HAL
open science

Parallelizing Heuristics for Generating Synchronizing Sequences

Sertaç Karahoda, Osman Tufan Erenay, Kamer Kaya, Uraz Cengiz Türker, Husnu Yenigun

To cite this version:

Sertaç Karahoda, Osman Tufan Erenay, Kamer Kaya, Uraz Cengiz Türker, Husnu Yenigun. Parallelizing Heuristics for Generating Synchronizing Sequences. 28th IFIP International Conference on Testing Software and Systems (ICTSS), Oct 2016, Graz, Austria. pp.106-122, 10.1007/978-3-319-47443-4_7 . hal-01643717

HAL Id: hal-01643717

https://inria.hal.science/hal-01643717

Submitted on 21 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Parallelizing Heuristics for Generating Synchronizing Sequences

Sertaç Karahoda[1], Osman Tufan Erenay[1], Kamer Kaya[1,2], Uraz Cengiz Türker[3], and Hüsnü Yenigün[1]

[1] Computer Science and Engineering, Faculty of Science and Engineering, Sabanci University, Tuzla, Istanbul, Turkey
{skarahoda,osmantufan,kaya,yenigun}@sabanciuniv.edu
[2] Dept. Biomedical Informatics, The Ohio State University, OH, USA
[3] Computer Engineering, Faculty of Engineering, Gebze Technical University, Gebze, Kocaeli, Turkey urazc@gtu.edu.tr

Abstract. Synchronizing sequences are used in the context of finite state machine based testing in order to initialize an implementation to a particular state. The cubic complexity of even the fastest heuristic algorithms known in the literature to construct a synchronizing sequence can be a problem in practice. In order to scale the performance of synchronizing heuristics, some algorithmic improvements together with a parallel implementation of these heuristics are proposed in this paper. An experimental study is also presented which shows that the improved/parallel implementation can yield a considerable speedup over the sequential implementation.

1 Introduction

Model Based Testing (MBT) uses formal models of system requirements to generate effective test cases. Most MBT techniques use state-based models, where the behaviour of the model is described in terms of states and state transitions. There has been much interest in testing from finite state machines (FSMs) (e.g., see [1–6]). Common to most FSM based testing methods is the need to bring the system under test (SUT) to a particular state. When there is a trusted *reset* input in the SUT, this is quite easy. However, sometimes such a reset input is not available, or even if it is available, it may be time consuming to apply the reset input. Therefore there are cases where the use of a reset input is not preferred [7–9].

A *synchronizing sequence*[4] for an FSM M is a sequence of inputs such that no matter at which state M currently is, if this sequence of inputs is applied, M is brought to a particular state. Therefore a synchronizing sequence is in fact a compound reset input, and can be used as such to simulate a reset input in the context of FSM based testing [10].

A synchronizing sequence may not exist for an FSM. However, as the size of the FSM gets larger, there almost always exists a synchronizing sequence [11].

[4] Synchronizing sequences are also known as *reset sequences*, or *reset words*.

For an FSM M with n states and alphabet size p, checking if M has a synchronizing sequence can be decided in time $O(pn^2)$ [12]. Since a synchronizing sequence will possibly be used many times in a test sequence, computing a shortest one for an FSM is of interest, but this problem is known to be NP-hard [12]. There exist a number of heuristics, called *synchronizing heuristics*, to compute short synchronizing sequences, such as Greedy [12] and Cycle [13] both with time complexity $O(n^3+pn^2)$, SynchroP and SynchroPL [14] with time complexity $O(n^5 + pn^2)$, and FastSynchro [15] with time complexity $O(pn^4)$. The upper bound for the length of the synchronizing sequence that will be produced by all of these heuristics is $O(n^3)$. Although synchronizing sequences are important for testing methods, the scalability of the synchronizing heuristics has not been addressed thoroughly. For practical applications, the use of even the fastest algorithms (Greedy and Cycle) with cubic complexity can be a problem.

In this work we investigate the use of modern multicore CPUs to scale the performance of synchronizing heuristics. We consider the Greedy algorithm to start with, as it is one of the two cheapest synchronizing heuristics (in practice as well [16]), known to produce shorter sequences than Cycle [16], and has been widely used as a baseline to evaluate the quality and speed of more advanced heuristics. To the best of our knowledge, this is the first work towards parallelization of synchronizing heuristics. Although, a parallel approach for constructing a synchronizing sequence for a *partial* machines is proposed in [17], the method proposed in [17] is not exact (in the sense that it may fail to find a synchronizing sequence even if one exists) and also it is not a polynomial time algorithm.

All synchronizing heuristics consist of a preprocessing phase, followed by synchronizing sequence generation phase. As presented in this paper, our initial experiments revealed that the preprocessing phase dominates the runtime of the overall algorithm for Greedy. Therefore for both parallelization and for algorithmic improvements of Greedy, we mainly focus on the first phase of the algorithm. With no parallelization, our algorithmic improvements alone yield a 20x speedup on Greedy for automata with 4000 states and 128 inputs. Furthermore, around 150x speedup has been obtained for the same class of automata, when the improved algorithm is executed in parallel with 16 threads.

The rest of the paper is organized as follows: In Section 2, the notation is given, and synchronizing sequences are formally defined. We give the details of Eppstein's Greedy construction algorithm in Section 3. The proposed improvements and the parallelization approach together with implementation details are described in Section 4. Section 5 presents the experimental results and Section 6 concludes the paper.

2 Preliminaries

FSMs are used to describe a reactive behaviour, i.e., when an input is applied to an FSM, it produces an output as a response. However, the output sequence produced by the application of a synchronizing sequence does not play a role. Therefore, in the context of synchronizing sequences, an FSM can simply be

considered as an automaton where the state transitions are only performed by the application of an input, and no output is produced.

In this work, we only consider complete deterministic automata. An *automaton* is defined by a triple $A = (S, \Sigma, \delta)$ where S is a finite set of n states, Σ is a finite set of p input symbols (or simply *inputs*) called the alphabet. $\delta : S \times \Sigma \to S$ is a transition function. If the automaton A is at a state s and if an input x is applied, then A moves to the state $\delta(s, x)$. Figure 1 shows an example automaton A with 4 states and 2 inputs.

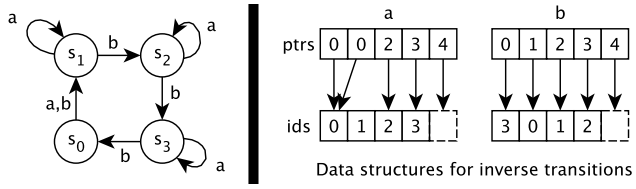


Fig. 1. A synchronizable automaton A (left), and the data structures we used to store and process the transition function δ^{-1} in memory (see Section 4.4 for the details). A synchronizing sequence for A is *abbbabbba*.

An element of the set $\Sigma^\star$ is called an *input sequence*. We use $|w|$ to denote the length of w, and ε is the empty input sequence. We extend the transition function δ to a set of states and to an input sequence in the usual way. We have $\delta(s, \varepsilon) = s$, and for an input sequence $w \in \Sigma^\star$ and an input symbol $x \in \Sigma$, we have $\delta(s, xw) = \delta(\delta(s, x), w)$. For a set of states $S' \subseteq S$, we have $\delta(S', w) = \{\delta(s, w) | s \in S'\}$.

We use the notation $\delta^{-1}(s, x)$ to denote the set of those states with a transition to state s with input x. Formally, $\delta^{-1}(s, x) = \{s' \in S | \delta(s', x) = s\}$.

Let $A = (S, \Sigma, \delta)$ be an automaton, and $w \in \Sigma^\star$ be an input sequence. w is said to be a *merging sequence for a set of states* $S' \subseteq S$ if $|\delta(S', w)| = 1$, and S' is called *mergable*. Any set $\{s\}$ with a single state is mergable, since ε is a merging sequence for $\{s\}$. w is called a *synchronizing sequence for* A if $|\delta(S, w)| = 1$. A is called *synchronizable* if there exists a synchronizing sequence for A. For example, the automaton given in Figure 1 is synchronizable, since *abbbabbba* is a synchronizing sequence for the automaton. Deciding if an automaton is synchronizable or not can be performed in polynomial time based on the following result.

Proposition 1 ([12, 18]). *An automaton $A = (S, \Sigma, \delta)$ is synchronizable iff for all $s_i, s_j \in S$, there exists a merging sequence for $\{s_i, s_j\}$.*

For a set of states $C \subseteq S$, let $C^{\langle 2 \rangle} = \{\langle s_i, s_j \rangle | s_i, s_j \in C\}$ be the set of all *multisets* with cardinality 2 with elements from C, i.e. $C^{\langle 2 \rangle}$ is the set of all subsets of C with cardinality 2, where repetition is allowed. An element $\langle s_i, s_j \rangle \in C^{\langle 2 \rangle}$ is called *a singleton* if $s_i = s_j$, otherwise it is called *a pair*.

As Proposition 1 makes it explicit, checking the existence of merging sequences for pairs of states is needed to decide if an automaton is synchronizable.

In addition, the heuristic algorithms also make use of the merging sequences for pairs. For both checking the existence of merging sequences and finding a merging sequence (in fact for finding a shortest merging sequence) for pairs of states of an automaton, one can use the notion of the pair automaton, which we define next.

Definition 1. *For an automaton* $A = (S, \Sigma, \delta)$*,* the pair automaton $\mathcal{A}$ of A *is defined as* $\mathcal{A} = (S^{\langle 2 \rangle}, \Sigma, \Delta)$*, where for a state* $\langle s_i, s_j \rangle \in S^{\langle 2 \rangle}$ *and an input symbol* $x \in \Sigma$*,* $\Delta(\langle s_i, s_j \rangle, x) = \langle \delta(s_i, x), \delta(s_j, x) \rangle$*.*

3 Eppstein's Algorithm

In this section, we explain Eppstein's Greedy algorithm, and we present an observation on the timing profile of the algorithm. This observation guided our work on the improvements and parallelization of the algorithm, which will be explained in Section 4. Greedy (and also all other synchronizing heuristics mentioned in Section 1) has two phases. In the first phase, a shortest merging sequence for each mergable pair of states is found. If all pairs are mergable, these merging sequences are used to construct a synchronizing sequence in the second phase.

For a pair of states s_i, s_j of an automaton $A = (S, \Sigma, \delta)$, checking the existence of a merging sequence for $\{s_i, s_j\}$, and computing a shortest merging sequence for $\{s_i, s_j\}$ can be performed in time $O(pn^2)$ by finding a shortest path from the state $\langle s_i, s_j \rangle$ of the pair automaton $\mathcal{A}$ to a singleton state in $\mathcal{A}$ using Breadth First Search (BFS). Since we will have to check the existence and find merging sequences for all pairs of states, one can instead use a *backward* BFS, seeded at singleton states of the pair automaton, as explained below.

For an automaton $A = (S, \Sigma, \delta)$, a function $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$, is called *a pairwise merging function (PMF) for* A, if for all $\langle s_i, s_j \rangle \in S^{\langle 2 \rangle}$, $\tau(\langle s_i, s_j \rangle)$ is a shortest merging sequence for $\{s_i, s_j\}$ if $\{s_i, s_j\}$ is mergable, and $\tau(\langle s_i, s_j \rangle)$ is undefined if $\{s_i, s_j\}$ is not mergable. Note that PMF for an automaton A is not unique, and it is a total function iff A is synchronizable. Algorithm 1 computes such a PMF τ for a given automaton A, where initially $\tau(\langle s, s \rangle) = \varepsilon$ for the singleton states in $S^{\langle 2 \rangle}$ (line 1), and $\tau(\langle s_i, s_j \rangle)$ is considered to be "undefined" for pair states in $S^{\langle 2 \rangle}$ (line 2). The algorithm iteratively computes the values of $\tau(.)$ as it discovers shortest merging sequences for more pairs in $S^{\langle 2 \rangle}$.

Algorithm 1 keeps track of a *frontier* set F which is initialized to all singleton states at line 3. Throughout the algorithm, R represents the remaining set of pairs with $\tau(\langle s_i, s_j \rangle)$ still being undefined. In each iteration of the algorithm (lines 5–6), a BFS step is performed by using BFS_step_F2R given in Algorithm 2. BFS_step_F2R constructs the next frontier F' from the current frontier F, by considering each $\langle s_i, s_j \rangle \in F$ (line 2). Lines 4-5 of BFS_step_F2R identify a pair $\langle s'_i, s'_j \rangle \in R$ such that $s'_i = \delta(s_i, x)$ and $s'_j = \delta(s_j, x)$ for some $x \in \Sigma$, and lines 6-7 performs the necessary updates. Since this algorithm considers, in a sense, the *reverse* transitions of $\langle s_i, s_j \rangle$ in the frontier F to reach to pairs $\langle s'_i, s'_j \rangle$ in R, we call it as "Frontier to Remaining (F2R)" BFS step.

Algorithm 1 eventually assigns a value to $\tau(\langle s_i, s_j \rangle)$ if $\{s_i, s_j\}$ is mergable. Based on Proposition 1, A is synchronizable iff there does not exist a pair state

Algorithm 1: Computing a PMF $\tau : S^{\langle 2\rangle} \to \Sigma^\star$ (F2R based)

input : An automaton $A = (S, \Sigma, \delta)$
output: A PMF $\tau : S^{\langle 2\rangle} \to \Sigma^\star$
foreach *singleton* $\langle s, s\rangle \in S^{\langle 2\rangle}$ **do** $\tau(\langle s, s\rangle) = \varepsilon$;
foreach *pair* $\langle s_i, s_j\rangle \in S^{\langle 2\rangle}$ **do** $\tau(\langle s_i, s_j\rangle) =$ *undefined*;
$F \longleftarrow \{\langle s, s\rangle | s \in S\}$; `// all singleton states of` $\mathcal{A}$
$R \longleftarrow \{\langle s_i, s_j\rangle | s_i, s_j \in S \wedge s_i \neq s_j\}$; `// all pair states of` $\mathcal{A}$
while F *is not empty* **do**
  $F, R, \tau \longleftarrow$ BFS_step_F2R(A, F, R, τ);

Algorithm 2: BFS_step_F2R

input : An automaton $A = (S, \Sigma, \delta)$, the frontier F, the remaining set R, τ
output: The new frontier F', the new remaining set R', and updated function τ
$F' \longleftarrow \emptyset$;
foreach $\langle s_i, s_j\rangle \in F$ **do**
  foreach $x \in \Sigma$ **do**
    foreach $\langle s'_i, s'_j\rangle$ *such that* $s'_i \in \delta^{-1}(s_i, x)$ *and* $s'_j \in \delta^{-1}(s_j, x)$ **do**
      if $\tau(\langle s'_i, s'_j\rangle)$ *is undefined* **then** // $\langle s'_i, s'_j\rangle \in R$
        $\tau(\langle s'_i, s'_j\rangle) \longleftarrow x\tau(\langle s_i, s_j\rangle)$;
        $F' = F' \cup \{\langle s'_i, s'_j\rangle\}$;
let R' be $R \setminus F'$;

$\langle s_i, s_j\rangle$ with $\tau(\langle s_i, s_j\rangle)$ being undefined when Algorithm 1 terminates. We can now present Eppstein's Greedy algorithm based on Algorithm 1.

The Greedy algorithm keeps track of a current set C of states yet to be merged, initialized to S at line 4. A pair $\langle s_i, s_j\rangle \in C^{\langle 2\rangle}$ is called an *active pair*. In each iteration of the while loop at line 7, an active pair $\langle s_i, s_j\rangle \in C^{\langle 2\rangle}$ is found such that it has a shortest merging sequence among all active pairs in C

Algorithm 3: Eppstein's Greedy Algorithm

input : An automaton $A = (S, \Sigma, \delta)$
output: A synchronizing sequence Γ for A (or fail if A is not synchronizable)
compute a PMF τ using Algorithm 1;
if *there exists a pair* $\langle s_i, s_j\rangle$ *such that* $\tau(\langle s_i, s_j\rangle)$ *is undefined* **then**
  report that A is not synchronizable and exit;
foreach $s_i, s_j, s_k \in S$ **do** compute $\delta(s_k, \tau(\langle s_i, s_j\rangle))$;
$C = S$; `//` C `will keep track of the current set of states`
$\Gamma = \varepsilon$; `//` Γ `is the synchronizing sequence to be constructed`
while $|C| > 1$ **do** `// we have two or more states yet to be merged`
  find a pair $\langle s_i, s_j\rangle \in C^{\langle 2\rangle}$ with minimum $|\tau(\langle s_i, s_j\rangle)|$ among all pairs in $C^{\langle 2\rangle}$;
  $\Gamma = \Gamma\, \tau(\langle s_i, s_j\rangle)$;
  $C = \delta(C, \tau(\langle s_i, s_j\rangle))$;

	$n = 1000$			$n = 2000$			$n = 4000$		
p	t_{ALL}	t_{PMF}	$\frac{t_{PMF}}{t_{ALL}}$	t_{ALL}	t_{PMF}	$\frac{t_{PMF}}{t_{ALL}}$	t_{ALL}	t_{PMF}	$\frac{t_{PMF}}{t_{ALL}}$
2	0,045	0,042	0,928	0,188	0,175	0,929	1,214	1,158	0,954
8	0,125	0,122	0,974	0,526	0,513	0,975	2,757	2,698	0,979
32	0,483	0,480	0,993	2,151	2,138	0,994	9,980	9,919	0,994
128	2,202	2,199	0,999	9,243	9,229	0,999	39,810	39,749	0,998

Table 1. Sequential PMF construction time (t_{PMF}), and overall time (t_{ALL}) for automata with $n \in \{1000, 2000, 4000\}$ states and $p \in \{2, 8, 32, 128\}$ inputs.

(line 8). The synchronizing sequence (initialized to the empty sequence at line 6) is extended with $\tau(\langle s_i, s_j \rangle)$ at line 9. Finally, $\tau(\langle s_i, s_j \rangle)$ is applied to C to update the current set of states. When $|C| = 1$, this means that Γ accumulated at that point is a synchronizing sequence.

The following results are shown in [12, Theorem 5]. For an automaton A with n states and p inputs, Phase 1 of Greedy (lines 1–3) can be implemented to run in time $O(pn^2)$ and Phase 2 of Greedy (lines 4–10) can be implemented to run in time $O(n^3)$. Hence the overall time for Greedy is $O(n^3 + pn^2)$.

We performed an experimental analysis to see how much Phase 1 (which we will call as the PMF construction phase[5]) and Phase 2 (the synchronizing sequence construction phase) of the algorithm contribute to the running time in practice for a sequential implementation. Based on these experiments, we observed that PMF construction actually dominates the running time of the algorithm (see Table 1). Hence, in order to improve the performance of Greedy, we developed approaches for parallel implementation of PMF construction, together with some algorithmic modifications, which we explain in Section 4.

4 Parallelization Approach and Improvements

Algorithm 1 necessarily performs a BFS on the pair automaton $\mathcal{A}$, and a BFS forest rooted at singleton states of $\mathcal{A}$ is implicitly obtained. At the roots of the forest (i.e. in the first frontier set F) we have singleton states of $\mathcal{A}$, which corresponds to the nodes at level 0 of the BFS forest. At each iteration of the algorithm, the current frontier F has all the nodes at level k in the BFS forest. These nodes are processed by Algorithm 2 to compute the next frontier F' which are the nodes at level $k+1$ in the BFS forest. The processing of the state pairs in F are the tasks to be performed at the current level. To process a state pair, Algorithm 2 considers incoming transitions of the pair (i.e., inverse transitions) based on the δ^{-1} function (line 4). Hence, the cost of each task can be different. Furthermore, the total number of edges of the tasks in F, i.e., frontier edges, determines the cost of the corresponding level's BFS_step_F2R execution and this also varies for each level. We used OpenMP for parallel implementation and employed the dynamic scheduling policy (with batches of 512-pairs) since the task costs are not uniform.

[5] Lines 2-3 of Phase 1 is easily handled as a part of PMF construction by checking if R is empty or not at the end of PMF construction.

4.1 Computing a PMF in parallel

When Algorithm 1 is implemented sequentially, handling two consecutive iterations is seamless: using a single queue to `enque` and `deque` the frontier pairs suffices to process them in the correct order (i.e. a pair at level $k+1$ is only found after all level k pairs are found). However, with multiple threads, a barrier (a global synchronization technique) is required after each iteration. Otherwise, a pair from the next frontier can be processed before another pair in the current frontier and an incorrect PMF function τ can be computed. Here we present Algorithm 1 iteratively, and isolate the BFS_step_F2R from the main flow of the algorithm since it will be our main target for efficiency.

Algorithm 4: BFS_step_F2R (in parallel)

input : An automaton $A = (S, \Sigma, \delta)$, the frontier F, the remaining set R, τ
output: The new frontier F', the new remaining set R', and updated function τ

1 **foreach** *thread* t **do** $F'_t \longleftarrow \emptyset$;
2 **foreach** $\langle s_i, s_j \rangle \in F$ **in parallel do**
3 **foreach** $x \in \Sigma$ **do**
4 **foreach** $\langle s'_i, s'_j \rangle$ *where* $s'_i \in \delta^{-1}(s_i, x)$ *and* $s'_j \in \delta^{-1}(s_j, x)$ **do**
5 **if** $\tau(\langle s'_i, s'_j \rangle)$ *is undefined* **then //** $\langle s'_i, s'_j \rangle \in R$
6 $\tau(\langle s'_i, s'_j \rangle) \longleftarrow x\tau(\langle s_i, s_j \rangle)$;
7 $F'_t = F'_t \cup \{\langle s'_i, s'_j \rangle\}$;
8 $F' \longleftarrow \emptyset$;
9 **foreach** *thread* t **do** $F' = F' \cup F'_t$;
10 let R' be $R \setminus F'$;

To parallelize BFS_step_F2R, we partition the current frontier F among multiple threads where only a single thread processes a frontier pair as shown in Algorithm 4 (line 2). Since there is no task-dependency among the pairs, all the threads can simultaneously work. However, a race condition occurs since the next frontier set F' is a shared object in the sequential implementation. To break dependency with a lock-free approach, in our parallel implementation, each thread t uses a local frontier array F'_t and when a new pair from the next frontier is found by thread t, it is immediately added to F'_t. When two threads find the same pair $\langle s'_i, s'_j \rangle$ at the same time, both threads insert it to their local frontiers (lines 5–7). Hence, when the local frontiers are combined at the end of each iteration (lines 8–9), the same pair can occur multiple times if no duplicate pair check is applied. In our preliminary experiments, we observed that at most one in a thousands extra pairs are inserted to F' when they are allowed. Hence, we let the threads process them since the total extra pair cost is negligible compared to the cost of checking and resolving duplicates.

4.2 Another approach for BFS steps

Algorithm 2 and Algorithhm 4 follow a natural and possibly the most common technique to construct the next frontier set F' from the current frontier set F by considering the incoming transitions. Another approach to construct the next frontier F' function, which we call "Remaining to Frontier (R2F)", is processing the remaining state pairs' edges instead of those in the frontier. As mentioned above, a state pair $\langle s_i, s_j \rangle$ stays in R, i.e., in the remaining pair set, as long as $\tau(\langle s_i, s_j \rangle)$ stays undefined. In the parallel R2F approach described by Algorithm 5, the threads process the transitions of the remaining state pairs instead of the ones in the frontier. Hence, instead of δ^{-1}, the original transition function δ is used and the pair found is checked to be in the frontier (lines 5–6). If a pair $\langle s_i, s_j \rangle$ has a transition to a pair $\langle s'_i, s'_j \rangle \in F$ (i.e., if $\langle s_i, s_j \rangle$ is in the next frontier), $\tau(\langle s_i, s_j \rangle)$ is set and the process ends (lines 7–9). Otherwise, $\langle s_i, s_j \rangle$ is kept in the remaining set (lines 10–11). Similar to parallel F2R, we use a local remaining pair array R'_t for each thread t in the lock-free parallelization of R2F.

Algorithm 5: BFS_step_R2F (in parallel)

```
   input  : An automaton A = (S, Σ, δ), the frontier F, the remaining set R, τ
   output: The new frontier F', the new remaining set R', and updated function τ
 1 foreach thread t do R'_t ⟵ ∅;
 2 foreach ⟨s_i, s_j⟩ ∈ R in parallel do
 3     connected ⟵ false;
 4     foreach x ∈ Σ do
 5         ⟨s'_i, s'_j⟩ ⟵ ⟨δ(s_i, x), δ(s_j, x)⟩;
 6         if τ(⟨s'_i, s'_j⟩) is defined then //  ⟨s'_i, s'_j⟩ ∈ F
 7             τ(⟨s_i, s_j⟩) ⟵ xτ(⟨s'_i, s'_j⟩);
 8             connected ⟵ true;
 9             break;
10     if not connected then
11         R'_t = R'_t ∪ {⟨s_i, s_j⟩};
12 R' ⟵ ∅;
13 foreach thread t do  R' = R' ∪ R'_t ;
14 let F' be R \ R';
```

4.3 A hybrid approach to construct the next frontier

Since the size of R decreases at each iteration, R2F becomes faster at each step. On the other hand, F2R is expected to be faster than R2F during the earlier iterations. Therefore it makes sense to use a hybrid approach, where either an F2R or an R2F BFS step is used depending on their respective cost for the current iteration. These observations have been used by Beamer et al. to implement a direction-optimized BFS [19]. Since the cost of each F2R/R2F

Algorithm 6: Computing a function $\tau : S^{\langle 2 \rangle} \rightarrow \Sigma^{\star}$ (Hybrid)

input : An automaton $A = (S, \Sigma, \delta)$
output: A function $\tau : S^{\langle 2 \rangle} \rightarrow \Sigma^{\star}$
1 **foreach** *singleton* $\langle s, s \rangle \in S^{\langle 2 \rangle}$ **do** $\tau(\langle s, s \rangle) = \varepsilon$;
2 **foreach** *pair* $\langle s_i, s_j \rangle \in S^{\langle 2 \rangle}$ **do** $\tau(\langle s_i, s_j \rangle) =$ *undefined*;
3 $F \longleftarrow \{\langle s, s \rangle | s \in S\}$; `// all singleton states of` $\mathcal{A}$
4 $R \longleftarrow \{\langle s_i, s_j \rangle | s_i, s_j \in S \wedge s_i \neq s_j\}$; `// all pair states of` $\mathcal{A}$
5 **while** F *is not empty* **do**
6 **if** $|F| < |R|$ **then**
7 $F, R, \tau \longleftarrow$ BFS_step_F2R(A, F, R, τ);
8 **else**
9 $F, R, \tau \longleftarrow$ BFS_step_R2F(A, F, R, τ);

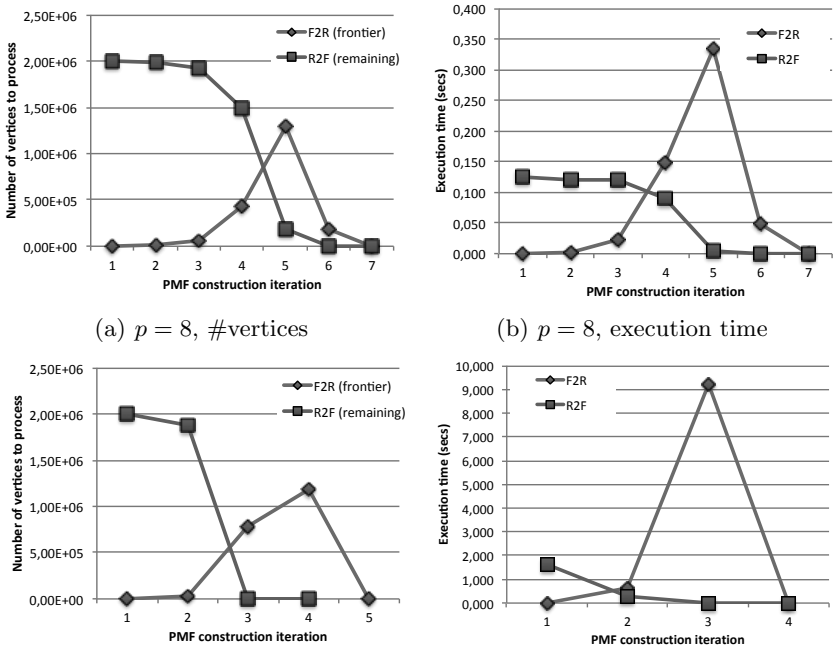
iteration depends on the number of edges processed, it is reasonable to compare the number of frontier/remaining pairs' edges to choose the cheaper approach at each iteration as in [19]. When the BFS is executed on a simple graph, this strategy is easy to apply. However, by only using δ^{-1}, it takes $O(p)$ time to count a new frontier pair's edges. Overall, the counting process takes $O(pn^2)$ time which is expensive considering that the overall sequential complexity is also $O(pn^2)$. In this work, we compared the size of R and F instead of the edges to be processed. The total additional complexity due to counting is $O(n^2)$ since each pair will be counted only once.

To analyze the validity of our counting heuristic and the potential improvement due to the Hybrid approach described in Algorithm 6, we compared the size of R and F, and the corresponding execution time of each F2R/R2F execution in Figure 2. As the figure shows, counting the pairs instead of transitions can be a good heuristic to guess the cheaper approach in our case. Furthermore, the performance difference of F2R and R2F at the each iteration shows that the proposed *Hybrid* approach can yield a much better performance.

4.4 Implementation details

To store and utilize the $\delta^{-1}(s, x)$ for all $x \in \Sigma$ and $s \in S$, we employ the data structures in Fig. 1 (right). For each symbol $x \in \Sigma$, we used two arrays $\texttt{ptrs}_{\texttt{x}}$ and $\texttt{js}_{\texttt{x}}$ where the former is of size $n+1$ and the latter is of size n. For each state $s \in S$, $\texttt{ptrs}_{\texttt{x}}[s]$ and $\texttt{ptrs}_{\texttt{x}}[s+1]$ are the start (inclusive) and end (exclusive) pointers to two $\texttt{js}_{\texttt{x}}$ entries. The array $\texttt{js}_{\texttt{x}}$ stores the ids of the states $\delta^{-1}(s, x)$ in between $\texttt{js}_{\texttt{x}}[\texttt{ptrs}_{\texttt{x}}[s]]$ and $\texttt{js}_{\texttt{x}}[\texttt{ptrs}_{\texttt{x}}[s+1]$ - 1]. This representation has a low memory footprint. Furthermore, we access the entries in the order of their array placement in our implementation hence, it is also good for spatial locality.

The memory complexity of the algorithms investigated in this study is $O(n^2)$. For each pair of states, we need to employ an array to store the length of the shortest merging sequence. To do that one can allocate an array of size n^2, Fig. 3 (left), and given the array index $\ell = (i-1) \times n + j$ for a state pair



(a) $p = 8$, #vertices (b) $p = 8$, execution time

(c) $p = 128$, #vertices (d) $p = 128$, execution time

Fig. 2. The number of frontier and remaining vertices at each BFS level and the corresponding execution times of F2R and R2F while constructing the PMF τ for $n = 2000$ and $p = 8$ (top) and $p = 128$ (bottom).

$\{s_i, s_j\}$ where $1 \leq i \leq j \leq n$, she can obtain the state ids by $i = \lceil \frac{\ell}{n} \rceil$ and $j = \ell - ((i-1) \times n)$. This simple approach effectively uses only the half of the array since for a state pair $\{s_i, s_j\}$, a redundant entry for $\{s_j, s_i\}$ is also stored. In our implementation, Fig. 3 (right), we do not use redundant locations. For an index $\ell = \frac{i \times (i+1)}{2} + j$ the state ids can be obtained by $i = \lfloor \sqrt{1 + 2\ell} - 0.5 \rfloor$ and $j = \ell - \frac{i \times (i+1)}{2}$. Preliminary experiments show that this approach, which does not suffer from the redundancy, also have a positive impact on the execution time. That being said, all the algorithms in the paper uses it and this improvement will not have change their relative performance.

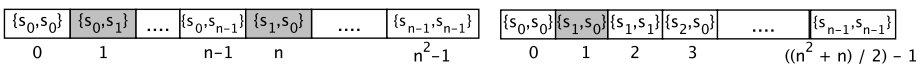


Fig. 3. Indexing and placement of the state pair arrays. A simple placement of the pairs (on the left) uses redundant places for state pairs $\{s_i, s_j\}$, $i \neq j$, e.g., $\{s_1, s_2\}$ and $\{s_2, s_1\}$ in the figure. On the right, the indexing mechanism we used is shown.

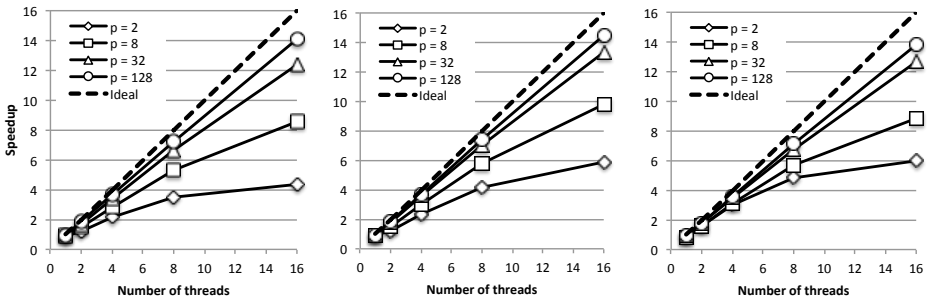
5 Experimental Results

All the experiments in the paper are performed on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). For the multicore implementations, we used OpenMP and all the codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled.

To measure the efficiency of the proposed algorithms, we used randomly generated automatons[6] with $n \in \{1000, 2000, 4000\}$ states and $p \in \{2, 8, 32, 128\}$ inputs. For each (n, p) pair, we randomly generated 20 different automatons and executed each algorithm on these automatons. The values in the figures and the tables are the averages of these 20 executions for each configuration, i.e., algorithm, n and p.

5.1 Multicore parallelization of PMF construction

Figure 4 shows the speedups of our parallel F2R implementation over the sequential baseline (that has no parallelism). Since F2R uses the same frontier extension mechanism with the sequential baseline, and R2F employs a completely different one, here we only present the speedup values of F2R. As the figure shows, when p is large, the parallel F2R presents good speedups, e.g., for $p = 128$, the average speedup is 14.1 with 16 threads. Furthermore, when compared to the single-thread F2R, the average speedup is 15.2 with 16 threads. A performance difference between sequential baseline and single-threaded F2R exists because of the parallelization overhead during the local queue management. Overall, we observed 10% parallelization penalty for F2R on the average over the sequential baseline for all (n, p) pairs.

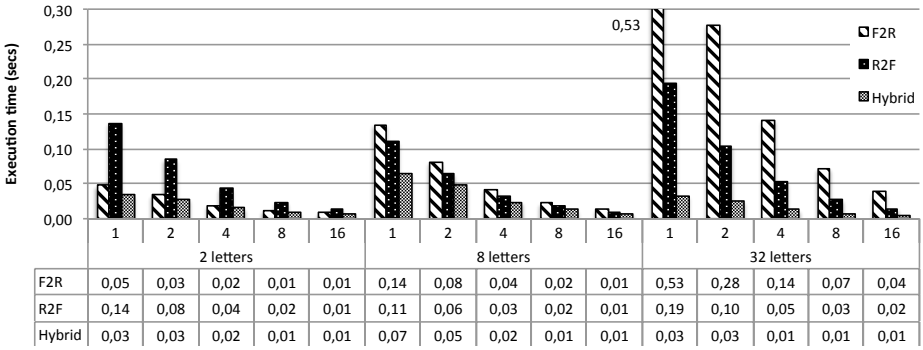


(a) $n = 1000$ (b) $n = 2000$ (c) $n = 4000$

Fig. 4. The speedup of our parallel F2R PMF construction over the sequential PMF construction baseline.

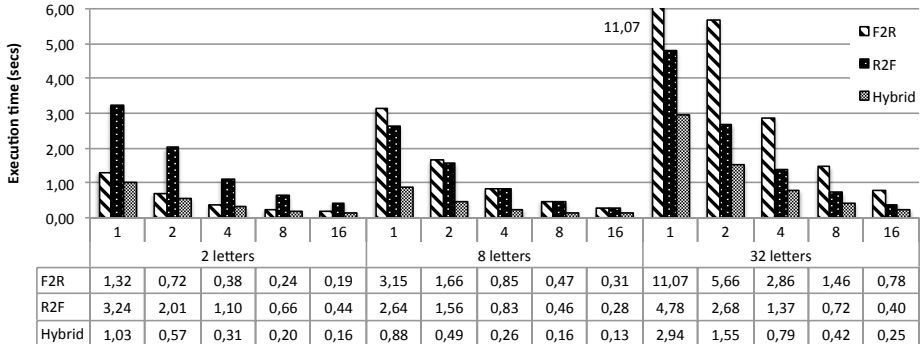
For p values smaller than 128, i.e., 2, 8, and 32, the average speedups are 5.4, 9.1, and 12.8, respectively, with 16 threads. The impact of the parallelization

[6] For each state s and input x, $\delta(s, x)$ is randomly assigned to a state $s' \in S$.



	2 letters					8 letters					32 letters				
	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
F2R	0,05	0,03	0,02	0,01	0,01	0,14	0,08	0,04	0,02	0,01	0,53	0,28	0,14	0,07	0,04
R2F	0,14	0,08	0,04	0,02	0,01	0,11	0,06	0,03	0,02	0,01	0,19	0,10	0,05	0,03	0,02
Hybrid	0,03	0,03	0,02	0,01	0,01	0,07	0,05	0,02	0,01	0,01	0,03	0,03	0,01	0,01	0,01

(a) $n = 1000$



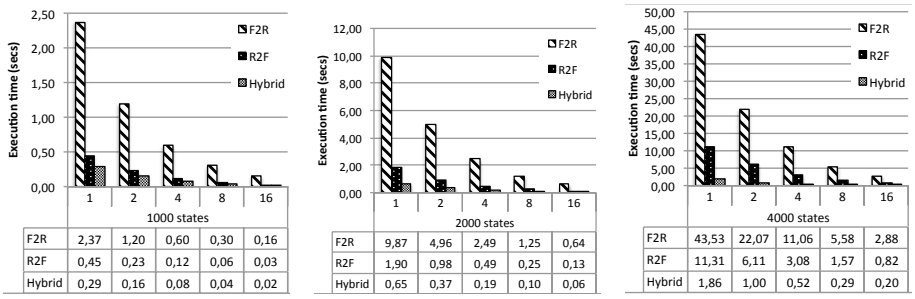
	2 letters					8 letters					32 letters				
	1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
F2R	1,32	0,72	0,38	0,24	0,19	3,15	1,66	0,85	0,47	0,31	11,07	5,66	2,86	1,46	0,78
R2F	3,24	2,01	1,10	0,66	0,44	2,64	1,56	0,83	0,46	0,28	4,78	2,68	1,37	0,72	0,40
Hybrid	1,03	0,57	0,31	0,20	0,16	0,88	0,49	0,26	0,16	0,13	2,94	1,55	0,79	0,42	0,25

(b) $n = 4000$

Fig. 5. Comparison of the parallel execution times of the three PMF construction algorithms: (1) F2R, (2) R2F, and (3) hybrid. The figures show the times for $n = 1000$ (top) and $n = 4000$ (bottom), $p \in \{2, 8, 32\}$, with $\{1, 2, 4, 8, 16\}$ threads (x-axis). For a better readability and figure scaling, the single-thread F2R bars with 32 inputs are allowed to exceed the max value on the y-axis.

overhead is more for such cases since the amount of the local-queue overhead is proportional to the number of states but not to the number of edges. Consequently, when p decreases the amount of total work decreases and hence, the impact of the overhead increases. Furthermore, since the number of iterations for PMF construction increases with decreasing p, the local queues are merged more for smaller p values. Therefore, one can expect more overhead, and hence, less efficiency for smaller p values as the experiments confirm.

Figure 5 compares the execution times of F2R, R2F and Hybrid algorithm for $n = 1000$ (top) and $n = 4000$ (bottom) states, $p \in \{2, 8, 32\}$ and $\{1, 2, 4, 8, 16\}$ threads (the results for $n = 2000$ are similar but omitted due to space limitations). For better figure scaling, the results for $p = 128$ is given in Figure 6. An interesting observation is that F2R is consistently faster than R2F for $p = 2$, however, it is slower otherwise. This can be explained by the difference in the number of required iterations to construct PMF: when p is large, the frontier expands very quickly and the PMF is constructed in less iterations, e.g., for $n = 2000$, the



	1	2	4	8	16
F2R	2,37	1,20	0,60	0,30	0,16
R2F	0,45	0,23	0,12	0,06	0,03
Hybrid	0,29	0,16	0,08	0,04	0,02

(a) $n = 1000$

	1	2	4	8	16
F2R	9,87	4,96	2,49	1,25	0,64
R2F	1,90	0,98	0,49	0,25	0,13
Hybrid	0,65	0,37	0,19	0,10	0,06

(b) $n = 2000$

	1	2	4	8	16
F2R	43,53	22,07	11,06	5,58	2,88
R2F	11,31	6,11	3,08	1,57	0,82
Hybrid	1,86	1,00	0,52	0,29	0,20

(c) $n = 4000$

Fig. 6. Comparison of the parallel execution times of the three PMF construction algorithms: (1) F2R, (2) R2F, and (3) hybrid. The figures show the times for $n = 1000$ (left), $n = 2000$ (middle), and $n = 4000$ (bottom), $p = 128$, with $\{1, 2, 4, 8, 16\}$ threads (x-axis).

PMF is generated in 16 iterations for $p = 2$, whereas only 7 iterations are required for $p = 8$. Since each edge will be processed once, the runtime of F2R always increases with p, i.e., with the number of edges. However, since the frontier expands much faster, the total number of remaining (R-)pairs processed by the R2F throughout the process will probably decrease. Furthermore, since when the frontier is large, while traversing the edge list of an R-pair, it is more probable to early terminate the traversal and add the R-pair to the next frontier earlier. Surprisingly, when p increases, these may yield a decrease in the R2F runtime (observe the change from $p = 2$ to $p = 8$ in Fig. 5). However, once the performance benefits of early termination are fully exploited, an increase on the R2F runtime with increasing p is more probable since the overall BFS work, i.e., the total number of edges, also increases with p (observe the change from $p = 8$ to $p = 32$ in Fig. 5).

Observing such performance differences for R2F and F2R on automatons with different characteristics, the potential benefit of a Hybrid algorithm in practice is more clear. As Figure 5 and Figure 6 show, the hybrid approach, which is just a combination of F2R and R2F, is almost always faster than employing a pure F2R or a pure R2F BFS-level expansion. Furthermore, we do not need parallelism to observe these performance benefits: the Hybrid approach works better even when a single thread is used at runtime. For example, when $n = 4000$ and $p = 128$, the Hybrid algorithm is 23 and 6 times faster than F2R and R2F, respectively. For the same automaton set, the speedups due to hybridization of the process become 14 and 4 with 16 threads on average.

When the Hybrid algorithm is used, the speedups on the PMF generation phase are given in Figure 7. As the figure shows, thanks to parallelism and good scaling of Hybrid (for large p values), the speedups increase when the number of threads increases. The PMF generation process becomes 95, 165, and 199 times faster when 16 threads used for 1000, 2000, and 4000 state automatons, respectively. Even with single thread, i.e., no parallelization, the Hybrid heuristic is 8, 14, and 21 times faster than the sequential algorithm.

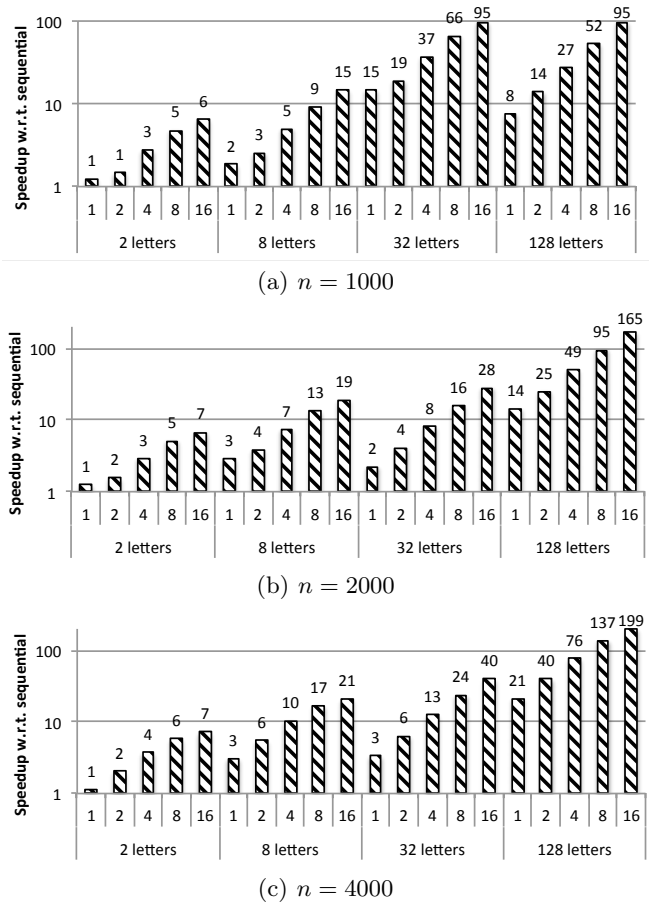


Fig. 7. The speedups of the Hybrid PMF construction algorithm with $n = 1000$ (top), 2000 (middle), 4000 (bottom) and $p \in \{2, 8, 32, 128\}$. The x-axis shows the number of threads used for the Hybrid execution. The values are computed based on the average sequential PMF construction time over 20 different automatons for each (n, p) pair.

Since we generate the PMF to find a synchronizing sequence, a more practical evaluation metric would be the performance improvement over the sequential reset sequence construction process. As Table 1 shows, for Eppstein's Greedy heuristic (also for some other heuristics such as Cycle [13]), the PMF generation phase dominates the overall runtime. For this reason, we simply conducted an experiment where the Hybrid approach is used to construct the PMF and no further parallelization is applied during the synchronizing sequence construction phase. Table 2 shows the speedups for this experiment for single thread and 16 thread Hybrid executions. As the results show, even when the sequence construction phase is not parallelized, more than 50x and more than 100x improvement is possible for $p = 32$ and $p = 128$, respectively.

	p (single thread)				p (16 threads)			
n	2	8	32	128	2	8	32	128
1000	1,2	1,8	13,4	7,5	4,6	10,8	58,2	83,7
2000	1,2	2,7	2,2	14,0	4,8	13,1	24,3	133,9
4000	1,1	2,9	3,3	20,7	5,5	14,8	31,7	154,0

Table 2. The speedups obtained on Eppstein's Greedy algorithm when the Hybrid PMF construction algorithm is used.

As noted before, F2R based PMF construction has $O(pn^2)$ time complexity. R2F based PMF construction, on the other hand, has $O(dpn^2)$ time complexity (where d is the diameter of the pair automaton $\mathcal{A}$), since states of $\mathcal{A}$ in the remaining set R will be processed at most d times. In practice, however, R2F based construction (and Hybrid computation which also has $O(dpn^2)$ time complexity since it performs R2F steps) can beat F2R based construction.

We did not perform an extensive study on automata with larger state numbers, since it takes too long with the sequential baseline implementation. For example, sequential PMF generation takes around 75 minutes for an automaton with 32000 states and 128 letters, whereas our Hybrid implementation completes in 3 minutes.

6 Conclusion and Future Work

We investigated the efficient implementation and use of modern multicore CPUs to scale the performance of synchronizing sequence generation heuristics. We parallelized one of the well-known heuristics Greedy. We mainly focused on the PMF generation phase (which is employed by almost all the heuristics in the literature), since it is the most time consuming part of Greedy. Even with no parallelization, our algorithmic improvements yielded a 20x speedup on Greedy for automatons with 4000 states and 128 inputs. Furthermore, around 150x speedup has been obtained with 16 threads for the same automata class.

To eliminate threats to validity, we checked and confirmed that the sequence constructed by each algorithm is indeed a synchronizing sequence. We also compared the length of the sequences constructed by the original implementation of Greedy and different versions of Greedy algorithms suggested in this paper. We observed that regardless of the PMF construction approach used, for each pair $\langle s_i, s_j \rangle$, we obtain the same length $|\tau(\langle s_i, s_j \rangle)|$ for the shortest merging sequences, but the actual shortest merging sequence $\tau(\langle s_i, s_j \rangle)$ can differ, which causes around $\pm 1\%$ difference in the length of the synchronizing sequences.

As a future work, we will apply our techniques to other heuristics in the literature that are relatively slower than Greedy but can produce shorter synchronizing sequences. For these heuristics, parallelizing only the PMF generation phase may not be sufficient since the synchronizing sequence construction part of these heuristics are much more expensive compared to Greedy. Hence, we aim to parallelize the whole sequence generation process. Another problem we want to study is the use of cutting-edge manycore architectures such as GPUs and FPGAs to make such heuristics faster and more practical for large scale automatons.

Acknowledgements

This work is supported by TÜBİTAK Grants #114E569 and #115C018.

References

1. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.
2. F. C. Hennie. Fault-detecting experiments for sequential circuits. In *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, Princeton, New Jersey, 1964.
3. H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Trans. Computers*, 46(1):93–99, 1997.
4. R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Trans. Computers*, 51(9):1111–1117, 2002.
5. A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, 2005.
6. A. S. Simão, A. Petrenko, and N. Yevtushenko. On reducing test length for FSMs with extra states. *Software Testing, Verification and Reliability*, 22(6):435–454, 2012.
7. R. M. Hierons and H. Ural. Generating a checking sequence with a minimum number of reset transitions. *Autom. Softw. Eng.*, 17(3):217–250, 2010.
8. P. Schrammel, T. Melham, and D. Kroening. Chaining test cases for reactive system testing. In *ICTSS*, volume 8254 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2013.
9. R. Groz, A. S. Simão, A. Petrenko, and C. Oriat. Inferring finite state machines without reset using state identification sequences. In *ICTSS*, volume 9447 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2015.
10. G. V. Jourdan, H. Ural, and H. Yenigun. Reduced checking sequences using unreliable reset. *Inf. Process. Lett.*, 115(5):532–535, 2015.
11. M. V. Berlinkov. On the probability of being synchronizable. In *CALDAM*, volume 9602 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2016.
12. D. Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990.
13. A. N. Trahtman. Some results of implemented algorithms of synchronization. In *10th Journees Montoises d'Inform*, 2004.
14. A. Roman. Synchronizing finite automata with short reset words. *Applied Mathematics and Computation*, 209(1):125–136, 2009.
15. R. Kudlacik, A. Roman, and H. Wagner. Effective synchronizing algorithms. *Expert Systems with Applications*, 39(14):11746–11757, 2012.
16. A. Roman and M. Szykula. Forward and backward synchronizing algorithms. *Expert Syst. Appl.*, 42(24):9512–9527, 2015.
17. U. C. Türker. Parallel algorithm for deriving reset sequences from deterministic incomplete finite automata - *submitted. IJFCS International Journal of Foundations of Computer Science*, -(–):–, —.
18. B. K. Natarajan. An algorithmic approach to the automated design of parts orienters. In *FOCS*, pages 132–142, 1986.
19. S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.