Qi RZ, Wang ZJ, Li SY. A parallel genetic algorithm based on Spark for pairwise test suite generation. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(2): 417–427 Mar. 2016. DOI 10.1007/s11390-016-1635-5

A Parallel Genetic Algorithm Based on Spark for Pairwise Test Suite Generation

Rong-Zhi Qi [1], *Member, CCF*, Zhi-Jian Wang [1], *Member, IEEE*, and Shui-Yan Li [2]

[1] *College of Computer and Information, Hohai University, Nanjing 211106, China*
[2] *College of Science, Hohai University, Nanjing 211106, China*

E-mail: {rzqi, zhjwang, lsy}@hhu.edu.cn

Received March 18, 2015; revised September 25, 2015.

Abstract Pairwise testing is an effective test generation technique that requires all pairs of parameter values to be covered by at least one test case. It has been proven that generating minimum test suite is an NP-complete problem. Genetic algorithms have been used for pairwise test suite generation by researchers. However, it is always a time-consuming process, which leads to significant limitations and obstacles for practical use of genetic algorithms towards large-scale test problems. Parallelism will be an effective way to not only enhance the computation performance but also improve the quality of the solutions. In this paper, we use Spark, a fast and general parallel computing platform, to parallelize the genetic algorithm to tackle the problem. We propose a two-phase parallelization algorithm including fitness evaluation parallelization and genetic operation parallelization. Experimental results show that our algorithm outperforms the sequential genetic algorithm and competes with other approaches in both test suite size and computational performance. As a result, our algorithm is a promising improvement of the genetic algorithm for pairwise test suite generation.

Keywords pairwise testing, parallel genetic algorithm, Spark, test generation

1 Introduction

Software testing is the activity of executing a system to detect failures. In some testing scenarios, many failures are triggered by interactions among parameters of the software. Real software systems become more complex and tend to have many parameters. This will create a huge number of possible combinations of all parameter values. Exhaustive testing tries all the possible parameter value combinations to detect interaction-triggered failures. However, exhaustive testing is not feasible for real situations. For example, if a system has 10 parameters and each parameter can be assigned one of 10 different values, we will generate 10^{10} possible combinations. If each combination is executed in one minute, it will take about 20 thousand years to complete the whole test.

Pairwise testing can alleviate the combination explosion problem of exhaustive testing. Pairwise testing has been proven to be an effective testing strategy for various types of systems[1]. It is based on the observation that most faults are caused by interactions of at most two factors[2]. How to generate minimum test suite to cover combinations of all pairs is an active research point of pairwise testing. As generating minimum test suite for pairwise testing is an NP-complete problem[1], researchers have tried various methods to generate near-minimum test suite. Nie and Leung presented four main groups of approaches: greedy algorithms, heuristic search algorithms, mathematic methods, and random methods[3]. Among these methods, the greedy algorithms are the most widely used for combinatorial test generation. They are usually faster than the metaheuristic algorithms but do not always produce the smallest test suites[4]. The heuristic search algorithms formulate combinatorial test generation problem as a search problem and apply search techniques such as Hill Climbing (HC), Simulated Annealing (SA), Genetic Algorithm (GA), Ant Colony Algorithm (ACA)

Regular Paper

This work is supported by the Fundamental Research Funds for the Central Universities of China under Grant Nos. 2010B06914 and 2013B07514.

and so on to solve the problem. These algorithms can often produce a smaller test suite than the greedy algorithms, but typically require a longer computation[3].

To deal with the heavy computation challenge, in this paper, we propose a parallel genetic algorithm based on Spark[5], called PGAS, to speed up the process of generating pairwise test suite. Parallelism can improve both performance and quality of the solutions. GA is naturally parallelizable since its fitness evaluation and evolution process including iterations of genetic operation can be performed in parallel. GA is also an iterative and CPU-intensive algorithm. Thus, it can benefit from Spark's in-memory computing ability. Spark, a fast and general cluster computing platform, is suitable for handling the parallelization of GA. However, to the best of our knowledge, there are no methods that have been proposed in literatures for generating pairwise test suite using Spark to parallelize GA. Therefore, in this paper, we use Spark to implement two-phase parallelization: fitness evaluation parallelization and genetic operation parallelization. The first phase evaluates each individual's fitness value in parallel; the second phase splits the population into different slices that can be evolved separately.

The main contributions of this paper can be summarized as follows.

1) We propose a two-phase parallelization algorithm (fitness evaluation and genetic operation) based on Spark for pairwise test suite generation.

2) We give a preliminary evaluation of the proposed algorithm through experiments to verify the performance and effectiveness.

The rest of the paper is organized as follows. In Section 2, we introduce related work. Section 3 gives the description of pairwise testing. Section 4 describes our GA for pairwise test generation. Section 5 presents our two-phase parallelization algorithm. Section 6 reports the evaluation of our algorithm. Section 7 concludes the paper.

2 Related Work

Metaheuristic search algorithms have been used for combinatorial test generation by many researchers. Ghazi and Ahmed[6] proposed a GA-based technique to generate pairwise test configurations and conducted some experiments. McCaffrey designed a GA for pairwise test case generation called GAPTS[7]. GAPTS encodes chromosome to represent a test set with an array of integer values. The fitness function is the total number of distinct pairs captured by the chromosome. GAPTS can produce pairwise test sets with smaller size compared with other methods. But it requires significantly longer processing time. Shiba *et al.*[8] used GA and ACA to generate 3-way test set. Flores and Yoonsik also used GA to generate pairwise test set, and designed an open source tool called PWiseGen[9]. Six variants from GA, PSO, and ACA by reversing and randomizing their mechanisms to generate 2-way covering array were designed by Nie *et al.*[10] They also gave some experiments which revealed that these variants required a longer computation. Cohen *et al.*[11-12] proposed approaches to generate variable covering array and 3-way covering array using SA. Petke *et al.*[13] used an SA-based tool called CASA to generate higher strength covering array which is constrained and prioritized. Henard *et al.*[14] proposed a scalable and flexible search-based technique based on (1+1) Evolutionary Algorithm to generate configurations under budget and time constraints for large feature models. Recently, harmony search (HS) has been used to generate combinatorial test case. Alsewari and Zamli[15] exploited HS to generate a complete test suite that covers the t-way interactions at least once in a greedy manner. Experiments showed that this method can get good test sizes for most of the considered configurations. An approach for test suite generation based on HS was presented by Li *et al.*[16] This method addresses the key issues of higher degree and variable strength combinatorial coverage.

Parallel metaheuristic search algorithms are new technologies for improving the performance of metaheuristic search algorithms. MRPGA is an extended MapReduce[17] model to automatically parallelize GA[18]. Verma *et al.*[19] provided an approach to scale GA using MapReduce model. Parallel genetic algorithm (PGA) was designed and implemented on Hadoop. Both Jin *et al.*[18] and Verma *et al.*[19] used MapReduce model to parallelize GA, but their approaches do not take into account the field of automatic test data generation. Few researchers have applied PGA for test case generation, including combinatorial test case generation. Geronimo *et al.*[20] proposed a PGA based on Hadoop MapReduce for JUnit test suite generation. The global parallelization model was exploited, and a preliminary evaluation of the algorithm was carried out to assess the speedup. Martino *et al.*[21] used MapReduce model to support the parallelization of GA for test data generation and the migration to the cloud. Three levels of parallelization mod-

els were suggested and the global parallelization model using Google App Engine framework was implemented. All the above work parallelized GA with MapReduce. A parallel test generation strategy called MC-MIPOG[22] was used for t-way test data generation on multicore architecture. Lopez-Herrejon *et al.*[23] used PGA to generate prioritized pairwise test suite for software product lines. The algorithm follows the master-slave model to parallelize the evaluation of the individuals using normal cluster. With respect to Lopez-Herrejon *et al.*[23], our proposed algorithm uses Spark to implement two-phase parallelization: fitness evaluation and genetic operation.

3 Pairwise Testing

When generating test suite with pairwise testing, the input space of the software under test (SUT) can be modeled as a collection of parameters, each of which assumes one or more values. Pairwise testing aims at selecting a subset from the complete set of parameter value combinations such that all pairs of parameter values are in the selected subset. Each selected parameter value combination will generate at least one test case for SUT. The set of test cases is often called test suite which is represented by covering array (CA) defined as below.

Definition 1 (Covering Array). *Let SUT have k parameters and each parameter p_i have v_i $(1 \leqslant i \leqslant k)$ values. A covering array $\boldsymbol{CA}(N; v_1^{p_1} v_2^{p_2} \cdots v_k^{p_k}, t)$ is an $N \times k$ matrix. Each row of this matrix is a test case. N is the number of test cases and t is the strength of the covering array. Each $N \times t$ subarray contains at least one occurrence of each t-tuple corresponding to the t columns. If $v_1 = v_2 = \cdots = v_k = v$, the corresponding covering array is said to be uniform. It is denoted as $\boldsymbol{CA}(N; v^k, t)$.*

When $t = 2$, it is called 2-way covering array. Testing with 2-way covering array is called pairwise testing. The intent of pairwise testing is to reduce the number of test cases.

For example, we assume SUT has three parameters: p_0, p_1, and p_2. The possible values for each parameter are $\{a_0, a_1\}$, $\{b_0, b_1\}$, and $\{c_0, c_1\}$ respectively. The total number of possible combinations of all parameter values is 2^3:

$$(a_0, b_0, c_0), (a_0, b_0, c_1), (a_0, b_1, c_0), (a_0, b_1, c_1),$$
$$(a_1, b_0, c_0), (a_1, b_0, c_1), (a_1, b_1, c_0), (a_1, b_1, c_1).$$

When testing this SUT with pairwise testing, the generated test cases will cover each pair of parameter values. There are 12 such pairs:

$$\{a_0, b_0\}, \{a_0, b_1\}, \{a_0, c_0\}, \{a_0, c_1\},$$
$$\{a_1, b_0\}, \{a_1, b_1\}, \{a_1, c_0\}, \{a_1, c_1\},$$
$$\{b_0, c_0\}, \{b_0, c_1\}, \{b_1, c_0\}, \{b_1, c_1\}.$$

In this case, the following set of four combinations suffices:

$$(a_0, b_0, c_1), (a_0, b_1, c_0), (a_1, b_0, c_0), (a_1, b_1, c_1).$$

Table 1 shows the covering array $\boldsymbol{CA}\left(4; 2^3, 2\right)$ for this SUT.

Table 1. $\boldsymbol{CA}(4; 2^3, 2)$

	p_0	p_1	p_2
t_1	a_0	b_0	c_1
t_2	a_0	b_1	c_0
t_3	a_1	b_0	c_0
t_4	a_1	b_1	c_1

In the above example, it can be seen that pairwise testing can reduce the required number of test cases from 8 to 4, a 50% reduction. With larger parameter value combinations, the result will be better. In this paper, we try to generate near-minimum covering array (test suite) to reach 100% pairwise coverage for SUT, especially in the scenario of large parameter value combinations.

4 Design of GA

GA is a metaheuristic search technique that simulates the evolution of natural systems. It is often exploited to solve search and optimization problems. It is usually infeasible to exhaustively evaluate the entire input space and thus GA is used to produce good solutions in reasonable time by evaluating only a small portion of the input space. The basic GA first constructs an initial population randomly and then iterates through the following procedures until stopping criteria hold. It assesses the fitness value of all the individuals in the population. Individuals with high fitness values have a better chance to evolve into the next generation by applying genetic operators such as selection, crossover, and mutation.

When using GA to solve pairwise test generation problem, the following design decisions have to be made: chromosome encoding, fitness function, and genetic operators.

4.1 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 of 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[7]. 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 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, denoted by m, is the number of the test cases. Our goal is to find the optimal m to cover 100% pairwise combinations of parameter value. If m has been reported in literatures, we set this value as the initial test suite size in our proposed algorithm and search for a solution. Otherwise, we use the binary search approach presented by Cohen *et al.*[11] to determine m.

According to Definition 1, SUT has k parameters and each parameter p_i has v_i $(1 \leqslant i \leqslant k)$ values. When using the above encoding, p_1 has values of 1, 2, 3, $\cdots$, v_1, p_2 has values of v_1+1, v_1+2, v_1+3, $\cdots$, v_2, and p_k has values of $v_{k-1}+1$, $v_{k-1}+2$, $v_{k-1}+3$, $\cdots$, v_k. These parameters and values are expressed by text file as illustrated in Fig.1. This text file will be used as input for our algorithm.

p_1: 1, 2, 3, $\cdots$, v_1
p_2: v_1+1, v_1+2, v_1+3, $\cdots$, v_2
$\vdots$
p_k: $v_{k-1}+1$, $v_{k-1}+2$, $v_{k-1}+3$, $\cdots$, v_k

Fig.1. Input text file of SUT.

The total number of pairs to be covered is denoted by AP and is calculated as:

$$AP = v_1 \sum_{i=2}^{k} v_i + v_2 \sum_{i=3}^{k} v_i + \cdots + v_{k-1}v_k.$$

4.2 Fitness Function

A fitness function for pairwise testing is often a given coverage criterion which measures the goodness of an individual. Grindal *et al.*[24] defined that 100% pairwise coverage requires that every possible pair of interesting values of any two parameters is included in some test case. We will use this 100% pairwise coverage as our fitness function. Thus, 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.

4.3 Genetic Operators

Another important issue of GA is genetic operators including selection, crossover, and mutation. As for the selection operator, we employ fitness proportionate selection to determine which individuals to be chosen as parents for reproduction. In fitness proportionate selection, individuals are selected in proportion to their fitness values: if individuals have higher fitness values, they are selected more often. Let $s = \sum_i f_i$ be the sum of all individuals' fitness values. A random number n is picked from 0 to s. If n falls within the range of some individual in the array of individual ranges, this individual is selected.

After selection, the selected parents are copied and then crossover mixes and matches parts of these two copied parents to form better children. Our crossover mechanism uses both single-point and multiple-point random crossover. Single-point random crossover chooses a number c randomly from 0 to the length of an individual and exchanges all the indexes smaller than c. In multiple-point crossover, individuals are regarded as a ring. Several unique points are picked at random, breaking the ring into several segments. A segment from one ring is exchanged with one from another ring to produce new offspring. In our testing scenarios, if the size of an individual is small, we use single-point random crossover to produce offspring; otherwise, we use multiple-point random crossover to get better solution.

After crossover, we use integer randomization mutation to change the genes of new generated offspring with a given higher mutation rate as suggested by Tate and Smith[25] to find a solution faster. As for the mutation mechanism, mutation operator uses both single-point and multiple-point integer random mutation. In single-point integer random mutation, a gene to be mutated is picked at random and then is replaced with randomly selected valid value of the parameters; while multiple-point integer random mutation picks several genes randomly and replaces them with randomly selected valid values of the parameters. After mutation operation, new mutated offspring will become individu-

als of the population by replacing two individuals with the lowest fitness values.

In order to find the solution faster, our algorithm mutates the best individual at the end of each generation to generate new mutant that replaces the individual with the lowest fitness value. This best individual is still kept in future population.

5 Parallel Genetic Algorithm

Luque and Alba[26] described four parallel models: global model, distributed model, cellular model, and hybrid model. Spark is based on the master-slave distributed computing model and it is suitable for the global and distributed model of GA parallelization. Therefore, we use Spark to parallelize GA and implement two-phase parallelization algorithm: fitness evaluation and genetic operation. Both of them aim to improve the performance and effectiveness of GA in searching near-minimum test suite.

5.1 Architecture

The proposed two-phase parallelization algorithm is based on Spark resilient distributed dataset (RDD)[5]. The whole population is stored as RDD and is cached in memory, which allows future actions to be much faster. The architecture of our approach is shown in Fig.2.

Fitness evaluation is the first phase parallelization which includes stages 1∼3. Stage 1 generates initial population which is then parallelized into different partitions of RDD in stage 2. The fitness value of each individual is evaluated on different workers from stage 2 to stage 3. The first phase parallelization fits the global model. Genetic operation is the second phase parallelization which includes stages 4∼6. Stage 4 divides population with fitness value into numbers of subpopulations that are cached into different partitions of RDD. Genetic operations are then performed in parallel from stage 4 to stage 5. Stage 6 collects the best individuals from different partitions. The second phase parallelization fits the distributed model.

5.2 Parallel Fitness Evaluation

Our basic idea of fitness evaluation parallelization is to parallelize initial population into RDD and evaluate each individual's fitness value on different workers. Then the driver collects the results and applies genetic operators. The lineage graph of parallel fitness evaluation is shown in Fig.3. First, initial population is parallelized into populationRDD by *parallelize*() method of Spark. Then a $map(\_.assessFitness())$ transformation is applied to transform populationRDD into fitnessRDD which contains (individual, fitness value) pairs. The function $assessFitness()$ that evaluates individual's fitness value is passed in the driver program to run on the cluster. Finally, the *collect*() action starts to collect these pairs to the driver.

5.3 Parallel Genetic Operation

After the result of parallel fitness evaluation returns to the driver, if the result does not contain the solution, the algorithm continues to apply parallel genetic operation. The lineage graph of parallel genetic operation is shown in Fig.4.

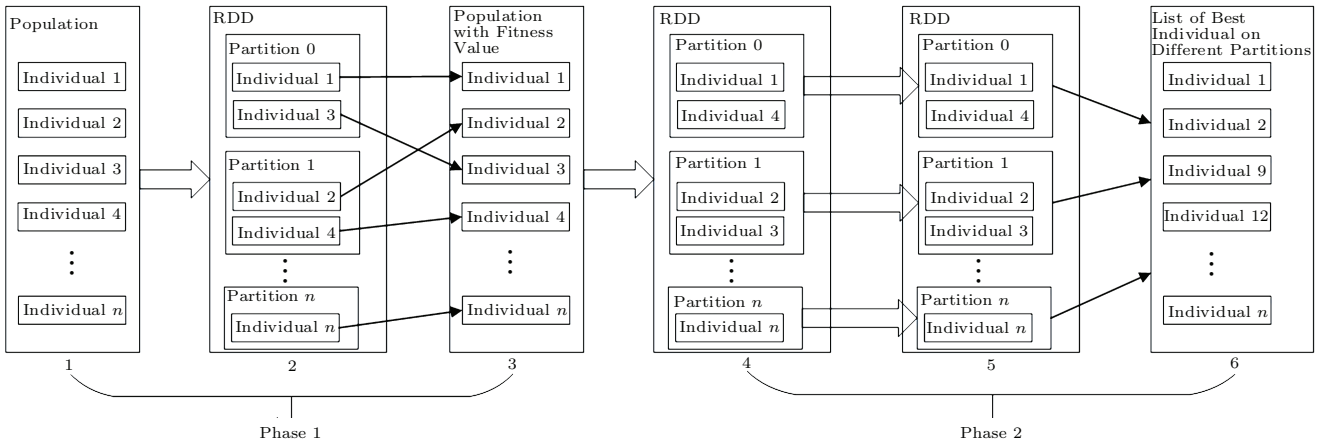


Fig.2. Architecture of two-phase parallelization.

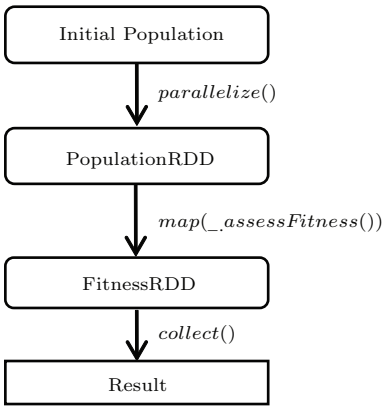


Fig.3. RDD lineage graph for phase 1.

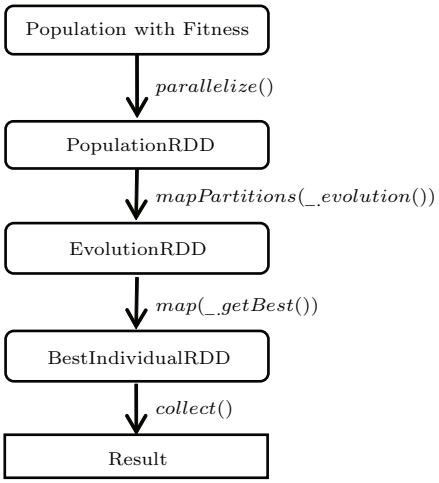


Fig.4. RDD lineage graph for phase 2.

First, population with fitness value is parallelized into populationRDD by *parallelize*() method of Spark. Then a *mapPartitions*(_.*evolution*()) transformation divides population into numbers of subpopulations that are cached into different partitions of populationRDD and transforms populationRDD into evolutionRDD. The number of partitions depends on the dimensions of the cluster. The function *evolution*() that applies genetic operators is passed in the driver program to run on the cluster. Then another *map*(_.*getBest*()) transformation is applied to transform evolutionRDD into bestIndividualRDD that contains (key, value) pairs, where key is a pair (individual, fitness value) and value is the number of generation needed to find a solution. The function *getBest*() that gets the best individual of each subpopulation is passed in the driver program to run on the cluster. Finally, the *collect*() action collects the best individuals on every worker to find the best solution.

5.4 Algorithms

Algorithm 1 sketches the pseudo-code of two-phase parallelization. It takes as inputs the parameter-values text file, the number of parameters, the number of values for each parameter, the test suite size, and the desired population size. The output is a near-minimum test suite with 100% pairwise coverage. At the beginning, the number of all pairs to be covered is generated (line 1). The initial population consists of *popsize* individuals, each of which is made up of m test cases that are created randomly by picking each slot uniformly among all possible values (line 2). Then the algorithm enters the first-phase parallelization (lines 3∼7) as illustrated in Fig.3. Then the (individual, fitness value) key-value pairs collected by the driver are sorted by the fitness value (line 8). If the first pair's value is AP, its key is the best individual that will be returned (line 9). Otherwise, the algorithm enters the second-phase parallelization (lines 10∼15) as illustrated in Fig.4. Then the list of (key, value) pairs, where key is a pair (individual, fitness value) and value is the number of generation needed to find a solution, is sorted by the fitness value and the number of generation (line 16). If the first pair's value is AP, its key is the best individual that will be returned (line 17).

Algorithm 1. Parallel Genetic Algorithm

Input: *pv.txt:* parameter-values text file
k: number of parameters in SUT
v_i: number of values for each parameter
m : test suite size
popsize: desired population size

Output: a pairwise test suite that covers all pairs of values at least once

1: $AP \leftarrow getNumOfAllPairs(k, v_i)$
2: $Population \leftarrow initializePop(m, popsize, \text{“}pv.txt\text{''})$
3: $populationRDD \leftarrow parallelize(Population)$
4: **For** each individual $I_j \in populationRDD$ **do**
5: $fitnessRDD \leftarrow I_j.map(\_.assessFitness())$
6: **End for**
7: $result \leftarrow fitnessRDD.collect()$
8: $sortByValue(result)$
9: **If** $result.top = AP$ **then return** top
10: $populationRDD \leftarrow parallelize(PopWithFitness)$
11: **For** each subPopulation $I_j \in populationRDD$ **do**
12: $evolutionRDD \leftarrow I_j.mapPartitions(\_.evolution())$
13: $bestIndividualRDD \leftarrow$ $evolutionRDD.map(\_.getBest())$
14: **End for**
15: $result \leftarrow bestIndividualRDD.collect()$
16: $sortByValue(result)$
17: **If** $result.top = AP$ **then return** top

In Algorithm 1, three functions are passed in the driver program to run on the cluster: $assessFitness()$, $evolution()$, and $getBest()$. Among these three functions, $evolution()$ that involves iterations of genetic operation including selection, crossover, and mutation is the most time-consuming phase.

Algorithm 2 describes the pseudo-code of the evolution process. It takes as inputs the population, the maximum number of generation, and the desired population size. Evolved populations on every worker are outputted. The external loop (lines 2∼12) conducts the whole evolution process. In each iteration of the external loop, the algorithm enters an inner loop which applies genetic operators including selection, crossover, mutation, and replacement (lines 3∼9). After leaving the inner loop, the algorithm mutates the best individual (line 10). Then the algorithm enters the external loop to start the evolution again. The evolution process is continued until it reaches the maximum number of generations or the solution has been found. Finally, it returns the evolved populations to Algorithm 1 (line 13).

Algorithm 2. Evolution Process

Input: P: population
max: maximum number of generation
$popsize$: desired population size
Output: evolved populations on every worker

```
1:  it ← 1
2:  While (it ⩽ max && the ideal solution not found)
3:    For popsize/2 times do
4:      Parent P1 ← Selection(P)
5:      Parent P2 ← Selection(P)
6:      Children C1, C2 ← Crossover(Copy(P1), Copy(P2))
7:      Mutate(C1, C2)
8:      ReplaceWorstIndividual(P)
9:    End for
10:   Mutate the best individual
11:   it ← it + 1
12: End While
13: Return the evolved populations
```

6 Preliminary Evaluation

In this section, we study the behavior of PGAS when solving pairwise test suite generation problem. We perform two categories of experiments: comparison with the Sequential Genetic Algorithm (SGA), comparison with other test generation approaches reported in literatures. Finally, we analyze the main threats to validity.

6.1 Experimental Design

To conduct experiments, we implement PGAS on Spark using Java and run PGAS on a small cluster consisting of five nodes, where each node has one Intel Core i5 750 Quad-Core at 2.66 GHz CPU, 4 GB RAM. One node is the namenode and the other four nodes are the datanodes that have 16 cores altogether. Each node is running at the Ubuntu 12.04, Java 1.7, Hadoop 2.4, and Spark 1.1.0.

In our experiments, we select 14 pairwise synthetic benchmarks and 5 real-world benchmarks presented by Jia *et al.*[27] and Garvin *et al.*[28] Table 2 shows these benchmarks. The tuples column represents the total number of pairs of each benchmark.

Table 2. Synthetic and Real-World Benchmarks

		Model	Tuples
Synthetic benchmarks	S 1	3^4	54
	S 2	$5^1 3^8 2^2$	492
	S 3	3^{13}	702
	S 4	$5^1 4^4 3^{11} 2^5$	1 944
	S 5	$6^1 5^1 4^6 3^8 2^3$	1 992
	S 6	$7^1 6^1 5^1 4^5 3^8 2^3$	2 175
	S 7	6^{16}	4 320
	S 8	7^{16}	5 880
	S 9	8^{16}	7 680
	S 10	8^{17}	8 704
	S 11	$4^{15} 3^{17}\ 2^{29}$	14 026
	S 12	$4^1 3^{39}\ 2^{35}$	17 987
	S 13	10^{20}	18 000
	S 14	4^{100}	79 200
Real-world benchmarks	SPIN-S	$2^{13} 4^5$	992
	Bugzilla	$2^{49} 3^1 4^2$	5 822
	SPIN-V	$2^{42} 3^2 4^{11}$	8 797
	Apache	$2^{158} 3^8 4^4 5^1 6^1$	66 930
	GCC	$2^{189} 3^{10}$	82 809

There are five parameters that impact the performance of PGAS: the population size, the number of crossover points, the number of mutation points, the maximum number of generations, and the number of partitions. A large population can increase the population diversity. When using large population, PGAS can still find solutions in reasonable time because of the parallelization. The number of crossover points and that of mutation points are directly proportional to the size of individual. The number of partitions equals the number of subpopulations generated by Spark. It is generally set to 16 based on our cluster. Table 3 shows three different parameter settings. Parameter setting

1 is the small setting that is used for the small size benchmarks (i.e., S 1∼S 3). Parameter setting 2 is the medium setting that is used for the medium size benchmarks (i.e., S 4∼S 10, SPIN-S, Bugzilla, and SPIN-V). Parameter setting 3 is the large setting which is used for the large size benchmarks (i.e., S 11∼S 14, Apache, and GCC). These parameter settings are obtained according to the actual experiments. Because of the stochastic nature of GA, we perform 30 independent runs of each benchmark.

Table 3. Parameter Settings

Parameter Name	Setting 1	Setting 2	Setting 3
Population size	4 800	8 000	12 800
Crossover	1-point	3-point	5-point
Mutation	2-point	5-point	5-point
Maximum number of generations	10 000	20 000	50 000

6.2 Comparison with SGA

We compare the performance between SGA and PGAS. Compared SGA approaches include GAPTS[7], GA[8], and PWiseGen[9]. In order to make a comprehensive comparison with SGA, we download PWiseGen designed by Flores and Yoonsik[9] and run all the synthetic benchmarks on one node of the cluster. We use the parameter settings shown in Table 3 for PGAS and PWiseGen.

Table 4 shows the smallest sizes of test suites generated by each algorithm. All the data for GAPTS and GA are taken from McCaffrey[7] and Shiba *et al.*[8] Entries marked with "-" represent that the data are not reported in these papers. The time column shows the average running time in second over 30 runs. The best known results are shown in bold in this table and subsequent tables.

As it can be seen from Table 4, PGAS outperforms SGA in 9 out of 14 benchmarks. In 4 out of 14 benchmarks, PGAS can produce test suites with sizes that equal SGA. S 13 is interesting as described by Jia *et al.*[27] and we leave further study to future work.

In order to compare the performance of PGAS with that of SGA comprehensively, we also evaluate them by using measures of computational effort, solution quality and speedup[26]. Computational effort is the execution time in finding the solution. Solution quality can be defined as the percentage of runs terminating with success. It is denoted by hit. Speedup is the ratio between SGA execution time and PGAS execution time.

Table 4. Comparison with SGA

Model	SGA			PGAS	
	GAPTS	GA	PWiseGen	Size	Time (s)
S 1	9	9	9	9	3
S 2	-	**15**	17	**15**	10
S 3	**15**	17	**15**	**15**	19
S 4	**-**	26	**26**	**24**	108
S 5	-	33	34	**31**	112
S 6	-	42	44	**42**	58
S 7	-	-	72	**68**	129
S 8	-	-	94	**90**	168
S 9	-	-	120	**115**	276
S 10	-	-	122	**118**	305
S 11	35	37	34	**32**	468
S 12	27	27	26	**23**	315
S 13	**196**	227	220	219	596
S 14	-	-	57	**48**	1 689

It is shown in Table 4 that SGA can get the same sizes of test suites as PGAS only in S 1 and S 3. We cannot get the execution time and hit of SGA when using the sizes of test suites generated by PGAS and cannot compare the performance between them. Thus we run PGAS again using the sizes of test suites generated by SGA in Table 4. Table 5 shows the average results of this comparison. This table reports the sizes of test suites (size), the average execution time in seconds (time), and the percentage of runs terminating with success (hit).

As it can be seen from Table 5, the time column reveals that when the benchmark size is smaller (S 1), PGAS is slower than SGA because of the communication overhead between the driver and workers in Spark. But with the increase of the benchmark sizes, PGAS can find solutions faster than SGA because of the two-phase parallelization. The last two columns reveal that PGAS can obtain a higher number of hits and better speedup than SGA. In 13 out of 14 benchmarks, the speedup varies from 2 to 18 times.

From Table 4 and Table 5, we can conclude that PGAS outperforms SGA in both the sizes of generated test suites and computational performance.

6.3 Comparison with Other Approaches

We compare the best reported sizes of test suites generated by other approaches with our results. Compared approaches include the greedy algorithm (IPO[1], mTCG[11,29], and mAETG[11,29]) and the heuristic search algorithm (ACA[8], PSO[30], and SA[27-28]). The benchmarks are synthetic benchmarks in Table 2.

Table 5. Average Results for PGAS and SGA

Model	Size	Algorithm	Time (s)	Hit(%)	Speedup
S 1	9	PGSA	3	100	-
		SGA	< 1	100	
S 2	17	PGSA	11	100	14.36
		SGA	158	70	
S 3	15	PGSA	19	50	7.68
		SGA	146	20	
S 4	26	PGSA	17	100	17.65
		SGA	300	20	
S 5	34	PGSA	20	100	5.70
		SGA	114	50	
S 6	44	PGSA	25	100	2.76
		SGA	69	40	
S 7	72	PGSA	108	60	3.58
		SGA	387	40	
S 8	94	PGSA	153	40	2.82
		SGA	432	20	
S 9	120	PGSA	223	40	2.18
		SGA	487	20	
S 10	122	PGSA	278	30	2.04
		SGA	566	10	
S 11	34	PGSA	418	80	1.56
		SGA	653	40	
S 12	26	PGSA	245	100	2.51
		SGA	614	60	
S 13	220	PGSA	546	40	1.52
		SGA	832	20	
S 14	57	PGSA	1 320	50	1.69
		SGA	2 234	20	

Table 6 reports the smallest sizes of test suites generated by each algorithm. The best column reports the best results produced by the compared approaches. In all of the 14 benchmarks except S 13, PGAS is superior to the greedy algorithm, ACA, and PSO. When compared with SA, PGAS can produce test suites with sizes that are equal to SA in 4 out of 14 benchmarks. In other 10 benchmarks, SA always generates smaller test suites than PGAS. We think that SA is better than PGAS because of its search strategy[11,27-28]. How to improve the search ability of GA needs to be further studied.

We also run the five real-world benchmarks in Table 2. Compared approaches include ACTS①, PICT①, and CASA[28]. Table 7 reports the smallest sizes of test suites generated by each algorithm. The results obtained by CASA are average sizes over 5 runs. The time column shows the average running time in second over 30 runs. From Table 7, we find that PGAS outperforms ACTS and PICT for every benchmark and can generate test suites with almost the same sizes as CASA.

6.4 Threats to Validity

In this subsection, we will discuss threats to validity of this paper. For the external validity, we use a relative small cluster when conducting our experiments and thus do not make full use of Spark's ability. We will use large-scale cluster for our future research. There are

Table 6. Results of Synthetic Benchmarks

Model	Best	Greedy			SA			ACA	PSO	PGAS	
		IPO	mTCG	mAETG	SA	CASA	HHSA			Size	Time (s)
S 1	9	9	9	9	9	9	9	9	9	9	3
S 2	**15**	-	18	20	**15**	**15**	**15**	16	17	**15**	10
S 3	**15**	19	17	17	16	**15**	**15**	17	18	**15**	19
S 4	**21**	36	28	28	**21**	23	**21**	25	27	24	108
S 5	**30**	-	35	35	**30**	**30**	**30**	32	35	31	112
S 6	**42**	-	**42**	44	**42**	**42**	**42**	**42**	43	**42**	58
S 7	**62**	-	-	70	**62**	64	63	-	-	68	129
S 8	**86**	-	-	94	87	**86**	**86**	-	-	90	168
S 9	**111**	-	-	120	112	112	**111**	-	-	115	276
S 10	**113**	-	-	123	114	114	**113**	-	-	118	305
S 11	**29**	-	34	37	30	30	**29**	37	38	32	468
S 12	**21**	-	26	27	**21**	22	**21**	27	27	23	315
S 13	**183**	218	213	198	**183**	185	189	225	213	219	596
S 14	**45**	-	56	56	**45**	46	**45**	-	-	48	1 689

①http://csrc.nist.gov/groups/SNS/acts/documents/ACTS.PICT.Comparison.xls, June 2015.

two threats to internal validity. First, the intrinsic randomness of GA may affect the performance of PGAS and we only consider the basic GA. Therefore, we will use hybrid GA to perfect our research. Second, there may be better parameter settings to get better results in some benchmarks, but how to choose these parameter settings is difficult. As for the construct validity, our approaches have two current limitations. First, we do not consider constraints in this paper. Second, we do not consider high strength test suite generation ($t > 2$). They are two potential challenges to be solved in our future work.

Table 7. Results of Real-World Benchmarks

Model	ACTS	PICT	CASA	PGAS	
				Size	Time (s)
SPIN-S	20	26	**16.4**	**17**	108
Bugzilla	18	20	**16.0**	**16**	43
SPIN-V	30	63	**26.4**	**27**	98
Apache	34	39	**32.0**	**32**	1 225
GCC	19	30	**17.0**	**17**	1 896

7 Conclusions

In this paper, we proposed a parallel genetic algorithm using Spark, called PGAS, for pairwise test suite generation. Based on the global and distributed model of GA parallelization, PGAS offers two-phase parallelization. The first phase is the fitness evaluation parallelization that evaluates each individual's fitness value on the workers. The second phase is the genetic operation parallelization that splits the population into different slices which can be evolved separately on different workers. To the best of our knowledge, PGAS is the first attempt to parallelize GA using Spark for generating pairwise test suite. We evaluated performance of PGAS against SGA in terms of computational effort, solution quality, and speedup. Results showed that PGAS outperforms SGA in both the sizes of generated test suites and execution time in 13 out of 14 benchmarks. We also compared PGAS with other test generation approaches reported in literatures in terms of sizes of generated test suites. Results showed that PGAS can produce test suites with sizes that are equal to other approaches in 9 out of 19 benchmarks. In summary, PGAS is a promising improvement of GA for pairwise test suite generation.

As for future work, first, we will run PGAS on large-scale cluster to find a smaller test suite size. Second, we plan to improve PGAS by addressing constrained and high strength test suite generation ($t > 2$). Third, we will improve PGAS by hybridizing with local search techniques and greedy techniques.

References

[1] Lei Y, Tai K C. In-parameter-order: A test generation strategy for pairwise testing. In *Proc. the 3rd IEEE International High-Assurance Systems Engineering Symposium*, Nov. 1998, pp.254-261.

[2] Kuhn D R, Wallace D R, Jr. Gallo A M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 2004, 30(6): 418-421.

[3] Nie C H, Leung H. A survey of combinatorial testing. *ACM Comput. Surv.*, 2011, 43(2): Article No. 11.

[4] Khalsa S K, Labiche Y. An orchestrated survey of available algorithms and tools for combinatorial testing. In *Proc. the 25th International Symposium on Software Reliability Engineering* (*ISSRE*), Nov. 2014, pp.323-334.

[5] Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin M J, Shenker S, Stoica I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. the 9th USENIX Conference on Networked Systems Design and Implementation*, April 2012, pp.15-28.

[6] Ghazi S A, Ahmed M A. Pair-wise test coverage using genetic algorithms. In *Proc. the 2003 Congress on Evolutionary Computation*, Dec. 2003, pp.1420-1424.

[7] McCaffrey J D. An empirical study of pairwise test set generation using a genetic algorithm. In *Proc. the 7th International Conference on Information Technology: New Generations* (*ITNG*), April 2010, pp.992-997.

[8] Shiba T, Tsuchiya T, Kikuno T. Using artificial life techniques to generate test cases for combinatorial testing. In *Proc. the 28th Annual International Computer Software and Applications Conference*, Sept. 2004, pp.72-77.

[9] Flores P, Yoonsik C. PWiseGen: Generating test cases for pairwise testing using genetic algorithms. In *Proc. IEEE International Conference on Computer Science and Automation Engineering* (*CSAE*), June 2011, pp.747-752.

[10] Nie C H, Wu H Y, Liang Y L, Leung H, Kuo F C, Li Z. Search based combinatorial testing. In *Proc. the 19th Asia-Pacific Software Engineering Conference* (*APSEC*), Dec. 2012, pp.778-783.

[11] Cohen M B, Gibbons P B, Mugridge W B, Colbourn C J. Constructing test suites for interaction testing. In *Proc. the 25th International Conference on Software Engineering*, May 2003, pp.38-48.

[12] Cohen M B, Colbourn C J, Ling A C H. Augmenting simulated annealing to build interaction test suites. In *Proc. the 14th International Symposium on Software Reliability Engineering*, Nov. 2003, pp.394-405.

[13] Petke J, Yoo S, Cohen M B, Harman M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proc. the 9th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2013*), August 2013, pp.26-36.

[14] Henard C, Papadakis M, Perrouin G, Klein J, Heymans P, Le Traon Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 2014, 40(7): 650-670.

[15] Alsewari A A, Zamli K Z. Interaction test data generation using Harmony Search algorithm. In *Proc. IEEE Symposium on Industrial Electronics and Applications* (*ISIEA*), Sept. 2011, pp.559-564.

[16] Li J H, Xing D D, Zhao Y Q. Combinatorial test suite generation of variable strength based on harmony search. *Journal of Network & Information Security*, 2013, 4(2): 177-188.

[17] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In *Proc. the 6th Symposium on Operating System Design and Implementation* (*OSDI*), Dec. 2004, Article No. 10.

[18] Jin C, Vecchiola C, Buyya R. MRPGA: An extension of MapReduce for parallelizing genetic algorithms. In *Proc. the 4th IEEE International Conference on eScience*, Dec. 2008, pp.214-221.

[19] Verma A, Llora X, Goldberg D E, Campbell R H. Scaling genetic algorithms using MapReduce. In *Proc. the 9th International Conference on Intelligent Systems Design and Applications*, Nov. 30-Dec. 2, 2009, pp.13-18.

[20] Geronimo D L, Ferrucci F, Murolo A, Sarro F. A parallel genetic algorithm based on Hadoop MapReduce for the automatic generation of JUnit test suites. In *Proc. the 5th IEEE International Conference on Software Testing, Verification and Validation*, April 2012, pp.785-793.

[21] Martino D S, Ferrucci F, Maggio V, Sarro F. Towards migrating genetic algorithms for test data generation to the cloud. In *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, Tilley S, Parveen T (eds.), IGI Global, 2013, pp.113-135.

[22] Younis M, Zamli K. MC-MIPOG: A parallel t-way test generation strategy for multicore systems. *ETRI Journal*, 2010, 32(1): 73-83.

[23] Lopez-Herrejon R E, Ferrer J, Chicano F, Haslinger E N, Egyed A, Alba E. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In *Proc. the 16th Genetic and Evolutionary Computation Conference*, July 2014, pp.1255-1262.

[24] Grindal M, Offutt J, Andler S F. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 2005, 15(3): 167-199.

[25] Tate D M, Smith A E. Expected allele coverage and the role of mutation in genetic algorithms. In *Proc. the 5th International Conference on Genetic Algorithms*, June 1993, pp.31-37.

[26] Luque G, Alba E. Parallel Genetic Algorithms: Theory and Real World Applications. Springer-Verlag Berlin Heidelberg, 2011.

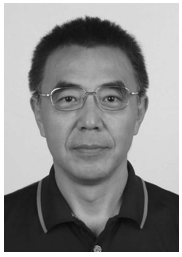
[27] Jia Y, Cohen M B, Harman M, Petke J. Learning combinatorial interaction test generation strategies using hyper-heuristic search. In *Proc. the 37th International Conference on Software Engineering* (*ICSE*), May 2015, pp.540-550.

[28] Garvin B J, Cohen M B, Dwyer M B. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 2011, 16(1): 61-102.

[29] Cohen M B. Designing test suites for software interaction testing [Ph.D. Thesis]. The University of Auckland, 2004.

[30] Chen X, Gu Q, Qi J X, Chen D X. Applying particle swarm optimization to pairwise testing. In *Proc. the 34th Annual IEEE Computer Software and Applications Conference*, July 2010, pp.107-116.

Rong-Zhi Qi is currently a Ph.D. candidate in the College of Computer and Information, Hohai University, Nanjing. He received his B.S. and M.S. degrees in computer science from the same university in 2002 and 2006 respectively. He is a member of CCF. His current research interest is search-based software engineering, especially on search-based combinatorial testing.



Zhi-Jian Wang received his B.S., M.S. and Ph.D. degrees in computer software from Nanjing University, Nanjing, in 1982, 1986, and 1990 respectively. He is currently a professor in the College of Computer and Information, Hohai University, Nanjing. He is a member of IEEE. His current research interests include domain software engineering and network security.

Shui-Yan Li is currently a Ph.D. candidate in the College of Computer and Information, Hohai University, Nanjing. She received her B.S. and M.S. degrees in applied mathematics from the same university in 2002 and 2006 respectively. Her current research interests include evolution computing and data mining.