have values in range [1..1000] or N, the size of the string could be between [0..128]). The input data associated to an individual can even represent complex data such as a file uploaded by a user or online data that a software application has to process.

[2]https://bitbucket.org/mihaila/bintrace

2.3 Fitness function

The main objective of our test data is to execute instructions of the program being evaluated that are difficult to reach using common input. To guide the individuals toward inputs that lead to rare paths of a program, the fitness function associates higher scores to individuals that take uncommon paths when the program is tested against them.

Assume we have an initial set of test inputs *TestDataSet* that produces (using the Tracer module) a set of traces *Traces* of the program P. Then we can evaluate the probability of one branching instruction B_i to occur immediately after another given B_j as follows. Informally, this probability, denoted by $\text{Prob}(B_i, B_j)$ is the number of all occurrences of sequence (B_i, B_j) in traces in the set *Traces* divided by the number of all occurrences in *Traces* of sequences (B_i, B) for any branching B. Formally, we distinguish two cases:

(1) If there exists at least one occurrence of (B_i, B_j) in *Traces*, then

$$\text{Prob}(B_i, B_j) := \frac{\text{no\_of\_occurrences}(B_i, B_j, \textit{Traces})}{\text{no\_of\_occurrences}(B_i, B, \textit{Traces})}$$

where $\text{no\_of\_occurrences}(B_i, B_j, \textit{Traces})$ is the number of distinct occurrences of (B_i, B_j) in the set of traces: $\sum_{\textit{trace} \in \textit{Traces}} \text{card}(\{\, k \mid (T_k, T_{k+1}) = (B_i, B_j) \text{ in } \textit{trace} \,\})$, and $\text{no\_of\_occurrences}(B_i, B, \textit{Traces})$ is the number of distinct occurrences of (B_i, B) for any B in the set of traces: $\sum_{\textit{trace} \in \textit{Traces}} \text{card}(\{\, k \mid (T_k, T_{k+1}) \text{ in } \textit{trace} \text{ and } T_k = B_i \,\})$.

(2) If there is no occurrence of (B_i, B_j) in *Traces*, we want to avoid probabilities with value 0, by using a value smaller than the minimum of the probabilities for pairs that appear in *Traces*:

$$\text{Prob}(B_i, B_j) = \min(\{\text{Prob}(B_x, B_y) \mid (B_x, B_y) \text{ in } \textit{Traces}\}) * F$$

where F is a factor between $[0.1, 0.5]$ (we used $F = 0.2$ in our experiments). This trick helps us differentiate between the fitness of two different traces that both contain an edge with no occurrence in *Traces*.

Now we can use $\text{Prob}(B_i, B_j)$ to define the fitness of a trace $\textit{trace} := B_0 B_1 \ldots B_n$, $n > 0$ produced by an individual as follows:

$$\text{Fitness}(\textit{trace}) := 1 - \prod_{(B_i, B_{i+1}) \in \text{Distinct}(\textit{trace})} \text{Prob}(B_i, B_{i+1})$$

where $\text{Distinct}(B_0 B1 \ldots B_n) := \{(B_i, B_{i+1}) \mid 0 \le i < n\}$, i.e., the set of pairs from the trace with no repeated elements.

Note that the fitness function does not consider the same pair (B_i, B_{i+1}) twice in a trace. The reason is that we do not want that the fitness value is artificially increased by the program loops (which may generate several equal sequences in a trace).

2.4 Initialization, selection and genetic operations

The initialization module generates a configurable number N of initial individuals in a genetic population G. The initial generation of individuals (input data) is obtained using a uniform random distribution for each gene (considering also the hints given by the user for different ranges of input variables). The initial population

of individuals is then improved over a configurable number of generations using the usual genetic algorithm operators: selection, mutation and crossover.

Selection is based on the fitness function presented in the previous subsection. The selection method in our implementation is a mix between Elitism [16], Rank-based Roulette Wheel Selection [14] and random selection, each one with a given percentage. One of the input parameters of our test data generation is the number of individuals to preserve between population, which will be denoted by K. Then, EL will represent the percentage of individuals selected with Elitism, RW using the Rank-based Roulette Wheel Selection and RR using random selection such that $EL + RW + RR = 1$.

The idea behind elitism is to keep between consecutive generations a given percentage of individuals that have the highest fitness values (candidates known as *elite*). The advantage of this in our approach is that rare branches inside a program (i.e., with higher fitness than common branches) are difficult to find and we want to preserve them across generations. The Rank-based Roulette Wheel Selection can add some variety by selecting probabilistically the individuals according to their rank in fitness value, while the random selection just adds some more variety of individuals by performing selection ignoring their fitness.

After the selection phase, mutation and crossover operations are applied to the $NL = N - K$ individuals left out of selection. Because at each new generation we keep the same number of individuals N, we create NL new individuals, which are created using the ones unselected. The mutation operation implies selecting a set of individuals to be mutated, each with a chosen probability P_m, then changing in each individual/chromosome one or more genes (bytes) to a new random value. The crossover operation selects pairs of individuals, with probability P_c. For each pair of individuals (A, B), a random number k for the "cut-point" for crossover is generated. The initial individuals A and B are replaced by the individuals $[A_0 \dots A_{k-1} B_k \dots B_S]$ and $[B_0 \dots B_{k-1} A_k \dots A_S]$, where S is the size of the input length used (and $0 \leq k \leq S$).

The number of generations has an upper bound given as a parameter `maxNumberOfGenerations`, but, before this limit is reached, a *plateau effect* is likely to be observed, i.e., when the individuals over a couple of generations do not improve significantly their fitness functions. More precisely, for two consecutive generations the change is checked as follows:

$$\sum_{x \in N} \Big[\text{Fitness}\big(G_i[x]\big) - \text{Fitness}\big(G_{i-1}[x]\big) \Big]^2 < \varepsilon.$$

One way to overcome this effect is to increase the parameters P_m and P_c temporarily until more diversity is added to the population of individuals [3]. In our implementation, if the average fitness is not improved in the last NG generations, then P_m and P_c are gradually increased until they get to P_{mMax} and P_{cMax} after a specified number of generations. If the plateau still occurs, the algorithm is restarted because there is a low probability that it can find any better tests from this point.

Figure 1 shows the pseudocode of the entire genetic algorithm described in this subsection.

In our experiments, we used $P_m = 0.2$, $P_c = 0.2$, $P_{mMax} = P_m + 0.3$, $P_{cMax} = P_c + 0.3$, $\varepsilon = 0.0001$, $K = N \times 0.2$, $EL = 0.8$, $RW = 0.1$, $RR = 0.1$. Also, the maximum number of generations produced by the genetic algorithm from the initial population `maxNumberOfGenerations` was set to 50.

G_0 := random population of N individuals
for i from 0 to `maxNumberOfGenerations` do
  S_i := select K individuals from G_{i-1} with probab. EL, RW, RR
  $Others_i$:= generate $N - K$ individuals from $G_{i-1} \backslash S_i$
    using mutations and crossover products
  $G_i := S_i \cup Others_i$
  Check for plateau and increase P_m and P_c if needed
  If plateau still occurs after a number of generations then STOP

Figure 1: The genetic algorithm

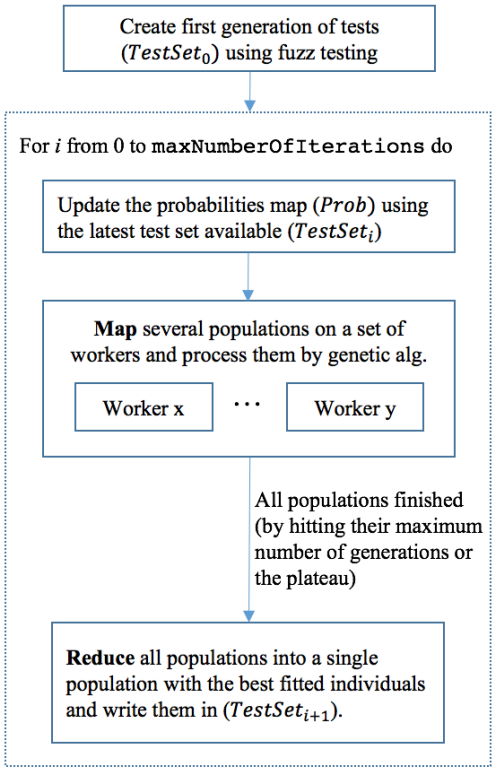

Figure 2: The workflow for computing the generations

2.5 The automated test data generation

The first generation of tests and traces mentioned in Subsection 2.2 is obtained using fuzz testing, i.e., random tests. The automated test data generation process runs continuously, and at each iteration, it produces a pair containing a set of test data and a set of traces obtained by executing these: $(TestSet_i, TraceSet_i)$. The probabilities map is dynamically updated, i.e., at each new iteration it is updated with the latest traces obtained in the set TraceSet_i. This is an important optimization, since paths that are uncommon at some point during the process may become common later with the new tests obtained. The overall process is depicted in Figure 2.

The user running the process has access to all the test data generated at any point during its execution: $TestSet := \cup_i \{TestSet_i\}$, which can be valuable for regression testing.

The maximum number of iterations, `maxNumberOfIterations`, was set to 100 in our simulations.

3 THE DISTRIBUTED IMPLEMENTATION USING APACHE SPARK

3.1 Apache Spark

Apache Spark[3] is an open-source framework for big data processing that allows parallelization of applications in cluster or cloud architecture. It is built around ease of use and it allows programmers to transparently use the advantage of data locality when performing computations, in a fault-tolerant manner. Apache Spark is currently one of the most popular and fastest distributed computing framework being also the largest open-source project in data processing[4].

The core data structure of Spark is the Resilient Distributed Datasets (RDDs) concept [20], which is shown to improve the distributed computation of iterative algorithms and interactive data mining tools by an order of magnitude over other map-reduce technologies. The RDDs are parallel data structures created initially from user data and code, and which can be transformed by applying different operations (e.g., map, filter, join etc.) on the same data multiple times. A program can be viewed then as a graph of transformations that operate on user's data.

The Spark scheduler can execute operations specified by RDDs in an efficient manner, exploiting data locality (i.e., avoiding data copies between nodes). Another optimization is that RDDs are lazy evaluated, which means that the work specified by the operations is performed only when the result is requested. This allows for higher parallelism and for low-level optimizations (e.g., Spark's built-in constraint solver may cut certain operations in the transformation graph). RDDs' structure also allows Spark to provide efficient fault-tolerance, since instead of doing backups of data between nodes, it just needs to know the history of RDDs transformations.

The traditional approaches of parallelizing the genetic algorithms use master-slave paradigms and libraries such as MPI or PVM [12]. One could think, at the first look, that they may achieve a slightly better performance than Spark because of less overhead in scheduling and lower level control. However, Spark presents several advantages when parallelizing genetic algorithms:

- It allows an easy setup for parallelization of genetic algorithms since the map-reduce pattern is very common in this approach, i.e., you create various tasks (see the list of `Worker` objects mentioned in Figure 3), you let them optimize internally their population, then reduce to get the best population from all the tasks. The code in Figure 3 depicts the entire code used for parallelization. Also, the overhead of Spark's scheduling is much smaller compared to the resources needed by the genetic algorithm execution for both map and reduce parts.
- It has automatic fault-tolerance and supports very well the heterogeneity of distributed systems, which is usually not the case with MPI, which must by adapted by hand for that.
- It can transparently traverse the barriers of local machine to a cluster, grid and cloud computing, without affecting user code. This would be much difficult to configure and use with MPI.

[3]https://spark.apache.org
[4]https://databricks.com/spark/about

```
workersRDD = sc.parallelize(Workers)
result = workersRDD.map(lambda w: w.init(N) ; w.solve())
              .reduce(lambda w1, w2: Worker.reduce(w1, w2))
```

Figure 3: Code snippet from the Spark implementation

3.2 Our distributed implementation

The solution described in Section 2 is easy to parallelize at multiple levels. In our implementation, we used Apache Spark and its map-reduce pattern [20] to take advantage of the computational power available in a (local) cluster and improve the quality of the test data over the serial version.

Our solution is to create an object `Worker` that has the following operations:

- `Init(`N`)`: generates a population of N random individuals.
- `Solve(`M`)`: iterates over the population of test data over (maximum) M generations, trying to improve their fitness using the genetic algorithm described above.
- `Merge(Worker other)`: merges the most promising individuals (with the highest fitness) from this worker and the worker sent as a parameter, generating the same number of N individuals.

A dynamic load balancing is needed to use the available computational power properly since the *Solve* operation in different workers can stop after a different number of generations computed internally by the genetic algorithm, due to the plateau effect (this cannot be always avoided by increasing the probabilities for mutation or crossover, as mentioned in Subsection 2.4). To deal with this aspect, and also to take into account the heterogeneity of the used systems, if the number of physical processes in our cluster is PY, then the number of `Worker` objects created is $NW = PY * R$, with R being a factor configured to minimize the overall idle time. In our experiments, a good value of R was 5, which minimised the overall idle time and at the same time kept the overhead of splitting the problem into many tasks that are small enough.

The list of workers is then split among physical cores using Spark's `parallelize` function, as seen in the Python code from Figure 3.

We provide a comparison between the time complexity for the serial vs the parallel implementation. Suppose that $Cost_{Solve}$ and $Cost_{Merge}$ denote the time complexities for *Solve* and *Merge* operations. Then, the time complexities of executing a single test data generation for the process described in Subsection 2.5 for a number of NW `Worker` objects, are:

- in serial case: $O(NW * Cost_{Solve})$
- in parallel case, using Spark over PY physical processes: $O(((NW * Cost_{Solve})/PY) + (\log NW) * Cost_{Merge})$. The first term is the cost needed to compute the solve task

operations and the second one is for merging the results into a single worker.

Then $Speedup = O(NW * Cost_{Solve})/(((NW * Cost_{Solve})/PY) + (NW/PY + \log NW) * Cost_{Merge})$, while the $Efficiency = Speedup/PY$. The $Cost_{Merge}$ in our implementation is $O(N)$, where N is the number of individuals in the population.

3.3 Lessons learned

We believe that our distributed implementation described above is general enough that can be reused for other types of applications of genetic algorithms. However, it is important to mention the various performance problems that we encountered and solved during the development from the initial version of the implementation to the current one. This may be of help to other researchers planning to use the same approaches.

One of the first optimizations done in the first phase of development was to use *fitness caching* as much as possible. For instance, in Figure 1, if the fitness of the individuals from the previous generation is already computed, then the selection phase can be ideally done in $O(N \log N)$ (needed by the sort method). Many of the best individuals from one generation will move through many consecutive generations so their fitness value does not need to be recomputed. In the backend, we use a "red" flag to signal whether an individual has been modified and consequently needs fitness re-computation. Thus the fitness function was only called when the algorithm needed an individual fitness (i.e., lazy evaluation) and its flag is red.

An exception from the lazy-evaluation rule was the optimization made in particular for the Spark's map-reduce pattern. The `Worker` objects created and evaluated in the map phase, needed to be reduced into a single `Worker` containing the best N merged population of individuals. The strategy used was to evaluate all red-flagged individuals and sort them according to their fitness value at the end of the `Worker.Solve` operation, which happens in the map phase. If we denote by F the average cost of computing a fitness value for a given test, the overall complexity for the additional step needed at the end of the `Solve` operation plus the `Merge` operation is: $T_{opt} = O(N*F+N \log N)*O(NW/PY)+O(N+N)*O(NW/PY+\log NW) = O(NW/PY * (N * F + N \log N + N)) + O(N * \log NW)$. Without this optimization, the fitness computations and sort would be needed at each pair of reduce operations, transforming the overall complexity in: $T_{base} = O(N * F + N \log N) * O(NW/PY + \log NW) = O(NW/PY * (N * F + N \log N)) + O(N * F + N \log N) * O(\log NW)$.

Then, $T_{diff} = T_{base} - T_{opt} = O(N * F + N \log N) * O(\log NW) - O(NW/PY * N)$. Knowing that NW is a factor for PY, and considering populations with large number of individuals (N), then the difference in time complexity represents an important optimization point.

The fitness evaluation of an individual in a genetic algorithm could be done in complex applications by an external process or application. In our use case, the fitness computation was done by sending the input to a different process that gives the trace of the execution, by calling a Tracer application. The initial way of getting the fitness from an individual was to instantiate a tracer process at each new fitness evaluation, give it the input on an input pipe, then the process executed the program under test with that input and outputted a text file with the trace. Finally, the resulted trace file was read by the main application and the fitness was computed using the Prob operation defined in Subsection 2.3.

This system had two severe performance drawbacks:

- The loading time of the fitness evaluation process (i.e., the execution of the tracer, in our case) can take significant time due to operating system dependencies and initialization times.
- Reading and writing data to disk can take significant time, especially in the case of distributed computing when the same machine can have several physical processes that can execute parallel tasks but usually share the same disk (i.e., writing to disk in parallel is much more expensive than writing to disk in serial because operating system is not able to optimize the sequence of read/writes operations). This is the case in Spark implementation too, since the same machine can have multiple Spark executors requesting fitness evaluations at the same time.

The solution we found for these problems was to impose that each `Worker` object created its own fitness evaluation process at initialization time and then used it for its entire lifetime. Moreover, instead of working with files read/write, we implemented a fully piped inter-process communication protocol where the fitness process was waked by a signal of a new task, read the task input (i.e., the test data to be evaluated in our application), processed it and sent out the binarized trace result on the pipe back to its owner `Worker` object.

This is depicted in Figure 4. Note that the fitness process is sleeping when he has nothing to do to avoid performance issues. Furthermore, at any time during execution, only one of the processes `Worker` or `Fitness` will run at the same time inside a Spark executor.

Another incremental improvement, specifically more for our project rather than from a general genetic algorithms framework (but with applicability in other domains too), was that instead of working with strings representing the modules names and paths, we mapped strings to integers to optimize searches and evaluations. The main motivation is that hashing with integers is much faster compared to hashing with strings. For instance, the probability computation between two pairs of (module string, offset) - Prob(B_i, B_j) - was actually done using a mapped integer for the module string. Since this operation was significantly used during evaluations, mapping once the string to integer then searching (hashing) for the integer value saved a lot of CPU cycles. The map data structure was updated at the beginning of each generation (since new strings associated to modules could appear after each new generation of tests).

4 EVALUATION

Our work can be evaluated from two points of view:

research question 1 is our genetic algorithm approach indeed as parallelizable as announced in Section 3; and also

research question 2 does the genetic algorithm cover more branching conditions of the evaluated program than fuzz testing.

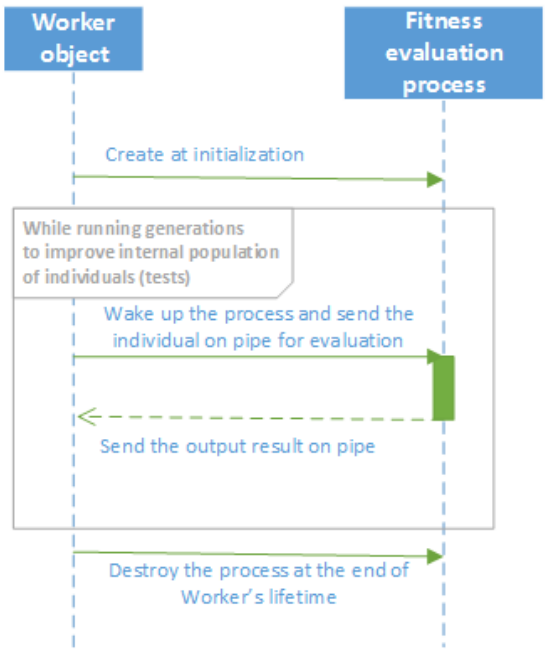

Figure 4: Sequence diagram for the communication between a `Worker` instance and the fitness evaluation process

For both research questions, we used as programs to be tested two open-source libraries (which were compiled to x86 binaries): `http-parser`[5] (library for parsing HTTP requests/responses) and `libxml`[6] (library for parsing and creating XML files).

For the first research question the usual metric in evaluating parallel programs is the throughput of the serial versus parallel implementation. The throughput in our application is the number of new tests obtained and evaluated per unit of time. The hardware platform used for testing was a cluster of 8 PCs, each one with 12 physical cores, totaling 96 physical cores of approximately same performance - the type of processor used was Intel Core i7-5930K 3.50 Ghz. A total of $NumWorkers = 480$ workers were instantiated (a factor of the 96 workers, to create enough tasks per physical workers as suggested in Subsection 3, avoiding this way the idle times for physical workers that hit the plateau before finishing the maximum number of allocated generations). The population size for each `Worker` object was $PopulationSize = 100$ individuals each with a length of 80 genes (bytes). `Worker` objects were let to optimize the internal population within a maximum of $maxNumberOfGenerations = 50$ generations, checking for plateau at each 5 generations. Technically speaking, the maximum number of tests expected for evaluation in this configuration is $MaxTests = PopulationSize * maxNumberOfGenerations * NumWorkers = 2,400,000$.

We let the algorithm execute in serial (a single process on one of the PCs) and in parallel on the configuration mentioned above, and stopped after 9 hours. The number of tests evaluated is shown in Table 1. The list of `Worker` objects was scheduled (on the 96 physical cores available) by Spark using its `parallelize` function, as shown in Figure 1. This setup with many instances of `Workers` did not affect the serial performance at all since the list of instanced `Workers` was executed one by one in this approach. Also, the parallel speedup and efficiency is expected to remain constant in all types of applications, despite the different costs of computing the traces. This is indeed the case for both examples as seen in Table 2. This is because, as mentioned in Section 3, in our framework each `Worker` has its own private tracer process spawned and this means that there is no contention between different physical `Workers`.

The memory footprint of our application is not a concern, and should not generally be, unless the fitness computation process would require significant memory (note that, if the fitness computation process would take N bytes, then creating PY processes on the same machine would require a minimum of $N * PY$ bytes of memory available).

For research question 2, we want to compare our genetic algorithms for automated testing data that finds rare paths inside programs' execution. In this case, we would like to see how much our approach helped us compared to fuzz testing (i.e. generating random tests). We executed the algorithm for 18 times, and computed different statistical metrics related to the number of different branch instructions encountered when testing `http-parser` and `libxml`, respectively, in an interval time of 1 hour. The results are provided in Tables 3 and 4.

Our genetic algorithms performs better in both cases, although by a small margin. Even if the difference in the number of instructions found is not big (4% and 6.5%, respectively, more branching executed, based on the mean metric), but one should take into account that usually rare paths and instructions are more difficult to find as the number of paths increases.

Table 1: Serial vs parallel throughput comparison

Library evaluated	Serial	Parallel
http-parser	1,593	130,843
libxml	446	34,891

Table 2: Speedup (i.e., parallel over serial throughput) and efficiency (i.e., speedup over number of physical processes used) metrics

Library evaluated	Speedup	Efficiency
http-parser	82.13	0.85
libxml	78.23	0.81

Table 3: Statistical results for the http-parser library

Metric	Fuzz testing	Genetic testing
mean	215.27	229.72
median	217	230
variance	75.03	1.03
mean absolute deviation	9.63	1.07
min	201	228
max	228	231

[5]https://github.com/nodejs/http-parser
[6]http://xmlsoft.org

Table 4: Statistical results for the libxml library

Metric	Fuzz testing	Genetic testing
mean	1219.94	1270.11
median	1225	1270
variance	118.64	0.81
mean absolute deviation	14.16	1.31
min	1202	1269
max	1231	1272

Although the results of our experiments look positive for both research questions (especially the first one), there are several threats to validity. First, we only run the experiments on two programs. This was due to the fact that our internal tracer module still misses some functionality and cannot be applied to any x86 program, but in the next period this will be fixed such that we can run the experiments on the cybersecurity grand challenge benchmark [6]. Moreover, we have not yet applied a full battery of statistical tests (including t-test and U-tests) as suggested in [2].

5 FUTURE WORK

We presented a parallel implementation of a genetic algorithm in Apache Spark using a custom fitness function. We also evaluated our approach on two open source libraries with promising results, especially for the parallelization capabilities. Although not groundbreaking, our approach has a couple of novel aspects mentioned already in the introduction.

We have several plans for future work:

- First, we would like to improve our approach through the best practices from the search-based testing literature [9] and experiment with other fitness functions.
- We would like to combine our technique with constraint solving [10] and dynamic symbolic execution [7]. As mentioned in Subsection 2.1, we are collaborating with the security company Bitdefender and the underlying analysis framework has already modules for dynamic symbolic execution and constraint solving, dedicated to x86 programs.
- We will check if we can extend the current performance through a combination between Spark and GPU for genetic algorithms: On the one hand, in industry there are already implementations combining Spark and GPU[7], while, on the other hand, first results of genetic algorithms' parallelization on GPUs were produced [5]. We will also seek inspiration from other approaches which parallelize genetic algorithms, such as [13, 15, 17].
- Last but not least, we may check which of the most recent approaches that use genetic methods for test generation [1, 18, 19] could be improved through parallelization.

We open-sourced our implementation at: https://github.com/paduraru2009/genetic-algorithm-with-Spark-for-test-generation

[7] https://www.oreilly.com/learning/accelerating-spark-workloads-using-gpus

ACKNOWLEDGMENTS

We would like to thank our colleagues Teodor Stoenescu and Alexandra Sandulescu from Bitdefender for fruitful discussions and collaboration. This work was partially supported by ITEA3 MEASURE project, funded through the Romanian National Authority for Scientific Research and Innovation (UEFISCDI), grant no. PN-III-P3-3.5-EUK-2016-0020.

REFERENCES

[1] Aldeida Aleti and Lars Grunske. 2015. Test data generation with a Kalman filter-based adaptive genetic algorithm. *Journal of Systems and Software* 103 (2015), 343–352.
[2] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 1–10.
[3] Donald S Burke, Kenneth A De Jong, John J Grefenstette, Connie Loggia Ramsey, and Annie S Wu. 1998. Putting more genetics into genetic algorithms. *Evolutionary Computation* 6, 4 (1998), 387–410.
[4] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[5] Alberto Cano and Sebastián Ventura. 2014. GPU-parallel subtree interpreter for genetic programming. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'14)*. ACM, 887–894.
[6] U.S. DARPA. 2016. Cyber Grand Challenge. http://cgc.darpa.mil. (2016).
[7] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE 24th Int. Symposium on Software Reliability Engineering, ISSRE'13*. IEEE, 360–369.
[8] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *Proc. of the 2nd International Workshop on Random testing: co-located with ASE'07)*. ACM, 1–1.
[9] M. Harman, Y. Jia, and Y. Zhang. 2015. Achievements, Open Problems and Challenges for Search Based Software Testing. In *Proc. of Int. Conference on Software Testing, Verification and Validation (ICST'15)*. IEEE, 1–12.
[10] Jan Malburg and Gordon Fraser. 2011. Combining search-based and constraint-based testing. In *Proc. of Automated Software Engineering (ASE'11)*. IEEE, 436–439.
[11] Phil McMinn. 2011. Search-Based Software Testing: Past, Present and Future. In *Proc. of the 2011 IEEE Fourth Int. Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, 153–163.
[12] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. 1999. Test-Data Generation Using Genetic Algorithms. *Softw. Test., Verif. Reliab.* 9, 4 (1999), 263–282.
[13] Carolina Salto, Francisco Luna, and Enrique Alba. 2014. Enhancing distributed EAs by a proactive strategy. *Cluster Computing* 17, 2 (2014), 219–229.
[14] Mandavilli Srinivas and Lalit M Patnaik. 1994. Genetic algorithms: A survey. *Computer* 27, 6 (1994), 17–26.
[15] Dirk Sudholt. 2015. *Parallel Evolutionary Algorithms*. Springer Berlin Heidelberg, 929–959.
[16] Dirk Thierens. 1997. Selection Schemes, Elitist Recombination, and Selection Intensity. In *Proc. of the 7th Int. Conference on Genetic Algorithms (ICGA'97)*. Morgan Kaufmann, 152–159.
[17] Leonardo Vanneschi, Daniele Codecasa, and Giancarlo Mauri. 2010. An empirical comparison of parallel and distributed particle swarm optimization methods. In *Proc. of Genetic and Evolutionary Computation Conference (GECCO'10)*. ACM, 15–22.
[18] Shunkun Yang, Tianlong Man, Jiaqi Xu, Fuping Zeng, and Ke Li. 2016. RGA: A lightweight and effective regeneration genetic algorithm for coverage-oriented software test data generation. *Inform. & Software Technology* 76 (2016), 19–30.
[19] Xiangjuan Yao and Dun-Wei Gong. 2014. Genetic Algorithm-Based Test Data Generation for Multiple Paths via Individual Sharing. *Comp. Int. and Neurosc.* 2014 (2014), 12 pp.
[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*. USENIX, 15–28.