Parallel Algorithms for Generating Distinguishing Sequences for Observable Non-deterministic FSMs

Robert M. Hierons, Brunel University
Uraz Cengiz Türker, Brunel University

A distinguishing sequence (DS) for a finite state machine (FSM) is an input sequence that distinguishes every pair of states of the FSM. There are techniques that generate a test sequence with guaranteed fault detection power and it has been found that shorter test sequence can be produced if DSs are used. Despite these benefits, however, until recently the only published DS generation algorithms have been for deterministic FSMs. This paper develops a massively parallel algorithm, which can be used in GPU Computing, to generate DSs from partial observable non-deterministic FSMs. We also present the results of experiments using randomly generated FSMs and some benchmark FSMs. The results are promising and indicate that the proposed algorithm can derive DSs from partial observable non-deterministic FSMs with 32,000 states in an acceptable amount of time.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Verification

Additional Key Words and Phrases: Finite state machine, distinguishing sequences

ACM Reference Format:
Robert M. Hierons and Uraz C. Türker. Parallel Algorithms for Generating Distinguishing Sequences for Observable Non-deterministic FSMs *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2015), 37 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

It is widely recognised that software testing is an important part of the software development process but it is typically manual, expensive, and error prone. For example, reports suggest that testing can constitute around 40% of software development costs [Jones 1986]. This has led to significant interest in approaches that automate aspects of software testing. One of the most promising approaches to automation is *model based testing (MBT)* in which automation is based on a model [Barnett et al. 2003; Cartaxo et al. 2011; Farchi et al. 2002; Garousi et al. 2008; Grieskamp et al. 2011; Pickin et al. 2007; Tretmans 1996, 2008]. Evidence from an industrial project that involved hundreds of testers suggests that MBT can lead to significant benefits [Grieskamp et al. 2011].

Many systems are state-based: they have an internal state that affects the behaviour of operations and is also updated by operations. State-based languages are used to model such systems and much of the corresponding MBT work has considered Finite State Machines (FSMs) [Smetsers et al. 2015].

Author's addresses: Robert M. Hierons & Uraz C. Türker, Department of Computer Science, Brunel University London, Uxbridge, Middlesex, UK.

DOI: http://dx.doi.org/10.1145/0000000.0000000

An FSM is defined by a finite set of states and a finite set of transitions, where a *transition* can be represented by a tuple (s, x, y, s') that says that if the FSM receives input x while in state s then it can output y and move to state s'. FSM-based testing takes as input an FSM but the tester might have produced a model in some more expressive language, such as Specification and Description Language (SDL) [ITU-T 1999] or State-Charts [Harel and Politi 1998], whose semantics can be expressed using FSMs. A tool then maps the model to an FSM and uses this FSM as the basis for test automation. As a result, FSM-based testing has been used in several application domains such as sequential circuits [Friedman and Menon 1971], lexical analysis [Aho et al. 1986], software design [Chow 1978], communication protocols [Brinksma 1988; Dahbura et al. Aug; Lee et al. 1996; Low 1993; Sabnani and Dahbura 1988; Sidhu and Leung 1989], object-oriented systems [Binder 1999], and web services [Haydar et al. 2004; Utting et al. 2012].

In FSM-based testing, typically a test sequence (an input/output sequence) is constructed from the specification and the input portion of this test sequence is applied to the implementation under test (IUT). The *actual* output sequence produced by the IUT is compared to the output sequences allowed by the specification. If the IUT produces an output sequence allowed by the specification then the IUT is said to conform to its specification (for that test run). The literature contains many techniques for generating test sequences from an FSM [Chow 1978; da Silva Simão and Petrenko 2010; Gonenc 1970; Hierons and Ural 2006; Lee and Yannakakis 1996; Moore 1956; Petrenko et al. 2012; Petrenko and Simao 2015; Ural et al. 1997]. Most of these approaches use components that solve the following problems: *initialisation*, *state identification*, and *transition verification* and this paper focuses on state identification.

There are several approaches that solve the state identification problem, such as *Distinguishing Sequences (DS)*, *Unique Input Output (UIO) Sequences* and *Characterising Sets (W-Set)*. DSs distinguish every pair of states of the specification FSM and so, when they exist, their use can lead to testing involving fewer test sequences [Hennie 1964; Lee and Yannakakis 1996]. There are two types of distinguishing sequences. A *Preset Distinguishing Sequence (PDS)* is a single input sequence for which different states of the FSM produce different output sequences. On the other hand, an *Adaptive Distinguishing Sequence (ADS)* (also known as a *Distinguishing Set* [Boute 1974]) can be thought of as a finite rooted decision tree. There are many automated test generation techniques that either use a known DS [Hierons et al. 2009; Hierons and Ural 2006; Simão and Petrenko 2008; Ural et al. 1997], or use a DS and some alternatives together [Kapus-Kolar 2014; Simao and Petrenko 2009; Yalcin and Yenigün 2006] for state identification.

There has been significant interest in algorithms for deriving distinguishing sequences from deterministic FSMs and there are many computational complexity results regarding ADSs and PDSs. It has been reported that checking the existence of a PDS is PSPACE-complete [Lee and Yannakakis 1994]. Although earlier bounds for ADSs are exponential in the number of states [Gill 1962], Sokolovskii proved that if an FSM M with n states has an ADS, then it has an ADS with height at most $\pi^2 n^2/12$ [Sokolovskii 1971]. Moreover, Kogan claimed that, for an FSM with n states the length of an ADS is bounded above by $n(n-1)/2$ [Kogan 1973]. Later Rystsov proved this claim [Rystsov 1976]. However, all of these results are for deterministic FSM specifications that are complete: for every state/input pair there is a corresponding transition that specifies the resultant output and state reached. It has long been known that sometimes FSM specifications are not complete: they are partial (*partial FSMs*) [Drumea and Popescu 2004; Tsai et al. Aug; Xie et al. 2008; Zarrineh and Upadhyaya 1999]. Recently, it was shown that for partial FSMs, it is possible to check the existence of an ADS in polynomial time and checking the existence of a PDS is

PSPACE-complete [Hierons and Türker 2014]. Unfortunately, not all FSMs possess an adaptive distinguishing sequence. For such cases, Hierons and Türker introduced the notion of incomplete ADSs, which distinguish some, but not all, states of the specification FSM [Hierons and Türker 2015]. They also showed that certain corresponding approximation problems related to incomplete ADSs are PSPACE-complete. Note that Kushik et al. [Kushik et al. 2016] considered a similar notion within the context of distinguishing states of a non-initialised FSM; an FSM in which no single state is identified as being the initial state.

A number of authors have devised algorithms for producing ADSs. Lee and Yannakakis proposed an algorithm (LY algorithm) that constructs an ADS with height at most $n(n-1)/2$ in $O(\alpha n^2)$ time, where α is the number of inputs of the FSM [Lee and Yannakakis 1994]. Türker and Yenigün proposed two heuristics for the LY algorithm [Türker and Yenigün 2014]. A greedy algorithm to construct incomplete ADSs has also been given [Hierons and Türker 2015]. Recently, Türker et al. presented a set of ADS generation algorithms to construct compact ADSs [Türker et al. 2014, 2016].

The research discussed above concerned deterministic FSMs, with the main focus being on complete deterministic FSMs. However, many systems are non-deterministic and so there is a need to consider wider classes of FSMs. In this paper we consider an important class of non-deterministic FSMs called *observable non-deterministic* FSMs [Kohavi 1978; Starke 1972]. A non-deterministic FSM is observable if for each input/output pair x/y and state s there exists at most one transition from s with input x and output y. The main benefit of restricting attention to observable FSMs is that the current state can be determined from the observed input/output sequence and the identity of the initial state. Almost all work on testing from non-deterministic FSMs has concentrated on observable FSMs.

There has been some work on distinguishing states of an observable non-deterministic FSM, using either a fixed input sequence or an adaptive process that chooses the next input based on the sequence of inputs and outputs that has been observed[1]. Alur et al. showed that the length of a shortest adaptive process that distinguishes two states of an observable non-deterministic FSM with n states is at most $n(n-1)/2$ [Alur et al. 1995]; upper bounds become exponential if one instead uses an input sequence to distinguish two states [Spitsyna et al. 2007]. There is also an algorithm that constructs a weighted sequence that distinguishes two states of an FSM [Zhang and Cheung 2003]. Kushik et al. presented an algorithm for constructing ADSs for observable non-deterministic FSMs [Kushik et al. 2013]. However, the algorithm presented by Kushik et al. (the BF-ADS algorithm) was only able to derive ADSs from relatively small FSMs as it uses a powerset construction[2]. This will affect the scalability of this algorithm, although there may be scope to reduce the impact of this by using techniques developed by the verification community. In this paper 'scalability' refers to the ability of an algorithm to process larger data as required. So a scalable algorithm can process larger inputs (FSMs) than a less scalable algorithm.

All of these approaches assume that the FSM is complete. If an FSM is observable but not complete then it is a *partial observable non-deterministic* FSM. Deterministic FSMs and observable non-deterministic FSMs can be seen as being special classes of partial observable non-deterministic FSMs and so partial observable non-deterministic FSMs form a larger group. For simplicity, from now on we use the term FSM to denote partial observable non-deterministic FSMs. It appears that there is only one published algorithm for generating a PDS from such an FSM [Kushik et al. 2014] and one published algorithm for generating an ADS from such an FSM [Kushik et al.

[1]Such an adaptive process is similar to a decision tree.
[2]Given set A, the powerset of A is the set of all subsets of A.

2013]. This is slightly surprising since there is a move towards non-deterministic systems, as a result of concurrency (distributed systems and multi-threaded systems). It is also reported that many models are partial [Sabnani et al. 1989] and this is reflected by interest in testing from partial FSMs [Bonifácio and Moura 2014a,b,c; da Silva Simão and Petrenko 2008; Kushik et al. 2014; Luo et al. 1994a,b; Petrenko and Yevtushenko 2005, 2006; Petrenko et al. 1996]. These observations motivate the work reported in this paper, which investigates the problem of devising practical algorithms for deriving DSs from an FSM.

A major concerns of the work in this paper is the desire to develop algorithms that scale well as the number of states of the FSM increases. To address this issue, we devised massively parallel algorithms for *Graphics Processing Units* (*GPUs*), motivated by the successful use of GPUs in a range of application domains [Busato and Bombieri 2015; Djidjev et al. 2015; Dümmler and Egerland 2015; Farid et al. 2015; Lai et al. 2015; Liu and Huang 2015; Luo et al. 2010; Merrill et al. 2012; Mytkowicz et al. 2014]. There has also been some recent work that used GPU computing on FSM-based testing problems [Hierons and Türker 2016a,b; Türker 2015], although this was for deterministic FSMs. Moreover, there also exists a line of work in which the execution of an FSM is simulated by GPUs [Mytkowicz et al. 2014; Mytkowicz and Schulte 2012]. Finally, there is a line of work in which the massively parallel processing power of GPUs is used to accelerate linear algebraic operations, like sparse matrix vector multiplication, in order to accelerate probabilistic model checking algorithms [Bošnački et al. 2010, 2011; Wijs and Bošnački 2012].

This paper makes several contributions. It provides algorithms, for constructing PDSs and ADSs from FSMs, that are designed to be run on massively parallel GPUs. This involved a number of challenges since, for example, GPU cores are relatively simple and have an instruction set that is much smaller than that of a standard CPU. The paper also reports on the results of experiments that evaluated the proposed algorithms, with the initial experiments using randomly generated FSMs. In these experiments it was found that the proposed ADS generation algorithm was able to process inputs of up to 2048 times larger than the existing ADS construction algorithm and the proposed PDS generation algorithm was able to process inputs of up to 8 times larger than the existing PDS generation algorithm. Interestingly, when we applied the proposed ADS generation algorithm to the problem of generating an ADS from randomly generated *deterministic* complete FSMs, we found that it outperformed the existing ADS generation algorithm. This was an unexpected result, since the existing ADS generation algorithm, for deterministic complete FSMs, has low order polynomial time complexity. We also performed experiments using deterministic FSMs where the height of the shortest ADS is close to the theoretical upper bound and found that the existing polynomial-time ADS generation algorithm is faster for such FSMs.

This paper is organised as follows. Section 2 introduces the terminology and the notation that we use throughout the paper. We summarise the parallel ADS and parallel PDS construction algorithms in Section 3. We present the results of the empirical studies in Section 4 and Section 5 discusses threats to validity. Finally, in Section 6, we provide conclusions and discuss possible lines of future work. The low-level design is given in the Appendix.

2. PRELIMINARIES

2.1. Sequences

This paper concerns state-based systems and so observations will be sequences of inputs and outputs. We will use ε to denote the empty sequence and given sequences $\bar{x}$ and $\bar{x}'$ we will use $\bar{x}\bar{x}'$ to denote the concatenation of $\bar{x}$ and $\bar{x}'$. Given input/output

pairs $x_1/y_1, \ldots, x_k/y_k$ we will use $x_1/y_1 \ldots x_k/y_k$ and also $x_1x_2 \ldots x_k/y_1y_2 \ldots y_k$ to denote the corresponding input/output sequence (or *trace*) σ: the sequence in which x_1/y_1 is followed by x_2/y_2, then ... and finally x_k/y_k. Further, we will let $in(\sigma) = x_1 \ldots x_k$ and $out(\sigma) = y_1 \ldots y_k$ denote the *input portion* and *output portion* respectively of σ. We write pre to denote a function that takes a sequence or a set of sequences and returns the set of prefixes of these sequences. For example, $pre(x_1x_2) = \{\varepsilon, x_1, x_1x_2\}$ and $pre(x_1/y_1\ x_2/y_2) = \{\varepsilon, x_1/y_1, x_1/y_1\ x_2/y_2\}$. Finally we use $|.|$ to denote the cardinality of a set or the length of a sequence and so, for example, $|x_1/y_1\ x_2/y_2| = 2$ (since the sequence contains two input/output pairs) and $|\{x_1/y_1\ x_2/y_2\}| = 1$ (since the set contains one sequence).

2.2. Finite State Machines

Definition 2.1. An FSM is defined by a tuple $\boldsymbol{M} = (S, s_0, X, Y, h)$ where:

— $S = \{s_1, s_2, \ldots, s_n\}$ is a finite set of states.
— $s_0 \in S$ is the initial state.
— $X = \{x_1, x_2, \ldots, x_\alpha\}$ is the finite set of inputs.
— $Y = \{y_1, y_2, \ldots, y_\beta\}$ is the finite set of outputs. We assume that X is disjoint from Y.
— $h \subseteq S \times X \times Y \times S$ is the set of transitions.

Throughout this paper $M = (S, s_0, X, Y, h)$ will denote an FSM. At any given time, M is in one of its states. If an input $x \in X$ is applied when M is in state s and $(s, x, y, s') \in h$ then M can change its state to $s' \in S$ and produce output $y \in Y$. We say that $\tau = (s, x, y, s')$ is a *transition* of M with *source state* s, *destination state* s', and *label* x/y. The label x/y has *input portion* x and *output portion* y and we say that x is *defined* at state s.

If for all $s \in S$ and $x \in X$, there exists a transition $(s, x, y, s') \in h$ then M is said to be a *complete FSM*, otherwise it is a *partial FSM*. FSM M is said to be a *deterministic FSM* if for all $s \in S$ and $x \in X$ there exists at most one transition with source state s and input x; otherwise M is a *non-deterministic FSM*. A non-deterministic FSM M is *observable* if for all $s \in S$, $x \in X$, and $y \in Y$ there is at most one state $s' \in S$ such that $(s, x, y, s') \in h$. In this paper we consider partial observable non-deterministic FSMs and will use the term FSM to denote such machines.

An FSM can be represented by a directed graph. Figure 1 represents an FSM M_1 with state set $\{s_1, s_2, s_3, s_4\}$, inputs $\{x_1, x_2\}$, and outputs $\{y_1, y_2, y_3\}$. A node represents a state and a transition $\tau = (s, x, y, s')$ is represented by a directed edge with label x/y that goes from a node with label s to a node with label s'.

The behaviour of an FSM M is defined in terms of the labels of walks that leave the initial state of M. A *walk* of M is a sequence $\rho = (s_1, x_1, y_1, s_2)(s_2, x_2, y_2, s_3) \ldots (s_m, x_m, y_m, s_{m+1})$ of consecutive transitions. The walk ρ has *source state* s_1, *destination state* s_{m+1}, and *label* $x_1/y_1\ x_2/y_2 \ldots x_m/y_m$. Here $x_1/y_1\ x_2/y_2 \ldots x_m/y_m$ is a *trace* of M. We use $s \xrightarrow{\sigma} s'$ (or $s \xrightarrow{\bar{x}/\bar{y}} s'$) to denote there being a walk with source state s, destination state s', and label $\sigma = \bar{x}/\bar{y}$. Furthermore, we use $S' \xrightarrow{\bar{x}/\bar{y}} S''$ to denote S'' being the set of states that are destination states of walks whose source state is in S' and have label $\bar{x}/\bar{y}$. Thus, state s is in S'' if and only if M has a walk $\rho = (s_1, x_1, y_1, s_2)(s_2, x_2, y_2, s_3) \ldots (s_m, x_m, y_m, s_{m+1})$ such that $s_1 \in S'$, $s = s_{m+1}$, $\bar{x} = x_1 \ldots x_m$, and $\bar{y} = y_1 \ldots y_m$. If we consider the FSM M_1 from Figure 1, $s_4 \xrightarrow{x_2x_2x_1/y_2y_2y_1} s_2$ denotes there being a walk that has source state s_4, destination state s_2, and label $x_2/y_2\ x_2/y_2\ x_1/y_1$. This label/trace can also be written $x_2x_2x_1/y_2y_2y_1$ and has input portion $x_2x_2x_1$ and output portion $y_2y_2y_1$.

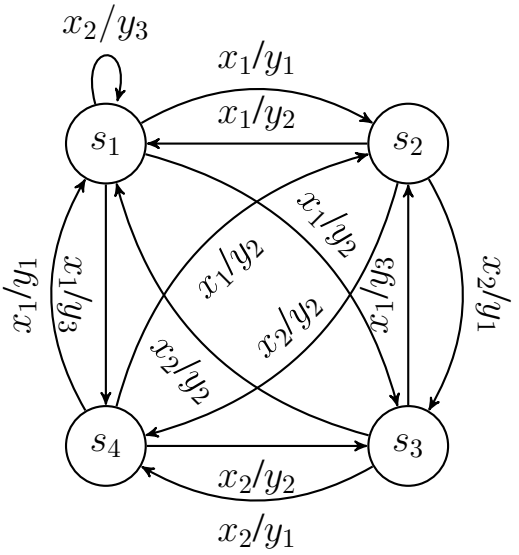


Fig. 1: FSM M_1 with four states, two inputs and three outputs.

An FSM M defines the language $L(M)$ of labels of walks with source state s_0 and we will use $L_M(s)$ to denote the language defined by making s the initial state of M. More formally, $L_M(s) = \{x_1 \dots x_m/y_1 \dots y_m \in X^*/Y^* | \exists s_1, \dots, s_{m+1}.s_1 = s \wedge \forall 1 \leq i \leq m.(s_i, x_i, y_i, s_{i+1}) \in h\}$. For example, $L_{M_1}(s_1)$ contains traces such as $x_2/y_3\ x_2/y_3$ and $x_1/y_1\ x_2/y_1\ x_2/y_2$. Clearly, $L(M) = L_M(s_0)$. Given $S' \subseteq S$ we let $L_M(S')$ denote the set of traces that can be produced if the initial state of M is in S' and so $L_M(S') = \cup_{s \in S'} L_M(s)$. Given state s and input sequence $\bar{x}$ we will use $M(s, \bar{x})$ to denote the set of traces in $L_M(s)$ that have input portion $\bar{x}$. We therefore have that $M(s, \bar{x}) = \{\sigma \in L_M(s) | in(\sigma) = \bar{x}\}$. States s, s' of M are *equivalent* if $L_M(s) = L_M(s')$ and FSMs M and N are *equivalent* if $L(M) = L(N)$. FSM M is *minimal* if there is no equivalent FSM that has fewer states. Note that since an (observable) FSM can be represented by a deterministic finite automaton, if we use the set of input/output pairs as the alphabet of the automaton, it is possible to minimise an FSM in polynomial time; one can just apply an algorithm for minimising a deterministic finite automaton. FSM M is *initially connected* if for every state s of M there is a walk that has source state s_0 and destination state s. Note that an initially connected FSM M is minimal if and only if $L_M(s) \neq L_M(s')$ for all $s, s' \in S$ with $s \neq s'$. As usual, in this work we consider only minimal FSMs.

Since FSMs can be partial, the result of applying an input sequence $\bar{x} = x_1 \dots x_k$ in state s may not be defined. This is the case if there is some $i < k$ such that $x_1 \dots x_i$ can take M from s to a state s' such that x_{i+1} is not defined in s'.

Definition 2.2. Input sequence $\bar{x}$ is said to be a *defined input sequence* for state s if

(1) $\bar{x} = \varepsilon$; or
(2) $\bar{x} = x\ \bar{x}'$ for an input x and input sequence $\bar{x}'$ such that x is defined in s and $\bar{x}'$ is defined in every state s' for which there exists an output y such that $(s, x, y, s') \in h$.

Similarly, input sequence $\bar{x}$ is *a defined input sequence* for set S' of states if $\bar{x}$ is a defined input sequence for all $s \in S'$.

2.3. Distinguishing states

In this paper we are interested in sequences that are guaranteed to distinguish states of M. If we start with an input sequence $\bar{x}$ from states s and s' and there is a common trace $\bar{x}/\bar{y} \in M(s,\bar{x}) \cap M(s',\bar{x})$ then $\bar{x}$ is not guaranteed to distinguish states s and s' (if we observe $\bar{y}$ in response to $\bar{x}$ then the state might have been s or s'). Further, if the walks from states s and s' with label $\bar{x}/\bar{y}$ have the same destination state then no extension to $\bar{x}$ can distinguish s and s'. We will want to identify such situations, when searching for input sequences to distinguish states, and so introduce the following terminology.

Definition 2.3. Input sequence $\bar{x}$ is said to be a *converging input sequence* for states s and s' if there exists $\bar{x}/\bar{y} \in M(s,\bar{x}) \cap M(s',\bar{x})$ and state s'' such that $s \xrightarrow{\bar{x}/\bar{y}} s''$ and $s' \xrightarrow{\bar{x}/\bar{y}} s''$. Otherwise $\bar{x}$ is non-converging. Further, $\bar{x}$ is a *converging input sequence* for state set S' if it is converging for two states $s, s' \in S'$ with $s \neq s'$ and otherwise $\bar{x}$ is non-converging for S'.

We now describe conditions under which an input sequence $\bar{x}$ distinguishes states of M; this requires that there is no common trace with input portion $\bar{x}$.

Definition 2.4. An input sequence $\bar{x}$ is a *separating sequence* for s, s' if $\bar{x}$ is a defined input sequence for s and s' and $M(s,\bar{x}) \cap M(s',\bar{x}) = \emptyset$.

This paper will introduce iterative algorithms for finding a DS. In each iteration the algorithms will search for input sequences that distinguish two or more states of $S' \subseteq S$ and also that have the potential to be extended to form a DS for S. The latter condition requires that $\bar{x}$ is non-converging.

Definition 2.5. An input sequence $\bar{x}$ is a *splitting sequence* for S' if $\bar{x}$ is a separating sequence for at least two states of S' and $\bar{x}$ is non-converging for S'.

We now define the two types of DSs.

Definition 2.6. An input sequence $\bar{x}$ is a *preset distinguishing sequence* (PDS) for M if $\bar{x}$ is a defined input sequence for S and for any pair of distinct states s, s' of set S, $\bar{x}$ is a separating sequence for s and s'.

Below we define the notion of an ADS. An ADS can be seen as being a decision tree that determines the next input to apply on the basis of the observations made. An ADS is applied as follows. The first step is to apply the input (say x), that labels the root of the ADS, to the FSM M and record the output y produced. The process then moves to the node of the ADS reached by the unique edge, from the root node, that has label y. If this node is labelled by an input then this input is applied and we repeat the above procedure. Otherwise, a leaf has been reached and this node is labelled by an ordered pair (s, s') of states. The experiment has then ended and it is known that the state of the FSM, before the experiment started, was s (and it is now s').

Definition 2.7. Let us suppose that M is an FSM with set of states S. An *adaptive distinguishing sequence (ADS)* for S is a rooted tree T. Each node N is associated with a set $\varsigma(N)$ of ordered pairs of states and a node that is not a leaf is also associated with an input $i(N)$. The root node is associated with the set $\{(s,s)|s \in S\}$ of pairs of states. Each edge has an output as label. The tree satisfies the following properties.

(1) The output labels of edges emanating from a node N are different.

(2) For every node N of T, if $\bar{x}$, $\bar{y}$ are the input and output sequences respectively formed by the node-edge labels on the path from the root node to N, then we have $(s, s') \in \varsigma(N)$ if and only if $s \xrightarrow{\sigma} s'$, where $\sigma = \bar{x}/\bar{y}$.
(3) Let us suppose that N is a node that is not a leaf and $x = i(N)$. For each output y in the set $\{y|\exists(s, s') \in \varsigma(N).x/y \in L_M(s')\}$ there is one edge from node N to a fresh node N' labelled with y such that $\varsigma(N') = \{(s, s'')|\exists(s, s') \in \varsigma(N).s' \xrightarrow{x/y} s''\}$.
(4) For every leaf of T, if $\bar{x}$, $\bar{y}$ are the input and output sequences respectively formed by the node-edge labels on the path from the root node to the leaf and if the leaf is labelled by a single pair (s, s') of states, then $\sigma \in L_M(s)$ and $\sigma \notin \cup_{s'' \in S \setminus \{s\}} L_M(s'')$ where $\sigma = \bar{x}/\bar{y}$.

In other words, if $\bar{x}$ is an input sequence, $\bar{y}$ is an output sequence and there is a walk with label $\bar{x}/\bar{y}$ from the root node to a leaf node whose label is the pair (s, s') then $\bar{x}/\bar{y} \in L_M(s)$ and there is no other state s'' in S that is the source of a walk with label $\bar{x}/\bar{y}$.

This paper explores the problem of automatically generating DSs from an FSM. The main focus will be on the development of parallel algorithms that can be run on Graphics Processing Units.

2.4. Graphics Processing Units

There has been significant recent interest in the use of *Graphics Processing Units* (GPUs) in general computing and GPUs have been used in many areas [Harish and Narayanan 2007; Luo et al. 2010; Merrill et al. 2012; Satish et al. 2009; Sengupta et al. 2007]. A GPU has a collection of *multiprocessors* (SMX) and each of these has a number of *processors*. Each multiprocessor has several types of storage: shared memory, which can be accessed by all of its processors; registers; texture memory (read only memory for the GPU); and constant (read only memory for the GPU, which has the lowest access latency) memory caches. A multiprocessor is referred to as a *single instruction multiple data* (SIMD) processor since each of its processors executes the same instruction (on different data) in a cycle. The individual multiprocessors communicate with one another through the *global device memory* and this memory is available to all of the processors in all of the multiprocessors.

When a GPU program executes, a number of threads are run in parallel, the programmer deciding on the number of threads to be launched. The threads that run simultaneously on a multiprocessor are called a *warp* and if the number of threads chosen by the programmer exceeds the warp size then the threads are time-shared. At a given time, a *block* of threads runs on a multiprocessor, with the maximum number of threads in a block being at most the warp size and varying between GPUs. Moreover, multiple blocks can be assigned to a single multiprocessor and their execution is again time-shared.

The threads executing on a multiprocessor share resources equally amongst themselves and each thread executes a piece of code called a *kernel*. During its execution, a thread t_i is given a unique ID and t_i can use its ID to access data residing in the GPU. Since the GPU memory is available to all threads, a thread can access any memory location. As a result of the global device memory, a GPU can be seen as being a Parallel Random Access Machine (PRAM) architecture. However, shared memory (which can only be accessed by threads within a block) can be accessed faster than global device memory and so performance is improved if shared memory is used. During GPU computation, the CPU can continue to operate and so the GPU programming model is a hybrid model, with a GPU being a co-processor (*device*) for the CPU (*host*).

3. PARALLEL ADS AND PDS GENERATION ALGORITHMS

This section describes the high-level ADS and PDS algorithms; implementation details are given in Appendix A. We start in Section 3.1 by discussing the design choices employed. In Section 3.2 we give an initial algorithm that is used in the PDS generation algorithm (described in Section 3.3) and the ADS generation algorithm (described in Section 3.4).

3.1. Design Choices

There are two main strategies that one might use in designing an algorithm for GPU computing. One approach, called the *fat thread strategy*, places a large amount of data in shared memory, and this minimises data access latency [Klingbeil et al. 2012]. However, a consequence of this is that there will be limits on shared memory and so the fat thread strategy may restrict the number of threads per block and so the degree of parallelism.

An alternative approach, called the *thin thread strategy*, is to use global device memory and so maximise the number of threads. When using this strategy, relatively little data is stored in shared memory and so there is the potential for performance to be adversely affected by the relatively high global memory latency [Klingbeil et al. 2012]. In this work we used the thin thread strategy, in order to maximise parallelism, and designed the algorithms to limit the effect of global memory latency.

3.2. Parallel state reduction algorithm

In this section we present an overview of a parallel algorithm, *the parallel states reduction algorithm* (PSR algorithm for short), that is used by the parallel ADS and parallel PDS algorithms. The aim of an ADS or PDS is to distinguish the states of the FSM M. The ADS and PDS generation algorithms both start with an initial 'state uncertainty' represented by the set S: the state of the FSM, before a PDS or ADS has been applied, could be any state in S. As inputs are applied and outputs are observed, the state uncertainty is refined to form a set of sets of states (one set of states for each input/output sequence that might have occurred). The aim of the PSR algorithm is to find an input sequence that reduces the size of a state uncertainty set. To achieve this, the PSR algorithm uses what we call an *output vector* (OV for short) and a *splitting vector* (SV for short).

Because the FSM M is observable, for a given set of states S' where $|S'| = \ell$, input sequence $\bar{x}$ and output sequence $\bar{y} \in Y^\star$, M has at most ℓ walks from states in S' that have label $\bar{x}/\bar{y}$. We will use an *output vector* to store information about what states can be reached, from states in S', using a walk with label $\bar{x}/\bar{y}$. An output vector (OV) $\mathcal{O}$ is a vector of pairs of states. We will be interested in a particular output vector.

Definition 3.1. Given state set S', defined input sequence $\bar{x}$, and output sequence $\bar{y}$, the resultant output vector $OV(S', \bar{x}, \bar{y})$ is a vector with $|S'|$ components such that for all $s \in S'$, we have a corresponding ordered pair (s, s') in $OV(S', \bar{x}, \bar{y})$ where s' is defined by:

(1) If there is no walk that has source state s and label $\bar{x}/\bar{y}$ then $s' = s^\star$, where $s^\star$ is a 'null' value.
(2) If there is a walk ρ that has source state s and label $\bar{x}/\bar{y}$ then s' is the destination state of ρ.

We will use delimiters $\langle$ and $\rangle$ for vectors and so, for example, $\langle (s_1, s_2), (s_2, s^\star), (s_4, s_1) \rangle$ is $OV(\{s_1, s_2, s_4\}, x_1, y_1)$ for M_1.

There may be more than one possible output sequence that can be produced if input sequence $\bar{x}$ is applied in states in S'; there can be at most $\beta^{|\bar{x}|}$ such output sequences

where β is the number of outputs of the FSM. We require an OV for each such output sequence and this leads to what we call a *splitting vector*, which is a vector of output vectors. Similar to output vectors, we will be interested in a particular splitting vector.

Definition 3.2. Given state set S' and defined input sequence $\bar{x}$ of length j, the resultant splitting vector $SV(S', \bar{x})$ contains β^j output vectors. Specifically, for all $\bar{y} \in Y^j$ we have that $SV(S', \bar{x})$ contains $OV(S', \bar{x}, \bar{y})$.

Given S', $\bar{x}$, and $\bar{y}$, the PSR algorithm will start with an initial SV in which all state pairs are of the form $(s, s^\star)$ (the output vectors are all of the form $\langle (s_1, s^\star), (s_2, s^\star), \ldots, (s_{|S'|}, s^\star)\rangle$). The PSR algorithm then builds $SV(S', \bar{x})$.

We now show how a splitting vector $SV(S', \bar{x})$ can be produced. Consider the FSM M_1 given in Figure 1, let $S' = \{s_1, s_3, s_4\}$, and let $\bar{x} = x_1x_1$. The corresponding SV starts with the plain SV that has $3^2 = 9$ output vectors. Thus, we start with $SV = \langle \mathcal{O}_{y_1y_1}, \mathcal{O}_{y_1y_2}, \mathcal{O}_{y_1y_3}, \ldots \mathcal{O}_{y_3y_1}, \mathcal{O}_{y_3y_2}, \mathcal{O}_{y_3y_3}\rangle$ where all of the output vectors are initially equal to $\langle (s_1, s^\star), (s_3, s^\star), (s_4, s^\star)\rangle$.

Now consider the output sequences of length 2. For output sequence y_1y_1, we have that there is only one walk whose source state is from S' and whose label has input portion x_1x_1 and output portion y_1y_1. This walk has source state s_4 and destination state s_2 and so $OV(S', x_1x_1, y_1y_1) = \langle (s_1, s^\star), (s_3, s^\star), (s_4, s_2)\rangle$. Now let us consider the output sequence y_3y_2. There are two walks that have source state in S', input portion x_1x_1 and output portion y_3y_2: one has source state s_1 and destination state s_2 and the other has source state s_3 and destination state s_1. As a result, we have that $OV(S', x_1x_1, y_3y_2) = \langle (s_1, s_2), (s_3, s_1), (s_4, s^\star)\rangle$. We similarly find that $OV(S', x_1x_1, y_1y_2) = \langle (s_1, s_1), (s_3, s^\star), (s_4, s_3)\rangle$. If we apply this to all output sequences of length 2 we obtain the splitting vector $SV(S', x_1x_1) = \langle OV(S', x_1x_1, y_1y_1), OV(S', x_1x_1, y_1y_2), OV(S', x_1x_1, y_1y_3), \ldots OV(S', x_1x_1, y_3y_1), OV(S', x_1x_1, y_3y_2), OV(S', x_1x_1, y_3y_3)\rangle$ in which:

$$
\begin{aligned}
OV(S', x_1x_1, y_1y_1) &= \langle (s_1, s^\star), (s_3, s^\star), (s_4, s_2)\rangle \\
OV(S', x_1x_1, y_1y_2) &= \langle (s_1, s_1), (s_3, s^\star), (s_4, s_3)\rangle \\
OV(S', x_1x_1, y_1y_3) &= \langle (s_1, s^\star), (s_3, s^\star), (s_4, s_4)\rangle \\
OV(S', x_1x_1, y_2y_1) &= \langle (s_1, s^\star), (s_3, s^\star), (s_4, s^\star)\rangle \\
OV(S', x_1x_1, y_2y_2) &= \langle (s_1, s^\star), (s_3, s^\star), (s_4, s_1)\rangle \\
OV(S', x_1x_1, y_2y_3) &= \langle (s_1, s_2), (s_3, s^\star), (s_4, s^\star)\rangle \\
OV(S', x_1x_1, y_3y_1) &= \langle (s_1, s_1), (s_3, s^\star), (s_4, s^\star)\rangle \\
OV(S', x_1x_1, y_3y_2) &= \langle (s_1, s_2), (s_3, s_1), (s_4, s^\star)\rangle \\
OV(S', x_1x_1, y_3y_3) &= \langle (s_1, s^\star), (s_3, s^\star), (s_4, s^\star)\rangle
\end{aligned}
$$

The PDS and ADS generation algorithms will start with the empty sequence and extend this. We will want to avoid extending an input sequence $\bar{x}$ if $\bar{x}$ cannot possibly be a prefix of a PDS or ADS for the set of states under consideration. We will therefore look to avoid using converging and undefined input sequences: if $\bar{x}$ is converging or undefined for S, then no PDS or ADS for M can have $\bar{x}$ as a prefix.

During the search for a PDS or ADS, an SV will contain OVs that represent possible state uncertainty; the set of possible current states of the FSM if a given input/output sequence has been observed. We aim to eliminate the state uncertainty (all of these sets are empty or singletons). As a result, we can see the reduction in the size of the largest such set (representing state uncertainty) as being progress towards returning a PDS or ADS. This motivates the following definition.

Definition 3.3. Input sequence $\bar{x}$ is a *reduction sequence* for state set S' if $\bar{x}$ is defined in S' and is non-converging for S' and also for every output sequence $\bar{y}$ with

$|\bar{y}| = |\bar{x}|$ we have the following

$$|\{s' \in S' | \exists s \in S.s' \xrightarrow{\bar{x}/\bar{y}} s\}| < |S'|$$

The following shows how we can determine whether an input sequence is a reduction sequence for S' by examining $SV(S', \bar{x})$ and is immediate from the definitions.

LEMMA 3.4. *Let S' be a set of states, $\bar{x}$ an input sequence that is defined for S', and $SV(S', \bar{x}) = \langle \mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_k \rangle$. Then $\bar{x}$ is a reduction sequence for S' if and only if the following properties hold.*

(*1*) *There is no $1 \leq i \leq k$ and states s, s', s'' with $s \neq s'$ such that $\mathcal{O}_i$ contains pairs (s, s'') and (s', s'').*
(*2*) *Every $\mathcal{O}_i$, $1 \leq i \leq k$, has a pair of the form $(s, s^\star)$.*

The first of these conditions ensures that $\bar{x}$ is non-converging for S' and the second ensures that for each possible output sequence $\bar{y}$, the corresponding set of state uncertainty ($\{s \in S' | \exists s' \in S.s \xrightarrow{\bar{x}/\bar{y}} s'\}$) is smaller than S'.

We now provide a brief overview of the PSR algorithm (see Algorithm 1). The PSR algorithm takes as input a state set S' and applies an iterative approach. In each iteration the algorithm applies the following steps.

(1) Generates an input sequence $\bar{x}$: starting from length $L = 1$ to an upper-bound $L = B$, the $Generate(L)$ function generates all input sequences of length L. Each time this function is called, it returns an input sequence not previously used in the current iteration of the loop.
(2) Generates the $SV(S', \bar{x})$ in parallel: it writes the source/destination states of each possible walk to corresponding OVs and checks if $\bar{x}$ is a defined input sequence for S'.
(3) Decides if $\bar{x}$ is a reduction sequence or not in parallel: the $Red(SV, \bar{x}')$ routine checks if $\bar{x}$ is a reduction sequence for S' by following a procedure as suggested by Lemma 3.4 and returns $True$ if $\bar{x}$ is a reduction sequence.
(4) If $Red(SV, \bar{x}')$ returns true then the algorithm returns $SV(S', \bar{x})$ and otherwise the algorithm continues.

ALGORITHM 1: Parallel state reduction algorithm.

Input: An FSM $M = (S, s_0, X, Y, h)$, state set $S' \subseteq S$, bound B
Output: Splitting vector for S'

```
begin
    IsRed = False
    L = 0
    while !IsRed ∧ L < B do
        L ← L + 1
        while !IsRed ∧ X^L not completely used do
            Initialise SV
            x̄ ← Generate(L)
            SV ← SV(S', x̄)
            if Red(SV, x̄) = True then
                IsRed = True
    if InRed then
        Return SV
    else
        Return {∅}
```

The PSR algorithm can generate all of the input sequences of length B or less, therefore it is guaranteed that the PSR algorithm will return an SV for S' if there exists a reducing sequence for S' of length at most B.

LEMMA 3.5. *Let $M = (S, X, Y, h)$ be an FSM, $S' \subseteq S$ be a set of states of M, and B a natural number. The PSR algorithm returns an SV for S' if and only if there exists a reducing sequence for S' of length at most B.*

In Appendix A we explain how we implemented the PSR algorithm.

3.3. The PDS generation algorithm

Recall that if two states of an FSM can be distinguished then they can be distinguished by an adaptive process[3] whose height[4] is at most $n(n-1)/2$. Thus, one might use a set of such adaptive process that, between them, distinguish all of the states of M. There is a polynomial upper bound on the size of such a set and so one might expect there to also be a polynomial upper bound on the size of any PDSs that are of interest[5]. This motivates an interest in a bounded PDS generation algorithm.

The PDS generation algorithm is very similar to the PSR algorithm. First note that an input sequence $\bar{x}$ is a PDS for a given set of states S' if every output vector in $SV(S', \bar{x})$ has at most one pair of the form (s, s') where $s' \neq s^\star$.

LEMMA 3.6. *Let $\bar{x}$ be a defined input sequence for S. If every output vector in $SV(S, \bar{x})$ has at least $|S| - 1$ pairs of the form $(s_i, s^\star)$, then $\bar{x}$ is a* PDS *for M.*

Consequently, in the PDS algorithm we include an additional step that checks whether the above property holds. The condition in the ninth line becomes the following in which $PDS(SV, \bar{x})$ is a call to a method that checks whether $\bar{x}$ is a PDS, using SV to check the conditions in Lemma 3.6.

9 If $PDS(SV, \bar{x}) = True$ **then**

For example, considering FSM M_1 where $S = \{s_1, s_2, s_3, s_4\}$ and $\bar{x} = x_2x_2x_2$. We set S' as S and $SV(S', \bar{x})$ contains the following output vectors.

$$
\begin{aligned}
OV(S', x_2x_2x_2, y_1y_1y_2) &= \langle (s_1, s^\star), (s_2, s_3), (s_3, s^\star), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_1y_2y_1) &= \langle (s_1, s^\star), (s_2, s^\star), (s_3, s_4), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_1y_2y_2) &= \langle (s_1, s^\star), (s_2, s^\star), (s_3, s_1), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_1y_2y_3) &= \langle (s_1, s^\star), (s_2, s_1), (s_3, s^\star), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_2y_2y_1) &= \langle (s_1, s^\star), (s_2, s_4), (s_3, s^\star), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_2y_2y_2) &= \langle (s_1, s^\star), (s_2, s_1), (s_3, s^\star), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_3y_3y_3) &= \langle (s_1, s_1), (s_2, s^\star), (s_3, s^\star), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_2y_3y_3) &= \langle (s_1, s^\star), (s_2, s^\star), (s_3, s_1), (s_4, s^\star) \rangle \\
OV(S', x_2x_2x_2, y_2y_1y_2) &= \langle (s_1, s^\star), (s_2, s^\star), (s_3, s^\star), (s_4, s_3) \rangle \\
OV(S', x_2x_2x_2, y_2y_2y_3) &= \langle (s_1, s^\star), (s_2, s^\star), (s_3, s^\star), (s_4, s_1) \rangle
\end{aligned}
$$

Note that each output vector has only one pair whose second element is a state from S (i.e. is not s^*) and $x_2x_2x_2$ is defined in all of the states in S'. Therefore, $\bar{x}$ is a PDS for S'.

[3]An adaptive process chooses the next input on the basis of the input/output sequence that has been observed.

[4]The height of an adaptive process is the length of the longest input sequence that can result from the application of this adaptive process.

[5]This is becuase there is a polynomial upper bound on the size of an alternative method, that of using a set of adaptive processes.

3.4. The ADS generation algorithm

The ADS generation algorithm iterates over a vector of output vectors called an *ADS vector*.

Definition 3.7. An *ADS vector* $\mathbb{O} = \langle \mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_{|\mathbb{O}|} \rangle$ is a vector of output vectors.

At each iteration, the algorithm receives an OV from $\mathbb{O}$. Given an OV $\mathcal{O}$, we let $C(\mathcal{O})$ and $I(\mathcal{O})$ denote the corresponding 'current states' and 'initial states' respectively. Thus $C(\mathcal{O}) = \{s \in S | \exists s' \in S.(s', s) \in \mathcal{O}\}$ and $I(\mathcal{O}) = \{s' \in S | \exists s \in S.(s', s) \in \mathcal{O}\}$, where $(s', s) \in \mathcal{O}$ denotes that $\mathcal{O}$ contains the pair (s', s). Then the algorithm attempts to find a reduction sequence for $C(\mathcal{O})$ using the PSR algorithm (using a bound L). If such a sequence $\bar{x}'$ is found, the algorithm generates the output vectors of the form $OV(C(\mathcal{O}), \bar{x}', \bar{y}')$ for every $\bar{y}'$ with $|\bar{y}'| = |\bar{x}'|$ and then it appends them to $\mathcal{O}$ and updates $\mathbb{O}$. The algorithm terminates when either (1) all unprocessed elements define singleton sets (an ADS has been found) or (2) an ADS cannot be found for the bound L. We present the ADS algorithm in Algorithm 2.

ALGORITHM 2: Parallel ADS algorithm.

Input: An FSM $M = (S, s_0, X, Y, h)$, $S' \subseteq S$ with $|S'| > 1$, and a bound L
Output: ADS for S'
begin
1 Initialise an OV $\mathcal{O}$ vector with S' and ε/ε
2 Initialise $\mathbb{O}$ with initial OV $\mathcal{O}$
3 **while** *There exists an unprocessed* OV *in* $\mathbb{O}$ *such that* $|C(OV)| > 1$ **do**
4 Retrieve an unprocessed OV vector $\mathcal{O}$ from $\mathbb{O}$ such that $|C(OV)| > 1$
5 Mark $\mathcal{O}$ as being processed and let $S' = C(\mathcal{O})$
6 $SV \leftarrow \text{PSR}(FSM, S', L)$
7 **if** $SV \neq \emptyset$ **then**
8 $Append(SV, \mathbb{O})$
else
9 Return "No ADS"
10 Return $\mathbb{O}$

We now demonstrate the execution of Algorithm 2 on the FSM M_1 given in Figure 1. After initialisation (Line 1) we have $\mathbb{O} = \langle OV(S, \varepsilon, \varepsilon) \rangle$ where

$$OV(S, \varepsilon, \varepsilon) = \langle (s_1, s_1), (s_2, s_2), (s_3, s_3), (s_4, s_4) \rangle$$

The algorithm then enters the while loop and chooses $OV(S, \varepsilon, \varepsilon)$. Let us assume that $\text{PSR}(M, S)$ returns $SV(S, x_2)$ in which we have

$$\begin{aligned} OV(S, x_2, y_1) &= \langle (s_1, s^\star), (s_2, s_3), (s_3, s_4), (s_4, s^\star) \rangle \dashrightarrow S'' = \{s_3, s_4\} \\ OV(S, x_2, y_2) &= \langle (s_1, s^\star), (s_2, s_4), (s_3, s_1), (s_4, s_3) \rangle \dashrightarrow S''' = \{s_4, s_1, s_3\} \\ OV(S, x_2, y_3) &= \langle (s_1, s_1), (s_2, s^\star), (s_3, s^\star), (s_4, s^\star) \rangle \dashrightarrow S^{iv} = \{s_1\} \end{aligned}$$

where $\dashrightarrow$ denotes current states extracted from corresponding OV. Upon receiving the SV from the PSR algorithm, the parallel ADS algorithm executes the append operation (Line 9) in which the above output vectors are added to the ADS vector $\mathbb{O}$. After this operation the ADS vector becomes

$$\mathbb{O} = \langle OV(S'', x_2, y_1), OV(S''', x_2, y_2), OV(S^{iv}, x_2, y_3) \rangle$$

As $\mathbb{O}$ contains elements that do not define singletons, the algorithm selects another OV and enters the second iteration of the while loop. Now assume that the algorithm selects $OV(S'', x_2, y_1)$. Let us assume that $\text{PSR}(M, S'')$ returns $SV(S'', x_1)$ which contains

$$OV(S'', x_1, y_1) = \langle (s_2, s^\star), (s_3, s_1) \rangle \dashrightarrow S^v = \{s_1\}$$
$$OV(S'', x_1, y_2) = \langle (s_2, s^\star), (s_3, s_2) \rangle \dashrightarrow S^{vi} = \{s_2\}$$
$$OV(S'', x_1, y_3) = \langle (s_2, s_2), (s_3, s^\star) \rangle \dashrightarrow S^{vii} = \{s_2\}$$

Therefore, after the append operation the ADS vector becomes

$$\mathbb{O} = \langle OV(S''', x_2, y_2), OV(S^{iv}, x_2, y_3), OV(S^v, x_2x_1, y_1y_1), OV(S^{vi}, x_2x_1, y_1y_2), OV(S^{vii}, x_2x_1, y_1y_3) \rangle$$

The algorithm might now select $OV(S''', x_2, y_2)$, which defines the set $S''' = \{s_3, s_4, s_1\}$ of states to process. Let us assume that during the third iteration, PSR(M, S''') returns $SV(S''', x_2)$ which is given as

$$OV(S''', x_2, y_1) = \langle (s_2, s^\star), (s_3, s^\star), (s_4, s_3) \rangle \dashrightarrow S^{viii} = \{s_3\}$$
$$OV(S''', x_2, y_2) = \langle (s_2, s_3), (s_3, s^\star), (s_4, s_1) \rangle \dashrightarrow S^{ix} = \{s_3, s_1\}$$
$$OV(S''', x_2, y_3) = \langle (s_2, s^\star), (s_3, s_1), (s_4, s^\star) \rangle \dashrightarrow S^x = \{s_1\}$$

Consequently, after the append the ADS vector becomes

$$\mathbb{O} = \langle OV(S^{iv}, x_2, y_3), OV(S^v, x_2x_1, y_1y_1), OV(S^{vi}, x_2x_1, y_1y_2),$$
$$OV(S^{vii}, x_2x_1, y_1y_3), OV(S^{viii}, x_2x_2, y_2y_1), OV(S^{ix}, x_2x_2, y_2y_2), OV(S^x, x_2x_2, y_2y_3) \rangle$$

Note that as the other output vectors define singleton sets, the algorithm will select $OV(S^{ix}, x_2x_2, y_2y_2)$ and the set of states to be processed becomes $S^{ix} = \{s_3, s_1\}$. Assume PSR(M, S^{ix}) returns $SV(S^{ix}, x_2)$, which contains the following OVs

$$OV(S^{ix}, x_2, y_1) = \langle (s_2, s_4), (s_4, s^\star) \rangle \dashrightarrow S^{xi} = \{s_4\}$$
$$OV(S^{ix}, x_2, y_2) = \langle (s_2, s_1), (s_4, s^\star) \rangle \dashrightarrow S^{xii} = \{s_1\}$$
$$OV(S^{ix}, x_2, y_3) = \langle (s_2, s^\star), (s_4, s_1) \rangle \dashrightarrow S^{xiii} = \{s_1\}$$

This is then followed by the append operation, which produces the following ADS vector.

$$\mathbb{O} = \langle OV(S^{iv}, x_2, y_3), OV(S^v, x_2x_1, y_1y_1), OV(S^{vi}, x_2x_1, y_1y_2),$$
$$OV(S^{vii}, x_2x_1, y_1y_3), OV(S^{viii}, x_2x_2, y_2y_1), OV(S^x, x_2x_2, y_2y_3)$$
$$OV(S^{xi}, x_2x_2x_2, y_2y_2y_1), OV(S^{xii}, x_2x_2x_2, y_2y_2y_2), OV(S^{xiii}, x_2x_2x_2, y_2y_2y_3) \rangle$$

As all elements of the ADS vector have current sets of states that are singletons, the algorithm terminates and returns $\mathbb{O}$. The ADS defined by $\mathbb{O}$ is given in Figure 2.

Similar to PDSs, our algorithm contains a bound. The parallel-ADS algorithm uses the PSR algorithm to generate a reduction sequence for an OV $\mathcal{O}$ of length at most L. Thus, if the value of L used is too small then the algorithm will fail to find ADSs. However, as we observed earlier, there is likely to be a bound on the size of an ADS that will be of interest and this bound might, for example, be based on the bound on the size of a set of adaptive processes that distinguish the states of M.

The algorithm generates what we called L-ADSs. The following, in which the out-degree of a node is the number of outgoing edges of that node, defines L-ADSs.

Definition 3.8. An ADS T is an *L-adaptive distinguishing sequence* (L-ADS) if for every path ρ in T of length L, from a node N to a node N', we have that N' has fewer current states than N.

The essential idea is that within an L-ADS 'progress' must be made within L steps. The following is clear.

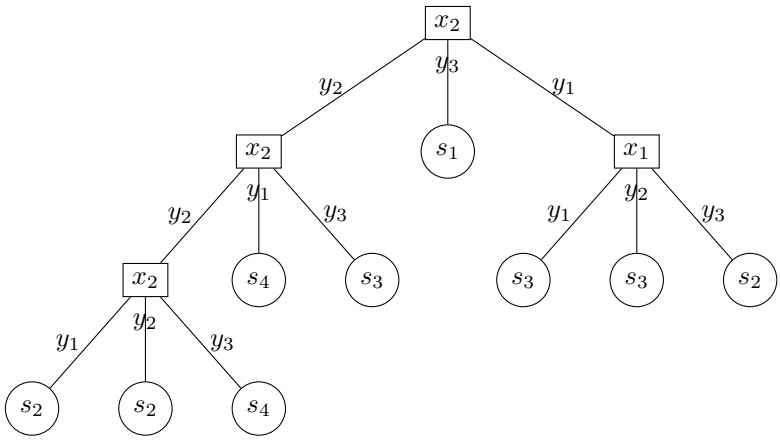


Fig. 2: An ADS for FSM M in Figure 1. We have not added the sets of states associated with nodes to reduce the visual complexity.

LEMMA 3.9. *If FSM M possesses an L-*ADS *and M and L are supplied to Algorithm 2, then Algorithm 2 returns a set of blocks $\mathbb{O}$ for M.*

From the definition of an ADS (or L-ADS), if Algorithm 2 returns an ADS vector then this defines an ADS (or L-ADS).

LEMMA 3.10. *If Algorithm 2 returns an* ADS *vector then the input sequences associated with this* ADS *vector define an* ADS *(or L-*ADS*) for M.*

4. EXPERIMENTS

In this section we present the results of our experiments. In these experiments we used an Intel Core I5 CPU (3550S) with 4GBs of RAM on a machine that had 64 bit Windows 7 Professional. The parallel algorithms were executed on an Nvidia TESLA K40 GPU card.

In the experiments, we evaluated four algorithms: the proposed parallel ADS (Parallel-ADS) algorithm, the proposed parallel PDS (Parallel-PDS) algorithm, the brute force ADS generation algorithm presented in [Kushik et al. 2013] (BF-ADS), and the brute force PDS (BF-PDS) generation algorithm [Kushik et al. 2014]. We now provide a brief overview of BF-ADS and BF-PDS algorithms.

4.1. Existing algorithms to construct ADSs/PDSs for FSMs

4.1.1. Constructing ADSs. Kushik et al. [Kushik et al. 2013] described an exponential algorithm (BF-ADS) to derive adaptive distinguishing sequences from FSMs. The BF-ADS algorithm has two phases. In the first phase the algorithm finds splitting sequences for subsets of S, and in the second phase the algorithm constructs an ADS by using these splitting sequences.

The algorithm first generates the power-set of S; we will denote this $\mathcal{P}$. All the singleton sets are then extracted from $\mathcal{P}$ to form $F = \{\{s_1\}, \{s_2\}, \ldots, \{s_n\}\}$. The algorithm then enters an iterative process in which it tries to add set $\{s_1, s_2, \ldots, s_n\}$ to F. At every iteration the algorithm considers all (x, S') for $x \in X$ and $S' \in \mathcal{P} \setminus F$ and determines whether it is the case that x is defined in S', x is non-converging in S', and for all $y \in Y$ we have that either y cannot be produced from a state in S' with input x or there is some S'' in F such that $S' \xrightarrow{x/y} S''$. If this condition holds then the algorithm adds S' to F. If the set $\{s_1, s_2, \ldots, s_n\}$ is added to F then the algorithm jumps to a second phase.

Otherwise, if at a given iteration, no addition is made to set F, the algorithm exits with a "No ADS" message.

In the second phase, the algorithm constructs an ADS. First, the algorithm forms $T = \{\{s_1, s_2, \ldots, s_n\}\}$ and then enters an iterative process. At every iteration the algorithm picks a set $t \in T$, that has not been picked before, and removes t from T. It then generates new sets by splitting t through using the splitting sequences found in the first phase. When all sets with cardinal larger than one are processed the algorithm terminates. We direct the reader to the literature [Kushik et al. 2013] for details of the algorithm.

4.1.2. Constructing PDSs. The algorithm provided in [Kushik et al. 2014] can construct one of the shortest PDSs for a given FSM. The algorithm uses the notion of a successor: given an ordered pair (s_p, s_q) of states, input x, and output y, we say that (s'_p, s'_q) is an x/y-successor of (s_p, s_q) if $(s_p, x, y, s'_p), (s_q, x, y, s'_q) \in h$. The algorithm (BF-PDS) relies on the construction of a tree data structure called the *successor tree*. The root of the tree contains a single set: the set of ordered pairs of states (s_p, s_q) of M such that $p < q$. A node u of the successor tree holds a set $NC(u)$ that contains sets of pairs of states. Each edge of the tree is labeled by an input $x \in X$. Let us suppose that u is a node, $A \in NC(u)$, input x is defined in all states in pairs in A, and y is an output. If there is an ordered pair $(s, s') \in A$ that has an x/y successor and v is the node reached from u via input x then one of the sets in $NC(v)$ is the set of x/y-successors of pairs in A. Thus, if A is a set in $NC(u)$ then A contains a set of pairs reached, from pairs in the root node, via a common input/output sequence.

If there exists an edge, with input x, from node u to node v such that $NC(v)$ contains a set that contains a singleton (a pair (s, s) for some state s), then x merges at least one pair in a set in $NC(u)$ and so the corresponding input sequence $\bar{x}$ cannot be a prefix of a PDS: $\bar{x}$ maps a pair of states to s with some common output sequence. If there exists an edge from node u to node v such that v is the empty set then concatenation of the inputs on the path from the root to that node defines a PDS for M. The tree is truncated when i) the node is empty, ii) the node has already been introduced to the successor tree, iii) the node is a singleton. The tree is constructed in the usual way, for a given node the algorithm applies defined inputs and depending on the outputs and the next states, introduce new nodes. Therefore, the tree requires exponential space. For further description of the algorithm we direct the reader to the reference [Kushik et al. 2014].

4.2. Aims of the experiments

The experiments explored two aspects of the proposed algorithms that are of practical importance: the time required to construct ADSs and PDSs and the cost (height for ADSs and length for PDSs) of these. Naturally, the lower the cost and the time of derivation, the better the approach.

4.3. FSMs used in the experiments

4.3.1. Randomly generated FSMs.

FSM SET I*:* The FSMs in this suite were designed to investigate the performance of the methods under varying FSM specifications (varying number of states and varying number of inputs and outputs). For a given $n \in \{8, 16, \ldots, 32768\}$ and number of inputs/outputs $\alpha/\beta \in \{2/2, 2/5, 2/8, 5/2, 5/5, 5/8, 8/2, 8/5, 8/8\}$, we constructed 100 FSMs[6]. So this test suite contained a total of 11700 FSMs.

[6]Note that we were unable to construct larger FSMs in an acceptable amount of time.

The FSMs were constructed as follows: first, for each input x and state s we randomly assigned the values of destination state (s') and output (y) of a transition, we repeated this operation $\alpha * n$ times (α transitions from each state) to obtain an FSM M. Afterwards, we randomly selected a *transition saturation ratio* ($\mathcal{T}$). For the FSMs in FSM SET I we randomly generated $\mathcal{T}$ from the range 0.10 to 0.30. We now explain how this value was used.

For a given FSM M, we marked $\mathcal{T} * (\alpha * n)$ transitions and we modified these transitions: we first removed a marked transition, say with input x, from M, and we either did nothing[7] or we added at least 2 and at most β additional transitions labelled with the same input (input x) but different outputs. These transitions were added randomly and the use of different outputs ensured that the FSM was observable. Note that as the value on $\mathcal{T}$ might affect the performance of the algorithms, in FSM SET II we used higher $\mathcal{T}$ values (see below).

After an FSM M was generated we ran the parallel ADS generation algorithm on M. If the algorithm returned an ADS within 1500 seconds then we kept this FSM and otherwise we discarded it and continued with the process. Consequently, all constructed FSMs had ADSs.

FSM SET II*:* These FSMs were used to explore the effect of the transition saturation ratio ($\mathcal{T}$). We constructed the same number of FSMs as in FSM Set I with the same properties (with the exception of the value of $\mathcal{T}$) and so FSM SET II also contained 11700 FSMs. However, the transition saturation ratio was higher: it was randomly chosen from the range 0.30 to 0.70.

FSM SET III*:* There is a polynomial time algorithm for constructing ADSs for deterministic complete FSMs [Lee and Yannakakis 1994] (LY algorithm). This paper explored a more general class of FSMs and so the algorithms devised can also be applied to deterministic complete FSMs. We therefore performed experiments on deterministic complete FSMs to see how the proposed parallel ADS algorithm compares with the LY algorithm on such FSMs.

The FSMs in this class were generated as follows. First, for each input x and state s we randomly assigned a next state and an output. Once an FSM M had been generated we tested whether M has an ADS using the LY algorithm. We discarded FSMs with no ADS.

By following this procedure we constructed FSMs with the same properties (number of states etc.) as in FSM SET I and so again used 11700 FSMs.

4.3.2. FSMs with long ADSs*:* FSM SET IV*.* Randomly generated FSMs will tend to have shorter separating/reducing sequences, hence ADSs and PDSs are much shorter than the upper bound. Sokolovskii introduced a special class of (complete and deterministic) FSMs that we called *s-FSMs* [Sokolovskii 1971]. The length of the shortest separating/reducing sequence for states s_1 and s_2 of an s-FSM is exactly $n - 1$, and the lower bound on the height of the ADS is $n^2/4 - 1$, where n is the number of states of the FSM. We performed additional experiments with a set of s-FSMs to explore how the algorithms perform when the separating/reducing sequences are relatively long. The next states and the outputs of an s-FSM are given as follows in which $n' = n/2$ and $n > 2$.

[7]Note that if we do not add a transition then the underlying transition will be deleted which allowed us to generate partial FSMs.

Table I: Properties of specifications used in the experiments, where $post|h|$ refers to number of transitions after non-deterministic transitions are added.

Property	*bbse*	*cse*	*ex2*	*ex3*	*ex5*	*ex7*	*keyb*	*kirkman*	*lion*	*mark1*	*planet*	*sand*	*sse*	*styr*	*train4*	*train11*
$\|S\|$	16	16	19	10	9	10	19	17	4	16	48	32	16	30	4	11
$\|X\|$	128	128	4	4	4	4	128	4096	4	42	128	2048	128	512	4	4
$\|Y\|$	22	17	4	4	4	4	4	64	2	5132	9292	129	22	211	2	2
$\|h\|$	512	416	13	6	14	13	0	1536	14	0	1664	51072	512	7024	11	19
$\|post(h)\|$	5264	6528	249	126	86	105	10266	228864	16	254656	321648	323712	5264	398256	17	31

$$\text{Next state if } x_j \text{ is received from state } s_i = \begin{cases} s_{i+1}, & \text{if } j = 1 \wedge i \neq n' \wedge i \neq n \\ s_1, & \text{if } j = 1 \wedge i = n' \\ s_{n'+1}, & \text{if } j = 1 \wedge i = n \\ s_i, & \text{if } j = 0 \wedge 1 \leq i \leq n' - 1 \\ s_{n'+1}, & \text{if } j = 0 \wedge i = n' \\ s_{i-n'}, & \text{if } j = 0 \wedge n' + 1 \leq i \leq n \end{cases} \quad (1)$$

$$\text{Output if } x_j \text{ is received from state } s_i = \begin{cases} y_0, & \text{if } j = 0 \wedge i = n \\ y_1, & \text{otherwise} \end{cases} \quad (2)$$

We generated 6 s-FSMs with n states, $n \in \{10, 20, \ldots, 60\}$.

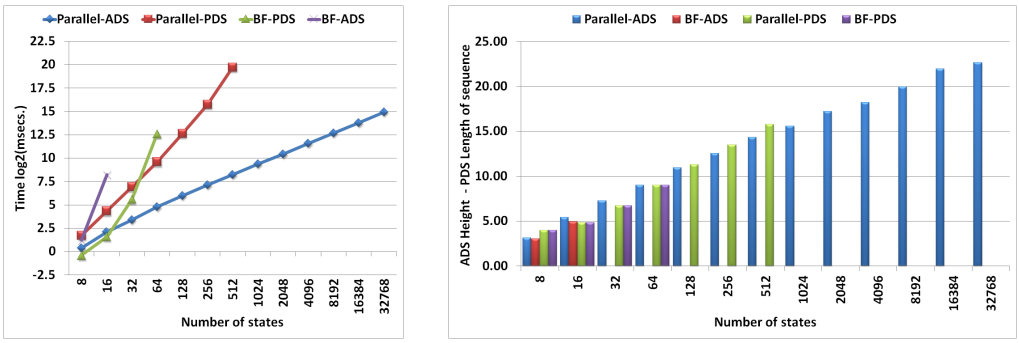
4.3.3. Benchmark FSMs. It is possible that FSM specifications of real life systems are unlike these randomly generated FSMs. We therefore complemented the experiments with some case studies: FSM specifications retrieved from the ACM/SIGDA benchmarks, a set of FSMs used in workshops in *1989–91–93* [Brglez 1996].

The specifications are given in the $kiss2$ format where state names are provided with alphanumeric symbols $(0, 1, \ldots, 9, A, a, B, b, \ldots, Z, z)$ and each input/output is represented by a symbol in $\{0, 1, -\}$. Note that a transition with outputs that contain $-$ defines a number of transitions. For example, if the FSM has a transition $(st1, 01, 0-, st2)$ then this transition actually encapsulates transitions $(st1, 01, 00, st2)$ and $(st1, 01, 01, st2)$. Therefore, such FSMs are in essence observable and non-deterministic. We observed that 32% of the FSM specifications were observable and non-deterministic. In order to use these FSMs we applied a process that produces the transitions that result from 'completing' a transition with a $-$ symbol. We present a number of properties of these FSMs in Table I in which $|h|$ is the number of transitions before the process was applied and $|post(h)|$ is the number of transitions once this process has completed.

4.4. Results

4.4.1. Effect of the number of states. We first investigate the effect of the number of states on the performance of the algorithms and we present the results for the FSMs in FSM SET I where $\alpha/\beta = \{2/2\}$.

Figures 3a and 3b show that the time required to construct ADSs and PDSs increases with the number of states. In addition, as the number of states increases, so does the height of ADSs and length of PDSs returned. The experiments also revealed that the time required to construct PDSs grows faster than the time to construct ADSs. The parallel PDS algorithm could only produce PDSs (in 1500 seconds) when $n \leq 512$, while the parallel ADS algorithm could produce ADSs when $n \leq 2^{15}$. The BF-ADS generation algorithm could only generate ADSs when $n \leq 16$. Hence our ADS derivation algorithm was able to process inputs of up to 2048 times larger than the existing ADS generation algorithm. The BF-PDS generation algorithm could only generate PDSs when $n \leq 64$. Thus, the parallel PDS derivation algorithm was able to process inputs of up to 8 times larger than the BF-PDS generation algorithm.



(a) Average time required to construct ADSs

(b) Average Height/Length of constructed ADSs/PDSs

Fig. 3: Results of experiments with FSM SET I where $\alpha/\beta = 2/2$.

The height/length of the ADSs and PDSs appears to grow roughly linearly with the log of the number of states. Similarly, earlier work using FSMs with up to 10 states found that randomly generated FSMs have relatively short separating sequences when we are distinguishing a pair of states [Spitsyna et al. 2007]. Moreover, we observe that Parallel-PDS and BF-PDS algorithms found PDSs with same length.

4.4.2. Effect of number of inputs. The results of the experiments on varying number of inputs are given in Figure 4. We present the results for the FSMs in FSM SET I. The results indicated that the number of inputs has a limited effect on the time and the heights of ADSs (Figures 4a, 4b).

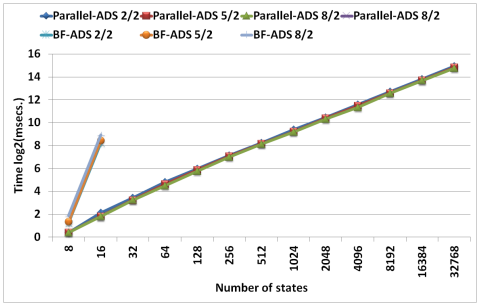
In order to investigate this further we applied the Kruskal-Wallis test [Kruskal and Wallis 1952], using the *R* tool [Stowell 2012]. We used the results generated by the Parallel-ADS/Parallel-PDS construction algorithms as these methods provide larger populations.

For a given n and number of outputs $\beta \in \{2, 5, 8\}$, we picked two different α values (number of inputs), $\alpha, \alpha' \in \{2, 5, 8\}$, where $\alpha \neq \alpha'$. Then for n and α/β we grouped (G_α) the lengths of ADSs and time required to derive the ADSs from FSMs with n states, α inputs, and β outputs. We also formed a group $G_{\alpha'}$ of the lengths of ADSs and time required to derive the ADSs for FSMs with n states, α' inputs, and β outputs.

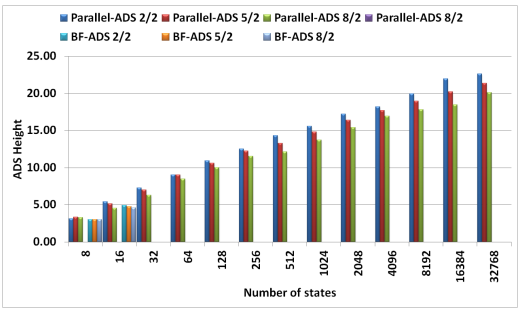
Afterwards we applied the Kruskal-Wallis test on G_α and $G_{\alpha'}$. To evaluate the effect of the number of inputs on the time required to derive ADSs we performed 234 tests (for each different n, β values and different α, α' pairings). This was then followed by reapplying the tests, for the heights of the ADSs. So in total we performed 468 test. We observed that at each test the null-hypothesis was accepted where the null-hypothesis is that 'the samples are from the same population'. As a result, we cannot reject the possibility of the different groups belonging to the same population.

However, we observe that the number of inputs can have an impact when considering PDSs. Although the length of PDSs reduces as the number of inputs increases, the time required to compute PDSs increases. When $n = 128$ and $\alpha \in \{5, 8\}$, the Parallel-PDS generation algorithm failed to terminate within 1500 seconds. The Kruskal-Wallis test rejected the null hypothesis in all different β values and different α, α' pairs when $32 \leq n < 128$.

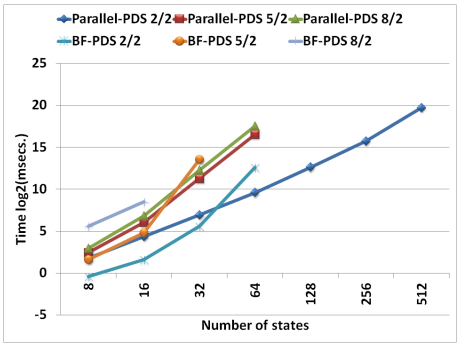
Similarly, we also found that the size of the input alphabet had almost no effect on the performance of the BF-ADS algorithm. However, we observed that the number of inputs has a significant effected on the performance of the BF-PDS algorithm. As



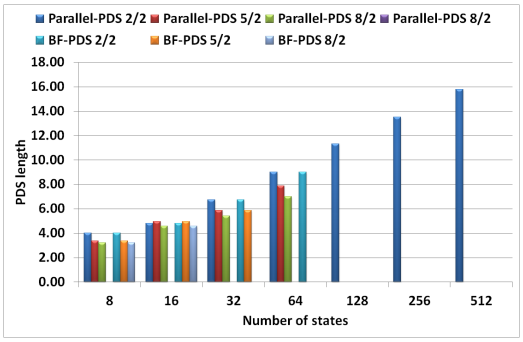
(a) Average time required to construct ADSs



(b) Average height of constructed ADSs



(c) Average time required to construct PDSs



(d) Average length of constructed PDSs

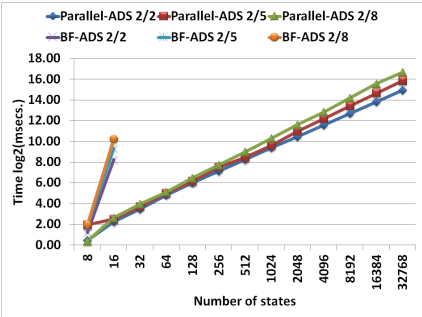
Fig. 4: Effect of number of inputs in FSM SET I where $\alpha \in \{2, 5, 8\}/\beta = 2$.

the number of inputs increases, the time necessary to compute PDSs increases. When $\alpha = 8$ and $n \geq 32$ the BF-PDS algorithm could not compute PDSs.

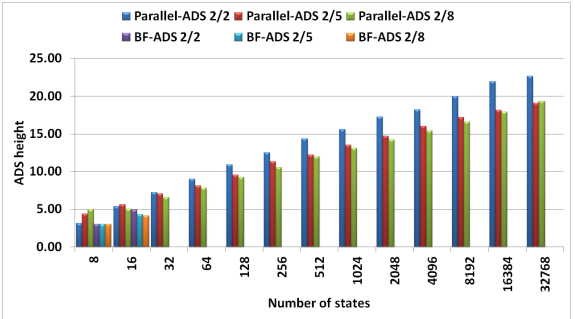
It is interesting to see that the number of inputs has a much more noticeable effect on the time taken to produce PDSs than to produce ADSs. One possible explanation relates to the fact that, when generating an ADSs, at each point in the algorithm there is a set of sets of current states and one can apply different inputs in these states. One might expect these sets of current states to be smaller than the corresponding set of current states produced when generating PDSs (for the same depth) and so the problem of finding a suitable next input will be more difficult for PDSs than for ADSs. If this is the case then this difference, in the impact that the number of inputs has, may result from the need to search through more inputs (when generating a PDS) in order to find a next input that can be applied.

4.4.3. Effect of number of outputs. We present the results of experiments conducted, using the FSMs from FSM SET I, to explore the effect of the number of outputs in Figure 5. Interestingly, the results obtained from the proposed approaches indicate that the time required to construct ADSs/PDSs increased with the number of outputs. This is not the expected result, since an increase in the number of outputs will tend to provide greater opportunity to separate states. In contrast, experimental results for deterministic complete FSMs [Türker et al. 2014, 2016; Türker and Yenigün 2014], indicate that there is an inverse relationship between the time required to construct ADSs and the number of outputs.

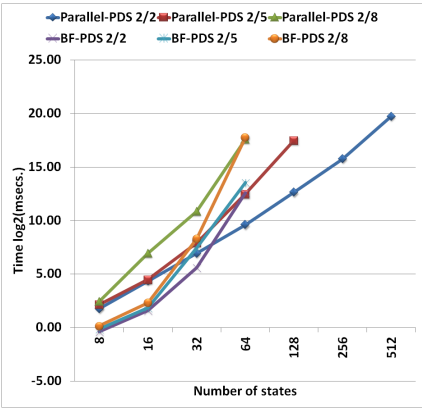
In order to further investigate this unexpected result we explored how many sets of states (OVs) were processed during the construction of ADSs. We present the results in



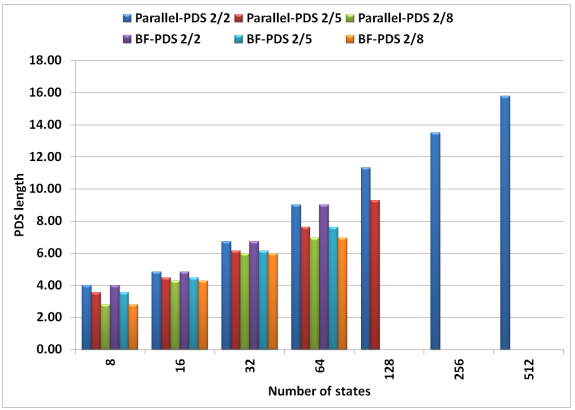
(a) Average time required to construct ADSs



(b) Average height of constructed ADSs

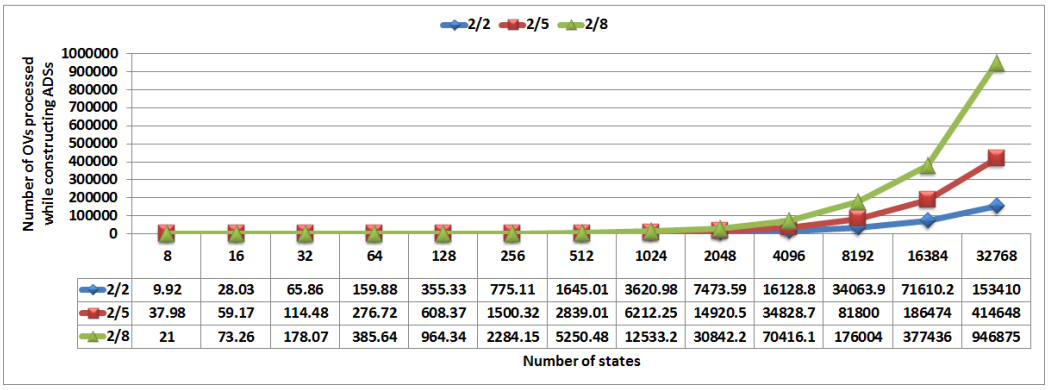


(c) Average time required to construct PDSs



(d) Average length of constructed PDSs

Fig. 5: Effect of number of outputs in FSM SET I where $\alpha = 2/\beta \in \{2, 5, 8\}$.



	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
2/2	9.92	28.03	65.86	159.88	355.33	775.11	1645.01	3620.98	7473.59	16128.8	34063.9	71610.2	153410
2/5	37.98	59.17	114.48	276.72	608.37	1500.32	2839.01	6212.25	14920.5	34828.7	81800	186474	414648
2/8	21	73.26	178.07	385.64	964.34	2284.15	5250.48	12533.2	30842.2	70416.1	176004	377436	946875

Fig. 6: Number of set of states (output vectors) processed while constructing the ADSs.

Figure 6. The results show that the number of sets of states processed increased with the number of outputs. In contrast, for deterministic FSMs, the number of 'current states' is always at most n.

Similarly, as we increase the number of outputs, the BF-ADS algorithm gets slower. However, the rate at which it slowed down was not as high as with the parallel ADS algorithm. In the BF-ADS much of the time was spent on the first phase of the algorithm. In other words, constructing the power set constitutes a large portion of the execution time of these methods and hence the effect of the sizes of the sets of inputs/outputs was limited. On the other hand, in the BF-PDS algorithm we observe a considerable improvement; the BF-PDS algorithm successfully computed all the PDSs when $n \leq 64$.

4.4.4. Effect of transition saturation ratio. The transition saturation ratio determines what percentage of the state/input pairs are (1) unspecified or (2) have multiple (nondeterministic) transitions. One would therefore expect the value of the transition saturation ratio to affect the performance of the proposed algorithms.

The results of the experiments indicate that the time required to construct ADSs and PDSs was almost twice as high (for $\alpha/\beta = 2/2$) when using the higher set of values for $\mathcal{T}$ (Figure 7). Moreover, we observe that the transition saturation ratio affects the length/height of PDSs/ADSs: as we increase the transition saturation ratio, the PDS/ADS cost increases.

What is more, we also observe that this is a consistent pattern; regardless of the other FSM properties (number of states and inputs/outputs) the time required to derive ADSs and PDSs and the cost of input sequences (height for ADSs and length for PDSs) increase as $\mathcal{T}$ increases.

To investigate the cause of this, we compared (Figure 8) the number of sets of states processed while constructing ADSs. We found that the number of sets of states almost doubled when the transition saturation ratio was increased from the interval $10\%-30\%$ to the interval $30\% - 70\%$.

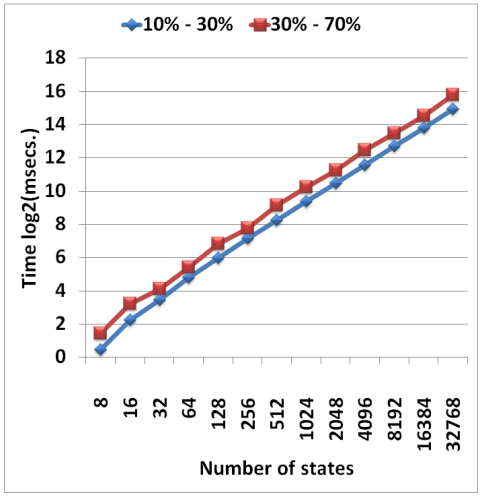
A similar pattern occurred with the BF-ADS and BF-PDS algorithms. As expected, the rate at which the BF-ADS algorithm slowed was not as high as in the parallel ADS algorithm. However, the rate at which the BF-PDS algorithm slowed was as high as in the parallel PDS algorithm.

4.4.5. Experiments with deterministic complete FSMs. The results on deterministic complete FSMs are promising. Figure 9 shows the time taken by the parallel ADS algorithm divided by the time taken by the LY algorithm. We observe that regardless of the properties of FSMs, the parallel ADS algorithm was faster than the sequential LY algorithm. Moreover, the difference in the speeds of algorithms appears to increase with the number of states. What is more, we also observe that the height of the ADSs were comparable (Figure 9).

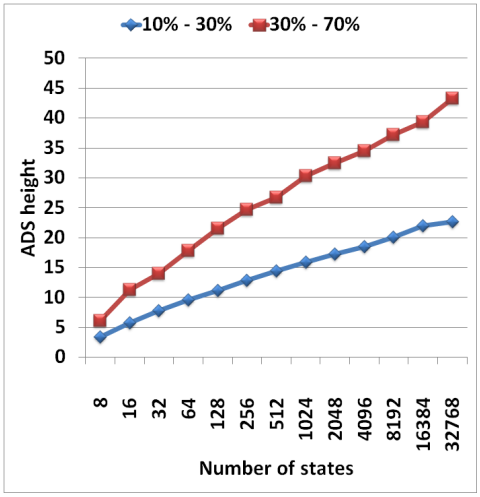
Recall that the parallel ADS algorithm tries to find reducing sequences for a given set of states (OVs) by searching all input sequences in a breadth-first manner. We conjectured that one reason for the parallel ADS algorithm performing better than the LY algorithm was that the FSMs in FSM SET IV were generated through a random process and thus input sequences that split/reduce a given set of states are expected to be relatively short.

In order to check this, we present the average length of reducing sequences found by the parallel ADS algorithm for deterministic FSMs in Figure 10. The results indicate that the average length of reducing sequences were less than 3.

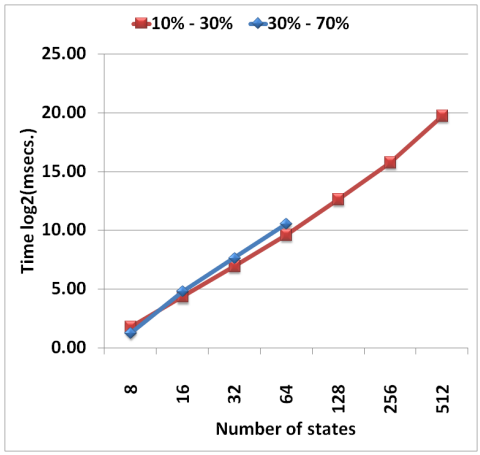
Moreover, our conjecture was also supported with experiments performed in s-FSMs (Table II). Note that for an s-FSM with n states, the longest reducing sequence has length $n-1$. We observe that the parallel ADS generation algorithm could not generate ADSs in 1500 seconds for s-FSMs with $n \geq 20$. On the other hand, the LY algorithm was able to construct ADSs for all s-FSMs.



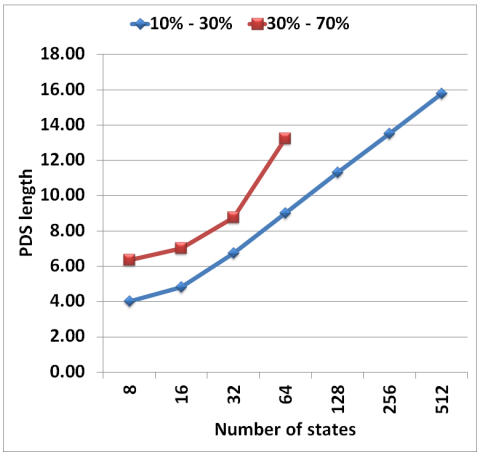
(a) Average time required to construct ADSs



(b) Average height of constructed ADSs



(c) Average time required to construct PDSs



(d) Average length of constructed PDSs

Fig. 7: Effect of $\mathcal{T}$. Comparison of results of experiments performed on FSM SET I and FSM SET II where $\alpha/\beta = 2/2$.

	10	20	30	40	50	60
LY	0.00	0.01	0.02	0.04	0.05	0.07
ADS	0.01					

Table II: Time required to construct ADSs for s-FSMs with the LY and the parallel ADS algorithms.

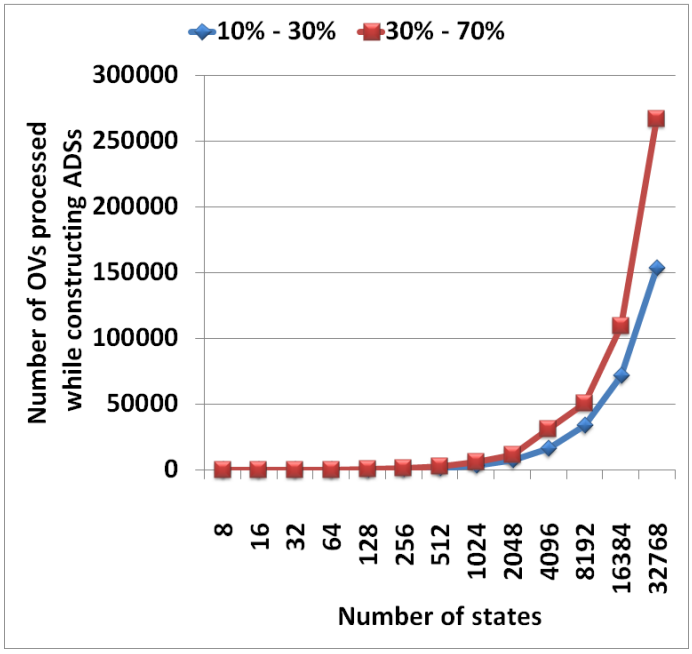


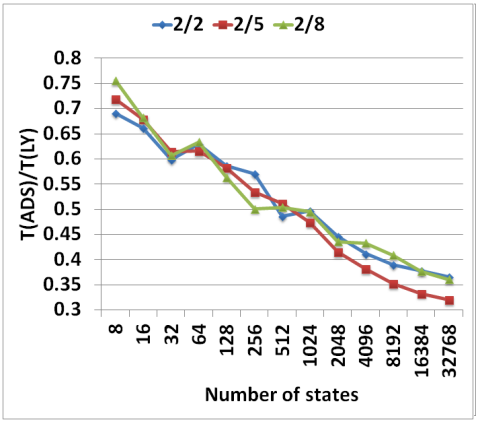
Fig. 8: Comparison of set of states (output vectors) processed while constructing the ADSs.

4.4.6. Experiments on benchmark FSMs. We applied the parallel ADS algorithm and the parallel PDS algorithm on each of the benchmark FSMs. The algorithms revealed that the specification $sand$ has an ADS but no PDS and the other specifications have no ADSs or PDSs. The parallel ADS algorithm used 5.98 seconds and processed 285 blocks. The height of the ADS was 2. The BF-ADS algorithm was unable to respond in 1500 seconds. Although the other specifications did not have ADSs (or PDSs), as noted above, it may be possible to find DSs for some sets of states (incomplete ADSs/PDSs). However, as the scope of this paper is constructing ADSs (or PDSs) for all the states of the FSMs, this is a topic for future work.

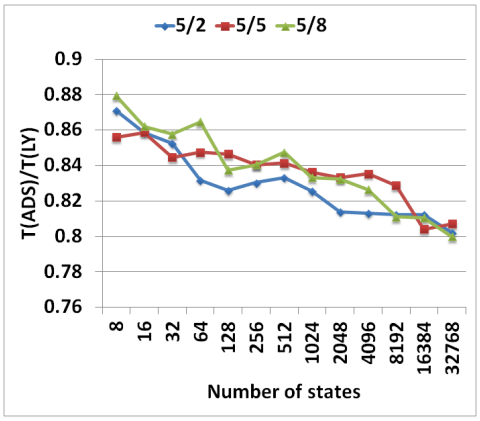
4.5. Discussion

Despite the importance of DSs in constructing test sequences, there were no published algorithms that use GPUs for generating DSs. In this paper we proposed massively parallel algorithms that can be executed on GPUs. The experimental results, with randomly generated and real-life FSMs, indicate that the proposed parallel algorithms are able to construct DSs from large FSMs. The proposed algorithm requires 2^{15} milliseconds to derive ADSs from FSMs with $32,000$ states on average. This is important: the results indicate that GPUs can effectively generate ADSs and PDSs from FSM specifications and this may lead to a line of research in which GPU-based parallel algorithms for FSM-based testing are investigated. Moreover, the proposed algorithm mostly uses GPU global memory and requires relatively little thread synchronisation. Therefore, the proposed algorithm can derive ADSs and PDSs from larger FSMs when multiple GPUs are used.

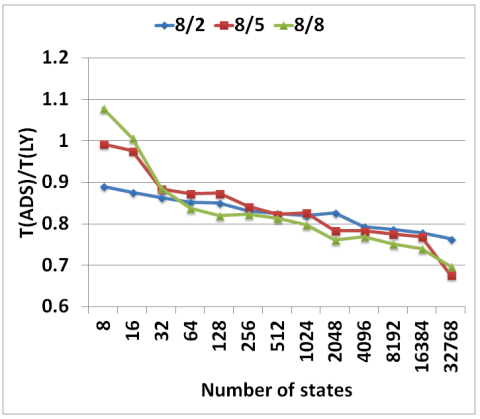
The proposed algorithm has one drawback: it is a brute-force algorithm. As the length of a shortest ADS for an FSM can be $2^n - 1 - n$ [Kushik et al. 2013], we cannot expect there to be a polynomial time algorithm for deriving DSs from FSMs. Conse-



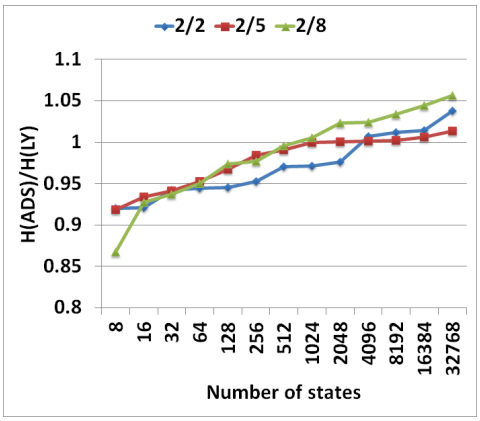
(a) Comparison of timings for LY and parallel ADS algorithms.



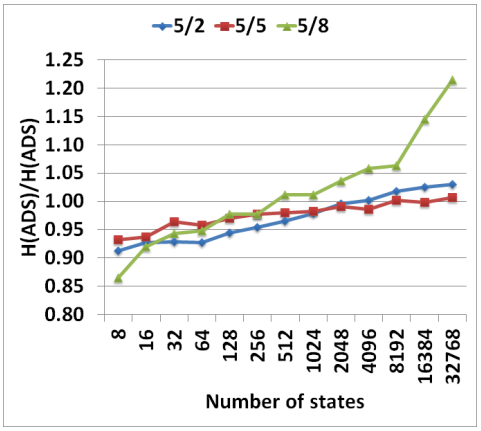
(b) Comparison of timings for LY and parallel ADS algorithms.



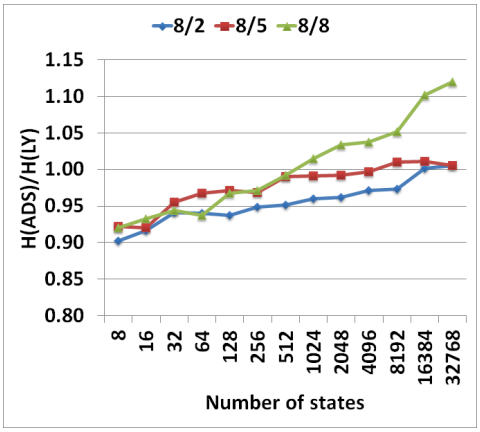
(c) Comparison of timings for LY and parallel ADS algorithms.



(d) Height comparison for LY and parallel ADS algorithms.

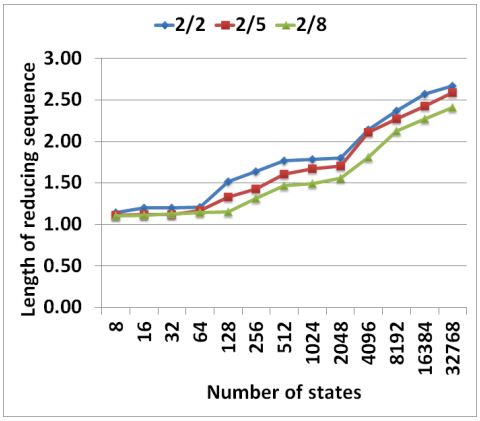


(e) Height comparison for LY and parallel ADS algorithms.

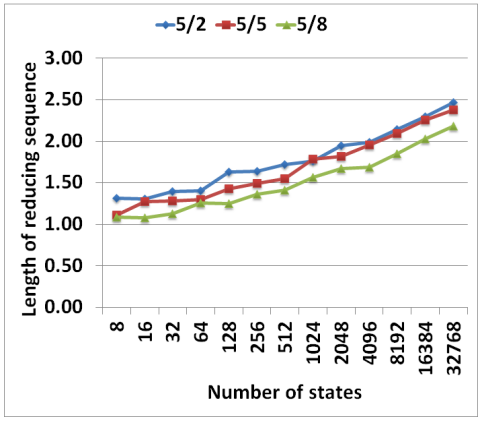


(f) Height comparison for LY and parallel ADS algorithms.

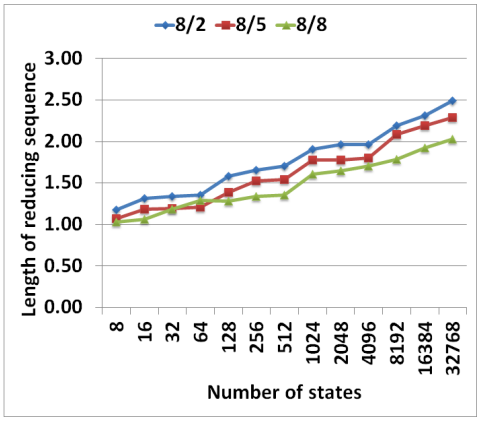
Fig. 9: Comparison of the parallel ADS and LY algorithms, where T stands for 'time' and H stands for 'height'



(a) FSMs with $\alpha = 2$



(b) FSMs with $\alpha = 5$



(c) FSMs with $\alpha = 8$

Fig. 10: Average length of reducing sequences constructed for FSMs in FSM SET IV

quently, we can only investigate heuristic algorithms for this problem and we consider this as a problem for future work.

5. THREATS TO VALIDITY

The threats to internal validity relate to the tools used in the experiments and the possibility that these are faulty. We carefully checked and tested the implementations of the algorithms. We also used a further procedure to check that an ADS/PDS returned was actually an ADS/PDS for the given FSM. The randomly generated FSMs were produced using a tool that has been successfully used in previous experiments.

Threats to construct validity refer to the possibility that we did not measure the properties that are of interest in practice. We were primarily interested in the time taken to generate DSs and the size of the largest FSMs that could be processed. The first factor is relevant since it is important that DSs can be produced in a reasonable amount of time; otherwise practitioners will not use tools that incorporate such algorithms. It is also important that techniques scale to large FSMs if FSM-based testing is to be used with larger systems. The size of a DS is also relevant since it affects the size of a test suite produced using the DS.

Threats to external validity concern our ability to generalise from the results. Unfortunately, there is no way of avoiding this threat since there is no know way of uniformly sampling from the set of all 'real' FSMs and this population is not known. We tried to reduce this threat by generating FSMs with varying properties and using some FSMs taken from a benchmark.

6. CONCLUSION

In this paper we addressed the scalability issue encountered while constructing adaptive and preset distinguishing sequences (ADSs/PDSs) from partial observable nondeterministic finite state machines (FSMs). We did this by utilising the parallelism available in GPU Computing. We outlined an initial algorithm that finds an input sequence to 'reduce' a set of states. We then presented the parallel ADS and parallel PDS algorithms. We provided high-level descriptions of the algorithms; to assist the reader we also present low-level descriptions of the algorithms in the Appendix.

The paper reported the results of an experimental study that used randomly generated FSMs and FSMs from a benchmark. In the experiments, we observed that the proposed parallel-PDS generation algorithm could derive PDSs from FSMs with 512 states and the proposed parallel-ADS generation algorithm could derive ADSs from FSMs with $32,000$ states. Moreover, the experimental results produced when using randomly generated complete deterministic FSMs suggest that the parallel-ADS generation algorithm is faster than the fastest known ADS generation algorithm (the LY algorithm) for complete deterministic FSMs. However, we also observed that unlike the LY algorithm, the parallel-ADS generation algorithm gets slower as the ADS height increases.

During the experiments we compared the results of the proposed algorithms with the results of existing algorithms and found that the proposed algorithms were much more scalable: the parallel-ADS generation algorithm was able to process inputs of up to 2048 times larger than the existing ADS generation algorithm and the parallel-PDS generation algorithm was able to process inputs of up to 8 times larger than the existing PDS generation algorithm.

As part of future work, we plan to investigate heuristics for deriving ADSs and PDSs from FSMs and aim to derive shorter ADSs and PDSs from FSMs. There is also the problem of generating 'incomplete' ADSs and PDSs that distinguish only some of the states of the FSM. It may be possible to further improve the performance of the parallel or brute-force algorithms by applying techniques developed by the verification community. Finally, there may be scope to extend this work to generate ADSs/PDSs to distinguish states of FSMs when testing is distributed.

Acknowledgments

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant #1059B191400424 and by the NVIDIA corporation.

REFERENCES

A. V. Aho, , R. Sethi, and J. D. Ullman. 1986. *Compilers, principles, techniques, and tools*. Addison-Wesley Pub. Co.

R. Alur, C. Courcoubetis, and M. Yannakakis. 1995. Distinguishing tests for nondeterministic and probabilistic machines. In *27th ACM Symposium on Theory of Computing*. 363–372.

M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. 2003. Towards a Tool Environment for Model-Based Testing with AsmL. In *Formal*

Approaches to Testing (LNCS), Vol. 2931. Springer-Verlag, Montreal, Canada, 252–266.

R. V. Binder. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley.

Adilson Luiz Bonifácio and Arnaldo Vieira Moura. 2014a. On the completeness of test suites. In *Symposium on Applied Computing (SAC 2014)*. ACM, 1287–1292.

Adilson Luiz Bonifácio and Arnaldo Vieira Moura. 2014b. Partial Models and Weak Equivalence. In *11th International Colloquium on Theoretical Aspects of Computing ICTAC 2014) (LNCS)*, Vol. 8687. Springer, 80–96.

Adilson Luiz Bonifácio and Arnaldo Vieira Moura. 2014c. Test Suite Completeness and Partial Models. In *12th International Conference on Software Engineering and Formal Methods (SEFM 2014) (LNCS)*, Vol. 8702. Springer, 96–110.

Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. 2010. GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*. 17–19.

Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. 2011. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Technology Transfer* 13, 1 (2011), 21–35.

R. T. Boute. 1974. Distinguishing Sets for Optimal State Identification in Checking Experiments. *IEEE Transactions on Computers* 23 (1974), 874–877.

Franc Brglez. 1996. ACM/SIGMOD benchmark dataset. Available online at http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html. (1996). Accessed: 2014-02-13.

E. Brinksma. 1988. A Theory For The Derivation of Tests. In *Proceedings of Protocol Specification, Testing, and Verification VIII*. North-Holland, Atlantic City, 63–74.

F. Busato and N. Bombieri. 2015. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems* 27, 8 (2015), 2222–2233. DOI:http://dx.doi.org/10.1109/TPDS.2015.2485994

Emanuela G. Cartaxo, Patricia D. L. Machado, and Francisco G. Oliveira Neto. 2011. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability* 21, 2 (2011), 75–100. DOI:http://dx.doi.org/10.1002/stvr.413

T. S. Chow. 1978. Testing Software Design Modelled by Finite State Machines. *IEEE Transactions on Software Engineering* 4 (1978), 178–187.

Adenilso da Silva Simão and Alexandre Petrenko. 2008. Generating Checking Sequences for Partial Reduced Finite State Machines. In *20th IFIP TC 6/WG 6.1 International Conference Testing of Software and Communicating Systems, 8th International Workshop on Formal Approaches to Testing of Software TestCom/FATES (LNCS)*, Vol. 5047. Springer, 153–168.

Adenilso da Silva Simão and Alexandre Petrenko. 2010. Checking Completeness of Tests for Finite State Machines. *IEEE Transactions on Computers* 59, 8 (2010), 1023–1032.

A.T. Dahbura, K.K. Sabnani, and M.U. Uyar. Aug. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE* 78, 8 (Aug), 1317–1326. DOI:http://dx.doi.org/10.1109/5.58319

Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. 2015. All-Pairs Shortest Path algorithms for planar graph for GPU-accelerated clusters. *Journal of Parallel and Distributed Computing* 85 (2015), 91–103. DOI:http://dx.doi.org/10.1016/j.jpdc.2015.06.008 {IPDPS} 2014 Selected Pa-

pers on Numerical and Combinatorial Algorithms.
A. Drumea and C. Popescu. 2004. Finite state machines and their applications in software for industrial control. In *27th Int. Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress*, Vol. 1. 25–29. DOI:http://dx.doi.org/10.1109/ISSE.2004.1490370
Jörg Dümmler and Sebastian Egerland. 2015. Interval-based performance modeling for the all-pairs-shortest-path problem on GPUs. *The Journal of Supercomputing* 71, 11 (2015), 4192–4214. DOI:http://dx.doi.org/10.1007/s11227-015-1514-9
E. Farchi, A. Hartman, and S. Pinter. 2002. Using a model-based test generator to test for standard conformance. *IBM systems journal* 41, 1 (2002), 89–110.
F. Al Farid, M. S. Uddin, B. Barman, A. Ghods, S. Das, and M. M. Hasan. 2015. A novel approach toward parallel implementation of BFS algorithm using graphic processor unit. In *International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*. 1–4. DOI:http://dx.doi.org/10.1109/ICEEICT.2015.7307536
A. D. Friedman and P. R Menon. 1971. *Fault detection in digital circuits*. Prentice-Hall.
Vahid Garousi, Lionel C. Briand, and Yvan Labiche. 2008. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *Journal of Systems and Software* 81, 2 (2008), 161–185. DOI:http://dx.doi.org/10.1016/j.jss.2007.05.037
A. Gill. 1962. *Introduction to The Theory of Finite State Machines*. McGraw-Hill, New York.
G. Gonenc. 1970. A method for the design of fault detection experiments. *IEEE Transaction on Computers* 19 (1970), 551–558.
Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor A. Braberman. 2011. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability* 21, 1 (2011), 55–71. DOI:http://dx.doi.org/10.1002/stvr.427
D. Harel and M. Politi. 1998. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, New York.
Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*. Springer, 197–208.
M. Haydar, A. Petrenko, and H. Sahraoui. 2004. Formal Verification of Web Applications Modeled by Communicating Automata. In *Formal Techniques for Networked and Distributed Systems FORTE (LNCS)*, Vol. 3235. Springer-Verlag, Madrid, 115–132.
F. C. Hennie. 1964. Fault-detecting experiments for sequential circuits. In *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*. Princeton, New Jersey, 95–110.
Robert M. Hierons, Guy-Vincent Jourdan, Hasan Ural, and Husnu Yenigun. 2009. Checking Sequence Construction Using Adaptive and Preset Distinguishing Sequences. In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*. IEEE Computer Society, 157–166.
Robert M. Hierons and Uraz C. Türker. 2014. Distinguishing Sequences for Partially Specified FSMs. In *NASA Formal Methods*, Julia M. Badger and KristiYvonne Rozier (Eds.). Lecture Notes in Computer Science, Vol. 8430. Springer International Publishing, 62–76. DOI:http://dx.doi.org/10.1007/978-3-319-06200-6_5
Robert M. Hierons and Uraz C. Türker. 2015. Incomplete Distinguishing Sequences for Finite State Machines. *The Computer Journal* 58, 11 (2015), 3089–3113. DOI:http://dx.doi.org/10.1093/comjnl/bxv041
Robert M. Hierons and Uraz C. Türker. 2016a. Parallel Algorithms for Testing Fi-

nite State Machines: Generating UIO sequences. *IEEE Transactions on Software Engineering (Accepted)* (2016).

Robert M. Hierons and Uraz C. Türker. 2016b. Parallel Algorithms for Testing Finite State Machines: Harmonised State Identifiers and Characterising Sets. *IEEE Transactions on Computers, (Accepted)* (2016).

Robert M. Hierons and Hasan Ural. 2006. Optimizing the Length of Checking Sequences. *IEEE Transactions on Computers* 55 (May 2006), 618–629. Issue 5.

Jared Hoberock and Nathan Bell. 2010. Thrust: A Parallel Template Library. (2010). http://thrust.github.io/ Version 1.7.0.

ITU-T. 1999. *Recommendation Z.100 Specification and Description Language (SDL)*. International Telecommunications Union, Geneva, Switzerland.

Capers Jones. 1986. *Programming Productivity (1st. ed.)*. McGraw-Hill, New York, NY.

M. Kapus-Kolar. 2014. On the Global Optimization of Checking Sequences for Finite State Machine Implementations. *Microprocessors and Microsystems* 38, 3 (2014), 208–215. DOI:http://dx.doi.org/10.1016/j.micpro.2014.01.007

David B. Kirk and Wen-mei W. Hwu. 2012. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann.

G. Klingbeil, R. Erban, M. Giles, and P. K. Maini. 2012. Fat versus Thin Threading Approach on GPUs : Application to Stochastic Simulation of Chemical Reactions. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (Feb 2012), 280–287.

I.V. Kogan. 1973. A Bound on the Length of the Minimal Simple Conditional Diagnostic Experiment. *Avtomat. i Telemekh.* 2 (1973).

Z. Kohavi. 1978. *Switching and Finite State Automata Theory*. McGraw-Hill, New York.

William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association* 47, 260 (1952), pp. 583–621.

N. Kushik, K. El-Fakih, and N. Yevtushenko. 2013. Adaptive Homing and Distinguishing Experiments for Nondeterministic Finite State Machines. In *Testing Software and Systems (Lecture Notes in Computer Science)*, H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.), Vol. 8254. Springer Berlin Heidelberg, 33–48.

Natalia Kushik, Khaled El-Fakih, Nina Yevtushenko, and Ana R. Cavalli. 2016. On adaptive experiments for nondeterministic finite state machines. *International Journal on Software Tools for Technology Transfer* 18, 3 (2016), 251–264.

Natalia Kushik, Nina Yevtushenko, and Ana R. Cavalli. 2014. On Testing against Partial Non-observable Specifications. In *9th International Conference on the Quality of Information and Communications Technology (QUATIC 2014)*. IEEE, 230–233.

Siyan Lai, Guangda Lai, Guojun Shen, Jing Jin, and Xiaola Lin. 2015. GPregel: A GPU-Based Parallel Graph Processing Model. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. 254–259. DOI:http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.184

D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul. 1996. Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach. *IEEE Transactions on Communications* 44, 5 (1996), 631–640. DOI:http://dx.doi.org/10.1109/26.494307

D. Lee and M. Yannakakis. 1994. Testing Finite-State Machines: State Identification and Verification. *IEEE Transactions on Computers* 43, 3 (1994), 306–320.

D. Lee and M. Yannakakis. 1996. Principles and Methods of Testing Finite-State Machines - A Survey. *Proceedings of the IEEE* 84, 8 (1996), 1089–1123.

Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 68, 12 pages. DOI:http://dx.doi.org/10.1145/2807591.2807594

S. H. Low. 1993. Probabilistic conformance testing of protocols with unobservable transitions. In *1993 International Conference on Network Protocols*. 368–375. DOI:http://dx.doi.org/10.1109/ICNP.1993.340890

G. Luo, A. Petrenko, and G. v. Bochmann. 1994a. Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines. In *The 7th IFIP Workshop on Protocol Test Systems*. Chapman and Hall, Tokyo, Japan, 95–110.

G. L. Luo, G. v. Bochmann, and A. Petrenko. 1994b. Test Selection Based on Communicating Nondeterministic Finite-State machines Using a Generalized Wp-Method. *IEEE Transactions on Software Engineering* 20, 2 (1994), 149–161.

Lijuan Luo, Martin Wong, and Wen mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th design automation conference*. ACM, 52–55.

Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.

E. P. Moore. 1956. Gedanken-Experiments. In *Automata Studies*, C. Shannon and J. McCarthy (Eds.). Princeton University Press.

Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. *ACM SIGPLAN Notices* 49, 4 (2014), 529–542.

Todd Mytkowicz and Wolfram Schulte. 2012. *Maine: A Library for Data Parallel Finite Automata*. Technical Report MSR-TR-2012-62. http://research.microsoft.com/apps/pubs/default.aspx?id=168379

Alexandre Petrenko, Adenilso da Silva Simão, and Nina Yevtushenko. 2012. Generating Checking Sequences for Nondeterministic Finite State Machines. In *Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 310–319. DOI:http://dx.doi.org/10.1109/ICST.2012.111

Alexandre Petrenko and Adenilso Simao. 2015. Generalizing the DS-Methods for Testing Non-Deterministic FSMs. *The Computer Journal* 58, 7 (2015), 1656–1672. DOI:http://dx.doi.org/10.1093/comjnl/bxu113

Alexandre Petrenko and Nina Yevtushenko. 2005. Testing from Partial Deterministic FSM Specifications. *IEEE Transactions on Computers* 54, 9 (2005), 1154–1165.

Alexandre Petrenko and Nina Yevtushenko. 2006. Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. In *5th Int. Workshop on Formal Approaches to Software Testing (LNCS)*, Vol. 3997. Springer, 118–133.

A. Petrenko, N. Yevtushenko, and G. v. Bochmann. 1996. Testing deterministic implementations from nondeterministic FSM specifications. In *Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of Communicating Systems*. Chapman and Hall, Darmstadt, Germany, 125–141.

Simon Pickin, Claude Jard, Thierry Jeron, Jean-Marc Jezequel, and Yves Le Traon. 2007. Test Synthesis from UML Models of Distributed Software. *IEEE Transactions on Software Engineering* 33, 4 (2007), 252–269.

I.K. Rystsov. 1976. Proof of an Achievable Bound on the Length of a Conditional Diagnostic Experiment for a Finite Automaton. *Cybernetics* 12, 3 (1976), 354–356.

K. Sabnani and A. Dahbura. 1988. A Protocol Test Generation Procedure. *Computer Networks* 15, 4 (1988), 285–297.

K.K. Sabnani, A.M. Lapone, and M.U. Uyar. 1989. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications* 37, 9 (Sep 1989), 940–948. DOI:http://dx.doi.org/10.1109/26.35374

Nadathur Satish, Mark Harris, and Michael Garland. 2009. Designing efficient sorting

algorithms for manycore GPUs. In *IEEE International Symposium on Parallel & Distributed Processing IPDPS 2009*. IEEE, 1–10.

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH '07)*. Eurographics Association, 97–106.

D. P. Sidhu and T.-K. Leung. 1989. Formal Methods for protocol testing: A detailed Study. *IEEE Transactions on Software Engineering* 15, 4 (1989), 413–426.

A.S. Simão and A. Petrenko. 2008. Generating Checking Sequences for Partial Reduced Finite State Machines. In *TestCom/FATES*. 153–168.

A.S. Simao and A. Petrenko. 2009. Checking Sequence Generation Using State Distinguishing Subsequences. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*. 48–56. DOI:http://dx.doi.org/10.1109/ICSTW.2009.25

Rick Smetsers, Joshua Moerman, and David N Jansen. 2015. Minimal Separating Sequences for All Pairs of States. *Unpublished Manuscript, available at http://cs.ru.nl/rick/files/sm2015.pdf* (2015).

M.N. Sokolovskii. 1971. Diagnostic Experiments with Automata. *Cybernetics and Systems Analysis* 7 (1971), 988–994. Issue 6. DOI:http://dx.doi.org/10.1007/BF01068822

Natalia Spitsyna, Khaled El-Fakih, and Nina Yevtushenko. 2007. Studying the separability relation between finite state machines. *Software Testing, Verification and Reliability* 17, 4 (2007), 227–241.

P. H. Starke. 1972. *Abstract Automata*. Elsevier, North-Holland, Amsterdam.

Sarah Stowell. 2012. *Instant R: An Introduction to R for Statistical Analysis*. Jotunheim Publishing. http://www.instantr.com/book

J. Tretmans. 1996. Conformance testing with labelled transitions systems: Implementation relations and test generation. *Computer Networks and ISDN Systems* 29, 1 (1996), 49–79.

Jan Tretmans. 2008. Model Based Testing with Labelled Transition Systems. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers (LNCS)*, Robert M. Hierons, Jonathan P. Bowen, and Mark Harman (Eds.), Vol. 4949. Springer, 1–38. DOI:http://dx.doi.org/10.1007/978-3-540-78917-8_1

Po-Chang Tsai, Sying-Jyan Wang, and Feng-Ming Chang. Aug. FSM-based programmable memory BIST with macro command. In *2005 IEEE International Workshop on Memory Technology, Design, and Testing, 2005. (MTDT)*. 72–77. DOI:http://dx.doi.org/10.1109/MTDT.2005.24

Uraz.C. Türker, T. Ünlüyurt, and H. Yenigün. 2014. Lookahead-Based Approaches for Minimizing Adaptive Distinguishing Sequences. In *26th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2014)*. 32–47. DOI:http://dx.doi.org/10.1007/978-3-662-44857-1_3

Uraz.C. Türker, T. Ünlüyurt, and H. Yenigün. 2016. Effective Algorithms for Constructing Minimum Cost Adaptive Distinguishing Sequences. *Information & Software Technology* 74 (2016), 69–85.

Uraz.C. Türker and H. Yenigün. 2014. Hardness and Inapproximability of Minimizing Adaptive Distinguishing Sequences. *Formal Methods in System Design* 44, 3 (2014), 264–294. DOI:http://dx.doi.org/10.1007/s10703-014-0205-0

Uraz C. Türker. 2015. Parallel Algorithm for Deriving Reset Sequences from Deterministic Finite Automata. – (2015).

H. Ural, X. Wu, and F. Zhang. 1997. On Minimizing the Lengths of Checking Sequences. *IEEE Transactions on Computers* 46, 1 (1997), 93–99.

Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-

based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

Anton Wijs and Dragan Bošnački. 2012. Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In *Model Checking Software: 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, Alastair Donaldson and David Parker (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–116. DOI:http://dx.doi.org/10.1007/978-3-642-31759-0_9

Lei Xie, Jiaolong Wei, and Guangxi Zhu. 2008. An improved FSM-based method for BGP protocol conformance testing. In *International Conference on Communications, Circuits and Systems*. 557–561. DOI:http://dx.doi.org/10.1109/ICCCAS.2008.4657835

M.C. Yalcin and H. Yenigün. 2006. Using Distinguishing and UIO Sequences Together in a Checking Sequence. In *Testing of Communicating Systems*, M. Ü. Uyar, A. Y. Duale, and M. A. Fecko. (Eds.). Lecture Notes in Computer Science, Vol. 3964. Springer Berlin Heidelberg, 259–273. DOI:http://dx.doi.org/10.1007/11754008_17

K. Zarrineh and S. J. Upadhyaya. 1999. Programmable memory BIST and a new synthesis framework. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers*. 352–355. DOI:http://dx.doi.org/10.1109/FTCS.1999.781072

Fan Zhang and To-Yat Cheung. 2003. Optimal transfer trees and distinguishing trees for testing observable nondeterministic finite-state machines. *IEEE Transactions on Software Engineering* 29, 1 (Jan 2003), 1–14. DOI:http://dx.doi.org/10.1109/TSE.2003.1166585

A. LOW-LEVEL DESIGN

A.1. Overview

The algorithms proposed in this paper are designed for many core architectures. Therefore some of the parts of the proposed algorithms need to be executed by the GPU (Device) and some portions are executed by the CPU (Host). In order to assist the reader we now explain how we refined the high-level design, given earlier, to produce an algorithm for GPU computing.

A.1.1. Performance considerations. It is important to reduce the effect of global memory access latency, especially since we apply the thin thread strategy in which relatively little data is stored locally. It is sometimes possible to combine several accesses, by a GPU, into one memory transaction, this being called a *coalesced memory transaction* [Kirk and Hwu 2012]. The basic idea is that when an item of global memory is accessed an entire line is retrieved and placed in cache. If another thread requests an adjacent item then this may be in the cache and so there is no need for an additional (slow) global memory access. We therefore used a storage layout that facilitates coalesced memory access.

All multiprocessors in a warp execute the same code and so there is a need to avoid *thread divergence* [Kirk and Hwu 2012]. Thread divergence occurs if a conditional statement leads to different branching on different threads, the problem being that the GPU will then serialise execution.

A.1.2. Data Structures. The GPU kernels called by the PSR, parallel PDS and parallel ADS algorithms use the following three data structures:

(1) FSM vector: the FSM vector stores the transition structure of the FSM. For state s and input x, for each output $y \in Y \cup \{\varepsilon\}$ the FSM vector returns next state $s' \in S \cup \{-1\}$. Here -1 denotes there not being a transition from s with label x/y.

The size of the FSM vector is therefore $n|X||Y|$. Note that for a thread t_i and input sequence $\bar{x}$ of length greater than 1, it might not be possible to coalesce reads on the FSM vector. For example, let us assume that we want to apply $\bar{x} = x_1x_2$ to s_i. For x_1, thread t_i will retrieve the next state ($S' = \{s_0, s_1, \ldots, s_{|Y|}\}$) information from the ith location of the FSM vector and it will then apply input x_2 to the states in S'. This causes thread t_i to access different locations of the FSM vector.

(2) Working vector (W): A working vector $W : \langle w_0, w_1, \ldots \rangle$ is a vector of any $type$ and is used as temporary storage.

(3) Output Node (N): An output node (ON) is a group of (four) working vectors to simulate an output vector. A single output vector can be simulated by working vectors ($N(I)$, $N(C)$, $N(i)$ and $N(o)$) where $N(I)$ holds the initial states, $N(C)$ holds the current states, $N(i)$ holds the input sequence, and $N(o)$ holds the output sequence associated with N. Please see Figure 11.

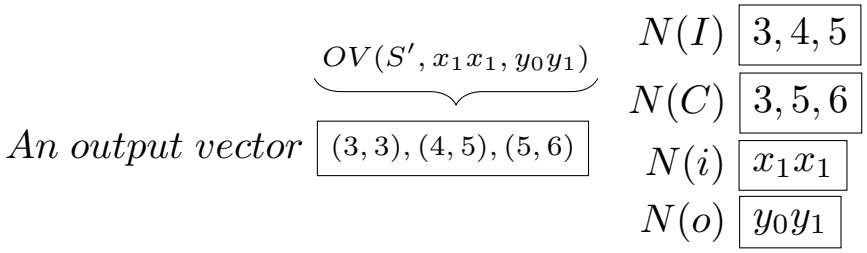


Fig. 11: How an output vector can be represented by a group of working vectors.

A.2. The parallel state reducing (PSR) algorithm

The PSR algorithm calls GPU code (1) while constructing the splitting vector (***SV** procedure*) and (2) to test if the SV corresponds to a reduction sequence for S' (***Red** procedure*). We first describe the **SV** procedure and then describe the **Red** procedure.

A.2.1. The SV procedure. When the **SV** procedure is called with an input sequence $\bar{x}$ and a set of states S', two vectors of output nodes ($Source$ and $Target$) are initialised in GPU memory. The algorithm then enters a loop that iterates over the inputs of $\bar{x}$.

At each iteration, a single input is applied to the current states in the $Source$ vector and the next states reached from these current states are written to the $Target$ vector. Therefore, an invariant of the **SV** procedure is that $|Target| = |Source| * \beta$. After the last input is applied the **SV** procedure ends.

We will be using brackets for elements of vectors. For example, $Target[i].N(C)[j]$ denotes the jth element of current state vector of the ith ON of the $Target$ vector. Moreover we will be using bold fonts to denote functions executed in GPU (kernels). The low-level design is given in Algorithm 9.

The $Source$ vector is initialised with a single ON $Source = \langle N_0 \rangle$ that contains the set of states and empty sequences $N_0 = ((s_1, s_1), (s_2, s_2), (s_m, s_m), \varepsilon, \varepsilon)$. The initialisation of N_0 is done by a copy kernel called $\mathbf{Copy}_{D \leftarrow H}$ that copies the set of states from host to device memory (i.e. copy S' to $N_0(I)$ and $N_0(C)$). The SV procedure then generates a copy (N_0') of N_0 (Line 1–2).

The $Target$ vector is initialised by the **CopyMultiple** kernel (Line 3). As discussed above, the $Target$ vector has β elements for each element in the $Source$ vector and so the **CopyMultiple** kernel copies N_0 to the $Target$ vector β times. To achieve this, the

CopyMultiple kernel receives the N_0 and $Target$ vectors, integer β, and is launched with $m * \beta$ threads where $|S'| = m$. Thread t_i computes the *source element index* (τ) which gives the index of the element to be read from $Source$. It also computes the *ON index* (κ), which gives the index of the element of ON in $Target$ to be written to.

$$\kappa = (i/m)$$
$$\tau = i - \kappa * m$$

Note that N_0 has four working vectors, but $N_0(i)$ and $N_0(o)$ are empty. In order to copy values from $N_0(I)$ and $N_0(C)$ to $Target$, thread t_i reads the τth element of a working vector associated with N_0 and writes it to the ith element of a working vector of the κth ON of the $Target$ vector. Thread t_i thus performs the following operations.

$$Target[\kappa].N(I)[\tau] \leftarrow N(I)[\tau]$$
$$Target[\kappa].N(C)[\tau] \leftarrow N(C)[\tau]$$

After the $Source$ and $Target$ vectors have been initialised, the **SV** procedure enters a while loop (host loop) (Line 4). The host loop iterates over the inputs from $\bar{x}$ and so the number of iterations is equal to the length of the input sequence. In each iteration, the **Apply** kernel is called (Line 5) with $|Source| * m$ threads. In the **Apply** kernel a thread t_i first computes κ and τ values, declares a variable $sum = 0$ and then enters a while loop (kernel loop).

The kernel loop iterates over the outputs of the FSM and so iterates β times. In the jth iteration, thread t_i retrieves the current state s from $Source[\kappa].N(C)[\tau]$. If $s < 0$, the thread t_i exits since the input sequence is not defined. Otherwise, given s and the label x_i/y_j, t_i finds the next state s' from the FSM vector. Thread t_i then adds the state number[8] of s' to sum and t_i writes s' and x_i/y_j to the $Target$ vector.

$$Target[\kappa * \beta].N(C)[\tau] = s'$$
$$Target[\kappa * \beta].N(i) = Target[\kappa * \beta].N(i)x_i$$
$$Target[\kappa * \beta].N(o) = Target[\kappa * \beta].N(o)y_j$$

After the kernel loop, t_i checks if $sum = -1 * m$. If so, x_i is not defined for s and so t_i writes -2 to $Target[\kappa * \beta].N(C)[\tau]$.

This is followed by copying the contents of the $Target$ vector to the $Source$ vector. In order to achieve this, the **SV** procedure needs to resize the $Source$ vector. This is simply done by reallocation of $|Target|$ elements to the $Source$ vector. Afterwards, the **Copy** kernel is called with $|Target| * m$ threads (Line 6). During the execution of the **Copy** kernel, thread t_i computes κ and τ values and copies all elements from the $Target$ to $Source$ vector. Thus, thread t_i performs the following operations.

$$Source[\kappa].N(I)[\tau] \leftarrow Target[\kappa].N(I)[\tau]$$
$$Source[\kappa].N(C)[\tau] \leftarrow Target[\kappa].N(C)[\tau]$$
$$Source[\kappa].N(i)[\tau] \leftarrow Target[\kappa].N(i)[\tau]$$
$$Source[\kappa].N(o)[\tau] \leftarrow Target[\kappa].N(o)[\tau]$$

Note that a single thread can read and write the input and output sequences of an ON, but we allow threads to write the same data to the same place to prevent thread divergence. This is followed by the expansion of the $Target$ vector from size $|Target|$ to size $|Source| * \beta$. Then the **CopyMultiple** kernel is called (Line 7) with N_0'.

[8]All states other than -1 have positive state numbers and state -1 has state number -1.

ALGORITHM 3: Procedure SV.

Input: A state set $S' \subseteq S$, an input sequence $\bar{x} = \{x_1, x_2, \ldots, x_L\}$
Output: Splitting vector for S'
begin
1 **Copy**$_{D \leftarrow H}(N_0, S')$, $N_0' \leftarrow N_0$
2 $Source = \langle N_0 \rangle$
3 **CopyMultiple**$_{D \leftarrow D}(Target, N_0, \beta)$, $i \leftarrow 1$
4 **while** $i \leq L$ **do**
5 **Apply**$(Source, Target, x_i)$
6 Set size of $Source$ vector to $|Target|$, **Copy**$_{D \leftarrow D}(Source, Target)$
7 **CopyMultiple**$_{D \leftarrow D}(Target, N_0', |Source| * \beta)$
8 $i \leftarrow i + 1$
9 Return $Source$

A.2.2. The Red procedure. After the **SV** procedure returns, the PSR algorithm invokes the **Red** procedure (Line 9 of Algorithm 1) with variables $Source$ vector and $\bar{x}$. In the **Red** procedure, the algorithm first checks if the input sequence $\bar{x}$ is converging or not. To achieve this the algorithm applies the **Parallel_Unique** procedure [Hoberock and Bell 2010] to each of the OVs of $Source$ vector. If for a given OV $\mathcal{O}$ parallel unique returns an OV vector $\mathcal{O}'$ such that $|\mathcal{O}| \neq |\mathcal{O}'|$, the algorithm writes -2 to an element of the $Source$ vector.

Afterwards, the algorithm calls a kernel called **ReduceCheck**.

The **ReduceCheck** kernel uses a $Cardinality$ vector, which is a working vector of integers and its size is $|Source|$. The **ReduceCheck** kernel receives the $Source$ vector, a $Cardinality$ vector, integer m and a boolean variable $isSplit$ (which is initially set to be $True$). For each ON of the $Source$ vector, there is one associated element in the $Cardinality$ vector that is initially set to 0. The **ReduceCheck** kernel is launched with $|Source| * m$ threads where each thread t_i first computes κ and τ values and increments $Cardinality[\kappa]$ by one if $Target[\kappa].N(C)[\tau] \in S$ using the GPU's $atomicIncrement()$ operator. If $Target[\kappa].N(C)[\tau] = -2$ or $Cardinality[\kappa] = m$, $isRed$ is set to $False$.

A.3. The PDS Algorithm

Recall that the parallel PDS algorithm only differs from the PSR algorithm on line 9. Instead of calling **ReduceCheck**, the PDS algorithm calls the **DistinguishedCheck** kernel. The **DistinguishedCheck** kernel is similar to the **ReduceCheck** kernel: the only difference is that after incrementing $Cardinality[\kappa]$, thread t_i checks whether $Cardinality[\kappa] > 1$ or $Cardinality[\kappa] = -2$ and if so $isRed$ is set to $False$.

A.4. The ADS Algorithm

As in the case of the PSR algorithm, while implementing the ADS algorithm we used output nodes ON that are formed of four working sets ($N(I)$, $N(C)$, $N(i)$, and $N(o)$). The ADS algorithm iterates over an output node's vector called an ADS *vector* ($\mathbb{O}$). The ADS algorithm receives M and S' and it constructs an output node (N_0) with S' using the **Copy** kernel. Afterwards, the ADS algorithm enters a loop (the host loop). In the host loop, the ADS algorithm receives one ON (N) from $\mathbb{O}$ and drops this item from $\mathbb{O}$ if $|N(I)| > 1$. The algorithm then calls the PSR algorithm for $N(C)$. If the PSR algorithm returns an SV, the Append procedure takes place. Otherwise, the algorithm returns "No ADS".

A.4.1. The Append procedure. The **Append** procedure is used to add SV to $\mathbb{O}$, however, this is not as straightforward as a classical append operation performed in CPU memory. This is because, in the GPU we are not allowed to extend the size of a vector. Instead, we have to allocate space in global memory with size $|SV| + |\mathbb{O}|$ and we then need to copy the ADS vector $\mathbb{O}$ and SV to this memory location. In order to achieve

this, we first take a copy of $\mathbb{O}$ (called $\mathbb{O}'$) and expand the size of $\mathbb{O}$ to $|SV| + |\mathbb{O}|$. We then invoke the **Copy** kernel with $\mathbb{O}$ and $\mathbb{O}'$. Note that the **Copy** kernel will fill the first $|\mathbb{O}'|$ elements of $\mathbb{O}$ with values retrieved from $\mathbb{O}'$. Afterwards, the **Append** kernel is invoked. **Append** receives vectors SV and $\mathbb{O}$ and integer ℓ, where $|\mathbb{O}| = \ell$, as its parameters and launches $|SV| * m$ threads. Thread t_i first computes κ and τ values and then appends the contends of SV to $\mathbb{O}$ by performing the following steps.

$$\mathbb{O}[\kappa + \ell].N(I)[\tau] \leftarrow SV[\kappa].N(I)[\tau]$$
$$\mathbb{O}[\kappa + \ell].N(C)[\tau] \leftarrow SV[\kappa].N(C)[\tau]$$
$$\mathbb{O}[\kappa + \ell].N(i)[\tau] \leftarrow SV[\kappa].N(i)[\tau]$$
$$\mathbb{O}[\kappa + \ell].N(o)[\tau] \leftarrow SV[\kappa].N(o)[\tau]$$