Paper

Parallel Mutant Execution Techniques in Mutation Testing Process for Simulink Models

Le Thi My Hanh, Nguyen Thanh Binh, and Khuat Thanh Tung

The University of Danang – University of Science and Technology, Vietnam

https://doi.org/10.26636/jtit.2017.113617

Abstract—Mutation testing – a fault-based technique for software testing – is a computationally expensive approach. One of the powerful methods to improve the performance of mutation without reducing effectiveness is to employ parallel processing, where mutants and tests are executed in parallel. This approach reduces the total time needed to accomplish the mutation analysis. This paper proposes three strategies for parallel execution of mutants on multicore machines using the Parallel Computing Toolbox (PCT) with the Matlab Distributed Computing Server. It aims to demonstrate that the computationally intensive software testing schemes, such as mutation, can be facilitated by using parallel processing. The experiments were carried out on eight different Simulink models. The results represented the efficiency of the proposed approaches in terms of execution time during the testing process.

Keywords—mutant execution, mutation testing, parallel processing, software testing.

1. Introduction

Software testing is an expensive process. It typically consumes more than half of the development budget [1], but it is an effective way to estimate the reliability of software. With the increasing expectations for software quality, developers are required to perform more effective testing on large and complex systems.

In this context, mutation testing has been used as a fault injection technique to measure the adequacy of test cases. This method adopts a "fault simulation mode". It has been advocated as a technique for generating test cases by inserting faults into an original program, and the effectiveness of a test suite is represented by its "mutation score". Thus, mutation testing is used to measure the robustness of a test suite. Though powerful, mutation testing is computationally intensive, as numerous mutants need to be produced and executed.

When a mutation is introduced to a large application, a huge number of mutants can be generated. Despite the existing techniques to reduce the costs of mutation analysis, the computational time required to apply mutation testing to large applications is still very long. The costs of mutation testing depend mainly on the number of mutants generated, as well as on the number of test cases. In a single sequential process, the total computational time of mutation testing includes the time spent generating the mutants and the time devoted to executing the tests against all the mutants and the original system (which must be executed at least once). The execution time E_t is always much higher than the generation time G_t. This is the reason why researchers have focused their efforts on reducing E_t. For example, mutant schemata [2] make it possible to speed up execution by including all mutants in a single file. This makes it possible to avoid continuous uploading of mutant files into memory, and thus launching a new process to execute each program version. Techniques such as random selection of mutants [3], selective or constrained mutation [4]–[7], and higher-order mutation [8] produce fewer mutants, which exerts a positive influence on the total execution time. Techniques such as byte-code translation [9], which remove the compilation-related tasks, may in turn reduce the time of processing.

In addition to the described techniques, parallel execution attempts to decrease the overall time by distributing the execution across different processors. This is to improve the performance of mutation testing without compromising the effectiveness of the process in which mutants and tests are executed on parallel processors. This method contributes to reducing the total time needed to perform the mutation analysis.

The size and complexity of the system under test determine the execution expense. The larger or the more complex a given application is, the more test cases are required to achieve the adequate coverage. It takes less time, hence, to generate mutants, but the execution time grows exponentially along with the number of mutants and test cases which, in turn, depends on the size of the application.

This paper presents a study of the parallel mutant execution technique, which is appropriate to reduce the computational cost of the execution phase. Three distribution strategies are proposed to parallelize this task.

The rest of this paper is organized as follows: Section 2 describes some related work on parallel mutation testing. Section 3 briefly introduces the Simulink environment, mutation testing for Simulink models, and a process of mutant generation and execution. In Section 4, three distribution algorithms are proposed to parallelize the execution phase. The experimental results are discussed in Section 5 and,

finally, Section 6 presents the conclusions and future work required.

2. Related Work

In the surveys on mutation testing, such as mutation testing cost reduction techniques, Jia and Harman [10], Mateo and Usaola [11] identified three directions of investigation: execution of mutants (i) in single instruction, multiple data machines (SIMD), (ii) in multiple instruction, multiple data (MIMD) machines, and (iii) with optimized serial algorithms.

Mathur and Krauser [12] were the first ones who performed mutation testing on a vector processor system. They suggested that vectorizable programs be created, each one incorporating several mutants of the same type. The authors hoped that a vector processor could execute the unified mutant programs and achieve a significant speed-up over a scalar processor. The proposed strategy had not been implemented yet, and the authors implied in their papers that only scalar variable replacement (SVR) type mutants are suitable for unification. A later paper by Krauser, Mathur, and Rego [13] proposed an approach for the concurrent mutant execution under SIMD machines. The authors suggested a strategy for efficient execution of mutants. Mutants of the same type are grouped together, and the groups are handled by different processors in a SIMD system. This strategy, however, has also not been implemented yet.

The second direction of research is based on MIMD machines. The work of Choi and Mathur [14] was the first study about the parallel mutant execution on these machines. These authors presented the Pmothra tool that is an adaptation of the Mothra tool [15] for the Ncube/7 Hypercube machine. It was based on a mutant generator, a mutant compiler, a mutant scheduler, several test case servers, and mutant executors. This tool used a dynamic distribution algorithm that executes a mutant when a node of the hypercube becomes available. Another important work was also proposed by Offutt *et al.* [16]. These authors presented the HyperMothra tool, an adaptation of the Mothra tool [13], to be executed in the Intel iPSC/2 hypercube machine with 16 processors. The HyperMothra tool was designed to generate mutants for Fortran systems with 22 mutation operators and to interpret them in parallel in the hypercube.

The last direction of research in the field of parallel mutation testing concerns optimized serial algorithms. Fleyshgakker and Weiss [17], [18] proposed some algorithms to reduce the number of executions and improve the efficiency of the mutation testing process. Although the algorithms were not implemented for parallel execution, the authors indicated that their structure made them easily parallelizable.

In the related works described above, all the mutation studies used programs written in Fortran. In recent years, programming languages, networks, and processors have evolved a great deal. Therefore, recent studies concerned with parallel mutation have adapted the existing cost reduction techniques to new programming languages such as Java. Mateo and Usaola [19] introduced Bacterio-P, which is a parallel extension of the mutation testing tool Bacterio [20] using Java-RMI for communicating among the nodes of the network. In addition, the authors presented five distribution schemes adopting dynamic and static distributions. Among these ones, the parallel execution with the dynamic ranking and ordering algorithm, which is a dynamic distribution algorithm based on factoring self-schedulling ideas [21], gave the best results. However, the mechanisms used in the communications are not the most adequate for high performance environments since a high degree of latency is introduced by this technology, and Java-RMI is much slower than MPI. To cope with this problem, Pablo *et al.* [22] proposed a dynamic distributed algorithm, known as EMINENT, to reduce the execution time associated with the classical mutation testing scheme. Their approach was implemented using the standard Message-Passing Interface (MPI) library to facilitate communications in high-performance environments. In another research, Saleh and Nagi [23] proposed the Hadoop Mutator framework, which is based on the MapReduce programming model to distribute and execute the mutant generation and the testing processes. Nonetheless, this approach follows a static schema, so it is not suitable for heterogeneous and dynamic environments.

As one may see, most studies on parallel mutation testing are applied to programming languages. To the best of our knowledge, this is the first work on parallel mutation testing concerning designs in the Simulink environment.

3. Mutation Testing for Simulink Models

Simulink [24] is a block diagram environment for multi-domain simulation and model-based design. It supports simulation, automatic code generation, continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries and solvers for modeling and simulating dynamic systems. It is integrated with Matlab software, enabling to incorporate Matlab algorithms into models and export simulation results to Matlab for further analysis.

Simulink has been popularly used as a high-level system prototype or a design tool in many domains, including aerospace, automobile, and electronic industries.

Simulink models are of the data-flow variety and consist of three levels of granularity: whole models, subsystems, and blocks. Models contain systems, and systems contain other subsystems and blocks. Blocks originate from predefined block libraries (covering generic functions such as addition or logical operators, but also domains like fuzzy logic or network communication). Blocks are connected by lines that provide a mechanism to transfer signals across the connections, and have their own semantics. Blocks receive a specific number of input signals from which output signals are computed. The underlying internal representation of Simulink models is stored as text either in a Simulink

MDL file or an XML file in the case of newer versions of Simulink.
Simulink plays an increasingly important role in system engineering, while verification and validation of Simulink models are becoming ever more vital to users [25].
This paper is concerned with mutation testing for Simulink models that contain basic blocks in predefined libraries, such as commonly used blocks, continuous, discrete, logic and relational operations, math operations, sinks and sources.

3.1. Mutation Testing for Simulink Models

Mutation testing, which is a fault-based testing technique proposed by DeMillo *et al*. [26], focuses on measuring the quality of a test set according to its ability to detect specific faults. It works in the following way: a large number of simple faults is introduced to a program (or a model), one at a time. The resulting changed versions of the program (or model) under test are called mutants. Test data are then constructed to cause these mutants to fail. The effectiveness of the test set is measured by the percentage of the number of mutants killed over the total number of mutants. Since the number of mutants that can be generated is very large (the number is usually in the order of N^2, where N is the number of variable references in the program), many methods have been suggested to reduce the computational expenses of this testing technique.
In mutation testing, faults are introduced to the program (or model) under test by using mutation operators. They are well-defined rules to make syntactic changes in the original program (or model). They are designed based on the experience of the target language usage and the most common faults. Mutation testing is usually applied to programming languages such as C++, Java, and C#. In this study, we apply mutation testing to Simulink designs, using a set of mutation operators proposed in [27] and shown in Table 1.

Table 1
Mutation operators

Operator	Description
VNO	Variable negation operator
VCO	Variable change operator
TRO	Type replacement operator
CCO	Constant change operator
CRO	Constant replacement operator
SCO	Statement change operator
SSO	Statement swap operator
DCO	Delay change operator
ROR	Relational operator replacement
AOR	Arithmetic operator replacement
ASR	Arithmetic sign replacement
LOR	Logical operator replacement
BRO	Block removal operator
SRO	Subsystem replacement operator

By applying a mutation operator to the model under test, i.e. by inserting a single fault into the model, a faulty model is obtained, which is called a mutant. Then, test data should be generated to reveal the fault introduced.
A test suite is considered good if it contains tests that are able to distinguish a large number of these mutants from the original model. If a mutant can be distinguished from the original model by at least one of the test cases in the test set, the mutant is considered to be killed. Otherwise, the mutant is alive. Sometimes the mutants cannot be killed due to the semantic equivalence of the mutants and the original model. These mutants are called equivalent mutants. Worse still, determining whether a mutant is equivalent is generally undecidable [28], and so typically the decision is left for testers to establish manually.
The proportion between mutants killed and all non-equivalent mutants is called the mutation score and is formally defined as:

$$MS(P,Ts) = \frac{K}{T-E}, \tag{1}$$

where P is the program under test, Ts is the test suite, K is the number of mutants that have been killed, T is the total number of mutants, and E is the number of equivalent mutants.
The process of mutation testing for Simulink models consists in generating mutants, executing mutants, analyzing results and in the generation of test suites. If this process is performed manually, it will require too much time. Hence, we have designed and implemented a *MuSimulink* tool to automate this process. The design details of this tool are presented in [29].

3.2. Mutant Generation and Execution

This section presents, in detail, the process of generating and executing mutants for Simulink models. As with any automated mutation testing system, there are several important steps that a tester must follow. Because there is a large number of mutants generated for each model, it has been considered impractical to compile and store each mutant model separately. Therefore, MuSimulink has been built as an interpretive system. Instead of creating, compiling, and storing many separate models, the Simulink model is translated once into an intermediate form, and each mutant is stored in the form of a short description of the changes required to create the mutants. In MuSimulink, these descriptions are stored in records of a mutant description table (MDT). The testing process continues until a satisfactory mutation score is attained or is forced to stop due to time or economic constraints. The major steps of mutation testing are listed below and illustrated in Fig. 1.

1. **Mutant Generator**. The mutation testing process begins with the construction of mutants, which are automatically created through mutation operators. First of all, the original model O is submitted, analyzed, and parsed to create an intermediate form

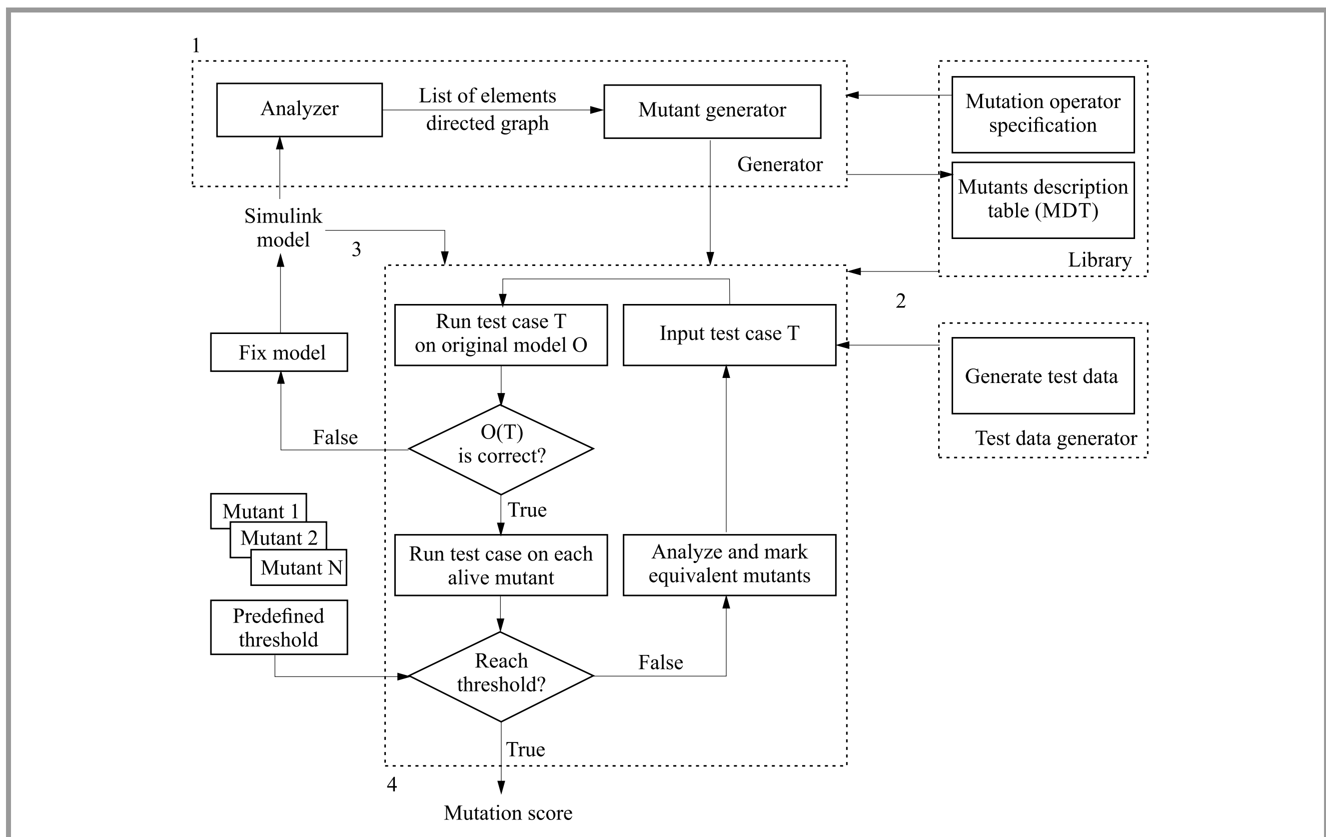

Fig. 1. MuSimulink mutation testing process.

ready for interpretation. One or more mutation operators will be selected to be applied to O. The testers typically use all mutation operators available. Based on the mutation operator specification, each of the selected operators is applied to O to produce the MDT that describes the M set of O mutants.

2. **Test data generator**. A set of test cases T is submitted. Each test case within T contains values for the input variables of O. T can be created manually by testers or generated automatically by the MuSimulink test data generator.
3. **Mutation analysis**. This task is to execute the model under test and its mutants against test cases. The goal is to determine how many mutants are killed by the tests. The results are then analyzed, and the mutation score is calculated to measure the mutation adequacy of the test suite.

The original model O is interpreted once for each test case in T by the MuSimulink tool. A set of expected outputs $O(T)$ is produced. The expected outputs of O can be examined at any point of time during the testing process to determine whether O is performed on T correctly. If any output is incorrect, a fault has been found, and the model must be fixed before the process restarts from step 1. If the output is correct, that test case is executed against each alive mutant within the set of mutants M. M is interpreted by MuSimulink using the input values for each test case in T. This produces a set of mutant outputs $M(T)$ that has at most $|M| \cdot |T|$ elements. Since mutants are not executed against new test cases after being killed, this set of mutant outputs will not be usually very large. In practice, T is usually small compared to M. Note that a typical test case will kill a large number of mutants, so the number of executions is usually much lower than $|M| \cdot |T|$. After that, each element of $M(T)$ is compared with the element of $O(T)$ generated from the same test data. If they are not the same, the mutant is killed. If, after output comparison, some mutants remain alive, either the test data in T are not adequate or the mutant is equivalent to O, and it can never be killed.

The results of testing are analyzed and, if necessary, further testing may be performed. If all mutants are killed (or reach the specified mutation score threshold), and all mutation operators have been applied to the original model O, then no further testing is necessary. If one or more non-equivalent mutants remain alive, then additional test cases should be added to T, and appropriate steps of the process should be repeated. If, as a result of testing, faults in O are uncovered, then O must be modified, and the testing process will be repeated as well. Existing test cases can usually be reused for the subsequent testing phases.

After all test cases have been executed against all mutants generated, each remaining mutant falls into one of two categories. The first one is composed of mutants that are functionally equivalent to the original model. The equiva-

lent mutant always yields the same output as the original model, so no test case can kill it. The second category consists of mutants that are killable, but the test set is insufficient to kill them. In this case, new test cases need to be added, and the process reiterates until the test set is strong enough to kill all mutants or until the specified mutation score threshold can be reached.

Steps are presented above to expose the inherent parallelism in mutation testing. The most computationally expensive parts of the mutation process are the execution of the original model, the mutant execution, the output comparison, and the test data generation. The mutant execution, which is performed once for each mutant and each test case, is considered to be an internal loop. The internal loop includes the interpretation of a mutant on a test case, the comparison of the mutant output with the output of the original model, and, if they differ, the killing of the mutant. The internal loop (shown in lines 11–23 of Algorithm 1, lines 14–29 of Algorithm 2, lines 16–31 of Algorithm 3) is by far the most computationally expensive part of mutation testing, and therefore it is the target of our parallelization efforts.

Algorithm 1. Mutant distribution strategy using the parfor paradigm

```
1: Read test data file T
2: Start N workers in Matlab
3: for each test case t in T do
4:     Execute original model O on t to produce Δ(O,t)
5:     if (abnormal termination O) then
6:         Send "error" message, close workers and finish
7:     else
8:         Record expected output Δ(O,t)
9:     end if
10:    Send t and expected output Δ(O,t) to workers
11:    for parallel each alive mutant m in M do
12:        Send the mutant information m to the worker
13:        Modify original model O to produce mutant model O′
14:        Interpret O′ on t to produce Δ(O′,t)
15:        if abnormal termination O′ then
16:            Mark m as killed
17:        else if Δ(O′,t) ≠ Δ(O,t) then
18:            Mark m as killed
19:        else
20:            Mutant m remains alive
21:        end if
22:        Update the killed mutant counter
23:    end for parallel
24: end for
25: Close all workers
26: Write execution outputs to a result file
```

Algorithm 2. The alternate-order mutant distribution strategy using Matlab's SPMD paradigm

```
1: Read test data file T
2: Start N workers in Matlab
3: for each test case t in T do
4:     Execute original model O on t to produce Δ(O,t)
5:     if (abnormal termination O) then
6:         Send "error" message, close workers and finish
7:     else
8:         Record expected output Δ(O,t)
9:     end if
10:    Send t and expected output Δ(O,t) to workers
11:    spmd in M
12:        k ← 0
13:        i ← N · k + labindex
14:        while i ≤ |M| do
15:            if mutant M(i) alive then
16:                Send infor. of mutant M(i) to worker labindex
17:                Modify original model O to produce mutant O′
18:                Interpret O′ on t to produce Δ(O′,t)
19:                if abnormal termination O′ then
20:                    Mark M(i) as killed
21:                else if Δ(O′,t) ≠ Δ(O,t) then
22:                    Mark M(i) as killed
23:                else
24:                    Mutant M(i) remains alive
25:                end if
26:            end if
27:            k ← k + 1
28:            i ← N · k + labindex
29:        end while
30:        Update the killed mutant counter
31:    end spmd
32: end for
33: Close all workers
34: Write execution outputs to a result file
```

4. Solutions Parallel to Mutant Execution

In a survey of parallel Matlab technologies [30], nearly 27 technologies were discovered. Many of them are defunct, while many others are currently under development, with a large user base and an active developer base. This paper uses the parallel computing toolbox (PCT) with the Matlab distributed computing server (MDCS), which is a novel technology.

While the core Matlab software itself supports multithreading, the PCT offers operations to run the Matlab code on multicore systems and clusters. The PCT provides functions for the parallel for-loop execution, creation/manipulation of distributed arrays, as well as message passing functions for implementing fine-grained parallel algorithms.

The MDCS enables to scale parallel algorithms to larger cluster sizes. The MDCS consists of the Matlab worker processes that run on the cluster and is responsible for parallel code execution and process control. Figure 2 illustrates the PCT and MDCS architecture.

The PCT also allows users to run up to 12 Matlab labs or workers on a single machine. This enables interactive development and debugging of parallel codes from a desktop. After parallel codes have been developed, they can be scaled up too much larger number of workers or labs in conjunction with the MDCS. Thus, the PCT addresses

Algorithm 3. The random-order mutant distribution strategy using Matlab's SPMD paradigm

```
1: Read test data file T
2: Start N workers in Matlab
3: for each test case t in T do
4:     Execute original model O on t to produce Δ(O,t)
5:     if (abnormal termination O) then
6:         Send "error" message, close workers and finish
7:     else
8:         Record expected output Δ(O,t)
9:     end if
10:    Send test case t and expected output Δ(O,t) to workers
11:    Get a list of alive mutants ℒ from M
12:    Randomly reorder list ℒ
13:    spmd in ℒ
14:        k ← 0
15:        i ← N·k + labindex
16:        while i ≤ |ℒ| do
17:            if mutant ℒ(i) alive then
18:                Send infor. of mutant ℒ(i) to worker labindex
19:                Modify original model O to produce mutant O′
20:                Interpret O′ on t to produce Δ(O′,t)
21:                if abnormal termination O′ then
22:                    Mark ℒ(i) as killed
23:                else if Δ(O′,t) ≠ Δ(O,t) then
24:                    Mark ℒ(i) as killed
25:                else
26:                    Mutant ℒ(i) remains alive
27:                end if
28:            end if
29:            k ← k+1
30:            i ← N·k + labindex
31:        end while
32:        Update killed mutants on M
33:    end spmd
34: end for
35: Close all workers
36: Write execution outputs to a result file
```

the challenge of getting codes to work well in a multicore system by enabling to select the programming paradigm that is most suitable for applications. The paper employs two most basic parts of these paradigms: parallel for-loops and Single Program Multiple Data (SPMD) blocks.

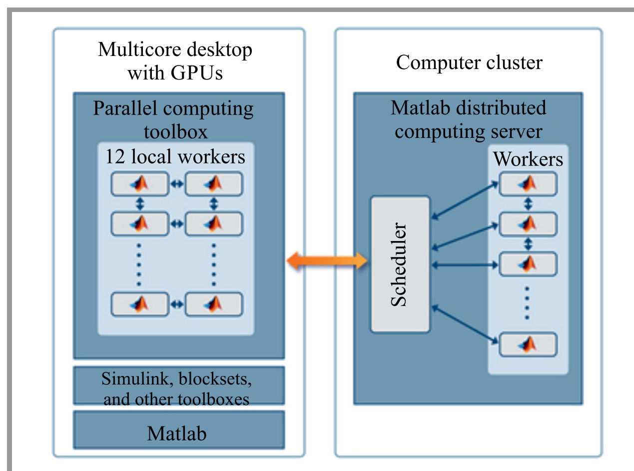

Fig. 2. The parallel computing toolbox and the Matlab distributed computing server.

4.1. Work Distribution Strategies

Parallel algorithms can be divided into two categories: task-parallel and data-parallel. Task-parallel algorithms take advantage of the fact that multiple processors can work on the same problem without communicating with each other. These algorithms can be used when the computations in a large loop are independent from each other and can be performed in any order without affecting the results. In such cases, multiple processors can analyze the subsets of the data simultaneously, without the need for inter-processor communication. Data-parallel algorithms typically involve some inter-processor communication. In such algorithms, the data are typically too large to be analyzed on a single processor. Therefore, parallel computing paradigms are used to distribute the data across processors, and each processor works on a smaller chunk of the same data. In such cases, there may be some communications required between different processors that involve the exchange of data to address boundary conditions. Based on how the data are distributed, each processor needs a small amount of data from its neighbor to complete the computations.

Since the size of T (the set of test cases) is usually small compared to the size of M (the set of mutants), our work uses the task-parallel approach. Parallelizing the mutant execution on multi-processor machines has been implemented by supplying each worker/lab with all test cases and a subset of mutants. The mutation testing using the parallel mechanism for mutants is a natural way to divide up the work because it does not necessarily guarantee an even distribution of work among workers/labs. Some mutants which are easily killed by a test case in the early stage of the mutant execution process will not be executed against most of the remaining test cases in the test set. On the other hand, equivalent mutants must be executed against all test cases, since they are not killed by any test case. Therefore, there is a wide volatility in the amount of execution time required for individual mutants. To achieve maximal speed-up, we should distribute mutants to workers/labs such that each worker/lab performs the same amount of execution. It is unfortunate that we have no way to know in advance how much execution time will be required for a mutant, or how many test cases need to be run against it. The optimum distribution of mutants, thus, cannot be determined.

The parallel mutation algorithm has two execution phases: original model execution and mutant execution. In the original model execution phase, workers/labs are not used, and only the host processor (Matlab client) computes and saves the expected outputs from the original model. In the mutant mode, the client begins with sending the startup information to the workers. For each test case, the input values and the expected outputs are sent to the workers, then once a worker interprets all its mutants on the test case, a counter of remaining alive mutants is sent back. If all the mutants within all the workers are killed (or mutation score reaches

a predetermined threshold value), then the algorithm takes an early exit and does not send more test cases to the workers. Otherwise, the next test case is sent. When all the test cases have been processed, the workers send the updated MDTs back to the client. There is recurring communication between the client and the workers. The client sends the test case information which includes the outputs of the original model and a list of elements referenced in the original model, and the workers send back the number of alive mutants.

Three dynamic distribution strategies are described below that attempt to balance the amount of work done by the workers. The distribution strategies are shown in Algorithms 1–3, where test data file T and mutant description table M are the inputs, while the output is the *Alive/Killed* list of mutants. In all algorithms, test data are delivered to workers sequentially by an outer loop (line 3 for all), while the inner iterations (line 11 in Algorithm 1, line 14 in Algorithm 2, and line 16 in Algorithm 3) take responsibility for executing mutants in parallel.

Algorithm 1 presents the first mutant distribution strategy using the Matlab parfor paradigm which divides the MDT list between the workers. Alive mutants are interpreted on each test case by the workers, and the process is referred to as internal loop iteration (lines 12–22 in Algorithm 1). The parfor statement is very simple to use due to the fact that it is based on the automatic data management, but the parfor iterations are executed in an unknown order. Thus, the effective distribution of mutants cannot be determined.

Algorithms 2 and 3 present two dynamic distribution strategies using Matlab's SPMD paradigm. For each test case, the first worker gets the first mutant in the MDT list, the second worker gets the next mutant, and so on. Dead mutants in the MDT list are not distributed to the workers. If there are N workers, the first N mutants will be assigned to the workers at a time. Then, the second N mutants will be assigned to the workers in the next iteration, and so on. Thus, if *labindex* is the index number of each worker, worker *labindex* gets mutants at position $N \cdot k + labindex$ in the MDT list (where k is from 0 to $|M|/N$ in turn, $|M|$ is the number of mutants). These two strategies distribute approximately the same number of mutants to each worker.

The second strategy, in Algorithm 2, is the distribution of MDTs in an alternate order of a list of all mutants. However, not all executions will take the same time because each mutant usually takes a different amount of time to run, so presumably, some processors will finish before others, and the total process time will be the time taken by the slowest processor. To avoid a large number of "hard to kill" mutants running on a worker, the third distribution strategy assigns each worker approximately the same number of mutants in a random order of the list of alive mutants. For each new test case, the MDT list will first be randomly reordered before alive mutants are delivered to the workers, and the parallel process is performed on this randomly reordered list (lines 11–12 of Algorithm 3).

5. Experimentation

5.1. Parallel Mutation Testing on a Single Multicore Machine

Three proposed distribution strategies were implemented in the MuSimulink tool [29], and experiments in this subsection were carried out on a single computer which uses the Intel Xeon E5520 2.27 GHz CPU with 8 GB RAM, and runs the Windows Server 2008 operating system. This computer has two processors with four cores each. Thus, to use all the cores, eight workers were run. For the distribution algorithms, a configuration parameter which needs to be established is the number of workers. In experiments, the MDT is generated for the models using mutation operators introduced in our previous work [27]. Each model under test was given 100 test cases generated randomly using MuSimulink's automatic test data generator. The original model is executed on the client, while the mutant ones are executed in parallel on eight workers using the different work distribution strategies (using parfor paradigm, alternate-order using SPMD, and random-order using SPMD).

The time of the execution phase within the mutation process is shown in Table 2, which describes the average time of ten runs per second for each model with regard to each distribution algorithm. The use of parallel strategies helps us reduce the execution time by up to 89.23% (from 4752.54 s down to 511.9 s).

Figure 3 is a chart that shows the speed-up achieved by MuSimulink using eight workers. Speedup for N workers is defined as the division between the serial execution time on one worker and the parallel execution time on N workers. *Speed-up* indicates by how much the execution time has been reduced. It may be seen from both Fig. 2 and Table 2 that the work distribution strategy 1 (using the

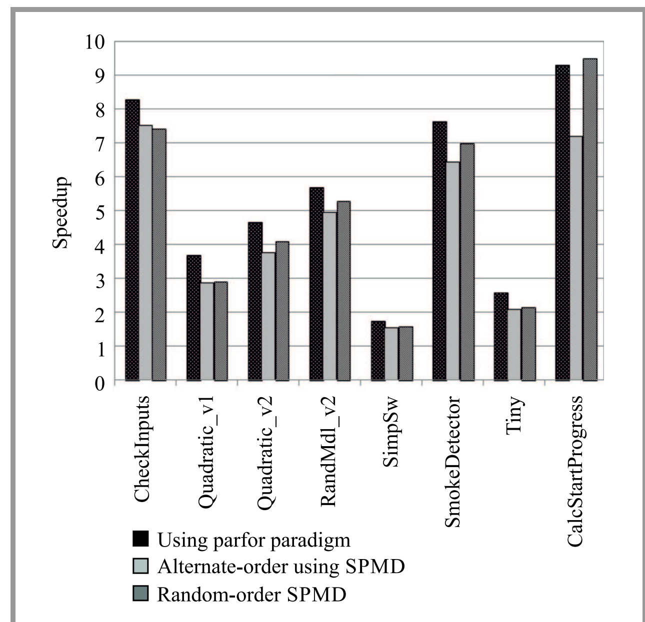

Fig. 3. *Speed-up* achieved using eight workers.

Table 2
The total execution time for each model with each distribution strategy

Model name	Mutant	Killed	MS	Time [s]			
				Serial	Using parfor paradigm	Alternate-order using SPMD	Random-order using SPMD
CheckInputs	154	130	83.77	2303.20	278.5	306.6	311.04
Quadratic_v1	161	129	90.43	636.94	173.2	221.37	220.18
Quadratic_v2	140	89	63.57	976.52	210.7	259.62	240.32
RandMdl_v2	188	138	73.40	1141.18	201.2	230.07	217.22
SimpSw	92	85	92.39	263.16	153.2	170.69	167.56
SmokeDetector	321	160	49.84	2685.83	353.3	418.08	385.11
Tiny	144	120	83.33	490.74	190.3	234.63	229.13
CalcStartProgress	458	183	39.96	4752.54	511.9	661.69	501.13

parfor paradigm) results in a much better speed-up than strategy 2 (alternate-order using SPMD), and strategy 3 (random-order using SPMD) is also marginally better than strategy 2 in general.

In the second distribution strategy, each worker receives the mutants located once at the fixed position in the list of mutants, before executing. Therefore, it is more likely a certain worker will always be available if all its mutants are killed soon. This situation causes an imbalance among workers in which some cores are overworked with hard-to-kill mutants, and so the second strategy has rendered the worst performance in comparison with the others. This drawback is partially overcome in the third strategy due to the random redistribution of alive mutants before delivering them to workers to execute with new test case. Hence, the execution time of the third strategy is shorter than that of the second one.

The second and third strategies can reduce the communication between workers and host, and the network traffic will be minimal because each worker receives fixed groups once, before starting the process of executing mutants for each test case. However, mutant model interpretation and execution times are much longer than the communication time. Meanwhile, the second and third algorithms use fixed groups of mutants, so not all executions will take the same amount of time. In such a case, some workers will finish before others and become available when they get many easy-to-kill mutants. This is the disadvantage of strategies 2 and 3 using the SPDM mechanism, while strategy 1 using parfor does not face this problem. In the first distribution strategy, the executions are split into smaller pieces, called tasks. The tasks are delivered to parallel workers several times until all the groups are delivered. In other words, the tasks are sent to the parallel workers on demand. The parfor mechanism has a smart scheduler that sends tasks to the free workers. When a worker finishes its executions, it receives a new task with its size depending on the number of remaining executions in the mutation process [31]. Hence, when the remaining unexecuted mutants are few, the task size is small. The loop of sending and executing mutants on the test case finishes if all tasks have been sent and run. The smart scheduler contributes to the reduction of the free time of a worker, as well as to the increase in the speed of mutant execution, so execution time of the strategy 1 is significantly reduced, in general terms, compared to that of the second and third strategies. Nonetheless, there are some rare exceptions such as the *CalcStartProgress* model, when the third strategy is better than the first one. As may be seen from the result of this model, the mutation score is low – it means that a lot of alive mutants exist, and these mutants are executed on all test cases in the test set. In this case, it is more likely that the number of hard-to-kill mutants on each worker is quite equal in both strategy 1 and strategy 3. Meanwhile, the first distribution strategy has to spend more time on communication, so the execution time in the strategy 1 is slightly longer.

In practice, a complex Simulink model with few mutants might run longer compared to a simple one with many mutants. This is shown in the *Serial* column of Table 2, where *CheckInputs* model with only 154 mutants shows the serial execution time being more than twice as long as that of *RandMdl v2* model with 188 mutants. The same applies also to the parallel mutant execution time. In general, the execution time will increase along with the growth of complexity of the models under test.

It would be more helpful if we could show how the distribution and communication times contribute to the total execution time. However, parallel computing toolbox is a high-level application programming interface (API), so we may only identify the total execution time, and there is no way to get the communication time between the host and workers. In the future work, therefore, we need to use other low-level APIs to ameliorate and further analyze the effect of communication time on the total mutant execution time.

5.2. Parallel Mutation Testing on Many Machines

The first experiment was conducted by parallelizing on only one multicore computer. To prove for the effectiveness of parallel programming with Matlab, we scaled up the distri-

bution strategy using parfor to much larger number of workers in the different environments using the MDCS. This ex- periment was performed in two environments with different characteristics, one of them being a homogeneous system (four computers with the same processor, memory and operating system) and the other a heterogeneous system (two computers with the different processors and memories), which are called DCS_A and DCS_B, respectively. The DCS_A environment is made up of four homogeneous computers (with 2.4 GHz Intel Core 2 Quad CPU Q6600 and 2 GB memory, running the Windows 7), which are configured for running four workers on each computer. The DCS_B environment is made up of two heterogeneous computers, including one computer (CPU Intel Xeon E5520 2.27 GHz and 8 GB RAM) running eight workers and one computer (AMD Operon Dual-Core 2.27 GHz, 4 GB memory, running the Windows Server 2008 operating system) running four workers.

Table 3
The total time in seconds for each model on two different environments using MDCS

Model	DCS_A	DCS_B
CheckInputs	74.99	189.35
Quadratic_v1	119.48	129.51
Quadratic_v2	133.21	148.67
RandMdl_v2	125.76	132.23
SimpSw	112.45	117.59
SmokeDetector	187.88	221.21
Tiny	126.87	173.48
CalcStartProgress	340.67	437.72

A drawback of PCT is that it can only get use of the maximum of 12 workers on a multicore machine. Hence, the purpose of this experiment is to show that parallel mutation testing can be executed on many machines with homogeneous and heterogeneous configurations by running the Matlab Distributed Computing Server. This ability is useful to execute mutation testing for large Simulink models.

As shown in Table 3, the parfor mechanism on the homogeneous combination of many machines (DCS A) with 16 workers takes much less time compared with that using 8 workers on a single multicore machine with the same configuration as shown in Table 2. It is also noted that the parallel execution for small models with a few mutants is less efficient than that for large models, because, in such cases, the communication cost is higher than the execution cost.

The goal of this subsection is too prove that mutation testing can be scaled up to many computers by running it on MDCS. Results in Table 3 have not yet concluded that the shorter execution time for the DCS_A configuration results from its homogeneity, or that it has more workers, as the number of workers in two different environments is not equal. The influence of the configuration type on the mutation execution time is out of scope of this paper, and it will be figured out more carefully in the future studies.

6. Conclusion

Parallel execution helps reduce the computational cost, which is one of the biggest problems in mutation testing, and increase the efficiency without compromising the effectiveness. Three different strategies were implemented by distributing different subsets of mutants to the workers, and an experimental comparison of these three distribution algorithms was conducted. The experimental results demonstrated that the distribution strategy of mutants using the parfor scheme is the best one.

As discussed above, the parallel execution is only useful for large models with a large number of mutants, so mutation testing should be done on sequential machines when models are small and moved to parallel machines only if the size of models requires much execution time.

Another problem is that the communication overhead is fairly high because the client broadcasts one test case at a time to all workers, and some small models do not require much time for mutant interpretation. To decrease the communication cost in all cases, test cases could be sent to the workers in blocks, since few large messages (*n* test cases at a time) are processed more efficiently than many small messages (one test case at a time) on multicore machines. This offers a great potential to improve the performance of MuSimulink in the future work.

Moreover, workers often sit idle waiting for the slowest node to finish executing mutants. If nodes are allowed to request work from the host rather than wait for the host to send the next test case, then the idle time could be significantly reduced. This demand-driven strategy may restrict overhead to the time necessary for communicating test case information.

Finally, mutants are assigned to processors before they are interpreted, and there is no way to redistribute mutants during interpretation if one or more processors become overworked. With dynamic load balancing, mutants can be reassigned during interpretation so that each processor performs approximately the same amount of work. In the future work, the study will be extended to validate this direction.

References

[1] B. Beizer, *Software Testing Techniques*, 2nd ed., Thomason Computer Press, 1990.

[2] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using program schemata", in *Proc. Int. Symp. on Software*, Cambridge, Massachusetts, 1993, pp. 28–30.

[3] K. N. King and A. J. Offutt, "A Fortran language system for mutation based software testing", *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[4] E. F. Barbosa, J. C. Maldonado, A. Marcelo, and R. Vincenzi, "Toward the determination of sufficient mutant operators for C", *Software Test., Verif. and Reliabil.*, vol. 11, no. 2, pp. 13–136, 2001.

[5] A. J. Offutt, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators", *ACM Trans. on Software Engin. and Methodol.*, vol. 5, no. 2, pp. 99–118, 1996.

[6] W. E. Wong and A. P. Mathur, "How strong is constrained mutation in fault deletion", in *Proc. Int. Computer Symp. ICS'94*, Hsinchu, Taiwan, Republic of China, 1994, pp. 515–520.

[7] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur, "Constrained mutation in C programs", in *Proc. 8th Simpósio Brasileiro de Engenharia de Software SBES 94*, Curitiba, PR, Brazil, 1994, pp. 439–452.

[8] M. Polo, M. Piattini, and I. Garcia-Rodriguez, "Decreasing the cost of mutation testing with 2-order mutants", *Softw. Test. Verif. Reliab.*, vol. 19, no. 2, pp. 111–131, 2008.

[9] Y. S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system", *Software Test., Verif. and Reliabil.*, vol. 15, no. 2, pp. 97–133, 2005.

[10] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing", *IEEE Trans. on Software*, vol. 37, no. 5, pp. 649–678, 2011.

[11] P. R. Mateo and M. P. Usaola, "Mutation testing cost reduction techniques: a survey", *IEEE Software*, vol. 27, no. 3, pp. 80–86, 2010.

[12] A. P. Mathur and E. W. Krauser, "Modeling mutation on a vector processor", in *Proc. 10th Int. Conf. on Software Engin. ICSE'88*, Singapore, 1988, pp. 154–161.

[13] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High performance software testing on SIMD machine", *IEEE Trans. on Software Engin.*, vol. 17, no. 5, pp. 403–423, 1991.

[14] B. Choi and A. Mathur, "High-performance mutation testing", *J. of Syst. and Software*, vol. 20, no. 2, pp. 135–152, 1993.

[15] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and J. Offutt, "An extended overview of the Mothra software testing environment", in *Proc. 2nd Worksh. on Softw. Test., Verif. and Anal.*, Banff, Alberta, Canada, 1988, pp. 142–151.

[16] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a MIMD computer", in *Proc. Int. Conf. on Parallel Process. ICPP 1992*, Chicago, Illinois, USA, 1992, pp. 257–266.

[17] V. N. Fleyshgakker and S. N. Weiss, "Efficient mutation analysis: a new approach", in *Proc. Int. Symp. on Software Test. and Anal. ISSTA 1994*, Seattle, WA, USA, 1994, pp. 185–195.

[18] S. N. Weiss and V. N. Fleyshgakker, "Improved serial algorithms for mutation analysis", in *Proc. Int. Symp. on Software Test. and Anal. ISSTA 1993*, Cambridge, MA, USA, 1993, pp. 149–158.

[19] P. R. Mateo and M. P. Usaola, "Parallel mutation testing", *J. of Software Test., Verif. and Reliabil.*, vol. 23, no. 4, pp. 315–350, 2013.

[20] P. R. Mateo and M. P. Usaola,"Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases", in *Proc. of the Int. Conf. on Software Mainten. ICSM 2012*, Trento, Italy, 2012, pp. 646–649.

[21] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops", *J. of Commun. of ACM*, vol. 35, no. 8, pp. 90–101, 1992.

[22] C. C. Pablo, G. M. Mercedes, and N. Alberto, "EMINENT: EMbarrassINgly parallEl mutatioN Testing", *Procedia Comp. Science*, vol. 80, pp. 63–73, 2016.

[23] I. Saleh and K. Nagi, "Hadoopmutator: A cloud-based mutation testing framework", in *14th Int. Conf. on Software Reuse ICSR 2015*, Miami, FL, USA, 2014, pp. 172–187.

[24] Matlab Inc. [Online]. Available: http://www.mathworks.com/products/simulink/ (accessed on March 10, 2017).

[25] K. Ghani, J. A. Clark, and Y. Zhan, "Comparing Algorithms for Search-based Test Data Generation of Matlab Simulink Model, in *Proc. 10th IEEE Congr. on Evol. Comput. CEC'09*, Trondheim, Norway, 2009, pp. 2940–2947.

[26] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: help for practicing for programmer", *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[27] L. T. M. Hanh and N. T. Binh, "Mutation Operators for Simulink Models", in *Proc. of the 4th Int. Conf. on Knowl. and Syst. Engin. KSE 2012*, Danang, Vietnam, 2012, pp. 54–59.

[28] T. A. Budd and D. Angluin, "Two notions of correctness and their relation", *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[29] L. T. M. Hanh and N. T. Binh, "Automatic generation of mutants for simulink models", in *Proc. 16th Nat. Conf.: Selec. Problems About IT and Telecommun.*, Danang, Vietnam, 2013, pp. 339–346.

[30] A. Krishnamurthy, S. Samsi, and V. Gadepally, "Parallel Matlab techniques", in *Image Processing*, Y.-S. Chen, Ed. InTech, 2009 [Online]. Available: http://www.intechopen.com/books/image-processing/parallel-matalab-techniques

[31] G. Sharma and J. Martin, "MATLAB: A language for parallel computing", *Int. J. of Parallel Programm.*, vol. 37, no. 1, pp. 3–36, 2009.

Le Thi My Hanh is currently a lecturer at the Information Technology Faculty, University of Science and Technology, Danang, Vietnam. She gained her M.Sc. degree in 2004 and the Ph.D. degree in Computer Science at the University of Danang in 2016. Her research interests are about software testing and, more generally, application of heuristic techniques to problems in software engineering.
Email: ltmhanh@dut.udn.vn
The University of Danang
University of Science and Technology
54 Nguyen Luong Bang, Lien Chieu
Danang, Vietnam

Nguyen Thanh Binh graduated from The University of Danang, University of Science and Technology in 1997. He got a Ph.D. degree in Computer Science from Grenoble Institute of Technology (France) in 2004. He is currently associate professor of the Information Technology Faculty, The University of Danang, University of Science and Technology, Vietnam. He has been dean of Information Technology Faculty at The University of Danang, University of Science and Technology since 2010. He has been directing a research team since 2009. His research interests include software testability, software testing and software quality.
Email: ntbinh@dut.udn.vn
The University of Danang
University of Science and Technology
54 Nguyen Luong Bang, Lien Chieu
Danang, Vietnam

Khuat Thanh Tung received the B.Sc. degree in Software Engineering from University of Science and Technology, Danang, Vietnam, in 2014. Currently, he is participating in the research team at DATIC Laboratory, University of Science and Technology, Danang. His research interests focus on software engineering, software testing, evolutionary computation, intelligent optimization techniques and applications in software engineering.

Email: thanhtung09t2@gmail.com
The University of Danang
University of Science and Technology
54 Nguyen Luong Bang, Lien Chieu
Danang, Vietnam