PARALLEL ALGORITHMS FOR DISTINGUISHING NONDETERMINISTIC FINITE STATE MACHINES

by

Mustafa Ali

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

February 2015

© 2015 Mustafa Ali. All rights reserved.

Approval Signatures

We, the undersigned, approve the Master's Thesis of Mustafa Ali

Thesis Title: Parallel Algorithms for Distinguishing Nondeterministic Finite State Machines

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

___________________________ _______________

Dr. Gerassimos Barlas
Professor, Department of Computer Science and Engineering
Thesis Advisor

___________________________ _______________

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

___________________________ _______________

Dr. Assim Sagahyroon
Head, Department of Computer Science and Engineering
Thesis Committee Member

___________________________ _______________

Dr. Dmitry Efimov
Assistant Professor, Department of Mathematical Sciences
Thesis Committee Member

________________________ _______________

Dr. Assim Sagahyroon
Head, Department of Computer Science and Engineering

___________________________ _______________

Dr. Mohamed El-Tarhuni
Associate Dean, College of Engineering

___________________________ _______________

Dr. Leland Blank
Dean, College of Engineering

___________________________ _______________

Dr. Khaled Assaleh
Interim Vice Provost for Research & Graduate Studies

Acknowledgments

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

I am thankful to the Department of Computer Engineering and the American University of Sharjah for offering me Graduate Teaching Assistantship, which allowed me to pursue my graduate studies.

In the end, I would like to thank my parents, brothers, and my fiancé Tahera, for their love, encouragement, and sacrifices. Without their support I wouldn't have been able to carry out my work.

Abstract

Many methods are used for the development of experiments and conformance tests based on the specification given in the form Finite State Machines (FSMs). In FSM-based testing, we have an FSM or a black-box Implementation Under Test (IUT) about which we lack some information, and we want to deduce this information by conducting experiments on the IUT. An experiment consists of applying input sequences, observing corresponding output responses, and drawing conclusions about the IUT. An experiment is adaptive if at each step of the experiment the next input is selected which is based on the previously observed outputs. A distinguishing experiment determines the initial state of the FSM. In this thesis, we consider two implementations of an existing sequential algorithm for deriving the minimal length of an adaptive distinguishing experiment for a nondeterministic FSM. We show that the execution time for both of these implementations grows exponentially as the size or the number of transitions of the FSM increases. Accordingly, in order to obtain a solution in a reasonable time, we develop four parallel implementations of the considered sequential algorithms, namely, a multi-core implementation on Central Processing Unit, two Graphical Processing Unit (GPU) implementations based on the platforms like CUDA and Thrust, respectively, and an implementation on a Network of Workstations (NoWs). Comprehensive experiments are conducted to assess and compare the performance and the speedup of the developed implementations. Based on the results obtained from these experiments, the parallel implementation on a NoW provides the best performance and speedup, followed by the CUDA, then the Thrust, followed by the multi-core CPU implementation.

Search Terms: Conformance Testing, Adaptive Distinguishing Experiments, Parallel Algorithms for Distinguishing Experiments.

Table of Contents

Abstract 5

List of Figures 8

List of Tables 10

List of Abbreviations 11

Chapter 1: Introduction 12

Chapter 2: Related Work and Literature Review 16

2.1 Related Work 16

2.2 Graphical Processing Units 17

2.2.1 Compute Unified Device Architecture 19

2.2.2 Thrust 23

2.3 Message Passing Interface 23

2.4 Divisible Load Theory 24

Chapter 3: Determining the Minimal Length of Adaptive Distinguishing Experiments for Nondeterministic FSMs 27

3.1 Preliminaries 27

3.2 Two Algorithms for Determining the Minimal Length of Adaptive Distinguishing Experiments for Complete Observable Nondeterministic FSMs 31

3.2.1 Algorithm A (For Determining the Minimal Length of an Adaptive Distinguishing Experiment) 32

3.2.2 Algorithm B (For Determining the Minimal Length of an Adaptive Distinguishing Experiment) 36

3.3 An Example Comparing Algorithms A and B 38

Chapter 4: Parallel Algorithms for Determining the Minimal Length of Adaptive Distinguishing Experiments for Nondeterministic FSMs 41

4.1 Multi-Threaded Implementation for Algorithm A (MT^A) 42

4.2 Parallel Algorithm for B 44

4.2.1 Multi-Threaded Implementation for Parallel Algorithm B (MT^B) 46

4.2.2 GPU Implementations for Parallel Algorithm B ($CUDA^B$ and $Thrust^B$) 48

4.2.3 Multiple-Node Implementation of Parallel Algorithm B (MN^B) 50

Chapter 5: Experimental Evaluation 55

5.1 Execution Time versus Number of Transitions of Sequential Algorithms A and B 57
5.2 Execution Time versus Number of Transitions of Sequential Algorithm A Against MT^A 58
5.3 Execution Time versus Number of Transitions of Sequential Algorithm B Against MT^B 60
5.4 Execution Time versus Number of Transitions for Sequential Algorithms Against Other Parallel Implementations 62
5.5 Execution Time versus Number of Transitions for Multi-Threaded Implementations Against Other Parallel Implementations 66
5.6 Achieved Speedup with Respect to Algorithm A 68
5.7 Achieved Speedup with Respect to Algorithm B 71
5.8 Achieved Speedup versus Number of Transitions with Respect to Algorithm A 74
5.9 Achieved Speedup versus Number of Transitions with Respect to Algorithm B 75
5.10 Summary of All Obtained Results 77
Chapter 6: Conclusion 82
References 84
Appendix A 90
Vita 94

List of Figures

Figure 1. Hardware Resources for the Fermi GPU 21
Figure 2. CUDA Programming Model 21
Figure 3. A Test Case P Over Alphabets $I = \{a, b\}$ and $O = \{0, 1\}$ 29
Figure 4. Considered FSM S with Three Initial States 30
Figure 5. The Intersection $S \cap P$ 30
Figure 6. Considered FSM S...34
Figure 7. Considered FSM S_2 37
Figure 8. Algorithm A versus Algorithm B for Small and Medium FSMs 57
Figure 9. Algorithm A versus Algorithm B for Big FSMs 57
Figure 10. Algorithm A versus MT^A for Small and Medium FSMs...........................58
Figure 11. Algorithm A versus MT^A for Big FSMs 58
Figure 12. Speedup for MT^A w.r.t. (Sequential) Algorithm A for Small and Medium FSMs 59
Figure 13. Speedup for MT^A w.r.t. (Sequential) Algorithm A for Big FSMs............59
Figure 14. Sequential Algorithm B versus MT^B for Small and Medium FSMs 60
Figure 15. Sequential Algorithm B versus MT^B for Big FSMs.................................61
Figure 16. Speedup for MT^B w.r.t. (Sequential) Algorithm B for Small and Medium FSMs 61
Figure 17. Speedup for MT^B w.r.t. Sequential Algorithm B for Big FSMs 62
Figure 18. Sequential Algorithms versus Other Parallel Implementations for Small and Medium FSMs...63
Figure 19. Sequential Algorithms versus Other Parallel Implementations for Big FSMs 63
Figure 20. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm A for Small and Medium FSMs...64
Figure 21. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm A for Big FSMs...64
Figure 22. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm B for Small and Medium FSMs 65

Figure 23. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm B for Big FSMs....66
Figure 24. Multi-Threaded Implementations versus Other Parallel Implementations for Small and Medium FSMs....67
Figure 25. Multi-Threaded Implementations versus Other Parallel Implementations for Big FSMs....67
Figure 26. Achieved Speedup w.r.t. Algorithm A (States = 100)....68
Figure 27. Achieved Speedup w.r.t. Algorithm A (states = 150)....69
Figure 28. Achieved Speedup w.r.t. Algorithm A (states = 200)....69
Figure 29. Achieved Speedup w.r.t. Algorithm A (states = 250)....70
Figure 30. Achieved Speedup w.r.t. Algorithm B (States = 100)....72
Figure 31. Achieved Speedup w.r.t. Algorithm B (states = 150)....72
Figure 32. Achieved Speedup w.r.t. Algorithm B (states = 200)....73
Figure 33. Achieved Speedup w.r.t. Algorithm B (states = 250)....73
Figure 34. Speedup versus Number of Transitions w.r.t. Algorithm A for the Considered Size-I FSMs....74
Figure 35. Speedup versus Number of Transitions w.r.t. Algorithm A for the Considered Size-II FSMs....75
Figure 36. Speedup versus Number of Transitions w.r.t. Algorithm B for the Considered Size-I FSMs....76
Figure 37. Speedup versus Number of Transitions w.r.t. Algorithm B for the Considered Size-II FSMs....76
Figure 38. Execution Time versus the Number of FSM Transitions, for All the Considered Nodes....91
Figure 39. Communication Time versus Message Size, for Two Network Nodes.....92

List of Tables

Table 1. Recent NVidia GPUs and their configurations....19
Table 2. Tabular Representation for the Considered FSM **S** in Figure 6. 34
Table 3. Tabular Representation for the Considered FSM $\mathbf{S_2}$ in Figure 7 38
Table 4. Combinations of Generated FSMs....55
Table 5. System Configuration & Platform Details....55
Table 6. Summary of Execution Time (Minutes) for All the Conducted Experiments 77
Table 7. Summary of Execution Time (Minutes) for Small FSMs....78
Table 8. Summary of Execution Time (Minutes) for Medium FSMs 78
Table 9. Summary of Execution Time (Minutes) for Big FSM....78
Table 10. Speedup w.r.t (Sequential) Algorithm A for All the Conducted Experiments 79
Table 11. Speedup w.r.t (Sequential) Algorithm A for Small FSMs....79
Table 12. Speedup w.r.t (Sequential) Algorithm A for Medium FSMs 79
Table 13. Speedup w.r.t (Sequential) Algorithm A for Big FSMs 80
Table 14. Speedup w.r.t (Sequential) Algorithm B for All the Conducted Experiments 80
Table 15. Speedup w.r.t (Sequential) Algorithm B for Small FSMs....80
Table 16. Speedup w.r.t (Sequential) Algorithm B for Medium FSMs....81
Table 17. Speedup w.r.t (Sequential) Algorithm B for Big FSMs 81
Table 18. Considered Nodes (*N*) in the NoW....90
Table 19. Computation Speed (*p*) for All the Considered Nodes 91
Table 20. Values Calculated for the *e* Parameter for All the Considered Nodes....92
Table 21. Communication Parameters....92

List of Abbreviations

A	Sequential algorithm which derives I/O-successors iteratively
B	Sequential algorithm which derives all the I/O-successors in advance
FSM	Finite State Machine
NFSM	Nondeterministic Finite State Machine
IUT	Implementation Under Testing
GPU	Graphical Processing Unit
CUDA	Compute Unified Device Architecture
Thrust	Software tool for programming on GPU
CPU	Central Processing Unit
MT^A	Multi-Threaded Implementation for Algorithm A
MT^B	Multi-Threaded Implementation for Algorithm B
$CUDA^B$	GPU Implementation for using software tool CUDA, for Algorithm B
$Thrust^B$	GPU Implementation for using software tool Thrust, for Algorithm B
MN^B	Multiple-Node Implementation for Algorithm B
FORTRAN	Formula Translator
SIMD	Single Instruction Multiple Data
DLT	Divisible Load Theory
DRAM	Dynamic Random Access Memory
SRAM	Static Random Access Memory
MPI	Message Passing Interface
MPP	Massively Parallel Processing
STL	Standard Template Library
SM	Streaming Multiprocessors
NoWs	Network of Workstations
N	Number of Nodes in Network of Workstations
M	Total number of pairs of different states of a given FSM
RAM	Random Access Memory
MB	Mega Byte

Chapter 1: Introduction

The advancements in computer technology have enabled systems to get larger so that they are capable of fulfilling more complicated tasks. As a result, these systems are also becoming less reliable and more vulnerable [1]. Consequently, software testing has become an integral part of system and software development; however, for complex systems, testing is known to be a formidable task [1]. This motivates the study of testing Finite State Machines (FSMs) to ensure the correct functioning of systems and to discover aspects of their behavior. An FSM is a state transition system that has a finite number of inputs, outputs, states, and a finite number of transitions, each labeled by an input/output pair. FSMs are widely used in various application domains such as telecommunication, communication protocols and other reactive systems. FSMs are the underlying models for formal description techniques, such as statecharts [58], Specification Description Language (SDL) [58], and Unified Modelling Language (UML) specification [58].

An FSM is deterministic if, for some input at some state, there is exactly one outgoing transition of the state under that input. Nondeterminism in the specification is also not unusual. An FSM is non-deterministic if, for some input at some state, there are more than one outgoing transition of the state under that input.

Many methods are known for the development of experiments and conformance tests based on the specifications given in the form of an FSM [1-7]. In FSM-based testing, we have a machine or a black-box Implementation Under Test (IUT) about which we lack some information, and we want to deduce this information by conducting experiments on this FSM. An *experiment* on an FSM consists of applying input sequences to the machine, observing corresponding output responses, and drawing conclusions about the machine under test. An experiment is a *preset* if all the input sequences are known before starting the experiment, and *adaptive* if at each step of the experiment the next input is selected based on the previously observed outputs [4] [8]. Distinguishing experiments are used when deriving FSM based tests with guaranteed fault coverage and those experiments are elaborated for different types of FSMs. An FSM is said to be *initialized* if it has one initial state; otherwise, it

is said to be *weakly-initialized* or *non-initialized.* An FSM is *observable* if at each state the machine has at most one transition under a given input/output pair.

A *distinguishing experiment* is defined as an experiment which determines the initial state of the FSM, i.e. a state of the FSM before the start of the experiment. Such experiments are widely used when checking the correspondence between transitions of an IUT and those of the specification FSM [34]. If a distinguishing sequence for a finite state machine exists, then one can determine the length k of that sequence.

Nowadays, nondeterministic systems are attracting lots of attention in the field of protocol analysis and testing. Adaptive experiments with nondeterministic FSMs are discussed in [11-15]. Petrenko and Yevtushenko in [13] came up with an idea of a test case which described an adaptive experiment as an initialized FSM with an acyclic transition diagram such that at each non-deadlock state only one input was defined with all possible outputs. This definition of a test case enables defining distinguishing test cases which are based on the properties of the intersection of a transition system under experiment and a given test case. The examples of how a distinguishing test case can be derived for two states of Nondeterministic Finite State Machines (NFSMs) are shown in [13-15]. In particular, Alur in [11] showed that the length of the shortest adaptive distinguishing test case that distinguishes two states of an observable nondeterministic FSM with n states is at most $n(n - 1)/2$.

In this thesis, we consider adaptive distinguishing experiments for a pair of states of complete observable nondeterministic FSMs. Lee and Yannakakis [6] proposed an approach for deriving an adaptive distinguishing sequence of a deterministic FSM that is based on refining a partition of the set of states based on different outputs. The work in [6] was extended in [34] dealing with nondeterministic FSMs. In particular, in [34] necessary and sufficient conditions for having adaptive distinguishing test cases are established and a sequential algorithm for deriving a distinguishing adaptive test case with minimal length is proposed.

In this thesis, we consider the sequential algorithm presented in [34] for determining the minimal length of an adaptive distinguishing experiment for nondeterministic FSMs. As the sequential algorithm in [34] works for any number of

pairs. We modify it, such that it works for a pair of states and develop two sequential algorithms for determining the minimal length of adaptive distinguishing experiments for a pair of states for complete observable nondeterministic FSM. Also, our experiments show that the execution time of these sequential algorithms increases drastically as the considered FSMs increase in size (i.e., the number of transitions of a machine). Therefore, to obtain the solution (i.e., the length of the distinguishing sequence) in a reasonable time, we develop many parallel algorithms/implementations of the two considered sequential algorithms. Parallel execution typically requires the partitioning of the computation and/or data. The two generic approaches for this problem are function decomposition and data decomposition. In the latter, the problem data are partitioned into disjoined sets and processed separately.

Data decomposition, also known as the data-parallel design approach, is widely applicable in a large number of domains, including numerical computations, biomedical informatics, and multimedia. In this thesis, we apply a data decomposition or data partitioning approach for the various parallel implementations of the adapted sequential algorithms. These parallel implementations include implementation on a multi-core CPU via multiple threads, implementation on parallel platforms like Graphical Processing Units (GPUs), and implementation on a Network of Workstations (NoWs). These parallel implementations are implemented using different software tools and platforms such as Qt Threads [55], CUDA/Thrust [35], and MPI (Message Passing Interface), a standard for portable message-passing [41].

To summarize the results, after conducting comprehensive experiments, we find that parallel implementation on a NoW gives the best performance amongst parallel implementations, and the speedup obtained is much more significant than in sequential algorithms. The parallel implementation on the GPU using the software platform CUDA gives the second best performance, and the parallel implementation on a multi-core CPU along with parallel implementation on a GPU using the software platform Thrust gives the third best performance.

The organization of this thesis is as follows. Chapter 2 includes related work on adaptive distinguishing experiments and preset distinguishing experiments for

deterministic/nondeterministic FSMs. It also includes a brief literature review on parallel platforms used in this thesis. Chapter 3 includes preliminaries, definition of a distinguishing test case, and algorithms to determine the minimal length of an adaptive distinguishing experiment for complete observable nondeterministic FSMs. Chapter 4 discusses parallel algorithms/implementations to determine the minimal length of an adaptive distinguishing experiment for complete observable nondeterministic FSMs. In Chapter 5 we discuss the experimental evaluation of the conducted experiments and Chapter 6 concludes this thesis.

Chapter 2: Related Work and Literature Review

2.1 Related Work

Research on preset and adaptive distinguishing experiments for deterministic FSMs started with the fundamental paper on "Gedanken experiments" by Moore [8]. Surveys and more information on FSM-based experiments with some related algorithms can be found in [4-6, 16]. Particularly, Gill [4] and Lee and Yannakakis [6] presented methods for deriving preset and adaptive distinguishing experiments for deterministic FSMs with corresponding evaluations of the complexity of these experiments.

Preset distinguishing experiments for nondeterministic FSMs are presented in [11, 12, 23-27]. In particular, Spitsyna in [23] presents a method for deriving a sequence that separates two initialized nondeterministic FSMs. An input sequence is a separating sequence of two FSMs if the sets of output sequences produced by the NFSMs to the input sequence do not intersect [28]. A tight upper bound on the shortest preset separating sequence is shown to be of the order 2^{n^2} where n is the number of states of a complete nondeterministic observable FSM [23]. Hwang in [26] examined the non-equivalence relation between two states of a complete FSM, invalidated the upper bound in [23] and determined that the upper bound on the length of a sequence distinguishing two states of a non-observable FSM with n states is $2^n - 2$. An FSM is said to be *complete*, if at every state of the machine there is an outgoing transition under each input. A complete FSM is *reduced* if at each two different states, the FSM does not have the same behavior. Kushik and Yevtushenko in [27] demonstrated that there is a special class of FSMs which contain n states and $(n - 1)$ inputs, whose shortest sequence can be given by the length $2^{n-1} - 1$ (i.e., its length is exponential with respect to the number of FSM states). Related problems were also studied by Zhang and Cheung when deriving transfer and distinguishing trees for observable nondeterministic FSMs with probabilistic and weighted transitions [29].

Adaptive experiments for nondeterministic FSMs are considered in [11-15]. In particular, Petrenko and Yevtushenko in [13] describe the notion of a test case for an adaptive experiment as an initialized observable FSM with an acyclic transition

diagram such that at each non-deadlock state only one input is defined with all possible outputs. In [13 - 15] the process for deriving a distinguishing test case for two states of a nondeterministic observable FSM is presented, provided that such a distinguishing test case exists. Alur in [11] showed the length of a shortest adaptive distinguishing test case that distinguishes two states of an observable nondeterministic FSMs with n states is at most $n(n-1)/2$. Petrenko and Yevtushenko in [15] considered a set of adaptive test cases which contained three parts: a preamble for reaching an appropriate state, a traversal input/output sequence, and a state identifier. In such cases, the length of an identifier can be optimized when distinguishing not two but several states with the same distinguishing test case. In addition to this, from [9, 10, 12] a distinguishing sequence derived for a non-initialized FSM can also be adaptive. Gromov in [30] and El-Fakih [31] presented adaptive experiments for timed nondeterministic observable FSMs, and some work on adaptive experiments for extended and communicating FSMs is reported in [25, 32, 33]. In [34], adaptive distinguishing experiments for non-initialized, possibly non-observable nondeterministic FSMs are considered. Lee and Yannakakis [6] also proposed an approach for deriving an adaptive distinguishing sequence for a deterministic FSM that is based on refining a partition of the set of states based on different outputs.

In this thesis we study adaptive distinguishing experiments for nondeterministic FSMs. We develop many parallel implementations of the sequential algorithm present in [34]. Our main objective is to reduce the execution time of deriving an experiment, as execution time increases drastically as the size of the FSM increases (i.e., the number of transitions of a machine). We applied a geometric decomposition pattern for parallelizing the sequential algorithm. We tested our parallel implementations on three hardware platforms (multicore CPU PC, GPU, and CPU/GPU cluster) and four software platforms (Qt threads, CUDA, Thrust and MPI+CUDA). A brief literature review of GPUs, GPU software platforms (i.e. CUDA and Thrust), and the MPI standard is provided in the next section.

2.2 Graphical Processing Units (GPUs)

The Graphical Processing Unit (GPU) is a chip that contains a large number of parallel microprocessors. It was originally designed to accelerate 2D or 3D graphic

processing, in order to reduce the workload of the CPU. However, recent GPUs are composed of a large number of computing cores which are able to perform operations in parallel, and are connected to high-speed memory (DDR5) with very wide buses (256bit or larger). This architecture features enables the chips to process large amounts of data in a fraction of the time traditional single or multi-core CPUs can.

The development of GPU hardware began from a single core and fixed function hardware [35] pipeline application towards a combination of highly parallel programmable cores which 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 that is constantly evolving towards a general purpose more CPU-like core.

In 2001, NVIDIA released GeForce 3 [35]. This was the first GPU with a programmable pipeline and the ability to program previously non-programmable parts of the pipeline. In the following years, fully programmable graphic cards were introduced and the first wave of GPU computing started with the introduction of DirectX9, which too added the advantage of programmability in the GPU hardware.

In 2006, NVIDIA introduced the GeForce 8 series [35]. This was a great evolution in the history of GPUs because it contained massive parallel processors. The GPU introduced in 2009 as NVIDIA's Fermi architecture featured a true memory cache hierarchy, concurrent kernel execution, better double precision performance, combined memory address space and dual warp schedulers [35]. Since then, rapid progress in the development of GPUs has occurred. Table 1 describes some of the recent NVidia GPUs and their configurations [36] [54], where Cores represents individual cores (computing unit) contained in a GPU card and each cores is capable of executing a thread, Streaming Multi-processor (SM) is a collection of cores, and Cores/SM represents the number of cores present in a SM.

The evolution of the GPU has brought enormous advantages for high speed computing. GPUs are not only being used for graphical processing but also for numerical computations. This motivated us to implement an algorithm for determining the minimal length of an adaptive distinguishing experiment for

nondeterministic FSMs on a GPU so that we can reduce the execution time for the algorithm and attain significant speedup as compared to the same algorithm running on a single core machine. Normally, execution on a GPU can be carried out using various platforms such as OpenCL [56], CUDA [56], Thrust [56], etc. All of these are software platforms that enable us to program complex problems on GPUs. In this thesis we use CUDA and Thrust to implement an algorithm for determining the minimal length of an adaptive distinguishing experiment for non-deterministic FSMs on a GPU. Further details for CUDA and Thrust are mentioned in the sections below.

Table 1. Recent NVidia GPUs and their configurations

Card	Cores	Cores/SM	SM	Compute Capability
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

2.2.1 Compute Unified Device Architecture (CUDA)

At the end of 2006, NVIDIA introduced CUDA™ [32], a general purpose parallel computing platform and programming model that made it possible to execute parallel computation on GPUs. Many complex computational problems could be solved in a more efficient and faster way compared to a traditional single core CPU. CUDA is a parallel platform that allows and supports high level programming languages, application programming interfaces, or directive-based approaches, such as C [32], FORTRAN [32], DirectCompute [32], and OpenACC [32]. CUDA was developed keeping several design goals in mind such as:

- CUDA provides a small set of extensions to standard programming languages, like C, which enables a straightforward implementation of parallel algorithms. Programmers experienced with C/C++ can simply focus on

parallelization of the algorithms rather than spending time on their implementation.

- CUDA was developed to support heterogeneous computation where applications and algorithms use both the CPU and GPU. Serial portions of applications and algorithms run on the CPU, whereas parallel portions are unloaded to the GPU. This enables CUDA to be applied to many research domains such as molecular dynamics [57], quantum chemistry [57], and bioinformatics [57]. As the CPU and GPU are counted as separated devices, they have their own memory spaces. This allows simultaneous computation on the CPU and GPU without contention for memory resources.

CUDA architectures are organized into multiprocessors, each multiprocessor having a number of cores. With the evolution of the technology, current architectures from NVidia include more multiprocessors per die, and/or more cores, registers or shared memory per multiprocessor. For example, the CUDA-enabled GPU, Fermi, has twice the number of cores and increases the clock frequency as compared to the previous generation, the GT200. This one also doubles the number of multiprocessors and 32-bit register within each multiprocessor as compared to the G80.

The processor for the G80 follows the Single Instruction Multiple Data (SIMD) parallel architecture, and it is equipped with 128 cores. These cores are organized into 16 multiprocessors, each consisting of 8192 registers, a 16 KB shared memory, which is very close to the registers in speed (both 32 bits wide), and a few kilobytes of constant and texture memory caches. Each multiprocessor is capable of running a variable number of threads, though the local resources are divided amongst them. Since the G80 is SIMD, in a given cycle, each core in a multiprocessor will execute the same instruction but with different data depending on its thread ID. Communication between the multiprocessors can take place through the global memory. These basic features were shared by every hardware generation until the Fermi architecture was fabricated. Figure 1 shows the hardware resources for the Fermi GPU.

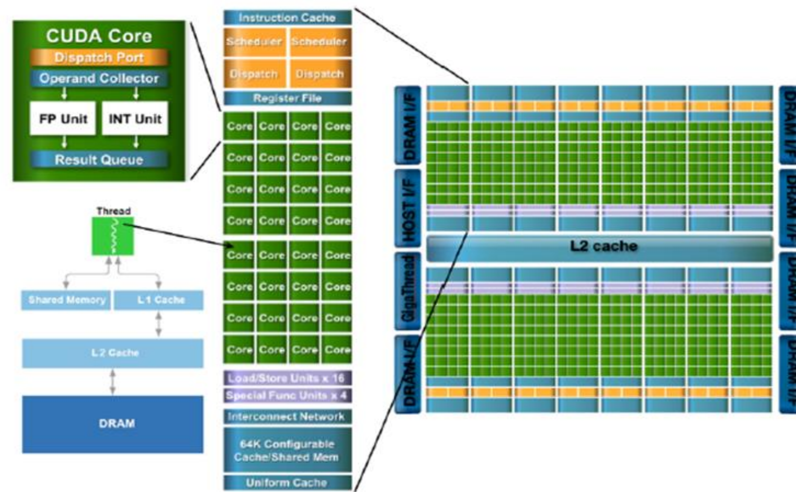

Figure 1. Hardware Resources for the Fermi GPU

The advantage of the CUDA programming model is that it guides the programmer to exploit fine-grained parallelism as required by massively parallel GPUs. In the CUDA programming model, the CPU host and GPU device maintain their own Dynamic Random Access Memory (DRAM) and address, denoted as host memory and global memory. However, multiprocessors have on-chip memory, which can be of two types: registers and shared memory, as depicted in Figure 2.

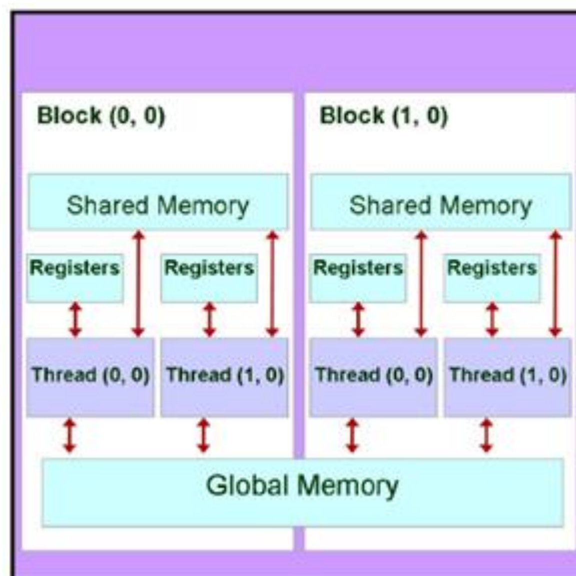

Figure 2. CUDA Programming Model

In the CUDA programming model, a program is decomposed into blocks which are running in parallel. A block is a group of threads which are being mapped to run on a single multiprocessor, where they can share Static Random Access Memory

(SRAM). Threads in the blocks are concurrently assigned to a single multiprocessor, and they divide the multiprocessor's resources equally amongst themselves.

The programming model also consists of warps. A warp consists of 32 threads that can physically run concurrently on all of the multiprocessors. Due to memory access limitations, warp size is less than the total number of cores. The programming model allows the programmer to determine the number of threads to be executed, but in case the number of threads exceeds the warp size, they are time-shared on the actual hardware resources.

In order to run the code on a GPU, the CUDA programming model calls for the creation of a kernel. A kernel is a function compiled according to the instructions of the device, which are downloaded and executed by all the threads on the GPU. Threads running on the different processors of the multiprocessors, sharing the same executable and global address space, may differ in the execution path, as the conditional execution of different operations on each multiprocessor can be achieved based on a unique thread ID. Threads also work independently on different data according to the SIMD model. Threads are organized into a grid as a set of thread blocks. A grid is defined as a collection of all the blocks in a single execution, which is explicitly defined by the developer and is assigned to a multiprocessor. When a kernel is invoked, it defines the sizes and dimensions of the thread blocks in the grid to be created. These sizes and dimensions must be carefully defined because they affect the performance of the GPU.

A thread block consists of a group of threads which are executed on a single multiprocessor. Threads in a block can communicate together by sharing data through the SM's shared memory. They can be synchronized together using the __syncthreads() primitive. Synchronization is mainly done for the coordination of the memory accesses. Each thread in a block has its own thread ID, which is the number of the thread within a 1D, 2D or 3D array of arbitrary size. Threads from the different blocks in the same grid cannot communicate, though threads belonging to the same block must all share registers and shared memory on a given multiprocessor. In order to maximize the execution efficiency, a programmer should wisely solve the tradeoff between parallelism and thread resources.

2.2.2 Thrust

Thrust is a C++ template library for CUDA that follows the Standard Template Library (STL) conventions. Thrust allows a programmer to implement high performance parallel applications with minimal programming effort through a high-level interface that is fully interoperable with CUDA C.

Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement complex algorithms with concise, readable source code. By describing a computation in terms of these high level abstractions the programmer provides Thrust with the freedom to select the most efficient implementation automatically. As a result, Thrust can be utilized in the rapid prototyping of CUDA applications, where programmer productivity matters most, as well as in production, where robustness and absolute performance are crucial.

2.3 Message Passing Interface (MPI)

MPI stands for Message Passing Interface, a portable message-passing standard that enables the development of parallel applications and libraries [37-40]. MPI specifies the names, calling sequences, and results of the subroutines or functions to be called from FORTRAN, C or C++ programs. Commercial and free, public domain implementations such as OpenMPI, MPICH, and pyMPI (MPI implementation in Python) are available. These implementations run on both tightly-coupled, Massively Parallel Machines (MPPs), and on Networks of Workstations (NoWs) [41].

MPI is used to specify the communication among a set of processes forming a concurrent program. The message passing paradigm is attractive because of its wide portability and scalability. It is compatible with both distributed-memory and shared-memory multiprocessors, and combinations of these elements. In MPI, the processes executed in parallel have separate memory address spaces. Communication between the processes takes place when part of the memory content of one process is copied into the memory content of another process. This operation is cooperative and occurs only when the first process executes a “send” operation and the second process executes a “receive” operation. In MPI, workload partitioning and mapping of tasks

are accomplished by the programmer. Programmers are responsible for management of the tasks computed by each process.

MPI consists of many communication models such as point-to-point, collective, one-sided, and parallel I/O operations. Point-to-point operations such as the "MPI_Send"/"MPI_Recv" pair facilitate communications between processes. Collective operations such as "MPI_Bcast" ease communications involving more than two processes. Regular MPI send/receive communication uses a two-sided model. This means that matching operations by sender and receiver is required. In the new versions of MPI, one-sided communications are also possible. One-sided communication decouples data transfer from synchronization and allows remote memory access. Three communication calls are provided: "MPI_Put" (remote write), "MPI_Get" (remote read), and "MPI_Accumulate" (remote update). Parallel I/O provides access to external devices exploiting data types and communicators [39].

2.4 Divisible Load Theory (DLT)

In recent times, the interest in network-based computing has grown significantly. Network-based computing consists of workstations or computers which are linked together through a communication network, forming a large, loosely coupled distributed computing system. This allows the use of shared resources and offers a user at any single node to exploit the considerable power of the complete network or a subset of it by partitioning and transferring its own processing load to the other processors in the network.

The two major approaches for designing parallel algorithms are function parallelism and data parallelism. Divisible Load Theory (DLT) is an application of data parallelism, in which data or load can be split and assigned to many processors. But the manner in which partitions can be created depends on the divisibility property of the data or load. A divisible load is the one that can be arbitrarily decomposed into smaller parts that do not have any interdependencies, i.e., they can be processed independently of each other. There are situations where a non-divisible (i.e., non-partition able) load "quantum" [42] exists, which have been addressed in [43]. The partitioning can occur at the beginning, or can be done dynamically when the computation is in progress and the computational requirements become clearer.

This framework of computing is best suited for applications which allow the partitioning of the processing load into smaller fractions or segments so that they can be processed independently.

DLT is applied when a single large load arrives at one of the nodes in the network. The processor or node partitions the load into more than one fraction, keeps one fraction for itself for processing, and sends the rest to other nodes in the network for processing. DLT is a powerful tool that provides polynomial time complexity [42] solutions to partitioning and scheduling problems.

An important issue is how to optimally partition the load between the processors so that computation is completed in the shortest possible time. There are two key limiting factors when applying the DLT approach for the optimal partitioning of the load or data: **(1)** the cost model employed, and **(2)** the number of load originating nodes (i.e., number of data sources) [42]. Most of the literature studies for DLT make use of linear cost models [44]; however, there are a few studies in which we consider start-up costs and other latencies [45, 46] while partitioning the data. Drozdowski and Wolniewicz in [46] used piecewise-affine models to account for the different speeds of a typical machine's memory hierarchy. Hung and Robertazzi in [47] made progress in the area by using quadratic and power-of-x computational cost models for multi and single-level trees, respectively, and predicted a superlinear speedup for nonlinear complexities [42]. According to [48], load or data partitioning becomes much more complicated when computational cost does not depend upon the data size. Although the applicability of DLT is not completely ruled out, it is something that would have to be examined on a case-by-case basis.

Most of the literature studied for DLT has also employed a single load origination node [42], with a few exceptions mentioned in [49-51]. The authors in [49] studied two scheduling strategies that partition the graphs representing the network joining sources (i.e., load origination loads) and sinks (i.e., workers) into disjoint subgraphs. However, there is a limitation for the proposed strategies that each of the sources carries a queue of individual loads. This hinders their application in the case of parallel filesystems or in cases where the sources share a load.

The same limitations are also applied to work presented in [51]. The authors in [51] suggests three resource-aware scheduling schemes that implicitly use multiple installments to process the loads which are present at the sources. The size of the installments is determined by the buffer space available to the workers at any given time.

In the next chapter, we discuss preliminaries, definitions of a distinguishing test case, and algorithms for determining the minimal length of an adaptive distinguishing experiment for a nondeterministic FSM.

Chapter 3: Determining the Minimal Length of Adaptive Distinguishing Experiments for Nondeterministic FSMs

This chapter includes preliminaries and definitions, mostly taken from [34], which can be used in the context of deriving/determining minimal length of an adaptive distinguishing experiment for a complete observable nondeterministic FSM. The chapter also includes two algorithms for determining the minimal length of an adaptive distinguishing experiment for a pair of initial states of a complete observable nondeterministic FSM.

3.1 Preliminaries

A *finite state machine* (FSM), or simply a *machine*, is a 4-tuple S = (S, I, O, h_S), where S is a finite nonempty set of states; I and O are finite input and output alphabets; and $h_S \subseteq S \times I \times O \times S$ is a (*behavior*) *transition relation*. An FSM is *nondeterministic* if, for some pair $(s, i) \in S \times I$, there can exist several pairs $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$. If the FSM has the designated initial state then the FSM is an *initialized* FSM, written (S, I, O, h_S, s_0). An FSM S is *complete* if for each pair $(s, i) \in S \in I$ there exists $(o, s') \in O \in S$ such that $(s, i, o, s') \in h_S$. An FSM S is *observable* if for each two transitions (s, i, o, s_1), $(s, i, o, s_2) \in h_S$ it holds that $s_1 = s_2$. An FSM S is *single-input* if at each state there is at most one defined input at the state, i.e., for each two transitions (s, i_1, o_1, s_1), $(s, i_2, o_2, s_2) \in h_S$ it holds that $i_1 = i_2$, and S is *output-complete* if, for each pair $(s, i) \in S \in I$ such that the input i is defined at state s, there exists a transition from s with i for every output in O. An initialized FSM S is *acyclic* if the FSM transition diagram has no cycles. An initialized FSM S is (*initially*) *connected* if each state is reachable from the initial state.

A *trace* of S at state s is a sequence of input/output pairs of sequential transitions starting from state s. The set of all traces of S at state s, including the empty trace, is denoted Tr(S/s). Let Tr(S/S'), $S' \subseteq S$, denote the union of Tr(S/s) over all states $s \in S'$. For state s and a sequence $\gamma \in (IO)^*$ of input-output pairs, the γ-*successor* of state s is the set of all states that are reached from s by γ. If γ is not a trace at state s, then the γ- successor of state s is the empty set. For an observable FSM S, the γ-successor

of s has at most one item. Given a nonempty subset S' of states of the FSM S, the γ-successor of S' is the union of the γ-successors over all $s \in S'$.

To characterize the common behavior of two weakly initialized machines, the operation of the intersection of initialized FSMs is extended as follows [34]. Given two complete FSMs S and P with the sets S' and P' of initial states, the *intersection* S ∩ P is the connected FSM Q such that states of Q are pairs (b, c) of sets of states of FSMs S and P. The initial state of Q is (S', P'), and h_Q is the smallest set and is derived using the following rule: given state (b, c), $b \subseteq S$ and $c \subseteq P$, and an input/output pair i/o, the FSM Q has a transition $((b, c), i, o, (b', c'))$ if there exist states $s \in b$ and $p \in c$ with an outgoing transition labeled by the pair i/o, and b' and c' are i/o–successors of subsets b and c. By definition, the FSM S ∩ P is observable even for non-observable FSMs S and P.

As an example of the FSM intersection, consider FSMs P (Figure 3) and S (Figure 4). FSM P is an initialized FSM while S has three initial states marked in bold. The intersection S ∩ P is shown in Figure 5. As usual, the intersection of two weakly initialized FSMs describes the common behavior of component FSMs, and in addition, it also provides some information about the structure of their transition sets. For example, a state of the intersection provides information about which states of the corresponding machines are reachable from the initial states under a corresponding trace.

In this thesis, we consider adaptive experiments with complete nondeterministic observable FSMs with an initial pair of states. An experiment can be described using an initialized single-input output-complete FSM with an acyclic transition graph that is usually referred to as a *test case* [34, 52].

Test Case:

Given an input alphabet I and an output alphabet O, a *test case* is an initially connected single-input output-complete observable initialized FSM P = (P, I, O, h_p, p_0) with the acyclic transition graph. A state of P that has no outgoing transitions is a deadlock state. Based on this definition, at each intermediate

state only a single input is defined with all outputs. A test case over alphabets I and O defines an adaptive experiment with any FSM S over the same alphabets.

In general, given a test case P, the *length* of the test case P is determined as the length of the longest trace from the initial state to a deadlock state of P and it specifies the length of the longest input sequence that can be applied to an FSM S during the experiment. As usual, for testing, one is interested in deriving a test case (experiment) with minimal length.

A test case P is a *distinguishing test case* for FSM S = (S, I, O, h_S, S') if **(1)** for each deadlock state (b, c) of the intersection S ∩ P, b is a singleton, and **(2)** for each transition ((b, c), i, o, (b', c')) of the intersection S ∩ P the subset b does not have two different states which have the same i/o–successor, i.e.,

$$\forall s_1, s_2 \in b\ ((s_1, i, o, s') \in h_S \ \&\ (s_2, i, o, s') \in h_S \Rightarrow s_1 = s_2).$$

In other words, a *distinguishing test case* can also be defined as: a test case P, over input and output alphabets I and O, is a *distinguishing test case* for the FSM S if every trace from the initial state to a deadlock state of P is a trace at most at one state of the set S. If there exists a distinguishing test case for the FSM S, then the set S is a *distinguishing* set, or the FSM is *distinguishable*, and the test case P is a *distinguishing test case* for the FSM. Otherwise, the FSM has no distinguishing set.

A test case P over alphabets I = {a, b} and O = {0, 1} is shown in Figure 3.

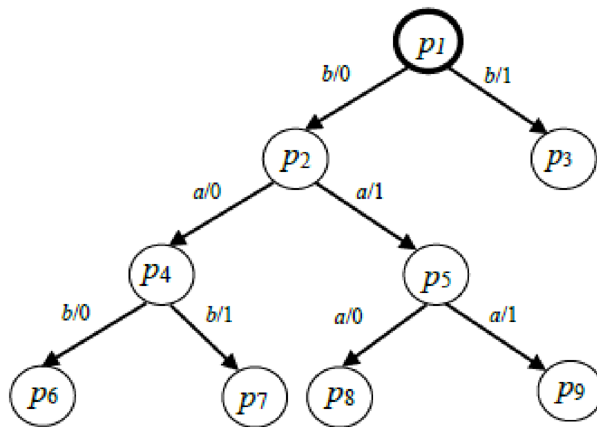

Figure 3. A Test Case P Over Alphabets I = {a, b} and O = {0, 1}

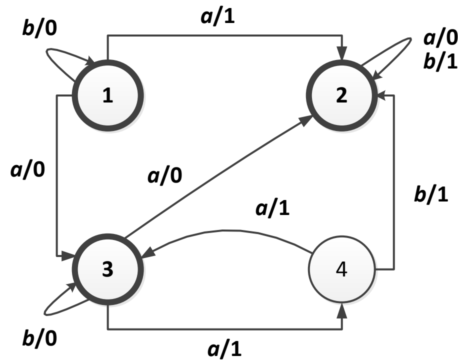

Figure 4. Considered FSM **S** with Three Initial States

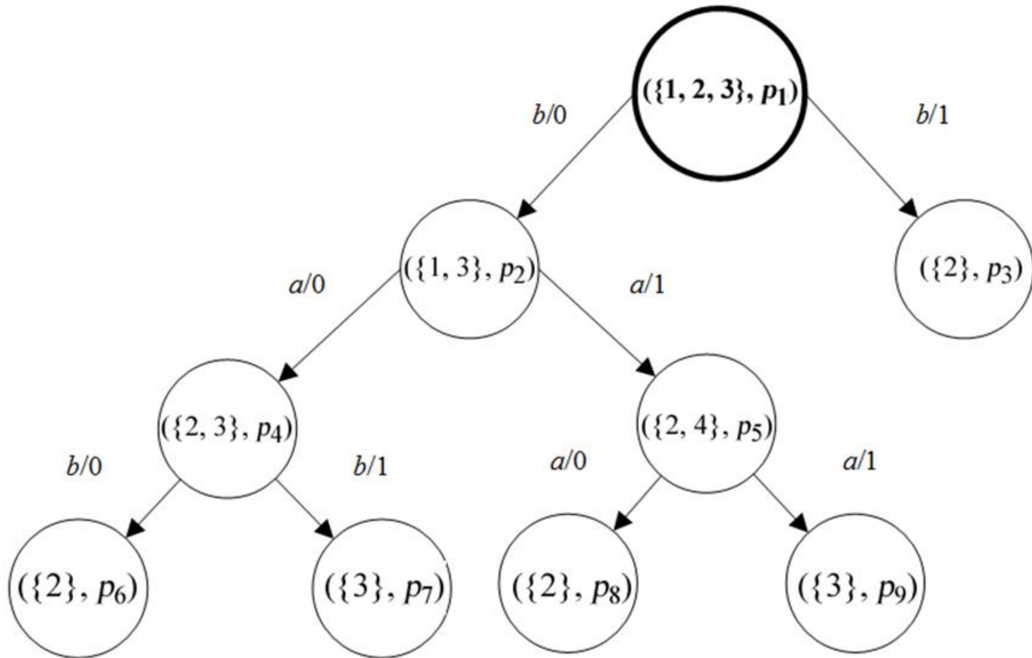

Figure 5. The Intersection **S** ∩ **P**

An example of a distinguishing test case is the weakly initialized FSM S presented in Figure 4 and the test case P presented in Figure. 3 By direct inspection, one can notice that each deadlock state of the intersection S ∩ P (Figure 5) is labeled by a pair of singletons and each two different states of any subset *b*, such that (*b*, *c*) labels an intermediate state of the intersection, do not have the same *i*/*o*-successor. Thus, the set {1, 2, 3} is a distinguishing set and the test case in Figure 3 is a distinguishing test case for the FSM S. For example, if the output 1 is produced to the input *b* at the initial state of the FSM S, then the FSM reaches state 2 after the experiment and we certainly know that the initial state before the experiment was 2.

A test case TC (I, O) over alphabets I and O defines an adaptive experiment with any FSM S over the same alphabets. As an example, consider the test case P in Figure 3. An adaptive experiment with an FSM S over alphabets $I = \{a, b\}$ and $O = \{0, 1\}$ is conducted using P as follows. At the first step the input b is applied to S, as this input is the only input defined at the initial state of P. If the output of the FSM S to this input is 1, then the experiment is over, since we reach the deadlock state p_3 of P. If the FSM S produces the output 0 to input b, then the experiment is not over, since the test case P enters the intermediate state p_2 where the single input a is defined. As this input does not take the test case to a deadlock state, the next input which is also a is applied. If the output to a is 0, then the next input is b; otherwise, the next input is a. For this example, the length of the longest trace of the test case is three, i.e., at most three inputs are applied during this adaptive experiment.

Given a complete observable FSM S $= (S, I, O, h_S)$, in order to derive a distinguishing test case with minimal length, the notion of *k-distinguishing* sets of states is usually introduced [52]. A subset $g \subseteq$ S is 0-*distinguishing* if g is a singleton. Let all $(k-1)$-distinguishing sets, $k > 0$, be already defined. A subset $g \subseteq S$ is a k-distinguishing set if **(1)** g is (k - 1)-distinguishing, or **(2)** there exists an input $i \in I$, such that for each $o \in O$, the i-o-successor of g is either empty or is a $(k-1)$ distinguishing set. In addition, the i-o-successors of two different states of g must not coincide. Given an observable complete FSM S, the set $g \subseteq$ S of states is k-distinguishing, $k > 0$, if and only if there exists a distinguishing adaptive experiment of length k for the set g. If S is k-distinguishing, $k > 0$, but is not $(k-1)$-distinguishing then k is the minimal length of a corresponding adaptive experiment.

3.2 Two Algorithms for Determining the Minimal Length of Adaptive Distinguishing Experiments for Complete Observable Nondeterministic FSMs

In this section, we describe two algorithms for determining the minimal length of adaptive distinguishing experiments for a complete observable nondeterministic FSM. The first algorithm is named Algorithm A, and the second is an alteration of A, named as Algorithm B. Algorithm A is taken from [34], and has been adapted for finding minimal length of adaptive experiments for a pair of initial states. Algorithm A derives the I/O-successors iteratively for each subset of states in the FSM S and

checks for the distinguishing sequence in the corresponding iteration. However, Algorithm B derives the I/O-successors for all the subsets of states in the FSM S in advance, and then it proceeds to check the distinguishing sequence for a given initial pair. In the following sections, we describe Algorithms A and B in detail and we provide an example for each.

3.2.1 Algorithm A (For Determining the Minimal Length of an Adaptive Distinguishing Experiment)

Distinguishability is defined as an experiment which allows one to determine the unknown current (initial) state of the machine under study. For a given FSM S = (S, I, O, h_S), the algorithm mentioned below can be used for deriving the minimal length for an adaptive distinguishing test case for a distinguishing set for an initial pair of states g in $S' \subseteq S$. In case the set g is not distinguishing, then the states contained in set S' cannot be distinguished by an adaptive experiment. The main idea of the procedure below is to iteratively derive subsets of states that are distinguished by adaptively applying an input sequence of the length $j \in 1, 2\ldots k$. The states of a test case under construction are labeled by subsets of S states.

In this section we describe an algorithm given in [34, 52] for determining the minimal length for an adaptive distinguishing experiment for complete observable nondeterministic FSMs. The algorithm given in [34] works for any number of pairs of initial states and as in this thesis, we target adaptive experiments for a pair of initial states. Thus, we re-write the algorithm given in [34] as Algorithm A given below:

Algorithm A

<u>**Description:**</u> Determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM **S**

Input: Complete observable nondeterministic FSM S = (S, I, O, h_S) with *initial* pair of states of S

Output: Minimal length k of a distinguishing sequence for the given *initial* pair of FSM S or a message "there is no adaptive distinguishing sequence for the pair".

Derive the set *Q* of all pairs of the set *S* of FSM S // *Q* represents pairs not distinguished

Step-1: //Set the values of variables *k*, *P* & *R* as follows:

$k = 1$:   // length of the sequence

Let the set *P* be empty; // represents pairs of *Q* that are already distinguished

Let the set *R* be empty;

Step-2:

Step-2.1:

For each pair, call it *current* in the set *Q*, do:

Step-2.2:

For each input $i \in I$:

Step-2.3: Derive the set of all *i-o-successors* of the *current* pair

Step-2.4: If the set of *i-o-successors* has a singleton then ***continue***

Step-2.5: If the set of *i-o-successors* of *current* is empty

{

If *current* is not *initial*,

Add *current* pair to *P* and ***break***

Else

Return *k* and **End Algorithm A (Exit)**

}

Step-2.6: If the set of *i-o-successors* is in *R*

{

If *current* is not *initial*,

Add *current* pair to *P* and ***break***

Else

Return *k* and **End Algorithm A (Exit)**

}

End-For

End-For

Step-3:

If $P = R$ then **End Algorithm A** and return message "there is no adaptive distinguishing sequence for the pair"

$Q = Q \setminus P$

$k = k + 1$

Let set $R = P$

Step-4: Go-to Step-2

Example 1: As an application example of Algorithm A, consider the FSM S in Figure 6 with four states, inputs $\{a, b\}$ and outputs $\{0, 1\}$. Applying Algorithm A, we proceed as follows: At Step 1, $Q = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$. Let *initial* = (1, 3). Also, we have $k = 1$; $P = \emptyset$; $R = \emptyset$.

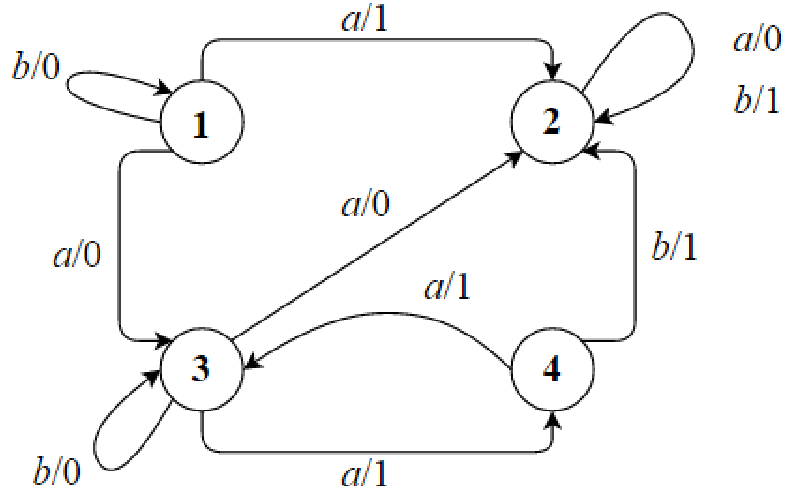

Figure 6. Considered FSM **S**

Table 2. Tabular Representation for the Considered FSM **S** in Figure 6.

Input\ State	**1**	**2**	**3**	**4**
a	**2** / 1 **3** / 0	**2** / 0	**2** / 0 **4** / 1	**3** / 1
b	**1** / 0	**2** / 1	**3** / 0	**2** / 1

At Step 2.1, let *current* = (1, 2) of Q. At Step 2.2, select input 'a'. At Step 2.3, derive the set of all *i/o-successors* of *current.* As the a-0-successors (1, 2) = {(2, 3)} and the a-1-successors (1, 2) = Ø, the a-successors (1, 2) = {(2, 3)}. At Step 2.4, the

set of a-successors (1, 2) is not a singleton, so proceed to Step 2.5. At Step 2.5, the set of a-successors (1, 2) is not empty, so proceed to Step 2.6. At Step 2.6, as the set of a-successors (1, 2) is not in *R*, then we go back to Step 2.2, select input 'b' and repeat Steps 2.3 to 2.6; derive the set of all *i/o-successors* of *current* where the b-0-successors (1, 2) = Ø and the b-1-successors (1, 2) = Ø; thus the b-successors (1, 2) = Ø. Since the set of b-successors (1, 2) is not a singleton, but the set of b-successors (1, 2) is empty and *current* is not *initial*; therefore add (1, 2) to *P* to obtain *P* = {(1, 2)}. Repeat Step 2 for the remaining subsets of states {(1, 3), (1, 4), (2, 3), (2, 4), (3, 4)} in the FSM S as follows. Let *current* = (1, 3) of *Q*, the set of a-successors (1, 3) = {(2, 3), (2, 4)}, and the b-successors (1, 3) = {(1, 3)}. As the set of all *i/o-successors* of *current* is not a singleton, not empty, and it is not present in the set *R*, thus we repeat Step 2 for the remaining subsets {(1, 4), (2, 3), (2, 4), (3, 4)} as follows. Let *current* = (1, 4) of *Q*, the set of a-successors (1, 4) = {(2, 3)}, and the b-successors (1, 4) = Ø. As the set of b-successors (1, 4) is not a singleton, but is empty, and *current* is not *initial*, add (1, 4) to *P* to obtain *P* = {(1, 2), (1, 4)}. We repeat Step 2 for remaining subsets {(2, 3), (2, 4), (3, 4)}. Let *current* = (2, 3) of *Q*, the set of a-successors (2, 3) = {(2, 2)}, and the b-successors (2, 3) = Ø. As the set of b-successors (2, 3) is not a singleton, but is empty, and *current* is not *initial*, add (2, 3) to *P* to obtain *P* = {(1, 2), (1, 4), (2, 3)}. We repeat Step 2 for remaining subsets {(2, 4), (3, 4)}. Let *current* = (2, 4) of *Q*, the set of a-successors (2, 4) = Ø. As the set of a-successors (2, 4) is not a singleton, but is empty and *current* is not *initial*, add (2, 4) to *P* to obtain *P* = {(1, 2), (1, 4), (2, 3), (2, 4)}. Repeat Step 2 for remaining subsets {(3, 4)}. Let *current* = (3, 4) of *Q*, the set of a-successors (3, 4) = {(3, 4)}, and the b-successors (3, 4) = Ø. As the set of b-successors (3, 4) is not a singleton, but is empty, and *current* is not *initial*, add (3, 4) to *P* to obtain *P* = {(1, 2), (1, 4), (2, 3), (2, 4), (3, 4)}. Since there are no further subsets remaining in the FSM S, we proceed to Step 3.

At Step 3, since *P* = {(1, 2), (1, 4), (2, 3), (2, 4), (3, 4)} is not equal to *Q* = {(1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (1, 4)}, we separate subsets of the states which have already been distinguished in Step 2 by subtracting *Q* from *P* such that we obtain *Q* = *Q**P* = {(1, 3)}. Also, we add 1 to the length *k* such that *k* = 2. Add all the distinguished subsets from *P* to *R* (*R* = *P* = {(1, 2), (1, 4), (2, 3), (2, 4), (3, 4)}).

At Step 4, since the solution is not found (i.e., length has not been determined yet), we go back to Step 2. Repeating Step 2, we proceed as follows: at Step 2.1, let *current* = (1, 3) of *Q*. At Step 2.2, select input ‘a’. Since the set of all *i/o-successors* of *current* has a-successors (1, 3) = {(2, 3), (2, 4)}, and a-successors (1, 3) is not a singleton, and it is not empty, but a-successors (1, 3) is present in the set *R* and *current* is *initial*, return $k = 2$ (the minimal length of the distinguishing sequence) and End Algorithm A.

3.2.2 Algorithm B (For Determining the Minimal Length of an Adaptive Distinguishing Experiment)

We modify Algorithm A so that instead of deriving *i/o-successors* for subsets of states iteratively as done in Steps 2.1 to 2.6 of Algorithm A, first we derive *i/o-successors* for all the subsets in advance. After the derivation of *i/o-successors* for all the subsets, we proceed to determine the minimal length of the distinguishing sequence by applying an adaptive input sequence of the length $j \in 1, 2\ldots k$. The reason for this variation is **(1)** to make derivation of *i/o-successors* independent from finding the solution, and **(2)** to check whether this independent derivation of *i/o-successors* can affect (i.e., either increase or decrease) the performance (i.e., the execution time) of Algorithm B to find the solution. The modified procedure is expressed in Algorithm B as given below:

Algorithm B

<u>Description:</u> Determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM **S**

Input: Complete observable nondeterministic FSM $\text{S} = (S, I, O, hs)$ with *initial* pair of states of *S*

Output: Minimal length *k* of a distinguishing sequence for the given *initial* pair of FSM S or a message “there is no adaptive distinguishing sequence for the pair”.

Derive the set *Q* of all pairs of the set *S* of FSM S // *Q* represents pairs not distinguished

Apply **Step-1** of Algorithm A // Initialization of variables

Step-1.1:

For each pair, call it *current* in the set Q, do:

Step-1.2:

For each input $i \in I$:

Derive the set of all *i-o-successors* of *current* pair

End-For

End-For

Step-2:

Step- 2.1:

For each pair, call it *current* in the set Q, do:

Step- 2.2:

For each input $i \in I$:

Apply **Step-2.4** of Algorithm A

Apply **Step-2.5** of Algorithm A

Apply **Step-2.6** of Algorithm A

End-For

End-For

Apply **Step-3 & Step-4** of Algorithm A

Example 2: As an application example of Algorithm B, consider the FSM $\mathbf{S_2}$ in Figure 7 with four states, inputs $\{a, b\}$, and outputs $\{0, 1\}$. Applying Algorithm B proceeds as follows: Apply Step 1 from Algorithm B, Q = {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}. Let *initial* = (0, 2). Also we have $k = 1$; $P = Ø$; $R = Ø$. Then we proceed to the next step, i.e., derivation of *i/o-successors* for all the subsets of states.

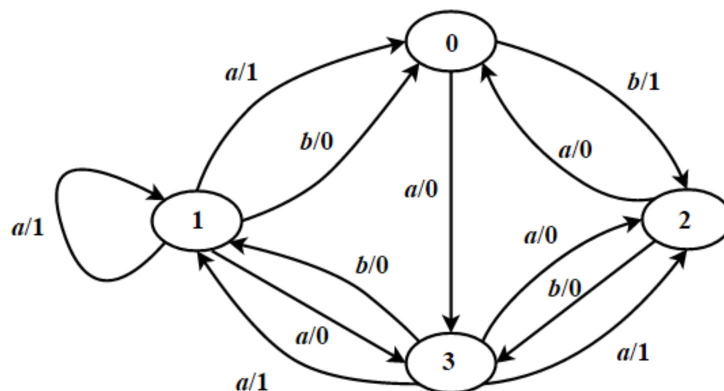

Figure 7. Considered FSM $\mathbf{S_2}$

Table 3. Tabular Representation for the Considered FSM $\mathbf{S_2}$ in Figure 7

Input\ State	**0**	**1**	**2**	**3**
a	**3** / 0	**0** / 1 **1** / 1 **3** / 0	**0** / 0	**2** / 0 **2** / 1 **1** / 1
b	**2** / 1	**0** / 0	**3** / 0	**1** / 0

At Step 1.1, set *current* = (0, 1) of *Q*. At Step 1.2, select input 'a'. Derive the set of all *i/o-successors* of *current.* As the a-0-successors = {(3, 3)} and a-1-successors = Ø, the a-successors (0, 1) = {(3, 3)}. Then we go back to Step 1.2 and select input 'b'. Derive the set of all *i/o-successors* of *current* where the b-0-successors = Ø and b-1-successors = Ø; thus the b-successors (0, 1) = Ø. The set of *i/o-successor* (0, 1) = {(3, 3)}. By repeating Step 1.1 for the remaining subsets of states {(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)} in the FSM S we obtain the following: *i/o-successors* (0, 2) = {(0, 3)}, *i/o-successors* (0, 3) = {(2, 3)}, *i/o-successors* (1, 2) = {(0, 3), (0, 3)}, *i/o-successors* (1, 3) = {(2, 3), (0, 1), (0, 2), (1, 1), (1, 2), (0, 1)}, and *i/o-successors* (2, 3) = {(0, 2), (1, 3)}. The derivation of *i/o-successors* for all the subsets (i.e., pairs) of the states in FSM S is completed, and we proceed to Step 2.

At Step 2.1, set *current* = (0, 1) of *Q*. At Step 2.2, select input 'a'. At Step 2.4, the set of a-successors (0, 1) is a singleton, so we go back to Step 2.2 and select input 'b'. As the set of b-successors (0, 1) is not a singleton, and is empty and *current* is not *initial*, add (0, 1) to *P* to obtain *P* = {(0, 1)}. We repeat Step 2 for the remaining subsets of states {(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)} in the FSM S as follows. Let *current* = (0, 2) of *Q*. As the set of a-successors (0, 2) is not a singleton, is not empty and is not present in the set *R*, we continue with the set of b-successors (0, 2). As the set of b-successors (0, 2) is not a singleton, is empty and *current* is *initial*, we return *k* = 1 (the minimal length of the distinguishing sequence) and End Algorithm B.

3.3 An Example Comparing Algorithms A and B

In this section, we compare the performance of both the sequential algorithms A and B. We demonstrate this through a simple example (i.e., Example 3) in which we re-apply Algorithm A on FSM $\mathbf{S_2}$ in Figure 7. We compare both examples (i.e., 2 and

3) and evaluate which algorithm (either A or B) gives the better performance in determining the minimal length of the distinguishing sequence.

Example 3: By applying Step 1 of Algorithm A on FSM $\mathbf{S_2}$, we proceed as follows: Let Q = {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}. Let *initial* = (0, 2). Also we have $k = 1$; $P = Ø$; $R = Ø$.

Recalling Step 2 from Algorithm A and by applying Steps 2.1 to 2.6, we proceed as follows. Let *current* = (0, 1) of Q, the set of a-successors (0, 1) = {(3, 3)}, and the b-successors (0, 1) = Ø. As the set of b-successors (0, 1) is not a singleton, but is empty and *current* is not *initial*, add (0, 1) to P to obtain P = {(0, 1)}. Repeat Step 2 for remaining subsets {(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)} as follows. Let *current* = (0, 2) of Q, the a-successors (0, 2) = {(2, 3)}, and the b-successors (0, 2) = Ø. As the set of b-successors (0, 2) is not a singleton, but is empty and current is *initial*, return k = 1 (the minimal length of the distinguishing sequence) and End Algorithm A.

After applying Algorithm B in Example 2 and Algorithm A in Example 3 for FSM $\mathbf{S_2}$ in Figure 7, we observe that the Algorithm A performs better than Algorithm B. Algorithm A performs better because it derives the I/O-successors for each subset (i.e., pair) of states in FSM S iteratively (i.e., this process might not derive all the I/O-successors) and then in the corresponding iteration it proceeds to check the solution (i.e., the length of the distinguishing sequence). However, by contrast, Algorithm B derives I/O-successors for all the subsets (i.e., pairs) of states in the FSM in advance. Once the derivation of I/O-successors is completed, only then it proceeds to check the solution (i.e., the length of the distinguishing sequence).

In Example 2 above, Algorithm B derives thirteen I/O-successors for all the subsets (i.e., pairs) of states in the FSM $\mathbf{S_2}$, thus iterating through all the transitions in FSM $\mathbf{S_2}$. Once the derivation of I/O-successors is completed, it proceeds to find the length of the distinguishing sequence for the given *initial* pair. However, on the other hand, Algorithm A in Example 3 only requires two iterations to find the solution. Algorithm A derives the I/O-successors for two subsets (i.e., pairs) of states in the FSM $\mathbf{S_2}$ and finds the distinguishing sequence in the second iteration, thus iterating through the minimum number of transitions required to find the solution. As a result, Algorithm B takes more time as it iterates through all the transitions to find the

solution, whereas Algorithm A finds the solution by iterating through the minimum number of transitions, resulting in better performance as compared to Algorithm B. However, in other cases as well, in which the length of the distinguishing sequence is greater than one (i.e. $k = 2, 3, \ldots$ and so on), the performance of Algorithm A should remain better as it iterates through the minimum number of transitions to find the solution, but in such cases where the length of the distinguishing is greater than one and we derive all the I/O-successors to obtain a solution, Algorithm B will give better performance than Algorithm A. Algorithms A and B will take the same amount of time to derive the I/O-successors, but Algorithm B will not iterate through transitions again to check the solution, resulting in better performance as compared to Algorithm A. Another added advantage of Algorithm B is that we develop independent derivation of I/O-successors, which in turn helps us to easily parallelize the process of I/O-successor derivation through a data decomposition/partitioning approach.

In the next chapter, we discuss the parallel algorithms for Algorithms A and B.

Chapter 4: Parallel Algorithms for Determining the Minimal Length of Adaptive Distinguishing Experiments for Nondeterministic FSMs

In this chapter, we discuss and describe parallel derivatives of the sequential algorithms A and B presented in Chapter 3. We present two algorithms: a parallel algorithm for A and a parallel algorithm for B. We present four different implementations of the Parallel Algorithm B, based on the different software and hardware platforms used. The software and hardware platforms used influence how the data is partitioned (i.e., distributed) for parallel execution and how the partial results (i.e., results obtained from the distributed computation) are handled.

The four implementations are as follows:

- MT^B: targets execution on a multicore CPU via multiple threads
- $CUDA^B$: targets execution on a GPU using the software platform CUDA.
- $Thrust^B$: targets execution on a GPU using the software platform Thrust.
- MN^B: targets execution on a Network of Workstations.

As Algorithm A derives the I/O-successors iteratively and checks for the solution in the corresponding iteration, this requires an inter-process and inter-thread communication in NoWs and in GPUs respectively. As inter-process and inter-thread communication is not possible in the current design of Algorithm A, the parallel algorithm for A has a single implementation on multi-core CPUs via multiple threads and from here on, it will be denoted as MT^A. As described above, MT^B, $CUDA^B$, $Thrust^B$, and MN^B are the parallel implementations of Algorithm B given in the previous chapter for deriving and determining the minimal length of an adaptive distinguishing experiment for non-deterministic FSMs and MT^A is the parallel implementation of Algorithm A. The purpose behind the development of the parallel algorithms is to reduce the execution time as observed in the sequential algorithms and obtain the solution (i.e., the length of the distinguishing sequence) in a reasonable time.

In the parallel algorithms we represent each subset (i.e., pair) of states of a given FSM by a unique integer value. For this purpose we present a function that maps all the pairs of states of FSM S to an integer representation.

Let $F(i,j)$ be the function, such that $F(i,j) \rightarrow \mathbb{N}$, which maps each subset (i.e., pair (i,j)) in FSM S to a unique integer value and is given by:

$$F(i,j) = \sum_{z=1}^{i-1} z + j + 1 \quad (1)$$
$$\forall\, i \neq j, \quad i > j$$

4.1 Multi-Threaded Implementation for Algorithm A (MT^{A})

In this section, we describe and discuss a parallel derivative of the (sequential) Algorithm A through its implementation on a multi-core CPU via multiple threads. In this algorithm, we divide the data (i.e., subsets (pairs) of states) by creating individual tasks and assigning each subset to the individual task. These individual tasks are then executed in parallel depending upon the number of threads available in the multi-core CPU and are scheduled by the scheduler of the CPU. This algorithm/implementation is specifically designed for execution on a multicore CPU via multiple threads.

Multi-Threaded implementation for Algorithm A (MT^{A}):

Description: A parallel (multi-threaded) variation of Algorithm A for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM $S = (S, I, O, hs)$ with *initial* pair of states of S.

Output: Minimal length k of a distinguishing sequence for the given *initial* pair of FSM S or a message "there is no adaptive distinguishing sequence for the pair".

Step-1:

Derive the set Q consisting of all pairs $(i,j)\ \forall\, i \neq j$, of the set S of FSM S. // set Q represents pairs not distinguished yet.

Let Q' be the ordered set that contains integer values representing the state of pairs (i,j) of the set Q, such that:

$$Q' = F(i,j) \ \ \forall\, i \in S, j \in S, i \neq j$$

$k = 1$; // length of the sequence

Let the set P be empty; // represents pairs of Q' that are already distinguished

Let the set R be empty;

Let M be the number of pairs of different states of S, calculate the total number of pairs of different states in S and assign it to M.

Step-2:

For each pair call it *current* in the set Q', do:

Create task T_v (where $v = 1\ldots.. M$) and assign *current* pair to each task T_v.

// T_v is an independent running task and does not relate to a process or a thread.

End-For

Do in Parallel

For each Task T_v call it *current task*, do the following:

For each input $i \in I$:

Derive the set of all *i-o-successors* of the pair in *current task*

If the set of *i-o-successors* has a singleton then ***continue***

If the set of *i-o-successors* of *current* is empty

{

If the pair in the *current task* is not *initial*,

Add the pair to P and ***break***

Else

Return k (Stop all the asynchronous running tasks) and **End Algorithm MT^A (Exit)**

}

If the set of *i-o-successors* is in R

{

If the pair in the *current task* is not *initial*,

Add the pair to P and ***break***

Else

Return k (Stop all the asynchronous running tasks) and **End Algorithm MT^A (Exit)**

}

End-For

End-For

End Parallel Execution

Step-3:

If $P = R$ then **End Parallel Algorithm MT^{A}** and return message "there is no adaptive distinguishing sequence for the pair"

$Q' = Q' \setminus P$

$k = k + 1$

Let set $R = P$

Step-4:

Go-to Step-2

4.2 Parallel Algorithm for B

In this section, we discuss and describe a generic parallel derivative of the (sequential) Algorithm B. This parallel algorithm is specifically designed for execution in parallel via different data partitioning approaches.

Parallel Algorithm B

<u>Description:</u> A parallel algorithm of Algorithm B for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM $S = (S, I, O, h_S)$ with *initial* pair of states of S.

Output: Minimal length k of a distinguishing sequence for the given *initial* pair of FSM S or a message "there is no adaptive distinguishing sequence for the pair".

Step-1:

Derive the set Q consisting of all pairs $(i, j) \forall\, i \neq j$ of the set S of FSM S. // set Q represents pairs not distinguished yet.

Let Q' be the ordered set that contains integer values representing the state of pairs (i, j) of set Q, such that:

$$Q' = \boldsymbol{F}(i, j) \quad \forall\, i \in S, j \in S, i \neq j$$

$k = 1$; // length of the sequence

Let the set P be empty; // represents pairs of Q' that are already distinguished

Let the set *R* be empty;

Let *M* be the number of pairs of different states of *S*, calculate the total number of pairs of different states in *S* and assign it to *M*.

Step-1.1: // Divide the data depending upon the data partitioning scheme

Divide:

Divide set *Q′* into disjoint subsets of ordered sets:

Step-1.2:

Do in Parallel:

For each subset of *Q′* do the following:

For each pair, call it *current* in the subset of *Q′*, do:

Derive the set of all *i-o-successors* of *current* pair

End-For

End-For

End Parallel

Step-1.3:

Join: // Join partial subsets created in Step 1.1 in the set *Q′*.

Step-2:

For each pair, call it *current* in set *Q′*, do:

For each input $i \in I$:

If the set of *i-o-successors* has a singleton then ***continue***

If the set of *i-o-successors* of *current* is empty

{

If *current* is not *initial*,

Add *current* pair to *P* and ***break***

Else

Return *k* and **End Parallel Algorithm B (Exit)**

}

If the set of *i-o-successors* is in *R*

{

If *current* is not *initial*,

Add *current* to *P* and ***break***

Else

Return k and **End Parallel Algorithm B (Exit)**

}

End-For

End-For

Step-3:

If $P = R$ then **End Parallel Algorithm B** and return message "there is no adaptive distinguishing sequence for the pair"

$Q' = Q' \setminus P$

$k = k + 1$

Let set $R = P$

Step-4:

Go-to Step-2

4.2.1 Multi-Threaded Implementation for Parallel Algorithm B (MT^B)

Below, we present a multi-threaded implementation of Parallel Algorithm B. In this implementation, we partition the data (i.e., subsets (pairs) of states) over a multi-core CPU via multiple threads. This parallel implementation is specifically designed for execution on a multi-core CPU via multiple threads as shown below.

Multi-Threaded Implementation (MT^B)

Description: A multi-threaded implementation of Parallel Algorithm B for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM $S = (S, I, O, hs)$ with *initial* pair of states of S, number of threads x.

Output: Minimal length k of a distinguishing sequence for the given *initial* pair of FSM S or a message "there is no adaptive distinguishing sequence for the pair".

Apply **Step-1** of Parallel Algorithm B

Modifying **Step-1.1, Step-1.2 & Step-1.3** of Parallel Algorithm B as below:

Step-1.1: // Dividing set Q' on multi-core CPU via multiple threads

Divide:

Divide the set Q' into disjoint subsets of ordered sets;

$Q'_1 \ldots\ldots Q'_x$ (where x is number of threads)

Let $G(y)$ be the function, such that $G(y) \rightarrow \mathbb{N}$, which calculates the index for the first item, (i.e., pair of states of S) of the ordered set Q'_x, and is given by:

$$G(y) = \begin{cases} 1 & if\ y = 1 \\ H(y-1)+1 & if\ y > 1 \end{cases} \quad (2)$$

where $y = 1, 2, \ldots x$

Let $H(y)$ be the function, such that $H(y) \rightarrow \mathbb{N}$, which calculates the index for the last item (i.e., pair of states of S) of the ordered set Q'_x, and is given by:

$$H(y) = \begin{cases} \frac{|Q'|}{x} & if \quad y = 1 \\ G(y) + \frac{|Q'|}{x} - 1 & if \quad 1 < y < x \\ |Q'| & if \quad y = x \end{cases} \quad (3)$$

where $y = 1, 2, \ldots x$ and $|Q'|$ is the size of set Q'

Step-1.2:

Do in Parallel

For each subset Q'_v (where v = 1….. x) do the following:

Let m_v be the first item (i.e., pair of states of S) of the ordered set Q'_v then:

$$m_v = G(v)$$

Let n_v be the last item (i.e., pair of states of S) of the ordered set Q'_v then:

$$n_v = H(v)$$

Let $current = m_v$;

While (*current* $\leq n_v$) do:

Derive the set of all *i-o-successors* of *current* pair

current++;

End-while

End-for

End Parallel

Step-1.3:

Join: // Join partial subsets created in Step 1.1 in the set *Q'*.

Apply **Step-2, Step-3 & Step-4** of Parallel Algorithm B

4.2.2 GPU Implementations for Parallel Algorithm B (CUDA^B and Thrust^B)

In this section, we present two implementations for Parallel Algorithm B on a GPU. In these implementations, we partition the data (i.e., subsets (pairs) of states) over the GPU cores depending upon the computing capability of the GPU device. The execution on the GPU can be carried out by using either of two software platforms: CUDA or Thrust.

GPU Implementation for execution on CUDA (CUDA^B)

Description: An implementation of Parallel Algorithm B on a GPU using CUDA for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM S = (S, I, O, hs) with *initial* pair of states of S.

Output: Minimal length k of a distinguishing sequence for the given *initial* pair of FSM S or a message “there is no adaptive distinguishing sequence for the pair”.

Apply **Step-1** of Parallel Algorithm B

Modify **Step-1.1, 1.2 & Step-1.3** of Parallel Algorithm B as below:

Step-1.1: // Dividing set *Q'* on the GPU by following kernel configurations:

Divide:

Divide set Q' into disjoint subsets of ordered sets, such that the number of disjoint subset equals the number of pairs *M*, and each disjoint subset contains a single pair (i, j):

$Q'_1 \ldots\ldots Q'_M$ (where *M* is the number of pairs)

Create threads *t* on the GPU such that the number of threads equals the number of disjoint subsets.

Assign each subset Q'_V to a corresponding thread t_V (where $V = 1\ldots\ldots M$).

Step-1.2:

Do in parallel

For each subset Q'_V (where $V = 1\ldots\ldots M$), do the following on the **GPU**:

Derive the set of all *i-o-successors* of the pair (i, j) in the subset Q'_V

End-for

End Parallel

Step-1.3:

Join: // Join partial subsets created in Step 1.1 in the set Q'.

Apply **Step-2, Step-3 & Step-4** of Parallel Algorithm B

GPU Implementation for execution on Thrust (Thrust^B)

Description: An implementation of Parallel Algorithm B on a GPU using Thrust for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM S = (*S*, *I*, *O*, *hs*) with *initial* pair of states of S.

Output: Minimal length *k* of a distinguishing sequence for the given *initial* pair of FSM S or a message “there is no adaptive distinguishing sequence for the pair”.

Apply **Step-1** of Parallel Algorithm B

Modify **Step-1.1, Step-1.2 & Step-1.3** of Parallel Algorithm B as below:

Step-1.1: // Dividing set Q' on the GPU through Functor using the following steps:

Divide:

Divide set Q' into disjoint subsets of ordered sets, such that the number of disjoint subsets equals the number of pairs *M* and each disjoint subset contains a single pair (i, j):

$$Q'_1 \ldots\ldots Q'_M \text{ (where } M \text{ is the number of pairs)}$$

Step-1.2:

Create a Functor and assign all the disjoint subsets to the functor which performs the following operations:

Start Functor:

Assign each subset Q'_V to a corresponding thread t_V (where $V = 1\ldots\ldots M$).

Do in parallel

For each subset Q'_V (where $V = 1\ldots\ldots M$), do the following on the **GPU**:

Derive the set of all *i-o-successors* of the pair (i, j) in the subset Q'_V

End-for

End Parallel

End Functor

Step-1.3:

Join: // Join partial subsets created in Step 1.1 in the set Q'.

Apply **Step-2, Step-3 & Step-4** of Parallel Algorithm B

4.2.3 Multiple-Node Implementation of Parallel Algorithm B (MN^B)

In this section, we present an implementation of Parallel Algorithm B on a NoW via multiple nodes. In this implementation, we partition the data (i.e., subsets (pairs) of states) over multiple nodes (*N*) in the NoW. We consider node computation capability, communication speed amongst the nodes, communication overhead, and constant overhead associated with the computation at the node. In order to partition the data optimally, we formulate a model which uses Divisible Load Theory (DLT), described in detail in Chapter 2. In order to attain optimum execution time, we take

into consideration that all the nodes must finish on time. The formulation of the model is shown below:

In the data partitioning model we make the following assumptions:

1. All the nodes read the machine description (N-port setup).
2. The master node collects the partial results and solves the problem by finding the length of the distinguishing sequence.

The following are notations which are used in the formulation.

- b_d: constant communication overhead during distribution (sec)
- l_d: (inverse) communication speed during distribution (sec/byte)
- b_c: constant communication overhead during result collection (sec)
- l_c: (inverse) communication speed during result collection (sec/byte)
- p_i: (inverse) computation speed of node P_i (sec/transition)
- e_i: constant overhead associated with the computation at node Pi (sec)
- $part_i$: part of load assigned to node P_i
- B: size of machine description in bytes
- T: number of transitions
- O: size of I/O successors table (bytes)
- N: number of nodes
- t^i_{distr}: Distribution time for the node i
- t^i_{comp}: Computation time for the node i
- t^i_{coll}: Collection time for the node i

The formulation for the model is presented below:

∀ node $i \in [1, N-1]$ compute:

$$t^i_{distr} = l_d \cdot B + b_d \quad (4)$$

$$t^i_{comp} = part_i \cdot p_i \cdot T + e_i \quad (5)$$

$$t^i_{coll} = l_c\, part_i \cdot O + b_c \quad (6)$$

For the master node we have $t^0_{coll} = 0$.

To minimize the overall execution time, all the nodes must finish at the same time. Thus we have:

$$t^0_{distr} + t^0_{comp} = t^i_{distr} + t^i_{comp} + t^i_{coll} \Rightarrow$$

$$l_d.B + b_d + part_0 p_0 T + e_0 = l_d.B + b_d + part_i p_i T + e_i + l_c.part_i + b_c \Rightarrow$$

$$part_0 p_0 T + e_0 = part_i p_i T + e_i + l_c.part_i O + b_c \Rightarrow$$

$$part_0 p_0 T + e_0 - e_i - b_c = part_i (p_i T + l_c O) \Rightarrow$$

$$part_i = part_0 \frac{p_0 T}{p_i T + l_c O} + \frac{e_0 - e_i - b_c}{p_i T + l_c O} \tag{7}$$

From the normalization equation we can calculate $part_0$:

$$\sum_{i=0}^{N-1} part_i = 1 \Rightarrow$$

$$part_0 \left(1 + \sum_{i=1}^{N-1} \frac{p_0 T}{p_i T + l_c O}\right) + \sum_{i=0}^{N-1} \frac{e_0 - e_i - b_c}{p_i T + l_c O} = 1 \Rightarrow$$

$$part_0 = \frac{1 + \sum_{i=1}^{N-1} \frac{e_i + b_c - e_0}{p_i T + l_c O}}{1 + \sum_{i=1}^{N-1} \frac{p_0 T}{p_i T + l_c O}} \tag{8}$$

From the model above, $part_i$ (where i = 0, 1, 2… N-1) represents computation load for a particular node; in particular, $part_0$ represents the load for the master node, and $part_i$ represents the percentage for the number of subsets (pairs) of the states that each node should compute for deriving I/O-successors. Appendix A provides more information about how the model parameters can be derived.

After computing the load for each node, $part_i$ can be applied as an input to the implementation, so that each node can be assigned with the number of subsets (pairs) of states for the derivation of I/O-successors.

We consider two types of nodes in the NoW: sequential nodes and nodes containing GPUs. To identify the type of each node, we apply node type as an input parameter in the implementation.

This implementation is well suited for execution on a NoW via multiple-node execution, and is shown below.

Multiple-Node Implementation (MN^B)

Description: An implementation of Parallel Algorithm B on a NoW via multiple-nodes for determining the minimal length of an adaptive distinguishing experiment for a pair of states of an FSM S

Input: Complete observable nondeterministic FSM $S = (S, I, O, h_S)$ with *initial* pair of states of S, $part_i$ (the computation load for each node *N*), type of node (i.e. either Sequential or GPU), number of nodes *N*.

Output: Minimal length *k* of a distinguishing sequence for the given *initial* pair of FSM S or a message "there is no adaptive distinguishing sequence for the pair".

Apply **Step-1** of Parallel Algorithm B

Modify **Step-1.1, Step-1.2 & Step-1.3** of Parallel Algorithm B as shown below:

Step-1.1: // Dividing set Q' on the NoW via multiple nodes on the Master Node

Divide:

Divide the set Q' into disjoint subsets of ordered sets;

$Q'_0 \ldots\ldots Q'_{N-1}$ (where *N* is number of nodes)

Let $U(y)$ be the function, such that $U(y) \rightarrow \mathbb{N}$, which calculates the index for the first item, (i.e., pair of states of S) of the ordered set Q'_{N-1}, and is given by:

$$U(y) = \begin{cases} 1 & if\ y = 0 \\ W(y-1)+1 & if\ y > 0 \end{cases} \tag{9}$$

where $y = 0,1,2 \ldots N-1$

Let $W(y)$ be the function, such that $W(y) \rightarrow \mathbb{N}$, which calculates the index for the last item (i.e., pair of states of S) of the ordered set Q'_{N-1}, and is given by:

$$W(y) = \begin{cases} round(M \times part_y) & if \quad y = 0 \\ U(y) + round(M \times part_y) - 1 & if \quad 1 < y < N-1 \\ M & if \quad y = N-1 \end{cases} \tag{10}$$

where $y = 0,1,2 \ldots N-1$ and M is the number of pairs of states S

For each $v \in N$ in [0, N – 1], do the following:

Let m_v be the first item (i.e., pair of states of S) of the ordered set Q' then:

$$m_v = U(v)$$

Let n_v be the last item (i.e., pair of states of S) of the ordered set Q'_v then:

$$n_v = V(v)$$

End-for

Step-1.2:

Do in Parallel

For each subset Q'_v (where v = 0...N-1), do on multiple nodes the following:

If the node is a Sequential Node

Apply **Step-2** of Algorithm B on subset Q'_v

Else if the node is a GPU Node

Apply **Step-1.1, Step-1.2 & Step-1.3** of $CUDA^B$ on subset Q'_v

End-for

End Parallel Execution

Step-1.3:

Join: // Join partial subsets created in Step 1.1 at the Master Node in the set Q'.

Apply **Step-2, Step-3 & Step-4** of Parallel Algorithm B

In the next chapter, we present the results of the experimental evaluation of the sequential algorithms presented in Chapter 3 and parallel algorithms presented in this chapter. For each algorithm, we record the execution time that it takes to determine the minimal length for a distinguishing sequence. Based on the recorded execution time for each algorithm, we perform related analyses as described in detail in Chapter 5.

Chapter 5: Experimental Evaluation

In this chapter, we present the results of the sequential and parallel implementations of algorithms described in Chapters 3 and 4. We conducted many experiments by generating different FSMs by a generator used in [53] with all the possible combinations shown in Table 4. For each combination we generated five FSMs and for each generated FSM we considered five different initial pairs. Therefore in total we experimented with 2000 machines (i.e., 80 combinations, 5 FSMs for each combination, and 5 different initial pairs for each FSM). For each experiment, we run sequential and parallel algorithms and determine the minimal length of an adaptive experiment. We compare (a) the time taken by each algorithm to reach a solution, (b) the speedup (defined as how much faster the parallel algorithm is in comparison to the sequential algorithms).

Table 4. Combinations of Generated FSMs

Inputs	Outputs	States	Determinism	Range
4	4	100	50	50-60
6	6	150	60	60-70
8	8	200	70	70-80
10	10	250	80	80-90
	12			

The system configuration and platform details of the test beds on which the experiments were conducted are shown in Table 5.

Table 5. System Configuration & Platform Details

	Dune	Kingpenguin	Dune2
CPU	Core(TM) 2 Quad CPU Q8200 @ 2.33 GHz	Intel(R) Xeon(R) CPU E5-2640 @ 2.50GHz	Intel(R) Core(TM) i7-4820K CPU @ 3.70GHz
CPU Cores	4	12	4
Threads/Core	1	2	2
RAM	4 GB	64 GB	32 GB
GPU	Quadro 5000	-	GeForce GTX 770
GPU Cores	352	-	1536
GPU RAM	2559 MBytes	-	2048 MBytes
Compute Capability	2.0	-	3.0
Number of GPUs	1	-	2

The software environment is the same for all the test beds in Table 5 and is mentioned below:

- Operating System: Kubuntu 14.04 (64 bit)
- Environment (IDE): Qt Creator 3.0.1
- Qt Version: Qt 5.2.1
- Compiler: GCC 4.8.2, 64 bit
- QMake version 3.0
- CUDA Driver Version / Runtime Version 6.5 / 6.5

Sequential algorithms A and B with corresponding parallel implementations MT^A, MT^B, $CUDA^B$, and $Thrust^B$ are tested on Dune as shown in Table 5. However, the parallel implementation MN^B is tested on the NoW. For this purpose, we consider three machines as shown in Table 5, which are inter-connected and correspond to multiple nodes in the network. For the execution of MN^B, we consider three CPU nodes and three GPU nodes which are listed below:

CPU Nodes:

- Dune
- Kingpenguin
- Dune2

GPU Nodes:

- Dune
- Dune2

Dune2 is considered as two GPU nodes because it consists of multiple (i.e., two) GPUs in its hardware specifications, and these GPUs are capable of independent concurrent executions.

In the sections below we compare the proposed algorithms and their implementations to determine which one gives the best performance under different circumstances. Special consideration is given to how the number of transitions (i.e., the size) and non-determinism of the FSM affects the overall execution time. For this purpose, we considered FSMs in three categories based on their sizes (i.e., number of transitions) which are as follows:

- Small FSMs: number of transitions ranges from 100,000 to 1 million.
- Medium FSMs: number of transitions ranges from 1 million to 1.5 million.
- Big FSMs: number of transitions ranges from 1.5 million to 5 million.

5.1 Execution Time versus Number of Transitions of Sequential Algorithms A and B

In this section, we compare the execution times of the two sequential algorithms (A and B) and determine which one gives the best performance as the number of transitions increases. Figures 8 and 9 depict the results for all the experiments conducted for sequential algorithms.

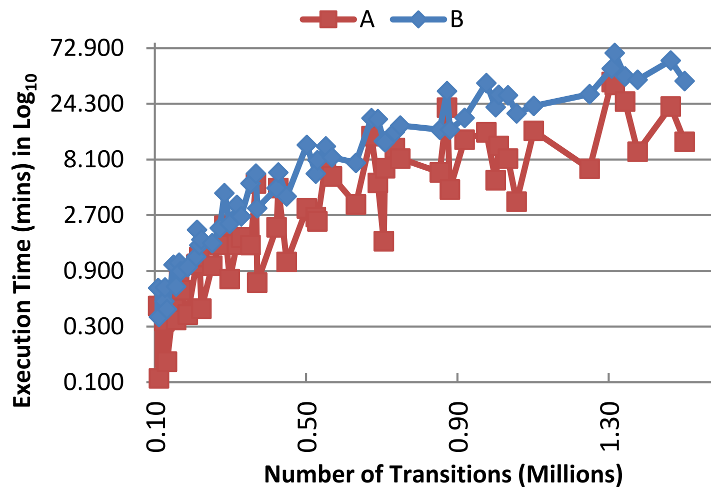

Figure 8. Algorithm A versus Algorithm B for Small and Medium FSMs

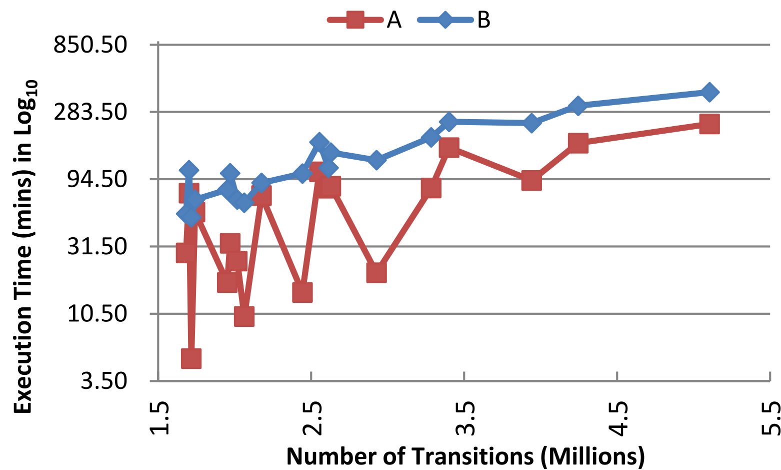

Figure 9. Algorithm A versus Algorithm B for Big FSMs

According to the results depicted in Figures 8 and 9, the similarity between both sequential algorithms is that the execution time increases exponentially with an increasing number of transitions. However, the performance of Algorithm A is better than Algorithm B. We also observe that in special cases where the length of the

distinguishing sequence is one (i.e. the distinct points in the Figure 9) the execution time of Algorithm A is significantly better than Algorithm B.

5.2 Execution Time versus Number of Transitions of Sequential Algorithm A Against MT^A

In this section, we compare the execution times of (sequential) Algorithm A with the multi-threaded implementation of Parallel Algorithm A. The purpose of this comparison is to analyze the difference in execution time between them and quantify the speedup that can be achieved. The results are summarized in Figures 10 and 11.

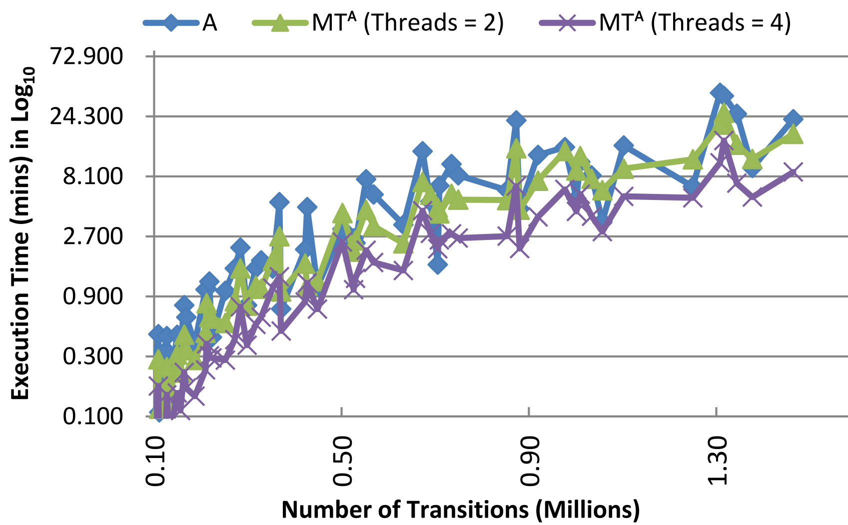

Figure 10. Algorithm A versus MT^A for Small and Medium FSMs

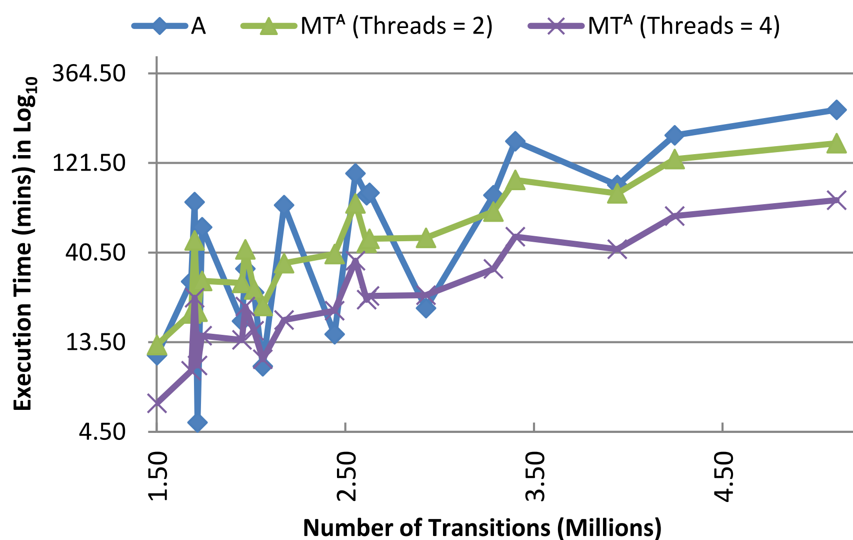

Figure 11. Algorithm A versus MT^A for Big FSMs

According to the results depicted in Figures 10 and 11, both algorithms exhibited an exponential increase in execution time with number of transitions. However, the speedup obtained is less favorable than MT^B as mentioned later in Section 5.3.

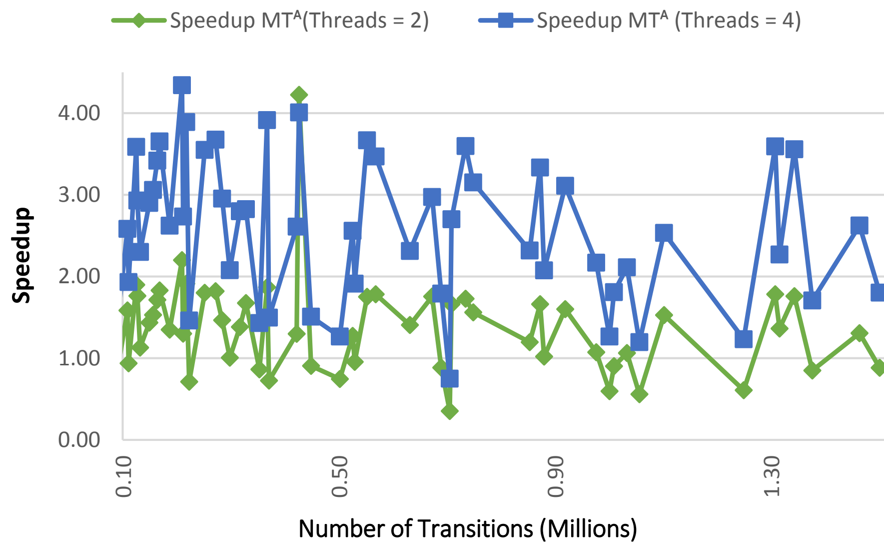

Figure 12. Speedup for MT^A w.r.t. (Sequential) Algorithm A for Small and Medium FSMs

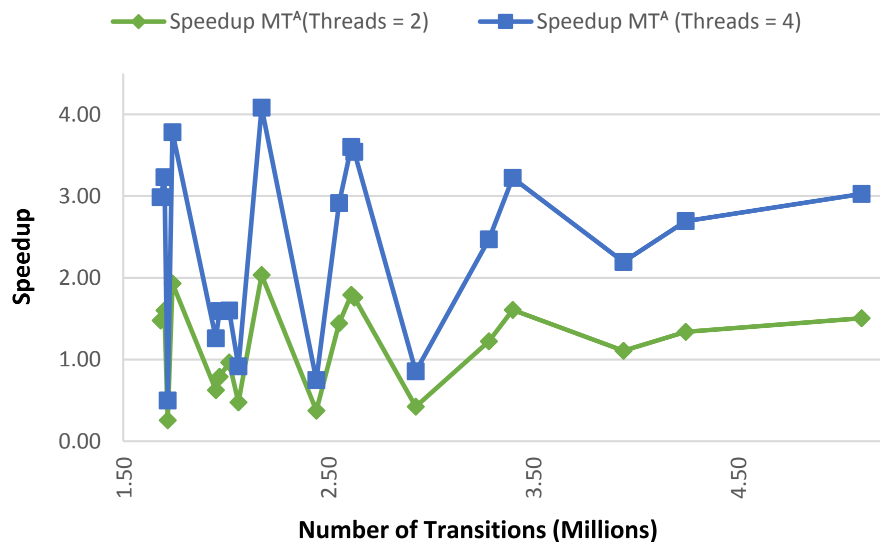

Figure 13. Speedup for MT^A w.r.t. (Sequential) Algorithm A for Big FSMs

Figures 12 and 13 illustrate a wide variation in the speedup. In the cases where the speedup for MT^A did not even scale above one, the length of the distinguishing

sequence is one and the sequential algorithm gives the better performance as compared to MT^A.

The variation in speedup is due to the varying length of the distinguishing sequence and I/O-successors derived. Since the (sequential) Algorithm A derives I/O-successors iteratively for each subset of states and then proceeds to check the solution, there is a possibility that it finds the solution before iterating through all the subsets and transitions in the FSM, especially when the solution (i.e., the length of the distinguishing sequence) is one. Therefore when the length of the distinguishing sequence is one, MT^A is much more costly in terms of execution time than (sequential) Algorithm A.

5.3 Execution Time versus Number of Transitions of Sequential Algorithm B Against MT^B

In this section, we compare the execution times for (sequential) Algorithm B and the multi-threaded implementation of Parallel Algorithm B. The purpose of this comparison is to analyze the difference in execution time between them and determine whether we attain speedup in multi-threaded implementation. Figures 14 and 15 depict the results for all the experiments conducted for sequential and parallel multi-threaded implementation of Algorithm B.

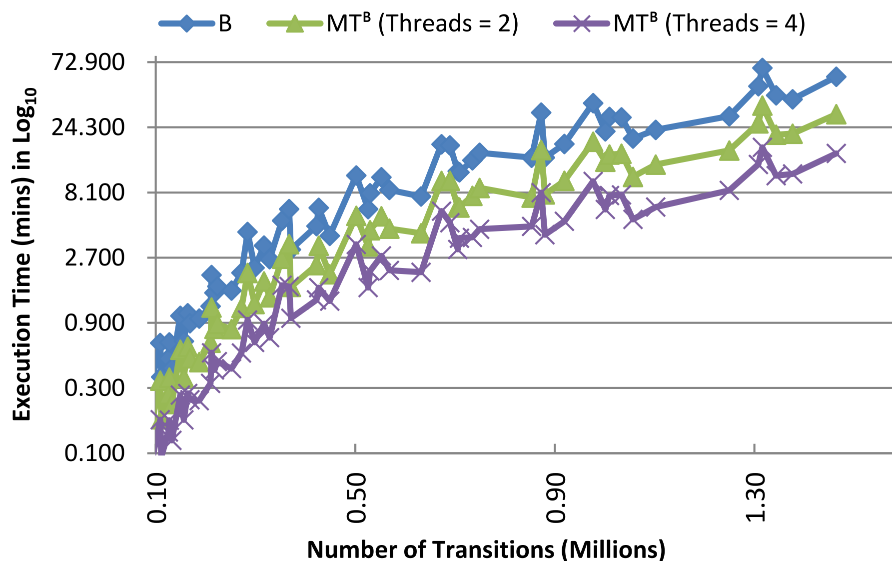

Figure 14. Sequential Algorithm B versus MT^B for Small and Medium FSMs

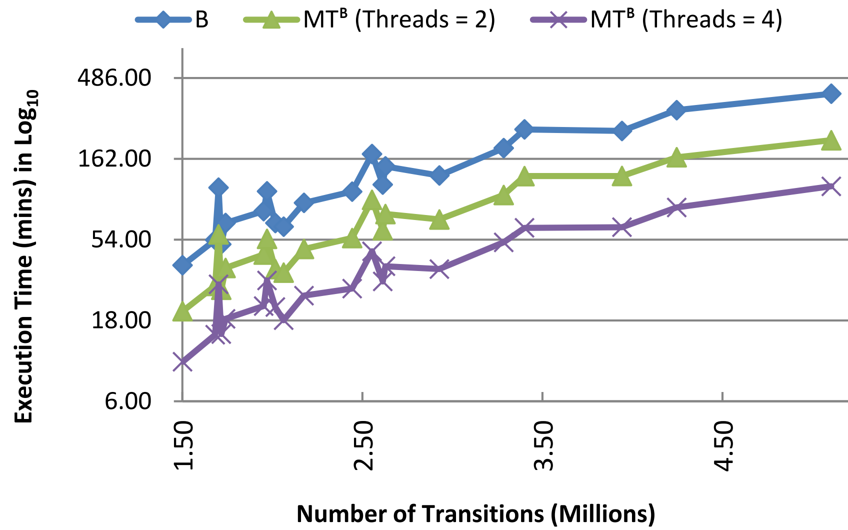

Figure 15. Sequential Algorithm B versus MT^B for Big FSMs

According to the results depicted in Figures 14 and 15, the similarity between both algorithms is that the execution time for both of them scales exponentially with an increasing number of transitions.

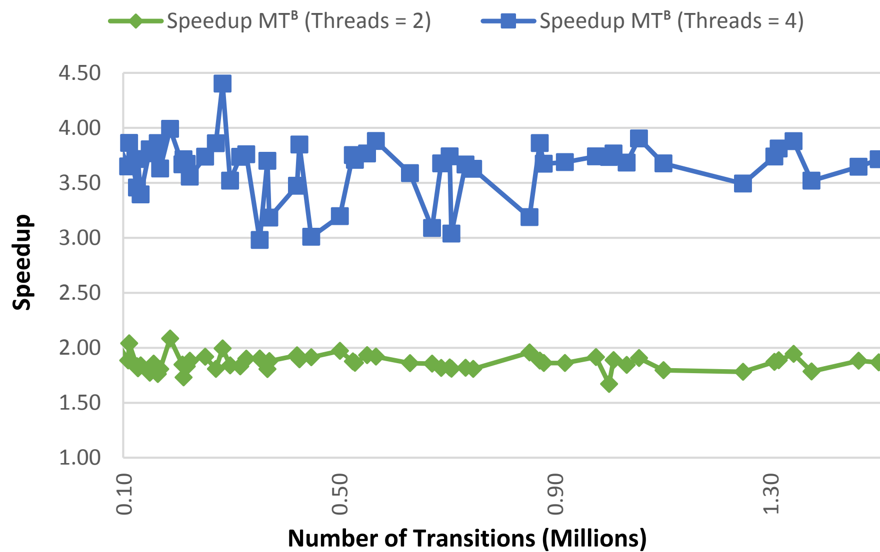

Figure 16. Speedup for MT^B w.r.t. (Sequential) Algorithm B for Small and Medium FSMs

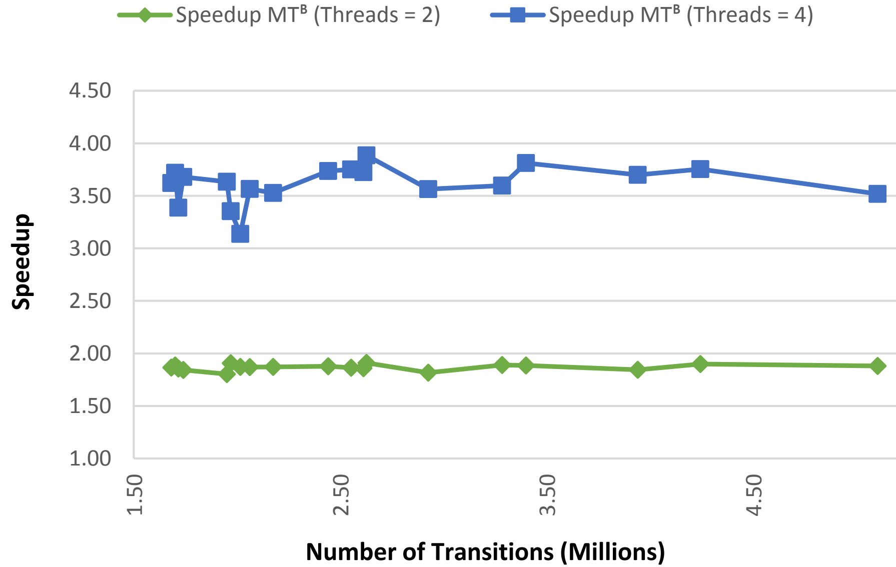

Figure 17. Speedup for MT^B w.r.t. Sequential Algorithm B for Big FSMs

According to the results depicted in Figures 17 and 18, we obtain nearly 2x speedup for two threads and nearly 4x speedup for four threads as compared to (sequential) Algorithm B.

The implementation of MT^B is based on equal partitioning of the problem data (i.e., subsets (pairs) of states are divided equally amongst the threads); therefore all the threads share the load equally, and we obtain a stable improvement in execution time.

5.4 Execution Time versus Number of Transitions for Sequential Algorithms Against Other Parallel Implementations

In this section, we compare the execution time for sequential algorithms A and B with other parallel implementations which include GPU implementations (CUDA^B and Thrust^B) and multiple-node implementation (MN^B) of Parallel Algorithm B. The purpose of this comparison is to analyze the speedup for GPU implementations and multiple-node implementation against sequential algorithms. Figures 18 and 19 depict the results for all conducted experiments for sequential algorithms, GPU implementations, and multiple-node implementations.

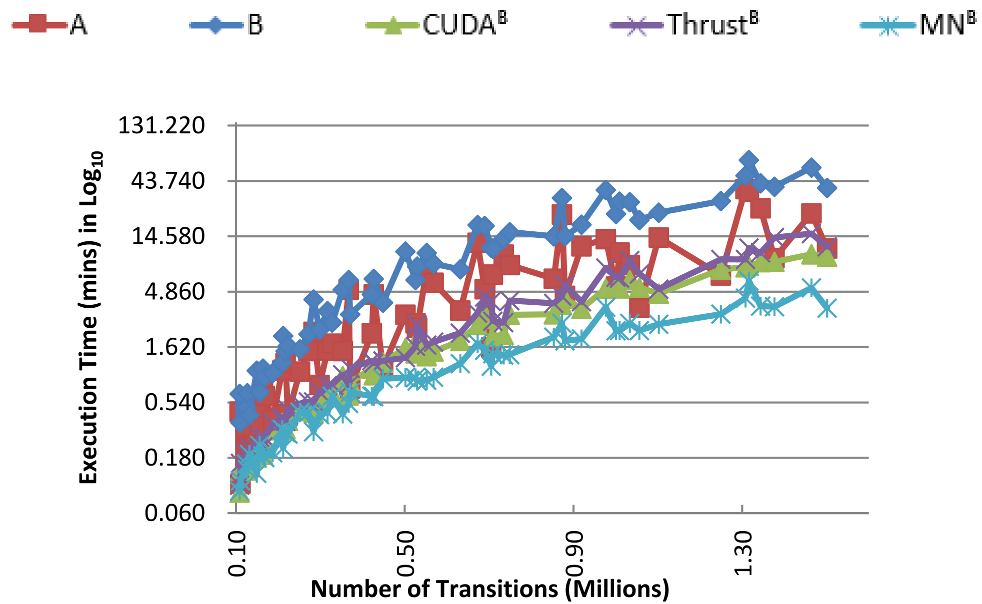

Figure 18. Sequential Algorithms versus Other Parallel Implementations for Small and Medium FSMs

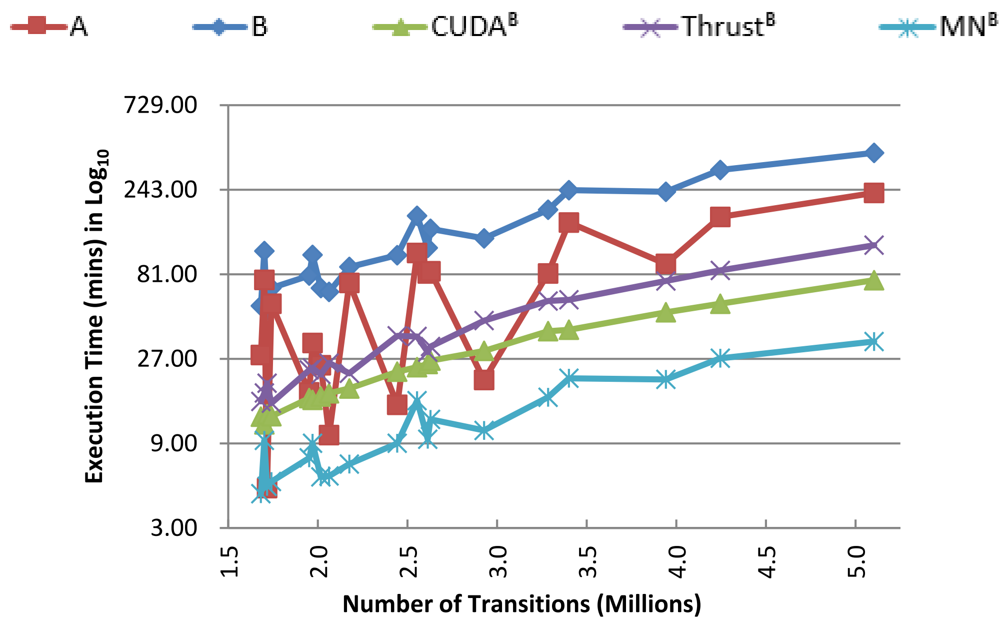

Figure 19. Sequential Algorithms versus Other Parallel Implementations for Big FSMs

According to the results depicted in Figures 18 and 19, the execution time for all algorithms and implementations increases exponentially with the number of transitions. The results also show that in the beginning when the number of transitions is less (i.e. for small FSMs), the difference in execution time is not

significant. However, when the number of transition increases (i.e. for medium and big FSMs), the difference in execution time increases for all CUDA^B, Thrust^B, and MN^B. MN^B gives the best performance throughout the experiments amongst the considered parallel implementations. The second best is CUDA^B and the third best is Thrust^B. There are distinct cases where (sequential) Algorithm A performs better than Thrust^B and CUDA^B; in such cases again the length of the distinguishing sequence is one and the (sequential) Algorithm A does not iterate through all the subsets and transitions, giving better performance than other implementations.

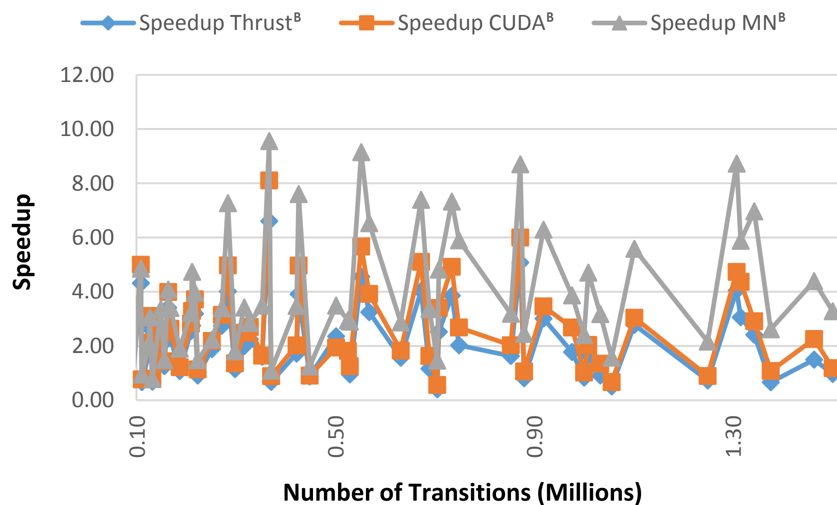

Figure 20. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm A for Small and Medium FSMs

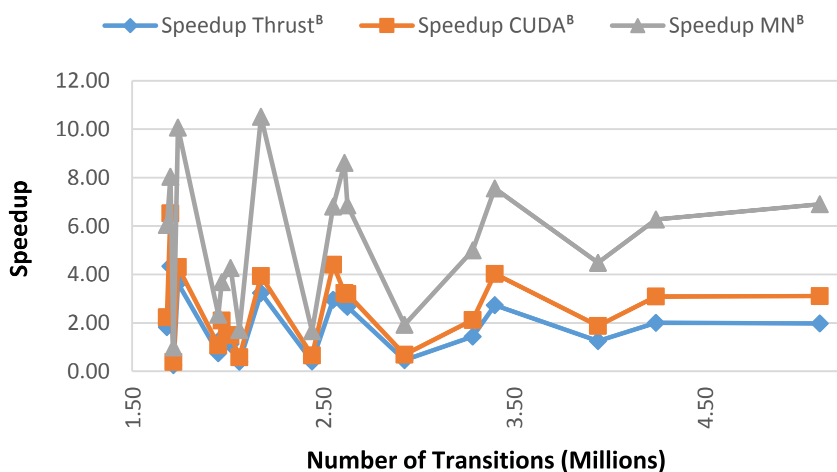

Figure 21. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm A for Big FSMs

Figures 20 and 21 illustrate the speedup w.r.t. to (sequential) Algorithm A. As observed, there is a constant variation in the speedup obtained for all small to big FSMs. Although there are cases in which MN^B scales up to 10x times, $CUDA^B$ scales up to 8x times, and $Thrust^B$ scales up to 6x times faster than the (sequential) Algorithm A. There are also some distinct cases (i.e., in which the length of the distinguishing sequence is one), but these implementations fail to scale above one, and in such cases parallel implementations are more costly in terms of execution time rather than (sequential) Algorithm A. The trend in the performance measure also remains the same (i.e., MN^B gives better performance than $CUDA^B$ and $Thrust^B$, and $CUDA^B$ gives better performance than $Thrust^B$).

The cause for the constant variation in this speedup is mentioned later in Section 5.6, where we analyze speedup from a different perspective and provide the reasons for this variation.

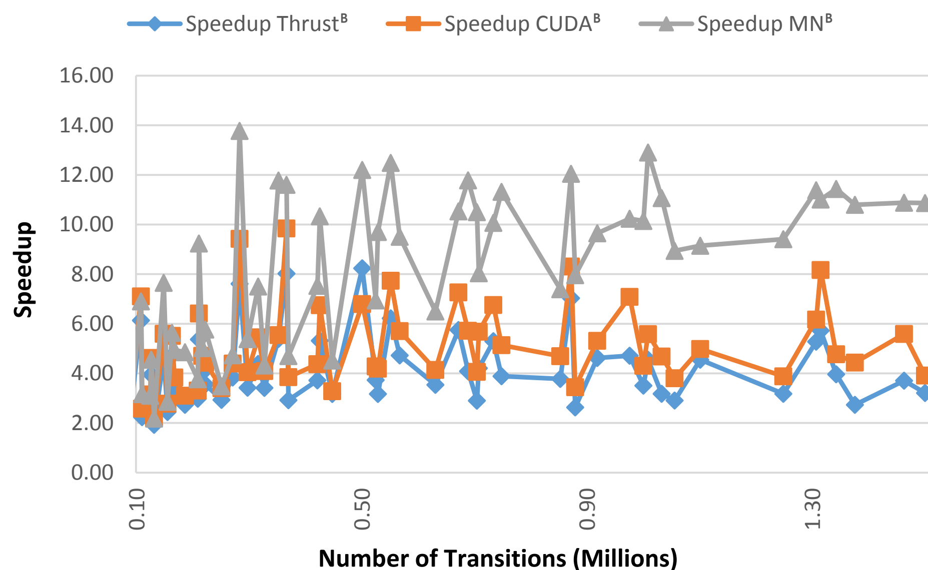

Figure 22. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm B for Small and Medium FSMs

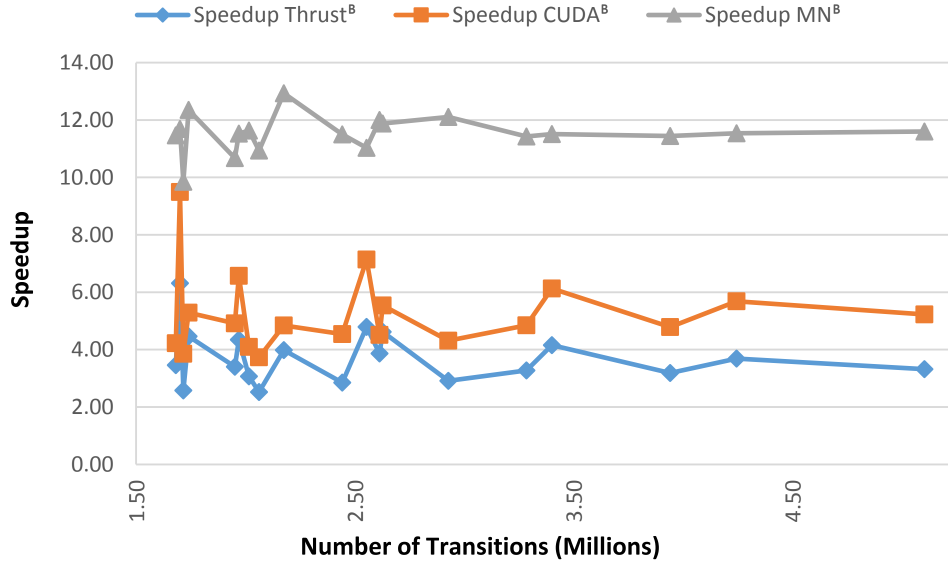

Figure 23. Speedup for Other Parallel Implementations w.r.t (Sequential) Algorithm B for Big FSMs

Figures 22 and 23 illustrate the speedup w.r.t. to (sequential) Algorithm B. For small to medium FSMs, the speedup varies frequently. However, in distinct cases, MN^B scales up to 14x times, $CUDA^B$ scales up to 10x times, and $Thrust^B$ scales up to 8x times faster than the (sequential) Algorithm B. For big FSMs, we observe a stable improvement in execution time. The speedup for MN^B is constantly above 11x times, and is significantly faster than $CUDA^B$ and $Thrust^B$. The speedup for $CUDA^B$ is nearly 5x times and gives better performance than $Thrust^B$. The speedup for $Thrust^B$ is around 3x times.

5.5 Execution Time versus Number of Transitions for Multi-Threaded Implementations Against Other Parallel Implementations

In this section, we compare the execution time for multi-threaded implementations against other parallel implementations which include $CUDA^B$, $Thrust^B$ and MN^B. We consider four threads for multi-threaded implementations. The purpose of this comparison is to determine the best parallel implementation amongst them. Figures 24 and 25 summarize the results for all the parallel implementations.

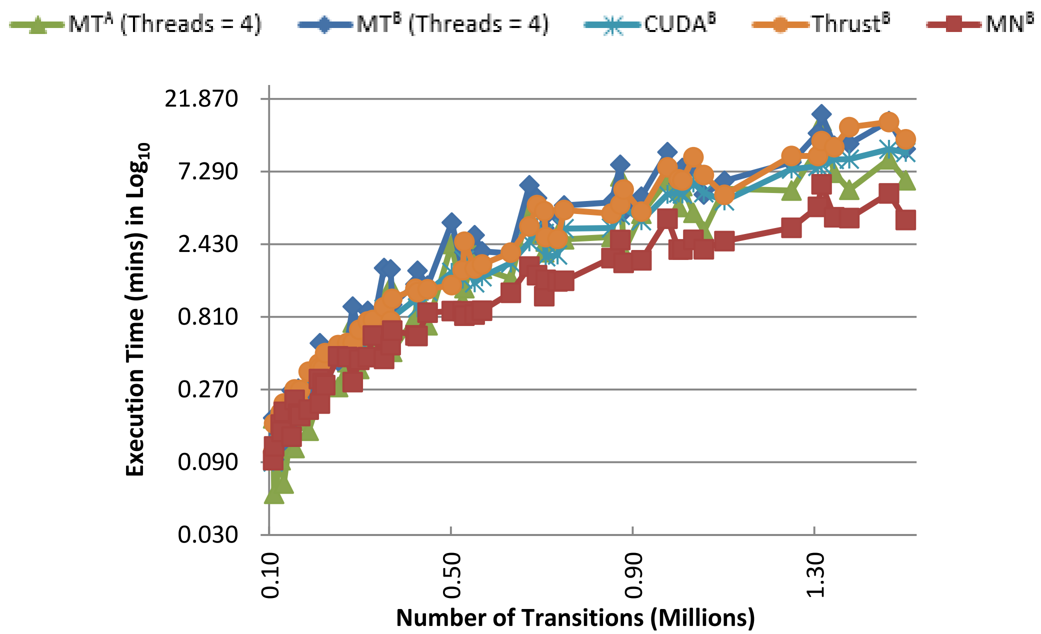

Figure 24. Multi-Threaded Implementations versus Other Parallel Implementations for Small and Medium FSMs

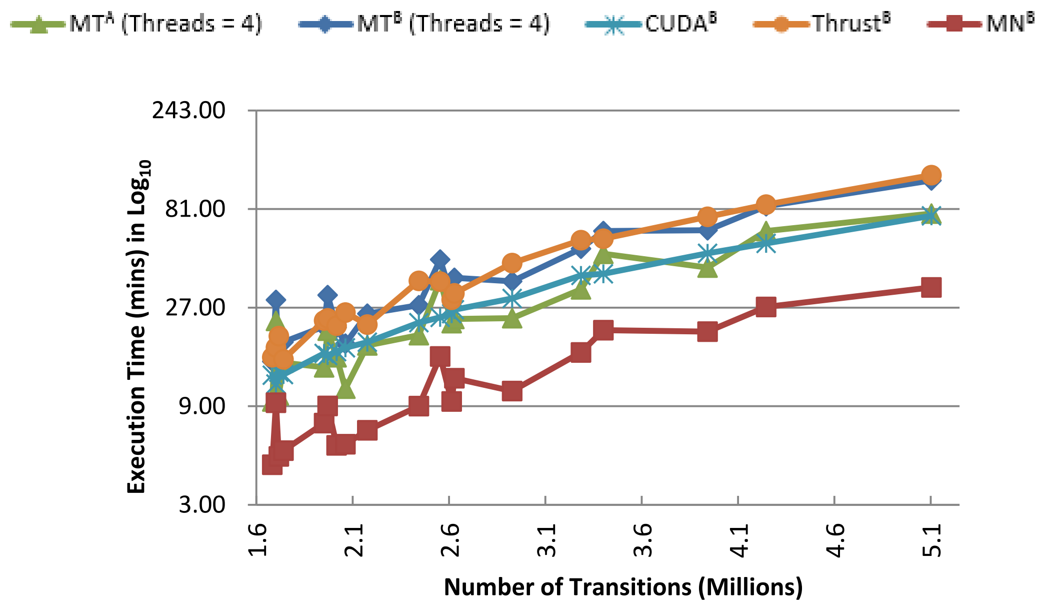

Figure 25. Multi-Threaded Implementations versus Other Parallel Implementations for Big FSMs

According to the results depicted in Figures 24 and 25, for all the implementations, execution time increases exponentially with an increasing number of transitions. For small FSMs, the execution time for all the parallel implementations is almost the same. For the medium FSMs, the execution times for

MT^A, MT^B, $Thrust^B$ and $CUDA^B$ are close to each other. However, MN^B performs slightly better than all the other implementations. For big FSMs, the performance of MN^B is significantly better than all the other implementations. MT^A and $CUDA^B$ have the same performance for execution time whereas MT^B and $Thrust^B$ have the same performance for execution time. Hence, we can conclude that MN^B has the best performance throughout (i.e. ignoring the performance of MN^B in the small FSMs). However, MT^B and $Thrust^B$ are costly in performance amongst the parallel implementations.

5.6 Achieved Speedup with Respect to Algorithm A

In this section, we study the relative performance improvement for all the parallel implementations through analyzing the speedup achieved in each experiment w.r.t. (sequential) Algorithm A. For this study, we categorize the considered FSMs according to the number of states. We have four main categories where the number of states are 100, 150, 200, and 250. These categories are further subdivided based on the number of inputs and outputs. Further, for each combination of number of states, number of inputs and number of outputs we consider four different ranges of non-determinism (i.e. R_1 = 50 %, R_2 = 60 %, R_3 = 70 %, R_4 = 80 %). Figures 26, 27, 28 and 29 depict the results of all conducted experiments.

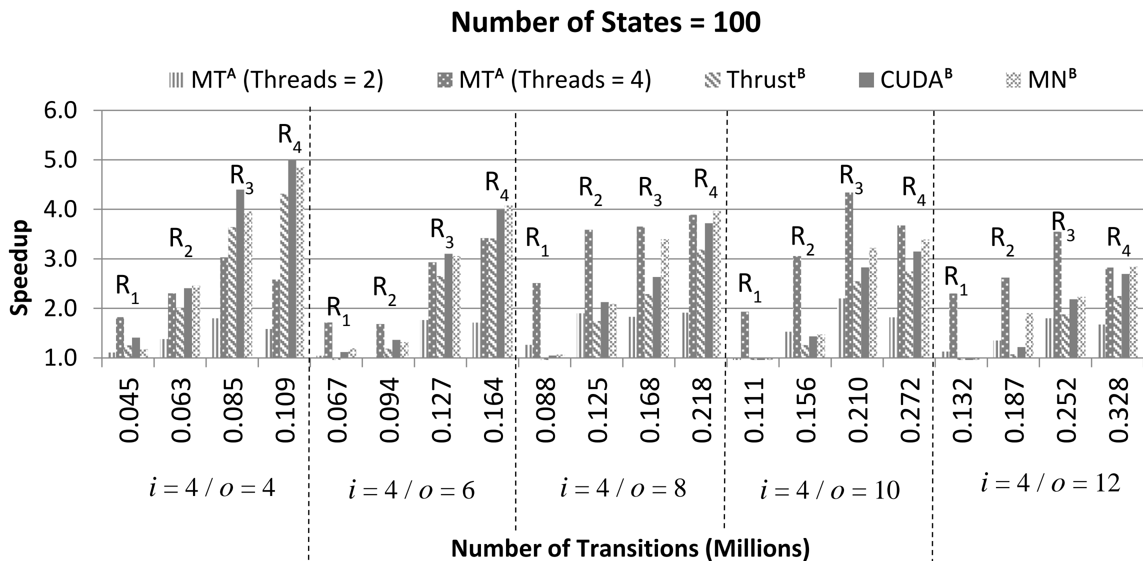

Figure 26. Achieved Speedup w.r.t. Algorithm A (States = 100)

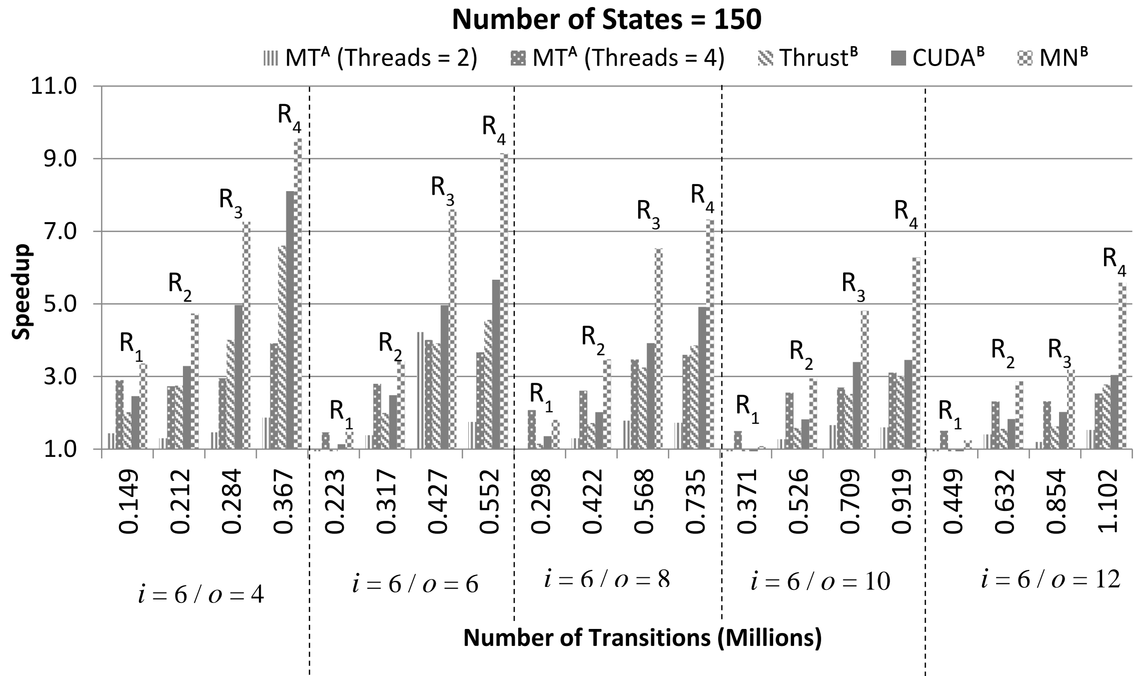

Figure 27. Achieved Speedup w.r.t. Algorithm A (states = 150)

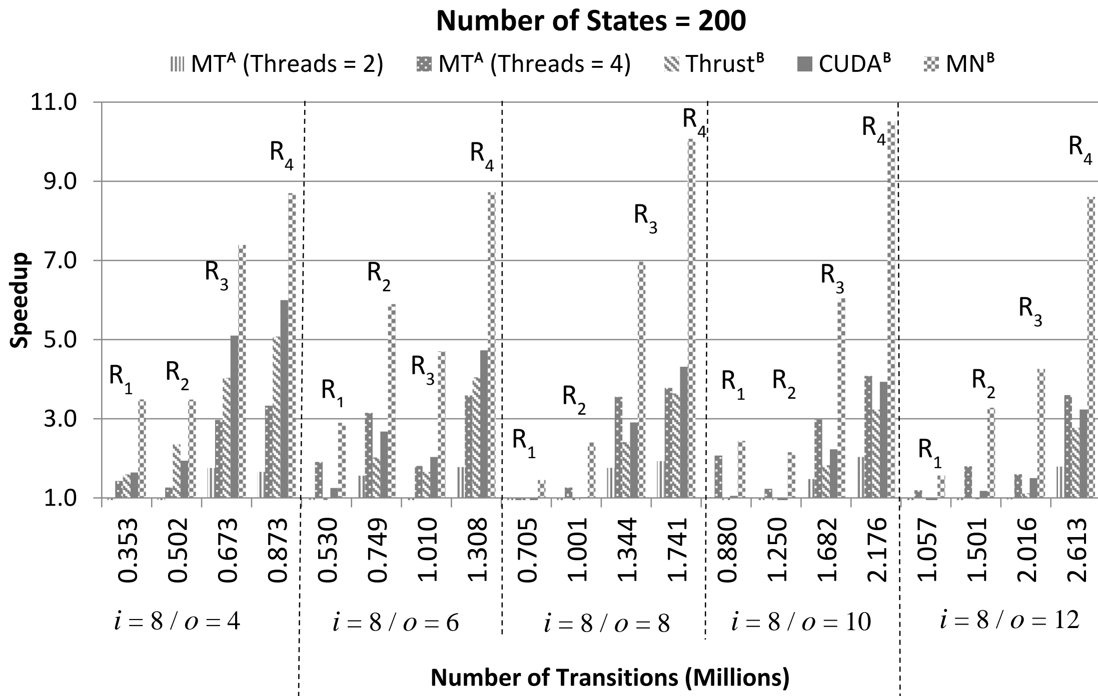

Figure 28. Achieved Speedup w.r.t. Algorithm A (states = 200)

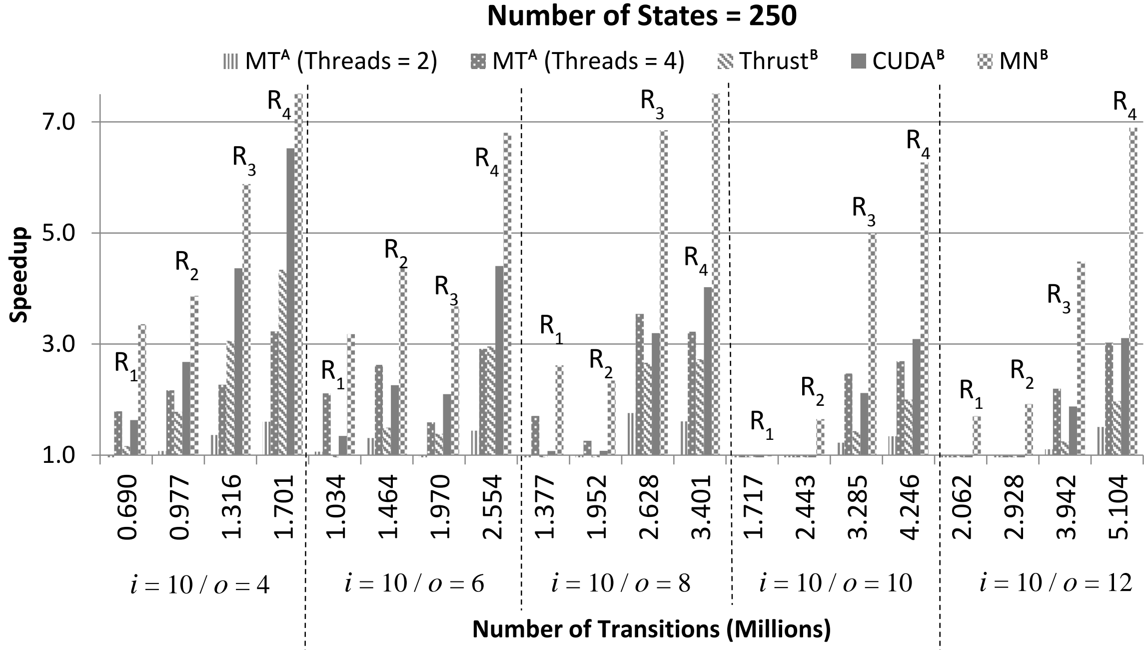

Figure 29. Achieved Speedup w.r.t. Algorithm A (states = 250)

According to the results depicted in Figures 26 to 29, we observe that when the number of states is 100, the speedup for Thust^{B}, CUDA^{B} and MN^{B} is less than the speedup for MT^{A} (Threads = 4), hence giving better performance. We also noted that the speedup for MT^{A} (Threads = 2) and MT^{A} (Threads = 4) varies throughout. This variation in speedup is due to the distinct cases where the length of the distinguishing sequence is one and there is a high probability that the sequential algorithm will find the solution without iterating through all the subsets and transitions, resulting in less execution time.

Similarly, there are distinct cases (i.e., in which the length of the distinguishing sequence is one). Most of these algorithms are not able to scale above speedup of one. In such cases, (sequential) Algorithm A performs better than the parallel implementations. However, when FSMs get bigger (i.e. number of states = 150, 200 and 250) we observe a trend in the variations for speedup in Thust^{B}, CUDA^{B} and MN^{B}. A summary of these variations is provided below:

a) As we increase the number of states, the speedup increases. This happens because GPUs are composed of thousands of computing cores which are able to perform operations in parallel. In order to utilize these cores efficiently, we need to put as much workload as possible on them, because the higher the ratio between computation and communication, the more efficient the execution is on a GPU. In the GPU implementations, we spawn as many threads as the number of subsets, hence when we increase the number of states, the number of subsets increases which puts more workload on the GPU, thus utilizing more cores and giving efficient execution. Hence, this results in increasing the speedup.
b) In general, increasing the range of non-determinism increases the number of transitions. Therefore, for a fixed number of states, number of inputs, and number of outputs, if we keep increasing the range of non-determinism the number of transitions increases, which puts more workload on GPUs, thus utilizing them more efficiently and resulting in increased speedup.
c) For a given number of states, if we increase the number of outputs while keeping the number of inputs unchanged, the speedup decreases. This happens because as we increase the number of outputs, the number of I/O-successors and number of transitions increases. Therefore, it takes more time to find the solution, and as a result speedup decreases.

5.7 Achieved Speedup with Respect to Algorithm B

In this section, we study the relative performance improvement for all the parallel implementations, through analyzing the speedup achieved in each experiment w.r.t. to the (sequential) Algorithm B. For this study, we categorize FSMs in the same order as mentioned in Section 5.6. Figures 30, 31, 32 and 33 depict the results of all conducted experiments.

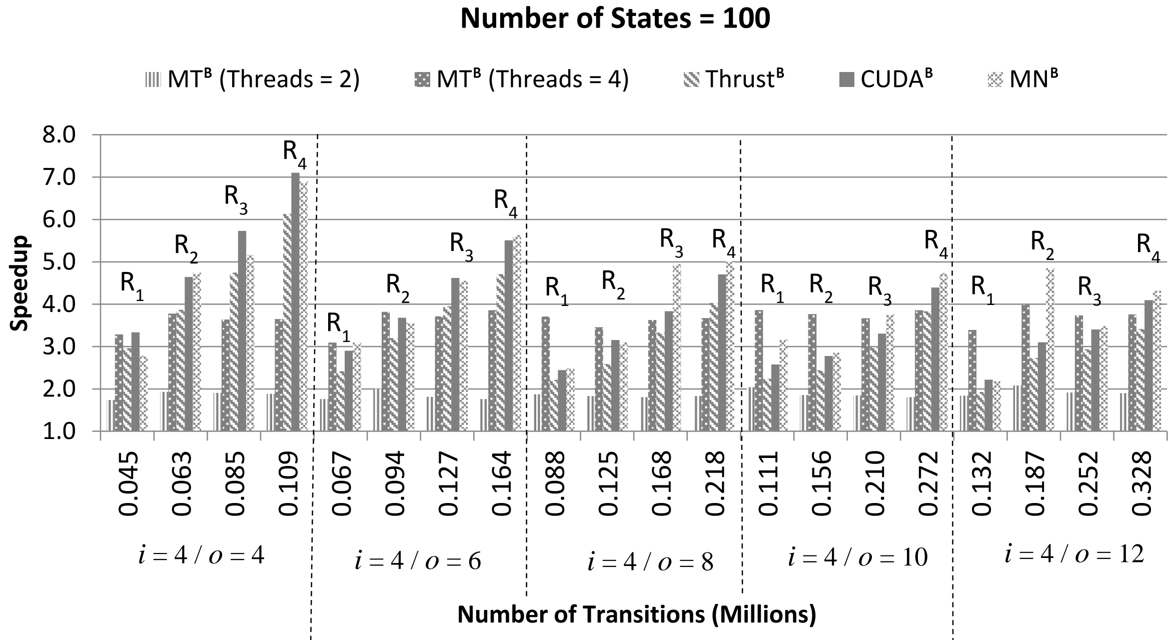

Figure 30. Achieved Speedup w.r.t. Algorithm B (States = 100)

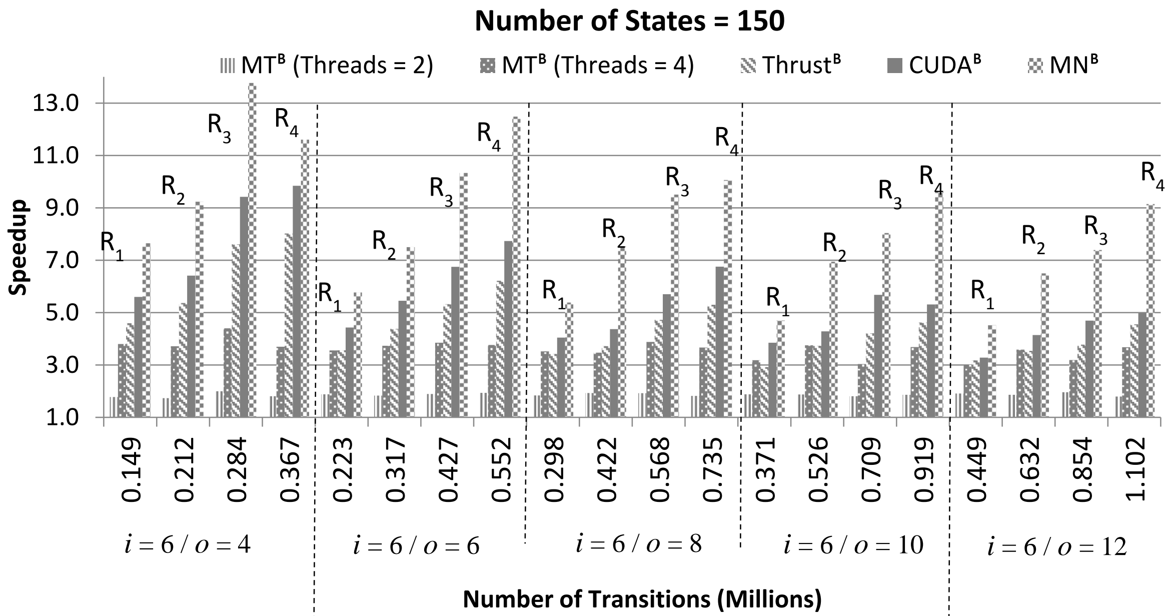

Figure 31. Achieved Speedup w.r.t. Algorithm B (states = 150)

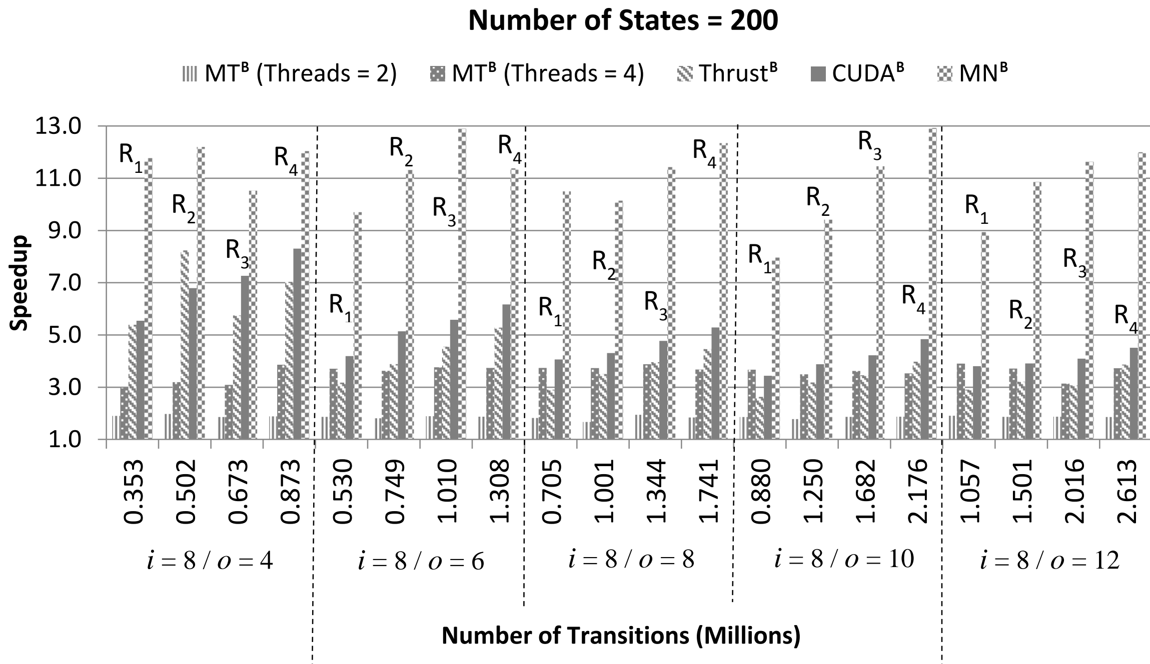

Figure 32. Achieved Speedup w.r.t. Algorithm B (states = 200)

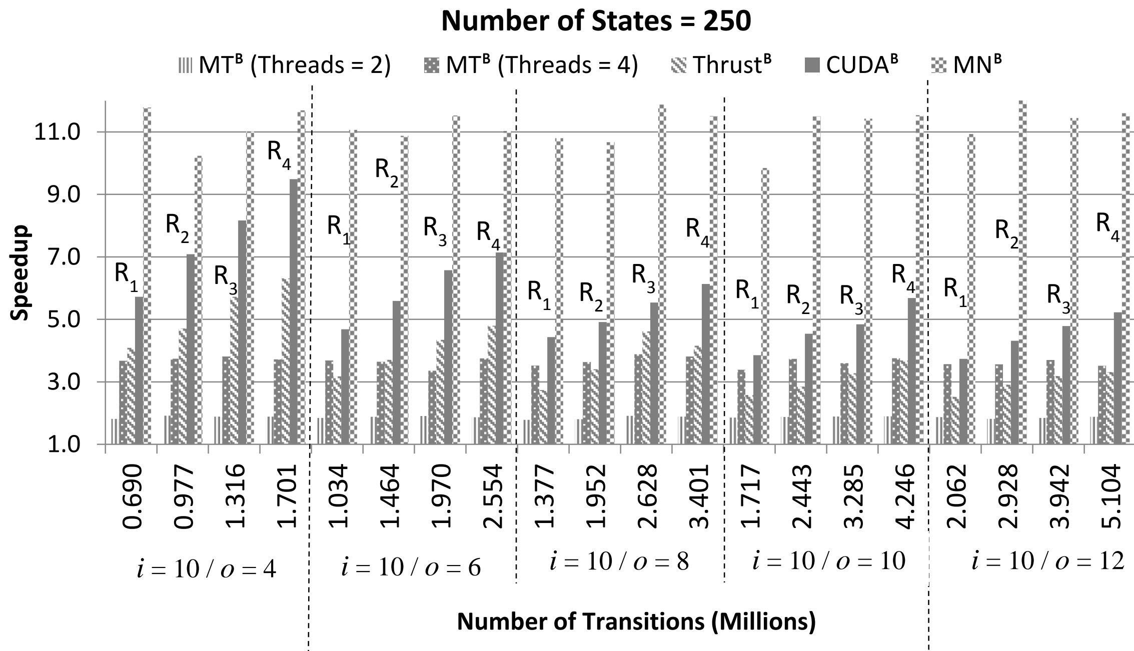

Figure 33. Achieved Speedup w.r.t. Algorithm B (states = 250)

According to the results depicted in Figures 30 to 33, we observe that the speedup for MT^B (Threads = 2) and MT^B (Threads = 4) remains constant throughout the conducted experiments (i.e., nearly up to 2 times and up to 4 times, respectively). When the number of states is 100, the speedup for MT^B (Threads = 4), $Thust^B$,

CUDA^B and MN^B is almost the same. However, for bigger FSMs (i.e., number of states = 150, 200 and 250) we observe a trend in the variations of speedup in Thust^B, CUDA^B and MN^B. The summary of these variations is the same as described in Section 5.6.

5.8 Achieved Speedup versus Number of Transitions with Respect to Algorithm A

In this section, we study the relative performance improvement by analyzing the speedup achieved over the increasing number of transitions for parallel implementations as compared to (sequential) Algorithm A. For this purpose, we divide FSMs in two categories: Size-I FSMs in which the number of transitions goes up to one million, and Size-II FSMs in which the number of transitions ranges from one million to five million. In this section, for studying the overall performance improvement, we did not consider each experiment individually; rather we took the average of the speedup at regular intervals for the number of transitions. Figure 34 depicts the results of speedup in Size-I FSMs, and Figure 35 depicts the results of speedup in Size-II FSMs.

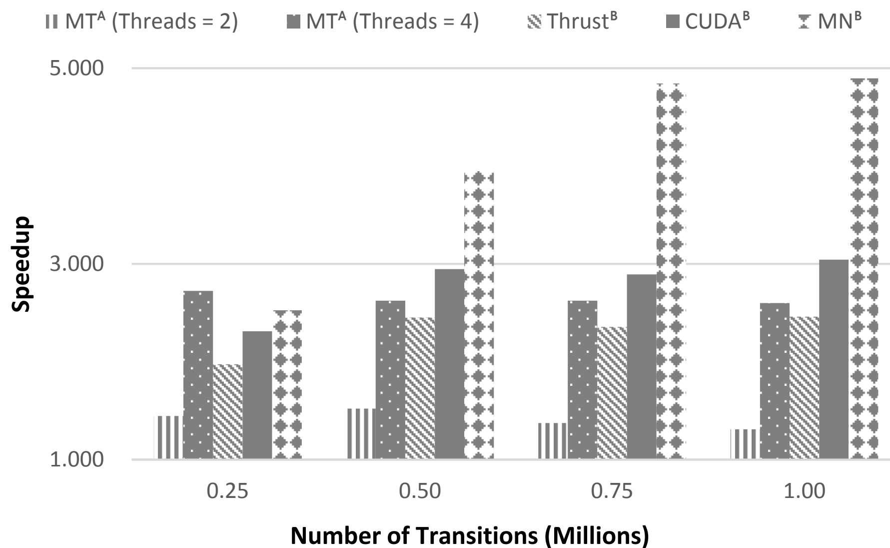

Figure 34. Speedup versus Number of Transitions w.r.t. Algorithm A for the Considered Size-I FSMs

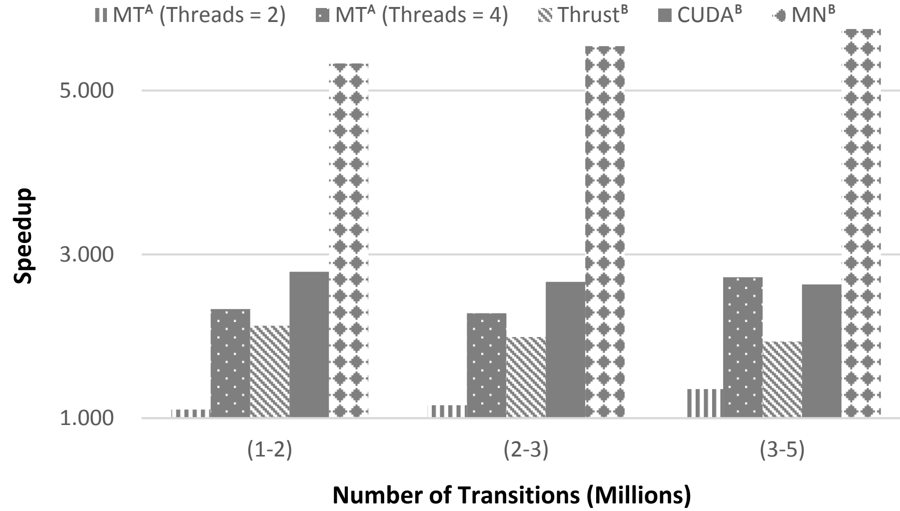

Figure 35. Speedup versus Number of Transitions w.r.t. Algorithm A for the Considered Size-II FSMs

According to the results depicted in Figures 34 and 35, speedup for MT^{A} (Threads = 2) and MT ^{A} (Threads = 4) in Size-I FSMs remains constant throughout the intervals (i.e., nearly 1.5 times and nearly 3 times, respectively). However, speedup for $Thust^{B}$, $CUDA^{B}$ and MN^{B} increases gradually, with intervals in Size-I FSMs.

However in Size-II FSMs we see slight variations. Speedup for MT ^{A} (Threads = 2) increases gradually with intervals, whereas the speedup for MT^{A} (Threads = 4), $Thust^{B}$, $CUDA^{B}$ and MN^{B} remains constant throughout the intervals in Size-II FSMs.

MN^{B} has significant gain in speedup throughout the intervals in both categories whereas the speedup for $CUDA^{B}$ and MT^{A} (Threads = 4) is close to each other in both the categories. However, the speedup for $Thust^{B}$ is the lowest amongst the parallel implementation, resulting in the worst performance amongst the parallel implementations.

5.9 Achieved Speedup versus Number of Transitions with Respect to Algorithm B

In this section, we study the relative performance improvement by analyzing the speedup achieved over the increasing number of transitions for parallel implementations as compared to (sequential) Algorithm B. For this purpose, we

divide FSMs in two categories as mentioned in Section 5.8. Figure 36 depicts the results of speedup in Size-I FSMs, and Figure 37 depicts the results of speedup in Size-II FSMs.

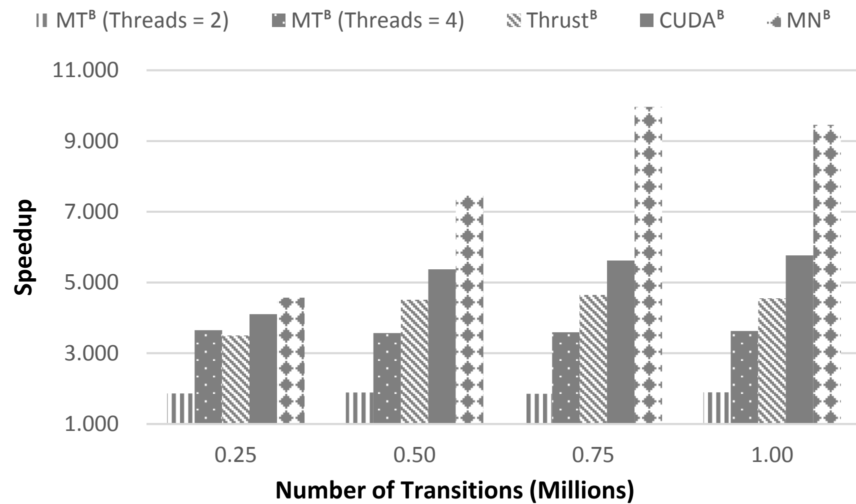

Figure 36. Speedup versus Number of Transitions w.r.t. Algorithm B for the Considered Size-I FSMs

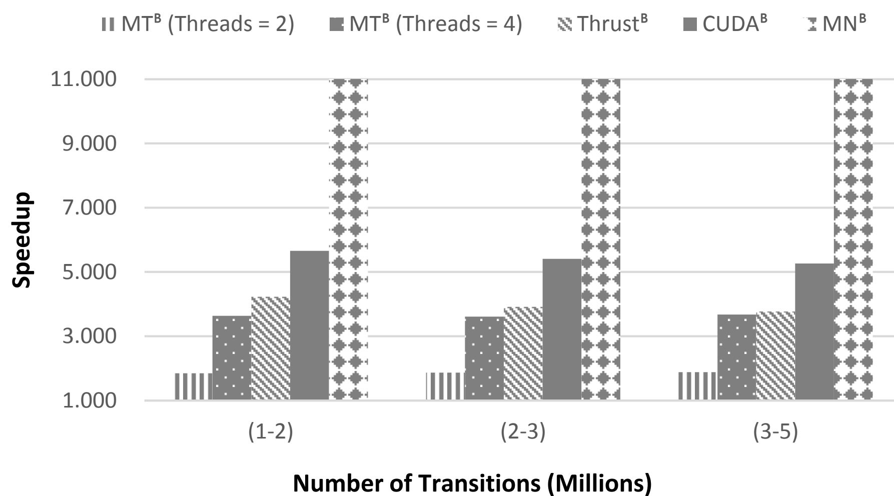

Figure 37. Speedup versus Number of Transitions w.r.t. Algorithm B for the Considered Size-II FSMs

According to the results depicted in Figures 36 and 37, speedup for MT^B (Threads = 2) and MT^B (Threads = 4) remains constant throughout (i.e., nearly 2 times and nearly 4 times, respectively). However, speedup for $Thust^B$, $CUDA^B$ and

MN^B increases, with intervals in the category of Size-I FSMs, whereas the speedup for $Thust^B$, $CUDA^B$ and MN^B remains constant throughout the intervals in Size-II FSMs.

MN^B has a significant gain in speedup throughout the intervals in both categories. Speedup for $CUDA^B$ is more significant than for $Thust^B$, and speedup for $Thust^B$ is more significant than for MT^B (Threads = 4).

5.10 Summary of All Obtained Results

Below we provide a summary of the results for all conducted experiments. We also provide summaries based on the three different categories of FSMs (small, medium and big). For all FSMs and for each category, we rank (from best to worst) the sequential/parallel algorithms according to **(1)** the execution time it takes to find the solution and **(2)** the speedup achieved with respect to the sequential algorithms. For the execution time, we rank (from best to worst) for lowest to highest execution time. For speedup we rank (from best to worst) for highest to lowest speedup.

a) Summary of execution time (minutes) for all the conducted experiments:

Table 6. Summary of Execution Time (Minutes) for All the Conducted Experiments

Rank w.r.t Execution Time	All Implementations	Average Execution Time (mins)
1	MN^B	5.53
2	$CUDA^B$	11.80
3	MT^A (Threads = 4)	12.12
4	$Thrust^B$	16.84
5	MT^B (Threads = 4)	17.19
6	Algorithm A	30.38
7	Algorithm B	62.27

b) Summary of execution time (minutes) for the considered categories of FSMs:

Table 7. Summary of Execution Time (Minutes) for Small FSMs

Rank w.r.t Execution Time	All Implementations	Average Execution Time (mins)
1	MN^{B}	0.72
2	CUDA^{B}	1.17
3	MT^{A} (Threads = 4)	1.23
4	Thrust^{B}	1.47
5	MT^{B} (Threads = 4)	1.80
6	Algorithm A	3.29
7	Algorithm B	6.45

Table 8. Summary of Execution Time (Minutes) for Medium FSMs

Rank w.r.t Execution Time	All Implementations	Average Execution Time (mins)
1	MN^{B}	3.44
2	MT^{A} (Threads = 4)	6.82
3	CUDA^{B}	7.30
4	Thrust^{B}	9.67
5	MT^{B} (Threads = 4)	9.97
6	Algorithm A	15.90
7	Algorithm B	37.00

Table 9. Summary of Execution Time (Minutes) for Big FSM

Rank w.r.t Execution Time	All Implementations	Average Execution Time (mins)
1	MN^{B}	12.42
2	CUDA^{B}	27.41
3	MT^{A} (Threads = 4)	27.82
4	MT^{B} (Threads = 4)	39.38
5	Thrust^{B}	39.79
6	Algorithm A	71.94
7	Algorithm B	143.36

c) Summary for speedup with respect to (sequential) Algorithm A for all the conducted experiments:

Table 10. Speedup w.r.t (Sequential) Algorithm A for All the Conducted Experiments

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	7.24
2	$CUDA^B$	3.87
3	MT^A (Threads = 4)	3.51
4	$Thrust^B$	2.12

d) Summary for speedup with respect to (sequential) Algorithm A for the considered categories of FSMs:

Table 11. Speedup w.r.t (Sequential) Algorithm A for Small FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	4.54
2	$CUDA^B$	2.81
3	MT^A (Threads = 4)	2.67
4	$Thrust^B$	2.23

Table 12. Speedup w.r.t (Sequential) Algorithm A for Medium FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	4.620
2	MT^A (Threads = 4)	2.330
3	$CUDA^B$	2.178
4	$Thrust^B$	1.645

Table 13. Speedup w.r.t (Sequential) Algorithm A for Big FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	5.792
2	$CUDA^B$	2.624
3	MT^A (Threads = 4)	2.586
4	$Thrust^B$	1.808

e) Summary for speedup with respect to (sequential) Algorithm B for all the conducted experiments:

Table 14. Speedup w.r.t (Sequential) Algorithm B for All the Conducted Experiments

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	10.40
2	$CUDA^B$	5.27
3	$Thrust^B$	3.95
4	MT^B (Threads = 4)	3.63

f) Summary for speedup with respect to (sequential) Algorithm B for the considered categories of FSMs:

Table 15. Speedup w.r.t (Sequential) Algorithm B for Small FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	8.91
2	$CUDA^B$	5.51
3	$Thrust^B$	4.38
4	MT^B (Threads = 4)	3.58

Table 16. Speedup w.r.t (Sequential) Algorithm B for Medium FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	10.75
2	$CUDA^B$	5.07
3	$Thrust^B$	3.83
4	MT^B (Threads = 4)	3.71

Table 17. Speedup w.r.t (Sequential) Algorithm B for Big FSMs

Rank w.r.t Speedup	Parallel Implementations	Average Speedup
1	MN^B	11.54
2	$CUDA^B$	5.23
3	MT^B (Threads = 4)	3.64
4	$Thrust^B$	3.60

Chapter 6: Conclusion

FSMs are widely used in various application domains, such as telecommunication, communication protocols and other reactive systems. In FSM-based testing, we apply experiments on a machine or a black-box Implementation Under Test (IUT) to deduce the required information. Experiments on FSMs consists of applying input sequences, observing corresponding output responses and drawing a conclusion about the machine under test. A distinguishing experiment determines the initial state of the FSM, and such experiments are widely used when checking the correspondence between transitions of an IUT and those of the specification FSM. In particular, Kushik *et al* in [34] proposed an algorithm for deriving the minimal length for an adaptive distinguishing experiment for any number of pairs of initial states for a complete observable nondeterministic FSM.

In this thesis, we studied adaptive distinguishing experiments for non-deterministic FSMs. To this end, we adapted the sequential algorithm proposed in [34] and developed two sequential algorithms (A and B) to derive the minimal length for an adaptive distinguishing experiment for a pair of initial states for a complete observable nondeterministic FSM. Algorithm A derives I/O-successors iteratively and checks for the solution (i.e., the length of the distinguishing sequence) in the corresponding iteration, while Algorithm B derives all the I/O-successors in advance and once the derivation is completed it proceeds to check the solution. We implemented both the sequential algorithms (A and B) and conducted comprehensive experiments on them. The sequential algorithms shows an exponential increase in the execution time as the number of transitions (i.e., size of the machine) increases. We also observed that in most cases, (sequential) Algorithm A gives the better performance as compared to (sequential) Algorithm B. Especially in cases where the length of the distinguishing sequence is one, we obtain significant gains in execution time for Algorithm A. To obtain the solution (i.e. the length of the distinguishing sequence) in a reasonable time, we implemented parallel versions of sequential algorithms (A and B) based on different hardware and software platforms. The parallel implementation of Algorithm B includes implementation on a multi-core CPU via multiple threads (MT^B), implementation on a GPU using tools like CUDA and Thrust ($CUDA^B$ and $Thrust^B$), and implementation on a NoW via multiple nodes

(MN^B). Algorithm A has a single parallel implementation on a multi-core CPU via multiple threads (MT^A). We conducted comprehensive experiments on all the parallel algorithms/implementations to assess their performance and to quantify the speedup that could be obtained using parallel algorithms/implementations versus sequential algorithms. We found that parallel implementations MN^B, $CUDA^B$, $Thrust^B$, MT^B, and MT^A scale exponentially with the number of transitions. The parallel implementation MN^B scales up to 14x times, $CUDA^B$ scales up to 10x times, and $Thrust^B$ scales up to 8x times w.r.t to Algorithm B. The parallel implementation MN^B scales up to 10x times, $CUDA^B$ scales up to 8x times, and $Thrust^B$ scales up to 6x times w.r.t to Algorithm A. For the multi-threaded implementation MT^B, we observed a stable speedup (i.e., nearly 2x speedup and 4x speedup for two and four threads, respectively). However, for the multi-threaded implementation MT^A we observed a wide variation in the speedup, and this was due to the varying length of the distinguishing sequence. As a result, the speedup obtained in MT^A was less favorable as compared to MT^B. Overall, MN^B gives the best performance amongst the parallel implementations, $CUDA^B$ and MT^A (Threads = 4) have the same performance, and $Thrust^B$ and MT^B (Threads = 4) have the same performance.

References

[1] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, volume 84, issue 8, pp. 1090-1123, 1996.

[2] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," *Proceedings of the ACM SIGSOFT International Symposium on Software testing and analysis*, pp. 109-124, 1994.

[3] 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*, volume 52, issue 12, pp. 1286-1297, 2010.

[4] A. Gill, "State-identification experiments in finite automata," *Information and Control,* volume 4, issues 2-3, pp. 132-154, 1961.

[5] Z. Kohavi, *Switching and finite automata theory*. New York: McGraw-Hill, 1978.

[6] D. Lee and M. Yannakakis, "Testing finite-state machines: State identification and verification," *IEEE Transactions on Computers*, volume 43, issue 3, pp. 306-320, 1994.

[7] A. Simao, A. Petrenko, and J. Maldonado, "Comparing finite state machine test coverage criteria," *IET Software*, volume 3, issue 2, pp. 91-105, 2009.

[8] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata studies*, Priceton Univeristy Press, volume 34, pp. 129-153, 1956.

[9] F. Hennine, "Fault detecting experiments for sequential circuits," *Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, Priceton, pp. 95-110, 1964.

[10] A. Petrenko, A. Simao, and N. Yevtushenko, "Generating checking sequences for nondeterministic finite state machines," *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 310-319, 2012.

[11] R. Alur, C. Courcoubetis, and M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines," *Proceedings of the Twenty-*

Seventh Annual ACM Symposium on Theory of Computing, pp. 363-372, 1995.

[12] R. M. Hierons, G.-V. Jourdan, H. Ural, and H. Yenigun, "Checking sequence construction using adaptive and preset distinguishing sequences," *Proceedings of the Seventh IEEE International Conference on Software Engineering and Formal Methods*, pp. 157-166, 2009.

[13] A. Petrenko and N. Yevtushenko, "Conformance tests as checking experiments for partial nondeterministic FSM," *Proceeding of the Formal Approaches to Software Testing*, pp. 118-133, 2005.

[14] M. Gromov, N. Evtushenko, and A. Kolomeets, "On the synthesis of adaptive tests for nondeterministic finite state machines," *Programming and Computer Software*, volume 34, issue 6, pp. 322-329, 2008.

[15] A. Petrenko and N. Yevtushenko, "Adaptive testing of deterministic implementations specified by nondeterministic FSMs," *Proceedings of the International Conference on Testing Software and Systems*, Lecture Notes in Computer Science 7019, ed: Springer, pp. 162-178, 2011.

[16] A. Mathur, *A Foundations of Software Testing.* Addison Wesley, 2008.

[17] G. Agibalov and A. Oranov, "Lectures on Automata Theory," *Tomsk State University Publishers*, 1984.

[18] S. Ginsburg, "On the length of the smallest uniform experiment which distinguishes the terminal states of a machine,"*Journal of the ACM (JACM)*, volume 5, issue 3, pp. 266-280, 1958.

[19] T. N. Hibbard, "Least upper bounds on minimal terminal state experiments for two classes of sequential machines,"*Journal of the ACM (JACM)*, volume 8, issue 4, pp. 601-612, 1961.

[20] S. Sandberg, "Homing and Synchronizing Sequences," *Model-Based Testing of Reactive Systems*, Lecture Notes in Computer Science 3472, ed: Springer, pp. 5-33, 2005.

[21] B. Ravikumar and X. Xiong, "Implementing sequential and parallel programs for the homing sequence problem," *Automata Implementation*, Lecture Notes in Computer Science 1388, ed: Springer, pp. 120-131, 1997.

[22] B. Ravikumar and X. Xiong, "Randomized parallel algorithms for the homing sequence problem," *Proceedings of the International Conference on Parallel Processing*, volume 3, pp. 82-89, 1996.

[23] N. Spitsyna, K. El-Fakih, and N. Yevtushenko, "Studying the separability relation between finite state machines," *Software Testing, Verification and Reliability*, volume 17, issue 4, pp. 227-241, 2007.

[24] N. Kushik, K. El-Fakih, and N. Yevtushenko, "Preset and adaptive homing experiments for nondeterministic finite state machines," *Proceedings of the Sixteenth International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science 6807, ed: Springer, pp. 215-224, 2011.

[25] N. Kushik, "Methods for deriving homing and distinguishing experiments for nondeterministic FSMs," Tomsk State University: PhD thesis, 2013.

[26] I. Hwang, N. Yevtushenko, and A. Cavalli, "Tight bound on the length of distinguishing sequences for non-observable nondeterministic Finite-State Machines with a polynomial number of inputs and outputs," *Information Processing Letters*, volume 112, issue 7, pp. 298-301, 2012.

[27] N. Kushik and N. Yevtushenko, "On the length of homing sequences for nondeterministic finite state machines," *Proceedings of the Eighteenth International Conference on Implementation and Application of Automata*, ed: Springer, pp. 220-231, 2013.

[28] P. H. Starke, *Abstract Automata*. University of Minnesota, USA: North-Holland Publishers and Company, 1972.

[29] F. Zhang and T. Cheung, "Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines," *IEEE Transactions on Software Engineering*, volume 29, issue 1, pp. 1-14, 2003.

[30] M. Gromov, K. El-Fakih, N. Shabaldina, and N. Yevtushenko, "Distinguing non-deterministic timed finite state machines," *Proceedings of the Formal Techniques for Distributed Systems*, ed: Springer, pp. 137-151, 2009.

[31] K. El-Fakih, M. Gromov, N. Shabaldina, and N. Yevtushenko, "Distinguishing experiments for timed nondeterministic finite state machines," *Acta. Cybern,* volume 21, pp. 205-222, 2013.

[32] C. Andres, N. Yevtushenko, A. Cavalli, "Modeling and testing the European train control system," Technical Report TechRca 14-03-2013, Telecom Sudparis, 2013.

[33] M. Leeke and A. Jhumka, "Evaluating the use of reference run models in fault injection analysis," *Proceedings of the Fifteenth IEEE Pacific Rim International Symposium on Dependable Computing*, *PRDC'09.*, pp. 121-124, 2009.

[34] N. Kushik, K. El-Fakih, and N. Yevtushenko, "Adaptive homing and distinguishing experiments for nondeterministic finite state machines," *Proceedings of the Twenty Fifth International Conference on Testing Software and Systems*, Lecture Notes in Computer Science 8254, ed: Springer, pp. 33-48, 2013.

[35] C. McClanahan, "History and Evolution of GPU Architecture," *A Survey Paper*, Georgia Institute of Technology College of Computing, Atlanta, Georgia, USA, 2010.

[36] M. Ali and T. Ozkul, "Review of Memory/Cache Management Technologies used on Heterogeneous Computing Systems," *International Journal of Computer and Information Technology*, volume 3, issue 3, 2014.

[37] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon*, et al.*, *Sourcebook of parallel computing* volume 3003: Morgan Kaufmann Publishers San Francisco, 2003.

[38] P. S. Pacheco, *Parallel programming with MPI*: Morgan Kaufmann, 1997.

[39] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, volume 1: MIT press, 1999.

[40] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*: MIT press, 1999.

[41] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An Introduction to the MPI standard," *Communications of the ACM*, 1995.

[42] G. Barlas, "An analytical approach to optimizing parallel image registration/retrieval," *IEEE Transactions on Parallel and Distributed Systems*, volume 21, issue 8, pp. 1074-1088, 2010.

[43] G. Barlas and B. Veeravalli, "Quantized load distribution for tree and bus-connected processors," *Parallel Computing*, volume 30, issue 7, pp. 841-865, 2004.

[44] B. Veeravalli, D. Ghose, V. Mani, and T. G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society Press, 1996.

[45] 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*, volume 9, issue 5, pp. 429-441, 1998.

[46] M. Drozdowski and P. Wolniewicz, "Out-of-core divisible load processing," *IEEE Transactions on Parallel and Distributed Systems*, volume 14, issue 10, pp. 1048-1056, 2003.

[47] J. T. Hung and T. G. Robertazzi, "Scheduling nonlinear computational loads," *IEEE Transactions on Aerospace and Electronic Systems*, volume 44, issue 3, pp. 1169-1182, 2008.

[48] A. Can, H. Shen, J. N. Turner, H. L. Tanenbaum, and B. Roysam, "Rapid automated tracing and feature extraction from retinal fundus images using direct exploratory algorithms," *IEEE Transactions on Information Technology in Biomedicine*, volume 3, issue 2, pp. 125-138, 1999.

[49] J. Jia, B. Veeravalli, and J. Weissman, "Scheduling multisource divisible loads on arbitrary networks," *IEEE Transactions on Parallel and Distributed Systems*, volume 21, issue 4, pp. 520-531, 2010.

[50] B. Veeravalli and G. Barlas, *Distributed multimedia retrieval strategies for large scale networked systems*, volume 29: ed: Springer, 2006.

[51] S. Viswanathan, B. Veeravalli, and T. G. Robertazzi, "Resource-aware distributed scheduling strategies for large-scale computational cluster/grid systems," *IEEE Transactions on Parallel and Distributed Systems*, volume 18, issue 10, pp. 1450-1461, 2007.

[52] N. Kushik, K. El-Fakih, N. Yevtushenko, and A. R. Cavalli, "On adaptive experiments for nondeterministic finite state machines," *International Journal on Software Tools for Technology Transfer*, pp. 1-14, 2014.

[53] N. Shabaldina, K. El-Fakih, and N. Yevtushenko, "Testing nondeterministic finite state machines with respect to the separability relation," *Testing of Software and Communicating Systems*, Lecture Notes in Computer Science, ed: Springer, pp. 305-318, 2007.

[54] Nvidia. (2014, September) GeForce GTX-980 Hardware Specifications [Online]. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications. (Accessed: December, 2014).

[55] Qt Company Ltd. (2014, December) Qt Developers Documentation [Online]. http://doc.qt.io. (Accessed: February, 2015).

[56] Wikipedia. (2014, May) Graphical Processing Units [Online]. http://en.wikipedia.org/wiki/Graphics_processing_unit. (Accessed: May, 2014).

[57] Nvidia. (March, 2014) GPU Accelerated Applications [Online]. http://www.nvidia.com/content/tesla/pdf/gpu-apps-catalog-mar14-digital-fnl-hr.pdf. (Accessed: February, 2015).

[58] A. Abdurazik, P. Ammann, Wei Ding, and J. Offutt, "Evaluation of three specification-based testing criteria," *Proceedings of Sixth IEEE International Conference Engineering* on *Complex Computer System.* pp. 179-187, 2000.

Appendix A

A.1 Nodes Considered in the NoW with Their Names and Respective Numbers

Table 18 shows, the considered nodes connected together in a NoW along with their respective numbers in the network.

Table 18. Considered Nodes (*N*) in the NoW

S.no	Machine Name	Node Number
1	Kingpenguin	0
2	DUNE2-CPU	1
3	DUNE2-GPU	2
4	DUNE2-GPU	3
5	DUNE-GPU	4
6	DUNE2-CPU	5

A.2 Estimating the Computational Speed for All Nodes

In order to obtain the computation speed (p) and constant overhead associated with the computation (e) for each node, we benchmarked the computation of the I/O successors table on each node individually.

The process was repeated for all the generated FSMs in order to obtain the execution time over a wide range of inputs. Figure 38 depicts the execution times against the number of transitions. In order to obtain the p and e parameters, we calculated the least-squares lines of the data shown in Figure 38. The p parameter values corresponding to the line slope are shown in Table 19.

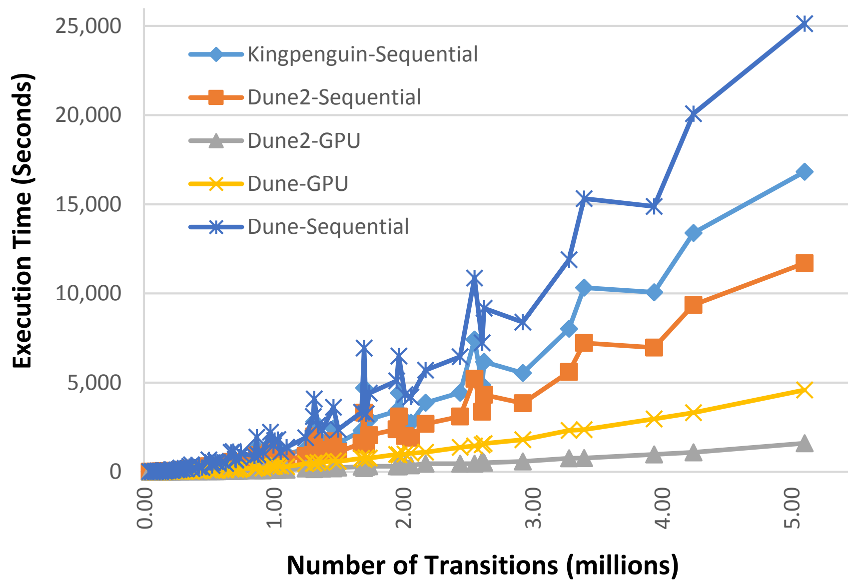

Figure 38. Execution Time versus the Number of FSM Transitions, for All the Considered Nodes

Table 19. Computation Speed (p) for All the Considered Nodes

Serial Number	Notations	Values
Kingpenguin	$p[0]$	0.00278 (sec/transition)
DUNE2-CPU	$p[1]$	0.00194 (sec/transition)
DUNE2-GPU	$p[2]$	0.00025 (sec/transition)
DUNE-GPU	$p[3]$	0.00075 (sec/transition)
DUNE-CPU	$p[4]$	0.00351 (sec/transition)

The constant overhead (e) corresponds to the intercept of the least-squares line. The e parameter values are shown in Table 20.

Table 20. Values Calculated for the *e* Parameter for All the Considered Nodes

Serial Number	Notations	Values
Kingpenguin	$e[0]$	-1057.06 (sec)
DUNE2-CPU	$e[1]$	-734.38 (sec)
DUNE2-GPU	$e[2]$	-89.52 (sec)
DUNE-GPU	$e[3]$	-282.87 (sec)
DUNE-CPU	$e[4]$	-1090.51 (sec)

A.3 Communication Speed Parameters

In order to measure the communication speed (l_c) and communication latency (b_c) during result collection between the NoW nodes, we used a "ping-pong" approach: two processes each on a different node, exchanging messages of known length in a "Send-Receive" or "Receive-Send" sequence.

The messages ranged between 100,000 Bytes and 17 MB (i.e. the maximum size of the I/O-successor tables used in testing), with an increment of 100,000 Bytes. The results are shown in Figure 39. The slope and intercept of the least-squares line correspond to the speed l_c and latency b_c. The actual values obtained are shown in Table 21.

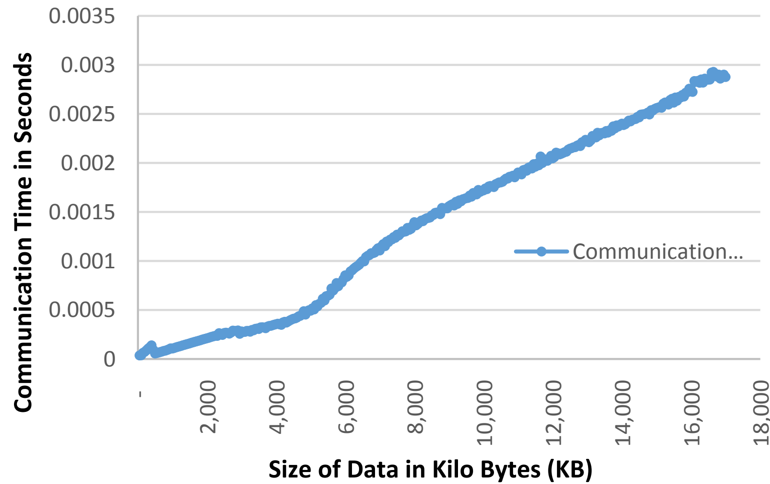

Figure 39. Communication Time versus Message Size, for Two Network Nodes

Table 21. Communication Parameters

Notations	Values
l_c	1.89E-10 (sec)
b_c	-0.000199 (sec/byte)

A.4 Remaining Parameters in the NoW Model

The remaining parameters in the DLT model were derived from the problem data:

1. Size of machine description in bytes (B): is the size of the given FSM in bytes for the experiment.
2. Number of transitions (T): is the total number of transitions present in the given FSM for the experiment.
3. Size of I/O successors table (O): is a size in bytes of the corresponding 2D array. The number of rows is equal to the number of different pairs (M) of a given FSM multiplied by the number of inputs. The number of columns is equal to the number of outputs plus 1 of the input FSM.

Vita

Mustafa Ali was born on July 20, 1988, in Karachi, Pakistan. He was educated in private schools in Pakistan and graduated from Hamdard College of Science and Commerce in 2006.

He studied at NED (Nadirshaw Eduljee Dinshaw) University of Engineering and Technology in Karachi, Pakistan, from which he graduated in 2010 and obtained a Bachelor's Degree in Computer Science & Information Technology.

Mr. Mustafa moved to the United Arab Emirates in 2012 where he joined the Master's program in Computer Engineering at the American University of Sharjah.