Pairwise Test Generation Based on Parallel Genetic Algorithm with Spark

R.Z. Qi
College of Computer and Information
Hohai University
Nanjing, Chin

Z.J. Wang
College of Computer and Information
Hohai University
Nanjing, China

S.Y. Li
College of Science
Hohai University
Nanjing, China

***Abstract*--Pairwise testing is an effective combinatorial test generation technique that can generate tests covering all pairs of parameter values. Genetic algorithm has been used for pairwise test generation by researchers. It can often produce smaller test suite, but typically require a longer computation. To solve this problem, in this paper we use spark, an in-memory and iterative computing framework, to parallelize genetic algorithm for pairwise test generation. We propose fitness evaluation parallelization, which evaluates each individual's fitness value on spark's workers. A preliminary evaluation of the proposal algorithm is conducted to verify the effectiveness compared with those of other algorithms published in the literature. Experiments show that the proposed algorithm can generate better results among these algorithms.**

Keywords-combinatorial testing; pairwise testing; parallel genetic algorithmp; spark; test generation

I. INTRODUCTION

Pairwise testing has been proven to be a very effective combinatorial testing strategy that is based on the observation that most faults are caused by interactions of at most two factors. How to generate the minimum test suite, which can cover combinations of all pairs, is an important research area of pairwise testing. As generating a minimum test suite for pairwise testing is an NP-complete problem [1], researchers have tried various methods to generate near-minimum test suite. In surveys [2] [3], there have been four main groups of methods: greedy algorithm, heuristic search algorithm, mathematic method, and random method. Among these methods, greedy algorithm is the most widely used method for combinatorial test generation. Heuristic search algorithm formulates combinatorial test generation problem as a search problem, and applies search techniques such as genetic algorithm (GA) to solve it. These algorithms can often produce a smaller test suite than that from the greedy algorithm, but typically require a longer computation [2].

To deal with the heavy computational effort challenge, we propose a parallel genetic algorithm based on Spark [4] (PGAS), and use PGAS to generate pairwise test suite. GA is metaheuristic that simulates the natural process of evolution to solve optimization problems [5]. GA is naturally parallelizable, since fitness evaluations for individuals and most variation operators can easily be performed in parallel. Spark, which is an in-memory and iterative computing framework, is suitable for handling the parallelization of GA for test generation.

The remainder of the paper is organized as follows. In Section 2, we introduce related work. Then, Section 3 describes our GA for pairwise test generation. Section 4 presents our approach to parallelize GA based on spark to generate pairwise test suite. Section 5 reports the evaluation of our approach. The conclusions and future work are drawn in Section 6.

II. RELATED WORK

GA has been used for combinatorial test generation by researchers. The paper [6] proposed a GA-based technique to generate pairwise test configurations. It also gave some experiments. In [7] the author gave a genetic algorithm for pairwise test case generation called GAPTS. GAPTS encodes chromosome, which represents a test set, with an array of integer values. The fitness function is the total number of distinct pairs captured by the individual. GAPTS can produce pairwise test sets with smaller size compared with other methods. But it requires significantly longer processing time. The paper [8] also used GA to generate pairwise test set, and described an open source tool called PWiseGen. [9] designed six variants from GA, Particle Swarm Optimization, and Ant Colony Algorithm by reversing and randomizing their mechanisms to generate 2-way covering array. It also gave some experiments, and these methods required a longer computation.

Parallel Genetic Algorithm (PGA) is new technologies for improving the performance of metaheuristic search techniques. Both [10] and [11] use MapReduce model to parallelize genetic algorithm, but their approaches don't take into account the field of automatic test data generation. The article [12] proposed a PGA based on Hadoop MapReduce for JUnit test suite generation. The global parallelization model has been exploited, and a preliminary evaluation of the algorithm has been carried out to assess the speed-up. In [13] the authors used MapReduce model to support the parallelization of GA

© 2015. The authors - Published by Atlantis Press

for test data generation and their migration to the cloud. Three levels of parallelization models were suggested and the global parallelization model using Google App Engine framework were implemented. The paper [14] used PGA to generate prioritized pairwise test suite for software product lines. The algorithm could obtain smaller covering arrays with an acceptable performance difference with the greedy algorithm. All the above work parallelized GA with MapReduce. Since GA is an iterative and CPU-intensive algorithm, it can benefit from Spark's in-memory and iterative computing ability. However, to the best of our knowledge, in the literature there are no methods that have been proposed for generating pairwise test suite using Spark to parallelize GA. In this paper we will try to parallelize genetic algorithm based on Spark to generate pairwise test suite.

III. GA FOR PAIRWISE TESTING

When generating test suite with pairwise testing, input space of the software under test (SUT) can be modeled as a collection of parameters where each parameter assumes one or more values. Pairwise testing aims at selecting a subset from the complete set of parameter values combinations such that all pairs of parameter values are in the selected subset.

GA is a metaheuristic search technique that simulates the evolution of natural systems. When using GA to solve pairwise test generation problem, the following design decisions have to be made: chromosome encoding, fitness function, and genetic operations.

A. *Chromosome Encoding*

Chromosome encoding is the representation of an individual which is the candidate solution of the problem. In the scenario of pairwise test generation, the solution is often a suitable test suite (a set of test cases) of the SUT. In the literature, there are several encoding methods such as bit strings, floating point, and integer. We will use integer encoding to represent the individual, according to [8]. This encoding method encodes a set of test cases as an array of integer values. Each integer corresponds to a possible value of a parameter of the SUT. Thus an individual is an array of lists of integers, and each list represents a test case. The length of each list is equal to the number of the parameters. The size of an individual is the number of the test cases.

B. *Fitness Function*

A fitness function for pairwise testing is often a given coverage criteria, to measure the goodness of an individual. The article [15] defines that 100% pairwise coverage requires that every possible pair of interesting values of any two parameters are included in some test case. We will use this 100% pairwise coverage as our fitness function. So, the fitness function is the total number of different pairs covered by all the test cases in an individual. If an individual covers more different pairs than others, it is better than others. An individual becomes a solution when it covers all pairs.

C. *Genetic Operations*

Another important issue of GA is genetic operations such as selection, crossover, and mutation. As for the selection operator, we employ fitness proportionate selection to determine which individuals to choose as parents for reproduction. As for the crossover operator, we use single-point crossover with a probability of 0.8 to produce offspring. We use integer randomization mutation to change each test case with a given probability of 0.2, and replace it with a new random test case.

IV. PARALLEL GENETIC ALGORITHM

A. *Parallelization*

According to [16], there are four parallel models: global model, distributed model, cellular model, and hybrid model. Spark is based on the master-slave distributed computing model; it is suitable for the global model of GA parallelization. So, PGAS is designed based on global model for fitness evaluations.

Our basic idea is to parallelize initial population into spark Resilient Distributed Dataset (RDD) and evaluate each individual's fitness value on the workers. Then the driver collects the results and applies genetic operations. The lineage graph of parallel fitness evaluation is shown in Figure1. First, initial population is parallelized into spark RDD by a parallelize() method. Then a map() is applied to transform each individuals of the population into the <individual, fitness> key-value pairs. Finally, the collectAsMap() action starts to collect these pairs as HashMap.

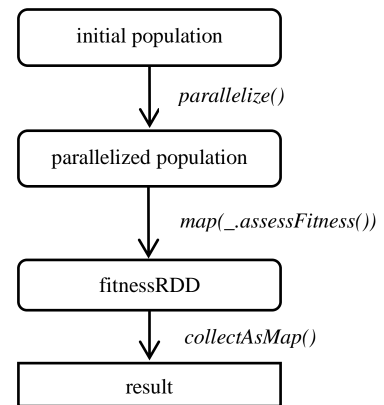

FIGURE I. RDD LINEAGE GRAPH FOR PARALLEL FITNESS EVALUATION

B. *Algorithm*

Algorithm 1 sketches the pseudocode of PGAS. At the beginning, AP, the number of all pairs of parameter values to be covered, is generated (Line1). The initial popsize individuals, each consisting of m test cases, which are created randomly by picking each slot uniformly among all possible values, are generated to form the initial population (Line2). The method parallelize() parallelizes the initial population (Line3). In the external while loop (Line4-22) the algorithm assigns one to the generation iterator, then enters the inner while loop (Line 5-6). In each iteration of the inner while loop (Line6-19) the algorithm first parallelizes fitness evaluations (Line7-10) as illustrated in Figure1. Then the <individual, fitness> key-value pairs collected by the driver are sorted by the fitness value (Line 11). If the first pair's value is AP, its key is the best individual, and will be returned (Line 12).

Otherwise, the algorithm enters a for loop which applies the genetic operators including selection, crossover, and mutation (Line13-17). After leaving the for loop, the algorithm enters the inner while loop to start the evolution again. The evolution process is continued until it reaches the maximum number of generation. If the algorithm can't find the best individual which covers all pairs, m will be incremented by one (Line 20) and one test case will be added to each individual of the population randomly (Line 21), then the external while loop runs again.

Algorithm 1 Parallel Genetic Algorithm

Input : pv.txt: parameter-values text file
k: number of parameters in SUT
n_i: number of values for each parameter
m: test suite size
popsize: desired population size
Output: the best individual

```
1: AP ← getNumOfAllPairs(k, n_i)
2: P ← initializePop(m, popsize, "pv.txt")
3: parallelize(P)
4: While (true)
5:     it ← 1
6:     While (it ≤ max)
7:         For each individual p_i ∈ P do
8:             fitnessRDD ← p_i.map(_.assessFitness())
9:         End for
10:        result ← fitnessRDD.collectAsMap()
11:        sortByValue(result)
12:        If result.top = AP then return top
13:        For popsize/2 times do
14:            parents ← selection(P)
15:            children ← crossover(P)
16:            P ← mutate(children)
17:        End for
18:        it ← it + 1
19:    End While
20:    m ← m + 1
21:    addTestCase(P)
22: End While
```

V. PRELIMINARY EVALUATION

We have implemented PGAS on spark using Java. In this section we performed a series of experiments to assess the effectiveness of PGAS. According to [1], effectiveness is measured by the number of test cases generated by the algorithm.

The experiments of PGAS were performed on a small cluster consisting of 5 nodes, where each node has one Intel Core i5 750 Quad-Core at 2.66GHz, 2GB RAM. One node is Namenode, and the other four nodes are Datanodes, which total have 16 cores. Each node is running at the Ubuntu 12.04, Java 1.7, Hadoop 2.4, and Spark 1.1.0.

The experiments were conducted using an input of k parameters, each with p distinct values. The parameter sizes can be represented as pk. As often used in literature, we exploited five benchmark problems of different size: 34, 313,R 415317229, 41339235, and 2100. The parameters of PGAS were described as follow. The population was composed of 500 individuals. PGAS used fitness proportionate selection, single-point crossover operator (with probability 0.8) and integer randomization mutation (with probability 0.2). The generation number on each core of the workers was 104, so the total generation number of PGAS is 16*104=1.6*105. Because of the stochastic nature of GA, we performed 30 independent runs of each benchmark to gain sufficient experimental data.

We compared the effectiveness of GPAS with those of other existing approaches: AETG [17], IPO [1], CTS [18], GAPTS [7], and PWiseGen [8]. Among these approaches, AETG and IPO uses greedy algorithm. CTS is algorithm which uses covering arrays. GAPTS and PWiseGen are based on serial genetic algorithm. The results of conducted experiments are shown in Table 1. In 30 independent runs, PGAS can generate fewer test cases than other algorithms in benchmark S3 and S4. In S3, the 33 test cases were found in the generation # 2561(the best case). In S4, the 25 test cases were found in the generation # 335(the best case). The size of test suite generated by PGAS in benchmark S1, S2, S5 was equal to the best result in other algorithms.

TABLE I. COMPARISON WITH EXISTING APPROACHES

Parameter sizes	AETG	IPO	CTS	GAPTS	PWise Gen	PGAS
$S1=3^4$	9	9	9	9	9	9
$S2=3^{13}$	15	19	15	15	15	15
$S3=4^{15}3^{17}2^{29}$	41	36	39	35	34	33
$S4=4^{1}3^{39}2^{35}$	28	29	29	27	26	25
$S5=2^{100}$	10	15	10	10	10	10

VI. CONCLUSION AND FUTURE WORK

In this paper, we explore the use of spark to parallelize genetic algorithm for pairwise test generation. Based on the global model of GA parallelization, we propose fitness evaluation parallelization, which evaluates each individual's fitness value on the workers. A preliminary evaluation of our PGAS is conducted on a small cluster to verify the effectiveness compared with those of other algorithms published in the literature. Experiments show that the proposed algorithm can generate better results among these algorithms.

In future, first we will try to parallelize genetic operation by spark to get more optimal results in shorter execution time. Second, we plan to conduct experiments on Amazon EC2 to check the effectiveness and scalability of PGAS.

ACKNOWLEDGMENTS

This research was supported in part by Fundamental Research Funds for the Central Universities 2010B06914; Nature Science Fund of Jiangsu Province 2013517111.

REFERENCES

[1] Y. Lei and K.C. Tai. In-parameter-order: a test generation strategy for pairwise testing. in Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium. Wshington, DC. p. 254-261. 1998.

[2] C.H. Nie and L. Hareton, A survey of combinatorial testing. ACM Comput. Surv. 43(2). p. 1-29. 2011.

[3] S.L.Y. Khalsa, An orchestrated survey of available algorithms and tools for Combinatorial Testing. Carleton University, Department of Systems and Computer Engineering. 2014.

[4] M. Zaharia, N.M.M. Chowdhury and M. Franklin, et al., Spark: Cluster Computing with Working Sets., EECS Department, University of California, Berkeley. 2010.

[5] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. 1st ed., Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. 1989.

[6] S.A. Ghazi and M.A. Ahmed. Pair-wise test coverage using genetic algorithms. in The 2003 Congress on Evolutionary Computation. p. 1420-1424. 2003.

[7] J.D. McCaffrey. An Empirical Study of Pairwise Test Set Generation Using a Genetic Algorithm. in 2010 Seventh International Conference on Information Technology: New Generations (ITNG). Las Vegas, NV. p. 992-997. 2010.

[8] P. Flores and C. Yoonsik. PWiseGen: Generating test cases for pairwise testing using genetic algorithms. in Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on. Shanghai. p. 747-752. 2011.

[9] C.H. Nie, H.Y. Wu and Y.L. Liang, et al. Search Based Combinatorial Testing. in Software Engineering Conference (APSEC), 2012 19th Asia-Pacific. Hong Kong. p. 778-783. 2012.

[10] C. Jin, C. Vecchiola and R. Buyya. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. in Proceedings of the 2008 Fourth IEEE International Conference on eScience. IEEE Computer Society. p. 214-221. 2008.

[11] A. Verma, X. Llor and D.E. Goldberg, et al. Scaling Genetic Algorithms Using MapReduce. in Proceedings of the 2009 Ninth International Conference on Intelligent Systems Design and Applications. IEEE Computer Society. p. 13-18. 2009.

[12] L. Di Geronimo, F. Ferrucci and A. Murolo, et al. A Parallel Genetic Algorithm Based on Hadoop MapReduce for the Automatic Generation of JUnit Test Suites. in Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE Computer Society. p. 785-793. 2012.

[13] F. Ferrucci, S. Di Martino and V. Maggio, et al., Towards Migrating Genetic Algorithms for Test Data Generation to the Cloud, in Software Testing in the Cloud: Perspectives on an Emerging Discipline. IGI Global. p. 113-135. 2012.

[14] R.E. Lopez-Herrejon, J. Ferrer and F. Chicano, et al. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. in 16th Genetic and Evolutionary Computation Conference, GECCO 2014. Vancouver, BC, Canada: Association for Computing Machinery. p. 1255-1262. 2014.

[15] M. Grindal, J. Offutt and S.F. Andler, Combination testing strategies: a survey. p. 167--199. 2005.

[16] G. Luque and E. Alba, Parallel Genetic Algorithms Theory and Real World Applications, in Studies in Computational Intelligence, Springer Berlin Heidelberg. Volume 367. p.15-25. 2011.

[17] D.M. Cohen, S.R. Dalal and M.L. Fredman, et al., The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7). p. 437-444. 1997.

[18] A. Hartman and L. Raskin, Problems and algorithms for covering arrays. Discrete Mathematics, 284(1-3). p. 149 - 156. 2004.