AUS Repository

Parallel Implementations for Eliminating Finite State Machine Mutants

Item Type	Thesis
Authors	Badawi, Emad Mohammad
Download date	2026-05-20 17:29:04
Link to Item	http://hdl.handle.net/11073/8866

PARALLEL IMPLEMENTATIONS FOR ELIMINATING FINITE STATE MACHINE MUTANTS

by

Emad Mohammad Badawi

A Thesis presented to the Faculty of the
American University of Sharjah
College of Engineering
In Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in
Computer Engineering

Sharjah, United Arab Emirates

May 2017

© 2017 Emad Mohammad Badawi. All rights reserved.

Approval Signatures

We, the undersigned, approve the Master's Thesis of Emad Mohammad Badawi.

Thesis Title: Parallel Implementations for Eliminating Finite State Machine Mutants.

Signature **Date of Signature**
(dd/mm/yyyy)

_____________________________ _______________
Dr. Khaled El-Fakih
Associate Professor, Department of Computer Science and Engineering
Thesis Advisor

_____________________________ _______________
Dr. Gerassimos Barlas
Professor, Department of Computer Science and Engineering
Thesis Co-Advisor

_____________________________ _______________
Dr. Raafat Aburukba
Assistant Professor, Department of Computer Science and Engineering
Thesis Committee Member

_____________________________ _______________
Dr. Usman Tariq
Assistant Professor, Department of Electrical Engineering
Thesis Committee Member

_____________________________ _______________
Dr. Fadi Aloul
Head, Department of Computer Science and Engineering

_____________________________ _______________
Dr. Mohamed El-Tarhuni
Associate Dean for Graduate Affairs and Research
College of Engineering

_____________________________ _______________
Dr. Richard Schoephoerster
Dean, College of Engineering

_____________________________ _______________
Dr. Khaled Assaleh
Interim Vice Provost for Research and Graduate Studies

Acknowledgements

I would like to express my utmost gratitude to my thesis advisors, Dr. Khaled El-Fakih, and Dr. Gerassimos Barlas, for their guidance and patience at every step throughout the thesis. I would also like to thank them for sharing their knowledge with me.

I am grateful to the Department of Computer Engineering and the American University of Sharjah for giving me the opportunity to pursue my graduate studies by offering me Graduate Teaching Assistantship.

In the end, I would like to thank my parents, brothers, uncle Dr. Imad Abu-yousef and friends, for their love, encouragement, and sacrifices. Without their support, I wouldn't have been able to carry out my work.

Abstract

In this thesis, the mutants' elimination problem considered in finite state machine (FSM) based mutation testing, fault diagnosis, and in the assessment of the effectiveness of test suites is targeted. Given a test suite of some test cases usually derived from a specification FSM and a set of mutants (or fault domain), derived from the specification with respect to some assumed types of faults, mutants' elimination deals with deleting/killing each mutant of the fault domain that has an output behavior different than that of the specification FSM in respect to some test case of the test suite. However, this process is time consuming, especially when the number of considered mutants is huge. Accordingly, three parallel implementations for the considered problem based on the Open Multi-Processing (OpenMP), Message Passing interface (MPI) and the Compute Unified Device Architecture (CUDA) parallel technologies are presented. Comprehensive experiments are conducted to assess the speedup and execution time of the proposed implementations. On average, over all conducted experiments with both randomly generated and real application FSMs, the speedup of OpenMp, MPI, and GPU against sequential implementation equals 6.4, 22.9, and 569.7 times, respectively. The relative speedup of MPI and CUDA with respect to OpenMp equals 3.5 and 121.5 times, respectively; and the relative speedup of CUDA with respect to MPI equals 96.12 times. In addition, the results obtained using real machines are compared with random machines with the same attributes. CUDA implementation is shown to be scalable in terms of considered number of mutants and FSM size. For instance, limited by the used hardware architecture, CUDA easily handled experiments with 500 Million mutants and operated on machines with 9.5 Million transitions. Experiments are also conducted to determine the experimental setup attributes such as test suite length, number of test cases, and attributes related to the parallel implementations such as threads number in OpenMP, processes number in MPI and number of inputs of a test case that will be applied to the mutants in each GPU invocation.

Search Terms: ***Model Based Testing, Mutation Testing, Parallel Testing, MPI, OpenMP, GPU, CUDA.***

Table of Contents

Abstract 5

List of Figures 8

List of Tables 9

List of Abbreviations 10

Chapter 1 : Introduction 11

Chapter 2 : Preliminaries 16

2.1 Finite State Machine (FSM) 16

2.2 Open Multi Processing (OpenMP) 18

2.3 Message Passing Interface (MPI) 18

2.4 Graphical Processing Units (GPUs) 19

2.4.1 Compute Unified Device Architecture (CUDA). 20

Chapter 3 : Proposed Implementations 24

3.1 Sequential Algorithm 24

3.2 OpenMP Implementation 26

3.3 MPI Implementation 28

3.4 CUDA Implementation 30

Chapter 4 : Experimental Results 35

4.1 Used FSMs, Software and Hardware platforms 35

4.1.1 Determining the TS length. 37

4.1.2 Determining the number of test cases in the TS. 38

4.2 Experiments with Randomly Generated FSMs 39

4.2.1 Sequential versus parallel implementations. 39

4.2.2 Speedup of MPI and CUDA relative to OpenMP 41

4.2.3 Speedup of CUDA relative to MPI. 42

4.2.4 CUDA Scalability. 44

4.3 Experiments with Real Application FSMs 44

4.3.1 Sequential versus parallel implementations. 44

4.3.2 Speedup of MPI and CUDA relative to OpenMP. 46

4.3.3 Speedup of CUDA relative to MPI. 48

4.3.4 CUDA Scalability. 49

4.4 Analysis 50

4.4.1 Execution time considering machine size. 50

4.4.2 Number of Threads, Processes and Run Size versus Speedup. 51

4.4.3 Randomly generated FSMs versus real applications FSMs 53
Chapter 5 : Related Work and Literature Review...58
Chapter 6 : Conclusion...61
References…………………………………………………………………………64
Vita..70

List of Figures

Figure 2-1 Finite State Machine S 17
Figure 2-2 CUDA Programming Model 23
Figure 3-1 MPI Implementation 29
Figure 3-2 CUDA Execution Time versus Block Size 31
Figure 3-3 Block Size 256 versus Block Size 512 32
Figure 4-1 Sequential execution time versus TS length 37
Figure 4-2 Percentage of killed mutants versus TS length 38
Figure 4-3 Sequential execution time versus number of test cases (FSM S_1) 39
Figure 4-4 Sequential execution time versus number of test cases (FSM S_7) 39
Figure 4-5 Sequential versus parallel implementations execution time (Machine S_1) 40
Figure 4-6 Speedup against sequential algorithm (Machine S_1) 41
Figure 4-7 Execution time for the parallel implementations (Machine S_1 in table 4-1) 41
Figure 4-8 MPI and CUDA speedup against OpenMP (Machine S_1 in table 4-1) 42
Figure 4-9 Execution time for MPI and CUDA parallel implementations (Machine S_7) 43
Figure 4-10 CUDA speedup against MPI (Machine S_7) 43
Figure 4-11 CUDA massive test (randomly generated machines) 44
Figure 4-12 Sequential versus parallel implementations execution time (Machine nucpwr) 45
Figure 4-13 Machine nucpwr speedup analysis 46
Figure 4-14 Parallel implementations execution time (FSM nucpwr in table 4-1) 47
Figure 4-15 MPI and CUDA speedup against OpenMP (FSM nucpwr in table 4-1) 47
Figure 4-16 MPI execution time for big real application FSMs 48
Figure 4-17 CUDA speedup against MPI in big real application FSMs 49
Figure 4-18 CUDA massive test (real FSMs application examples) 50
Figure 4-19 Number of transitions versus execution time (Machines S_1-S_7) 51
Figure 4-20 OpenMP speedup versus number of threads (Machines S_1-S_7) 52
Figure 4-21 MPI speedup versus number of processes (Machines S_1-S_7) 52
Figure 4-22 CUDA speedup versus Run size (Machines S_1-S_7) 53
Figure 4-23 Sequential execution time of real applications FSMs versus. raandom FSMs with the same attributes 54
Figure 4-24 OpenMp speedup of real applications FSMs versus random FSMs with the same attributes 55
Figure 4-25 MPI speedup of real applications FSMs versus random FSMs with the same attributes 55
Figure 4-26 CUDA speedup of real applications FSMs versus random FSMs with the same attributes 56
Figure 4-27 CUDA execution time of real applications FSMs versus. random FSMs with the same attributes 57

List of Tables

Table 2-1 Recent NVIDIA GPUs and their configurations 21
Table 4-1 Attributes combinations of randomly generated FSMs 35
Table 4-2 Attributes combinations of real application FSMs 36
Table 4-3 System Configuration & Platform Details 36

List of Abbreviations

ATPG	Automatic Test Pattern Generation
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DS	Distinguishing Sequence
FORTRAN	Formula Translator
FSM	Finite State Machine
GPU	Graphical Processing Unit
HTD	Hard-to-Detect
IUT	Implementation Under Testing
KAI	Kuck and Associates Inc.
MBT	Model Based Testing
MD	Mutant Descriptor
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NoWs	Network of Workstations
OpenMP	Open Multi-Processing
PRAM	Parallel Random Access Memory
QA	Quality Assurance
SDL	Standard Description Language
SGI	Silicon Graphics Inc.
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors
SMX	Collection of Multiprocessors
TC	Test Case
TOPTGS	Topological Partitioning Test Generation System
TS	Test Suite
UIO	Unique Input Output Sequence
UML	Unified Modeling Language

Chapter 1 : Introduction

Since the beginning of the second half of the last decade, computer programming has evolved into a discipline of engineering. In 1968, the principle "the establishment and use of sound engineering principles in order to obtain reliable, efficient and economically viable software" was introduced and the appliance of quality assurance (QA) for products became essential in the designing process [1].

Quality assurance and system testing is crucial, yet very expensive, manual, prone to error and time consuming process to guarantee reliability and quality of software testing [2-5]. In 1970s, approximately 50 percent of the project development time spent on the testing, and its cost was more expensive with over than 50 percent of the total cost. These percentages did not change till this time [3, 6, 7].

The development of the techniques and tools required for the derivation of (functional) tests from given models is one of the promising approaches for reducing the expensive cost of testing. A state model, usually representing a implementation under test (IUT), typically consists of states and transitions between states. Prominent state based models include (Mealy) finite state machines (FSMs), labeled transition systems (LTSs), finite state automata (FSA). Various versions of these models were extended to handle variables, variable update statements, and guards specifying enabling conditions of transitions execution.

In this thesis, systems models are considered as FSMs where a transition between a source and a destination (or next) state of an FSM is labeled by an input/output pair illustrating the fact that if the machine is currently at the source state and receives the specified input, it produces the specified output while moving to the next state. Accordingly, an FSM has a behavior that can be described as traces of pairs of input/output sequences. FSMs are extensively used in diverse application domains like lexical analysis [8], communication protocols [9] and other reactive systems. Furthermore, FSMs are the elemental models for formal description techniques, such as Standard Description Language (SDL) [10], Unified Modeling Language (UML) and Statecharts. Application areas of FSM-based testing span a vast range of domains; some of these domains are testing of web services [11-15], communication protocols [16-18],

lexical analysis [8], software design [19], graphical user interfaces [20, 21], sequential circuits [22], embedded systems [23-25], industrial projects [26], object oriented systems, and properties related to security, timing, performance, reliability, and others.

Test derivation from a given FSM specification can be done in many ways and for various purposes. In all cases, based on the given FSM specification, one can identify some selected types of faults that may exist in a black-box FSM IUT, and accordingly enumerate using the specification machine, the set of FSM mutants, called the *fault domain*, according to the selected types of faults. Typical types of faults include output and/or transfer faults [27, 28]. For example, a mutant *M* of a given specification *S* has an output (transfer) if it has a transition with an output (next state) different than the corresponding transition of *S*. Then, starting from an initial set of test cases (test suite), one can run the test cases on the mutants of the considered fault domain and eliminate/kill those mutants whose behavior, with respect to the considered tests, is different from that of the specification machine. An FSM mutant is *killed* (or *eliminated*) by a test case if the output responses (sequences) of the mutant and the specification machines to the input sequence of the test case are different. A test case *TC* is a pair of an input sequence and its corresponding (expected) output sequence. An FSM mutant *M* is *distinguishable* from another (mutant or the specification) FSM *S* if there exists an input sequence (test case) that when applied to these machines, they produce different output sequences; otherwise, if such an input sequence does not exist, the machines are *not distinguishable* [29-33]. This process is called mutation testing.

The tale of mutation testing goes back to1971 in a student paper by Lipton [34]. The field birth can also be determined in published papers in the late 1970s by Hamlet [35] and DeMillo et al. [36, 37].

Mutation testing can be used for testing software at many levels such as, the unit level, the integration level, and the specification level. It has been adapted to many programming languages as a white box unit test technique. For example, it has been adapted to Ada programs [38], FORTRAN programs [39], Java programs [40], C programs [41] and SQL code [42]. Furthermore, mutation testing has also been used for integration testing, software implementation level and applied at the design level to test the specifications or models of a program. For example, at the design level, mutation testing has been adapted to network protocols [26, 32, 43, 44], security

policies [45], FSMs [11, 46, 47] and Web services [48]. The reader can refer to [37, 49] for more surveys and researches about mutation testing.

Another typical FSM test derivation method is based on finding tests that can distinguish the specification machine from one or more mutants in the selected fault domain. Then, tests are run to reduce the mutants (mutants' elimination) of the fault domain, and the process is repeated till all mutants of the fault domain that are distinguishable from the specification machine are eliminated, and thus a complete test suite with respect to the assumed fault domain is derived. This type of test derivation is a form of specification FSM-based mutation testing. Specification based mutation testing is a common mutation testing approach that has many application areas. For more information about mutation and specification based, including FSM-based, mutation testing with related methods and tools, the reader may refer to the survey in [37].

Mutants' elimination is also used while assessing the fault coverage of many types of test suites. Given that many test suites are derived with respect to different test coverage derivation criteria or fault models, different categories of mutants are derived with respect to some selected mutation operators, and then the test suites are run against the mutants. The coverage of the test suites, usually assessed using a considered mutation score, is determined. A huge number of papers are proposed in the literature for studying the effectiveness of typical types of test suites. Studies on the effectiveness of test suites are mostly summarized in [37, 50-58]. It is worth mentioning that FSM-based mutants' elimination is used in a recent assessment presented by El-Fakih et al. [59]. Many types of FSM test suites and the fault coverage of these test suites are assessed with respect to many fault domains (types of faults). In this assessment, tools, including the one given by Simao et al. [60], are used to assess the mutation scores of the test suites with respect to the considered FSM mutants of the assumed fault domains.

Another interesting application area of the mutants' elimination problem is in FSM-based fault diagnosis [28, 61-66]. Given a faulty (black-box) IUT, the objective of fault diagnosis is identifying the faulty IUT, i.e. locating the faults in the faulty FSM. This is carried out by the derivation of a set of diagnostic candidates. FSM mutants, representing the fault domain, based on the observed behavior of the IUT with respect to an initial test suite, and then on further tests, called *diagnostic tests*, are derived and

run against the diagnostic candidates and the IUT to eliminate the candidates that do not have the same behavior with respect to the applied tests as the given IUT. This process of test derivation and mutants' elimination is repeated until all candidates that are distinguishable from the given IUT are eliminated; and thus, the faulty implementation is located.

Parallel machines in the form of multicore CPUs and manycore GPUs have become widely available in the last 10 years. This has in turn helped solving complex problems with less time. There are many options that researchers can use to write parallel code like Open Multiprocessing (OpenMP) which utilizes many cores in the same machine to execute more than one thread simultaneously [67, 68]. Message Passing Interface (MPI) which utilizes disrupted memory systems which are usually clusters of computers Network of Workstations (NoWs) with isolated memory [69, 70]. Also, Compute Unified Device Architecture (CUDA) development toolkit permits GPU programming in a C-like language, and exploits the computation power of the Graphical Processing Unit (GPU). But the problem with these tools is that they do not indicate "what-to" parallelize; rather they specify "how-to" parallelize. Software development professionals must defy the challenge of developing software that takes advantage of this hardware [71].

Thus, it is clear from above that the process of eliminating mutants is a major step in many FSM-based testing related activities as it is the case in mutation testing, fault diagnosis, and the assessment of the effectiveness of test suites. However, in general, the number of mutants in a considered fault domain can be extremely huge. Accordingly, in this thesis, the aim is reducing the execution time (and thus energy) of mutants' elimination. To this end, three parallel implementations for reducing the time efforts of mutants' elimination utilizing state-of-the-art parallel technologies are presented and assessed. The first is an Open Multi-Processing (OpenMP) implementation that utilizes many cores in the same machine. The second implementation uses a clustered system using Message Passing Interface (MPI) standard. Finally, the third is based on the Compute Unified Device Architecture (CUDA) that exploits the computation power of the Graphical Processing Unit (GPU).

A comprehensive assessment is presented where the objective of the assessment is to determine and compare the performance of the presented implementations in terms

of execution time and speed up. The experiments carried out in this thesis show that the sequential implementation does not scale for big application examples, and accordingly, the performance of the parallel implementations is assessed for big examples. The experiments reveal that the GPU implementation using the software platform CUDA gives the best performance amongst all the parallel implementations, and the speedup obtained is much more significant than the OpenMP and MPI implementations. The parallel implementation on the NoW using the MPI gives the second-best performance, and the parallel implementation on a multi-core CPU gives the third best performance. Finally, as CUDA performance was extremely better than the other parallel implementations, the scalability of CUDA to huge application examples is studied. The assessment also includes some analysis on the attributes related to the considered mutants' elimination problem. Namely, the FSM size (i.e., number of transitions), and number of considered mutants to eliminate, *TS* length and number of test cases of a *TS*. In addition, an experimental analysis is given to determine the parallel implementations attributes such as the number of invoked threads in OpenMP, the number of launched processes in MPI, CUDA block size as well as CUDA RunSize which is the size of GPU shared memory that is allocated to save the test case in the elimination process. Last but not least, it is worth mentioning that an experimental assessment was carried out using both randomly generated FSMs and real application FSMs.

This thesis is organized as follows; Chapter 2 includes preliminaries related finite state machines, mutants, and test cases, as well it introduces the considered parallel technologies. Chapter 3 includes the proposed sequential, OpenMP, MPI and CUDA parallel implementations used for mutants' elimination. Chapter 4 includes the experimental evaluation. Chapter 5 includes related work and Chapter 6 concludes this thesis.

Chapter 2 : Preliminaries

2.1 Finite State Machine (FSM)

A deterministic *finite state machine* is an initialized complete deterministic Mealy machine that can formally be defined as a 6-tuple $S = (S, I, O, \delta, \lambda, s_0)$ where S is a finite set of states, s_0 is the *initial state*, I is a finite set of input symbols, O is a finite set of output symbols, δ is a next state (or transition) function: $\delta: S \times I \rightarrow S$, λ is an output function: $\lambda: S \times I \rightarrow O$. Usually, functions δ and λ are extended to input sequences.

For an input sequence $\alpha = i_1 i_2 \ldots i_k \in I^*$ at a state s, $\beta = o_1 o_2 \ldots o_k \in O^*$ denote the corresponding output sequence obtained by applying α at s. The pair α / β is an *Input/Output (I/O) sequence* at state s.

A transition of an FSM a 4-tuple $t = (s, i, o, s')$ representing the fact that if the machine is at source state s, upon receiving the input i, it produces the output o while moving to the next state s'. Common types of FSM faults are transfer, output, extra state, and mixed faults [27]. Given an FSM specification S, a transition t of a mutant M of S has an *output* (*transfer*) *fault* if it has a transition with an output (next state) different from that specified at the corresponding transition of S, i.e., for $t = (s, i, o, s')$ of S, M has an output fault at t if M has the transition (s, i, o', s') where $o' \neq o$, and M has a transfer fault (at t) if M has the transition (s, i, o , s'') where $s'' \neq s'$. A mutant M of S has *multiple* faults if has many faulty transitions.

Given two FSMs, a specification S and a *mutant* M, defined over the same input and output alphabets, an input sequence α is a *distinguishing sequence* for S and M (that is for the initial states of S and M) if the output responses of S and M to the input sequence α are different. In this case, the input sequence *distinguishes* S and M or simply α *kills* (*eliminates*) the mutant M. If there is no such an input sequence, S and M are *indistinguishable*.

A *test case TC* is a pair of an input sequence $i_1 i_2 \ldots i_k$ of the specification FSM S and its corresponding (expected) output sequence $o_1 o_2 \ldots o_k$, that is, a test case is an input/output sequence of the initial state of S. A test case might be written using its

corresponding input/output pairs, i.e, as i_1/o_1 i_2/o_2 ... i_k/o_k. *Length* of a test case *TC* is the number of inputs of the test case. A *test suite TS* is a finite set of test cases. *Length* of a test suite *TS* is the total length of its corresponding test cases. The number of test cases of a test suite is denoted as |*TS*|. A set of *k* mutants, of the specification machine *S*, is denoted as $\Im = \{ M_1, ….. M_k \}$. A test suite of *m* test cases is denoted as $TS = \{ TC_1, …. TC_m \}$. As the considered sets $\Im$ has huge number of mutants, in order to reduce storage space, instead of explicitly saving the mutants in $\Im$, for each mutant M_i, only the descriptor Md_i that includes the faulty transition(s) of the mutant in respect to the specification *S* is saved. Thus, when needed, the mutant M_i itself can be derived from both its descriptor Md_i and the specification *S*.

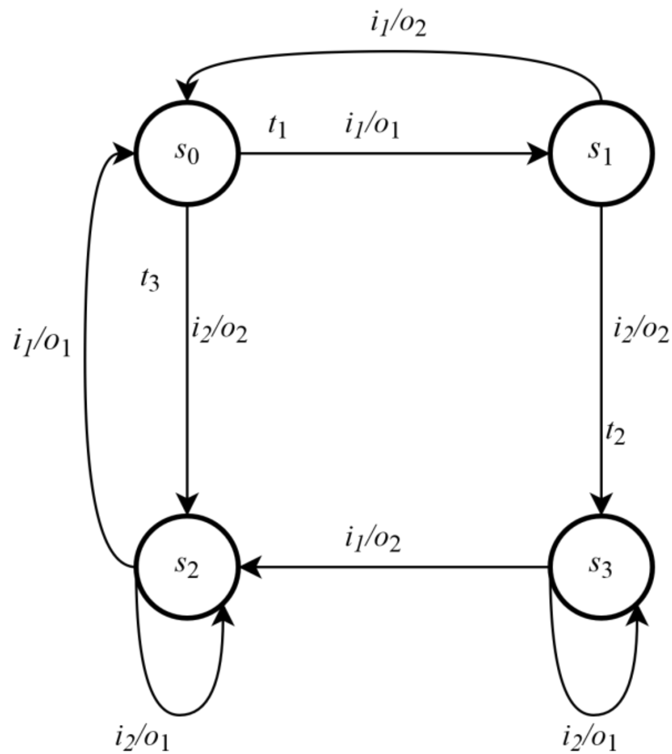

Figure 2-1 Finite State Machine *S*

As an example, consider the FSM *S* in Figure 2-1 defined over the sets of inputs $I = \{i_1, i_2\}$, outputs $O = \{o_1, o_2\}$, and states $S = \{s_0, s_1, s_2, s_3\}$, respectively. A mutant descriptor Md_1 that includes the transition (s_0, i_1, s_2, o_1) represents the fact that the

mutant M_1 of S has the transfer fault (s_0, i_1, s_2, o_1) where the mutant transfers to the state s_2 instead of s_1 as specified in the corresponding transition (s_0, i_1, s_1, o_1) in S.

2.2 Open Multi Processing (OpenMP)

OpenMP (Open Multi-Processing) is a standard [57] for shared-memory parallel programming. OpenMP was the result of joined project between Kuck and Associates Inc. (KAI) and Silicon Graphics Inc (SGI) in the spring of 1996. The combining efforts of these two companies gave birth to the idea of an industry sponsored directive-based symmetric multiprocessor (SMP) programming standard, soon known as OpenMP [72].

At its most elemental level, OpenMP is a collection of compiler directives and callable runtime library routines that extend C/C++ and FORTRAN so that programmer assisted shared-memory parallel programming can be delivered with minimal effort. OpenMP compiler directives specify how workload is shared among threads, while also controlling threads synchronization and determining the scope of variables. A parallel program can be easily developed by the application programmers through incrementally inserting directives into time critical sequential codes.

2.3 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a standard for distributed memory parallel programming [70, 73-75]. The MPI standard specifies the names, calling sequences, and results of the subroutines or functions to be called from C, C++ or FORTRAN programs. Several implementations are available for MPI such as OpenMPI, pyMPI (MPI implementation in Python) and MPICH; some of them are commercial and others are free. These implementations can be executed on both tightly-coupled Massively Parallel Machines (MPPs), and on Networks of Workstations (NoWs) [69].

MPI specifies the communication among a set of processes composing a concurrent program. The message passing paradigm is very attractive because it is portable and scalable. MPI programs can be deployed on both shared-memory and distributed-memory multiprocessors, and combinations of both of them.

MPI processes are assumed to reside in disjoint memory spaces. Data exchange is performed via message exchange. MPI is responsible for process identification, message routing and buffering.

MPI provides a wide gamut of communication primitives that cater for point-to-point, collective as well as one-sided communication. Collective operations such as "MPI_Bcast" ease communications involving more than two processes, i.e. broadcasting specific information from a particular process to all other processes. Point-to-point operations like the "MPI_Send"/ "MPI_Recv" pair provide rudimentary communications between processes using a two-sided model, meaning that both communicating processes must issue matching calls; one for sending and one for receiving. One-sided communications were introduced in recent versions of the MPI standard, decoupling data transfer from synchronization and allowing remote memory access [74].

2.4 Graphical Processing Units (GPUs)

A Graphical Processing Unit (GPU) is a specialized computing architecture unit designed to accelerate computer gaming applications graphics operations. A GPU chip contains a large number of parallel microprocessors, designed to reduce the workload of the CPU and accelerate 2D or 3D graphic processing. Recent GPUs are composed of a large number of computing cores that are connected to high-speed memory (DDR5) with very wide buses (256bit or larger), and they are able to execute multiple threads n parallel. The multi-thread architecture of the GPU allows execution of many threads in parallel to attain high speed of execution compared to traditional single or multi-core CPUs [76].

GPU hardware development started as a single core. Fixed function hardware pipeline application then evolved to a combination of highly parallel programmable cores that can be used for general purpose computation and scientific computation. GPU technology has always progressed by adding more programmability and parallelism to a GPU core architecture. Individual GPU cores have gained over the years features common to CPU cores, such as multi-level cache memories and dedicated floating print co-processors [76].

GeForce 3 was released by NVIDIA in 2001; this was the first GPU with a programmable pipeline and ability to program previously non-programmable parts of the pipeline. Afterwards, fully programmable graphic cards were introduced. The introduction of DirectX9, which provided the programmability in the GPU hardware, started GPU computing wave [76].

In 2006, NVIDIA introduced the GeForce 8 series. This series, which contained massive parallel processors, was a great evolution in the history of GPUs. NVIDIA's Fermi architecture GPU, introduced in 2009, featured a concurrent kernel execution, true memory cache hierarchy, combined memory address space, better double precision performance and dual warp schedulers. Since then, breakneck progress in the development of GPUs has occurred. Readers can refer to Appendix A for more information on recent GPUs [77, 78].

Some of the recent NVIDIA GPUs and their configurations are described in Table 2-1, where Streaming Multi-processor (SM) represents a collection of cores with Cores/SM as the number of cores in each SM. Cores represents individual (single) cores (computing unit) contained in a GPU card, each core is capable of executing a thread [79].

As observed in literature results, GPUs offer an enormous performance boost to scientific computing. This motivated me to implement a GPU version of the mutation testing algorithm. By applying test cases to kill FSM mutants on a GPU, the execution time can be reduced and significant speedup can be attained compared to the corresponding CPU solution.

Various software platforms can be used to execute code on GPUs such as Thrust, CUDA, OpenCL, etc. [80]. In this thesis, CUDA is used, which is arguably one of the most mature tools, delivering at the same time maximum performance by allowing explicit machine control.

2.4.1 Compute Unified Device Architecture (CUDA). Compute Unified Device Architecture (CUDA) is a parallel computing platform that was introduced by NVIDIA at the end of 2006. CUDA provides an API and a toolkit (SDK) for harnessing GPU hardware for general purpose computing. CUDA supports development in C/C++

and Fortran. Higher level platforms such as OpenACC and Thrust also work on top of CUDA [81].

Table 2-1 Recent NVIDIA GPUs and their configurations

Card	**Cores**	**Cores/SM**	**SM**	**Compute Capability**
Nvidia Titan X	3584	128	28	6.1
GeForce GTX 1080	2560	128	20	6.1
GeForce GTX 1070	1920	128	15	6.1
GeForce GTX Titan X	3072	128	24	5.2
GeForce GTX 980 Ti	2816	128	22	5.2
GTX 980	2048	128	16	5.2
GTX 970	1664	128	13	5.2
GTX 960	1024	128	8	5.2
GTX TITAN Z	5760	480	12	3.5
GTX TITAN Black	2880	240	12	3.5
GTX Titan	2688	192	14	3.5
GTX 780	2304	192	12	3.5
GTX 770	1536	192	8	3.0
GTX 760	1152	192	6	3.0
GTX 690	3072	192	16	3.0

At the hardware level, a CUDA-capable GPU processor is a collection of multiprocessors (SMX); each having a number of cores (processors). Each multiprocessor has its own shared memory which is common to all its processors. It

also has a texture memory (a read only memory for the GPU), constant (a read only memory for the GPU that has the lowest access latency) memory caches and a set of 32-bit registers. In any given cycle, the same instruction is executed in each SMX core in a synchronous fashion. Each core can operate on different data hence each SMX is a Single Instruction Multiple Data (SIMD) processor. GPU card memory (referred to as global memory) is available to all the available cores, making it suitable for holding shared data or structures [5]. CUDA provides a set of atomic primitives for operating on global memory locations without the introduction of race conditions.

The CUDA programming model dictates the use of fine-grained parallelism as required by massively parallel GPUs. In the CUDA programming model, the host CPU memory (host memory) and the GPU device memory (global memory) are disjoint, necessitating the explicit data transfer between the two.

From the programmer's point of view, the CUDA model is a collection of threads executing in parallel. All threads run a function called a kernel. Kernel invocations are asynchronous, i.e. the CPU can continue to operate during GPU computation; therefore, the CUDA programming model is a hybrid computing model [82].

The threads are launched as a 3-D grid of 3-D blocks of threads. The maximum size per dimension and the overall size of the grid and block structures are determined by the characteristics of the target GPU. Each block is executed on one SMX. As the block size may exceed the number of cores on an SMX, each block is divided in so called "warps". A warp is a set of 32 threads running synchronously on the cores of a SMX. Multiple warps may be active at the same time in a SMX, depending on the exact structure and the warp schedulers of the SMX[5]. Applications that require synchronization between the threads of a block, can use the __syncthreads() primitive to achieve this. No synchronization is available between different blocks.

The SMX memory architecture incorporates two additional types of on-chip memory: a register which is a private memory for each thread, and shared memory which is a common memory for all the threads within a block, as illustrated in Figure 2-2 [82]. Shared memory is significantly faster than global memory, albeit limited in size (currently 48kB).

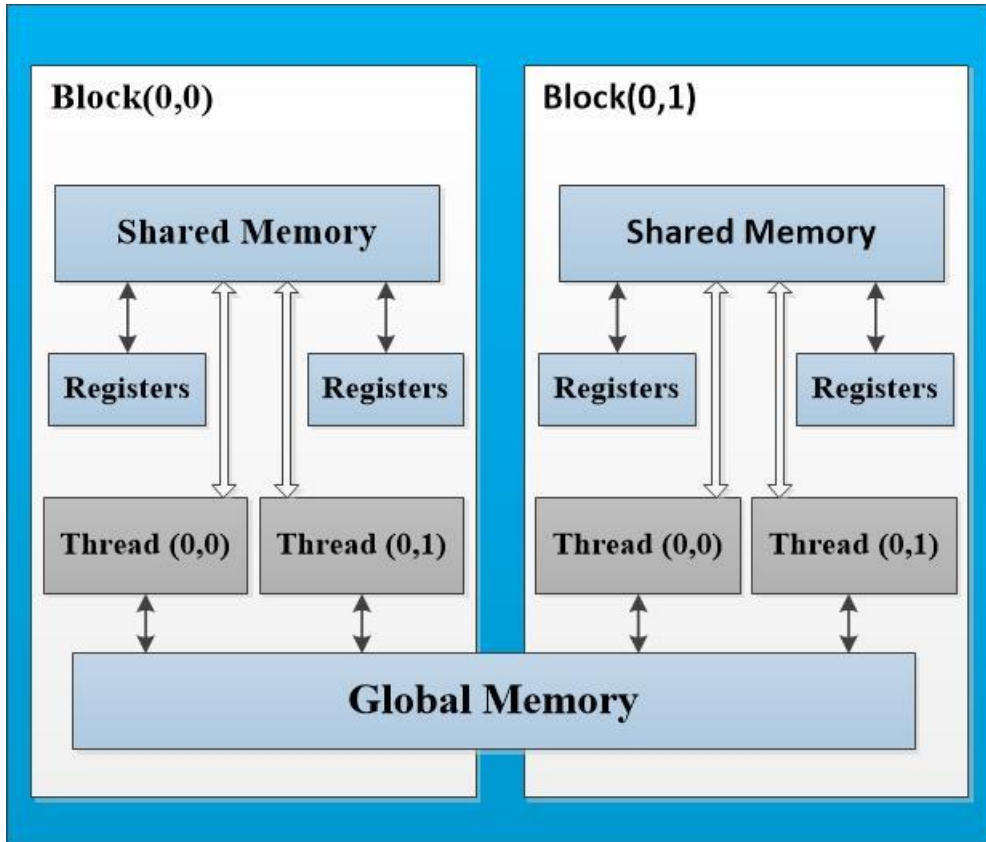

Figure 2-2 CUDA Programming Model

Chapter 3 : Proposed Implementations

In this chapter, a sequential algorithm for eliminating FSM mutants is presented, followed by three parallel implementations of the algorithm, namely an OpenMP, MPI, and CUDA implementation. The first is an OpenMP implementation that utilizes many cores in the same machine. The second is an implementation that targets a cluster using the MPI standard. Finally, the third is based on CUDA that exploits the computation power of the GPU.

3.1 Sequential Algorithm

The input of the algorithm is a complete deterministic FSM $S = (S, I, O, \delta, \lambda, s_0)$, a test suite TS of m test cases, $TS = \{TC_1, \ldots. TC_m\}$, and set of k mutants descriptors $MD = \{Md_1, \ldots. Md_k\}$ for the assumed set of mutants $\Im = \{M_1, \ldots. M_k\}$ of S. The descriptor Md_i of a mutant M_i represents only the differences between the specification and the mutants; and thus, when needed, the mutant M_i can be derived using S and the corresponding descriptor Md_i. The algorithm operates on one mutant at a time. For each test case, the elimination process starts by feeding the mutant with the inputs of the test case in sequence, and observe the corresponding outputs. If the observed output differs from the expected output, applying tests to the mutant is stopped, and the mutant is marked as killed (or eliminated). Otherwise, if the mutant is not killed by any test case, then it is labeled a *survived mutant* and it is added to a list of survived mutants (*SMD*), which is a subset of the set *MD*. The output of the algorithm is the list *SMD*.

Sequential Algorithm:

For each Md_i in *MD*, do **(Loop-1)**

Modify S to create the mutant M_i using Md_i

killed = **FALSE**

While *TS* contains a *TC* not applied to M_i AND *killed* = **FALSE** do:

killed = **Eliminate_Mutant(***TC***,** M_i**)**

End While

If *killed* is **FALSE**

$SMD = SMD \cup M_i$

End If

End For (Loop-1)

Return *SMD*

Procedure Eliminate_Mutant(M_i, TC)

While *TC* sequence is not exhausted AND M_i is not killed, do:

Obtain output generated by M_i for the next input in *TC*

If the obtained output is different than the corresponding output in *TC*

Return TRUE

End If

End While

Return FALSE

Example 1: As an application example of the mutation elimination process, consider the FSM *S* in Figure 2-1. Let the set of mutants descriptors (*MD*) contains three descriptors Md_1, Md_2 and Md_3. Where transitions $t_1 = (s_0, i_1, s_1, o_1)$, $t_2 = (s_1, i_2, s_3, o_2)$ and $t_3 = (s_0, i_2, s_2, o_2)$ of *S* have transfer faults and they change to (s_0, i_1, s_2, o_1), (s_1, i_2, s_1, o_2) and (s_0, i_2, s_1, o_2) respectively. The initial test suite contains two test cases; $TC_1 = (i_1/o_1\ i_1/o_2)$ and $TC_2 = (i_1/o_1\ i_2/o_2\ i_2/o_1)$

In the sequential algorithm, all the mutants are checked one by one. For each mutant, the test cases are applied one by one until the mutant is eliminated. If there was

no test case capable of discovering and eliminating the mutant, the mutant will be added to a set of survived mutants for further analysis.

The algorithm can be applied on Md_1, Md_2 and Md_3 as follows: Using Md_1, S is modified to create mutant M_1, now TC_1 is applied on M_1; at source state s_0, upon receiving the input i_1, it produces the output o_1 while moving to the next state s_2. At source state s_2, upon receiving the input i_1, it produces the output o_1 while moving to the next state s_0. Here, the produced output o_1 is not equal to the expected output o_2, hence the mutant is killed and no need to apply TC_2 on the mutant.

For Md_2 and Md_3, the same steps are executed. First, M_2 and M_3 are created from S using Md_2 and Md_3 respectively, then the test cases on both of them are applied. As a result, M_2 will be eliminated by TC_2 while M_3 will survive and Md_3 will be added to *SMD*.

3.2 OpenMP Implementation

In this section, a parallel implementation of the sequential algorithm is included based on a multi-core CPU via multiple threads using OpenMP. Similar to the sequential algorithm, the test cases are applied on the mutants one by one. However, different threads are used for applying a test case in parallel to a number of mutants. The number of mutants that can be checked in parallel depends on the number of invoked threads, and this can be automatically scheduled by the OpenMP scheduler, or can be specified by the programmer. In this implementation, the number of threads equals the number of logical cores. Hyperthreading was enabled in the used test platforms, maximizing the potential performance extracted from the hardware.

The distribution of mutants among threads depends on the schedule type being used, static or dynamic. In static scheduling, the mutants are divided evenly among the threads. In case the mutants cannot be divide evenly between the threads, only the last thread share may be different than the others. In dynamic scheduling, the mutants are divided into subsets and each thread works on one subset at a time. When a thread completes its subset, it takes another available subset. Dynamic scheduling performance can suffer from the coordination cost of acquiring new workload. Thus, the results reported in this paper are based on static scheduling.

In the OpenMP implementation shown below, the only major difference from the sequential algorithm is the partitioning of Loop-1, i.e. each of the N threads is assigned a $1/N$-sized subset of the mutant set.

OpenMP Implementation

Let *SM* be a vector of arrays to save survived mutants in each thread

Do In Parallel: For Each Md_i in *MD* (**Loop-1**)

Let *tID* be the unique thread identifier

Modify *S* to create the mutant M_i using Md_i

killed = **FALSE**

While *TS* contains a *TC* not applied to M_i AND *killed* = **FALSE** do:

killed = **Eliminate_Mutant(***TC***,** M_i**)**

End While

If *killed* is **FALSE**

$SM[tID] = SM[tID] \cup M_i$

End If

End-For (**Loop-1**)

$SMD = \cup_{\forall\, tID} \boldsymbol{SM[tID]}$

Return *SMD*

Example 2: As an application example of OpenMP, considering Machine *S* in Figure 2-1. Let the set of mutants descriptors (*MD*) contains ten thousand descriptors $\{Md_0, Md_1, \ldots, Md_{9999}\}$. The test suite *TS* has two test cases and the number of invoked threads $N = 10$.

The mutants are distributed evenly between the threads; each thread contains $10000/N = 1000$ mutants. Within each thread, the sequential algorithm is applied to

eliminate the set of mutants assigned to it. Each thread saves the set of survived mutants in a pre-allocated vector dedicated for it. When all the threads complete their work, the survived mutants are collected from the dedicated vectors in *SMD*.

3.3 MPI Implementation

Similar to OpenMP, in MPI several mutants are eliminated in parallel, but here a test case is applied in parallel to many mutants using different processes rather than different threads, where each process has its own isolated memory and the communication between processes is done via messages. The number of mutants that can be checked in parallel by a process depends on the number of launched processes. In the conducted experiments, the number of launched processes equals the total number of physical cores in the considered cluster. The used test platform consisted of four heterogeneous workstations.

As in OpenMP, employ two ways for distributing the mutants among the processes can be applied. In a static distribution, an equal number of mutants is assigned to each process. However, unlike the OpenMP case where all cores are identical, the execution time is not the same across all the cluster machines, as they are equipped with different CPUs. For this reason, a dynamic distribution based on the master-worker pattern is employed. The master process coordinates the distribution of the mutants, while the worker processes eliminate the mutants assigned to them. When the master process starts, it divides the set of mutant descriptors into smaller subsets. Subsequently, it assigns a subset to each worker process. The master then listens for incoming load requests in order to hand-out unprocessed subsets, until set $\Im$ is exhausted.

On the other hand, upon initiation a worker process works on the initially assigned subset. When this is processed, the worker requests a new subset from the master. This process continues until the end signal is received from the master. At that moment, the survived mutants in the workers are collected by the master process.

The worker processes work exactly as the sequential algorithm, as each worker has a subset of mutant descriptors and works on eliminating these mutants sequentially. The master process is only responsible for load balancing, which is,appropriately assigning subsets of descriptors to the workers. To improve the implementation

efficiency, the master process collects only descriptors indices of the survived mutants. MPI algorithm is shown in the form of a flowchart in Figure 3-1.

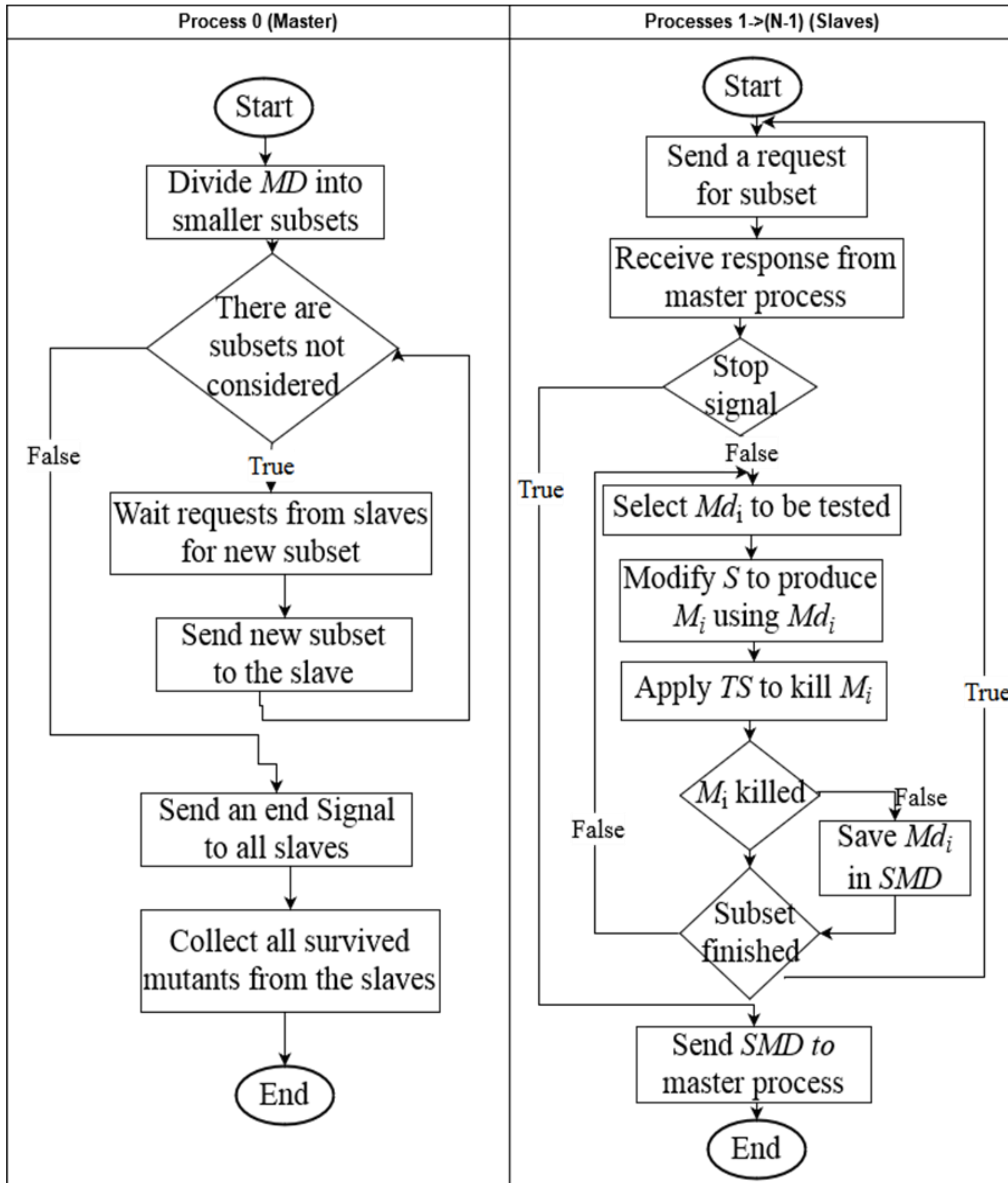

Figure 3-1 MPI Implementation

Example 3: MPI implementation is similar to OpenMp in which the set of mutants is divided into N processes $\{P_{0}, \ldots P_{N-1}\}$ where P_0 is the master process, but,

in MPI, dynamic distribution is used through a master-worker arrangement. As an application example of the MPI implementation, consider the same FSM S in Figure 2-1, let the set of mutants descriptors (*MD*) contains ten thousand descriptors $\{Md_0, Md_1 \ldots Md_{9999}\}$. The test suite *TS* has two test cases and the number of launched processes is twenty; one serving as the master and 19 as the workers.

The 10000 mutants are distributed evenly into several subsets. The number of mutants in each subset is calculated in the master process P_0 using the formula ($|subset| = |MD|/ (N*8)$). P_0 assigns the first N-1 subsets to the worker processes $\{P_1,\ldots,P_{N-1}\}$ respectively, then starts listening for requests from them; each time responding with a new subset. When all the subsets are distributed, an "end" message is sent to all workers.

Each worker process executes the sequential algorithm on the initial subset assigned from P_0. When the worker process completes the assigned subset, it asks for a new subset to operate on until an "end" signal is received. In response to the "end" signal, a worker sends the survived mutants it collected to the master process.

3.4 CUDA Implementation

In this section, parallel implementation is presented based on the CUDA platform. In CUDA a massive number of threads can be launched at the same time allowing the assign of one thread per mutant.

The GPU is employed as a co-processor for filtering the mutants. The CPU transfers to the GPU memory the machine specification, mutant descriptors and test suite, and collects the survived mutants once the GPU completes its execution.

The following notations and variables are used in the implementation:

- *RunSize*: represents the number of inputs from a test case (sub-sequence length) that will be applied to the set of mutants in each GPU invocation.
- α_c: represent a subsequence of a *TC*, each *TC* will be divided into l disjoined subsequences $\alpha_1 \ldots \alpha_l$, each with length equal to *RunSize*.
- *ReachedLoc*: array where the state (of the mutant) reached after completing the application of subsequence α_c is saved.

- *notKilled*: array where the surviving mutant identifiers are saved after each kernel invocation. Subsequent kernel launches use this array for mapping mutants to threads. Initially, this array contains the indices of all the descriptors in *MD*.
- *ThreadSize*: represents the number of launched threads inside the GPU.
- *BlockSize*: represents the number of threads inside each block in the GPU.

To determine the optimum BlockSize, an experiment was executed on FSMs S_1, S_3, S_5 and S_7 (see Table 4-1). The used *TS* length equals $2 \times n \times |I|$ and has $n/2$ test cases. The number of considered mutants is fixed to 2.5 million. And *BlockSize* ranging from 32 to 1024 with step equals 32.

As depicted in Figure 3-2, there were two optimum *BlockSize* values; 256 was optimum in FSMs S_1 and S_7 while 512 was optimum in FSMs S_3 and S_5.

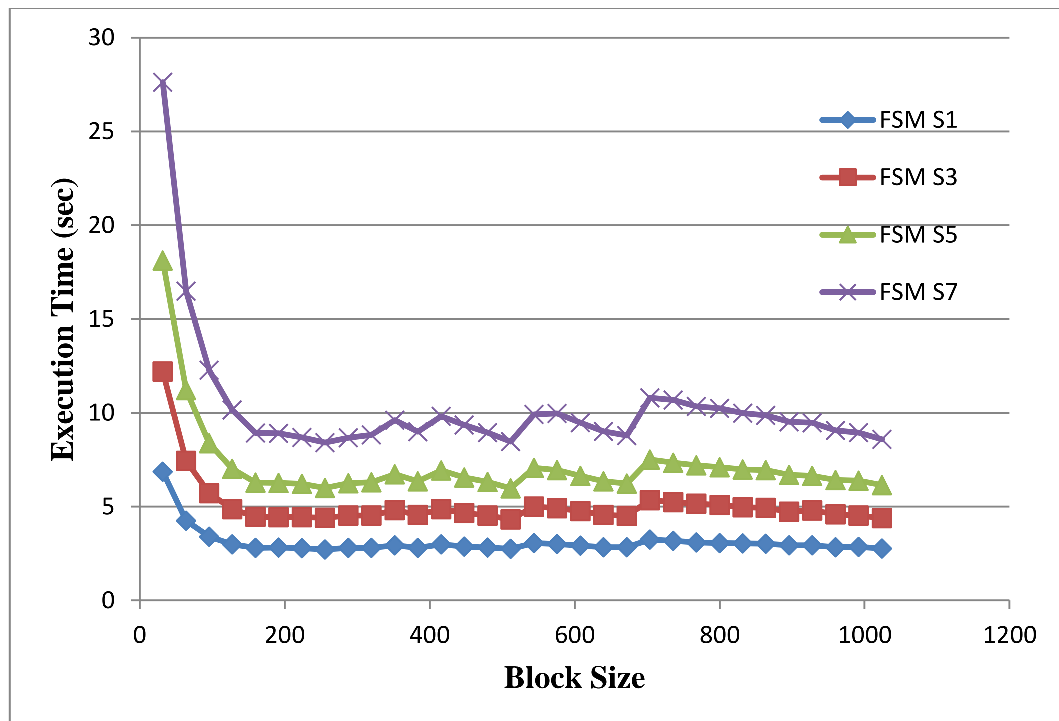

Figure 3-2 CUDA Execution Time versus Block Size

Another Experiment is executed using FSMs S_8, S_9 and S10 from Table 4-1. In this experiment, A massive test with number of considered equals to 500 million is executed. TS length that has only 1 test case with length equals to half the average of the used FSMs transitions. *BlockSize* with values 256 and 512.

As presented in Figure 3-3, for massive experiments *BlockSize* equals to 256 is better than 512. Thus, the used *BlockSize* in the conducted experiments is 256.

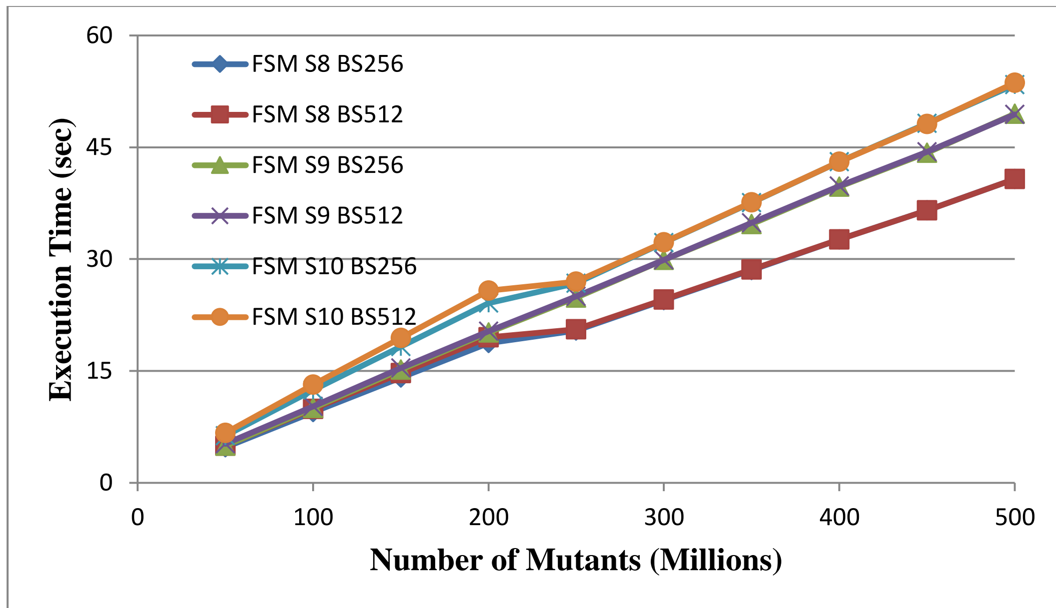

Figure 3-3 Block Size 256 versus Block Size 512

CUDA

CPU part

Copy machine *S* specification, *MD* set to GPU global memory

While *TS* contains a test case TC_i not applied to *M* AND *MD* is not empty, do:

Copy TC_i to GPU global memory

Divide TC_i into l disjoint subsequences $\alpha = \alpha_1 \ldots \alpha_l$ of length ($|TC_i|/$ *RunSize*),

For each subsequence α_c of α, and as long as *MD* not empty, do:

Set *ThreadSize* to the size of the *MD* set

Set *reachedState* to the state before the start of α_c

Invoke GPU Kernel to apply α_c on each member of *MD*

Set *MD* to the set of survived mutants

End For

End While

Copy *MD* (survived mutants) from global memory to host memory

GPU part (kernel function):

Calculate global thread ID, t_{ID}

Copy α_c into shared memory

Copy the mutant descriptor Md_{tID} into thread memory (registers)

Load *reachedState* from global memory into thread memory

If Eliminate_Mutant (α_c, Md_{tID}) is **False**

Copy Md_{tID} to survived mutants

Copy *reachedState* to global memory

End if

Example 4: As an application example of the CUDA implementation, consider the FSM *S* in Figure 2-1. Let the set of mutants descriptors (*MD*) contains ten thousand descriptors {Md_0, Md_1 …Md_{9999}} and the test suite *TS* has one test case TC_1 with length 6000.

CPU Work

The CPU will copy the FSM *S* and set of mutants descriptors *MD* into the GPU global memory. After that, TC_1 will be divided into into *l* disjoined subsequences α_1 … α_l, each with length equal to *RunSize*. For each subsequences α_c, *ThreadSize* is initialized to the number of not-killed mutants , then the GPU kernel is invoked to apply α_c on the mutants set. After each GPU kernel invocation, the *ThreadSize* will be recalculated, thus each time the number of launched threads in the GPU is reduced.

GPU Work

GPU kernel is responsible only on applying the subsequence α_i on the set of mutants to eliminate them.

Chapter 4 : Experimental Results

This chapter presents the experiments conducted to determine the execution time and speedup of the parallel implementations in comparison to the sequential algorithm. Experiments are conducted using both randomly generated and real application FSMs. The experiments goal is to assess both the relative speedup and the scalability of the implementation. Furthermore, a thorough analysis is presented to assess the impact of the machine size and the number of mutants on the execution time

4.1 Used FSMs, Software and Hardware platforms

The attributes of the randomly generated FSMs used in the experiments are shown in Table 4-1. These FSMs were derived using the generator used in [83]. Table 4-2 shows the attributes of the real application FSMs taken from the ACM/SIGDA benchmarks [84].

Table 4-1 Attributes combinations of randomly generated FSMs

FSM machine name	Number of states (n)	Number of inputs ($\|I\|$)	Number of outputs ($\|O\|$)	*Number of transitions* $n\times\|I\|$
S_1	500	100	100	50000
S_2	625	120	125	75000
S_3	625	160	160	100000
S_4	625	200	200	125000
S_5	750	200	200	150000
S_6	875	200	200	175000
S_7	1000	200	200	200000
S_8	5000	100	100	500000
S_9	6000	125	125	750000
S_{10}	10000	100	100	1000000
Random s208	18	2048	4	36864
Random Nucpwr	29	8192	20	237568
Random ram_testO	72	65536	27	4718592
Random Rs820o	25	262144	22	6553600

Table 4-2 Attributes combinations of real application FSMs

FSM machine name	Number of states (n)	Number of inputs ($\|I\|$)	Number of outputs ($\|O\|$)	$n*\|I\|$
Nucpwr	29	8192	20	237568
ram_testO	72	65536	27	4718592
s820o	25	262144	22	6553600
s8320	25	262144	22	6553600
s420O	18	524288	4	9437184

Table 4-3 System Configuration & Platform Details

Computer name	**Kingpenguin**	**Dune-970**	**Dune-Titan**	**Dune-770**	**Dune-Frg**	**Setup-T3600**
CPU	Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz	Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz	Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz	Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz
CPU Cores	12	12	6	4	6	4
Threads / Core	2	2	2	2	2	2
RAM	64 GB	32 GB	64 GB	32 GB	32 GB	16 GB
GPU	-	GeForce GTX 970	GeForce GTX TITAN X	GeForce GTX 770	-	Quadro 2000
GPU Cores	-	1664	3072	1536	-	192
GPU RAM	-	4GB	12GB	2GB	-	1GB
Compute Capability	-	5	5	3	-	2
Number of GPUs	-	2	1	2	-	1

The software environment is the same for all the test beds in **Table** 4-3 and is mentioned below:

- Operating System: Ubuntu 16.04.1 LTS (64 bit)
- The GNU C++ 4.9.3 compiler was used with CUDA SDK 8.0.44

The sequential and OpenMP implementations were executed on Dune-970; the CUDA implementation was executed on Dune-Titan, and the MPI parallel implementation was tested on a NoW formed by the six machines Kingpenguin, Dune-770, Dune-Frg and Setup-T3600 shown in Table 4-3.

4.1.1 Determining the TS length. The sequential implementation was used to determine the impact of TS length over execution time and over the percentage of killed mutants. These corresponding experiments were conducted using the small and medium size FSMs S_1 and S_7 (see Table 4-1), respectively. In these experiments, test suites with length ranging from $0.5\times(n\times|I|)$ to $10\times(n\times|I|)$ were used against a number of test cases in a test suite equals n and fixed number of 0.5 million mutants.

Figure 4-1 and Figure 4-2 depict the execution time and the percentage of killed mutants respectively, as *TS* length increases. As Figure 4-1 illustrates, the execution time of the sequential algorithm increases at a rate which is analogous to log(*TS*). In fact, beyond a *TS* length of $4\times(n\times|I|)$, there is hardly any increase in execution time.

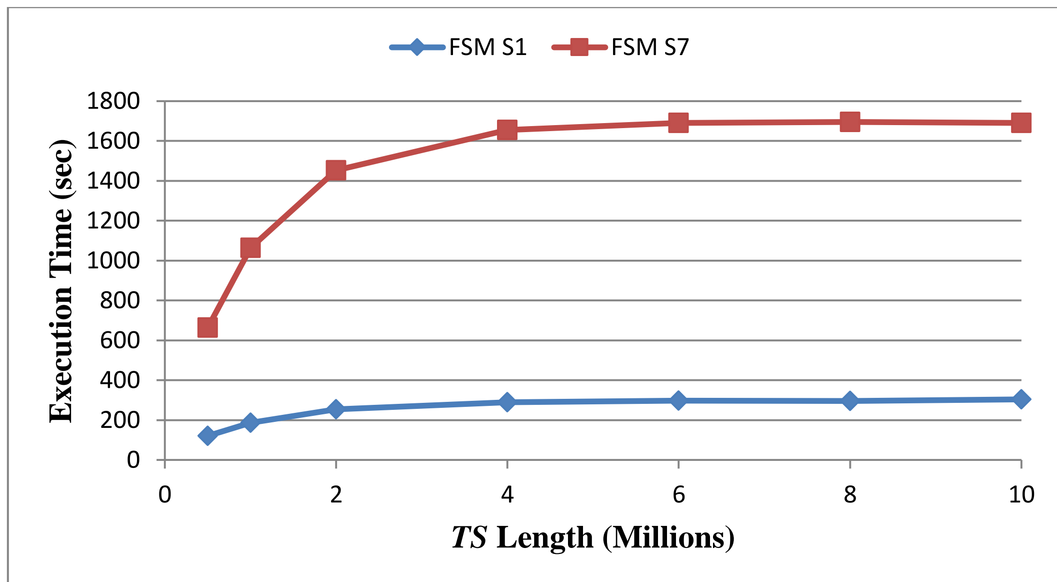

Figure 4-1 Sequential execution time versus *TS* length

Figure 4-2 shows that for *TS* with length $2\times(n\times|I|)$, the percentage of killed mutants reached 86.5%, and no major increase in this percentage is observed when *TS*

length is bigger than or equals $4\times(n\times|I|)$. Thus, in the conducted experiments test suites of length $2\times(n\times|I|)$ is used.

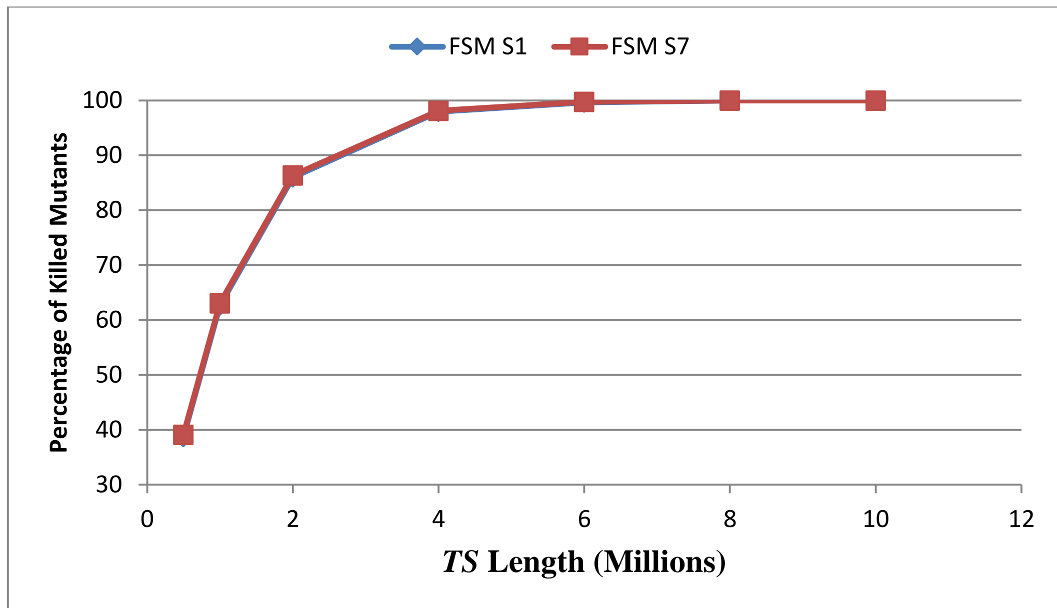

Figure 4-2 Percentage of killed mutants versus *TS* length

4.1.2 Determining the number of test cases in the TS. A test sequence can be broken into smaller subsequences that can be used in turn to reduce the number of mutants that have to be examined as testing progresses. This can be beneficial for execution time as it promotes better cache locality.

For this experiment, FSMs S_1 and S_7 were used. The number of subsequences was set to 1, 0.125n, 0.25n, 0.5n, n to 2n, while the overall *TS* length was set to $2\times(n\times|I|)$ as per the results of the previous section.

Figure 4-3 and Figure 4-4 depicts the execution time as the number of subsequent increases for FSM S_7. According to the results, the least execution time is obtained when a *TS* is broken into $n/2$ test cases. The same pattern was observed for FSM S_1. For both machines, a minor difference of 0.003% is obtained between the percentages of killed mutants among the different numbers of used test cases. Given the above results, in the experiments described below, the overall input length $2\times(n\times|I|)$ is $n/2$ test cases.

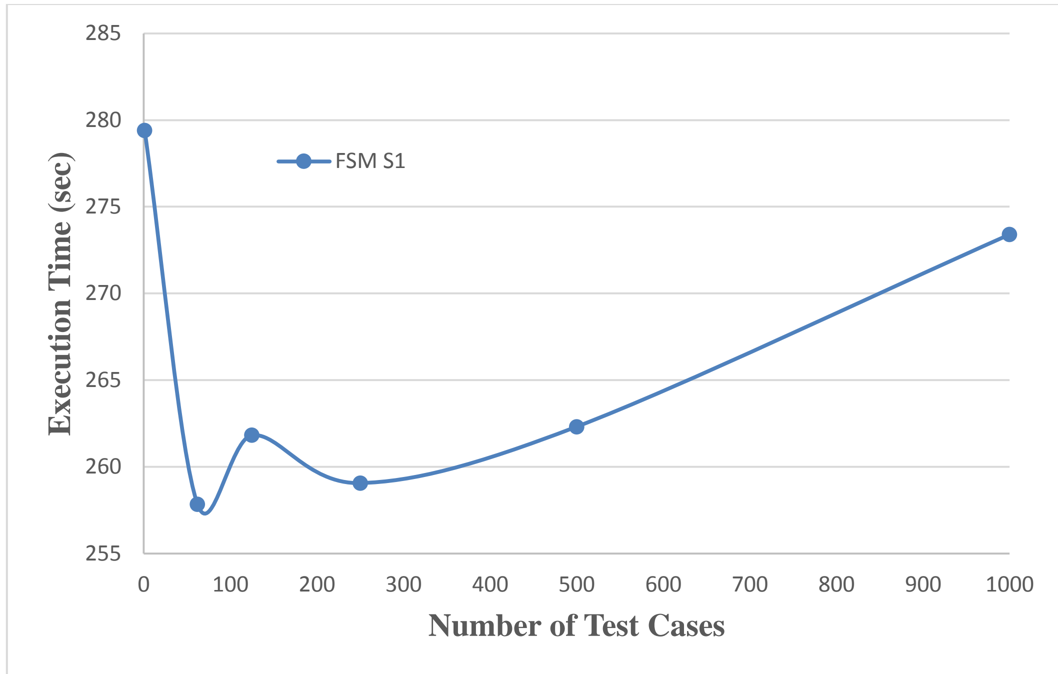

Figure 4-3 Sequential execution time versus number of test cases (FSM S_1)

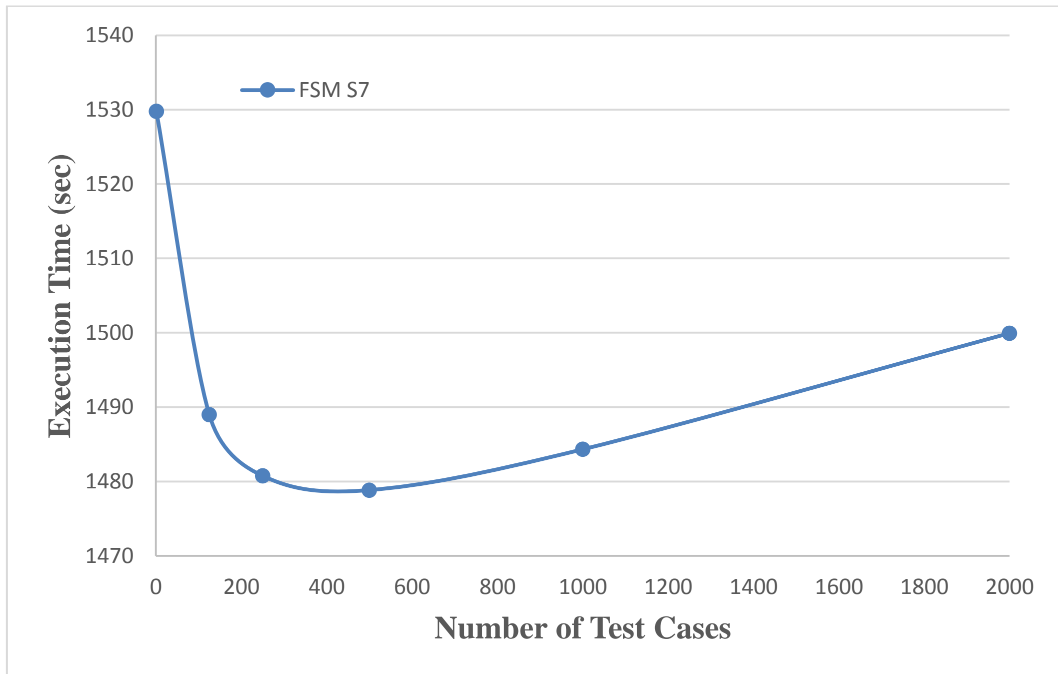

Figure 4-4 Sequential execution time versus number of test cases (FSM S_7)

4.2 Experiments with Randomly Generated FSMs

4.2.1 Sequential versus parallel implementations. The experiments with randomly generated FSMs are designed to illustrate what can be achieved with the

current state-of-the-art. In this section, the performance obtained by the sequential algorithm is compared against the parallel OpenMP, MPI and CUDA implementations.

Because of the extremely large execution time required by the sequential algorithm for medium and above sized FSMs, the testing is limited on FSM S_1.

Figure 4-5 depicts the execution time for the sequential algorithm as well as the parallel implementations as the number of mutants increases. According to these experiments, the execution time of the sequential algorithm increases almost exponentially as the number of considered mutants increases. When the execution time of the sequential algorithm reaches 2 hr and 10 minutes, OpenMP, MPI and CUDA execution times does not exceed 940, 252 and 10.6 seconds, respectively.

Figure 4-6 represents the speedup achieved by the OpenMP, MPI and CUDA implementations. On average, the OpenMP speedup is 8.3x, the MPI speedup is 30.9x while the CUDA one was significantly higher at 637.5x.

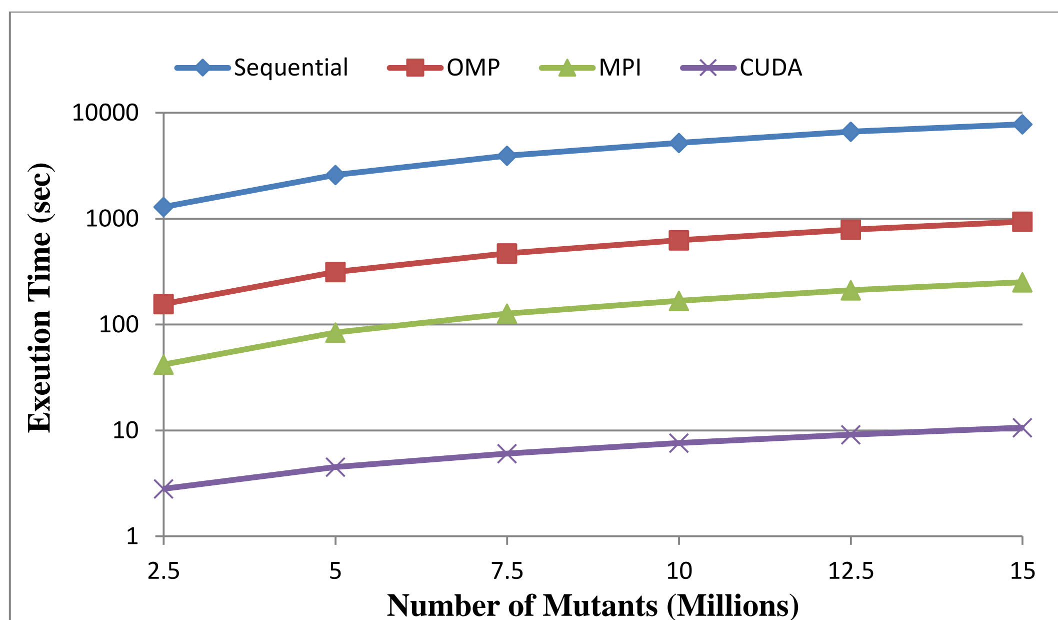

Figure 4-5 Sequential versus parallel implementations execution time (Machine S_1)

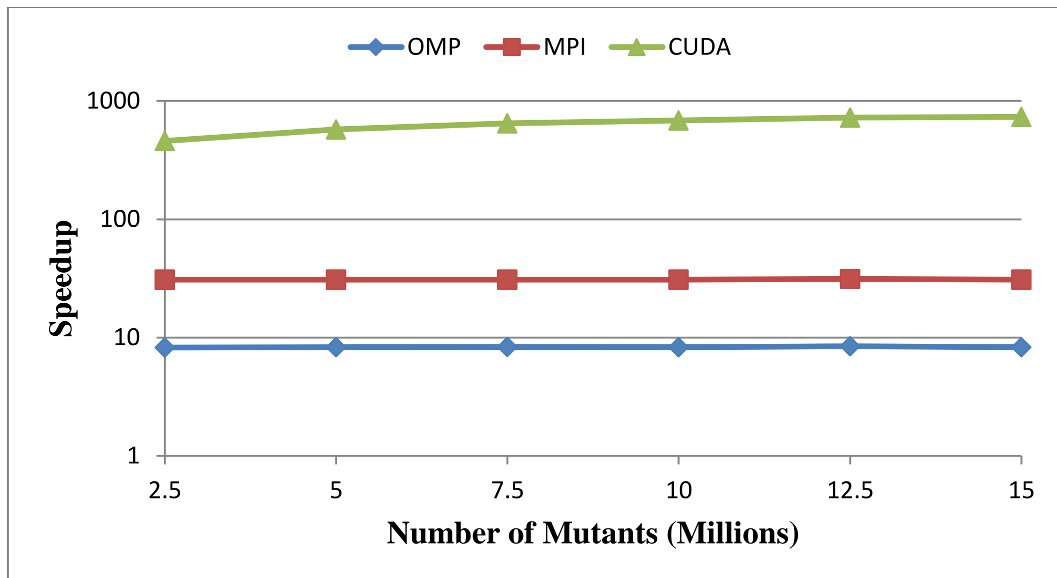

Figure 4-6 Speedup against sequential algorithm (Machine S_1)

4.2.2 Speedup of MPI and CUDA relative to OpenMP. In this section, the execution time of OpenMP, MPI and CUDA, and the relative speedup of MPI and CUDA over OpenMP are assessed. As in the previous subsection, FSM S_1 is used to show that the parallel implementations can scale more than the sequential algorithm as the number of considered mutants increases.

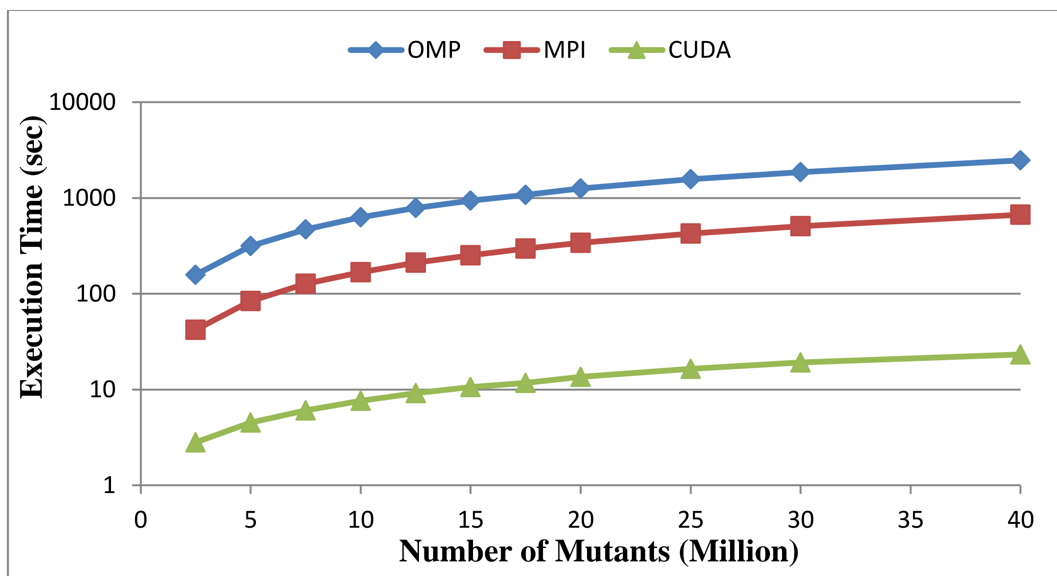

Figure 4-7 Execution time for the parallel implementations (Machine S_1 in table 4-1)

Figure 4-7 depicts the execution time as the number of mutants increases. According to this figure, the CUDA implementation is much faster than the OpenMP one. When the number of mutants equals 50 million, the execution time of OpenMP is 52.2 minutes while that of MPI and CUDA is only 841 and 28.7 seconds, respectively.

The speedup analysis for this experiment is presented in Figure 4-8. On average, the achieved speedup of MPI and CUDA over OpenMP is 3.7x and 87.7x, respectively.

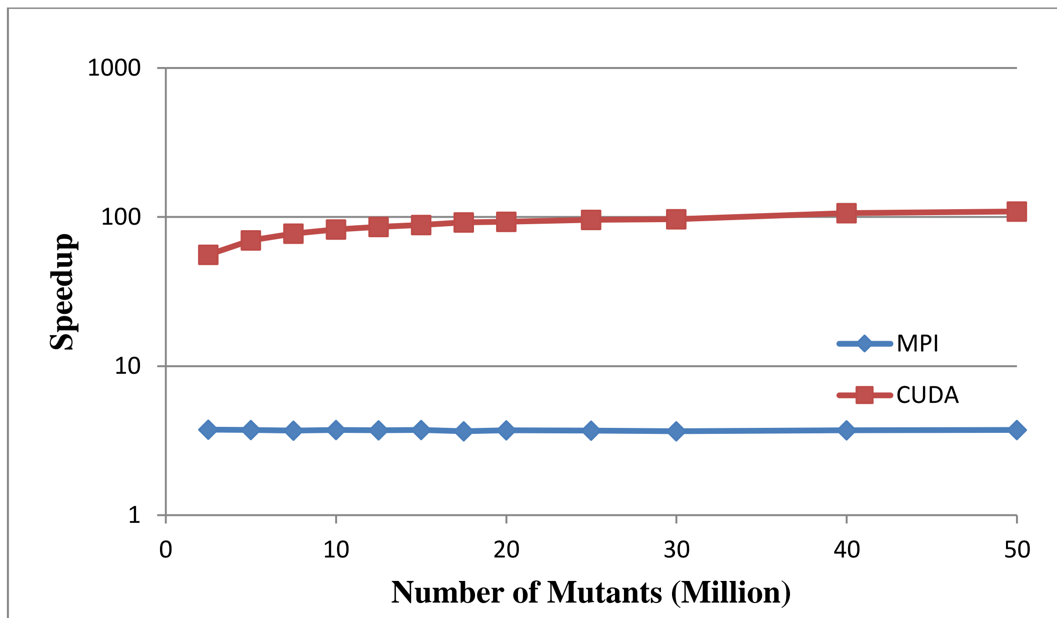

Figure 4-8 MPI and CUDA speedup against OpenMP (Machine S_1 in table 4-1)

4.2.3 Speedup of CUDA relative to MPI. In this section, the execution time of MPI and CUDA is assessed. In the conducted experiments, FSM S_7 from Table 4-1 is used. The testing expanded to larger FSM since FSM S_1 is small and can't assess the power of MPI parallel implementation.

Figure 4-9 depicts the execution time as the number of mutants increases. According to this figure, the CUDA implementation is much faster than the MPI one. When the number of mutants equals 50 million, the execution time of MPI is 152 minutes while that of CUDA is only 2.6 minutes, i.e. CUDA is 58.6x faster than MPI.

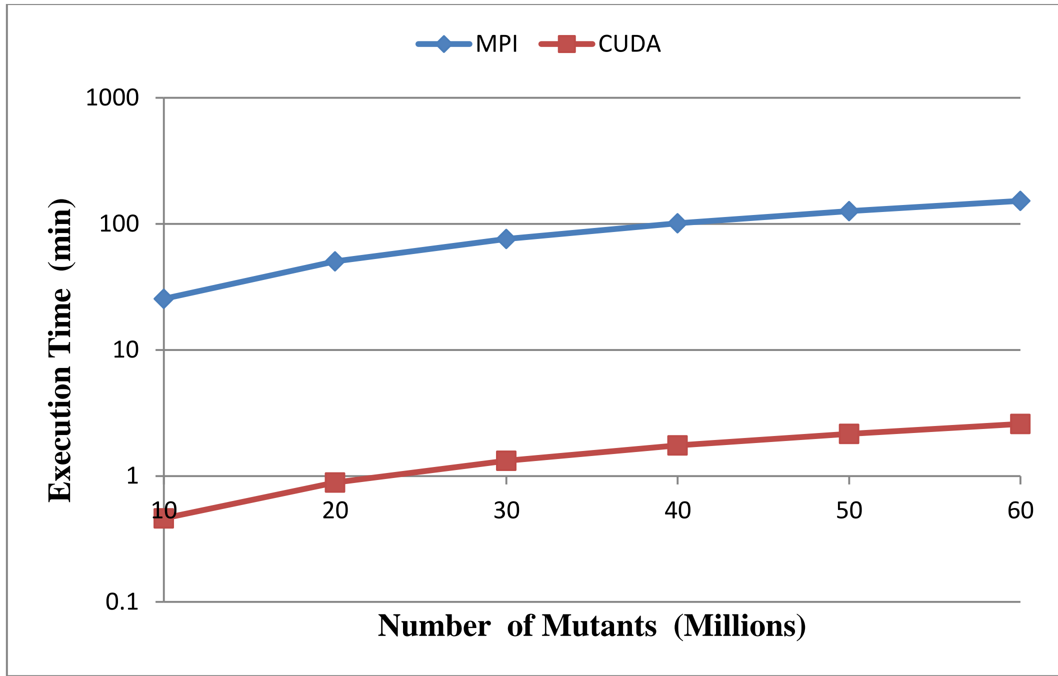

Figure 4-9 Execution time for MPI and CUDA parallel implementations (Machine S_7)

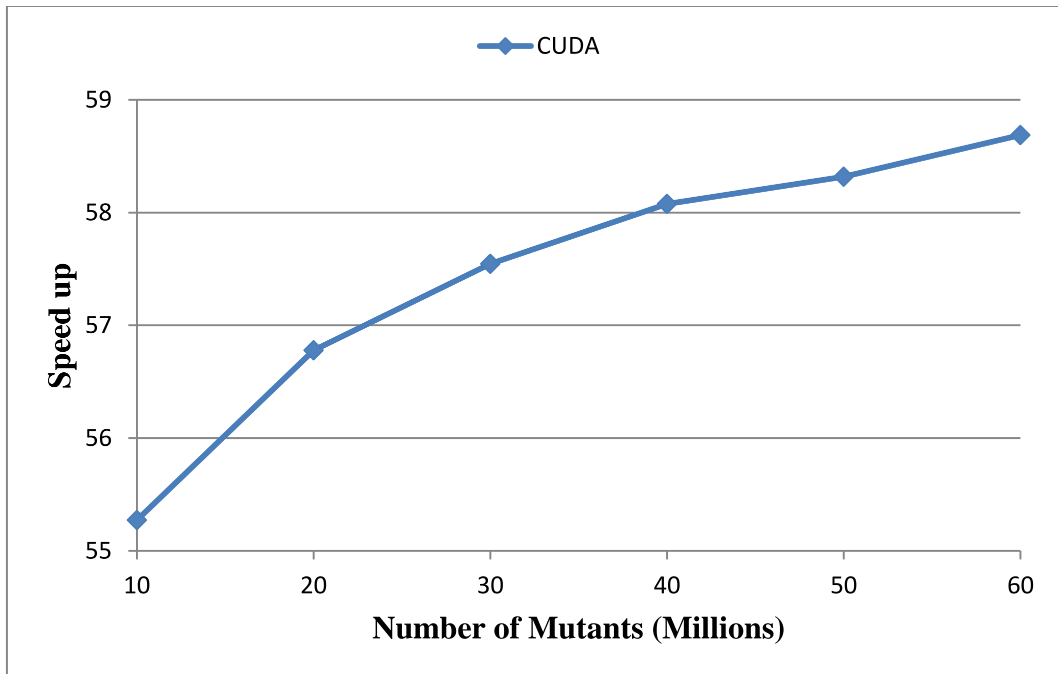

Figure 4-10 CUDA speedup against MPI (Machine S_7)

The speed up analysis for this experiment is presented in Figure 4-10. On average, the achieved speedup of CUDA over MPI is 57.4x.

4.2.4 CUDA Scalability. In this section, the CUDA solution is analyzed relative to how it can scale in terms of handling bigger FSMs and larger number of mutants. Accordingly, S_8, S_9, and S_{10} (see Table 4-1) are utilized. In these experiments, test suites of only 1 test case are used, each of length 750,000. *TS* was calculated as the average number of transitions of the considered FSMs.

According to the results depicted in Figure 4-11, the CUDA implementation scales almost linearly. The number of mutants tested, reach the limit on what is currently possible to fit on the majority of GPU accelerators' memory. The 500 million mutants require a total of approximately 8GB of RAM for storage.

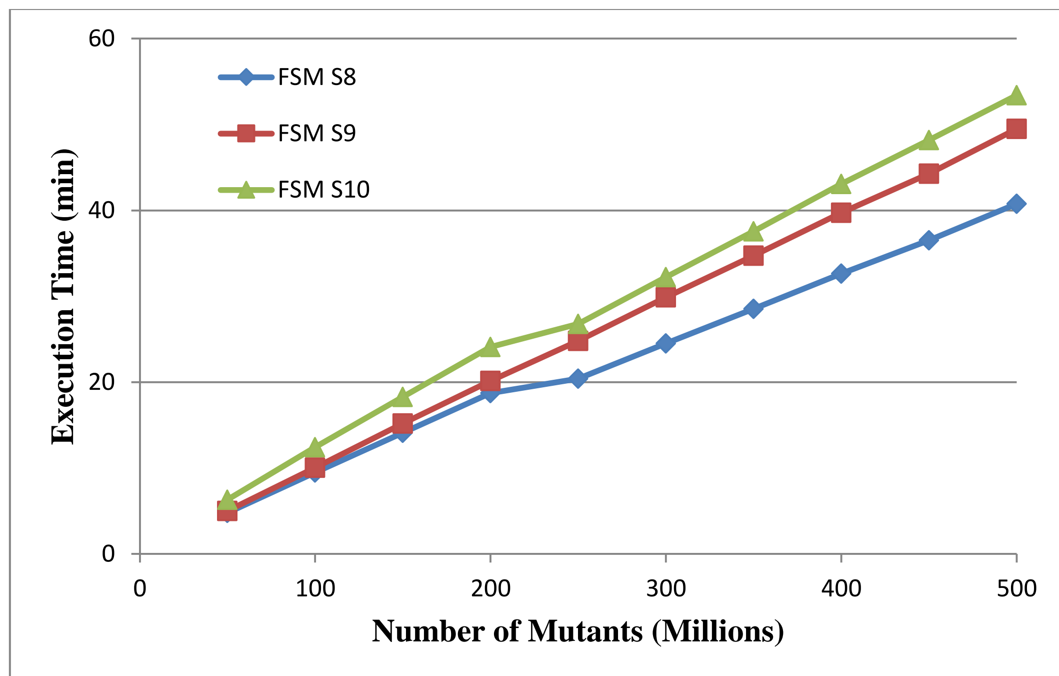

Figure 4-11 CUDA massive test (randomly generated machines)

4.3 Experiments with Real Application FSMs

4.3.1 Sequential versus parallel implementations. The experiments with randomly generated FSMs are designed to illustrate what can be achieved with the current state-of-the-art. In this section, the performance obtained by the sequential algorithm is compared against the parallel OpenMP, MPI and CUDA implementations.

Because of the extremely large execution time required by the sequential algorithm for medium and above sized FSMs, the testing is limited on FSM nucpwr (see Table 4-2).

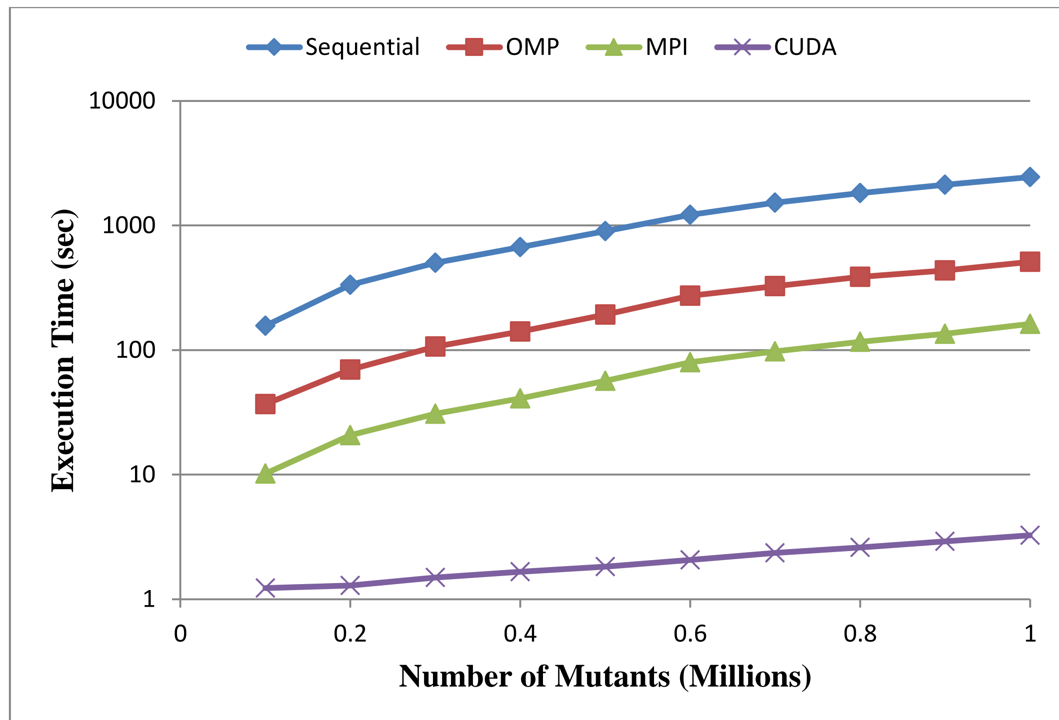

Figure 4-12 Sequential versus parallel implementations execution time (Machine nucpwr)

Figure 4-12 depicts the execution time for the sequential algorithm as well as the parallel implementations as the number of mutants increases. According to these experiments, the execution time of the sequential algorithm increases almost exponentially as the number of considered mutants increases. When the execution time of the sequential algorithm reaches 40.7 minutes, OpenMP, MPI and CUDA execution times does not exceed 509, 162 and 3.2 seconds, respectively.

Figure 4-13 represents the speedup achieved by the OpenMP, MPI and CUDA implementations. On average, the OpenMP speedup is 4.6x, the MPI speedup is 15x while the CUDA one was significantly higher at 502x.

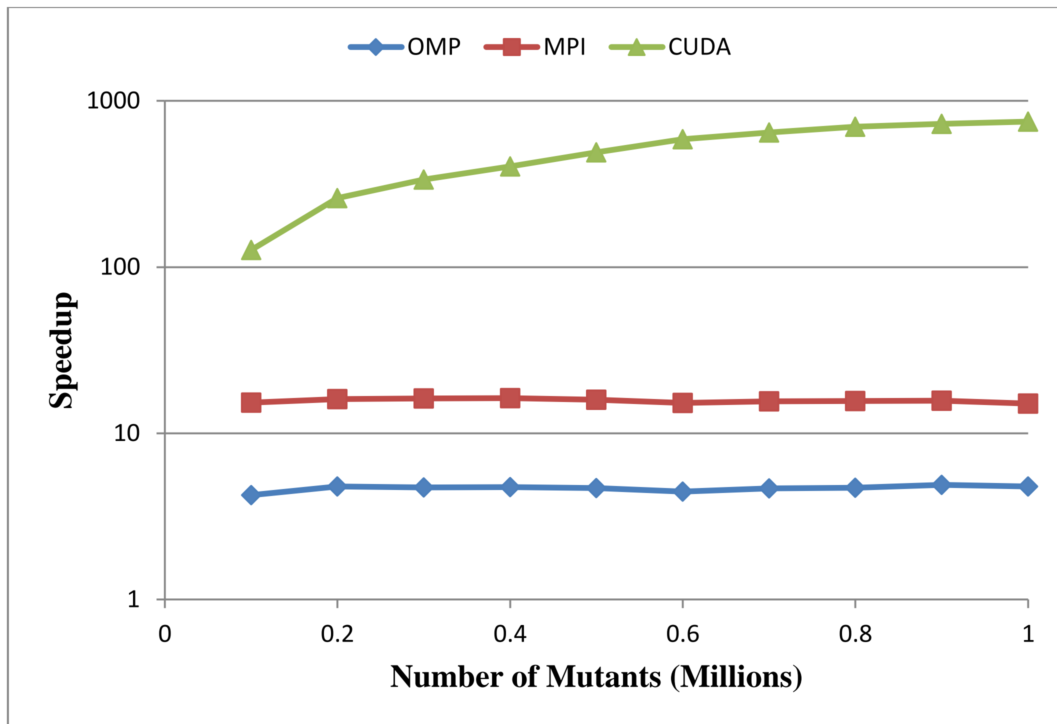

Figure 4-13 Machine nucpwr speedup analysis

4.3.2 Speedup of MPI and CUDA relative to OpenMP. In this section, the execution time of OpenMP, MPI and CUDA, and the relative speedup of MPI and CUDA over OpenMP are assessed. As in the previous subsection, FSM nucpwr from Table 4-2 is used to show that the parallel implementations can scale more than the sequential algorithm as the number of considered mutants increases.

Figure 4-14 depicts the execution time as the number of mutants increases. According to this figure, the CUDA implementation is much faster than the OpenMP one. When the number of mutants equals 10 million, the execution time of OpenMP is 2 hr and 18 minutes while that of MPI and CUDA is only 41.6 minutes and 34.6 seconds, respectively.

The speed up analysis for this experiment is presented in Figure 4-15. On average, the achieved speedup of MPI and CUDA over OpenMP is 3.4x and 156x, respectively.

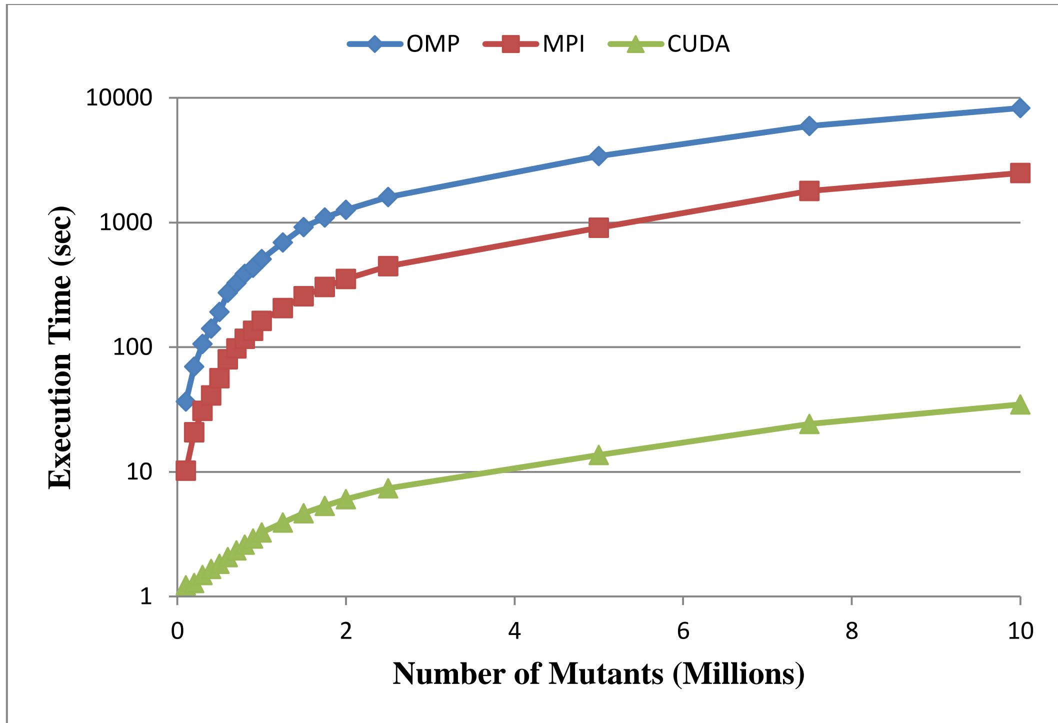

Figure 4-14 Parallel implementations execution time (FSM nucpwr in table 4-1)

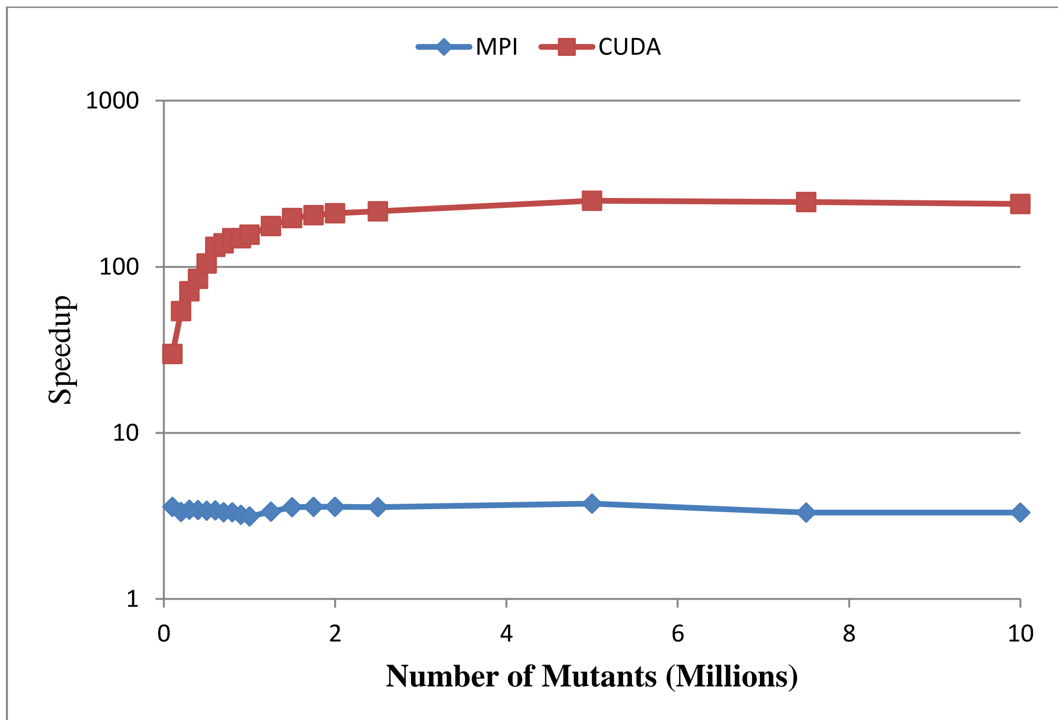

Figure 4-15 MPI and CUDA speedup against OpenMP (FSM nucpwr in table 4-1)

4.3.3 Speedup of CUDA relative to MPI. In this section, the execution time of MPI and CUDA is assessed. In the conducted experiments, FSMs ram_testO, s820o, s8320 and s420O from Table 4-2 were used. The testing expanded to larger FSMs since FSM nucpwr is small and can't assess the power of MPI parallel implementation. In these experiments, *TS* with fixed length equals to the (average number of transitions for these FSMs over 16) is used. This TS is used since the considered FSMs are relatively extra-large with average machine size equal to 6.5 million transitions.

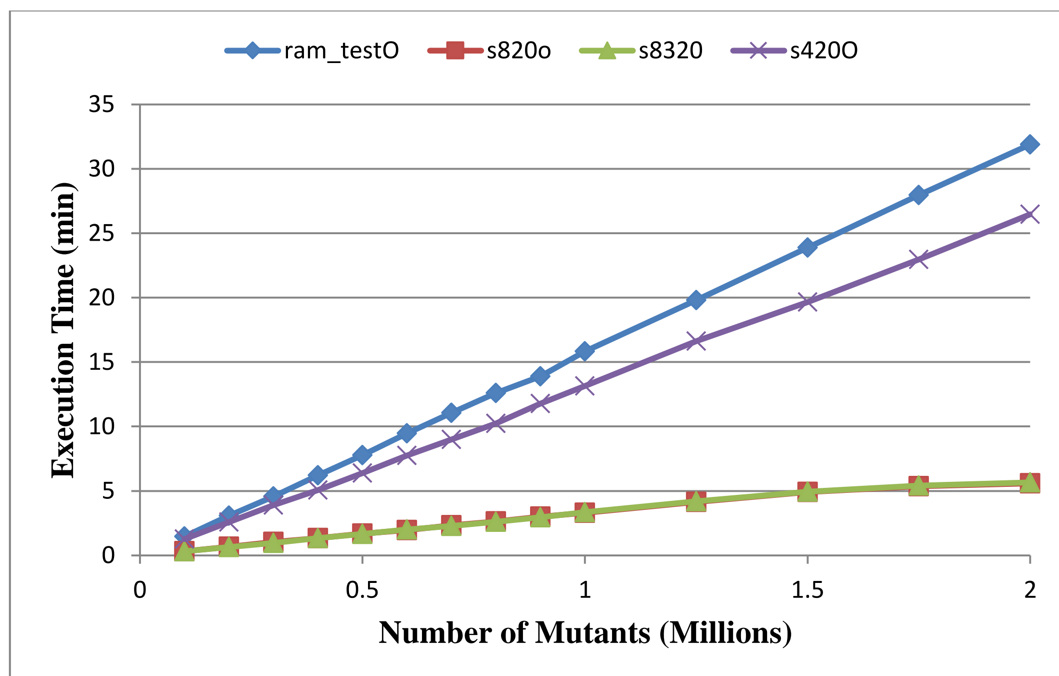

Figure 4-16 MPI execution time for big real application FSMs

Figure 4-16 depicts the execution time as the number of mutants increases. According to this figure, the MPI implementation can work on large and extra-large FSMs with a reasonable execution time.

The speed up analysis for this experiment is presented in Figure 4-17. On average, the achieved speed for CUDA over MPI is 134.8x, 27.9x, 27.8x and 109.8x for FSMs ram_testO, s820o, s8320 and s420O, respectively. It can be seen from the graph that increasing the number of mutants will increase the CUDA speedup.

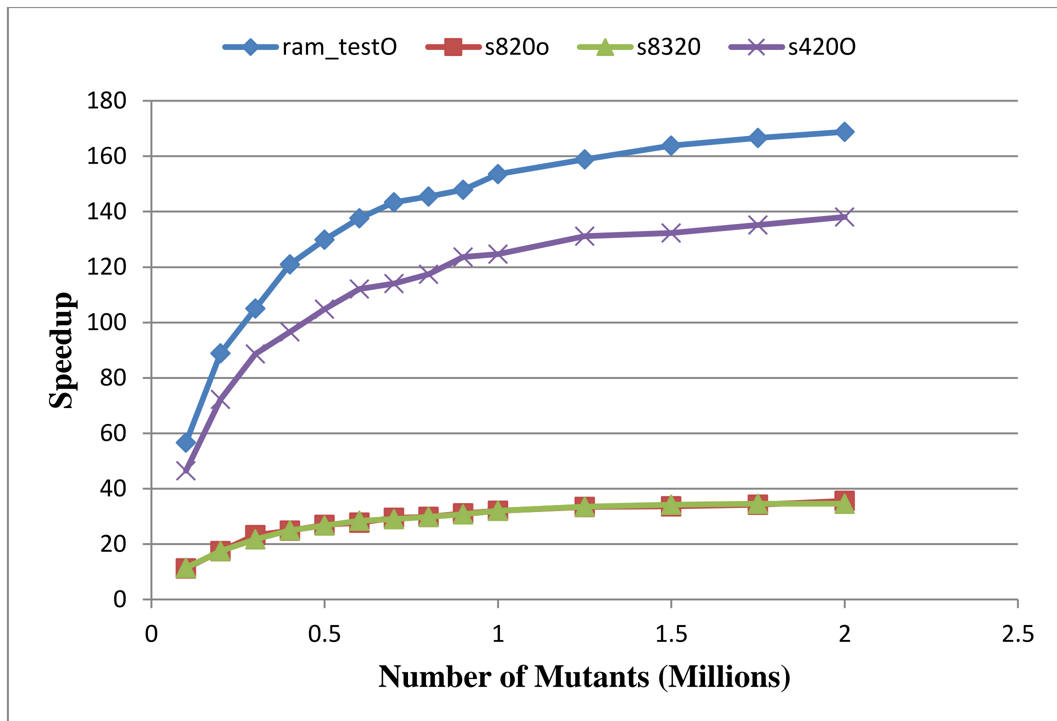

Figure 4-17 CUDA speedup against MPI in big real application FSMs

4.3.4 CUDA Scalability. In this section, the CUDA solution is analyzed relative to how it can scale in terms of handling bigger FSMs and larger number of mutants. Accordingly, FSMs ram_testO, s820o, s8320 and s420O from Table 4-2 are utilized. As illustrated above (see 4.3.3), test suites of only 1 test case are used, each of length 425,984. TS was calculated as (the average number of transitions of the considered FSMs/16).

Figure 4-18 the CUDA implementation scales almost linearly. The number of mutants tested, reach the limit on what is currently possible to fit on the majority of GPU accelerators' memory. The 250 million mutants require a total of approximately 4GB of RAM for storage, without considering the memory needed for the biggest of the test FSMs which reach a size of 9.7 million transitions.

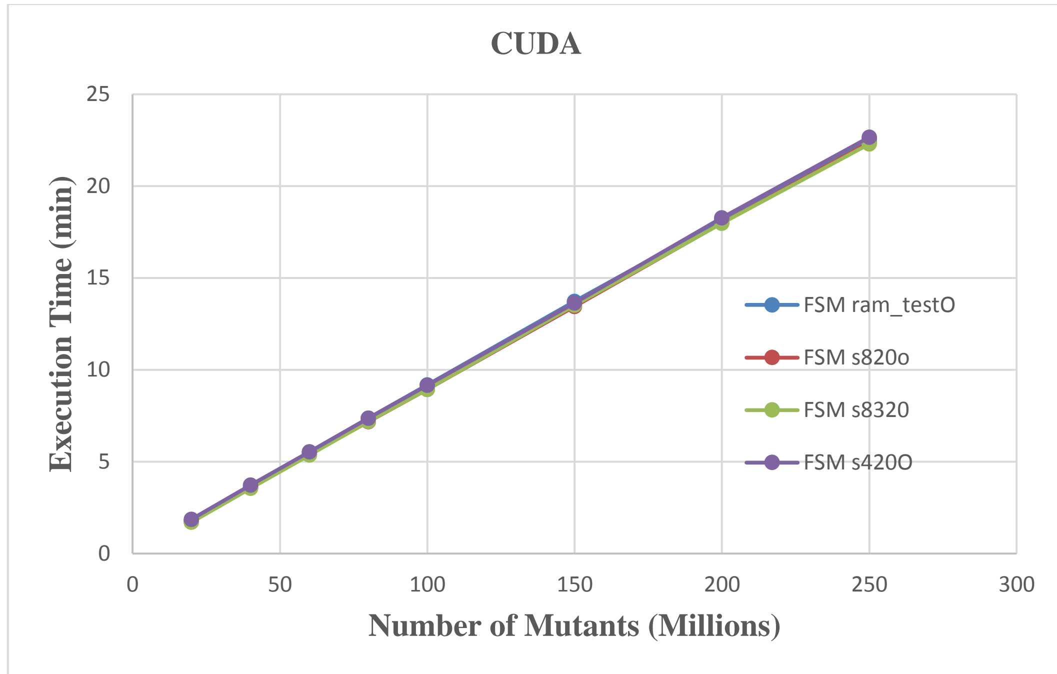

Figure 4-18 CUDA massive test (real FSMs application examples)

4.4 Analysis

In this section, attributes related to the considered mutants' elimination problem are assessed. More precisely, the impact of machine size on execution time, the impact of number of threads in OpenMP, the impact of number of processes in MPI and the impact of CUDA RunSize on speedup. This analysis is conducted using randomly generated machines. Also, the obtained analysis using real application FSMs is confirmed on randomly generated machines with the same attributes.

4.4.1 Execution time considering machine size. The sequential implementation was used to determine the impact of the machine size as well as the number of mutants over execution time. These corresponding experiments were conducted using the FSMs S_1 to S_7 from Table 4-1.In these experiments, the used test suites length equals 125,000 calculated as the average number of transitions of the considered FSMs. As before, each test suite includes n/2 test cases. The number of mutants fixed to 4 million.

Figure 4-19 depicts the execution time as machine size increase. According to the obtained results, the execution time of the sequential algorithm increases exponentially as the machine size increases.

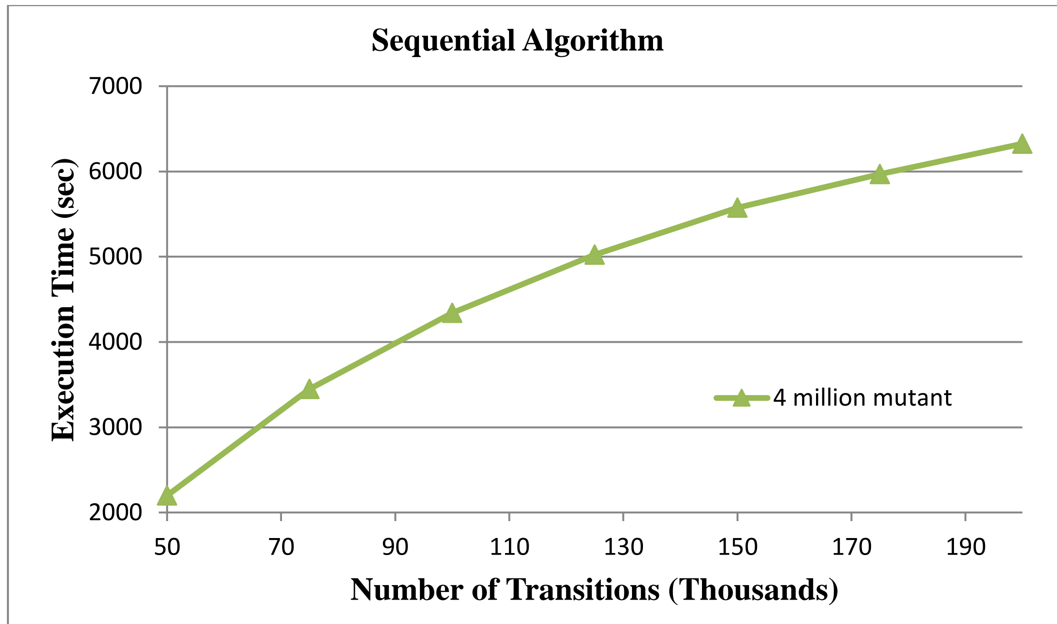

Figure 4-19 Number of transitions versus execution time (Machines S_1-S_7)

4.4.2 Number of Threads, Processes and Run Size versus Speedup. The parallel implementations were used to determine the impact of the parallel implementations attributes overs execution time. Namely, the number of invoked threads in OpenMP, the number of launched process in MPI and CUDA RunSize $|\alpha_c|$. These corresponding experiments were conducted using FSMs S_1 to S_7. illustrated above (see 4.4.1), test suites of length equal 125,000 each has $n/2$ test cases is used. The number of considered mutants is fixed to 1 million

Figure 4-20 depicts the execution time as the number of invoked threads increases. According to this figure, the OpenMP speedup increases as the number of invoked threads increases. However, as the machine size increases, increasing the number of invoked threads will cause a speedup drop. The number of invoked threads in the conducted experiments is the number of logical cores in the used computer.

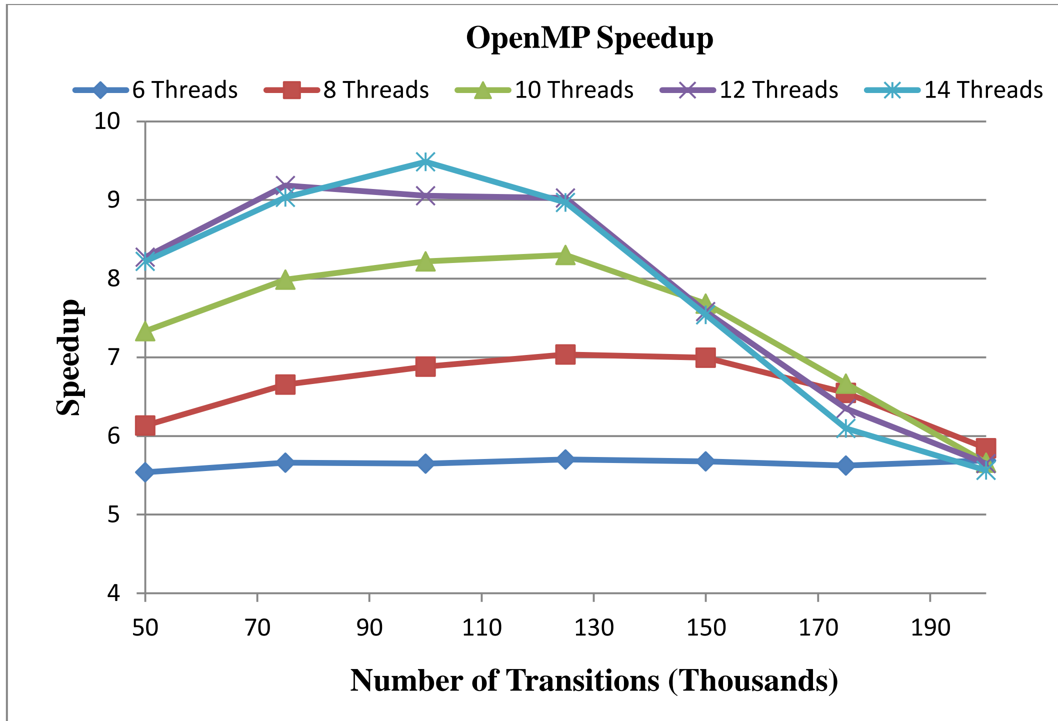

Figure 4-20 OpenMP speedup versus number of threads (Machines S_1-S_7)

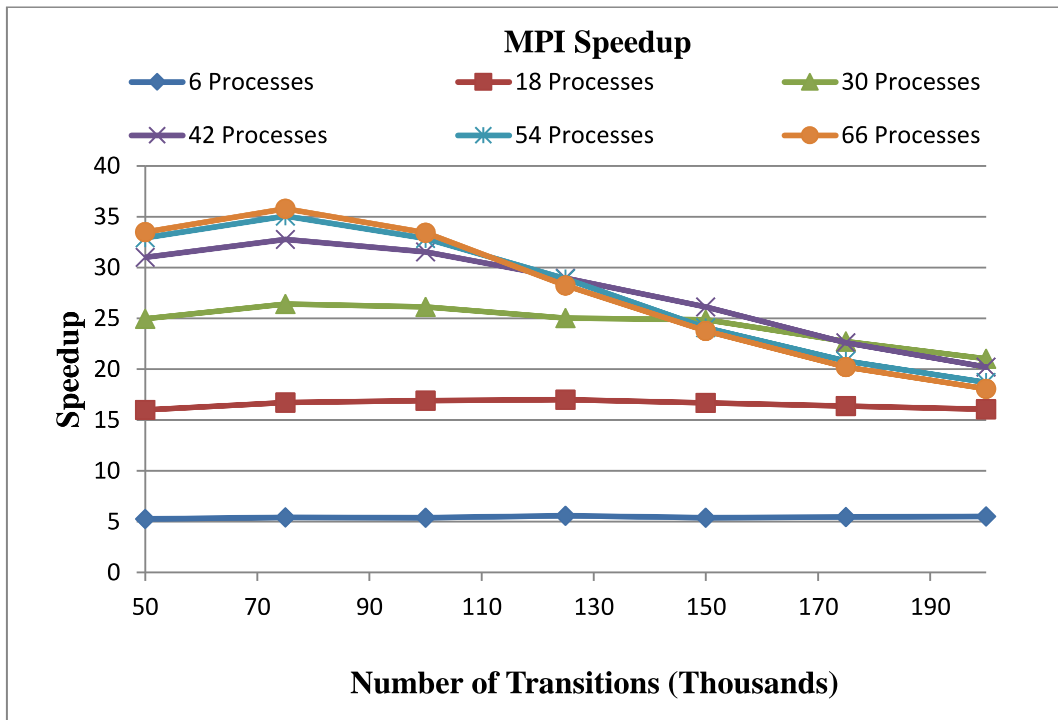

Figure 4-21 MPI speedup versus number of processes (Machines S_1-S_7)

Figure 4-21 depicts the execution time as the number of launched processes increases. According to this figure, the MPI speedup increases as the number of launched processes increases. However, as the number of launched processes exceeds 42 and approaches the number of logical cores in the used NoW the speedup starts to drop for large machines. The number of available logical cores in the used test platform is 52 cores, and the number of launched processes is 42 processes.

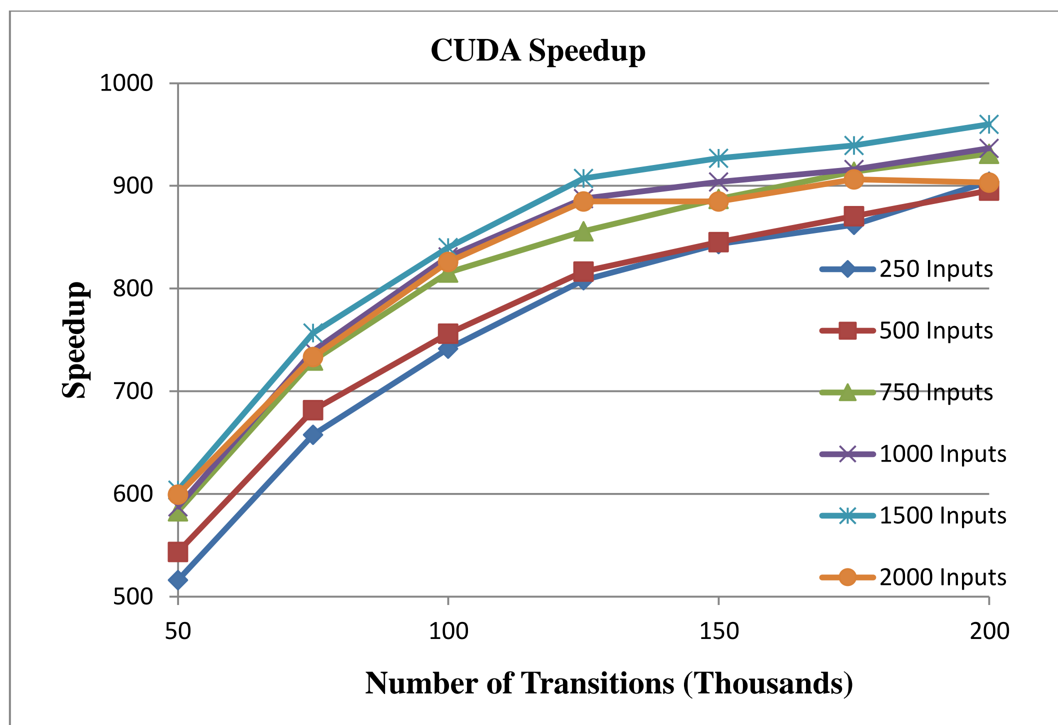

Figure 4-22 CUDA speedup versus Run size (Machines S_1-S_7)

Figure 4-22 depicts the execution time as CUDA *RunSize* increases. According to this figure, the CUDA speedup increases as the *RunSize* increases. However, as *RunSize* starts approaching 2000, the speedup starts to decline. In the conducted experiments, a *RunSize* equal to 1000 is used.

4.4.3 Randomly generated FSMs versus real applications FSMs. The sequential and parallel implementations were used to confirm the obtained analysis over randomly generated FSMs described in the previous section. To this end, some real application FSMs are considered and then for each considered FSM, a randomly

generated FSM with the same attributes (number of states, inputs, outputs) is derived. Then, an analysis similar to the ones described above using the derived FSM is conducted to compare the obtained results with those of the real application FSMs. In these experiments the attributes of the real application FSMs s208, nucpwr, ram_testO and s820o (see Table 4-2), are used to derive corresponding randomly generated FSMs with the same attributes, hereafter named Random-s208, Random nucpwr, Random ram_testO and Random-s820o, respectively.

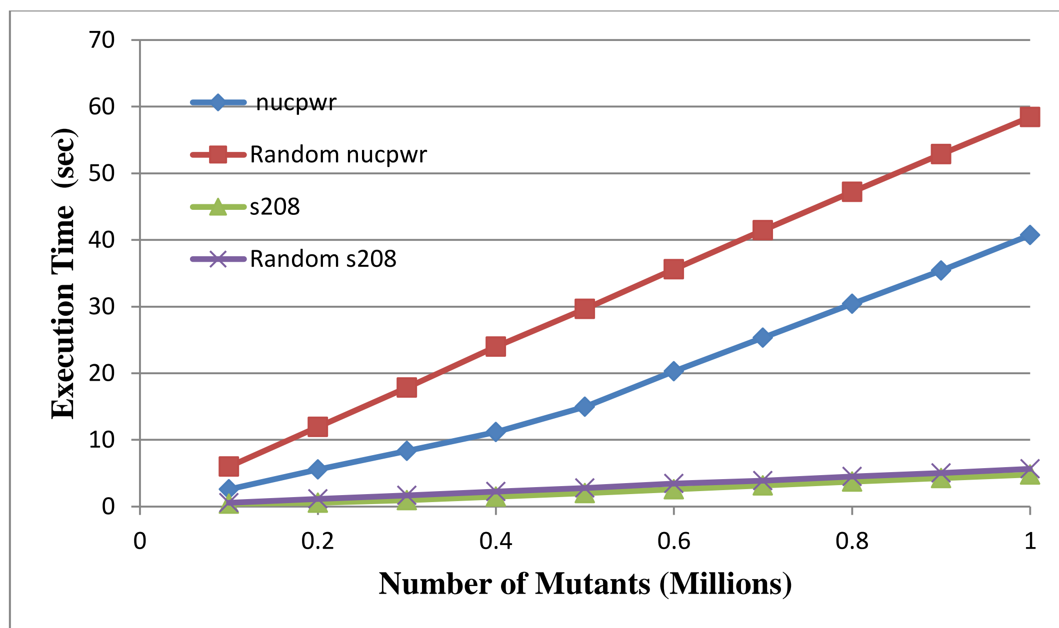

Figure 4-23 Sequential execution time of real applications FSMs versus. raandom FSMs with the same attributes

Figure 4-23 depicts the execution time for the sequential algorithm for FSMs s208, nucpwr, Random-s208 and Random-nucpwr as the number of mutants increases. According to this figure, real application FSMs execution time has the same pattern as the randomly generated FSMs. However, real application FSMs execution time is less than that of the randomly generated FSMs. On average, the execution time for FSMs s208 and nucpwr is 0.776x and 0.599x of that of Random-s208 and Random-nucpwr, respectively. This difference in the execution time happens since the real application FSMs has a unique output and next state pattern, in which the used randomly generator cannot mimic to derive similar random FSMs.

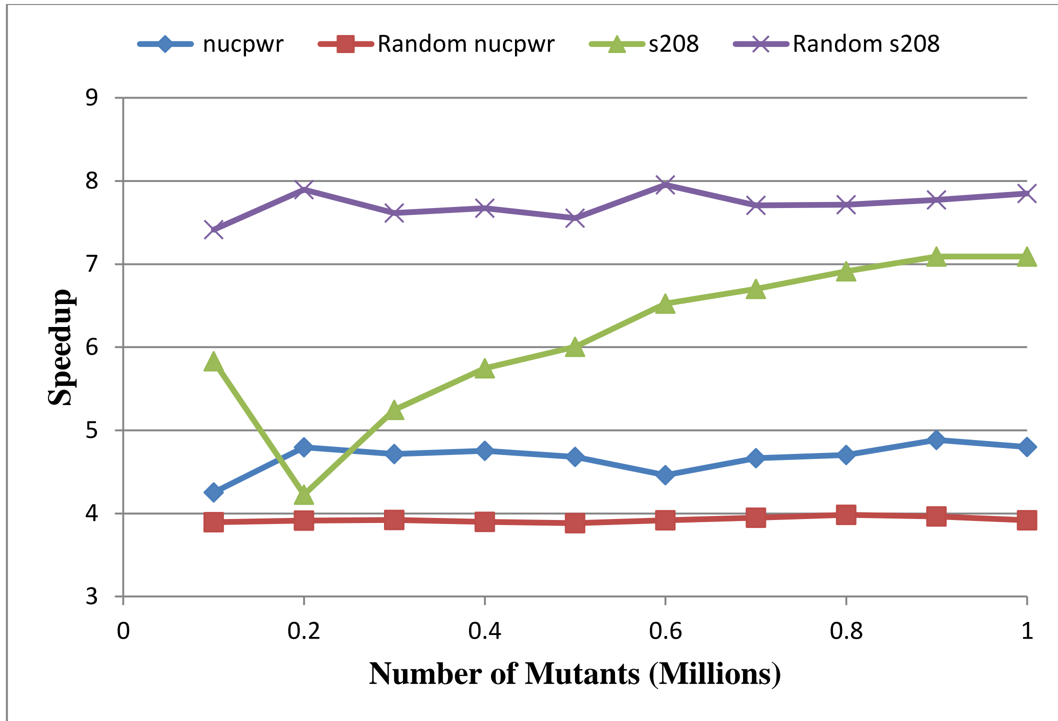

Figure 4-24 OpenMp speedup of real applications FSMs versus random FSMs with the same attributes

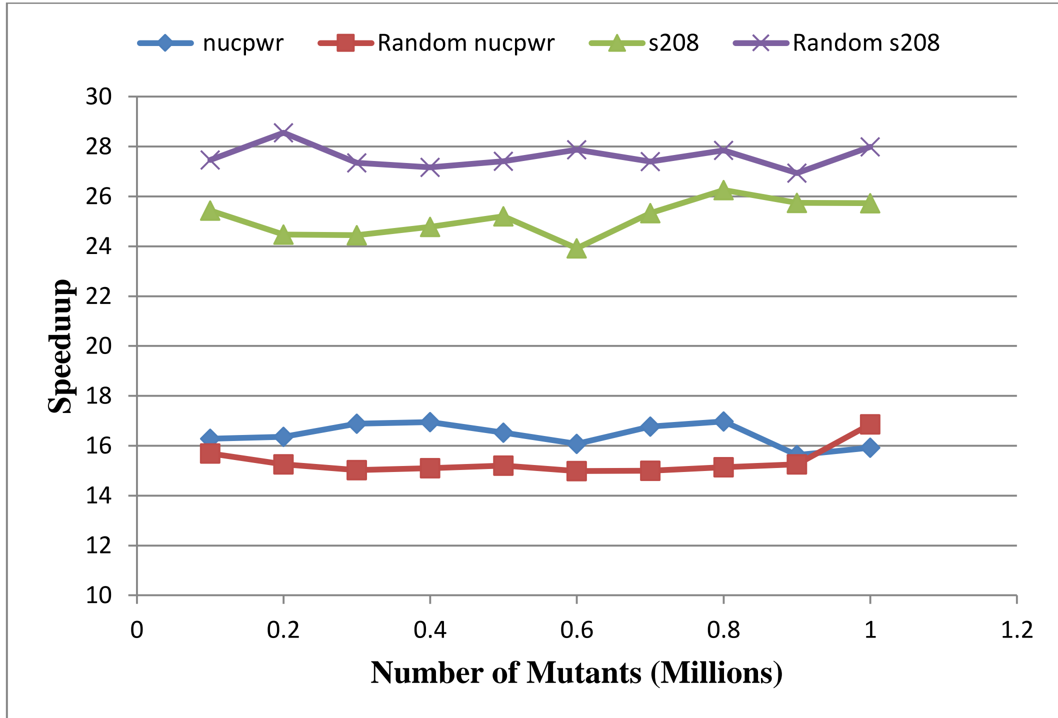

Figure 4-25 MPI speedup of real applications FSMs versus random FSMs with the same attributes

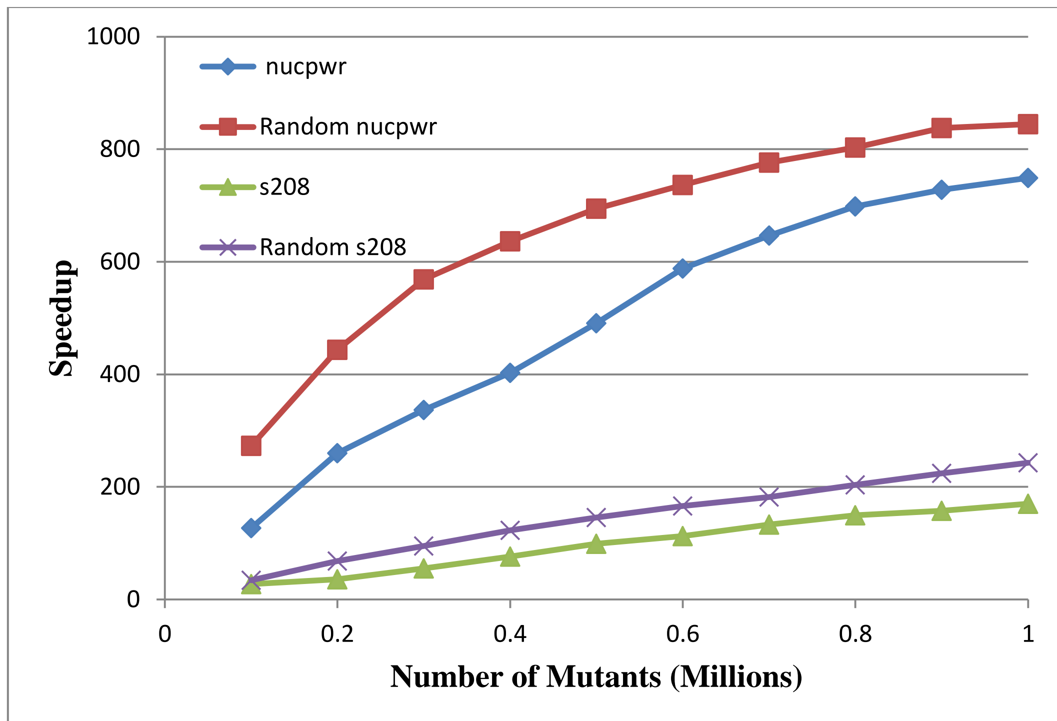

Figure 4-26 CUDA speedup of real applications FSMs versus random FSMs with the same attributes

Figure 4-24, Figure 4-25 and Figure 4-26 include the speedup analysis. According to these figures, both real application FSMs and randomly generated FSMs have the same speedup pattern. In OpenMP and MPI, nucpwr speedup is higher than that of Random-nucpwr, while s208 speedup is less than that of Random-s208. In CUDA, both randomly generated FSMs obtained better speedup than the real application FSMs.

Extra experiments using CUDA implementation were conducted using the large and X-large FSMs ram_testO, s820o, Random-ram_testO, and Random-s820o. In these experiments test suites of only 1 test each of length 425,984 calculated as the average number of transitions of the considered FSMs over 16 were used. The number of considered mutants ranges from 2.5 to 22.5 million.

According to the results depicted in Figure 4-27, large and X-large real application FSMs have the same pattern as the randomly generated FSMs. Moreover, the execution time of ram_testO and Random-ram_testO are approximately the same.

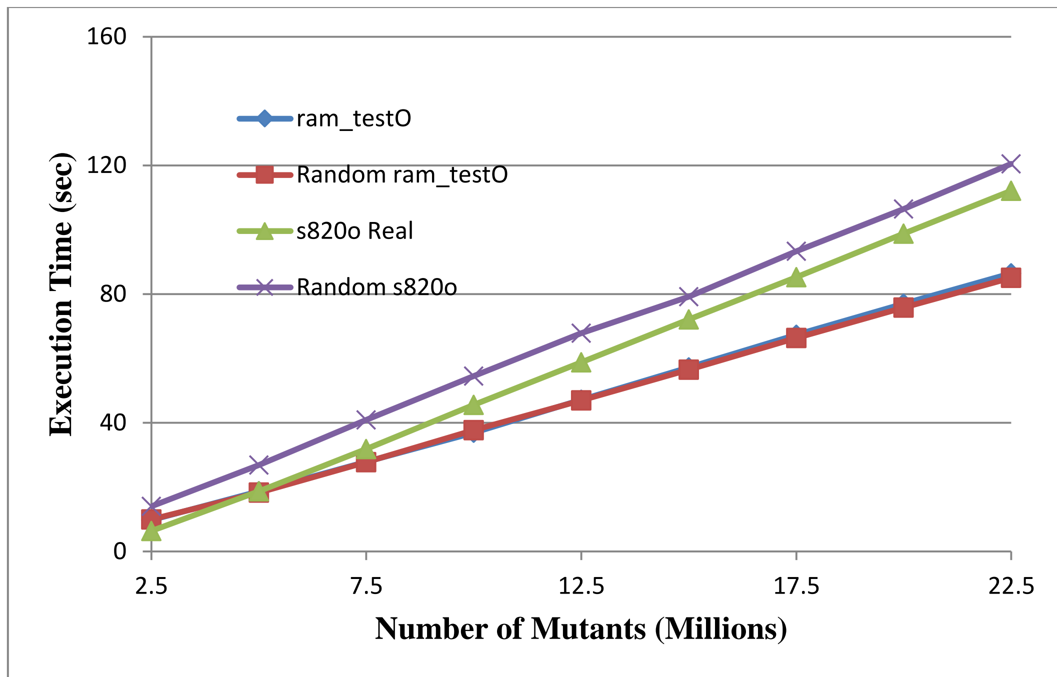

Figure 4-27 CUDA execution time of real applications FSMs versus. random FSMs with the same attributes

Chapter 5 : Related Work and Literature Review

As described in the Introduction of this work, i.e. the process of eliminating mutants, can be used in various FSM-based testing problems such as in mutation testing, fault diagnosis, and studying the effectiveness of test suites.

In general, methods, tools, application areas, and summaries, related to mutation testing and studying the effectiveness of test suites, are covered in the traditional textbooks by Mathur [55], Ammann and Offutt [50], and in Jia and Harman [37] survey. In particular, for FSMs, Fabri et al. [85] proposed FSM related mutation operators and Febri et al. [86] and Simao et al. [60] described tools used in this context.

Ghedamsi et al. proposed many FSM based fault diagnosis methods. They proposed diagnostic tests for systems represented as FSMs in [61, 64-66] , as well as for communicating FSMs [62, 63]. These methods are used in El-Fakih et al. [59] in the context of extended FSM based fault diagnosis. El-Fakih et al. [28] presented a method to determine if it is possible to locate faulty component FSM, and derive tests for locating faults if it is possible to locate them. El-Fakih et al. [87] also followed a similar approach for deriving an adaptive diagnostic test suite for a single Extended FSM (EFSM). Pap et al. [88] studied diagnosis of a single transition or output fault in an FSM.

For studying the effectiveness of extended FSM test suites, recently El-Fakih et al. [59] used the FSM-based mutation testing tools, such as those in Simao et al. [60], in studying the effectiveness of extended FSM test suites. El-Fakih et al. [89] also presented a fault coverage assessment of several EFSM based test suites. Habib in [90], worked on assessing extended finite state machine (EFSM)-based and random TS fault coverage as well as extended FSM-based fault localization capabilities. It is clear that the proposed implementations used in this work, can be used to enhance the time performance which thus enhance scalability (in terms of considering bigger fault domains) of the previous related work.

As a result, for the advancement of parallel programming machines and the publication of easy-to-use libraries that support the parallel software development, many researchers used the aforementioned technique to speed up their testing process.

Recently, many researches, including this research, worked on utilizing the execution power of recent parallel technologies in the context of solving some FSM-based testing problems. For example, El-Fakih et al. [91] considered systems specified as non-deterministic FSMs, and proposed and assessed implementations based on multicore CPUs and many-core GPUs for the derivation of all the successors of all pairs of states of an FSM. This is done in order to reduce the time and efforts for deriving sequences for distinguishing nondeterministic FSMs. Haddad et al. [92] also considered non-deterministic FSMs, and proposed a parallel multithreaded implementation, using Open Multi-Processing, for deriving distinguishing states of an FSM.

Hierons and Turker in [93] worked on improving the computational complexity of previous existed algorithm for generating Characterizing Sets (CSs) which is a set of input sequences that distinguishes all pairs of states, and Harmonized State Identifiers (HSIs) which allow different sets of input sequences for different states is an improvement on CSs- from partial FSMs which have exponential worst case time complexity. In their work, they tackled the scalability from two directions; the invention of new polynomial time sequential algorithm for generating CSs with massively parallel implementation of their algorithm and devising a parallel HSI construction algorithm. Moreover, they paralyzed the brute-force algorithms for generating CSs and HSIs (based on previous work done in [94]). By exploiting the GPU power to paralyze CSs and HSIs generation, they invented new faster and more scalable algorithm. Also, they found that the parallel version of the previous brute force algorithm outperformed the newly developed polynomial one. As for the parallel version of their new algorithm, it was the best in terms of time complexity, but it did not scale very well like the brute force due to its memory requirements.

Hierons and Turker [5] also employed GPUs to automatically derive Unique Input Output sequences (UIOs) which is an input sequence that can distinguish a state s_0 from the rest of inputs in FSM S from Finite State Machines. They addressed the scalability problem that may arise while constructing UIOs for completely specified FSMs through the use of massively parallel GPU technology. The new parallel UIO algorithm utilizes the capability of more than GPU and distributes the workload between them.

At the end, it is worth mentioning that many researchers used parallel testing in different fields than ours. In the late 80's and early 90's, many researches were conducted on parallelizing test pattern generation for digital circuits and very large integrated circuits (VLSI). Bollinger and Midkiff [95] conducted an investigation of the theoretical available amount of parallelism in topologically partitioned parallel automatic test patterns generation (ATPG), which aims to find an upper bound of the parallelism amount present in conflict-& test generation. Chandra and Patel [96] introduced the concept of heuristic parallelization which includes ATPG. Also, Motohara in [97] used functional partitioning in which an algorithm is divided into independent sub tasks that can be executed in parallel. Klenke et al. [98, 99], presented a parallel architecture of a topologically ATPG on a distributed memory multiprocessor. Another parallel test generation method which tries to achieve high fault coverage for hard-to-detect (HTD) faults in a reasonable amount of time was proposed by Patil and Banerjee in [100].

Chapter 6 : Conclusion

Given a test suite of test cases usually from a given specification finite state machine (FSM) and fault domain, compromising mutants derived of the specification derived with respect to selected types of faults. The process of mutants' elimination deals with removing/killing mutants, of the fault domain, that have different behavior than the specification machine with respect to some test case of the test suite. Mutants' elimination is an essential step in FSM-based mutation testing, fault diagnosis, and in the assessment of the effectiveness of test suites. However, this process is time consuming especially with a huge number of considered mutants. Accordingly, three parallel implementations for reducing the time efforts of mutants' elimination are presented and assessed in this thesis. The first is an Open Multi-Processing (OpenMP) implementation that utilizes many cores in a single machine. The second uses a clustered system using Message Passing Interface (MPI) standard and the third is based on the Compute Unified Device Architecture (CUDA) that exploits the computation power of the Graphical Processing Unit (GPU).

Comprehensive experiments are conducted to assess the presented work using both randomly and real application FSMs. According to the obtained results the following holds:

- For the randomly generated machines, the speedup of OpenMP, MPI, and GPU against the sequential implementation equals 8.3, 30.9, and 637.5 times, respectively; while for the real application machines, this speedup equals 4.6, 15.0, and 502, respectively.

- For the experiments conducted using real application machines and randomly generated machines with the same attributes (i.e., number of states, inputs, and outputs) the following is obtained. For the randomly generated machines, the speedup of OpenMP, MPI, and GPU against the sequential implementation equals 5.8, 21.4, and 405, respectively; while for the corresponding real application machines, this speedup equals 5.4, 20.7, and 302, respectively. The difference is due to the fact that the execution time of the real application machines is less than that of the corresponding random machines since many

states of the real machines have the same output and next state pattern. However, it is worth mentioning that both the real and corresponding random machines have the same pattern in terms of an increase in execution time due to the increase in the number of mutants.

- The relative speedup of MPI and CUDA with respect to OpenMP, assessed using randomly generated machines, equals 3.7 and 87.7 times, respectively, while this speedup equals 3.43, and 156 for real application machines.

- For the experiments conducted using real application machines and randomly generated machines with the same attributes the following holds: The relative speedup of MPI and CUDA with respect to OpenMP, assessed using randomly generated machines, equals 3.69, and 69.7 times, respectively, while this speedup equals 3.8 and 55.9 for real application machines. This is due to the fact mentioned above. Here again, the same pattern in terms of an increase in execution time due to the increase in number of mutants is obtained for both types of machines.

- The relative speedup of CUDA with respect to MPI, assessed using randomly generated and real application machines equals 57.4 and 75.0 times, respectively.

- For the experiments conducted using real application machines and randomly generated machines with the same attributes the following holds: the relative speedup of CUDA with respect to MPI, assessed using randomly generated and real application machines equals 18.8 and 14.5 times, respectively. This is due to the fact mentioned above. Here again, the same pattern in terms of an increase in execution time due to the increase in number of mutants is obtained for both types of machines.

- CUDA implementation is scalable in terms of machine size and the number of considered mutants to eliminate. Limited by the used hardware architecture, CUDA easily handled experiments with 500 Million mutants and operated on machines with 9.5 Million transitions.

- In OpenMp, as the number of invoked threads increases, the speed up increases. As the number of threads approaches and exceeds the number of logical cores of

the used computer, the obtained speedup starts to decline for large machines size. Based on this, the number of threads used in the conducted experiments equals the number of logical cores (automatically determined by OpenMP) of the used computer.

- In MPI, as the number of launched processes increases, the speed up increases till the number of processes approaches the number of logical cores of the used NoW. At that moment, the obtained speedup starts to decline, especially with an increase in the machine size. Based on this, the number of processes used in the conducted experiments is less than the number of logical cores. In the conducted experiments, they were 42 processes.
- In CUDA, dividing the test case into l disjoined subsequences each with length equals RunSize utilizes the GPU shared memory by invoking the kernel l times when applying the test case. This is shown to increase the speedup. According to conducted experiments, RunSize of 1000 inputs is used in the assessment as this number provided the best results.

References

[1] T. Repasi, "Software testing - State of the art and current research challanges," *International Symposium on Applied Computational Intelligence and Informatics*, Timisoara, 2009, pp. 47-50.

[2] L. B. R. Oliveira and E. Y. Nakagawa, "A service-oriented reference architecture for software testing tools," In *Proceedings of the 5th European Conference on Software Architecture*, Essen, Germany, 2011.

[3] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. John Wiley and Sons, 2013.

[4] M. J. Harrold, "Testing: a roadmap," In *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, 2000, pp. 61-72.

[5] R. M. Hierons and U. C. Turker, "Parallel Algorithms for Testing Finite State Machines: Generating UIO sequences," *IEEE Transactions on Software Engineering*, vol. 42, no. 99, pp. 1077-1091, 2016.

[6] P. C. Jorgensen, *Software Testing: A Craftsman's Approach*,4th ed. Auerbach Publications, 2013 , p. 494.

[7] J. Pan. (1999, Oct, 19). *Software Testing*. Available: https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

[8] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*, 2ed. Addison wesley, 1986.

[9] R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software*, vol. 62, no. 1, pp. 21-46, 2002.

[10] A. Rockstrom and R. Saracco, "SDL-CCITT specification and description language," *IEEE Transactions on Communications*, vol. 30, no. 6, pp. 1310-1318, 1982.

[11] A. Benharref, R. Dssouli, M. A. Serhani, A. En-Nouaary, and R. Glitho, "New approach for EFSM-based passive testing of web services," *Testing of Software and Communicating Systems*, pp. 13-27,2007.

[12] S. Hallé and R. Villemaire, "Runtime monitoring of web service choreographies using streaming XML," in *Proceedings of the ACM symposium on Applied Computing*, pp. 2118-2125, 2009.

[13] M. Haydar, A. Petrenko, and H. Sahraoui, "Formal verification of web applications modeled by communicating automata," in *International Conference on Formal Techniques for Networked and Distributed Systems*, 2004, pp. 115-132.

[14] G. Morales, S. Maag, A. Cavalli, W. Mallouli, E. M. De Oca, and B. Wehbi, "Timed extended invariants for the passive testing of web services," in *IEEE International Conference on Web Services (ICWS)*, 2010, pp. 592-599.

[15] J. Simmonds, "Dynaamic analysis of web services," Ph.D. dissertation, University of Toronto, 2011.

[16] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: a survey annotated with experimental evaluation," *Information and Software Technology*, vol. 52, no. 12, pp. 1286-1297, 2010.

[17] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Computer Networks and ISDN systems*, vol. 15, no. 4, pp. 285-297, 1988.

[18] D. P. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions on Software Engineering*, vol. 15, no. 4, pp. 413-426, 1989.
[19] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. 4, no. 3, pp.178-187, 1978.
[20] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings of the 12th International Symposium on Software Reliability Engineering*, 2001 , pp. 34-43.
[21] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, no. 5, pp. 53-59, 2015.
[22] A. D. Friedman and P. R. Menon, *Fault Detection in Digital Circuits*. Prentice Hall, 1971.
[23] Y. Dong, Z. Li, Y. Cheng, and H. Zhao, "A Model DrivenTesting Solution for Embedded System with Simulink/Stateflow Model," in *Second International Conference on Trustworthy Systems and Their Applications (TSA)*, 2015, pp. 24-29.
[24] M.-C. Qu, N.-G. Cui, Y.-N. Zhang, X.-H. Wu, and B.-S. Zou, "Embedded Software Testing Requirements Modeling and Automatic Test Case Generation Based on Multiple Graphs," *Advanced Science Letters*, vol. 21, no. 11, pp. 3530-3535, 2015.
[25] K. El-Fakih and N. Yevtushenko, "Test translation for embedded finite state machine components," *The Computer Journal*, vol. 59, pp. 1805-1816, 2016.
[26] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55-71, 2011.
[27] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1994, pp. 109-124.
[28] K. El-Fakih, N. Yevtushenko, and G. v. Bochmann, "Diagnosing multiple faults in communicating finite state machines," in *Formal Techniques for Networked and Distributed Systems*, 2002, pp. 85-100.
[29] J.-h. Li, G.-x. Dai, and H.-h. Li, "Mutation analysis for testing finite state machines," in *Second International Symposium on Electronic Commerce and Security, 2009. ISECS'09*, 2009, vol. 1, pp. 620-624: IEEE.
[30] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing—Approach and case studies," *Science of Computer Programming*, vol. 120, pp. 25-48, 2016.
[31] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours," *IEEE Transactions on Communications*, vol. 39, no. 11, pp. 1604-1615, 1991.
[32] X. Zhang, W. Luo, X. Li, and B. Yan, "A Transfer Fault Diagnosing Method for Protocol Conformance Test Based on FSMs," in *Asia-Pacific Conference on Information Processing*, vol. 1, 2009, pp. 173-177.
[33] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090-1123, 1996.
[34] R. Lipton, "Fault diagnosis of computer programs," *Student Report, Carnegie Mellon University*, 1971.

[35] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 4, pp. 279-290, 1977.
[36] R. J. Lipton, R. A. DeMillo, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34-41, 1978.
[37] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, 2011.
[38] A. J. Offutt, J. Voas, and J. Payne, "Mutation operators for Ada," Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University1996.
[39] A. Offutt VI and K. N. King, "A Fortran 77 interpreter for mutation analysis," in *ACM SIGPLAN Notices*, vol. 22, no. 7, pp. 177-188, 1987.
[40] S. Kim, J. A. Clark, and J. A. McDermid, "Class mutation: Mutation testing for object-oriented programs," in *Proc. Net. ObjectDays*, 2000, pp. 9-12.
[41] H. Shahriar and M. Zulkernine, "Mutation-based testing of buffer overflow vulnerabilities," in *32nd Annual IEEE International Computer Software and Applications Conference*, 2008, pp. 979-984.
[42] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL injection vulnerability checking," in *The Eighth International Conference on Quality Software*, 2008, pp. 77-86.
[43] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu, "Mutation testing of protocol messages based on extended TTCN-3," in *22nd International Conference on Advanced Information Networking and Applications*, 2008, pp. 667-674.
[44] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. Wurden, "Model-Based Quality Assurance of Windows Protocol Documentation," in *1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 502-506.
[45] T. Mouelhiv, F. Fleurey, and B. Baudry, "A generic metamodel for security policies mutation," in *IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW'08*, 2008, pp. 278-286.
[46] R. M. Hierons and M. G. Merayo, "Mutation Testing from Probabilistic Finite State Machines," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION)*, Windsor, 2007, pp. 141-150.
[47] S. S. Batth, E. R. Vieira, A. Cavalli, and M. Ü. Uyar, "Specification of timed EFSM fault models in SDL," in *International Conference on Formal Techniques for Networked and Distributed Systems*, 2007, pp. 50-65.
[48] S. Lee, X. Bai, and Y. Chen, "Automatic mutation testing and simulation on OWL-S specified Web services," in *41st Annual Simulation Symposium*, 2008, pp. 149-156.
[49] R. A. DeMillo, "Completely validated software: test adequacy and program mutation (panel session)," in *Proceedings of the 11th International Conference on Software Engineering*, 1989, pp. 355-356.
[50] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.
[51] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, "Evaluation of three specification-based testing criteria," in *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems*, 2000, pp. 179-187.

[52] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235-253, 1997.

[53] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability,* vol. 4, no. 1, pp. 9-31, 1994.

[54] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *International Conference on Software Testing, Verification and Validation Workshops*, 2009, pp. 220-229.

[55] A. P. Mathur, *Foundations of Software Testing, 2/e*. Pearson Education India, 2008.

[56] A. Vincenzi, M. Delamaro, E. Höhn, and J. C. Maldonado, "Functional, control and data flow, and mutation testing: Theory and practice," in *Testing Techniques in Software Engineering*, 2010, pp. 18-58.

[57] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software, Practice and Experience*, vol. 26, no. 2, pp. 165-176, 1996.

[58] W. E. Wong, A. P. Mathur, and J. C. Maldonado, "Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness," in *Software Quality and Productivity*, 1995, pp. 258-265.

[59] K. El-Fakih, A. Simao, N. Jadoon, and J. C. Maldonado, "An assessment of extended finite state machine test selection criteria," *Journal of Systems and Software*, vol. 123, pp. 106-118, 2017.

[60] A. da Silva Simao, A. M. Ambrósio, S. C. Fabbri, A. S. M. S. do Amaral, E. Martins, and J. C. Maldonado, "Plavis/FSM: an environment to integrate FSM-based testing tools," in *Tool Session of XIX Brazilian Symposium on Software Engineering*, 2008, pp. 1-6.

[61] A. Ghedamsi and G. V. Bochmann, "Test result analysis and diagnostics for finite state machines," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 244-251.

[62] A. Ghedamsi, G. v. Bochmann, and R. Dssouli, "Diagnosis of single transition faults in communicating finite state machines," in *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993, pp. 157-166.

[63] A. Ghedamsi, G. V. Bochmann, and R. Dssouli, "Diagnostic tests for communicating finite state machines," in *Proceedings of Phoenix Conference on Computers and Communications*, 1993, pp. 254-260.

[64] A. Ghedamsi, R. Dssouli, and G. v. Bochmann, "Diagnostic tests for single transition faults in non-deterministic finite state machines," in *Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems V*, 1992, pp. 105-116.

[65] A. Ghedamsi, G. Bochmann, and R. Dssouli, "Multiple fault diagnosis for finite state machines," in *INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future*, 1993, pp. 782-791.

[66] R. Belhassine-Cherif and A. Ghedamsi, "Diagnostic tests for communicating nondeterministic finite state machines," in *Proceedings pf the Fifth IEEE Symposium on Computers and Communications*, 2000, pp. 424-429.

[67] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46-55, 1998.
[68] K. Yang-Suk, K. Jin-Soo, and H. Soonhoi, "ParADE: An OpenMP Programming Environment for SMP Cluster Systems," in *Supercomputing ACM/IEEE Conference*, 2003, pp. 6-6.
[69] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard," Technical Report, University of Tennessee Knoxville, 1995.
[70] P. S. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.
[71] R. P. Tan, P. Nagpal, and S. Miller, "Automated black box testing tool for a parallel programming library," in *International Conference on Software Testing Verification and Validation*, 2009, pp. 307-316.
[72] J. Throop, "OpenMP: shared-memory parallelism from the ashes," *Computer*, vol. 32, no. 5, pp. 108-109, 1999.
[73] J. Dongarra et al., *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers San Francisco, 2003.
[74] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT press, 1999.
[75] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT press, 1999.
[76] C. McClanahan, "History and evolution of GPU architecture," *A Survey Paper*, College of Computing, Georgia Tech, 2010.
[77] M. Ali and T. Ozkul, "Review of Memory/Cache Management Technologies used on Heterogeneous Computing Systems," *International Journal of Computer and Information Technology*, vol. 3, no. 3, pp. 515-522, 2014.
[78] G. D. Barlas, "Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 5, pp. 429-441, 1998.
[79] (November 15th). *List of Nvidia graphics processing units*. Available: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units
[80] (November 15th). *Graphical Processing Units*. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit
[81] C. Andres, N. Yevtushenko, and A. Cavalli, "Modeling and testing the European train control system," Technical Report TechRca, Telecom Sudparis, 2013.
[82] M. Ali "Parallel algorithms for distinguishing nondeterministic finite state machines," M.S. thesis, American University of Sharjah, Sharjah, United Arab Emirates, 2015.
[83] N. Shabaldina, K. El-Fakih, and N. Yevtushenko, "Testing nondeterministic finite state machines with respect to the separability relation," in *Testing of Software and Communicating Systems*, 2007, pp. 305-318.
[84] F. Brglez. (2017, Jan. 15). *ACM/SIGMOD benchmark dataset*. Available: http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html
[85] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, "Mutation analysis testing for finite state machines," in *Proceedings of the 5th International Symposium on Software Reliability Engineering*, 1994, pp. 220-229.
[86] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro,“Proteum/FSM: A Tool to Support Finite State Machine Validation

Based on Mutation Testing," in *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC'99)*, Talca, Chile, 1999, pp. 96-104.

[87] K. El-Fakih, S. Prokopenko, N. Yevtushenko, and G. v. Bochmann, "Fault diagnosis in extended finite state machines," in *IFIP International Conference on Testing of Software and Communicating Systems*, 2003, pp. 197-210.

[88] Z. Pap, G. Csopaki, and S. Dibuz, "On FSM-based fault diagnosis," in *IFIP International Conference on Testing of Communicating Systems*, 2005, pp. 159-174.

[89] K. El-Fakih, T. Salameh, and N. Yevtushenko, "On Code Coverage of Extended FSM Based Test Suites: An Initial Assessment," in *IFIP International Conference on Testing Software and Systems*, 2014, pp. 198-204.

[90] M. H. Hassoun "Fault coverage and diagnosis of protocols and systems modeled as extended finite state machines," M.S. thesis, American University of Sharjah, Sharjah, United Arab Emirates, 2015.

[91] K. El-Fakih, G. Barlas, M. Ali, and N. Yevtushenko, "Parallel algorithms for reducing derivation time of distinguishing experiments for nondeterministic finite state machines," *International Journal of Parallel, Emergent and Distributed Systems*, pp. 1-14, 2017.

[92] A. Haddad, K. El-Fakih, and G. Barlas, "Parallel Implementation for Deriving Preset Distinguishing Experiments of Nondeterministic Finite State Machines," *Seventh International Conference on Modeling, Simulation, and Applied Optimization*, Sharjah, UAE, 2017.

[93] R. Hierons and U. Turker, "Parallel algorithms for generating harmonised state identifiers and characterising sets," *IEEE Transactions on Computers,* vol. 65, no. 11, pp. 3370-3383, 2016.

[94] G. Luo, A. Petrenko, and G. v. Bochmann, "Selecting test sequences for partially-specified nondeterministic finite state machines," in *Protocol Test Systems*, 1995, pp. 95-110.

[95] S. W. Bollinger and S. F. Midkiff, "An investigation of circuit partitioning for parallel test generation," *IEEE VLSI Test Symposium,* Atlantic City, NJ, USA, 1992, pp. 119-124

[96] S. J. Chandra and J. H. Patel, "Test generation in a parallel processing environment," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD'88.,* 1988, pp. 11-14.

[97] A. Motohara, "A parallel scheme for test pattern generation," in *Proccedings og the IEEE Int'l Confnference in Computer-Aided Design*, 1986, pp. 156-159.

[98] R. H. Klenke, R. D. Williams, and J. H. Aylor, "Parallelization methods for circuit partitioning based parallel automatic test pattern generation," in *VLSI Test Symposium*, 1993, pp. 71-78.

[99] R. H. Klenke, R. D. Williams, and J. H. Aylor, "Parallel-processing techniques for automatic test pattern generation," *Computer*, vol. 25, no. 1, pp. 71-84, 1992.

[100] S. Patil and P. Banerjee, "A parallel branch and bound algorithm for test generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 3, pp. 313-322, 1990.

Vita

Emad Badawi was born on March 15, 1991, in Aqqaba, Palestine. He studied at Arab American University (AAUJ) in Jenin, Palestine, from which he graduated in 2010 and obtained a Bachelor's Degree in Computer Systems Engineering.

Mr. Badawi worked as WebSphere system administrator for almost one year. After that, he moved to the United Arab Emirates in 2015 where he joined the Master's program in Computer Engineering at the American University of Sharjah.