Automatic Generation of Valid Parallel-Processes Transition Using Genetic Algorithms and Communication-Flow Analysis

Ahmed S. Ghiduk$^{1,2}$  Alaa I. Elnashar$^{1,3}$

$^1$College of Computers and Information Technology, Taif University, KSA
$^2$Department of Mathematics and Computer Science, Faculty of Science, Beni-Suef University, Egypt
$^3$Department of Computer Science, Faculty of Science, Minia University, Egypt

asaghiduk@tu.edu.sa  nashar_al@yahoo.com

Abstract

Parallel programming has become an essential programming paradigm in the last years. Testing of parallel programs is more difficult than that of sequential programs due to the anomalous effects which occur due to concurrency and interactions among the parallel processes. In parallel programs, states grow exponentially with the number of concurrent processes. Therefore, finding the valid sequence of transitions through a set of parallel processes is a demand and also a challenging task due to state explosion. In this paper, we present an approach based on genetic algorithms and communication-flow analysis for generating the valid sequence of transitions through the set of parallel processes. The approach uses the communication-flow analysis to generate all sending-receiving pairs between the set of parallel processes. Then, the proposed approach applies a genetic algorithm to generate a set of valid sequences of processes transition. This set of valid sequences of transitions is passed to a feasibility checker to find the proper executable one. We conducted a set of experiments to evaluate the proposed approach. The results of the experiments showed that our proposed approach is more efficient than the random technique that is used to generate the executable transition sequence through the parallel processes.

1. Introduction

Software testing of parallel programs is more complicated than that of the sequential ones due to the increased complexity and the additional anomalous effects occur due to concurrency and interactions among parallel processes [1]. Nondeterministic behaviour [2] and finding the execution trace [3] are some of the hardest problems in testing such programs. In the case of send/wait synchronization, finding guaranteed ordering of processes (execution trace) is an NP-complete problem [4].

Checking the conflict among the parallel processes in a given trace can be completed in a polynomial-time [5, 6, 7]. This problem is known as the monitoring problem [7, 8]. Another problem of interest is the checking of conflict for all possible interleavings of a given trace. This problem is called the prediction problem. In predictive analysis, a violation is discovered either in the observed trace or in an alternate interleaving of events in that trace.

Static verification [9, 10, 11, 12, 13] and model checking [14, 15, 16, 17, 18, 19] are two approaches for program tracing and finding bugs in parallel programs.

Model checking is an exhaustive search technique and does not scale with program size. Several exhaustive techniques [20, 21, 22, 23] have been developed for testing parallel programs. These techniques exhaustively explore all interleavings of a parallel program by switching processes at synchronization points. Randomized algorithms for model checking like Monte Carlo Model Checking [24] have also been developed. It uses a random walk on the state space to give a probabilistic guarantee of the validity.

Randomized depth-first search [25] has been developed to reduce the state space search. A randomized partial order sampling algorithm [26] is used to sample partial orders almost uniformly at random. Race directed random testing [27] uses dynamic analysis tools to identify a set of pairs of statements that could potentially race in a parallel execution. Preissl et al. [28] developed a hybrid approach, with trace analysis to identify inefficient patterns in computation and communication.

Strout et al. [29] developed a dataflow analyzer for MPI programs, which models information flow through communication edges.

De Souza et al. [30] presented Intel Message Checker (IMC) to perform a post-mortem analysis by collecting all information on MPI calls in a trace file. After executing the program, this trace file is analyzed by a separate tool or compared with the results from previous runs [31].
There are some different message-checking tools like MPI-CHECK [32], Umpire [33], and MARMOT [34]. These debuggers are effective in program tracing but still poor to detect semantics-related bugs [35].

Huang et al. [36] presented a static trace simplification technique for reducing the context switches in a parallel program execution trace. The technique constructs a dependence graph model of events; it scales linearly to the trace size and quadratic to the number of nodes in the dependence graph.

Sinha et al. [5] presented a method to report a trace violation. The reported interleavings are guaranteed to be feasible in the actual program execution. Park and Sen [37] introduced a technique to detect real atomicity problems in parallel programs.

Recently, Kelk et al. [38] employed genetic algorithms to implement an automated system for finding a feasible trace for parallel Java programs such that this trace doesn't contain deadlocks and data races.

The main contributions of this paper are: introducing a new genetic algorithm based approach for generating the valid sequence of processes transition through set of parallel processes, presenting a testing tool by implementing the proposed approach, and conducting a set of experiments to evaluate the efficiency of the proposed approach and answer the following research questions:

RQ1: How effective is the proposed approach in generating the required processes transition sequence compared to the random technique?

RQ2: How effective is the proposed technique in reducing the explosion in the number of the processes transition sequences?

The paper is organized as follows: Section 2 introduces some basic definitions and concepts. Section 3 presents our proposed technique for generating the executable transition sequence among set of parallel processes. Section 4 shows the experiments which are conducted to evaluate the proposed technique. Section 5 presents the conclusion and future work.

2. Background

2.1 Testing of parallel programs

Recently, multiprocessors and supercomputers are widely used due to the variety advantages of parallelizm. In addition, new programming languages and libraries are introduced to enable the users to create and control multiple processes executing simultaneously on such systems.

A process is the execution of a limited number of actions on a fixed set of data [48, 49]. Two processes are parallel, if the execution of one process starts before the termination of the other process [48, 49].

Processes synchronization is essential when one process must wait for another one to finish before proceeding. For example, one process (called a writer) may be writing/sending data to a certain main memory area (variable definition), while another process (a reader) may be reading/receiving data from that area (variable use). The reader and writer processes must be synchronized so that writer doesn't overwrite [48, 49].

Processes synchronization superimposes restrictions on the order of process performing. These restrictions are synchronization rules, which are described by means of synchronization mechanisms (primitives). A great variety of such mechanisms have been proposed so far, and their characteristics have been discussed [39, 40].

Communication of parallel process means information exchange among these processes which needs some kind of synchronization either implicit or explicit to guarantee a proper information exchange among these processes [48, 49].

A parallel program $pp$ consists of several parallel executing, interacting, and communicating tasks, which are concurrently processing a given input to compute the desired output. It can be expressed as set $pp = \{t_1, t_2, ..., t_n\}$ where $t_i$ represents the parallel tasks, ($1 \leq i \leq n$), and $n \geq 2$ [41]. Each task contains a set of statements that will be executed sequentially.

An execution trace $T = p_1, p_2, ..., p_n$ is a sequence of processes, each process is an instance of a visible operation during the execution of the parallel program [3].

Happens-before principle is a relation between “reads” and “writes” of variables. It controls the possible values that a "read" of a variable may return. For a given variable: if a "write" of the value $v_1$ happens-before the "write" of a value $v_2$, and the "write" of $v_2$ happens-before a "read", then that "read" may not return $v_1$. Properly ordered "reads" and "writes" ensure that a "read" can only return the most recently written value. If an action A synchronizes with an action B then A happens-before B. So correct use of synchronization ensures a "read" can only return the most recently written value [42].

2.2 Genetic algorithms

The basic concepts of genetic algorithms were developed by Holland [43]. GAs start by creating an initial population of individuals, each represented by a randomly generated binary string called a chromosome. Figure 1 shows the basic algorithm for GAs. The fitness of each individual is determined. A selection method is used to find pairs of individuals that will be mated to contribute to the next generation. The process
of crossover involves two chromosomes swapping chunks of data. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The above algorithm iterates until the population has evolved to form a solution to the problem, or until a termination condition is satisfied.

Simple Genetic Algorithm ()
{
    initialize population;
    evaluate population;
    while termination criterion not reached
    {
        select solutions for next population;
        perform crossover and mutation;
        evaluate population;
    }
}

Figure 1: The basic algorithm for GAs.

3. Proposed processes transition generation technique (PTGT)

This section describes our proposed technique for generating the valid sequence of processes transition in the parallel program. This technique utilizes the communication-flow analysis to find all def-use pairs (sending-receiving pairs) in the parallel program. Then, the algorithm applies the genetic algorithms to construct the valid sequences of transitions. After that, the set of valid sequences of processes transition is passed to a feasibility checker to find the executable sequence of transition. The proposed technique consists of three main parts: communication-flow analyzer, processes transition generator, and feasibility checker. Figure 2 shows the overall diagram of PTGT. In the following, we discuss the main three parts of our proposed technique in more details.

3.1 Communication-flow analyzer (CFA):

Control-flow and Data-flow analysis techniques represent the tested program by its control-flow graph, CFG, which consists of a set of nodes and edges. Each node represents a basic block which is a set of consecutive statements of the program, and each edge represents the control flow between these blocks. The goal of data-flow analysis techniques is to identify which definitions of program variables can affect which uses.

The sequential program specification contains the definitions and uses of all program variables (e.g., statement "Define y = use(x)" in Figure 3 means that the variable y will be defined by referencing the variable "x"). Figure 3 shows a program specification example that defines four different variables and their uses. Ordinary data-flow analysis techniques fail to analyze such specifications because some references are encountered before their definitions. The sequential program can be divided into a set of processes containing some definitions and uses.

These processes have to be ordered in a way that satisfies all sending-receiving association among the processes to achieve the proper data flow. Therefore, a new program representation that enforces the references to wait till their definitions are available and also sending the required variables to the proper process is required. This can be achieved by using a special data structure to represent the sequential flow inside each process, and the parallel flow and synchronization among all processes [47].

The communication-flow analyzer is an adapted version of the data-flow analysis technique proposed by Elnashar [47]. This analyzer takes the sequential program specification as its input. It builds the CFG of the program according to the above definition. The constructed CFG is then used by the communication-flow analyzer to identify the def-use associations among the basic blocks. After that, the communication-flow analyzer divides the sequential program specifications into the corresponding set of parallel processes containing variables definitions and their uses constituting the synchronized-processes. Then, it represents these synchronized processes by a graph called message passing interface control-flow graph (MPI-CFG) [47]. Table 1 shows the synchronized processes correspond to the program specifications shown in Figure 3. Figure 4 shows the MPI control flow graph (MPI-CFG) for the four synchronized processes given in Table 1. In this graph, a definition-use association among processes can be achieved only by calling inter-process communication message passing (sending and receiving) routines that satisfy the valid processes transition. This MPI-CFG includes
three different types of edges that represent the sequential flow, parallel flow, and inter-processes communication and also four different types of nodes that represent send, receive, computation, data input blocks.

Table 1: Four MPI Synchronized Processes.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive x → P3</td>
<td>Receive v → P3</td>
<td>Receive y → P1</td>
<td>Define x</td>
</tr>
<tr>
<td>Define y = use(x)</td>
<td>Define t = use(v)</td>
<td>Define v = use(y)</td>
<td>Send t → P4</td>
</tr>
<tr>
<td>Send y → P3</td>
<td>Send t → P4</td>
<td>Send v → P2</td>
<td>Use (t)</td>
</tr>
</tbody>
</table>

Using the information of MPI synchronized processes; the communication-flow analyzer finds the set of valid sending-receiving pairs of the synchronized processes. Table 2 shows the set of valid sending-receiving pairs of the example program shown in Figure 3. In addition, the communication-flow analyzer computes the length of the valid sequence of transitions among the set of synchronized processes as the number of valid sending-receiving pairs plus one. Therefore, for the example program, the length of the valid sequence of transitions is 5. The communication-flow analyzer passes the set of valid sending-receiving pairs of the synchronized processes and the length of valid sequence of transitions to the "Processes Transition Generator".

Table 2: Valid Sending-Receiving Pairs.

<table>
<thead>
<tr>
<th>ID of Sending Process</th>
<th>ID of Receiving Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

3.2 Processes transition generator (PTG):

This part of our technique uses a genetic algorithm to generate the valid transition sequences. The details of our proposed genetic algorithm are given below.

To illustrate our proposed processes transition generator, we consider the program specification shown in Figure 3. The program contains 4 processes as shown in Table 1. In addition, there are 4 sending-receiving pairs in this program as given in Table 2 and the length of the sequence of transitions through this program is \( L = 5 \).

Our proposed genetic algorithm uses an array of integers as a chromosome to represent the sequence of processes transition. The length, \( L \), of the chromosome depends on the required number of transitions among the processes in the parallel program. The "Processes Transition Generator" gets the length of the chromosome from the communication-flow analyzer.

The chromosome representation of program \( pp \) is \([3, 1, 3, 2, 4]\) which corresponds to the transition sequence of processes: \( p#3, p#1, p#3, p#2, p#4 \). The proposed genetic algorithm randomly generates \( ps \) single-dimension arrays of length \( L \) to represent the initial population, where \( ps \) is the population size which is experimentally determined. In our experiments, \( ps \) is 10, 30, 40, 110. Table 3 shows the initial population of the program \( PP \).

To measure the validity of each generated sequence of processes transition we proposed a new fitness function \( f(a_i), i=1,..., ps \). The fitness function is the ratio of all valid transitions in the transition sequence to the total number of transitions in the sequence (the length of the required transition). If the fitness function equals 1 for a sequence \( a\text{best} \) this means that the algorithm found the required sequence of transitions.

If the proposed genetic algorithm fails to find the required sequence of transitions in the current generation, it generates a new population (set of \( ps \) new sequences) of transitions by applying the roulette wheel selection method, the uniform crossover, and single point mutation [44].
3.3 Feasibility checker (FC):

After the end of "Processes Transition Generator" phase, as shown in Figure 2, a set of processes transition sequences is obtained. All the members of this set satisfy the valid transition criterion from one process to another one and each member has fitness value equal to one.

The problem now is how to select a sequence from this set that represents a feasible trace. Therefore, we built a "feasibility checker" module that takes the set of processes transition sequences generated during the previous phase as input and returns only one valid executable processes transition sequence.

Suppose that the processes transition generator generates for the example shown in both Figure 4 and Table 1 a set of valid sequences as shown in Figure 5 and each member of this sequence has fitness value 1.

All sequences listed in Figure 5 satisfy "sending-receiving" pairs listed in Table 2. We can’t guarantee that each one of them represents a proper program execution. As an example, the sequence "2 4 1 3" is a valid one since each pair satisfies the happen-before principle, but it is not a feasible execution trace since process 2 can’t proceed until it receives the value of the variable "v" from process 3.

The task of "Feasibility Checker" is to select the sequence that does not only satisfy the "sending-receiving" pairs, but also satisfies the synchronization primitives among all synchronized processes.

Communication-Flow Analyzer creates three data structures: the first data structure represents the receive or wait statements and the second one represents the send statements within the inputted program. Each data structure contains, for each wait or send primitive, the process ID that includes that primitive, the statement number, variable name that this process will wait for or send to another process, and also the process ID that will send or receive that variable. The third generated data structure is the def-use structure that contains for each process: its ID, variable(s) name(s) and variable(s) status "Def" or "Use". These structures are essential for the feasibility checker.

The first step in this phase is the detection of the variable that will be used as key access [45]. A write (read) access is a key access if there doesn’t exist any other write (read or write) access to that variable within a block such that aₖ → aⱼ [45].

In our example the variable "x" is the key access since there is no write "happens before" its definition. So the process containing this variable should be selected as the first one that starts the message passing scenario as shown in data input block described in Figure 4.

The checker is divided into three passes. Pass1 uses "wait" and "send" structures to extract the key access variable [45] and also the start process and preserves them to be used by Pass2 which uses the "def-use" structure with the sequences set S to traverse each sequence $Seq_i \in S$, $1 \leq i \leq \text{number of sequences}$, $Seq_i : a_{i1}, a_{i2}, ... , a_{iL}$ is the sequence length. To find the sequence that satisfies the following two conditions: 1) start process = aⱼ, 2) $\exists$ synchronization edge $e : a_{k+1}, 1 \leq k \leq L - 1$, $\forall a_k \in Seq_i$, the checker rejects all the sequences that don’t satisfy these

<table>
<thead>
<tr>
<th>Sequence #</th>
<th>No. of valid transitions in each sequence</th>
<th>Total no. of valid transitions in each sequence</th>
<th>Fitness value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>3</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>a₂</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>a₃</td>
<td>1</td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td>a₄</td>
<td>4</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>a₅</td>
<td>2</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>a₆</td>
<td>2</td>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>a₇</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>a₈</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>a₉</td>
<td>1</td>
<td></td>
<td>0.25</td>
</tr>
<tr>
<td>a₁₀</td>
<td>2</td>
<td></td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 3: Initial Population.

Table 4: Validity Values.
conditions by marking them as "infeasible", otherwise the checked sequence will be concerned and is marked as "feasible".

If Pass2 fails to find such a sequence, then Pass3 starts and invokes "PTGT" to generate another sequences set S that may include a feasible sequence.

**Pass1**

1. Scan "wait" and "send" to detect the key access variable "keyAccessVar" and the start process
   1.1 kVar = keyAccessVar;
   1.2 defProcess = start process;
   1.3 useProcess = NULL;
2. traceLength = 0;
3. call Pass2;
4. end Pass1;

**Pass2**

for each $Seq_i \in S, 1 \leq i \leq number$ of sequences do
    read $Seq_i: a_1, a_2, ..., a_L, L$
    If $a_1 \neq defProcess$ then
        reject $Seq_i$: mark this sequence as "infeasible"
        get next sequence
    else
        defVar= kVar;
        traceLength = traceLength + 1
    endif
for each $a_j \in Seq_i, 2 \leq j \leq L$ do
    for each record in def-use file do
        If $\exists e: a_{j-1} \rightarrow a_j, 2 \leq j \leq L$ then
            useProcess = $a_j$
            traceLength = traceLength + 1
            defProcess = useProcess
            get next $a$
        else
            get next record
        endif
    endfor
if traceLength < L then
    reject $Seq_i$: mark this sequence as "infeasible"
    get next sequence
else
    traceLength = L
    accept $Seq_i$: mark this sequence as "feasible"
    found=true
endif
endfor

**Pass3**

if not found call PTGT to generate another sequences set S

Figure 6 shows the output of the checker for our example program which is shown in both Figure 4 and Table 1. Figure 7 shows the dynamic execution to confirm the static (generated) one. The dynamic execution is profiled using Jumpshot [46].

```
1:  2  4  1  3  2  infeasible
2:  1  3  2  4  1  infeasible
3:  2  4  1  3  2  infeasible
... ... ... ... ...
25:  3  2  4  1  3  infeasible
26:  2  4  1  3  2  infeasible
27:  4  1  3  2  4  feasible
```

Figure 6: Part of checker result file, containing all the sequences which have fitness value 1 and their feasibility.

Figure 7: Jumpshot- Time Line for the Synchronized Processes.

4. Experiments

This section introduces the conducted experiments to evaluate the efficiency of the proposed approach and answer the following research questions:

- RQ1: How effective is the proposed approach in generating the required processes transition sequence compared to the random technique?
- RQ2: How effective is the proposed technique in reducing the explosion in the number of the processes transition sequences?

We implemented the proposed approach in java and used it to conduct the experiments. We used a set of eight different parallel programs in our experiments. Five of these programs contains 5, 6, 7, 8, and 9 parallel processes. Three of these programs are different in their task but contains 10 parallel processes. Table 5 gives the details of the subject programs, the first column gives the number of the subject program, the second column gives the number of parallel processes in each program, the third column gives the length of the required sequence of processes transition, the fourth column gives the minimum number of existing transition sequences, the fifth column gives the number of valid sending-receiving pairs among the processes in each program, and the sixth gives the number of invalid sending-receiving pairs among the processes in each program.
Table 5: The Subject Parallel Programs.

<table>
<thead>
<tr>
<th>Program#</th>
<th>Processes</th>
<th>Length of Required Transition Sequence</th>
<th>Minimum No. of Sequence</th>
<th>No. of Valid Sending-Receiving Pairs</th>
<th>No. of Invalid Sending-Receiving Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP#1</td>
<td>5</td>
<td>1.20E+02</td>
<td>5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>PP#2</td>
<td>6</td>
<td>7.20E+02</td>
<td>6</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>PP#3</td>
<td>7</td>
<td>5.04E+03</td>
<td>7</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>PP#4</td>
<td>8</td>
<td>4.03E+04</td>
<td>8</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>PP#5</td>
<td>9</td>
<td>3.63E+05</td>
<td>9</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>PP#6</td>
<td>10</td>
<td>3.63E+06</td>
<td>10</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>PP#7</td>
<td>10</td>
<td>3.63E+06</td>
<td>10</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>PP#8</td>
<td>10</td>
<td>3.63E+06</td>
<td>10</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

We conducted the set of experiments as follows.
1. Adapt the GA parameters as flows. Population size (number of transition sequences in each generation) = 10 for programs PP#1, PP#2, and PP#3, Population size = 20 for programs PP#4, and PP#5, and Population size = 110 for programs PP#6, PP#7, and PP#8. Maximum number of generations = 20000 for all programs. Probability of crossover = 0.85 for all programs. Probability of mutation = 0.15 for all programs.
2. Adapt the random technique to generate the same number of transition sequences in each cycle as in GA technique and maximum number of generation is taken from 20000 to 3.4E18.
3. Run the communication-flow analyzer of our prototype to find the set of all valid sending-receiving pairs and the length of the required processes transition sequence.
4. Run the processes transition generator or the random technique to find set of valid processes transition sequences. Each sequence in this set has fitness value equal to 1. Some of the sequences in this set are not executable sequences. Therefore, we run the feasibility checker to find the executable sequence which represents the required one.
5. Because genetic algorithm is a heuristic approach, we repeat processes transition generator and random technique five times and take the average.

Table 6: No. of Generations of the Proposed Technique.

<table>
<thead>
<tr>
<th>Program#</th>
<th>Run1</th>
<th>Run2</th>
<th>Run3</th>
<th>Run4</th>
<th>Run5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP#1</td>
<td>119734</td>
<td>182708</td>
<td>166629</td>
<td>25643</td>
<td>117404</td>
<td>123349.8</td>
</tr>
<tr>
<td>PP#2</td>
<td>265632</td>
<td>110324</td>
<td>196273</td>
<td>196399</td>
<td>109912</td>
<td>175706.8</td>
</tr>
<tr>
<td>PP#3</td>
<td>656999</td>
<td>713707</td>
<td>3828054</td>
<td>600233</td>
<td>716941</td>
<td>1914378</td>
</tr>
<tr>
<td>PP#4</td>
<td>1.21E+08</td>
<td>1.08E+08</td>
<td>3.23E+08</td>
<td>9.1487500</td>
<td>9.1487546</td>
<td>1.471E+08</td>
</tr>
<tr>
<td>PP#5</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
</tr>
<tr>
<td>PP#7</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
</tr>
<tr>
<td>Total</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
</tr>
</tbody>
</table>

By comparing the results of our proposed technique and random technique which are given in Table 6 and Table 7, we find that the proposed approach is more effective than the random technique in generating the required processes transition sequence. The proposed approach needs less number of generations than random technique to find the target sequence of processes transition for all programs. In addition, by considering the reduction ratio (RRNG) of the number of generations as follows.

$$RRNG = \frac{\text{No. of Generations of RT} - \text{No. of Generations of PTGT}}{\text{No. of Generations of RT}}$$  \tag{1}$$

Table 7: No. of Generations of RT Technique.

<table>
<thead>
<tr>
<th>Program#</th>
<th>Run1</th>
<th>Run2</th>
<th>Run3</th>
<th>Run4</th>
<th>Run5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP#1</td>
<td>1.19734</td>
<td>1.82708</td>
<td>1.66629</td>
<td>2.5643</td>
<td>1.17404</td>
<td>1.23349.8</td>
</tr>
<tr>
<td>PP#2</td>
<td>2.65632</td>
<td>1.10324</td>
<td>1.96273</td>
<td>1.96399</td>
<td>1.09912</td>
<td>1.75706.8</td>
</tr>
<tr>
<td>PP#3</td>
<td>6.56999</td>
<td>7.13707</td>
<td>3.828054</td>
<td>6.00233</td>
<td>7.16941</td>
<td>19.14378</td>
</tr>
<tr>
<td>PP#4</td>
<td>1.21E+08</td>
<td>1.08E+08</td>
<td>3.23E+08</td>
<td>9.1487500</td>
<td>9.1487546</td>
<td>1.471E+08</td>
</tr>
<tr>
<td>PP#5</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
<td>2.31E+08</td>
</tr>
<tr>
<td>PP#7</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
<td>3.40E+15</td>
</tr>
<tr>
<td>Total</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
<td>1.02E+16</td>
</tr>
</tbody>
</table>

According to the reduction ratio equation (1), the proposed technique reduced the number of generations by 92% for program PP#1, 96% for program PP#2, and 99% for programs PP#3, PP#4, and PP#5, for programs
PP#6, PP#7, and PP#8 the random technique cannot find the required sequence of transitions. Table 8 gives the reduction ratio of number of generations for each subject program.

Table 8: Reduction Ratio of No. of Generations.

<table>
<thead>
<tr>
<th>Program</th>
<th>GA</th>
<th>RT</th>
<th>RRGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP#1</td>
<td>90734</td>
<td>200000</td>
<td>55%</td>
</tr>
<tr>
<td>PP#2</td>
<td>61792</td>
<td>200000</td>
<td>69%</td>
</tr>
<tr>
<td>PP#3</td>
<td>140144</td>
<td>300000</td>
<td>95%</td>
</tr>
<tr>
<td>PP#4</td>
<td>183915</td>
<td>14687594540</td>
<td>99.999%</td>
</tr>
<tr>
<td>PP#5</td>
<td>376306</td>
<td>23132218800</td>
<td>99.999%</td>
</tr>
<tr>
<td>PP#6</td>
<td>960410</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
<tr>
<td>PP#7</td>
<td>1821930</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
<tr>
<td>PP#8</td>
<td>1343320</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
</tbody>
</table>

By comparing the results of our proposed technique and the random technique which are given in Table 9, we find that the proposed approach is more effective than random technique in reducing the number of generated processes transition sequences to find the target sequence of transition. The proposed approach needs less number of generated transition sequences than random technique to find the target sequence of processes transition for all programs. In addition, by considering the reduction ratio (RRGS) of the number of generated sequences as follows:

$$RRGS = \frac{\text{No. of Generated Sequences by RT}}{\text{No. of Generated Sequences by PTGT}}$$

According to the reduction ratio equation (2), the proposed technique reduced the number of generated transition sequences by 55% for program PP#1, 69% for program PP#2, 95% for programs PP#3, 99.999% for programs PP#4, and PP#5, and for programs PP#6, PP#7, and PP#8 the random technique cannot find the required sequence of transitions. Table 9 gives the reduction ratio of number of generated sequences of processes transition (RRGS) for each subject program.

Table 9: No. of Generated Sequences of our Technique and Random Technique.

<table>
<thead>
<tr>
<th>Program</th>
<th>GA</th>
<th>RT</th>
<th>RRGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP#1</td>
<td>90734</td>
<td>200000</td>
<td>55%</td>
</tr>
<tr>
<td>PP#2</td>
<td>61792</td>
<td>200000</td>
<td>69%</td>
</tr>
<tr>
<td>PP#3</td>
<td>140144</td>
<td>300000</td>
<td>95%</td>
</tr>
<tr>
<td>PP#4</td>
<td>183915</td>
<td>14687594540</td>
<td>99.999%</td>
</tr>
<tr>
<td>PP#5</td>
<td>376306</td>
<td>23132218800</td>
<td>99.999%</td>
</tr>
<tr>
<td>PP#6</td>
<td>960410</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
<tr>
<td>PP#7</td>
<td>1821930</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
<tr>
<td>PP#8</td>
<td>1343320</td>
<td>3.4E+17</td>
<td>100%</td>
</tr>
</tbody>
</table>

Figure 8 shows the number of the required generations by the proposed technique and the random technique. It is clear that random technique needs more number of generations than the proposed technique. Figure 9 shows the generated number of transition sequences by the proposed technique and the random technique. It is clear that the random technique generated more number of transition sequences than the proposed technique till finding the required transition sequence for each subject program.

5. Conclusion

Testing of parallel programs is more difficult than that of sequential programs due to the anomalous effects which occur due to concurrency and interactions between the parallel processes. In parallel programs, states grow exponentially with the number of parallel processes. Therefore, finding the valid sequence of transitions through the set of parallel processes is a demand and challenging task due to the state explosion. In this paper, we proposed an approach based on genetic algorithms and communication-flow analysis for generating the valid sequence of transitions through the set of parallel processes. The approach uses the communication-flow analysis to generate all def-use pairs between the set of parallel processes. Then, the proposed approach applies a genetic algorithm to generate a set of valid sequences of processes transition. This set of valid sequences of transitions is
passed to the feasibility checker to find the executable one. The results of the experiments showed that our proposed approach is more efficient than the random technique in generating the executable transition sequence through the parallel processes. In addition, the results of the experiments showed that the efficiency of the proposed technique in reducing the state explosion compared with the random technique. Our future work will focus on using the proposed technique for finding parallel programming bugs.

6. References


