Chapter VI

Towards a Parallel Approach for Test Data Generation for Branch Coverage with Genetic Algorithm using the Extended Path Prefix Strategy

1.1. Introduction

In this chapter we present a proposal for an approach to test data generation for branch coverage with a structured genetic algorithm (GA) using the extended path prefix strategy. The structured GA implements a parallel master-slave distributed model in which each slave implements an elitist panmictic GA. Branches to be covered are selected by the master using the extended path prefix strategy and then dispatched to slaves. The slaves then conduct search for test data to cover the assigned target branch. The master keeps track of all the branches that have been covered and also stores a specific number of individuals (test data) that traverse them in memory (referred to as test-data-memory here). The extended path prefix strategy ensures that each time a branch is selected for coverage, the sibling branch is already covered and that individuals are available in the test-data-memory that traverse the sibling. Thus along with the target branch, individuals are also dispatched to slaves that cover the sibling branch. Slaves keep track of additional branches that are traversed in the course of search and return the individuals that traverse them to the master. The extended path prefix strategy permits a variable number of slaves to be used which can help speed up the test data generation process. The proposed structured GA was implemented using MPI in C and run on a 24 blade, dual Xeon X5650, Dell® Cluster on some benchmark programs.

Remaining sections are organized as follows. Section 6.2 presents a description of panmictic and structured GAs. Section 6.3 describes the proposed parallel model.
Section 6.4 presents the experiments on two benchmark programs and Section 6.5 describes the results.

1.2. Genetic Algorithm and its Structured Implementation

In the standard GA, also called the panmictic GA, the whole population is dealt with as a single pool of individuals. As opposed to this, individuals can be arranged spatially giving rise to structured GAs (Alba and M. Tomassini, 2002). Traditionally structured GAs are associated with parallel implementations. There are three main types of parallel implementation models (Alba and M. Tomassini, 2002; Paz, 1998; Paz 2001; Konfrst, 2004): (a) global single-population master-slave model, (b) single-population fine-grained model, and (c) Multiple-population coarse grained model. These are illustrated in Figure 6.1.

![Parallel Models](image)

**Figure 6.1. Parallel Models**

In the master-slave model there is a single panmictic population, but the evaluation of fitness is distributed among several processors. In the single-population fine grained (or cellular) model, the interaction of individuals is restricted to a neighbourhood and these overlap thereby permitting interaction between all the individuals. This can be implemented very efficiently on massively parallel computers. In the multiple-population coarse grained (or distributed) model, there exist several subpopulations called islands which exchange individuals occasionally. This exchange of individuals is called
migration and is controlled by several parameters. Hybridizations also exist, which are also referred to as hierarchical models, in which each island can implement the master-slave or the fine-grained model.

1.3. **Parallel Test Data Generation**

We need to address the following issues for a parallel implementation

1. What parallel model?

2. How to select branches for coverage in parallel?

3. How to implement the parallel model with the branch selection strategy?

These are addressed in Section 6.3.1, Section 6.3.2 and Section 6.3.3 respectively.

1.3.1. **A Parallel Model for Test Data Generation**

In this chapter, we adopt a hybrid master-slave and island model strategy in which the master selects the branches for coverage passes one each to the slaves. The slaves implement a panmictic genetic algorithm to generate test data to cover the assigned branch. Figure 6.2 captures the essence of the model.

![Parallel Model adopted for Test Data Generation](image.png)
1.3.2. The Extended Path Prefix Strategy for Branch Selection

An untraversed branch can be selected for coverage using the path prefix strategy. In order to achieve path coverage, as opposed to branch coverage, (Prather and Myers, 1987) suggested the use of an adaptive strategy in which one new test path, or sub-path, is added at a time and previous paths serve as a guide for selection of subsequent paths using some inductive strategy. This strategy can be used to select a branch for coverage and is explained as follows. For a path \( p \) that is traversed in an execution, a *reversible prefix* \( q \) is defined as the minimal initial portion of path \( p \) to a decision node \( d \), whose branches are not fully covered, and the branch that is covered by \( p \). For example, in Figure 6.3, if path traversed is \( p: s, \ldots, d, b, \ldots, f \), branch \((d, b')\) is not covered and \( s, \ldots, d \) is the minimal initial portion of \( p \) that satisfies the condition above, then \( s, \ldots, d, b \) is a reversible prefix. The path \( s, \ldots, d, b' \) is said to be the *reversal* of path \( s, \ldots, d, b \). Accordingly, branch \((d, b')\) is a candidate for selection for coverage.

This can be extended to selection of multiple branches for coverage. If at any stage, \( p_1, \ldots, p_k \) are distinct paths traversed by sets of individuals \( M_1, \ldots, M_k \) respectively and \( r_1, \ldots, r_k \) are the reversible prefixes over \( p_1, \ldots, p_k \) respectively, then the reversals \( r_1, \ldots, r_k \) yield \( k \), possibly non-distinct, branches that can potentially be covered in parallel. The reason why these \( k \) branches are good choices for coverage is that there are sets of individuals \( M_1, \ldots, M_k \) for which execution paths traverse sibling branches. It may be thus become easier to ‘optimize’, i.e., generate new individuals to cover the chosen branches if parallel populations are initialized with individuals that traverse sibling branches. The use of reversible path prefixes over \( p_1, \ldots, p_k \) to select branches to cover in parallel is referred to as the *extended path prefix strategy.*

Figure 6.3 Example Graph for Path Prefix Strategy
1.3.3. **Test-Data-Memory**

In one generation of a GA run, since the program under test is executed once for each individual in the population, it is possible that branches other than the target branch which have not been covered earlier are traversed. Further, it is possible that when a branch is selected as the target branch, although its sibling branch has been traversed earlier, no individual in the current population results in an execution path that reaches the corresponding predicate node. In this situation, it would clearly be helpful if the population is supplemented with those individuals whose execution paths cover the target’s sibling branch. In order to facilitate this, each time a branch is traversed for the first time; up to five individuals that cause the branch to be traversed are stored. This is what we refer to as test-data-memory or simply *memory*. Now each time a branch is selected for coverage, up to five worst individuals in the current population are replaced with individuals from memory that traverse its sibling branch. However, it should be noted here that additionally an elitist strategy may be required to preserve individuals that traverse the sibling, across generations.

1.3.4. **The Implementation of the Parallel Model**

As described in Section 6.3.1, a parallel model which is a hybrid of the island and master-slave model is adopted. The basic task of the master is to coordinate the distribution of branches to the slaves and to keep track of the branches that are already covered. The master can choose to assign distinct branches to distinct slaves or multiple branches to a slave and thus employ a variable number of slaves depending on availability of slaves and performance demands. The steps executed by the master are described in Figure 6.4 and the slave operation in Figure 6.5.

The master starts with a randomly generated population $M$ in step 1 and executes steps 2 to 24 till the termination criterion is met. The termination criterion is a predefined maximum number of iterations. This is to ensure that problems like infeasibility do not result in a non-terminating loop. For each individual $q$ in $M$, steps 4 to 12 execute $P$ on the encoded inputs in $q$; check if new
branches are covered and store individual \( q \) in the memory associated with each new branch that is traversed with \( q \) and step 11 stores \( q \) in the memory associated with a traversed branch if the number of stored individuals is less than five. These steps update the ‘memory’ associated with each branch. Step 17 identifies the branches to be covered using the extended path prefix strategy, Step 18 assigns these branches to the slaves and step 19 dispatches the pair \((b, Mb_i)\) to slave \( s_i \). The pair consists of the allocated branch and the individuals that traverse \( b_i \)'s sibling branch. These individuals are obtained from the memory associated with \( b_i \)'s sibling branch. The extended path prefix strategy ensures that \( Mb_i \) is not empty. The master then waits for each slave to terminate. Each slave on termination returns a set of individuals to the master which the master adds to an initially empty set \( M \) in step 23. \( M \) is thus the union of the sets of individuals returned by the slaves to the master and is the input to the next iteration in the master.

The slave implements an elitist panmictic GA. On receiving the pair \((b, Mb)\) from the master, the GA run begins with a randomly generated population \( M \) in step 4. The GA generations are coded from steps 6 to 25. In each generation, for each individual \( x \) in the current population \( pop_{cur} \), program \( P \) is executed on \( x \) and if some new branch \( q \) is traversed, \( x \) is stored in a separate set of individuals \( X \) in Step 11. If the selected branch is traversed the GA is terminated and step 27 is executed. In step 27, the union of the current population \( M \) and \( X \) is returned. This thus includes at least one individual that traverses the target branch and individuals that traverse branches that were untraversed earlier. The slaves do not coordinate amongst themselves to keep track of newly traversed edges. The Slave implements an elitist GA. Individuals in \( Mb \) are replaced with fitter individuals in \( pop_{cur} \) in Step 23 and these individuals replace the worst individuals in the population for the next generation in Step 25. Steps 16 to 22 implement the steps in a simple genetic algorithm.

Let \( P_t \) be the instrumented program under test and let \( s_1, ..., s_n \) be the \( n \) slaves to which branches can be assigned. Let \( T \) be the set of traversed branches.
Figure 6.6 outlines the communication that takes place between the master and slaves over time with a sequence diagram.
Algorithm Master

1. Set $T = \emptyset$;
2. Randomly initialize population $M$;
3. While (termination criterion is not met) {
   4. For each individual $q$ in $M$ {
      5. Execute $P_t$ on input decoded from $q$ and record the path $p$ traversed in $P_t$;
      6. If ($p$ covers a previously untraversed branch $b$) {
         7. Store $q$ in the memory associated with $b$;
         8. Add $b$ to $T$;
      9. }
      10. If ($p$ covers a previously traversed branch $b$)
         11. Store $q$ in the memory associated with $b$ if the number of stored branches is less than 5;
   12. }
   13. If ($B == T$) {
      14. Decode and output all individuals stored in the memory associated with the branches;
      15. break;
   16. }
   17. Let $p_1, ..., p_m$ be the distinct paths traversed by individuals in $M$. Use the extended path-prefix strategy to identify a set of $k$ distinct, reversible prefixes $r_1, ..., r_{\overline{p_1}, ..., p_m}$ and the corresponding reversals $rr_1, ..., rr_k$. Let $b_1, ..., b_k$ be the branches identified for coverage using $rr_1, ..., rr_k$ respectively and let $Mb_i$ be the individuals that traverse sibling of branch $b_i$ in reversible prefix $r_i$;
   18. Assign $b_1, ..., b_k$ to $n$ slaves;
   19. Dispatch $(b_1, Mb_1, T), ..., (b_k, Mb_k, T)$ to slaves $s_1, ..., s_n$. If more than one triple is assigned to slave $j$, then slave $s_j$ is run sequentially on these triples;
   20. Wait for each slave to complete its task;
   21. Receive population $M_i$ from each slave $s_i$;
   22. Set $M = \emptyset$;
   23. Set $M = \bigcup_{i=1}^{k} M_i$;
   24. }

Figure 6.4 Master Algorithm
Algorithm Slave

1. Let \((b, Mb, T)\) be the data received from the Master;
2. Set \(T_{\text{temp}} = T\);
3. Set \(X = \emptyset\);
4. Randomly generate population \(M\) of \(n\) individuals for \(P\);
5. Replace worst \(|Mb|\) individuals in \(M\) with individuals in \(Mb\);
6. While (termination criterion is not met) {
   7. For each decoded individual \(x\) in the current population \(pop_{\text{cur}}\) {
      8. Execute \(P\), on \(x\)
      9. If \((q \notin T_{\text{temp}})\) {
         10. Add \(q\) to \(T_{\text{temp}}\);
         11. Add \(x\) to \(X\);
      }
      12. If (selected target branch \(b\) is traversed)
         13. Go to 27;
   }
   14. Calculate the fitness \(f(x)\) of each individual \(x\) in the population with respect to target \(b\);
   15. Initialize intermediate population \(pop_{\text{inter}}\) to empty;
   16. Repeat { // generate a new population
      17. Select a pair of parent individuals from \(pop_{\text{cur}}\);
      18. With probability \(P_c\) (the crossover probability), crossover the parents to form two offspring (or children);
      19. Mutate the two offspring with probability \(P_m\) (the mutation probability), and place the resulting individuals in \(pop_{\text{inter}}\);
      20. } until \((n\) individuals have been added to \(pop_{\text{inter}}\);
   21. Replace individuals in \(Mb\) with fitter individuals from \(pop_{\text{inter}}\);
   22. Replace individuals in \(pop_{\text{cur}}\) with individuals from \(pop_{\text{inter}}\);
   23. Replace worst \(|Mb|\) individuals in \(M\) with individuals in \(Mb\) (to implement elitism);
   24. }
25. Return \(M \cup X\);
26. }
27. Figure 6.5 Slave Algorithm: Panmictic GA for Test Data Generation

1.4. Experiments
In this section we describe the various experiments carried out to test the performance of test data generation with panmictic and parallel genetic algorithm. We first describe the benchmark program used and we then describe the main experiments.

**Benchmark Programs**

Benchmark programs chosen for the experiments have been taken from (D’iaz et al., 2008; Blanco et al., 2009). These programs have a number of features such as real inputs, equality conditions with the AND and OR operators, multiple predicate conditions and deeply nested predicates that make them suitable for testing different approaches for test data generation.

- **Line in a Rectangle Problem (Rectangle):** This program takes eight real inputs, four of which represent the coordinates of a rectangle and other four represents the coordinates of a line. The program determines the position of the line with respect to the position of rectangle and generates one out of four possible outputs:
  
  A. The line is completely inside the rectangle;
  B. The line is completely outside the rectangle;
  C. The line is partially covered by the rectangle; and
  D. Error: The input values do not define a line and/or a rectangle.

  The maximum nesting level is 12. The program’s CFG has 54 nodes with 18 predicate nodes. The length of Chromosome (Binary String) used is 168.

- **Sthamer’s Triangle Classifier Problem (triangle):** This program also classifies a triangle on the basis of its input sides as non-triangle or a triangle that is isosceles, equilateral, right angle triangle or scalene. It takes three real inputs; all of them represent the sides of the triangle but with different predicate conditions. It's CFG has 29 Nodes with 13 predicate nodes. The maximum nesting level is 12. It has equality conditions with AND operator and complex relational operators. The length of Chromosome (Binary String) used is 63.

1.4.1. **Experimental setup**
Experiments with the two approaches were carried out and compared independently for sequential Genetic Algorithm and for Parallel Genetic Algorithm. For each population size, hundred experiments were carried out and the following statistics were collected:

- **Mean number of generations.** For the parallel case, the number of generations in each experiment was taken to be the sum of the number of generations over all the executions of the slaves. The termination criterion was taken to be coverage or $10^7$ generations whichever occurred earlier. For the panmictic case coverage implied full branch coverage and in the parallel implementation coverage implied coverage of assigned target branch by a slave. The number of generations to termination over hundred experiments was used to compute the mean. The mean, however, does not tell us if all the branches were covered.

- **Mean percentage coverage achieved.**

Additionally ANOVA was carried out using SYSTAT 9.0 to determine significant difference in means for experiments. Table 6.1 lists the various operator and parameter settings for the genetic algorithm used in this study.
Table 6.1 GA Parameter and Operator Settings

<table>
<thead>
<tr>
<th>Parameter/Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>6, 10, 16, 20, 26, …, 110.</td>
</tr>
<tr>
<td>Crossover type</td>
<td>Two point crossover</td>
</tr>
<tr>
<td>Crossover Probability</td>
<td>1.0</td>
</tr>
<tr>
<td>Mutation Probability</td>
<td>0.01</td>
</tr>
<tr>
<td>Selection Method</td>
<td>Binary tournament</td>
</tr>
<tr>
<td>Adequacy Criterion</td>
<td>Branch Coverage</td>
</tr>
<tr>
<td>Branch Ordering Scheme</td>
<td>Panmictic Genetic Algorithm Path Prefix Strategy</td>
</tr>
<tr>
<td></td>
<td>Parallel Genetic Algorithm Extended Path Prefix Strategy</td>
</tr>
<tr>
<td>Fitness Function</td>
<td>Approach Level with Normalized Branch Distance</td>
</tr>
<tr>
<td>Population Replacement Strategy</td>
<td>Elitism with up to 10% carry forward</td>
</tr>
<tr>
<td>Maximum Number of Generations</td>
<td>10</td>
</tr>
<tr>
<td>Memory</td>
<td>Yes</td>
</tr>
<tr>
<td>Number of Slaves for Parallel Genetic Algorithm</td>
<td>Rectangle Program 10</td>
</tr>
<tr>
<td></td>
<td>Sthamer Triangle Program 10</td>
</tr>
</tbody>
</table>

1.5. Results

Figure 6.7 and Figure 6.8 graph the mean number of generations and mean percentage coverage for different populations for the triangle program and the rectangle program. For both the programs the results with the parallel approach are distinctly improved. Although with large population sizes, the sequential implementation is able to achieve full branch coverage. Table 6.2 summarizes the results of ANOVA for the differences in mean number of generations between panmictic and parallel GA. The table clearly show that the differences are significant for all population sizes.
Table 6.2 Statistical Results on Benchmark Programs for Mean Number of Generations
(Panmictic Genetic Algorithm Vs Parallel Genetic Algorithm)

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>ANOVA test</th>
<th>Sthamer triangle Program</th>
<th></th>
<th>Rectangle Program</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>F value</td>
<td>P value</td>
<td>F value</td>
<td>P value</td>
</tr>
<tr>
<td>6</td>
<td>121.708</td>
<td>0.0</td>
<td>26.388</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>85.241</td>
<td>0.0</td>
<td>22.45</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>40.498</td>
<td>0.0</td>
<td>33.334</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>32.531</td>
<td>0.0</td>
<td>16.343</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>40.503</td>
<td>0.0</td>
<td>35.783</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>29.3</td>
<td>0.0</td>
<td>22</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>28.901</td>
<td>0.0</td>
<td>14.394</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>21.498</td>
<td>0.0</td>
<td>17.249</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>22.017</td>
<td>0.0</td>
<td>17.012</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>12.811</td>
<td>0.0</td>
<td>9.37</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>24.254</td>
<td>0.0</td>
<td>23.51</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>9.77</td>
<td>0.002</td>
<td>25.565</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>10.623</td>
<td>0.001</td>
<td>23.229</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>9.948</td>
<td>0.002</td>
<td>30.35</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>8.162</td>
<td>0.005</td>
<td>26.639</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>15.112</td>
<td>0.0</td>
<td>26.06</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>8.693</td>
<td>0.004</td>
<td>23.704</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>6.98</td>
<td>0.009</td>
<td>24.324</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>7.403</td>
<td>0.007</td>
<td>19.321</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>4.933</td>
<td>0.027</td>
<td>43.149</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6.328</td>
<td>0.013</td>
<td>24.678</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>2.709</td>
<td>0.011</td>
<td>25.806</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>
Figure 6.7 Results on Sthamer Triangle Program

Figure 6.8 Results on Rectangle Progr