Journal of Parallel and Distributed Computing xxx (xxxx) xxx

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Multicore and manycore parallelization of cheap synchronizing sequence heuristics ☆

Sertaç Karahoda [a,*], Osman Tufan Erenay [a], Kamer Kaya [a], Uraz Cengiz Türker [b], Hüsnü Yenigün [a]

[a] Computer Science and Engineering, Faculty of Science and Engineering, Sabanci University, Tuzla, Istanbul, Turkey
[b] Department of Informatics, University of Leicester, Leicester, United Kingdom

ARTICLE INFO

Article history:
Received 11 May 2019
Received in revised form 25 October 2019
Accepted 13 February 2020
Available online xxxx

MSC:
03D05
65Y05

Keywords:
Software testing
Finite state automata
Synchronizing sequences
Parallel algorithms
Graphics processing units

ABSTRACT

An important concept in finite state machine based testing is *synchronization* which is used to initialize an implementation to a particular state. Usually, *synchronizing sequences* are used for this purpose and the length of the sequence used is important since it determines the cost of the initialization process. Unfortunately, the shortest synchronization sequence problem is NP-Hard. Instead, heuristics are used to generate short sequences. However, the cubic complexity of even the fastest heuristic algorithms can be a problem in practice. In order to scale the performance of the heuristics for generating short synchronizing sequences, we propose algorithmic improvements together with a parallel implementation of the cheapest heuristics existing in the literature. To identify the bottlenecks of these heuristics, we experimented on random and slowly synchronizing automata. The identified bottlenecks in the algorithms are improved by using algorithmic modifications. We also implement the techniques on multicore CPUs and Graphics Processing Units (GPUs) to take benefit of the modern parallel computation architectures. The sequential implementation of the heuristic algorithms are compared to our parallel implementations by using a test suite consisting of 1200 automata. The speedup values obtained depend on the size and the nature of the automaton. In our experiments, we observe speedup values as high as 340x by using a 16-core CPU parallelization, and 496x by using a GPU. Furthermore, the proposed methods scale well and the speedup values increase as the size of the automata increases.

1. Introduction

Due to the complex nature of current systems, developing a correct one is not an easy task. Several validation techniques are used to build some confidence during development, but testing stands out as one of the most practical validation method [21]. Testing in practice is quite a labor intensive activity if performed manually, and it is reported to consume more than half of the development cost [21]. Therefore, automated test techniques have to be used in order to bring down the cost of testing.

The automation of testing is performed over a wide spectrum of activities; e.g., test generation, test execution, management of the test suite, etc. Among these activities, automatic test generation is defensibly the most scientifically challenging one. It is also possibly one of the most important factors that can improve the effectiveness of practical testing.

Model Based Testing (MBT) is one of the techniques used to generate effective and practical test cases. In MBT, formal models of system requirements are used. For describing the behavior of a system formally, state based formalisms are used and there has been much interest in testing with finite state machines (FSMs) (e.g., see [5,10,11,14,15,23,28,32]). While test tools might allow the user to use richer formalisms and languages, these models can usually be mapped to FSMs. To employ FSMs for testing, one needs to recognize the state of the system under test (SUT) and bring the SUT to a particular state. The state recognition can be accomplished by using special sequences like distinguishing sequences [30] or Unique Input Output sequences [16], when such sequences exist. To bring the SUT to a particular state, some works assume the existence of a trusted *reset* input in the SUT, however, such a reset input is not always available. Even if a reset input is available, performing the reset operation can be time consuming. Hence, there are cases where the use of a reset input is not preferred [13,17,27].

☆ A preliminary version of this article appeared in *S. Karahoda, O.T. Erenay, K. Kaya, U. C. Türker, H. Yenigün: "Parallelizing Heuristics for Generating Synchronizing Sequences". ICTSS 2016: 106-122.*

* Corresponding author.
E-mail addresses: skarahoda@sabanciuniv.edu (S. Karahoda), osmantufana@sabanciuniv.edu (O.T. Erenay), kaya@sabanciuniv.edu (K. Kaya), u.c.turker@leicester.ac.uk (U.C. Türker), yenigun@sabanciuniv.edu (H. Yenigün).

https://doi.org/10.1016/j.jpdc.2020.02.009
0743-7315/

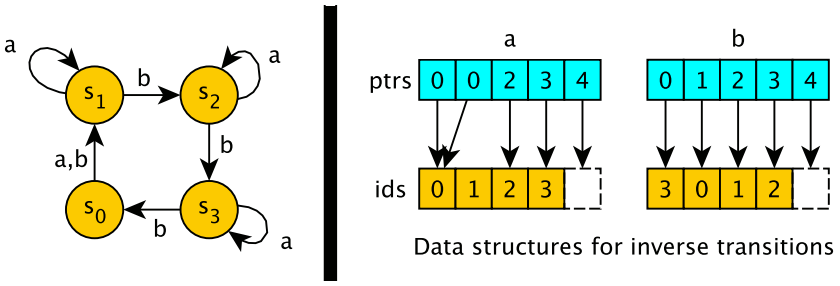


Fig. 1. A synchronizable automaton $\mathcal{A}$ (left), and the data structures we used to store and process the transition function δ^{-1} in memory (see Section 4.3 for the details). A synchronizing sequence for $\mathcal{A}$ is *abbbabbba*.

A *synchronizing sequence* for an FSM (also known as a *reset sequence*, or a *reset word*), is a sequence of inputs such that when it is applied to the FSM, the machine ends up in a particular state no matter at which state it initially is [18]. Therefore a synchronizing sequence can be considered as a compound reset input. The motivation to study reset sequences comes from different fields including automata theory, robotics, bio-computing, set theory, propositional calculus, model based testing and many more [1,3,6,10,12,18,22,24,31,33].

For a given FSM, a synchronizing sequence may or may not exist. Yet, we know that a large-scale FSM is almost always synchronizable [4]. The complexity of checking the existence of a synchronizing sequence is $O(pn^2)$ for an automata with p inputs and n states [12]. Although the shortest such sequence is usually better in terms of synchronization cost and energy used, unfortunately, the problem of finding the shortest sequence is NP-hard. Hence, various *synchronizing heuristics* have been proposed to find short sequences, e.g., [12,19,25,29]. It has been conjectured that the length of the shortest synchronizing sequence is at most $(n-1)^2$ [7]. However, the upper bound on the sequence length is $O(n^3)$ for the above-mentioned synchronizing heuristics.

To the best of our knowledge, the parallelization and scalability of these heuristics have not been thoroughly addressed. Even the fastest algorithms, Greedy [12] and Cycle [26], has $O(n^3+pn^2)$ complexity. In this work, we focus on the runtime performance of these heuristics, design parallel algorithms, and experiment on multicore CPUs and manycore GPUs to prove their scalability. Since Greedy is one of the two fastest heuristics in the literature and yields shorter sequences compared to Cycle, in this study, we mainly focus on its parallelization. However, the proposed techniques can be effectively used to parallelize other heuristics in the literature. As far as we know, this is the first study towards efficient parallelization of synchronizing heuristics.

All the synchronizing heuristics mentioned consist of a preprocessing phase, followed by synchronizing sequence generation phase. As presented in this paper, our initial experiments revealed that the relative cost of the preprocessing depends on the structure of the automaton. For a random automaton, preprocessing is the most expensive part. However, for special types of automata classes, the preprocessing cost can be negligible compared to the cost of sequence construction. Therefore in this work, we focus on both phases. However, depending on the automata structure, the improvement, i.e., the speedup w.r.t. to the number of threads differ. For instance, for a random automaton with $n = 8000$ and $p = 128$, the proposed algorithmic approach improves the naive, sequential implementation of Greedy by 33× and with 16 threads, the speedup increases to 340×. Furthermore, with a single GPU, we obtain 496× speedup over the sequential implementation.

The paper is organized as follows: Section 2 presents the background and notation and formally define synchronizing sequences. The Greedy algorithm is described in Section 3 whereas the proposed parallelization approach together with implementation details are described in Section 4. The experimental results are presented in Sections 5 and 6 concludes the paper.

2. Preliminaries

Although FSMs produce outputs when fed with an input, the response is not important in the context of synchronization. In this work, we consider an FSM as a complete automaton where only the state transitions are performed in case of an input. Formally, an *automaton* is a triple $\mathcal{A} = (S, \Sigma, \delta)$ where S and Σ are finite sets of n states and p input symbols, respectively. $\delta : S \times \Sigma \to S$ is the transition function of the automata. If $\mathcal{A}$ is at state s and x is applied as an input, then $\delta(s, x)$ will be the new state of $\mathcal{A}$. Fig. 1 shows a toy, 4-state, 2-input automaton.

In this paper, $|w|$ denotes the length of an input sequence $w \in \Sigma^\star$ where the zero-length input sequence is denoted as $\varepsilon \in \Sigma^\star$. For the empty sequence, $\delta(s, \varepsilon) = s$. For an input sequence $w \in \Sigma^\star$ and an input symbol $x \in \Sigma$, let xw be the concatenated sequence. Let us extend the transition function δ in a naive way for input sequences as

$$\delta(s, xw) = \delta(\delta(s, x), w) \text{ for } x \in \Sigma \text{ and } w \in \Sigma^\star$$

and for sets of states as

$$\delta(S', w) = \{\delta(s, w) | s \in S'\} \text{ for } S' \subseteq S.$$

We also use

$$\delta^{-1}(s, x) = \{s' \in S | \delta(s', x) = s\}$$

to denote the set of those states with a transition to state s with input x.

For an automaton $\mathcal{A} = (S, \Sigma, \delta)$, an input sequence $w \in \Sigma^\star$ is a *merging sequence* for $S' \subseteq S$ if $|\delta(S', w)| = 1$. We call the sets of states with at least one merging sequence *mergable*. An input sequence $w \in \Sigma^\star$ is a *synchronizing sequence* for $\mathcal{A}$ if $|\delta(S, w)| = 1$. We call automata with at least one synchronizing sequence *synchronizable*. Another way to define such automata is below:

Proposition 1 ([12,22]). *An automaton $\mathcal{A} = (S, \Sigma, \delta)$ 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 $C^{(2)} = \{\{s_i, s_j\} | s_i, s_j \in C\}$ be the set of all *multisets* of C with cardinality two. If $s_i = s_j$ an element $\{s_i, s_j\} \in C^{(2)}$ is called *a singleton*. Otherwise, it is called *a pair*.

Due to Proposition 1, we can understand if an automaton is synchronizable by checking the existence of merging sequences for all pairs. To efficiently perform this check, let us first define the notion of a pair automaton.

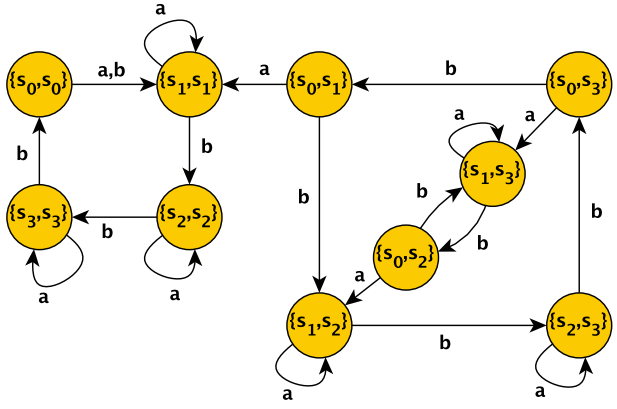


Fig. 2. The pair automaton $\mathcal{A}^{(2)}$ of the automaton in Fig. 1.

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

Fig. 2 shows an example of a pair automaton.

As stated above in the introduction, Černý has conjectured that the length of the shortest synchronizing word of an automaton with n states is at most $(n-1)^2$ [7]. Černý has also provided a class of automata $\mathcal{A}_c$, called *Černý automata*, which hits to this conjectured upper bound. An example of a Černý automaton is given in Fig. 1. Given a number of states n, the Černý automaton with n states has a well-defined structure. There are two inputs in the alphabet. The transitions of the first input (a in the figure) are all loops except one. That is $\delta(s_i, a) = s_i$ for $1 \leq i \leq n$ and $\delta(s_0, a) = s_1$. The transitions of the second input (b in the figure) form a cycle, i.e., $\delta(s_i, b) = s_{i+1}$ for $0 \leq i \leq n-1$ and $\delta(s_{n-1}, b) = s_0$.

2.1. Graphics Processing Units and CUDA

At the hardware level, a CUDA capable GPU is a collection of *multiprocessors* (SMX), each having hundreds of *cores*. In total, there are thousands of cores on a modern GPU. Each multiprocessor has its own shared memory which is common to all its cores. It also has a set of registers, texture memory (a read only memory for the GPU), and constant (a read only memory for the GPU that has the lowest access latency) memory caches.

At the software level, the CUDA model is a collection of threads running in parallel. The programmer decides the number of threads to be launched. A group of threads, called a *warp*, run simultaneously. GPU's single instruction, multiple thread (SIMT) execution manages all threads in the same warp via a single controller and hence, they execute the same instruction in parallel usually with different data items. At any given time, *blocks* of warps run on a multiprocessor. Due to the zero-overhead warp scheduling technique implemented by modern GPUs, the shared memory data used by the threads in a block stay there as long as the block is alive. Although this technique makes the shared memory a scarce, critical resource, it also makes a GPU excelled in latency-hiding and managing millions of concurrent threads.

In the CUDA model, each thread executes a piece of code called a *kernel*. The kernel is the core code to be executed by all the threads. During its execution, a thread t_i accesses data residing on the GPU global memory by using its thread ID. Since the GPU memory is available to all the threads, a thread can access any memory location. When the kernel has '**if**'s and '**else**'s, i.e., when the computation is branched (also called divergent in GPU computing), the threads in the same warp may be serialized since they are controlled by a single controller. When this happens, concurrency decreases and performance degrades.

3. Understanding the mechanics of Eppstein's Greedy algorithm

As mentioned before, almost all the synchronizing heuristics in the literature run in two phases; in the first phase of the Eppstein's Greedy algorithm, a shortest merging sequence for each mergeable pair of states is found and stored in an auxiliary data structure. Later, in the second phase, a synchronizing sequence is constructed by concatenating the words found at each iteration. Computing a shortest merging sequence for a mergeable state pair $\{s_i, s_j\} \subset S$ of an automaton $\mathcal{A} = (S, \Sigma, \delta)$, can be done in time $O(pn^2)$. This is equal to the number of transitions inside the pair automaton; indeed, one can use a *backwards* Breadth First Search (BFS) with the same complexity seeded from the singleton states, i.e., $\{s_i, s_i\}$s, of the pair automaton, as we will explain below.

The function $\tau : S^{(2)} \to \Sigma^\star$ is called *a pairwise merging function (PMF) for* $\mathcal{A} = (S, \Sigma, \delta)$, where $\tau(\{s_i, s_j\})$ is a shortest merging sequence for all mergeable $\{s_i, s_j\}$ pairs. It is undefined for all non-mergeable pairs; note that non-mergeable pairs do not exist if $\mathcal{A}$ is a synchronizable automaton. For a given $\mathcal{A}$, Algorithm 1 computes a PMF. Initially, $\tau(\{s, s\})$ is set to ε for all $\{s, s\} \in S^{(2)}$ (line 1). Furthermore, $\tau(\{s_i, s_j\})$ is set to *undefined* for all $\{s_i, s_j\} \in S^{(2)}$ where $s_i \neq s_j$ (line 2). The algorithm iteratively updates these values throughout its search for merging sequences for all the pairs in $S^{(2)}$.

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

```
   input  : An automaton A = (S, Σ, δ)
   output: A PMF τ : S^(2) → Σ*
1  foreach singleton {s, s} ∈ S^(2) do τ({s, s}) ⟵ ε;
2  foreach pair {s_i, s_j} ∈ S^(2) do τ({s_i, s_j}) ⟵ undefined;
3  F ⟵ {{s, s}|s ∈ S}; // all singleton states of A^(2)
4  R ⟵ {{s_i, s_j}|s_i, s_j ∈ S ∧ s_i ≠ s_j}; // all pair states of
       A^(2)
5  while F is not empty do
6  |  F, R, τ ⟵ BFS_step(A, F, R, τ);
```

Throughout the BFS, Algorithm 1 maintains a *frontier* set F which at the beginning, is set to the all singleton states (line 3). In the algorithm, R represents the set of pairs $\{s_i, s_j\}$ with $\tau(\{s_i, s_j\})$ is currently undefined. At each iteration (lines 5–6), a single BFS step is performed where a possible implementation, "Frontier to Remaining (F2R)", is given in Algorithm 2. As the name suggests, the *BFS_step_F2R* function constructs the next frontier F' from the current frontier F and the edges/transitions are processed from the frontier vertices to the remaining, unvisited vertices. Starting from the pairs $\{s_i, s_j\} \in F$ (line 2), the algorithm identifies a pair $\{s'_i, s'_j\} \in R$ such that $s'_i = \delta(s_i, x)$ and $s'_j = \delta(s_j, x)$ for some $x \in \Sigma$ (lines 4 and 5). The value of the PMF, τ, at this pair is set (line 6), and then the next frontier is updated (line 7).

Based on the PMF computed as described above, one can implement the Greedy algorithm as described in Algorithm 3. The algorithm keeps track of the set of active states C yet to be merged, where C is initially equal to S (line 5). An active state pair $\{s_i, s_j\} \in C^{(2)}$ with the shortest merging sequence is first computed by the algorithm (line 8). The overall synchronizing sequence Γ is then extended with $\tau(\{s_i, s_j\})$ (line 9). Finally, the same merging sequence $\tau(\{s_i, s_j\})$ is applied to C to find the next set of active states. When only a single active state remains, the automata is synchronized and Γ is a synchronizing sequence.

For an automaton with n states and p inputs, the first phase (lines 1–3 of Algorithm 3) can be implemented to run in time

Algorithm 2: BFS_step_F2R

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

```
1 F' ⟵ ∅;
2 foreach {s_i, s_j} ∈ F do
3     foreach x ∈ Σ do
4         foreach {s'_i, s'_j} such that s'_i ∈ δ^{-1}(s_i, x) and
          s'_j ∈ δ^{-1}(s_j, x) do
5             if τ({s'_i, s'_j}) is undefined then // {s'_i, s'_j} ∈ R
6                 τ({s'_i, s'_j}) ⟵ xτ({s_i, s_j});
7                 F' ⟵ F' ∪ {{s'_i, s'_j}};
8 R' ⟵ R \ F';
```

Algorithm 3: Eppstein's Greedy Algorithm

input : An automaton $\mathcal{A} = (S, \Sigma, \delta)$
output: A synchronizing sequence Γ for $\mathcal{A}$ (or fail if $\mathcal{A}$ is not synchronizable)

```
1  compute a PMF τ using Algorithm 1;
2  if there exists a pair {s_i, s_j} such that τ({s_i, s_j}) is undefined
   then
3      report that A is not synchronizable and exit;
4  foreach s_i, s_j, s_k ∈ S do compute δ(s_k, τ({s_i, s_j}));
5  C ⟵ S; // C will keep track of the current set of
   states
6  Γ ⟵ ε; // Γ is the synchronizing sequence to be
   constructed
7  while |C|> 1 do // we have two or more states yet to
   be merged
8      {s_i, s_j} ⟵ FindMin(C, τ) in C^⟨2⟩;
9      Γ ⟵ Γ τ({s_i, s_j});
10     C ⟵ δ(C, τ({s_i, s_j}));
```

Algorithm 4: FindMin

input : Current set of state C and the PMF function τ
output: A pair of state $\{s_i, s_j\}$ with minimum $|\tau(\{s_i, s_j\})|$ among all pairs in $C^{\langle 2\rangle}$

```
1 {s_i, s_j} ⟵ undefined;
2 foreach {s_k, s_ℓ} ∈ C^⟨2⟩ do
3     if {s_i, s_j} is undefined or |τ({s_k, s_ℓ})|< |τ({s_i, s_j})| then
4         {s_i, s_j} ⟵ {s_k, s_ℓ};
5         if |τ({s_k, s_ℓ})|= 1 then
6             break;
```

$O(pn^2)$ and Phase 2 of Greedy (lines 4–10 of Algorithm 3) can be implemented to run in time $O(n^3)$. Hence, the overall time for Greedy is $O(n^3 + pn^2)$ [12, Theorem 5].

The practical performance of Greedy and its two phases depend on the structure of the automaton. If the automaton synchronizes quickly, that is if the cardinality of C decreases quickly at each iteration of the while loop in Algorithm 3, then the PMF construction phase given in Algorithm 1 will be the main bottleneck. For many randomly chosen automata, this is what we have observed. To further test this, we performed an experimental analysis and measured how much Phase 1 (the PMF construction) and Phase 2 (the synchronizing sequence construction) contribute to the running time in practice for a sequential implementation. To perform this experiment, we randomly generated automata by choosing the target of each transition in a uniformly random manner from the state space. For each automata class, i.e., with given number of states and number of inputs, we generated 100 random automata and reported the average value. The first four rows of Table 1 show that for a randomly chosen automaton, the first phase actually dominates the running time of the algorithm; it took more than 90% of the runtime for all the automata classes with $n \in \{2000, 4000, 8000\}$ states and $p \in \{2, 8, 32, 128\}$ inputs. Based on these results, in order to improve the performance of Greedy, one can argue that the PMF is the bottleneck in practice with high probability.

Table 1
The PMF construction time (t_{PMF}), and overall execution time (t_{ALL}) for $n \in \{2000, 4000, 8000\}$-state and $p \in \{2, 8, 32, 128\}$-input automata. For each entry in the first four rows, we generated 100 random automata and report the average value. For the last row, the algorithm is run on the same automata 5 times and the average of these runs is reported.

	$n = 2000$			$n = 4000$			$n = 8000$		
p	t_{PMF}	t_{ALL}	$\frac{t_{PMF}}{t_{ALL}}$	t_{PMF}	t_{ALL}	$\frac{t_{PMF}}{t_{ALL}}$	t_{PMF}	t_{ALL}	$\frac{t_{PMF}}{t_{ALL}}$
2	0.172	0.185	0.929	1.184	1.240	0.954	5.899	6.325	0.933
8	0.504	0.517	0.975	2.709	2.768	0.978	14.289	14.721	0.971
32	2.113	2.126	0.994	9.925	9.986	0.994	51.783	52.233	0.991
128	9.126	9.140	0.999	40.356	40.418	0.998	193.548	193.982	0.998
Černý	0.096	4.836	0.020	1.026	42.771	0.024	5.584	797.692	0.007

Within the synchronization context, there are special classes of automaton which are harder to synchronize, i.e., slowly synchronizing. As mentioned in Section 2, it is conjectured that for a synchronizing automaton with n states, the length of the shortest synchronizing sequence is at most $(n-1)^2$, which is known as the Černý Conjecture in the literature. Posed half a century ago, the conjecture is still open and claimed to be one of the longest standing open problem in automata theory. Arguably, the most famous slowly synchronizing automata in the literature are Černý automata [8,9], for which the length of the shortest synchronizing word is $(n-1)^2$, the bound stated by the Černý conjecture. For instance, the automaton presented on the left side of Fig. 1 is a Černý automaton with 4 states. Hence, the length of the shortest synchronizing sequence for this automaton is 9. The BFS tree for a Černý automaton has depth $n \times (n-1)/2$, where each level has only one node as shown in Fig. 3. In a sense, Černý automata is possibly the toughest class of automata for Phase 2 of Greedy. Fig. 3 presents the shortest path tree for the 4-state Černý automaton given in Fig. 1.

To identify the bottleneck of Greedy on slowly synchronizing automata, we generated Černý automata with $n \in \{2000, 4000, 8000\}$ states and run the algorithm 5 times on these automata. The averages of these runs are reported in the last row of Table 1. This time almost all the execution time is spent by Phase 2.

4. Parallel generation of synchronizing sequences on multicore and manycore architectures

As the results in the previous section show, the relative costs of Phase 1 and Phase 2 completely differ for random and slowly synchronizing automata classes. This is why optimization and parallelization for both phases are important in practice. Here we present our parallel implementation of Greedy, together with some algorithmic modifications.

4.1. Computing a PMF in parallel

Breadth First Search is the main kernel used to construct a PMF τ for a given automaton $\mathcal{A}$. The BFS is performed on the pair

Please cite this article as: S. Karahoda, O.T. Erenay, K. Kaya et al., Multicore and manycore parallelization of cheap synchronizing sequence heuristics, Journal of Parallel and Distributed Computing (2020), https://doi.org/10.1016/j.jpdc.2020.02.009.

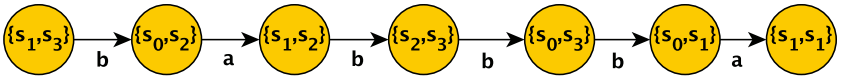


Fig. 3. The shortest path tree for the pairs of the 4-state Černý automaton given in Fig. 1.

automaton $\mathcal{A}^{\langle 2\rangle}$ graph where the state pairs correspond to the vertices and inversely oriented transitions among the state pairs are the edges of the graph. In traditional BFS, the graph traversal starts from a single vertex; however, for synchronization, the kernel is seeded from multiple vertices corresponding to all the singleton state pairs of $\mathcal{A}^{\langle 2\rangle}$. One can consider this as performing a BFS from a virtual vertex with no incoming edge set and outgoing edges to all the vertices corresponding to the singleton pairs in the pair automaton. This vertex set is the first frontier, i.e., level 0 vertices of the BFS kernel. At each iteration, the frontier is updated by using the BFS_step_F2R given in Algorithm 2. The process will continue until all the pairs are visited and their shortest merging sequences are computed.

On a multicore/manycore architecture, a single execution of BFS_step_F2R is parallelized by assigning a single vertex (i.e., a single state pair) of F to a single thread. The threads process the frontier edges (which correspond to the incoming transitions of the pairs based on δ^{-1}) of the assigned vertices to find the next frontier F'. After each step, the threads are synchronized before starting to the next step from the next frontier. On the CPU, this is achieved by a barrier. On a GPU, each step is a different kernel execution. Such a global synchronization is necessary since one cannot be sure about the correctness of the next frontier without traversing all the edges of the previous frontier. Thanks to the inverse automata data structure presented in Fig. 1, each edge can be processed in constant time. Hence the work-load of each phase is proportional to the number of incoming transitions to the frontier pairs, and the total work is $O(pn^2)$.

To efficiently access the transitions from the endpoints, i.e., to quickly find $\delta^{-1}(s,x)$ for all $x \in \Sigma$ and $s \in S$, we utilize the $\texttt{ptrs}_x$ and $\texttt{js}_x$ arrays for each input $x \in \Sigma$ as shown in Fig. 1 (right). The length of $\texttt{ptrs}_x$ is $n+1$ and the length of $\texttt{js}_x$ is n. The entries

$$\texttt{js}_x[\texttt{ptrs}_x[s]],\ \texttt{js}_x[\texttt{ptrs}_x[s]+1],\ \texttt{js}_x[\texttt{ptrs}_x[s]+2],\ \ldots,\ \texttt{js}_x[\texttt{ptrs}_x[s+1]-1]$$

store the ids of the states in $\delta^{-1}(s,x)$. Since we store only a single entry for each edge the memory footprint of these array-based inverse automata representation is optimal. Furthermore, in both sequential and parallel F2R, the states from $\texttt{js}_x[\texttt{ptrs}_x[s]]$ to $\texttt{js}_x[\texttt{ptrs}_x[s+1]-1]$ are accessed one after the other. Hence, the proposed data structure is good for spatial locality and better cache utilization.

In the sequential version of BFS_step_F2R, the next frontier never have duplicates; line 5 of Algorithm 2 guarantees that only the unvisited pairs, i.e., the ones with an undefined PMF value is inserted to the next frontier. This check also guarantees that a pair is not inserted to the next frontier more than once. However, such a check is not sufficient for the parallel execution since two threads can perform the same check for the same vertex at the same time and they both can try to insert it to the next frontier. One can eliminate such duplicates by using a mutex-like global structure for the checks and updates. Although mutex is an expensive tool, a single, global list for the next frontier F' is sufficient for this approach. One can also keep a separate local list F'_t for each thread t, insert the visited pairs to these local lists, and then perform a duplicate pair elimination after each BFS_step_F2R. However, this approach also suffers from the extra overhead, and keeping a local queue may not be feasible for manycore architectures.

Algorithm 5: BFS_step_F2R (parallel)

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

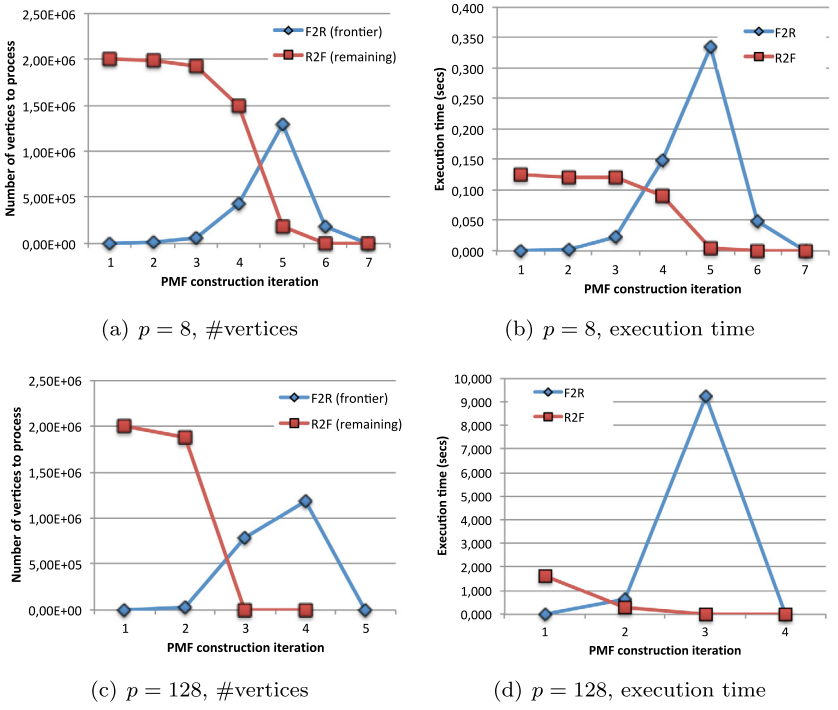
```
1  foreach thread t do F'_t ⟵ ∅ ;
2  foreach {s_i, s_j} ∈ F in parallel do
3      foreach x ∈ Σ do
4          foreach {s'_i, s'_j} where s'_i ∈ δ^{-1}(s_i, x) and s'_j ∈ δ^{-1}(s_j, x)
           do
5              if τ({s'_i, s'_j}) is undefined then // {s'_i, s'_j} ∈ R
6                  τ({s'_i, s'_j}) ⟵ xτ({s_i, s_j});
7                  F'_t ⟵ F'_t ∪ {{s'_i, s'_j}};
8  F' ⟵ ∅;
9  foreach thread t do F' ⟵ F' ∪ F'_t ;
10 R' ⟵ R \ F';
```

In our parallel algorithm (Algorithm 5) for multicore CPUs, 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 $\{s'_i, s'_j\}$ at the same time, both of them insert it to their local frontiers (lines 5–7). However, we ignore the duplicate pairs since they will not have an impact on the correctness of Algorithm 5. Although the duplicates will yield extra and unnecessary work performed in the next step, they will probably not generate more duplicates after the next step due to the if check at line 5 of the algorithm. Furthermore, in our preliminary experiments on a multicore CPU, we observed that at most one out of thousand extra pairs are inserted to F' when they are allowed. Hence, we let the threads create duplicates since the total cost due to these extra pairs is negligible compared to the cost of checking and resolving duplicates.

The cost of maintaining a local queue for each thread is not feasible when a manycore architecture has more than hundreds or thousands of threads which is the case for GPU. Hence, for manycore architectures, we follow another approach that helps us to avoid queue management overhead. This approach will be described in Section 4.1.2.

4.1.1. An alternative approach for a BFS step and hybrid PMF construction

In addition to "Frontier to Remaining", one can perform a single BFS step (hence all the BFS) by using the original transition function δ instead of the inverse transition function δ^{-1}. This approach is called "Remaining to Frontier (R2F)" since the edges/transitions are processed w.r.t. their original orientation; i.e., from remaining vertices to the ones in the frontier. At each iteration, the next frontier is formed by all the $\{s_i, s_j\} \in R$ state pairs with undefined PMF values that can reach to at least one pair inside the frontier with a single input. Algorithm 6 follows this approach. In the parallel implementation, the threads process the transitions of the remaining state pairs instead of the ones in the frontier. Similar to the F2R approach, a pair (this time from R) is assigned to a single thread, and a local remaining pair array R'_t is used for each thread t for a lock-free parallelization of R2F.



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

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

Fig. 4. 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).

As the algorithm shows, for each pair in $\{s_i, s_j\} \in R$ and each input $x \in \Sigma$, first $\{s'_i, s'_j\} = \{\delta(s_i, x), \delta(s_j, x)\}$ is computed. Then $\{s'_i, s'_j\}$ is checked to be in the frontier (lines 5–6). Note that it is sufficient to check if $\tau(\{s'_i, s'_j\})$ is defined or not, since all the edges from the remaining vertices to already visited/processed vertices have their latter endpoint in the frontier. If $\tau(\{s'_i, s'_j\})$ is defined then a shortest merging sequence for $\{s_i, s_j\}$ can be obtained by concatenating x and $\tau(\{s'_i, s'_j\})$ (line 7). A boolean variable, *connected*, is used to check if a remaining vertex will stay as a remaining node in R (lines 10 and 11) or not.

Algorithm 6: BFS_step_R2F (parallel)

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

1 **foreach** *thread* t **do** $R'_t \longleftarrow \emptyset$;
2 **foreach** $\{s_i, s_j\} \in R$ **in parallel do**
3 *connected* $\longleftarrow$ **false**;
4 **foreach** $x \in \Sigma$ **do**
5 $\{s'_i, s'_j\} \longleftarrow \{\delta(s_i, x), \delta(s_j, x)\}$;
6 **if** $\tau(\{s'_i, s'_j\})$ *is defined* **then** // $\{s'_i, s'_j\} \in F$
7 $\tau(\{s_i, s_j\}) \longleftarrow x\tau(\{s'_i, s'_j\})$;
8 *connected* $\longleftarrow$ **true**;
9 **break**;
10 **if** *not connected* **then**
11 $R'_t \longleftarrow R'_t \cup \{\{s_i, s_j\}\}$;
12 $R' \longleftarrow \emptyset$;
13 **foreach** *thread* t **do** $R' \longleftarrow R' \cup R'_t$;
14 $F' \longleftarrow R \setminus R'$;

Fig. 4 shows the execution times of both F2R and R2F steps during PMF construction for automata having $n = 2000$ states and $p = \{8, 128\}$ inputs. There two important observations; first the execution times of F2R and R2F are correlated with the number of vertices in the frontier and remaining sets, respectively. Second, as expected, the frontier size increases and decreases fast. Furthermore, when the number of remaining vertices is much smaller than the number of frontier vertices the F2R-based steps become more expensive than R2F-based steps. Such observations have also been made by Beamer et al. who propose a direction-optimized BFS algorithm for generic graphs [2]. Using two different parallel GPU kernels which perform similar tasks on different phases of the execution have shown to be beneficial in the literature also for other problems, e.g., see [20]. Due to the first observation, it indeed makes sense to start with F2R and switch to R2F once the number of the edges of the frontier vertices becomes less than the number of the edges of the remaining vertices. In this study, to get rid of the overhead due to edge counting, we used vertex counts instead of edge counts; we perform the switch when the number of frontier vertices becomes less than the number of remaining, unprocessed vertices. Since each pair is counted only once, the total counting overhead will be $O(n^2)$. The parallel PMF construction algorithm is presented in Algorithm 7.

4.1.2. Searching in entire set for manycore architectures

In Algorithms 5 and 6, each thread constructs their local frontier and remaining sets in parallel. A drawback of this approach is the increased memory footprint; since we cannot predict the local frontier sizes at each step, to fully avoid locks and other parallelization constructs, we need to allocate a space large enough to store all possible pairs. This approach is feasible for multicore processors since we only have tens of cores.

As explained in Section 2.1, a GPU is a high-performance accelerator that can concurrently execute thousands of threads at the same time. However, the global memory size on a GPU is not as large as the memory we have on the host server. Hence, the previous approach taken is not feasible on GPUs. Furthermore, it can be costly to merge thousands of local frontier sets. In addition, the GPU implementation of Algorithm 5 can create a large number of duplicate pairs, since the probability of a pair

Algorithm 7: Computing a function $\tau : S^{(2)} \rightarrow \Sigma^{\star}$ (Hybrid)

input : An automaton $\mathcal{A} = (S, \Sigma, \delta)$
output: A function $\tau : S^{(2)} \rightarrow \Sigma^{\star}$

foreach *singleton* $\{s, s\} \in S^{(2)}$ **do** $\tau(\{s, s\}) \longleftarrow \varepsilon$;
foreach *pair* $\{s_i, s_j\} \in S^{(2)}$ **do** $\tau(\{s_i, s_j\}) \longleftarrow$ *undefined*;
$F \longleftarrow \{\{s, s\} | s \in S\}$; // all singleton states of $\mathcal{A}^{(2)}$
$R \longleftarrow \{\{s_i, s_j\} | s_i, s_j \in S \wedge s_i \neq s_j\}$; // all pair states of $\mathcal{A}^{(2)}$
while *F is not empty* **do**
  if $|F| < |R|$ **then**
    $F, R, \tau \longleftarrow$ BFS_step_F2R($\mathcal{A}, F, R, \tau$);
  else
    $F, R, \tau \longleftarrow$ BFS_step_R2F($\mathcal{A}, F, R, \tau$);

visited by more than a single thread increases with the number of threads. Therefore, we need another approach instead of local frontiers.

For GPU parallelization, the algorithm processes the entire pair set $S^{(2)}$, instead of R or F. We call this approach S2R and S2F, respectively. At each iteration of S2R, $S^{(2)}$ is used and the algorithm checks if the current pair is in F or not. If the pair is in F, then the algorithm continues as in F2R. S2F has the same idea of S2R. However, S2F checks if the pair is in R or not. If it is in R it executes the same logic in R2F. Algorithms 8 and 9 present the pseudocodes for the proposed, parallel, GPU-based algorithms.

Algorithm 8: BFS_step_S2R (parallel)

input : An automaton $A = (S, \Sigma, \delta)$, the frontier level f, and a partial PMF τ
output: updated function τ

1 **foreach** $\{s_i, s_j\} \in S^2$ **in parallel do**
2 **if** $|\tau(\{s_i, s_j\})| = f$ **then**
3 **foreach** $x \in \Sigma$ **do**
4 **foreach** $\{s'_i, s'_j\}$ *where* $s'_i \in \delta^{-1}(s_i, x)$ *and* $s'_j \in \delta^{-1}(s_j, x)$ **do**
5 **if** $\tau(\{s'_i, s'_j\})$ *is undefined* **then** // $\{s'_i, s'_j\} \in R$
6 $\tau(\{s'_i, s'_j\}) \longleftarrow x\tau(\{s_i, s_j\})$;

Algorithm 9: BFS_step_S2F (parallel)

input : An automaton $A = (S, \Sigma, \delta)$, and a partial PMF τ
output: updated function τ

1 **foreach** $\{s_i, s_j\} \in S^2$ **in parallel do**
2 **if** $\tau(\{s_i, s_j\})$ *is undefined* **then**
3 **foreach** $x \in \Sigma$ **do**
4 $\{s'_i, s'_j\} \longleftarrow \{\delta(s_i, x), \delta(s_j, x)\}$;
5 **if** $\tau(\{s'_i, s'_j\})$ *is defined* **then** // $\{s'_i, s'_j\} \in F$
6 $\tau(\{s_i, s_j\}) \longleftarrow x\tau(\{s'_i, s'_j\})$;
7 **break**;

4.2. Parallel synchronizing sequence construction

The second phase of the algorithm has two major sub-phases which are repetitively applied: (i) finding a pair having the minimum length merging sequence and (ii) applying this merging sequence to the current active state set. The algorithm applies these two sub-phases until the automata is synchronized. To observe the behavior of the second phase, we extended our preliminary experiments and measure the execution times for these sub-phases. Since the second phase takes less than only one second for random automata, only Černý automata with $n \in \{2000, 4000, 8000\}$ states are used for this set of experiments. To reduce the variance on the measured individual execution times, each experiment is repeated 5 times. Table 2 presents the averages of these executions.

The table shows that Algorithm 4 dominates the execution time of the second phase. To parallelize this sub-phase, we make each thread to find a local pair in parallel. Later, these pairs are sequentially merged to obtain a global state pair with a shortest merging sequence. Algorithm 10 presents the pseudocode of this approach.

Algorithm 10: FindMin (parallel)

input : Current set of state C and the PMF function τ
output: A pair of state $\{s_i, s_j\}$ with minimum $|\tau(\{s_i, s_j\})|$ among all pairs in $C^{(2)}$

1 **foreach** *thread* t **do** $\{s_{i_t}, s_{j_t}\} \longleftarrow$ *undefined* ;
2 **foreach** $\{s_k, s_\ell\} \in C^{(2)}$ **in parallel do**
3 **if** $\{s_{i_t}, s_{j_t}\}$ *is undefined or* $|\tau(\{s_k, s_\ell\})| < |\tau(\{s_{i_t}, s_{j_t}\})|$ **then**
4 $\{s_{i_t}, s_{j_t}\} \longleftarrow \{s_k, s_\ell\}$;
5 **if** $|\tau(\{s_k, s_\ell\})| = 1$ **then**
6 **break**;
7 $\{s_i, s_j\} \longleftarrow$ *undefined*;
8 **foreach** *thread* t **do**
9 **if** $\{s_i, s_j\}$ *is undefined or* $|\tau(\{s_{i_t}, s_{j_t}\})| < |\tau(\{s_i, s_j\})|$ **then**
10 $\{s_i, s_j\} \longleftarrow \{s_{i_t}, s_{j_t}\}$;
11 **if** $|\tau(\{s_i, s_j\})| = 1$ **then**
12 **break**;

To parallelize Algorithm 10, we took a simple approach. The index set $\{i : 1 \leq i \leq |C|\}$ is divided into a number of chunks where each chunk is assigned to a single thread. For each index i, the state pairs at locations $\{C[i], C[j]\}$ such that $i < j \leq |C|$ are processed by the same thread and a minimum $|\tau(.)|$ value is computed for these pairs. After all the chunks are processed, with CUDA, each GPU thread performs an `atomicMin` operation to update the global minimum. Instead, on the CPU, we perform a sequential synchronization (as in Lines 8–12).

In our preliminary experiments, we observed that it is not only the balanced distribution that matters; cache utilization is proven to be more important. Let $\{s_i, s_j\}$ be the pair returned by the *FindMin* function in Algorithm 3 (Line 8). The set C is updated by applying $\tau(\{s_i, s_j\})$ to every state in C (Line 10). Later, the duplicates are removed. As the algorithm shows, the new C is fed to the *FindMin* function in the next iteration. Since a path can transform a state to an arbitrary one, the state IDs in this new C are unsorted. This makes a thread access to the distance array in an irregular fashion. Hence, although this incurs an extra overhead, we choose to sort C after each time it is updated. We measured the impact of this preprocessing step in Section 5.

4.3. A better pair automata indexing for manycore architectures

As explained before, the algorithms S2R and S2F visit all the pairs and the threads process a pair if it is in the current frontier (for S2R) or it is a remaining pair (for S2F). In our implementation, the pairs are ordered by using an indexing scheme. With a simple indexing, one can use pair IDs $\{0, 1, \ldots, n^2 - 1\}$ as in Fig. 5 (top) and store the PMF values (i.e., letters) within an array in this order. In this scheme, the ID of a pair $\{s_i, s_j\}$ where

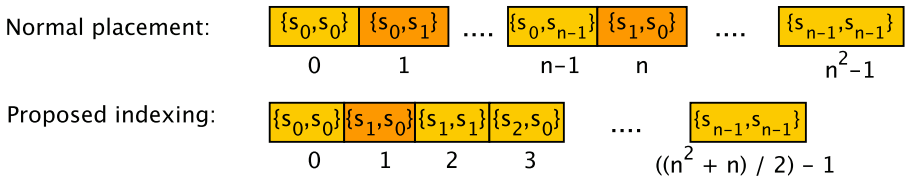


Fig. 5. Indexing and placement of the state pair arrays. A simple placement of the pairs (at the top) uses redundant places for state pairs $\{s_i, s_j\}$, $i \neq j$, e.g., $\{s_0, s_1\}$ and $\{s_1, s_0\}$ in the figure. At the bottom, the indexing mechanism we used is shown.

Table 2
Comparison of the run time of Algorithm 4 ($t_{FindMin}$), i.e., the first sub-phase, and the second phase (t_{Phase_2}).

n	$t_{FindMin}$	t_{Phase_2}	$t_{FindMin}/t_{Phase\_2}$
2000	4.73	4.74	0.997
4000	41.03	41.10	0.998
8000	1035.09	1035.48	1.000

$1 \leq i \leq j \leq n$ is computed as $\ell = (i-1) \times n + j$. Vice versa, given ℓ, one can obtain the IDs of the states by

$$i = \left\lceil \frac{\ell}{n} \right\rceil \quad \text{and} \quad j = \ell - ((i-1) \times n).$$

On a GPU, this scheme yields an inefficient work distribution since when the consecutive threads are assigned consecutive pair IDs, some of the threads in a warp will not work and only wait for the kernel to be finished. This happens since $\{s_j, s_i\}$s with $i < j$ are not actually considered a state pair hence, the IDs for such pairs will incur no work. Even one can preprocess the IDs and eliminate such pairs' IDs, this indexing effectively uses only the half of the array(s); for a state pair $\{s_i, s_j\}$, a redundant entry for $\{s_j, s_i\}$ is also stored. Hence, the threads in some warps will access more than a single block and memory accesses will be slower for these warps.

In this work, we propose to use a better indexing scheme that does not use redundant locations as shown in Fig. 5 (bottom). Given a pair ID

$$\ell = \frac{i \times (i+1)}{2} + j$$

the state IDs are computed in this scheme as

$$i = \lfloor \sqrt{1 + 2\ell} - 0.5 \rfloor \quad \text{and} \quad j = \ell - \frac{i \times (i+1)}{2}.$$

Note that this indexing scheme not only distributes the load work better but also reduces the memory used which is indeed crucial for memory restricted devices such as GPUs.

5. Experimental results

We performed experiments on a server 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). We used OpenMP for multicore parallelism and all the codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled. With OpenMP, we employed the dynamic scheduling policy for PMF construction phase (with batches of 512-pairs) since the task costs are not uniform.

The machine we use has a NVIDIA K40 GPU with 12GB of global memory and 15 SMs each having 192 cores. The manycore GPU parallelization is achieved with CUDA. All the codes are compiled with `CUDA 7.5` and `nvcc` with `-O3` optimization flag. Although there exist studies showing the importance of overlapping the data transfer to GPU with computation, e.g., see [34], for the synchronization problem, the data, i.e., the automata, is small and the data transfers take insignificant time compared to the processing time. Hence, in this study, we did not tune and optimize the overhead of data transfers to the device.

To measure the efficiency of the proposed algorithms, we used randomly generated automata[1] with $n \in \{2000, 4000, 8000, 16000, 32000\}$ states and $p \in \{2, 8, 32, 128\}$ inputs. For each (n, p) pair, we randomly generated 100 different automata and executed each algorithm on them. The values in the figures and the tables are the averages of these 100 executions for each configuration, i.e., algorithm, n and p.

5.1. Parallelization of PMF construction

Fig. 6 shows the speedups of our parallel F2R implementation over the sequential baseline with no parallelism. Both F2R and sequential baseline use the same frontier update mechanism, whereas R2F, S2R and S2F employ a different one. Hence, here we only present the speedup values of F2R to understand the scalability of the proposed algorithm and implementation. As the figure shows, when p is large, parallel F2R presents good speedups, e.g., for $p = 128$, the average speedup is 13.4 with 16 threads. We observed around 10% performance difference between sequential baseline and single-thread F2R due to the extra overhead performed for local queue management. When compared to the single-thread F2R, the average speedup of parallel F2R with $p = 128$ and 16 threads is 14.9. With the same number of threads, for automata with less than 128 inputs, i.e., $p \in \{2, 8, 32\}$, the average speedups are 7.9, 10.4, and 12.7, respectively. The increase on the speedup with p is expected since increasing p only increases the work but not the queue-management overhead on multicore architectures.

Table 3 compares the execution times of F2R, R2F and Hybrid algorithm for $n \in 2000, 4000, 8000$, $p \in \{2, 8, 32, 128\}$ and $\{1, 2, 4, 8, 16\}$ threads. All in all the Hybrid approach is superior to both standalone F2R and standalone R2F. For example, when $n = 8000$ and $p = 128$, the Hybrid algorithm is 38× and 14× faster than F2R and R2F, respectively. For the same automaton set, the speedups due to hybridization of the process become 29× and 16× on average with 16 threads.

Based on the results, F2R is consistently faster than R2F for $p = 2$. However, it is slower otherwise due to the differences in the number of required iterations to construct PMF. The difference on the behavior with different p values can be explained as follows; as Fig. 4 shows, when p is large, the frontier expands quickly and the PMF is constructed in less iterations. Hence, there exist less number of pairs remaining in R and less number of edges to process. Furthermore, the probability of an edge from a fixed remaining pair hitting to a frontier pair increases with the size of the frontier. The impact of these is visible at extreme, i.e., even by a reduction on R2F execution time 3 (observe the change from $p = 2$ to $p = 8$). That being said, 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 $p = 32$).

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

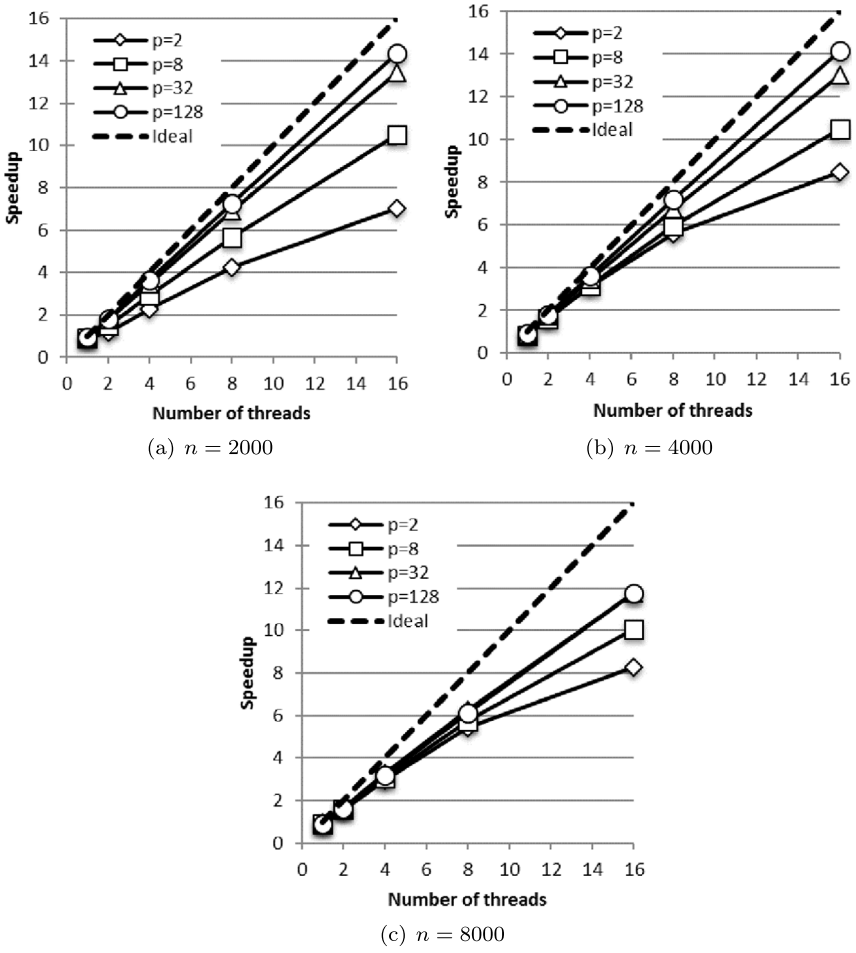


Fig. 6. The speedup of parallel F2R PMF construction over the sequential baseline.

Table 3
Comparison of the parallel execution times of the PMF construction algorithms.

		$n = 2000$				$n = 4000$				$n = 8000$			
	p	2	8	32	128	2	8	32	128	2	8	32	128
	Sequential	0.17	0.50	2.11	9.13	1.18	2.71	9.92	40.36	5.90	14.29	51.78	193.55
1	F2R	0.19	0.56	2.31	9.89	1.35	3.21	11.24	44.57	6.43	16.01	56.71	219.46
	R2F	0.59	0.46	**0.85**	1.91	3.17	2.61	4.72	11.39	19.41	18.17	34.35	86.60
	Hybrid	**0.14**	**0.18**	0.96	**0.65**	**1.06**	**0.89**	**2.99**	**1.92**	**5.16**	**8.42**	**8.93**	**5.80**
2	F2R	0.15	0.34	1.21	5.00	0.73	1.68	5.74	22.41	3.82	9.14	31.40	120.23
	R2F	0.37	0.27	**0.46**	0.98	2.00	1.55	2.62	6.04	13.73	11.96	21.54	52.67
	Hybrid	**0.12**	**0.14**	0.53	**0.37**	**0.58**	**0.50**	**1.57**	**1.01**	**3.09**	**4.86**	**5.10**	**3.34**
4	F2R	0.08	0.17	0.61	2.50	0.38	0.86	2.88	11.11	1.99	4.67	15.79	60.42
	R2F	0.20	0.15	**0.24**	0.50	1.09	0.82	1.36	3.05	7.43	6.43	11.34	27.21
	Hybrid	**0.06**	**0.07**	0.27	**0.19**	**0.31**	**0.26**	**0.80**	**0.52**	**1.62**	**2.49**	**2.61**	**1.73**
8	F2R	0.04	0.09	0.31	1.26	0.21	0.46	1.47	5.60	1.09	2.49	8.31	31.55
	R2F	0.11	0.08	0.12	0.25	0.64	0.45	0.71	1.55	4.36	3.68	6.36	14.91
	Hybrid	**0.03**	**0.04**	**0.14**	**0.10**	**0.17**	**0.15**	**0.42**	**0.28**	**0.89**	**1.34**	**1.39**	**0.93**
16	F2R	**0.02**	0.05	0.16	0.64	0.14	0.26	0.76	2.85	0.71	1.42	4.41	16.50
	R2F	0.06	0.04	0.06	0.13	0.41	0.26	0.38	0.81	2.78	2.38	4.06	9.10
	Hybrid	**0.02**	**0.02**	**0.07**	**0.05**	**0.12**	**0.09**	**0.23**	**0.16**	**0.59**	**0.80**	**0.80**	**0.57**

To understand the performance improvements due to GPU, we run S2R, S2F and Hybrid algorithms on the same set of automata used before. Table 4 presents the results of these experiments. Overall for $n = 8000$ and varying p values, 1.2×–2.1× improvement is obtained on the GPU with traditional indexing compared to 16-thread CPU execution. With the smart indexing scheme proposed in Section 4.3, the improvements increase to 1.5 × −2.9×. Similar to the multicore experiments, for most of the (n, p) tuples, the Hybrid variant combining S2R and S2F is superior to either of the standalone variants on the GPU (see Table 5).

To see the impact of the GPU usage better, we increased the automata sizes to $n = 16\,000$ and $n = 32\,000$ and generated additional random automata. On this automata, we run the multicore implementations with only 16 threads, since the sequential implementation would take too much time. In fact, even the

Table 4
Comparison of the 16 thread CPU and GPU execution times for PMF construction.

		$n = 2000$				$n = 4000$				$n = 8000$			
	p	2	8	32	128	2	8	32	128	2	8	32	128
	Sequential	0.17	0.50	2.11	9.13	1.18	2.71	9.92	40.36	5.90	14.29	51.78	193.55
16	Hybrid	**0.02**	**0.02**	**0.07**	**0.05**	**0.12**	**0.09**	**0.23**	**0.16**	**0.59**	**0.80**	**0.80**	**0.57**
	S2R	**0.02**	**0.02**	0.05	0.17	**0.07**	0.09	0.20	0.77	**0.31**	0.39	0.92	3.36
GPU	S2F	0.03	0.04	0.11	0.46	0.14	0.19	0.61	2.54	0.65	0.91	4.04	13.29
	Hybrid	**0.02**	**0.02**	**0.03**	**0.05**	0.09	**0.07**	**0.12**	**0.14**	0.41	**0.39**	**0.46**	**0.48**
GPU	S2R	**0.01**	0.02	0.04	0.16	**0.05**	0.07	0.19	0.76	**0.21**	0.31	0.85	3.33
Mem.	S2F	0.02	0.02	0.06	0.19	0.08	0.10	0.29	1.16	0.34	0.49	1.74	6.36
Opt.	Hybrid	**0.01**	**0.01**	**0.03**	**0.04**	0.06	**0.05**	**0.10**	**0.13**	0.24	**0.28**	**0.36**	**0.39**

Table 5
Comparison of the 16 thread CPU and GPU execution times for PMF construction on larger automata.

		$n = 16\,000$				$n = 32\,000$			
	p	2	8	32	128	2	8	32	128
	F2R	3.06	6.23	18.85	73.84	13.14	26.06	78.17	386.41
16	R2F	13.85	11.71	20.08	49.00	62.52	52.95	94.29	233.33
	Hybrid	**2.53**	**2.56**	**2.39**	**2.02**	**10.86**	**8.58**	**8.23**	**156.19**
	S2R	**1.45**	1.95	4.76	14.72	**12.76**	23.29	68.26	261.61
GPU	S2F	4.75	6.48	20.59	63.97	71.23	124.56	355.29	1016.56
	Hybrid	2.05	**1.65**	**1.92**	**1.93**	20.26	**13.49**	**13.32**	**117.40**
GPU	S2R	**0.99**	1.51	4.17	13.45	7.38	10.61	21.42	73.19
Mem.	S2F	2.12	2.90	9.78	31.37	11.31	15.49	47.19	142.76
Opt.	Hybrid	1.06	**1.10**	**1.42**	**1.36**	**7.14**	**6.37**	**6.28**	**36.42**

16-thread F2R implementation takes more than 6 min for $n = 32\,000$ and $p = 128$ whereas multicore Hybrid takes 2.5 min. Based on our observations, we predict that the sequential algorithm would take more than an hour. However, for this set of automata, our hybrid GPU implementation takes only 36 seconds. To understand the overall impact based on the sequential execution, Fig. 7 presents the speedup values with the Hybrid variant both on CPU and GPU. As the figure shows, thanks to scaling behavior of Hybrid, the speedups increase when the number of threads increases, especially for large p values. With the CUDA implementation, the PMF generation process becomes even much faster: on average, the one with smart indexing obtains 25×, 51×, 143× and 494× speedups for $p = 2$, 8, 32, and 128 letter automata, respectively, with $n = 8000$ states compared to sequential CPU execution.

5.2. Parallelization of sequence generation

As mentioned in Section 4.2, the execution time of second phase is worthy to take into account for slowly synchronizing automata. In fact, even for a random automaton, after parallelizing Phase 1 with the Hybrid algorithm, the cost of a sequential Phase 2 can become significant. To analyze this statement further, we simply conducted an experiment where the Hybrid GPU approach is used to construct the PMF and no parallelization is applied during the second phase. Table 6 shows the speedups for this experiment. As the results show, when Phase 1 is parallelized, a sequential Phase 2 takes more than half of the time. Hence, even for a random automaton, using parallelism for the second phase will also be beneficial in terms of performance.

We used Černý automata with $n \in \{2000, 4000, 8000\}$ for this set of experiments. For CPU parallelization of the second phase, we implemented Algorithm 10 and tested with 1, 2, 4, 8 and 16 threads. For GPU parallelization, we implemented the same algorithm with 256 threads per block and 256 blocks. We also implemented the approach that sorts the set of current active states before the process as mentioned in Section 4.2.

Table 6
The speedups obtained on Eppstein's Greedy algorithm when the memory optimized CUDA implementation of Hybrid PMF construction algorithm is used.

	Speedup				$\frac{t_{PMF}}{t_{ALL}}$			
n\p	2	8	32	128	2	8	32	128
2000	6.69	18.55	50.60	159.54	0.52	0.53	0.69	0.77
4000	10.94	25.29	61.33	213.27	0.50	0.46	0.62	0.68
8000	9.55	20.66	64.28	234.91	0.36	0.39	0.45	0.47

Table 7
The execution times (in s) of Algorithm 4 (the first two rows) and Algorithm 10 (the other rows). The first of each row pair, i.e., the rows labeled with *unsorted*, uses the active state set C as is, whereas the rows labeled with *sorted* order the states in C w.r.t. increasing state ids as a preprocessing step in every iteration as explained in Section 4.2.

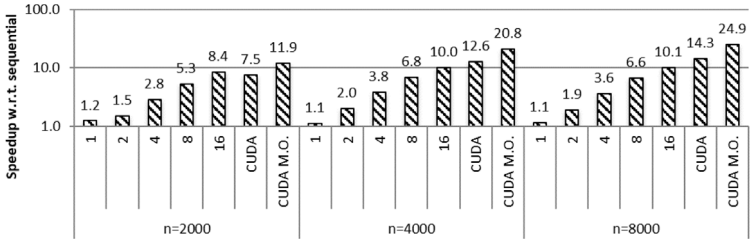
	n	2000	4000	8000
Sequential	Unsorted	4.73	41.03	1035.10
	Sorted	1.60	12.70	109.76
1 thread	Unsorted	5.10	47.02	896.38
	Sorted	2.55	20.27	168.12
2 threads	Unsorted	3.87	37.35	874.25
	Sorted	1.94	15.09	132.77
4 threads	Unsorted	2.31	22.71	522.93
	Sorted	1.18	8.95	75.67
8 threads	Unsorted	1.26	13.13	289.75
	Sorted	0.72	5.04	40.84
16 threads	Unsorted	0.69	6.67	154.80
	Sorted	0.72	3.35	22.40
GPU	Unsorted	0.68	5.51	51.28
	Sorted	0.39	1.56	9.61

Table 7 shows that sorting the set of current pairs has a remarkable impact on the performance. Even in the sequential implementation, we observed 3× to 9.5× speedups. When both the implementation improvement and GPU parallelization is applied, we observed between 12× and 107× speedups over the sequential implementation of the second phase.

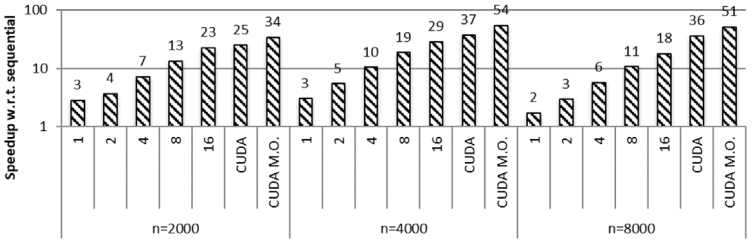
6. Conclusion and future work

We investigated the efficient implementation and the use of modern multicore CPUs to scale the performance of synchronizing sequence generation heuristics. We parallelized one of the well-known heuristics Greedy. We focused on both the PMF generation phase (which is employed by almost all the heuristics in the literature) and the second phase where the synchronizing sequence is generated. For instance, for a random automaton with $n = 8000$ and $p = 128$, the proposed approach for the first phase improves the naive, sequential implementation by 33× and with 16 threads, the speedup increases to 340×. Furthermore, with a single GPU, we obtain 496× speedup over the sequential implementation. We also propose a naive parallelization with a

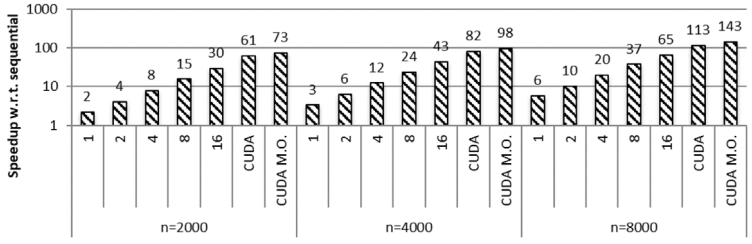
Please cite this article as: S. Karahoda, O.T. Erenay, K. Kaya et al., Multicore and manycore parallelization of cheap synchronizing sequence heuristics, Journal of Parallel and Distributed Computing (2020), https://doi.org/10.1016/j.jpdc.2020.02.009.



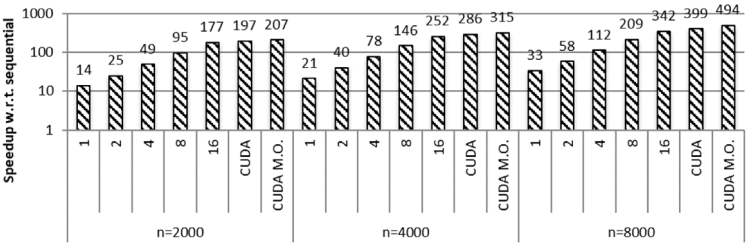
(a) $p = 2$



(b) $p = 8$



(c) $p = 32$



(d) $p = 128$

Fig. 7. The speedups of the Hybrid PMF construction with $p = \{2, 8, 32, 128\}$ and $n \in \{2000, 4000, 8000\}$ on CPU and GPU. The x-axis shows the number of threads used for Hybrid. The values are computed based on the average sequential PMF construction time over 100 different automata for each (n, p) pair.

good spatial locality for the second phase which is shown to be useful especially for slowly synchronizing automata.

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 the PMF generation phase may not be as efficient as this work since the synchronizing sequence construction part of these heuristics are much more expensive compared to Greedy.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2020.02.009.

Acknowledgments

This work is supported by TÜBİTAK Grant #114E569. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

References

[1] D.S. Ananichev, M.V. Volkov, Synchronizing monotonic automata, Theoret. Comput. Sci. 327 (3) (2004) 225–239.

[2] S. Beamer, K. Asanović, D. Patterson, Direction-optimizing breadth-first search, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 12:1–12:10, URL http://dl.acm.org/citation.cfm?id=2388996.2389013.

[3] Y. Benenson, T. Paz-Elizur, A. Rivka, E. Keinan, Z. Livneh, E. Shapiro, Programmable and autonomous computing machine made of biomolecules, Nature 414 (6862) (2001) 430–434, http://dx.doi.org/10.1038/35106533.

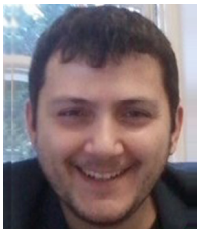
[4] M.V. Berlinkov, On the probability of being synchronizable, in: CALDAM, in: Lecture Notes in Computer Science, vol. 9602, Springer, 2016, pp. 73–84.

[5] A.L. Bonifácio, A.V. Moura, A. da Silva Simão, Experimental comparison of approaches for checking completeness of test suites from finite state machines, Inf. Softw. Technol. 92 (2017) 95–104.
[6] R. Boute, Distinguishing sets for optimal state identification in checking experiments, IEEE Trans. Comput. 23 (8) (1974) 874–877, http://doi.ieeecomputersociety.org/10.1109/T-C.1974.224043.
[7] J. Černý, A note on homogeneous experiments with finite automata English, Mat.-fyz. cas. 14 (1964) 208–216.
[8] J. Černỳ, Poznámka k homogénnym experimentom s konečnỳmi automatmi, Mat. fyz. cas. 14 (3) (1964) 208–216.
[9] J. Černỳ, A. Pirická, B. Rosenauerová, On directable automata, Kybernetika 7 (4) (1971) 289–298.
[10] T.S. Chow, Testing software design modeled by finite-state machines, IEEE Trans. Softw. Eng. 4 (3) (1978) 178–187.
[11] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, N. Yevtushenko, FSM-based conformance testing methods: A survey annotated with experimental evaluation, Inf. Softw. Technol. 52 (12) (2010) 1286–1297.
[12] D. Eppstein, Reset sequences for monotonic automata, SIAM J. Comput. 19 (3) (1990) 500–510.
[13] R. Groz, A.S. Simão, A. Petrenko, C. Oriat, Inferring finite state machines without reset using state identification sequences, in: ICTSS, in: Lecture Notes in Computer Science, vol. 9447, Springer, 2015, pp. 161–177.
[14] F.C. Hennie, Fault-detecting experiments for sequential circuits, in: Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, New Jersey, 1964, pp. 95–110.
[15] R.M. Hierons, H. Ural, Reduced length checking sequences, IEEE Trans. Comput. 51 (9) (2002) 1111–1117.
[16] R.M. Hierons, H. Ural, UIO sequence based checking sequences for distributed test architectures, Inf. Softw. Technol. 45 (12) (2003) 793–803.
[17] R.M. Hierons, H. Ural, Generating a checking sequence with a minimum number of reset transitions, Autom. Softw. Eng. 17 (3) (2010) 217–250.
[18] G.V. Jourdan, H. Ural, H. Yenigun, Reduced checking sequences using unreliable reset, Inform. Process. Lett. 115 (5) (2015) 532–535, http://dx.doi.org/10.1016/j.ipl.2015.01.002.
[19] R. Kudlacik, A. Roman, H. Wagner, Effective synchronizing algorithms, Expert Syst. Appl. 39 (14) (2012) 11746–11757.
[20] K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2795–2808, http://dx.doi.org/10.1109/TPDS.2016.2516988.
[21] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, third ed., John Wiley and Sons, 2011.
[22] B.K. Natarajan, An algorithmic approach to the automated design of parts orienters, in: FOCS, 1986, pp. 132–142.
[23] A. Petrenko, N. Yevtushenko, Testing from partial deterministic FSM specifications, IEEE Trans. Comput. 54 (9) (2005) 1154–1165.
[24] H. Robert M., Minimizing the number of resets when testing from a finite state machine, Inform. Process. Lett. 90 (6) (2004) 287–292.
[25] A. Roman, Synchronizing finite automata with short reset words, Appl. Math. Comput. 209 (1) (2009) 125–136.
[26] A. Roman, M. Szykula, Forward and backward synchronizing algorithms, Expert Syst. Appl. 42 (24) (2015) 9512–9527.
[27] P. Schrammel, T. Melham, D. Kroening, Chaining test cases for reactive system testing, in: ICTSS, in: Lecture Notes in Computer Science, vol. 8254, Springer, 2013, pp. 133–148.
[28] A.S. Simão, A. Petrenko, N. Yevtushenko, On reducing test length for FSMs with extra states, Softw. Test. Verif. Reliab. 22 (6) (2012) 435–454.
[29] A.N. Trahtman, Some results of implemented algorithms of synchronization, in: 10th Journees Montoises D'Inform, 2004.
[30] U.C. Türker, T. Ünlüyurt, H. Yenigün, Effective algorithms for constructing minimum cost adaptive distinguishing sequences, Inf. Softw. Technol. 74 (2016) 69–85.
[31] U.C. Türker, H. Yenigün, Complexities of some problems related to synchronizing, non-synchronizing and monotonic automata, Internat. J. Found Comput. Sci. 26 (01) (2015) 99–121, http://dx.doi.org/10.1142/s0129054115500057.
[32] H. Ural, X. Wu, F. Zhang, On minimizing the lengths of checking sequences, IEEE Trans. Comput. 46 (1) (1997) 93–99.
[33] M. Vasilevskii, Failure diagnosis of automata, Cybernetics 9 (4) (1973) 653–665, http://dx.doi.org/10.1007/BF01068590.
[34] W. Yang, K. Li, K. Li, A hybrid computing method of SpMV on CPU-GPU heterogeneous computing systems, J. Parallel Distrib. Comput. 104 (2017) 49–60, http://dx.doi.org/10.1016/j.jpdc.2016.12.023.

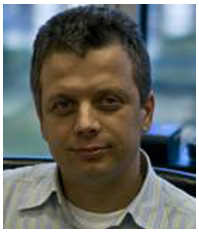
Sertaç Karahoda is received his BSc and MSc degrees in 2015 and 2018 from Sabancı University, Computer Science and Engineering.

Osman Tufan Erenay is received his BSc degree in 2016 from Sabancı University, Computer Science and Engineering.

Kamer Kaya is an Assistant Professor at the Faculty of Engineering and Natural Sciences at Sabancı University. He got his Ph.D. from Dept. of Computer Engineering at Bilkent University in 2009. His current research interests include Parallel Programming, High Performance Computing, and Cryptography.



Uraz Cengiz Türker is a Senior Lecturer (Assistant Professor) in the Department of Informatics at the University of Leicester. He received BA, MSc and PhD degrees in Computer Science (Sabanci University, Turkey), in 2006, 2008, and 2014, respectively. His research interests include model based testing, automata theory, theory of computation, intelligance testing of distributed cyber-physical systems.



Hüsnü Yenigun received his Ph.D. degrees from Electrical and Electronics Engineering Department of Middle East Technical University (Ankara) in 2000. Since 2001, he is a faculty member at Computer Science and Engineering Program of Faculty of Engineering and Natural Sciences at Sabanci University (Istanbul). His research interests are automata and concurrency theory, formal methods, software quality assurance, testing, model checking, complexity relief techniques for practical software verification.