Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 3544 – 3556

doi:10.3906/elk-1809-1

Parallel brute-force algorithm for deriving reset sequences from deterministic incomplete finite automata

Uraz Cengiz TÜRKER*
Department of Computer Engineering, Faculty of Engineering, Gebze Technical University, Kocaeli, Turkey

Received: 01.09.2018 • **Accepted/Published Online:** 26.03.2019 • **Final Version:** 18.09.2019

Abstract: A reset sequence (RS) for a deterministic finite automaton $\mathscr{A}$ is an input sequence that brings $\mathscr{A}$ to a particular state regardless of the initial state of $\mathscr{A}$. Incomplete finite automata (FA) are strong in modeling reactive systems, but despite their importance, there are no works published for deriving RSs from FA. This paper proposes a massively parallel algorithm to derive short RSs from FA. Experimental results reveal that the proposed parallel algorithm can construct RSs from FA with 16,000,000 states. When multiple GPUs are added to the system the approach can handle larger FA.

Key words: Finite automata, incomplete finite automata, brute-force approach reset sequences, GPGPU programming

1. Introduction

Finite automata (FA) have many practical applications. They have been used in various fields, including automata theory, robotics, biocomputing, set theory, propositional calculus, model-based testing, and many more [1–12].

For example, in model-based testing, checking the experiment construction requires a reset sequence to bring the implementation to the specific state in which the designed test sequence is to be applied (e.g., see [13–15]). In [5], the authors studied a practical problem related to automated part orienting on an assembly line. In this work the authors showed that after some assumptions the part orienting problem is reducible to the problem of constructing reset sequences for deterministic finite automata. Another interesting example arises in biocomputing. In [16, 17], the researchers showed that in a controlled environment they ran $3 * 10^{12}$ automata per microliter, performing 6.6×10^{10} transitions per second, and in order to run these automata, we need to construct reset words from synthetic nucleotides. Moreover, in [18], the authors proposed a molecular automaton that plays Tic-Tac-Toe against a human opponent. Such an automaton, after the game ends, requires a reset word to bring the automaton to the "new game" state.

1.1. Formal background

A deterministic finite automaton (or simply an automaton) is defined by a triple $\mathscr{A} = (Q, \Sigma, \delta)$ where Q is a finite set of states, Σ is a finite input alphabet, and $\delta : Q \times \Sigma \to Q$ is a transition function. For a given state q, we say that input x is defined at q if $\delta(q, x) \in Q$. Automaton $\mathscr{A}$ is an incomplete automaton if for some state input pairs the function δ is undefined. Otherwise, it is a complete automaton. Throughout this paper we

*Correspondence: urazc@gtu.edu.tr

use automaton (or automata), finite automaton (or finite automata), or FA to denote an incomplete automaton (or incomplete automata). We assume that only one input can be applied to the automaton at a time. Upon receiving the input, the automaton changes its states according to its transition function. Application of inputs one after another forms a sequence of inputs $w = x_1x_2, x_3...$ and is called an input sequence. The transition function can be extended to input sequences in the usual way: for all $q \in Q$, input sequence $w \in \Sigma^\star$, and input symbol $x \in \Sigma$, $\hat{\delta}(q, \varepsilon) = q$, $\hat{\delta}(q, wx) = \delta(\hat{\delta}(q, w), x)$ where ε denotes the empty input sequence. Throughout this paper instead of $\hat{\delta}$ we use δ. Moreover, for a set $\bar{Q} \subseteq Q$, we use $\delta(\bar{Q}, w)$ to denote the set $\bigcup_{q\in\bar{Q}}\{\delta(q, w)\}$. For a FA, a word $w \in \Sigma^\star$ is said to be defined at a state $q \in Q$ if $\forall w', w'' \in \Sigma^\star$, $\forall x \in \Sigma$ such that $w = w'xw''$, $\delta(\delta(w'), x)$ is defined. We use symbol e to denote the state reached from a given state q with application of an undefined input at q, i.e. $\delta(q, x) = e$ iff x is not defined on state q. We use Σ^ℓ to denote the set of input sequences of length ℓ and we use n to denote the number of states of the automaton. An automaton $\mathscr{A} = (Q, \Sigma, \delta)$ is synchronizable if there exists an input sequence $w \in \Sigma^\star$ such that $|\delta(Q, w)| = 1$ and w is defined for Q. An automaton has a reset functionality if it can be reset to a single state by reading a special sequence w. In this case w is called a reset sequence (RS). On the other hand, an input sequence w is called a collapsing sequence for a set of states $\bar{Q}$ if and only if $|\delta(\bar{Q}, w)| < |\bar{Q}|$ [19].

1.2. Problem statement

Although reset sequences are important in many areas, the scalability of methods for constructing such sequences has not been addressed thoroughly. By scalability we refer to the maximum size of FA that can be processed in an acceptable amount of time. Thus, a more scalable algorithm can process larger FA. Besides, although FA can model a wide range of systems, previous approaches (except the methods given in [20–22]) for deriving RSs have been developed for deriving short reset sequences from complete FA [23–28] and surprisingly, to our knowledge, there are no proposed works for effectively constructing short RSs from FA.

One reason for this is clearly the difficulty of managing high memory/time demand when constructing RSs from FA. The only method for deriving RSs from such FA is the "successor tree" method [20–22]. In this method, a tree data structure called a successor tree for an FA is constructed by following a breadth-first search process until (1) the RS is computed or (2) the upper-bound is reached. However, as the size of a successor tree grows exponentially, an approach that constructs a successor tree would not process very large FA and hence leads to poor scalability. Therefore, considering the importance of RSs, we believe that the scalability problem is very important and hence needs to be addressed.

On the other hand, we have been witnessing that researchers have been taking advantage of general-purpose graphics processing units (GPGPU) technology in various problems including shortest path, breadth-first search, sorting, and many more [29–34]. Moreover, recently researchers proposed a parallel algorithm to accelerate the execution of a FA [35, 36]. Recently massively parallel algorithms have also been presented for deriving state identification sequences from finite state machines [37, 38]. Karahoda et al. recently represented a fast version of an existing greedy RS generation algorithm ([4]) for deterministic complete automata [39]. However, despite these significant advances, surprisingly, algorithms for deriving short RSs from complete and partial FA have not changed in 30 years and in this paper we aim to address this gap.

1.3. Contributions

In this paper, we introduce a revolutionary strategy to derive RSs. Existing approaches to generate RSs from FA use successor trees [23–28]. However, due to the high memory demand of such trees, we abolish this strategy

and introduce a vector-based approach by introducing a new and an efficient data structure that (1) is suitable for the memory architecture of GPUs and (2) demands less memory and effectively encapsulates data that are required to derive RSs. In other words, we propose a parallelizable data structure that demands less memory for generating RSs from FA. Based on the proposed formalism, we propose a massively parallel algorithm that can derive RSs from FA with many states. We present in Section 4 how the proposed algorithm can be implemented by using the CUDA tool-kit for CUDA-enabled GPUs. We present the results of experiments conducted to measure the scalability of the proposed algorithm. Experimental results show that the proposed algorithm can derive reset sequences from the FA with 16 million states in about 6 minutes.

1.4. Organization of the paper

This paper is organized as follows: in Section 2, we introduce data structures that are suitable for modern GPGPU hardware, and after that, we present a massively parallel algorithm for deriving reset sequences from deterministic incomplete finite automata. In Section 3, we present the experimental results. Experimental results suggest that the proposed algorithm is more scalable and effective in deriving reset sequences compared to the existing search-tree-based approaches. In Section 4, we draw conclusions and suggest some future work. We also provide implementation-related details in Section 4.

2. Parallel RS generation algorithm (PRSGA)

2.1. Algorithm design

The proposed algorithm relies on the thin thread strategy, in which threads are exposed to a very limited amount of data. Therefore, the number of threads that can be launched by a kernel is usually high [40].

The existing RS generation algorithms for partial FA aim to construct a successor tree for the given FA to derive RSs. Edges expanded from a common node of a successor tree have different input labels. Nodes are associated with the current states reached from the initial states through application of the input sequence formed by concatenating the edge labels on the path from the root to that node. Although successor trees are good in providing required information, they are very expensive to be kept in CPU memory (see Figure 1).

To address this bottleneck, we need a data structure that demands less memory space and is parallelizable. In order to achieve this, we need to highlight important aspects of a successor tree. First note that a path from the root to a leaf of a successor tree labels a unique input sequence w. Moreover, we say that w is a RS if the current states within the leaf are all the same. Therefore, if we somehow associate input sequences with current states, we can check whether w is a RS for the FA under consideration.

Unlike any other RS sequence derivation methods, the proposed parallel RS generation algorithm uses a states-vector to construct RSs. This, we believe, allows us to exploit the massive parallelism provided by modern GPUs. We now define states-vectors and present some of their properties.

Definition 1 *For a given automaton $\mathscr{A} = (Q, \Sigma, \delta)$, a states-vector V is a vector for states $\bar{Q} \subseteq Q$, associated with an input sequence $w \in \Sigma^\star$ having $|\bar{Q}| = m$ elements such that each element v is associated with an initial state $v_i = q \in Q$ and a current state $v_c = q' \in Q$ such that $\delta(q, w) = q'$. We use the notation $V[i]$ to denote the ith element of the states-vector and we use $U(V)$ to denote the number of distinct current states associated to V.*

The elements of a states-vector "advance" with input symbol x. Let v be an element of a states-vector

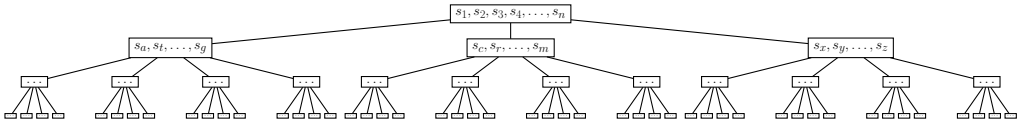


Figure 1. An example of a successor tree (edge labels are not given to reduce visual complexity).

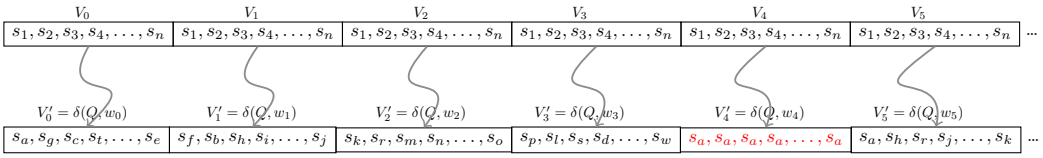


Figure 2. An example of a vector approach. Note that w_4 is a RS as after application of w_4 to Q we reach state s_a only.

associated with an initial state v_i and a current state v_c. If x is an input symbol then $adv(v, x) = v'$ if and only if $v_i = v'_i$ and $\delta(v_c, x) = v'_c$.

Note that when the elements of a states-vector advance with an input sequence w, one among four possible cases should occur. Let V be a states-vector for a set of states $\bar{Q}$, and then:

1. For each element $v_c \in V$, we have that $v_c = s$ where $s \in S$.
2. For at least two elements $v, v' \in V$, we have that $v_c = v'_c = s$ where $s \in S$, and for each element $v'' \in V \setminus \{v, v'\}$, we have that $v''_c = s'$ for some $s' \in S$.
3. For at least one element v_c of V, we have that $v_c = e$ such that $e \notin S$.
4. For any pair of elements v, v', $v \neq v'$, we have that $v_c \neq v'_c$ and $v_c = s, v'_c = s'$ and $s, s' \in S$.

Note that the first two possibilities state that w is a collapsing/reducing sequence for $\bar{Q}$. Moreover, if $\bar{Q} = Q$ and after advancing V with w, the first possibility occurs, and then w corresponds to a reset sequence for $\mathscr{A}$.

The following shows how states-vectors are related to collapsing sequences.

Lemma 1 *Let V be a states-vector for a set of states $\bar{Q}$ associated with input sequence $w \in \Sigma$. If the elements of vector V are advanced with w and for at least two elements $v, v' \in V$, we have $v_c = v'_c$, and for each element v of V we have that $v_c \neq e$, then w is a collapsing sequence for $\bar{Q}$.*

Although states-vectors provide some level of parallelism, we can further exploit the massive parallelism provided by a GPU through a group of states-vectors called a cluster.

Definition 2 *A cluster $\mathcal{C}_k$ for a set of states $\bar{Q}$ is a vector having k states-vectors for $\bar{Q}$. Intuitively, $\mathcal{C}_k[i]$ returns the ith states-vector and $\mathcal{C}_k[i][j]$ returns the jth element of the ith states-vector.*

Here, k is the cardinal of the cluster.

We are now ready to provide the intuition behind the proposed algorithm. The algorithm keeps a states-vector (V_{min}) and an input sequence (w_{min}) that will be used to keep the best choices made throughout the execution.

The algorithm iteratively forms a reset sequence R. At each iteration, for a given set of states V_{min}, the algorithm forms a cluster ($\mathcal{C}_k$) and then generates k different input sequences ($w_1, w_2, \ldots w_k$) from Σ^κ,

where k, κ are positive integers supplied to the algorithm. This is then followed by advancing the elements of cluster C_k. Upon completion, the algorithm checks whether the current states of at least one states-vector (V_i) of cluster $\mathcal{C}_k$ are the same. If so, then the algorithm concatenates w_i to R and declares that a RS is found and returns R. Otherwise, the algorithm searches for a minimum states-vector V_j on $\mathcal{C}$. A states-vector V_j of $\mathcal{C}_k$ is called minimum if (1) $U(V_j) < U(V_{min})$ and (2) for every element $V_i \neq V_j$ of $\mathcal{C}_k$ we have that $U(V_j) \leq U(V_i)$. If there exists one minimum states-vector V_j, it sets V_{min} as V_j and w_{min} as w_j and repeats these steps until either it finds a sequence that resets all the current states or all input sequences from Σ^κ are applied.

When all input sequences from set Σ^κ are applied to the elements of $\mathcal{C}_k$, then depending on the outcome, the algorithm follows one of the following options: (1) if neither a minimum states-vector nor a sequence that resets all the current states is found, then the algorithm increments κ by one and checks if κ is less then an integer (current length value) ℓ, where ℓ is supplied to the algorithm. If $\kappa < \ell$ the algorithm resets (i.e. generates k copies V_{min} and forms $\mathcal{C}_k$) $\mathcal{C}_k$ then it repeats the process mentioned above. Otherwise it returns the message "$\mathscr{A}$ *does not have a reset sequence*". (2) If all input sequences from set Σ^κ are applied and a minimum states-vector is found, i.e. V_{min} is updated, then the algorithm concatenates R with w_{min} and resets $\mathcal{C}_k$ and then repeats the process mentioned above.

Since all input sequences from set Σ^ℓ may be applied, the proposed algorithm requires exponentially many steps to find a RS or declare that the underlying automaton has no RS. However, as checking the existence of a RS from the FA is a PSPACE-complete problem, we cannot escape this consequence.

2.2. High-level algorithm overview

The outline of the algorithm is given in Algorithm. The algorithm receives an automaton $\mathscr{A}$ and positive integers $\ell, k \in \mathbb{Z}_{>0}$. The algorithm has three nested loops: OuterLoop, MiddleLoop and InnerLoop.

OuterLoop will iterate until all input sequences from set Σ^ℓ have been applied or a RS is constructed. In OuterLoop, the algorithm first checks whether all input sequences have been applied or not. If not, then it increments the current length value (κ) by one and enters MiddleLoop. Otherwise, it declares that "$\mathscr{A}$ does not have a reset sequence" and terminates (Lines 2–7).

MiddleLoop iterates at most $floor(|\Sigma|^\kappa/k) + 1$ times. In the MiddleLoop the algorithm carries out the following steps:

1. It resets[1] $\mathcal{C}_k$ (in parallel) (Line 9).
2. It retrieves k different input sequences $w_1, w_2, \ldots, w_k$ of length κ (Line 10).
3. It evolves elements of $\mathcal{C}_k$ (in parallel) (Line 11).
4. It enters the InnerLoop, which iterates over input sequences: at the ith iteration $1 \leq i \leq k$ the algorithm checks whether elements in $\mathcal{C}_k[i]$ are the same. If so, the algorithm returns w_i as a reset sequence (in parallel) (Lines 13–14). Besides, the algorithm searches for a minimum states-vector, and if it finds it it updates V_{min} (Lines 15–17).

Note that for a given k, the algorithm may not always retrieve k input sequences (Line 10) as k may not be a factor of $|\Sigma|^\kappa$. This happens when $|\Sigma|^\kappa \bmod k \neq 0$. In such a case, after $floor(|\Sigma|^\kappa/k)$ iterations (at the $floor(|\Sigma|^\kappa/k)+1$th iteration), the algorithm returns $|\Sigma|^\kappa \bmod k$ input sequences and continues to execute.

[1] For each i, j, it writes initial state value (q_j) to $\mathcal{C}[i][j]$.

Algorithm: Parallel RS construction algorithm. Highlighted states are executed in parallel.

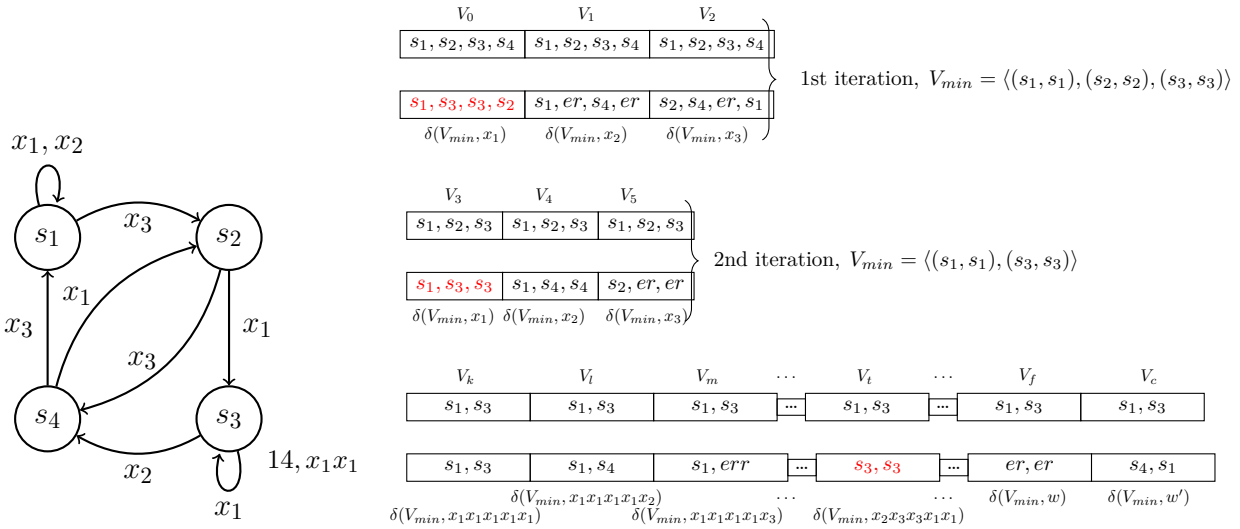
Input: An automaton $\mathscr{A} = (Q, \Sigma, \delta)$ and positive integers ℓ, k
Output: A reset seqeunce for $\mathscr{A}$
begin
1 $Execute \leftarrow True$, $\kappa \leftarrow 0$, $R \leftarrow \varepsilon$, $isFound \leftarrow False$, $w_{min} \leftarrow \varepsilon$, $V_{min} \leftarrow \langle (q_1, q_1), (q_2, q_2), \ldots, (q_n, q_n) \rangle$
`// Outerloop`
2 **while** *Execute is true* **do**
3 **if** $!isFound$ **then**
4 **if** $\kappa < \ell$ **then**
5 $\kappa \leftarrow \kappa + 1$
end
else
6 Declare that $\mathscr{A}$ is not synchronizing and terminate
end
end
else
7 $\kappa = 1$, $isFound \leftarrow False$
end
`// Middleloop`
8 **while** *There exists unprocessed* $w \in \Sigma^{\kappa}$ *and Execute is True* **do**
9 Reset $\mathcal{C}_k$ with V_{min}
10 Retrieve k input sequences $w_1, w_2, \ldots, w_k \in \Sigma^{\kappa}$ and associate them with $\mathcal{C}_k$
11 Evolve elements of $\mathcal{C}_k$
`// Innerloop`
12 **foreach** $w_i, 1 \leq i \leq k$ **do**
13 **if** *All elements of vector* $\mathcal{C}_k[i]$ *are the same* **then**
14 Return $R.w_i$
end
15 **if** $U(V_i) < U(V_{min})$ **then**
16 $V_{min} \leftarrow V_i$ and $w_{min} \leftarrow w_i$
17 $isFound \leftarrow True$
end
end
end
end
end

We now show the execution of the PRSGA using an example. Consider the FA $\mathscr{A}_1$ given in Figure 3(a). Let us suppose that $\mathscr{A}_1$, $\ell = 10$, and $k = 50$ are provided to the PRSGA. Then the algorithm will first set and increment κ and hence we have $\kappa = 1$. Since $k > 3$, the algorithm first constructs $\Sigma^1 = 3$ states-vectors. This is then followed by advancing the states-vectors with these inputs (top image of Figure 3(b)). Note that the algorithm fails to find a RS, but it finds that input sequence x_1 results in a minimum states-vector. Thus, the algorithm updates V_{min} with s_1, s_2, and s_3, and then the algorithm moves to the second iteration.

In the second iteration, the algorithm again constructs $\Sigma^1 = 3$ states-vectors by using the V_{min} vector and then it evolves the states-vectors with these inputs. Again the algorithm fails to find a RS, but it finds that

collapsing sequence x_1 results in a minimum states-vector (consisting of s_1, s_3). Thus, the algorithm updates V_{min} with s_1 and s_3, and then the algorithm moves to the next iterations.

The algorithm will keep applying this procedure until it finds a collapsing sequence that resets states s_1 and s_3. Note that when $\kappa = 5$, the PRSGA finds an input sequence $x_1x_3x_3x_1x_1$ that merges states s_1 and s_3.



(a) FA $\mathscr{A}_1$ with 3 inputs and 4 states

(b) Execution steps of PRSGA when $\mathscr{A}_1$, $\kappa = 10$, and $k = 50$ are provided to the algorithm, where er denotes the state labeled $ERROR$

Figure 3. An example of FA and execution steps of the PRSGA.

Clearly, for sufficiently large ℓ values, the PRSGA algorithm returns a RS if the underlying automaton $\mathscr{A}$ possesses one such reset sequence. However, as mentioned above, the algorithm may require an exponential amount of time and space for constructing such a sequence.

3. Experiments

In this section we first discuss what tools were used to construct FA and present the test data. Afterwards we present results of experiments conducted on FA and finally we present some discussions. We implemented the parallel reset generation algorithm (PRSGA) using CUDA. We present the implementation details in Section 4. We implemented the brute-force algorithm (BF(CPU)) described in [22][2] using C++. The experiments explored two aspects that are of practical importance: the time required to construct RSs and the length of these RSs. Naturally, the lower the length and the time of derivation, the better the approach.

3.1. FA generation

We used the automata generation method and the tool used in [41] to generate test cases: in this approach for a given n and Σ, we first randomly pick a completeness ratio $\mathcal{R}$ from an interval that we call the completeness interval, $\mathcal{I} = [low, high]$. Afterwards, we set transitions of FA $\mathscr{A}$: for each state q and for each input $x \in \Sigma$ we randomly set a state q'. Then we randomly drop $\mathcal{R} * n * |\Sigma|/100$ number of transitions from $\mathscr{A}$. This is

[2]Through constructing a successor tree as described in Chapter 13.2.

then followed by testing if the produced automaton had a computable[3] RS. If not, we discard this automaton; otherwise, we keep the produced automaton. Following this method, we constructed three sets of automata:

- The aim of test set SET_1 was to understand the performance of the algorithm under varying state sizes. We randomly generated automata with n states where $n \in \{2^6, 2^8, \ldots 2^{24}\}$ with ternary input and with $\mathcal{I} = 15\% - 25\%$. In SET_1 for each n we generated 100 FA.
- In SET_2 our aim was to see the effect of the number of inputs on the performance of the algorithm. Therefore, we randomly generated FA with $n = 2^8$ and $|\Sigma| \in \{2^4, 2^6, 2^8, 2^{10}\}$ and with $\mathcal{I} = 15\%$–25%. In SET_2 for each different $|\Sigma|$ value we constructed 100 FA.
- The aim of test set SET_3 was to understand the effect of the completeness interval. Hence, we randomly generated automata with 2^8 states with ternary input where $\mathcal{I} \in \{[5, 10], [15, 20], [25, 30], [35, 40], [45, 50], [55, 60], [65, 70]\}$. Again in SET_3 for each $\mathcal{I}$ we generated 100 FA.

In total, we used 2100 FA. In order to accomplish experiments in an acceptable amount of time, throughout the experiments, we set $\ell = 20$[4] and we allowed algorithms to finish their execution in 1250 seconds. Experiments are carried out on an Intel Core 2 Extreme CPU (Q6850) with 8GB RAM and NVIDIA TESLA K40 GPU under the 64 bit Windows Server 2008 R2 operating system. During generation we also counted the number of FA with RSs. Figure 4 gives the result. While forming SET_1, we noted the chance of constructing a FA with a RS. The probability of constructing an automaton with a RS is $> 89\%$.

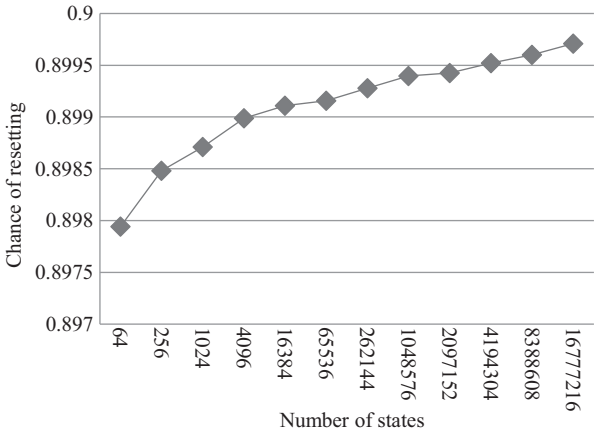


Figure 4. Probability of generating FA with RS.

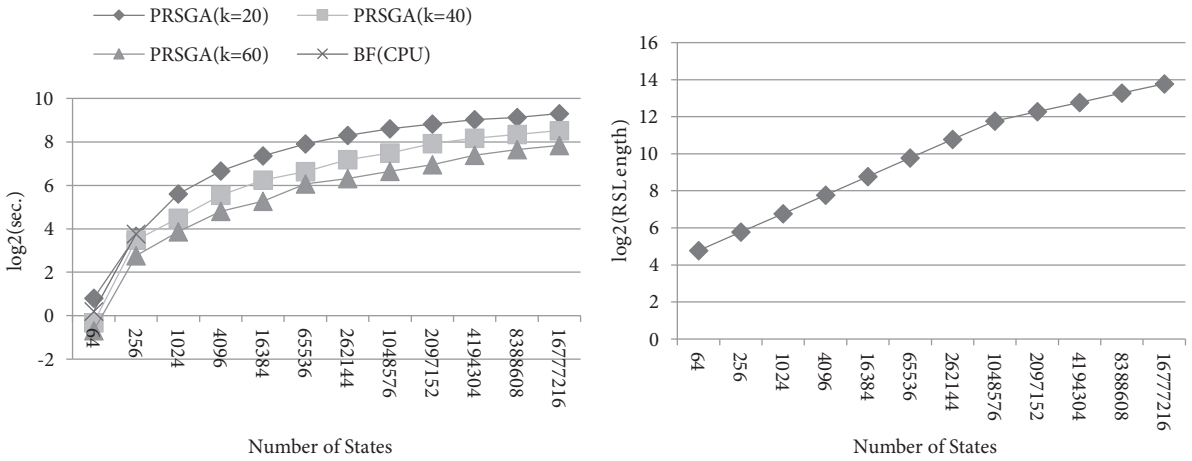
3.2. Effect of number of states

For each automaton in SET_1 we constructed RSs by using the PRSGA and BF(CPU). The results for the experiments performed on SET_1 are given in Figure 5(a). The results verify our intuition: the GPU-accelerated algorithm (PRSGA) is 9 times faster than the BF(CPU) construction algorithm (when $k = 60$ and $n \leq 256$). Note that the proposed algorithm construct RSs for FA with 16 million states in less than 228 seconds (in about 4 minutes) when $k = 60$ on average. As the proposed algorithm exploits the parallelism provided by the

[3]Note that a FA with n states may have a RS of exponential length. Therefore we set 1250 seconds to derive RSs.
[4]Note that for Σ and κ the number of input sequences to be generated is bounded above by Σ^κ.

GPU through processing k states-vectors, the number k affects the performance of the proposed algorithm. From Figure 5(a) we see that as the k value drops the time required to construct RSs increases. Moreover, we observe that BF(CPU) fails to construct RS for FA with $n > 256$ states within 1245.1 seconds, indicating that the proposed algorithm increases the scalability of constructing RSs of partial FA by 65536.

Note that the length of the RSs returned by the PRSGA algorithm does not depend on the parameter k and so we present the results obtained when $k = 60$. We see in Figure 5(b) that the average length of RS is not larger than 20 and increases with the number of states.



(a) Average time required to construct RSs for FA in SET_1 (b) Average length of RSs constructed for FA in SET_1

Figure 5. Experimental results on test set SET_1.

3.3. Effect of number of inputs

The results of the experiments conducted on SET_2 are given in Figure 6(a). Results are interesting; we observed that BF(CPU) could only return 2 RSs for FA with 16 inputs. We noted that lengths were both 3; moreover, BF(CPU) could return 1 RS for FA with 64 inputs where the length of the RS was 3. For all other cases BF(CPU) failed to return RSs due to insufficient memory or long computation time. However, the PRSGA algorithm could compute RSs for all of the FA. We observed that as the number of inputs increases, the time required to construct RSs increases as well regardless of the parameter k. Clearly this is as expected since the number of input sequences to be processed should increase with the number of inputs. On the other hand, we also observe from Figure 6(b) that the length of the RSs reduces as the number of inputs increases.

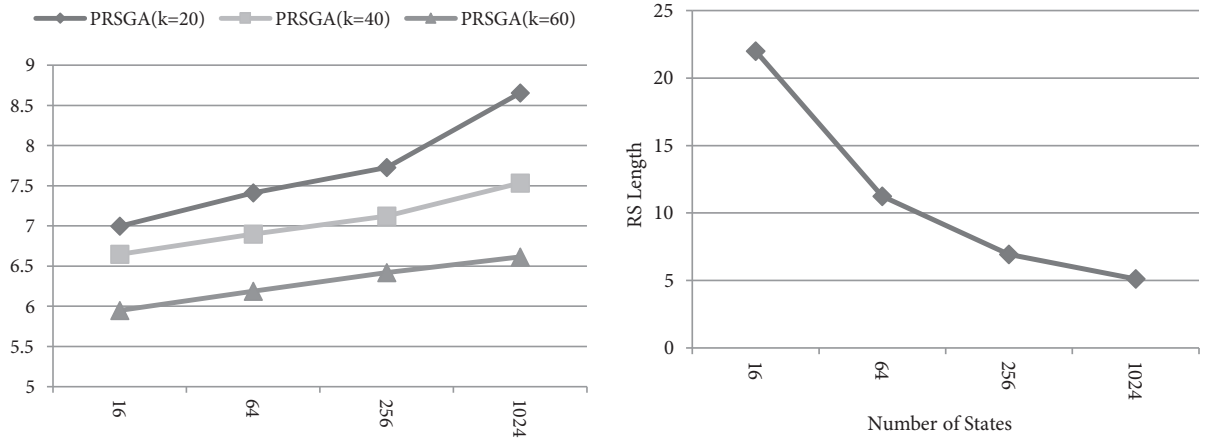
3.4. Effect of completeness interval ($\mathcal{I}$)

The completeness interval ($\mathcal{I}$) determines what percentage of the state/input pairs are unspecified. One would therefore expect the value of the completeness interval to affect the performance of the proposed algorithm.

The results of experiments indicate that the time required to construct RSs increases exponentially when using the higher set of values for $\mathcal{I}$ (Figure 7(a)). Moreover, we observe that the transition saturation ratio affects the length of RSs; as we increase the completeness interval, the length of RSs increases exponentially.

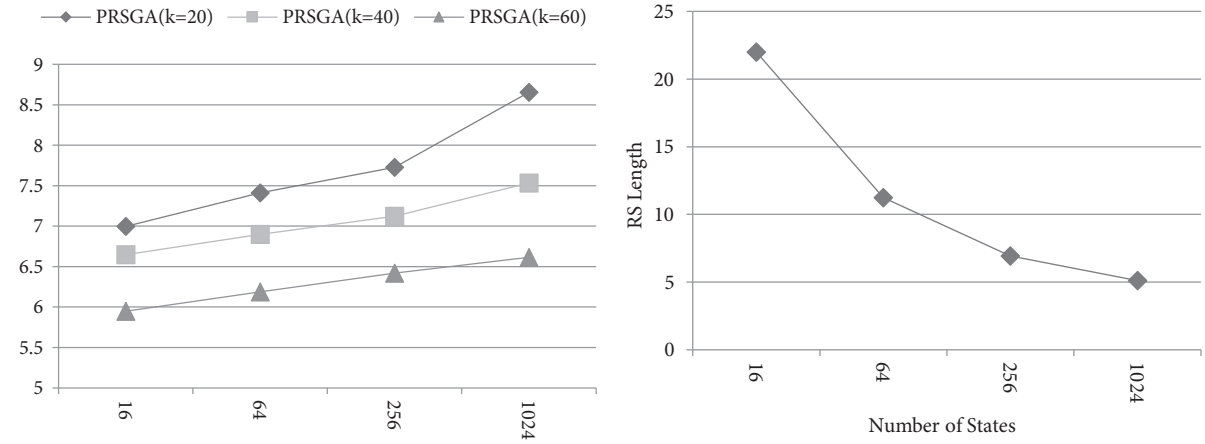
3.5. Threats to validity

This section briefly reviews the threats to validity and how these were reduced. We consider two types of threats: those to internal validity and external validity.



(a) Average time required to construct RSs for FA in SET_2 (b) Average length of RSs constructed for FA in SET_2

Figure 6. Experimental results on test set SET_2.



(a) Average time required to construct RSs for FA in SET_3 (b) Average length of RSs constructed for FA in SET_3

Figure 7. Experimental results on test set SET_3.

Threats to internal validity concern factors that might introduce bias. The main source of such threats is the tools used to run the experiments. The FA generation tool has been used in a number of projects and was tested. The implementations of these algorithms were carefully checked and also tested with a range of FA. To further reduce this threat, we also used an existing tool that checks if a given input sequence is a RS for the underlying FA. This tool was used to check all of the RSs generated by the PRSGA and BF(CPU).

Threats to external validity concern our ability to generalize from the experiments. There is always such a threat to validity since we do not know the space of relevant FA and certainly have no good way of sampling from this. We reduced this threat by using randomly generated FA. We also varied the number of inputs and states and the completeness interval.

3.6. Discussion

Recall that in Section 1, we observed that the PRSGA is an exponential algorithm; this cannot be avoided since determining the existence of a RS is a PSPACE-hard problem. As the lengths of the RS sequences generated

from the FA given in SET_1, SET_2, and SET_3 are not longer than the logarithm of the number of states, it appears that we have not found such long executions.

Another important point related to the PRSGA is the need to select the k-cluster parameter k. The experiments revealed that when we select a value for k that is too large, the algorithm gets faster as the size of input sequences to be processed increases. However, on the other hand, if we select a value of k that is too small then there is a risk of decreasing the GPU occupancy and increasing the memory traffic between the CPU memory and the GPU memory. Furthermore if we select a value of k that is too large, then there is a risk of calling too many threads that reduce the GPU performance due to the high interleaved transactions. Therefore, before the PRSGA, the parameter k should be selected carefully.

Experiment results presented in this paper suggest that the algorithm is capable of deriving RSs from FA and complete FA with 16 million states in less than 6 minutes. These results are very important for two reasons. First of all, there has been no work that can process such large FA. Second, with this method researchers can investigate properties of RSs on larger FA, which may lead to interesting research directions.

4. Conclusion

In this paper we addressed the scalability issue encountered while constructing RSs from incomplete FA through massively parallel GPGPUs. We first provided a high-level overview of the algorithm. We then explained the results of an experimental study done to measure the performance of the algorithm by using randomly generated FA. The results showed that the proposed algorithm can effectively derive RSs from large partial FA. We provide low-level descriptions for the algorithm in the Appendix.

There are several lines of future work. First, it would be interesting to study massively parallel heuristic algorithms for deriving RSs from FA. Second, despite the high energy requirement, it would be interesting to investigate multicore approaches for constructing RSs. Finally, it would also be interesting to extend this work to nondeterministic partial-complete FA.

References

[1] Jourdan G, Ural H, Yenigün H. Reduced checking sequences using unreliable reset. Information Processing Letters 2015; 115 (5): 532-535. doi:10.1016/j.ipl.2015.01.002

[2] Türker UC, Yenigün H. Complexities of some problems related to synchronizing, non-synchronizing and monotonic automata. International Journal of Foundations of Computer Science 2015; 26 (1): 99-122. doi:10.1142/S0129054115500057

[3] Ananichev DS, Volkov MV. Synchronizing monotonic automata. Theoretical Computer Science 2004; 327 (3): 225-239.

[4] Eppstein D. Reset sequences for monotonic automata. SIAM Journal of Computing 1990; 19 (3): 500-510.

[5] Natarajan BK. An Algorithmic Approach to the Automated Design of Parts Orienters. In: 27th Annual Symposium on Foundations of Computer Science; Toronto, Canada; 1986. pp. 132-142.

[6] Chow TS. Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 1978; 4 (3): 178-187.

[7] Boute R. Distinguishing sets for optimal state identification in checking experiments. IEEE Transactions on Computers 1974; 23 (8): 874-877. doi:10.1109/T-C.1974.224043

[8] Petrenko A, Bochmann G. Selecting test sequences for partially-specified nondeterministic finite state machines. In: International Workshop on Protocol Test Systems; London, UK; 1995. pp. 95-110.

[9] Hierons R. Minimizing the number of resets when testing from a finite state machine. Information Processing Letters 2004; 90 (6): 287-292.

[10] Hierons R, Ural H. Generating a checking sequence with a minimum number of reset transitions. Automated Software Engineering 2010; 17 (3): 217-250.

[11] Rezaki A, Ural H. Construction of checking sequences based on characterization sets. Computer Communications 1995; 18 (12): 911-920.

[12] Vasilevskii M. Failure diagnosis of automata. Cybernetics 1973; 9 (4): 653-665. doi:10.1007/BF01068590

[13] Gonenc G. A method for the design of fault detection experiments. IEEE Transactions on Computers 1970: 19 (6): 551-558. doi:10.1109/T-C.1970.222975

[14] Ural H, Wu X, Zhang F. On minimizing the lengths of checking sequences. IEEE Transactions on Computers 1997; 46 (1): 93-99.

[15] Hierons RM, Ural H. Reduced length checking sequences. IEEE Transactions on Computers 2002; 51 (9): 1111-1117.

[16] Benenson Y, Paz-Elizur T, Rivka A, Keinan E, Livneh Z et al. Programmable and autonomous computing machine made of biomolecules. Nature 2001; 414 (6862): 430-434. doi:10.1038/35106533

[17] Benenson Y, Adar R, Paz-Elizur ZLT, Shapiro E. DNA molecule provides a computing machine with both data and fuel. Proceedings of the National Academy of Sciences of the United States 2003; 100 (5): 2191-2196. doi:10.1073/pnas.0535624100

[18] Stojanovic MN, Stefanovic D. A deoxyribozyme-based molecular automaton. Nature Biotechnology 2003; 21 (9): 1069-1074.

[19] Cherubini A, Gawrychowski P, Kisielewicz A, Piochi B. A combinatorial approach to collapsing words. Mathematical Foundations of Computer Science 2006; 4162: 256-266.

[20] Gill A. Introduction to the Theory of Finite State Machines. New York, NY, USA: McGraw-Hill, 1962.

[21] Hennie FC. Finite-State Models For Logical Machines. New York, NY, USA: Wiley, 1968.

[22] Kohavi Z. Switching and Finite State Automata Theory. New York, NY, USA: McGraw-Hill, 1978.

[23] Roman A. Synchronizing finite automata with short reset words. Applied Mathematics and Computation 2009; 209 (1): 125-136.

[24] Roman A. Genetic algorithm for synchronization. In: Language and Automata Theory and Applications, Third International Conference; Tarragona, Spain; 2009. pp. 684-695.

[25] Roman A. New algorithms for finding short reset sequences in synchronizing automata. In: International Enformatika Conference; Prague, Czech Republic; 2005. pp. 13-17.

[26] Kudłacik R, Roman A, Wagner H. Effective synchronizing algorithms. Expert Systems and Applications 2012; 39 (14): 11746-11757.

[27] Kisielewicz A, Szykuła M. Generating small automata and the Černý conjecture. In: Implementation and Application of Automata; Halifax, Canada; 2013. pp. 340-348. doi:10.1007/978-3-642-39274-0_30

[28] Kisielewicz A, Kowalski J, Szykuła M. A fast algorithm finding the shortest reset words. In: Computing and Combinatorics; Hangzhou, China; 2013. pp. 182-196. doi:10.1007/978-3-642-38768-5_18

[29] Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. In: IEEE International Symposium on Parallel & Distributed Processing; Rome, Italy; 2009. pp. 1-10.

[30] Merrill D, Garland M, Grimshaw A. Scalable GPU graph traversal. ACM SIGPLAN Notices 2012; 47: 117-128.

[31] Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW et al. A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 2008; 68 (10): 1370-1380.

[32] Luo L, Wong M, Hwu WM. An effective GPU implementation of breadth-first search. In: Design Automation Conference; Anaheim, CA, USA; 2010. pp. 52-55.

[33] Harish P, Narayanan P. Accelerating large graph algorithms on the GPU using CUDA. In: High Performance Computing; Goa, India; 2007. pp. 197-208.

[34] Kirk DB, Wen-Mei WH. Programming Massively Parallel Processors: A Hands-On Approach. Oxford, UK: Newnes, 2012.

[35] Mytkowicz T, Musuvathi M, Schulte W. Data-parallel finite-state machines. ACM SIGPLAN Notices 2014; 49 (4): 529-542.

[36] Mytkowicz T, Schulte W. Maine: A Library for Data Parallel Finite Automata. Technical Report. New York, NY, USA: Microsoft Research, 2012.

[37] Hierons RM, Türker UC. Parallel algorithms for testing finite state machines: generating UIO sequences. IEEE Transactions on Computers 2016: 42 (11): 1077-1091.

[38] Hierons RM, Türker UC. Parallel algorithms for testing finite state machines: harmonised state identifiers and characterising sets. IEEE Transactions on Computers 2016; 65 (11): 3370-3383.

[39] Karahoda S, Erenay OT, Kaya K, Türker UC, Yenigün H. Parallelizing heuristics for generating synchronizing sequences. In: Testing Software and Systems; Graz, Austria; 2016. pp. 106-122.

[40] Klingbeil G, Erban R, Giles M, Maini P. Fat versus thin threading approach on GPUs: application to stochastic simulation of chemical reactions. IEEE Transactions on Parallel and Distributed Systems 2012; 23 (2): 280-287.

[41] Hierons RM, Türker UC. Distinguishing sequences for partially specified FSMs. In: NASA Formal Methods; Houston, TX, USA; 2014. pp. 62-76.

Appendices

Low-level algorithm details.

In developing the PRSGA we used several data structures to perform efficient memory transactions and efficient thread utilization.

- The $\mathcal{A}$ vector holds the transitions of the underlying FA: given state q_i and an input symbol x, $\mathcal{A}$ returns either a state q_j such that $\delta(q_i, x) = q_j$ or $ERROR$, indicating that the input is not defined. We also add $|\Sigma|$ elements to $\mathcal{A}$ for the $ERROR$ state, such that for all $x \in \Sigma$, $\delta(ERROR, x) = ERROR$. Hence, the size of $\mathcal{A}$ is $(n+1)|\Sigma|$.
- The *CurrentStates* vector corresponds to the elements of k-cluster $\mathcal{C}_k$, i.e. it holds the relationship between initial and current states.
  Hence, the size of *CurrentStates* is at most kn.
- The *InputSequence* vector corresponds to the input sequences associated with the states-vectors of k-cluster $\mathcal{C}_k$. That is, the *InputSequence* vector holds the input sequences that are going to be applied to the elements of states-vectors of k-cluster $\mathcal{C}_k$. Since there are k input sequences, and the current length value of input sequences (ℓ) varies between 1 and κ, the size of *InputSequence* is at most $k\kappa$.

The PRSGA algorithm uses three kernels: *Reset* (called on line 9 of Algorithm), *Apply* (called on line 11 of Algorithm), and *Test* (called on line 13 of Algorithm).

The *Reset* kernel is used to reset the *CurrentStates* vector. In order to do this the *Reset* kernel receives the *CurrentStates* vector, V_{min} vector, and the size of the V_{min} vector and the value of k. The *Reset* kernel is called with $k\eta$ threads. During the execution thread t_i (where $0 \leq i < k\eta$) copies the $i - floor(i/\eta)\eta$th value of the V_{min} vector to the ith value of the *CurrentStates* vector. Note that read/write operations carried out in the *Reset* kernel are coalesced and do not cause thread divergence.

The *Apply* kernel is used to evolve elements of the $\mathcal{C}_k$ vector. In other words, threads that execute the *Apply* kernel apply input sequences retrieved from the *InputSequence* vector to the current states retrieved from the *CurrentStates* vector. Therefore, the *Apply* kernel receives *CurrentStates*, *InputSequence*, number of input sequences k, and current length value κ. The *Apply* kernel is launched with $k\eta$ threads. The *Apply* kernel is an iterative procedure and the number of iterations is equal to κ.

At the jth iteration, thread t_i retrieves the value (state id) from $CurrentStates[i]$ and computes *input-index* value ($\Sigma(i, j, \eta, \kappa)$) and fetches the input symbol from $InputSequence[\Sigma(i, j, \eta, \kappa)]$. The input-index value is computed as follows:

$$\Sigma(i, j, \eta, \kappa) = floor(i/\eta)\kappa + j. \tag{1}$$

After the input and the current states have been retrieved, thread t_i retrieves the next state value from vector $\mathcal{A}$ and writes the next state value to $CurrentStates[i]$. Thread t_i repeats these steps as far as $j < \kappa$.

The algorithm evaluates the outcome of the *Apply* kernel through the *Test* kernel. The *Test* kernel is called within InnerLoop, which iterates over input sequences $w_1, w_2, \ldots, w_k$. At each call, the *Test* kernel receives *CurrentStates*, *index* (indicating the current index of input sequence (w_{index})). It also receives an integer vector (*bulk*) of n elements with all elements set to 0.

Note that the *CurrentStates* vector holds $k\eta$ elements (in other words, k states-vectors). For input sequence w_{index}, threads that execute the *Test* kernel should evaluate the *index*th states-vector, i.e. all the

elements in interval $[(\eta)(index-1)+1, ((\eta)(index-1)+\eta)-1)$ of the *CurrentStates* vector. To allow this we launch the $Test$ kernel with η threads. During the execution of the $Test$ kernel, thread i first reads the value (ς) that resides in the $(\eta)(index-1)+1+i$th element of the CurrentStates vector and then performs an (atomic) increment on the ςth element of the *bulk* vector. If $\varsigma = ERROR$, then the $Test$ kernel writes $(2n)$ to the first element of the *bulk* vector.

When the $Test$ kernel returns, we apply a generic *parallel-MAX* function and receive the maximum value on the *bulk* vector. Note that the maximum value of the *bulk* vector corresponds to the maximum number of states that are merged/collapsed by advancing the V_{min} vector with input sequence w_{index}. Hence, if the maximum value returned by the parallel-MAX function equals η then we append w_{index} to R and terminate; otherwise, if the maximum value is larger than $U(V_{min})$, then we copy w_{index} to w_{min} and parallel copy the $index$th states-vector of $\mathcal{C}_k$ to V_{min}. If the maximum value is equal to or larger than $2n$, we deduce that input sequence w_{index} is not defined for V_{min} and therefore the algorithm jumps to input sequence $w_{index+1}$ and continues.