Boosting Expensive Synchronizing Heuristics

N. Ege Saraç[a], Ömer Faruk Altun[b], Kamil Tolga Atam[b], Sertaç Karahoda[b], Kamer Kaya[b,*], Hüsnü Yenigün[b]

[a] *IST Austria, Klosterneuburg, Austria*
[b] *Computer Science and Engineering, Faculty of Engineering and Natural Sciences, Sabanci University, Tuzla, Istanbul, Turkey*

Abstract

For automata, synchronization, the problem of bringing an automaton to a particular state regardless of its initial state, is important. It has several applications in practice and is related to a fifty-year-old conjecture on the length of the shortest synchronizing word. Although using shorter words increases the effectiveness in practice, finding a shortest one (which is not necessarily unique) is NP-hard. For this reason, there exist various heuristics in the literature. However, high-quality heuristics such as SynchroP producing relatively shorter sequences are very expensive and can take hours when the automaton has tens of thousands of states. The SynchroP heuristic has been frequently used as a benchmark to evaluate the performance of the new heuristics. In this work, we first improve the runtime of SynchroP and its variants by using algorithmic techniques. We then focus on adapting SynchroP for many-core architectures, and overall, we obtain more than 1000× speedup on GPUs compared to naive sequential implementation that has been frequently used as a benchmark to evaluate new heuristics in the literature. We also propose two SynchroP variants and evaluate their performance.

*Corresponding author

**A preliminary version appeared in Altun et al. (2017) Synchronizing Heuristics: Speeding up the Slowest. ICTSS 2017. Lecture Notes in Computer Science, vol 10533. Springer, Cham

Email addresses: ege.sarac@ist.ac.at (N. Ege Saraç), ofarukaltun (Ömer Faruk Altun), atam (Kamil Tolga Atam), skarahoda (Sertaç Karahoda), kaya (Kamer Kaya), yenigun@sabanciuniv.edu (Hüsnü Yenigün)

Keywords: Synchronizing heuristics; parallel algorithms; GPU programming

1. Introduction

Given an automaton A, a *synchronizing word* w is an input sequence such that when applied to the automaton, it brings A to a particular state no matter at which state A currently is. If such a word exists for A it is called a *synchronizing automaton*. Otherwise, it is not synchronizing.

There exist several applications of synchronizing words in practice. For instance, in model- and finite-state-machine-based (FSM) testing (Broy et al., 2005; Lee & Yannakakis, 1996), the tests usually require a particular initial state to be started. When modeled as an automaton, the implementation, i.e. the system under test, must be properly initialized, which can be done by applying a synchronizing word w. Hence, w is used to prepare the system for a test. Also, synchronizing words are employed for test-case generation that can synchronize circuits without having a reset feature (Cho et al., 1993) They are also used as compound reset operations when resetting a circuit is too expensive (Jourdan et al., 2015). As another application, synchronizing words can be stored inside a tamper-proof region of an autonomous mobile device. Such a device can be an autonomous car or a robot in a factory; once the internal system is modeled as an FSM, the device can reset itself even when it is remotely hacked, all the sensors are out of order, or whatever state it is in. Natarajan (1986) puts forward part orienters as another application; a part on the conveyor belt must be put into a particular orientation by the placed obstacles. The initial orientation is assumed to be unknown, and the obstacles must perform the task regardless of the initial orientation.

It is easy to see that, if w is a synchronizing word for an automaton A, then any word for which w is a subword is also a synchronizing word for A. Therefore, for a synchronizing automaton, there exist many (in fact, infinitely many) synchronizing words. However, having and using a shortest one (which is not necessarily unique) is more effective and efficient. In all the examples above,

using a shortest word is of interest for practical reasons, such as shorter tests, less energy usage, or less number of obstacles for part orienters. For more examples, theoretical results, and practical use-cases on synchronizing automata, we refer the reader to (Volkov, 2008).

Checking if an automaton with n states and p inputs is synchronizing can be done in polynomial $O(pn^2)$ time (Eppstein, 1990). However, finding a shortest synchronizing word (or finding the shortest synchronizing word length) is NP-hard (Eppstein, 1990), and coNP-hard (Olschewski & Ummels, 2010). Černý conjectured that for a synchronizing automaton A, the length of a shortest synchronizing word is not longer than $(n-1)^2$ (Černỳ, 1964; Černỳ et al., 1971). Posed more than 50 years ago, the Černý conjecture is still open. The best upper bound known for a shortest synchronizing word length is $114n^3/685 + O(n^2)$ (Szykula, 2018).

Although a shortest synchronizing word is hard to find, there exist *synchronizing heuristics* in the literature to compute relatively short words. For an n state, p input automaton, the fastest heuristics Greedy and Cycle have time complexity $O(n^3 + pn^2)$. Some other heuristics are FastSynchro and SynchroP/SynchroPL (Roman, 2009; Kudlacik et al., 2012); the former has time complexity $O(pn^4)$ and the latter heuristics have time complexity $O(n^5 + pn^2)$ The actual performance of the heuristics is in concordant with their theoretical order; see (Kudlacik et al., 2012; Roman & Szykula, 2015) for experimental performance comparison. As expected, the heuristics SynchroP/SynchroPL produce much shorter sequence compared to Greedy/Cycle.

SynchroP and its variants, e.g., SynchroPL, have been frequently used as a baseline to benchmark new heuristics (e.g., see Kudlacik et al. (2012); Kowalski & Szykula (2013); Roman & Szykula (2015)). However, only a set of small-scale automata have been used for comparison purposes since these heuristics are slow. There exist attempts in the literature to solve this performance problem. For instance, a much faster SynchroP variant, FastSynchro, chooses the paths to follow in a greedier, and hence faster, manner to generate synchronizing words. Nonetheless, the improvement also increases the length of

the synchronizing words found (Roman, 2009; Kudlacik et al., 2012). In addition to these, the only parallelization attempt for synchronizing heuristics are for Greedy and Cycle (Karahoda et al., 2016, 2020). To the best of our knowledge, the parallelization of the slower heuristics, which produce shorter synchronizing words, have not been investigated before.

In this work, we modify SynchroP/SynchroPL by using algorithmic techniques to avoid unnecessary steps. Second, we focus on adapting SynchroP-like heuristics for many-core architectures. We carefully analyze the details of the structure of the heuristic, modify and parallelize the necessary steps, and propose various implementations on GPUs. Overall, we obtain more than 1000× speedup on two different GPU architectures compared to naive sequential implementation that has been frequently used to benchmark new heuristics. Third and last, we propose two additional SynchroP variants and evaluate their performance in terms of execution time and synchronizing word length.

The rest of the paper is organized as follows: In Section 2, we introduce the notation used in the paper, explain SynchroP, SynchroPL and many-core architectures in detail. The proposed GPU-based parallelization is described in Section 3 and additional structural improvements on SynchroP are given in Section 4. Section 5 will put forth the two new variants of SynchroP, and experimental results are given in Section 6. Threats to validity are discussed in Section 7. Finally, Section 8 concludes the paper.

2. Background and Notation

In the rest of the paper, the triple $A = (S, \Sigma, \delta)$ denotes a complete and deterministic *automaton* where $S = \{0, 1, \ldots, n-1\}$ is a finite set of n states, Σ is a finite alphabet consisting of p input letters (or simply *letters*), and $\delta : S \times \Sigma \to S$ is a transition function. When δ is a total function, i.e. when $\delta(i, x)$ is defined for every state $i \in S$ and for every letter $x \in \Sigma$, A is called *complete*. In this paper, we consider only complete automata.

An element of the set $\Sigma^\star$ is called a *word* or a *sequence* (we use "word" and

"sequence" interchangeably). For a word $w \in \Sigma^\star$, we use $|w|$ to denote the length of w, and ε is the empty word. We extend the transition function δ to a set of states and to a word in the usual way. We have $\delta(i, \varepsilon) = i$, and for a word $w \in \Sigma^\star$ and a letter $x \in \Sigma$, we have $\delta(i, xw) = \delta(\delta(i, x), w)$. For a set of states $C \subseteq S$, we have $\delta(C, w) = \{\delta(i, w) | i \in C\}$.

For a set of states $C \subseteq S$, let $C^2 = \{\langle i, j\rangle \mid i, j \in C\}$ be the set of all *unordered pairs* of elements of C. An element $\langle i, j\rangle \in C^2$ is called a *pair*. Furthermore, it is called *a singleton pair* (or *an s–pair*) if $i = j$, otherwise it is called *a different pair* (or *a d–pair*). The set of s–pairs and d–pairs in C^2 are denoted by C^2_s and C^2_d, respectively.

A word w is said to *merge* a pair $\langle i, j\rangle \in S^2$ if $\delta(\{i, j\}, w)$ is singleton. A word that merges a pair $\langle i, j\rangle$ is called a *merging word for* $\langle i, j\rangle$. For an s-pair $\langle i, i\rangle$, each word, including ε, is a *merging word*. A word w is said to *synchronize* an automaton $A = (S, \Sigma, \delta)$ if $\delta(S, w)$ is singleton. A word that synchronizes an automaton A is said to be a *synchronizing word for* A. If there exists a *synchronizing word* for an automaton A then A is called *synchronizing*. Deciding if an automaton is synchronizing can be done in $O(pn^2)$ time (Eppstein, 1990). In this paper, we consider only synchronizing automata.

In the rest of the paper, $\delta^{-1}(i, x)$ denotes the set of states which transition to state i when $x \in \Sigma$ is applied. Formally, $\delta^{-1}(i, x) = \{j \in S \mid \delta(j, x) = i\}$. A similar notation $\delta^{-1}(\langle i, j\rangle, x) = \{\langle k, \ell\rangle \mid k \in \delta^{-1}(i, x) \wedge \ell \in \delta^{-1}(j, x)\}$ is also employed for pairs.

2.1. The SynchroP *Heuristic*

The heuristic we focus in this work, SynchroP, has two phases: The first phase generates a shortest merging word $\tau_{\langle i,j\rangle}$ for each $\langle i, j\rangle \in S^2$ as in Algorithm 1. This is achieved by using a Breadth-first Search (BFS) on a larger automaton A^2, called *pair automaton* in the literature. Formally, given an automaton $A = (S, \Sigma, \delta)$, the *pair automaton* $A^2 = (S^2, \Sigma, \delta^2)$ of A is an automaton where the states of A^2 are pairs of states of A. The automaton A^2 has the same set Σ of input letters as A. The transition function δ^2 of A^2 is defined

as:

$$\text{for all } \langle i,j\rangle \in S^2, x \in \Sigma : \delta^2(\langle i,j\rangle, x) = \langle \delta(i,x), \delta(j,x)\rangle$$

Based on this definition of the pair automaton, it is easy to see that if for a word $w \in \Sigma^\star$ and for a state $\langle i,j\rangle \in S^2$ of A^2, we have $\delta^2(\langle i,j\rangle, w) = \langle k,k\rangle$ for some $\langle k,k\rangle \in S^2$, then w is a merging word for the pair $\langle i,j\rangle$. Hence, the shortest merging sequences of pairs of states of A can be found by performing a BFS on A^2.

Algorithm 1 computes shortest merging words of pairs of states of a given automaton A, by using a single BFS applied in a backward manner using δ^{-1}. During this search, for every pair $\langle i,j\rangle \in S^2$, a shortest path from the state $\langle i,j\rangle$ of A^2 to a state $\langle k,k\rangle$ of A^2, corresponding to a singleton pair, is constructed.

Algorithm 1: Computing shortest merging words for pairs (Phase 1)

input : An automaton $A = (S, \Sigma, \delta)$
output: A shortest merging word $\tau_{\langle i,j\rangle}$ for all $\langle i,j\rangle \in S^2$

```
Q = ∅ // An empty queue for BFS frontier
P = ∅ // The set of pairs discovered so far
foreach ⟨i,i⟩ ∈ S_s^2 do
    push ⟨i,i⟩ onto Q
    insert ⟨i,i⟩ into P
    set τ_⟨i,i⟩ = ε
while P ≠ S^2 do // not all pairs are discovered yet
    ⟨i,j⟩ = pop element from Q
    foreach x ∈ Σ do
        foreach ⟨k,ℓ⟩ ∈ δ^{-1}(⟨i,j⟩, x) do
            if ⟨k,ℓ⟩ ∉ P then
                τ_⟨k,ℓ⟩ = x τ_⟨i,j⟩
                push ⟨k,ℓ⟩ onto Q
                P = P ∪ {⟨k,ℓ⟩}
```

Algorithm 1 constructs a BFS forest, rooted at s–pairs $\langle i,i\rangle \in S_s^2$, where these s–pair nodes are the nodes at level 0 of the BFS forest. A d–pair $\langle i,j\rangle$ appears at level k of the BFS forest if $|\tau_{\langle i,j\rangle}| = k$.

The second phase of SynchroP is given in Algorithm 2. For an automaton $A = (S, \Sigma, \delta)$, Algorithm 2 generates a synchronizing word $\Gamma \in \Sigma^\star$ in a con-

structive, step-by-step fashion. Γ is initialized to the empty sequence (line 2) and extended by appending a sequence (line 10) in every iteration of the algorithm. For $\Gamma \in \Sigma^\star$ accumulated so far during the execution of Algorithm 2, the set of states $C = \delta(S, \Gamma)$, called *the current active state set*, is tracked. This is handled by initializing the current active state set C as S (line 1), since initially, we have $\Gamma = \varepsilon$. In addition, C is updated at line 11, by using the sequence τ' which extends Γ in the current iteration. When the algorithm terminates we have $|C| = 1$, which means $|\delta(S, \Gamma)| = 1$ (since $C = \delta(S, \Gamma)$), and hence Γ is a synchronizing word for A.

Algorithm 2 exploits the fact that $\tau_{\langle i,j \rangle}$ is a merging word for i and j. During the execution of the algorithm, when we have a current active set C such that $|C| > 1$, the algorithm chooses a d–pair $\langle i, j \rangle \in C_d^2$ (lines 5 through 9), and the merging sequence $\tau_{\langle i,j \rangle}$ of the chosen d–pair $\langle i, j \rangle$ is applied to C at line 11. Since states $i, j \in C$ are merged by $\tau_{\langle i,j \rangle}$, we always have $|\delta(C, \tau_{\langle i,j \rangle})|$ strictly smaller than $|C|$ for every iteration of the algorithm. This ensures a reduction on the cardinality of the active state set $|C|$ at every iteration. Thus, the algorithm constructs a synchronizing word if A is synchronizing. The variants of the algorithm differ by picking the d–pair $\langle i, j \rangle \in C_d^2$ in a different way at each iteration.

For a set of states $C \subseteq S$, let the *cost* $\phi(C)$ of C be defined as

$$\phi(C) = \sum_{i,j \in C} |\tau_{\langle i,j \rangle}|$$

where $\phi(C)$ is an estimation on the hardness of bringing C to a singleton. It is assumed that when the estimation $\phi(C)$ is larger, the expected length of a random synchronizing word for C is longer. In SynchroP, $\langle i, j \rangle \in C_d^2$ is selected by favoring the pair with the minimum cost $\phi(\delta(C, \tau_{\langle i,j \rangle}))$. Algorithm 2 presents the second phase of SynchroP based on this idea.

2.2. Memory and Core Structure of GPUs

GPUs are devices that are built for a vast amount of parallelism. Hence, they can simultaneously run many numbers of *threads*. In CUDA, a *warp* contains

Algorithm 2: Computing a synchronizing word (Phase 2)

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j \rangle}$ for all $\langle i, j \rangle \in S^2$
output: A synchronizing word Γ for A

```
1  C = S; // C: current active state set
2  Γ = ε; // Γ: synchronizing word to be constructed, initially empty
3  while |C| > 1 do // still not a singleton
4      minCost = ∞
5      foreach d–pair ⟨i, j⟩ ∈ C_d^2 do
6          thisPairCost = φ(δ(C, τ_⟨i,j⟩))
7          if thisPairCost < minCost then
8              minCost = thisPairCost
9              τ' = τ_⟨i,j⟩
10     Γ = Γ τ'; // append τ' to the synchronizing word
11     C = δ(C, τ'); // update current active state set with τ'
```

several adjacent threads that can run simultaneously on a streaming multiprocessor (abbrv. SM). The warp size is currently determined as 32 threads. The programmer decides some number of threads to be in a group called *block*, which runs on SMs. A collection of blocks is called a *grid*. All threads running at any given time are obliged to share some resources, one of which is the memory of different hierarchies.

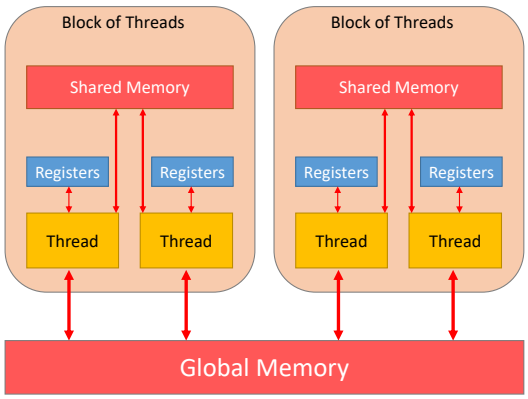

Figure 1: The basic organization of cores and memory of NVIDIA GPUs.

Modern GPUs possess thousands of small cores that are distributed over multiple SMs. On each device, there exist a large *global memory* shared by all the cores/SMs on the GPU. However, it is relatively slow; it takes hundreds of cycles to retrieve data from global memory. Threads of the same block share

faster but smaller means of memory regions called *shared memory*. Although it is usually capable of storing data in the order of KBs, threads can acquire data located in shared memory only in several cycles. Last, each thread has its own set of registers which are the fastest and the most scarce memory units available to a thread.

3. Speeding up SynchroP and SynchroPL on GPU

In this section, we will introduce a from-scratch implementation of the heuristics on a GPU and several performance improvements applied to this very first version. Section 3.1 describes the initial GPU implementation. In Section 3.2, we propose an improved memory access scheme on top of the initial implementation. Section 3.3 introduces a technique for better load-balancing. Taking up from this point, Section 3.4 proposes an additional improvement, obtained by keeping the current active state set sorted.

3.1. Naive Implementation

When each pair in the active set C is assigned to a single thread on the GPU, parallel SynchroP works with a large amount of data that well exceeds the order of KBs, by which the shared memory blocks are limited. Hence, almost all the auxiliary data, e.g., path, Γ, and distance information, $|\tau_{\langle i,j\rangle}|$, of all active pairs, are stored in global memory.

If Algorithm 2 is analyzed carefully, one can observe that from line 5 up to line 9, the cost $\phi(\delta(C, \tau_{\langle i,j\rangle}))$ for each d–pair $\langle i, j\rangle \in C_d^2$ is calculated and the sequence with the minimum cost is extracted. This part takes a significant portion of the runtime. Furthermore, for each d–pair $\langle i, j\rangle$, the cost calculation can be run independently for all pairs and hence, is a feasible candidate for effective and scalable parallelization. Our first implementation assigns a single d–pair $\langle i, j\rangle$ to a single thread to compute its SynchroP cost. With this fine-grain approach, all available cores in the GPU are assigned independent tasks and utilized at the same time. While applying the path to the current active set, to decide on the inclusion of any state in the next active state set, we employ a

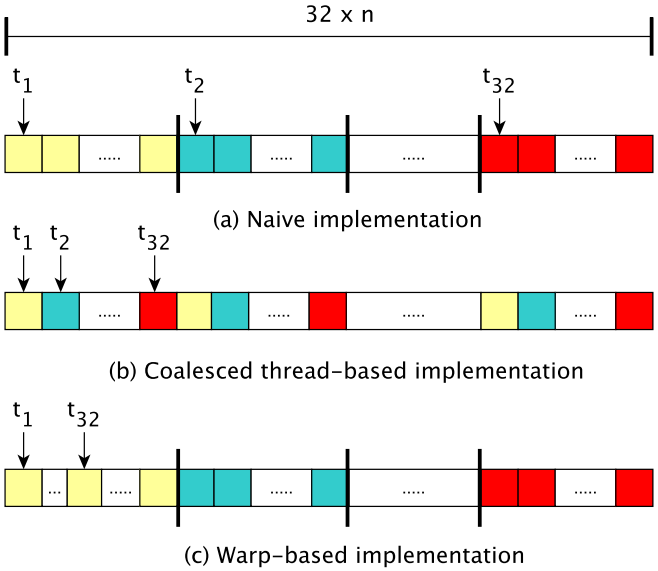

Figure 2: Memory access patterns during Step 1 for different GPU implementations for $n = 32$. The first two subfigures ((a) and (b)) are for thread-based implementations whereas the subfigure (c) shows the memory access pattern for warp-based implementation.

per-thread *marker* array of size n to mark the indices of the next active states found. The steps of the operations regarding a single d–pair $\langle i, j \rangle$ are given as follows in order to be referenced for upcoming improvements:

- **Step 1:** for each $s \in C$, $\tau_{\langle i,j \rangle}$ is applied to s and stored in a local array C',
- **Step 2:** the value of the cost function is accumulated by traversing each d–pair $\langle i', j' \rangle \in C'^2_d$,
- **Step 3:** the cost value is stored in the shared memory.

3.2. Coalesced Memory Access to Active Set

Although assigning threads to pairs is arguably intuitive, it inhibits the overall performance: the memory locations for the active state sets accessed by the consecutive threads are located at least n positions apart; hence, the naive

implementation suffers from poor memory coalescing in Step 1 (Fig. 2 (a)). To improve the memory access performance in Step 1, instead of providing each thread a continuous block of space, we rearrange the positions of the arrays and order them with respect to the warp and state IDs instead of thread IDs. That is, we consider all the active state-set arrays used by a single warp as a single memory block: a warp has 32 threads, and the first 32 locations are used for the first state by each thread, respectively, then the next 32 locations are used for the second state, and so on (Fig. 2 (b)). This approach yields a coalesced pattern of memory accesses as opposed to the strided one in the previous implementation (Fig. 2 (a)).

3.3. Better Automata Accesses and Load Balancing

In thread-based implementations, each thread processes a single d–pair at once, hence a different sequence. These sequences can differ from each other in many ways. In GPUs, all 32 threads in the same warp are controlled by the same control unit; hence they are inherently synchronized. For any computation with load imbalance, the least-loaded thread(s) (31 threads in the worst case) must wait for the most loaded one. Hence in SynchroP, the fluctuation among the merging sequence lengths in Step 1 frequently results in many of the threads becoming idle. Even if their lengths are the same, when the sequences are different, the size of the next active set can be different which can also incur an imbalance during cost computation in Step 2.

Another possible performance drawback, when different sequences are processed in the same warp, is automata accesses. Different sequences have different input letters in the same location. We keep the automata in input-wise order, i.e., first, n target states of the first input are stored, then the second and the third one, and so on, are stored consecutively in memory. Hence, when the input letters are different for two threads in the same warp, they will access a different block of length n. This reduces the chance of having the locations in the same memory block.

We target these problems and propose the warp-based method in which a

sequence is assigned to a single warp instead of a single thread. In this approach, each thread in a warp applies the same sequence to a different state in C and keep its final state in C'. By making the threads work on the same sequence, a better load balancing is obtained, and the inefficiency due to the line-synchronization is avoided. In terms of load balancing, the warp-based approach is expected to be more efficient when $|C| \gg 32$. However, when $|C|$ is small, the thread-based approaches tend to be better to balance the load. Fortunately, the first SynchroP iteration, which is significantly more expensive than the rest of the iterations, has $C = S$, i.e., $|C| = n$. That is, the warp-based approach is already a better alternative for the most dominating part of the computation. This approach also reduces the number of memory blocks accessed at once, i.e., increases the chance of having coalesced memory accesses, since all the threads (of the same warp) accesses the same length n block of the automata. Furthermore, the memory access to C can be kept as coalesced as shown in Fig. 2 (c).

3.4. Improved Memory Accesses with Sorting the Active Set

Before performing the cost computation, Step 2 requires a preprocessing: when the corresponding sequence is applied, some active states end up in the same state, which results in repeating occurrences of the final state. With a single thread, one can sequentially traverse the array C'_{before} once and eliminate all these multiple occurrences with an auxiliary marker array of size n. For example, the array

$$C'_{before} = [2, 10, 2, 11, 11, 1, 10, 12]$$

can be easily transformed into

$$C'_{after} = [2, 10, 11, 1, 12]$$

after the preprocessing, which is consistent with the order of appearance in the initial array C'. Even with a single-thread execution, the complexity of removing multiple occurrences is $\Theta(|C'_{before}|)$, whereas the later cost computation

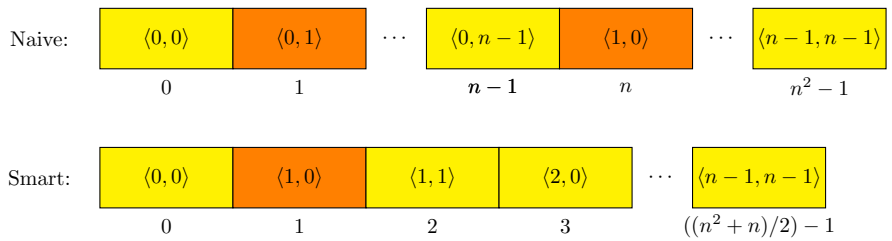

Figure 3: A better placement of the distance/letter information used in Step 2. For a pair $\langle i, j\rangle$ with $j < n-1$ in the naive approach, the next pair is $\langle i, j+1\rangle$. When $j = n-1$, the next pair is $\langle i+1, 0\rangle$. This approach allocates two memory locations for each unordered pair. The smart approach, which is proposed in Karahoda et al. (2018), removes these redundancies with a better indexing scheme.

has complexity $\Theta(|C'_{after}|^2)$. Hence, the preprocessing is significantly cheaper compared to the overall Step 2 cost and is not expected to incur a significant overhead. However, as we will explain later in this section, the unsorted order in C'_{after} hurts the performance while computing the sequence cost.

While computing the cost, the threads frequently access the array storing the lengths of the pairwise merging sequences. Each value is read and immediately added to the cost; so the cost computation is heavily memory bound. This is why the memory access pattern to this array, as well as its organization, has a significant impact on the performance. A naive organization of the length array uses pair IDs $\{0, 1, \ldots, n^2-1\}$ as in Fig. 3 (top) and stores the lengths in this order. In this scheme, the ID of a pair $\langle i, j\rangle$ where $0 \le i \le j \le n-1$ is computed as $\ell = i \times n + j$. Vice versa, given ℓ, one can easily obtain the IDs of the states by the following equations:

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

However, this scheme has redundancies, since it allocates two memory locations for each pair.

In this work we use a better indexing scheme from Karahoda et al. (2018) that does not use redundant locations, as shown in Fig. 3 (bottom). In this approach, pairs are considered for $0 \le j \le i \le n-1$. The ID of a pair $\langle i, j\rangle$ is

computed as follows:

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

Conversely, given a pair ID ℓ, the state IDs are computed in this scheme by using the following equations:

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

This indexing scheme reduces the memory used which is indeed crucial for memory restricted devices such as GPUs. Furthermore, such a scheme will decrease the expected number of accessed memory blocks when the states in C'_{after} are sorted. For example, assume the toy C'_{after} array above is sorted as

$$C'_{after} = [1, 2, 10, 11, 12].$$

By fixing the second state at each iteration, we can concurrently process first the pairs

$$[\langle 1, 12\rangle, \langle 2, 12\rangle, \langle 10, 12\rangle, \langle 11, 12\rangle],$$

then the pairs

$$[\langle 1, 11\rangle, \langle 2, 11\rangle, \langle 10, 11\rangle]$$

and so on. With this approach and the smart indexing scheme, the locations are expected to be much closer compared to the unsorted variant. In this scheme, the pairs are distributed to the threads in a round-robin fashion. Although the last iterations do not have enough number of pairs to feed all the threads in the warp, when C' is large, the threads are assigned an almost equal number of pairs and compute the partial cost in a load-balanced way.

4. Making SynchroP Faster

In this section, we will introduce three improvements for increasing SynchroP's performance. The first improvement (explained in Section 4.1) eliminates some redundant cost computations by precomputing $\delta(C, \tau_{\langle i,j\rangle})$ whenever possible. The improvement explained in Section 4.2 delays all the precomputations until they are actually required. Finally in Section 4.3, we explain a

particular improvement that accelerates SynchroP's first iteration, which is almost always the most expensive one. All these improvements can be easily adapted to the parallel implementation described in Section 3.

4.1. Eliminating Redundant Cost Computations

Given an active state set C, Algorithm 2 first computes the cost $\phi(\delta(C, \tau_{\langle i,j \rangle}))$ for each d–pair $\langle i, j \rangle \in C_d^2$. For all pairs in C_d^2, this cost only depends on the path $\tau_{\langle i,j \rangle}$. Hence, when $\tau_{\langle i',j' \rangle} = \tau_{\langle i,j \rangle}$ for another active pair $\langle i', j' \rangle \in C_d^2$, computing the same cost $\phi(\delta(C, \tau_{\langle i',j' \rangle})) = \phi(\delta(C, \tau_{\langle i,j \rangle}))$ is a redundant operation. To eliminate the redundant cost computations, we only consider the set of non–empty words $\Sigma^{\leq k}$ of length at most $k \geq 1$, i.e.,

$$\Sigma^{\leq k} = \{\sigma \mid \sigma \in \Sigma^\star, 1 \leq |\sigma| \leq k\}.$$

In the proposed modification, each iteration of SynchroP precomputes the cost $\phi(\delta(C, \sigma))$ for all $\sigma \in \Sigma^{\leq k}$. Then, when $|\tau_{\langle i,j \rangle}| \leq k$, it simply looks up the precomputed cost $\phi(\delta(C, \tau_{\langle i,j \rangle}))$ for any d–pair $\langle i, j \rangle \in C_d^2$. Let $\Phi(C, \sigma)$ be this previously computed cost of $\phi(\delta(C, \sigma))$ for a word $\sigma \in \Sigma^{\leq k}$. The modified second phase operates using these previously computed costs, as shown in Algorithm 3.

4.2. Lazy Computation and Memoization of Path Costs

The previous approach precomputes $\Phi(C, \sigma)$ for all $\sigma \in \Sigma^{\leq k}$. However in a single iteration, the only $\Phi(C, \sigma)$ values Algorithm 3 requires are the ones with $\sigma = \tau_{\langle i,j \rangle}$ for some $\langle i, j \rangle \in C_d^2$. Therefore, precomputing $\Phi(C, \sigma)$ for only those $\sigma \in \Sigma^{\leq k}$ is a promising approach. An efficient way to do this is using a lazy computation approach and compute $\Phi(C, \sigma)$ only when necessary. That is one can compute $\Phi(C, \sigma)$ for $\sigma = \tau_{\langle i,j \rangle}$ only for the first time it is actually required, and memoize it for further use. Algorithm 4 presents this technique implemented to improve the performance of SynchroP.

Algorithm 3: Computing a synchronizing word (modified Phase 2 of SynchroP)

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j \rangle}$ for all $\langle i, j \rangle \in S^2$, an integer $k \geq 1$
output: A synchronizing word Γ for A

```
1  C = S; // C:  current active state set
2  Γ = ε; // Γ:  synchronizing word, initially empty
3  while |C| > 1 do // still not a singleton
4      foreach σ ∈ Σ^{≤k} do // precompute Φ(C, σ)
5          Φ(C, σ) = φ(δ(C, σ))
6      minCost = ∞
7      foreach d–pair ⟨i, j⟩ ∈ C_d^2 do
8          if |τ_⟨i,j⟩| ≤ k then
9              thisPairCost = Φ(C, τ_⟨i,j⟩)
10         else
11             thisPairCost = φ(δ(C, τ_⟨i,j⟩))
12         if thisPairCost < minCost then
13             minCost = thisPairCost
14             τ' = τ_⟨i,j⟩
15     Γ = Γ τ'; // append τ' to the synchronizing word
16     C = δ(C, τ'); // update current active state set with τ'
```

4.3. Accelerating the First Iteration

The last improvement which focuses on the runtime of the first iteration of the algorithm is based on the following observation.

Lemma 4.1. *Let $C \subseteq S$ be a subset of states and $\langle i, j \rangle, \langle i', j' \rangle \in C_d^2$ be two d–pairs such that $\tau_{\langle i,j \rangle} = \sigma \tau_{\langle i',j' \rangle}$ for some $\sigma \in \Sigma^\star$. If $\delta(C, \sigma) \subseteq C$ then $\phi(\delta(C, \tau_{\langle i,j \rangle})) \leq \phi(\delta(C, \tau_{\langle i',j' \rangle}))$.*

Proof. We have $\delta(C, \tau_{\langle i,j \rangle}) = \delta(\delta(C, \sigma), \tau_{\langle i',j' \rangle}) \subseteq \delta(C, \tau_{\langle i',j' \rangle})$, where the last step is due to the fact that $\delta(C, \sigma) \subseteq C$. Since $\delta(C, \tau_{\langle i,j \rangle}) \subseteq \delta(C, \tau_{\langle i',j' \rangle})$, we have $\phi(\delta(C, \tau_{\langle i,j \rangle})) \leq \phi(\delta(C, \tau_{\langle i',j' \rangle}))$. □

Based on Lemma 4.1, when two d–pairs $\langle i, j \rangle, \langle i', j' \rangle$ exist in C_d^2 such that $\tau_{\langle i,j \rangle} = \sigma \tau_{\langle i',j' \rangle}$ for some $\sigma \in \Sigma^\star$, we have $\phi(\delta(C, \tau_{\langle i,j \rangle})) \leq \phi(\delta(C, \tau_{\langle i',j' \rangle}))$. Hence, we can eliminate the redundant consideration of $\langle i', j' \rangle$ in the same iteration. Although it is hard to find such pairs in later iterations, for the first iteration, $\delta(C, \sigma) \subseteq C$ for any σ since $C = S$. Furthermore, the first iteration takes much more time compared to other iterations. Hence, in the first iteration

Algorithm 4: Computing a synchronizing word (modified Phase 2 of SynchroP) with lazy computation and memoization

input : An automaton $A = (S, \Sigma, \delta)$ and $\tau_{\langle i,j \rangle}$ for all $\langle i, j \rangle \in S^2$, an integer $k \geq 1$
output: A synchronizing word Γ for A

```
1  C = S; // C:  current active state set
2  Γ = ε; // Γ:  synchronizing word, initially empty
3  while |C| > 1 do // still not a singleton
4      foreach σ ∈ Σ^{≤k} do // init Φ(C, σ)
5          Φ(C, σ) = ∞
6      minCost = ∞;
7      foreach d–pair ⟨i, j⟩ ∈ C_d^2 do
8          if |τ_⟨i,j⟩| ≤ k then
9              if Φ(C, τ_⟨i,j⟩) = ∞ then
10                 Φ(C, τ_⟨i,j⟩) = φ(δ(C, τ_⟨i,j⟩))
11             thisPairCost = Φ(C, τ_⟨i,j⟩)
12         else
13             thisPairCost = φ(δ(C, τ_⟨i,j⟩))
14         if thisPairCost < minCost then
15             minCost = thisPairCost
16             τ' = τ_⟨i,j⟩
17     Γ = Γ τ'; // append τ' to the synchronizing word
18     C = δ(C, τ'); // update current active state set with τ'
```

of SynchroP, we only consider only those d–pairs $\langle i, j \rangle \in S_d^2$ such that $\tau_{\langle i,j \rangle}$ is not a suffix of $\tau_{\langle i',j' \rangle}$ for any other d–pair $\langle i', j' \rangle \in S_d^2$.

5. New SynchroP Variants

Here we introduce two new variants that can be adapted to any SynchroP implementation: Cardinality and Multiplicative. These two algorithms add new perspectives to the cost computations of the original SynchroP heuristic.

5.1. Cardinality *Algorithm*

The original SynchroP algorithm comes with the time complexity $O(n^5 + pn^2)$. While the early sections of this paper show that it is possible to apply optimizations and parallelization, the theoretical complexity bound still limits

the capacity of the algorithm in big automata. The CARDINALITY algorithm aims to decrease the original complexity to $O(n^3 + pn^2)$ by disregarding the highest quality concern.

For a set of states $C \subseteq S$, the CARDINALITY algorithm defines the cost $\phi(C)$ of C as

$$\phi(C) = |C|$$

and skips $O(n^2)$ calculations by the lack of need for traversing the d–pairs in C_d^2 in every iteration. The intuition for choosing $|C|$ is twofold. First, it is quite fast to compute $|C|$. Second, SYNCHROP and SYNCHROPL measures the quality of a set C of states by their respective cost functions. In both of these algorithms, the current active state set C starts with $|C| = n$ and the ultimate aim is to get down to an active state set C such that $|C| = 1$. Therefore, using $|C|$ as the measure of the quality of C is quite natural as well. As we will show later in Section 6, CARDINALITY is much faster than SYNCHROP and SYNCHROPL with comparable performance in quality.

5.2. MULTIPLICATIVE *Algorithm*

This variant suggests a small tweak to SYNCHROPL cost function. SYNCHROPL successfully combines the applicability of a sequence by using its length and the reducibility of the subset it carries the active state set to. In the original proposal, these two factors are added. However, the magnitude of the reducibility index is far greater than the length of the sequence that often, the extra logic SYNCHROPL supplies is not effectively blended in. To account for this proposition, we propose the multiplicative version of the algorithm, which uses the multiplication of these two indices.

Let $C \subseteq S$ be the current active state set. For a sequence $\sigma \in \Sigma^*$, MULTIPLICATIVE uses the cost function

$$\phi_{ML}(\delta(C, \sigma)) = \phi(\delta(C, \sigma)) * |\sigma| = \sum_{i,j \in C} |\tau_{\langle i,j \rangle}| * |\sigma|.$$

6. Experimental Results

The experiments were performed on a machine running on Ubuntu 16.04.2 equipped with a 192GB of memory and a dual-socket Intel Xeon E5-2620 v4 clocked at 2.10GHz. Besides, GPU experiments were run on NVIDIA GeForce GTX 980 and TITAN. The code was written in C++ with CUDA and compiled using `gcc` version 5.4.0 and `nvcc` version 8.0.61 with `-O3` option enabled for both.

In order to evaluate the performance of our implementations, we used three different sets of automata. The first set of automata is randomly generated. The results of the experiments on random automata are given in Section 6.1. There are some known classes of slowly synchronizing automata with long shortest synchronizing sequences. Section 6.2 gives the results of the experiments performed on a set of slowly synchronizing automata. Finally in Section 6.3, we give the result of the experiments we performed on some benchmark automata taken from (Neider et al., 2019).

The source code of all the algorithms and the automata we used in the experiments are publicly available[1].

6.1. Experiments on Random Automata

We generated a set of automata randomly with $n \in \{1024, 2048, 4096, 8192\}$ states and $p \in \{2, 8, 32\}$ inputs. For each (n, p) pair, we generated 20 different automata and ran each algorithm on them. The values presented in this section are the averages of these 20 executions for each configuration. GPU implementations use 256 threads per block where the number of blocks is determined by the automaton size and the available device memory.

6.1.1. CPU Experiments

Table 1 presents the execution times of SynchroP CPU implementations for different parameters. The standard algorithm, as used in the literature for benchmarking purposes, is denoted as *ORG*. The SynchroP variants described

[1] http://bitbucket.org/egesarac/boostexpsyncheur/

in Section 4.1 and Section 4.2, which apply pre-computations for the costs and the lazy cost computation, are denoted as *PC* and *Lazy*, respectively. In both variants, the optimization from Section 4.3 is implemented.

p \ n	1024			2048			4096	8192
	ORG	PC	Lazy	ORG	PC	Lazy	Lazy	Lazy
2	33.61	6.94	0.83	425.21	89.90	4.51	39.31	400.00
8	98.52	37.42	1.90	1407.60	161.39	21.39	216.35	3828.50
32	159.00	11.38	9.02	2368.61	1142.32	46.33	362.93	12807.72

Table 1: Execution times (in seconds) of original SynchroP on CPU with and without pre-computation and lazy cost computation.

There is no difference on the quality of synchronizing words produced by ORG, PC, and Lazy. In fact, they all report the same synchronizing word length for every automaton.

For the time comparison, the original SynchroP implementation takes much more time compared to the variants with the optimizations, as expected. Depending on the automata size, n and p, with pre-computation 2.1×–14.5× speedup can obtained. The lazy computation improves the runtime much more: when $n = 2048$ and $p = 32$, the execution time is reduced from 2368.6 seconds to 46.3 seconds. Although its performance is superior, even for medium-scale automata, e.g., $n = 8192$ and $p = 32$, the Lazy variant of SynchroP takes more than 3.5 hours. Fortunately, as we will show, by using GPUs, synchronization can be performed in two minutes, even for such automata sizes.

6.1.2. GPU Experiments

We label our GPU implementations by i_X as follows:

- $\mathbf{1_T}$ is the thread-based naive GPU implementation described in Section 3.1 which assigns a merging sequence to a single thread.
- $\mathbf{2_T}$ is the thread-based GPU implementation in Section 3.2 which uses a reorganized memory in Figure 2 (b).
- $\mathbf{3_W}$ is the warp-based GPU implementation Section 3.3 describes which assigns a merging sequence to a single warp and uses unordered active state sets for cost computation.

- $\mathbf{4_W}$ is the warp-based GPU implementation Section 3.4 which sorts the active state sets before performing the cost computation.
- $\mathbf{5_W}$ is the warp-based GPU implementation which applies the optimizations described in Section 4.

There is no difference on the quality of synchronizing words produced by the different GPU implementations given above. In fact, they all report the same synchronizing word length for every automaton.

Even though no meaningful difference on the length of the synchronizing word is expected, GPU versions and CPU versions do not necessarily produce the same synchronizing word. In every iteration, the algorithms perform a search for a pair $\langle i, j\rangle \in C_d^2$ with a minimal cost. There can be multiple pairs in C_d^2 giving the minimal cost. Due to the different search order followed by the CPU and GPU versions among the pairs in C_d^2, the CPU version can pick a pair $\langle i, j\rangle \in C_d^2$ and the GPU version can pick another pair $\langle i', j'\rangle \in C_d^2$, both with the minimal cost. The lengths of $\tau_{\langle i,j\rangle}$ and $\tau_{\langle i',j'\rangle}$ can be different. Moreover, the set of active states that would be obtained by using $\tau_{\langle i,j\rangle}$ and $\tau_{\langle i',j'\rangle}$ would also be different. Therefore, the rest of the iterations of the algorithm will have to deal with different set of active states based on this different sequence selection. We observe exactly such an effect on the length of the synchronizing words constructed by CPU and GPU versions. For some automata, the CPU version finds a shorter synchronizing word, and for some automata, the GPU version finds a shorter synchronizing word. However, as expected, the difference on the lengths of synchronizing words is not meaningful. On average, the difference observed is in the order of 1% – 3%.

Table 2 and Table 3 show the execution times of the proposed GPU implementations and SynchroP variants on GeForce GTX and TITAN, respectively. As expected, the execution times on TITAN are usually shorter compared to GTX. For $n = 2048$ and $p = 32$, SynchroP becomes 14× and 16× faster on GTX and TITAN, compared to the best CPU implementation Lazy. Among the heuristic variants, Cardinality is the fastest one, and Multiplicative is

slightly slower than the original proposal. However, the performance difference among the CARDINALITY and the others reduces when more optimizations are added to the implementations. For instance, for 5_W on GTX, SYNCHROP takes only 3.31 seconds on average whereas CARDINALITY takes 2.44 seconds. On TITAN, the gap is even smaller; SYNCHROP takes only 2.88 seconds on average whereas CARDINALITY takes 2.48 seconds.

		SYNCHROP				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	8.41	4.15	2.16	0.89	0.14
	8	28.40	10.42	7.04	1.27	0.26
	32	46.50	15.40	11.00	2.04	0.80
2048	2	125.02	73.25	51.54	13.30	0.53
	8	485.43	241.88	176.70	31.41	1.39
	32	795.43	386.42	280.03	45.00	3.31
		SYNCHROPL				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	8.41	4.15	2.15	0.89	0.14
	8	28.41	10.42	7.04	1.29	0.26
	32	46.33	15.45	10.98	2.08	0.78
2048	2	124.94	73.25	51.55	13.05	0.52
	8	485.34	241.90	176.65	30.98	1.38
	32	795.54	386.37	279.97	43.15	3.31
		MULTIPLICATIVE				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	8.54	4.26	2.24	0.99	0.22
	8	29.04	10.65	7.20	1.40	0.35
	32	47.88	15.84	11.38	2.23	0.86
2048	2	126.59	73.96	52.34	13.67	0.91
	8	499.63	248.36	181.93	32.74	1.89
	32	829.11	401.95	292.64	46.42	4.33
		CARDINALITY				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	7.22	1.88	0.70	0.59	0.13
	8	4.19	3.09	0.73	0.49	0.24
	32	3.87	3.31	1.18	0.93	0.70
2048	2	63.67	25.66	5.39	3.54	0.47
	8	33.45	29.70	5.57	3.16	0.96
	32	29.94	33.03	7.32	4.58	2.44

Table 2: Execution times (in seconds) of SYNCHROP and its variants with various optimizations on GeForce GTX.

To further analyze the performance and scalability of the proposed GPU implementation 5_W, we used larger automata with $n = 4096$ and $n = 8192$. Ta-

		SynchroP				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	6.25	1.69	0.75	0.56	0.12
	8	22.44	2.83	1.23	0.68	0.24
	32	36.28	4.08	2.08	1.25	0.70
2048	2	117.82	46.66	29.66	6.79	0.44
	8	427.96	164.80	119.35	16.18	1.11
	32	704.14	269.36	197.24	23.80	2.88
		SynchroPL				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	6.26	1.70	0.74	0.57	0.12
	8	22.44	2.84	1.23	0.69	0.24
	32	36.33	4.11	2.30	1.25	0.70
2048	2	117.78	46.66	29.74	6.78	0.44
	8	428.01	164.84	119.42	15.97	1.09
	32	704.28	269.14	197.64	23.47	2.87
		Multiplicative				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	6.35	1.81	0.81	0.64	0.18
	8	22.82	2.97	1.30	0.76	0.29
	32	37.36	4.25	2.21	1.31	0.76
2048	2	118.94	47.15	30.15	7.14	0.70
	8	439.68	169.16	122.75	16.88	1.37
	32	733.33	279.80	205.70	25.08	3.31
		Cardinality				
n	p	1_T	2_T	3_W	4_W	5_W
1024	2	5.97	0.69	0.51	0.43	0.12
	8	3.43	0.77	0.58	0.36	0.23
	32	3.38	1.27	0.98	0.78	0.68
2048	2	51.45	4.96	4.05	2.21	0.42
	8	26.61	5.66	4.12	2.07	0.89
	32	24.45	7.70	6.14	3.59	2.48

Table 3: Execution times (in seconds) of SynchroP and its variants with various optimizations on TITAN.

ble 4 displays the execution times on both GTX and TITAN. The performance on both devices for this challenging setting demonstrates that our implementation scales very well for large inputs. For $p = 32$, the best CPU implementation takes 362.9 and 12807.7 seconds, respectively, for $n = 4096$ and $n = 8192$. On GTX, the proposed 5_W implementation takes 16.8 and 180.6 seconds which yield 22× and 71× speedup compared to Lazy. On a relatively recent GPU, TITAN, these speedup values are 25× and 108×, respectively, for $n = 4096$ and $n = 8192$.

		SynchroP		SynchroPL	
n	p	G	T	G	T
4096	2	2.70	2.18	2.68	2.23
	8	6.95	5.41	6.79	5.38
	32	16.83	14.70	16.84	14.64
8192	2	15.78	12.06	16.03	11.49
	8	65.64	43.92	66.40	42.79
	32	180.56	119.04	186.24	119.80
		Multiplicative		Cardinality	
n	p	G	T	G	T
4096	2	5.41	3.87	2.18	1.94
	8	11.76	8.10	4.21	3.78
	32	23.34	18.00	12.16	12.07
8192	2	37.55	24.35	11.08	8.75
	8	103.99	65.07	21.03	17.30
	32	228.92	144.35	56.69	52.45

Table 4: Execution times (in seconds) of the SynchroP variants with our proposed implementation 5_W on larger automata of GTX (G) and TITAN (T).

Table 5 shows the speedup values of our GPU implementation 5_W against the naive sequential implementation ORG. The final GPU implementation 5_W outperforms the original SynchroP proposal more than 1000× on both GTX and TITAN.

			GTX				
n	p	ORG	1_T	2_T	3_W	4_W	5_W
1024	2	45.3	4.5	9.1	17.4	42.3	275.0
	8	54.6	3.7	10.0	14.7	81.7	391.8
	32	18.2	3.5	10.7	15.0	80.7	204.3
2048	2	107.0	3.9	6.6	9.4	36.3	916.9
	8	74.8	3.3	6.6	9.1	51.0	1153.1
	32	53.6	3.1	6.4	8.9	55.2	749.7
			TITAN				
n	p	ORG	1_T	2_T	3_W	4_W	5_W
1024	2	45.3	6.0	22.2	50.1	67.1	313.0
	8	54.6	4.6	36.7	84.3	152.6	432.3
	32	18.2	4.5	40.3	79.1	131.6	235.0
2048	2	107.0	4.1	10.3	16.3	71.1	1097.0
	8	74.8	3.7	9.7	13.4	98.9	1442.2
	32	53.6	3.5	9.2	12.6	104.3	862.4

Table 5: Speedup values of our implementations based on the naive, original sequential SynchroP implementation.

6.1.3. Length of the Synchronizing Words

Table 6 lists the average lengths of synchronizing sequences computed by SynchroP, SynchroPL, Multiplicative, and Cardinality heuristics. As expected, compared to SynchroP, SynchroPL achieves to find 2.76% shorter sequences on average. Although Multiplicative computes 4.19% longer sequences than that of SynchroPL on average, it yields minor improvements especially when the number of inputs is large. Cardinality falls behind SynchroPL with a margin of 6.07% considering the sequence lengths. However, it is only a small setback due to Cardinality's superiority in running times, e.g., it is more than 2× faster for $n = 8192$ as Table 4 shows.

n	p	SynchroP	SynchroPL	Multiplicative	Cardinality
1024	2	115.15	113.9	123.65	123.56
	8	68.00	67.25	70.14	72.60
	32	55.78	54.39	52.90	56.82
2048	2	167.25	165.20	177.15	178.73
	8	98.08	97.60	101.70	104.26
	32	77.85	76.65	76.41	81.74
4096	2	242.05	239.05	260.60	253.38
	8	144.43	142.11	147.08	148.61
	32	110.25	108.55	109.45	115.00
8192	2	350.35	344.80	385.15	365.75
	8	204.65	201.91	211.85	212.19
	32	161.25	159.30	157.75	163.43

Table 6: Synchronizing sequence lengths for the proposed SynchroP variants.

6.2. Experiments on Slowly Synchronizing Automata

There are some classes of automata whose shortest synchronizing sequences are known to be long. These are known as *slowly synchronizing automata*. The most famous of such classes is possibly the `cerny` automata which actually hits to the upper bound conjectured by (Černỳ, 1964). In other words, a `cerny` automaton with n states is known to have a shortest synchronizing sequence of length $(n-1)^2$.

We also consider two classes of slowly synchronizing automata introduced in (Ananichev et al., 2006). The class `bactrian` given in Section 3 of (Ananichev et al., 2006) has a shortest synchronizing sequence of length $(n-1)(n-2)$ for

an automaton with n states, where $n > 3$ is an odd number. In Section 4 of (Ananichev et al., 2006), another slowly synchronizing class of automata, `dromedary`, is introduced. A `dromedary` automaton with n states has a shortest synchronizing sequence of length of $(n-2)^2 + 1$.

Another class of slowly synchronizing automata is introduced by (Don et al., 2020, Theorem 3). The authors provide automaton structure with 5,4, and 3 input letters which we call here as classes `fix5`, `fix4`, `fix3`, respectively. It is shown that the length of the shortest synchronizing sequences for `fix5`, `fix4` and `fix3` are $n^2 - 3n + 2$, $n^2 - 3n + 3$, and $n^2 - 3n + 4$, respectively.

Volkov (2019) introduces a *transformation* that can take any automaton A with n states to produce another automaton $\mathbb{H}(A)$ with $2n$ states. If A is synchronizing, then so is $\mathbb{H}(A)$. Furthermore, the length of the shortest synchronizing sequence for $\mathbb{H}(A)$ is 2 times that of A. For example, if we take A as the `cerny` automaton with n states, then $\mathbb{H}(A)$, which we call `doubleCerny`, is an automaton with $2n$ states, and the shortest synchronizing sequence length for $\mathbb{H}(A)$ is $2(n-1)^2$.

We test the performance of the methods suggested in this paper for slowly synchronizing automata as well. Note that, such automata are among the hardest classes of automata that the heuristic algorithms will have to deal with. Both the number of iterations and the length of the synchronizing words used in the iterations increase for such automata. Therefore, the running time of the algorithms also increases accordingly. Table 7 gives the result of the experiments using CPU. It is seen that the algorithms given in Section 4.1 and Section 4.2 do not help to improve the running time for slowly synchronizing automata.

The speedup results of obtained on slowly synchronizing automata by using GPU algorithms are given in Table 8. Again, the final GPU implementation 5_W gives the best speedup values in general. Even though there are some cases where 5_W is slower than the original naive SynchroP (especially for small automata sizes), depending on the class of automata 5_W becomes hundreds or even thousands times faster than the original SynchroP as the size of automata gets larger.

Automata Class	n	ORG	PC	Lazy
`cerny`	32	0.01	0.01	0.01
	64	0.24	0.20	0.21
	128	4.49	4.52	4.53
	256	130.80	130.73	130.85
	512	3980.30	3981.20	3980.81
`bactrian`	31	0.01	0.01	0.01
	63	0.18	0.16	0.19
	127	3.95	3.93	3.99
	255	116.80	116.75	116.77
	511	3614.04	3694.61	3704.43
`dromedary`	32	0.01	0.01	0.01
	64	0.21	0.23	0.16
	128	4.45	4.46	4.67
	256	133.42	132.96	133.58
	512	4050.31	4026.03	4038.46
`fix5`	32	0.01	0.01	0.01
	64	0.24	0.19	0.17
	128	4.62	4.73	4.79
	256	135.41	134.52	134.80
	512	4051.87	4087.65	4083.93
`fix4`	32	0.01	0.01	0.01
	64	0.21	0.18	0.17
	128	4.88	4.81	4.55
	256	135.18	135.27	130.67
	512	3985.33	4030.96	3991.23
`fix3`	32	0.01	0.01	0.01
	64	0.20	0.18	0.21
	128	4.54	4.55	4.51
	256	130.50	130.58	130.54
	512	3991.55	3980.02	3976.20
`doubleCerny`	32	0.01	0.01	0.01
	64	0.081	0.11	0.12
	128	2.04	2.06	2.08
	256	59.94	59.71	59.85
	512	1846.43	1925.43	1839.90

Table 7: Execution times (in seconds) of original SynchroP on CPU with and without precomputation and lazy cost computation.

Table 9 gives the results of the experiments for the lengths of the synchronizing sequences found by the algorithms. The CPU algorithms and 1_T and 2_T running on GTX and TITAN all find the same length synchronizing sequences. This length is given by the column labeled as "CPU, 1_T, 2_T" in Table 9.

Automata Class	n	GTX 1_T	GTX 2_T	GTX 3_W	GTX 4_W	GTX 5_W	TITAN 1_T	TITAN 2_T	TITAN 3_W	TITAN 4_W	TITAN 5_W
`cerny`	32	0.91	1.02	4.70	4.70	0.32	0.96	1.07	5.68	5.14	0.57
	64	2.40	3.02	9.30	10.42	5.71	2.59	3.22	12.93	12.66	9.53
	128	3.84	6.54	9.08	9.08	35.92	6.00	8.95	12.20	12.33	46.04
	256	1.35	11.35	17.37	17.25	111.34	2.49	16.19	28.28	28.20	163.46
	512	0.77	13.15	18.44	18.43	228.75	0.93	18.63	29.13	29.13	388.97
`bactrian`	31	0.82	0.99	3.89	3.70	0.23	0.97	1.19	4.63	4.63	0.40
	63	2.61	3.12	8.39	8.24	4.69	3.14	3.71	10.80	10.49	8.59
	127	4.09	6.19	9.75	9.72	52.37	7.41	11.99	15.43	15.39	79.24
	255	0.91	8.18	18.00	18.00	372.82	1.81	12.00	29.54	29.59	507.41
	511	0.76	8.00	31.64	31.67	1819.03	0.95	11.16	52.21	52.08	2387.08
`dromedary`	32	0.72	0.81	3.90	3.90	0.69	0.89	0.99	5.13	4.56	1.22
	64	2.24	2.72	8.61	8.47	11.12	2.91	3.24	11.96	11.75	15.58
	128	3.96	6.55	8.96	8.97	38.07	6.67	10.15	14.01	14.03	57.98
	256	1.38	11.66	17.74	17.75	99.89	2.56	16.57	28.62	28.70	159.91
	512	0.80	13.38	18.45	18.45	204.09	0.95	18.78	29.29	30.01	335.55
`fix5`	32	0.84	0.94	4.81	4.39	0.47	1.01	1.16	5.94	5.61	0.81
	64	2.45	3.03	9.74	9.66	8.36	3.34	3.77	13.45	13.22	14.35
	128	3.96	6.75	9.40	9.40	37.15	6.68	10.57	14.73	14.67	56.92
	256	1.40	11.82	18.08	18.08	100.81	2.58	16.72	29.46	29.32	162.34
	512	0.78	13.42	18.81	18.81	200.38	0.95	19.00	29.32	29.58	333.99
`fix4`	32	0.80	0.91	4.62	4.22	0.80	0.97	1.10	5.11	5.39	1.24
	64	2.16	2.69	8.73	8.55	10.81	2.95	3.33	11.79	11.65	15.01
	128	4.19	7.13	9.92	9.93	46.65	7.06	11.16	15.53	15.40	69.98
	256	1.39	11.80	18.05	18.04	119.52	2.58	16.69	29.39	29.19	190.66
	512	0.77	13.20	18.50	18.50	229.66	0.94	18.68	29.07	28.88	382.98
`fix3`	32	0.86	0.97	4.95	4.73	0.99	1.06	1.17	6.12	5.78	1.44
	64	2.09	2.60	8.41	8.34	10.83	2.86	3.17	11.50	11.37	17.70
	128	3.89	6.62	9.23	9.23	43.76	6.57	10.37	14.46	14.42	64.98
	256	1.35	11.39	17.43	17.43	115.54	2.49	16.11	28.31	28.26	184.27
	512	0.77	13.23	18.53	18.53	229.83	0.94	18.71	29.12	29.14	382.49
`doubleCerny`	32	0.77	0.83	3.08	3.08	0.38	0.91	1.00	3.64	3.64	0.63
	64	1.82	2.21	8.37	8.12	5.24	2.19	2.67	13.76	12.49	8.20
	128	4.25	6.53	16.31	16.20	25.13	6.68	10.82	25.13	24.56	39.57
	256	1.27	10.17	17.73	17.73	33.52	2.37	15.00	28.94	28.92	55.00
	512	0.84	12.32	32.63	32.66	35.87	0.95	17.17	53.27	53.37	57.44

Table 8: Speedup values of our implementations based on the naive, original sequential SynchroP implementation.

Automata Class	n	CPU, 1_T, 2_T	GTX 3_W	GTX 4_W	GTX 5_W	TITAN 3_W	TITAN 4_W	TITAN 5_W	Shortest
`cerny`	32	961	961	961	976	961	961	976	900
	64	3969	3969	3969	4000	3969	3969	4000	3844
	128	16129	16129	16129	16192	16129	16129	16192	15876
	256	65025	65025	65025	65152	65025	65025	65152	64516
	512	261121	261347	261347	261376	261347	261347	261376	260100
`bactrian`	31	870	870	870	870	870	870	870	870
	63	3782	3782	3782	3782	3782	3782	3782	3782
	127	15750	15750	15750	15750	15750	15750	15750	15750
	255	64262	64262	64262	64262	64262	64262	64262	64262
	511	259590	259590	259590	259590	259590	259590	259590	259590
`dromedary`	32	915	915	915	915	915	915	915	901
	64	3875	3875	3875	3875	3875	3875	3875	3845
	128	15939	15939	15939	15939	15939	15939	15939	15877
	256	64643	64643	64643	64643	64643	64643	64643	64517
	512	260355	260326	260326	260355	260197	260197	260355	260101
`fix5`	32	931	931	931	930	931	931	930	930
	64	3907	3907	3907	3906	3907	3907	3906	3906
	128	16003	16003	16003	16002	16003	16003	16002	16002
	256	64771	64771	64771	64770	64771	64771	64770	64770
	512	260611	260837	260837	260610	260708	260708	260610	260610
`fix4`	32	931	931	931	945	931	931	945	931
	64	3907	3907	3907	3937	3907	3907	3937	3907
	128	16003	16003	16003	16065	16003	16003	16065	16003
	256	64771	64771	64771	64897	64771	64771	64897	64771
	512	260611	260837	260837	260865	260708	260708	260865	260611
`fix3`	32	933	933	933	947	933	933	947	932
	64	3909	3909	3909	3939	3909	3909	3939	3908
	128	16005	16005	16005	16067	16005	16005	16067	16004
	256	64773	64773	64773	64899	64773	64773	64899	64772
	512	260613	260839	260839	260867	260710	260710	260867	260612
`doubleCerny`	32	450	450	450	464	450	450	464	450
	64	1922	1922	1922	1952	1922	1922	1952	1922
	128	7938	7938	7938	8000	7938	7938	8000	7938
	256	32258	32258	32258	32384	32258	32258	32384	32258
	512	130050	130265	130265	130304	130243	130243	130304	130050

Table 9: Synchronizing sequence lengths found by the algorithms.

The length of the synchronizing sequence found by all of the heuristics (ORG, PC, Lazy) are the same and, depending on the class, it is either the same as, or very close, to the shortest synchronizing sequence of the automaton. There are small differences in the length of the sequence obtained by different algorithms, or by the same algorithm running on different platforms, which can be explained by the different outcome of race conditions happening during the executions of the algorithms. However, the lengths are in general very close to each other. In addition, the lengths are very close to (or in some cases, equal to) the known shortest synchronizing sequence length as given by the column labeled as "Shortest" in Table 9.

6.3. Experiments on Benchmark Automata

Neider et al. (2019) introduces a collection of benchmark automata and finite state machines (FSMs). We considered automata/FSMs which are not random and which have more than 100 states. Among such automata/FSMs, we identified 22 of them, which are complete and synchronizing. For an FSM, we simply neglected the output of the transitions, and considered it as an automaton.

Even for the largest of these automata, the length of the synchronizing sequence is at most 2. This means that synchronizing sequence algorithms will iterate at most two times and the length of the merging word used in an iteration is very short. The results for these experiments are given in Table 10. Even though synchronizing sequences are very short, the final GPU implementation 5_W and the CPU algorithm "Lazy" manage to reach around 60x speed up for the largest automaton "m65" appearing at the last row of Table 10.

	n	p	ORG	PC	Lazy	1_T	2_T	3_W	4_W	5_W	1_T	2_T	3_W	4_W	5_W
ABP9	101	102	0.0089	0.0072	0.0037	0.0079	0.0069	0.0082	0.0076	0.0065	0.0072	0.0085	0.0086	0.0081	0.0087
m172	113	98	0.0082	0.0055	0.0040	0.0086	0.0088	0.0086	0.0087	0.0063	0.0085	0.0088	0.0090	0.0083	0.0095
m34	115	72	0.0070	0.0049	0.0031	0.0074	0.0073	0.0072	0.0067	0.0049	0.0068	0.0069	0.0075	0.0071	0.0069
ABP10	122	123	0.0119	0.0095	0.0069	0.0120	0.0125	0.0112	0.0118	0.0090	0.0122	0.0116	0.0122	0.0116	0.0129
m201	128	87	0.0084	0.0068	0.0065	0.0101	0.0095	0.0092	0.0091	0.0077	0.0099	0.0096	0.0093	0.0097	0.0096
m49	142	75	0.0113	0.0075	0.0084	0.0100	0.0097	0.0101	0.0097	0.0079	0.0111	0.0110	0.0105	0.0112	0.0102
m167	163	152	0.0182	0.0184	0.0217	0.0235	0.0191	0.0209	0.0213	0.0185	0.0207	0.0211	0.0235	0.0206	0.0208
m55	181	156	0.0290	0.0207	0.0269	0.0327	0.0219	0.0251	0.0246	0.0225	0.0277	0.0241	0.0263	0.0263	0.0256
m45	184	32	0.0133	0.0047	0.0079	0.0159	0.0098	0.0118	0.0109	0.0071	0.0112	0.0097	0.0103	0.0102	0.0099
m185	190	71	0.0175	0.0118	0.0134	0.0230	0.0147	0.0182	0.0152	0.0120	0.0180	0.0150	0.0161	0.0145	0.0149
m27	198	201	0.0458	0.0349	0.0360	0.0466	0.0358	0.0344	0.0383	0.0324	0.0421	0.0390	0.0375	0.0362	0.0343
m76	210	26	0.0162	0.0085	0.0076	0.0228	0.0110	0.0122	0.0123	0.0074	0.0174	0.0106	0.0110	0.0104	0.0102
m24	284	100	0.0775	0.0409	0.0278	0.0653	0.0484	0.0492	0.0468	0.0397	0.0648	0.0408	0.0395	0.0390	0.0409
m189	289	138	0.0681	0.0647	0.0429	0.0800	0.0622	0.0637	0.0605	0.0482	0.0793	0.0506	0.0500	0.0569	0.0518
m190	456	52	0.1436	0.0627	0.0688	0.2002	0.0793	0.0886	0.0814	0.0536	0.1802	0.0770	0.0795	0.0742	0.0633
m173	483	27	0.2039	0.0372	0.0380	0.2452	0.0624	0.0795	0.0674	0.0358	0.2148	0.0596	0.0648	0.0619	0.0488
m182	657	75	0.4211	0.1313	0.1417	0.6364	0.1984	0.2176	0.1932	0.1475	0.5399	0.1779	0.1895	0.1893	0.1572
m131	1017	181	1.8029	0.7843	0.7095	2.3351	0.7733	0.9479	0.9228	0.7142	2.0885	0.7663	0.8385	0.7522	0.7349
m181	1347	93	3.3085	0.7205	0.6863	4.2568	0.8474	1.2814	1.0893	0.7032	3.7943	0.7678	0.9432	0.8554	0.6867
m85	2221	121	14.2641	2.5167	2.3612	19.8946	2.8749	5.2810	4.2660	2.3037	16.7258	2.6687	3.5244	3.0745	2.1984
m132	2441	190	197.7770	4.8146	3.3667	26.6992	5.0072	8.1220	6.8112	4.0930	23.2344	4.7675	5.8406	5.3041	4.0666
m65	3966	33	201.8811	4.9910	3.3018	116.665	9.7811	21.7384	15.6723	3.4665	91.5066	7.7424	12.2897	9.7855	3.2314

Table 10: Results of the experiments on benchmarks taken from (Neider et al., 2019).

7. Threats to Validity

We designed experiments and evaluated the results considering several threats to validity. On the design side, we try to avoid any issues by applying the following: each randomly generated automaton is checked if it can be synchronized via the polynomial-time algorithm described in (Eppstein, 1990). As the synchronizing sequence is computed in parts at each step, we check the lengths of subsequences and the number of active states. Finally, we also check if the computed synchronizing sequence w is correct by computing $\delta(S, w)$.

To understand the representativeness of the performance samples, we measured the dispersion on the sampled values. For each algorithm, the number of states and number of inputs, we computed the coefficient of variation (CV), i.e., the ratio of the sample standard deviation to the sample mean. For the sequential implementations (ORG, PC, and Lazy), the average CV value is 5.4%, where the standard deviations of these CV values for each algorithm are 5.8%, 3.2%, and 3.4%, respectively. For the GPU implementations, the average CV values are in between 0.5% and 6.5%. Furthermore, the maximum CV values range between 1% and 19.7% where the standard deviations are between 0.3% and 5.9%. Hence for all algorithms and possible parameter sets, whose averages are reported in the paper, the dispersion among the random trials is low.

In this paper, we consider the speedup values over our SynchroP implementation (ORG). This raises the question of, how well the implementation of ORG is. In order to evaluate the time performance of ORG objectively, we compared the execution times of ORG and another implementation of SynchroP from the literature (Roman & Szykula, 2015). The results of these comparisons are given in Table 11. The figures in the table show that the performance of our original implementation is comparable to the state-of-the-art used in the literature. These experiments were performed on a machine running on Ubuntu 16.04.2 equipped with a 16GB of memory and Intel Xeon E5-1650 clocked at 3.20GHz. The code was written in C++ and compiled using `gcc` version 5.4.0 with `-O3` option enabled for both ORG and baseline.

		n				
p		128	256	512	1024	2048
2	Baseline	0.10	0.70	6.68	84.97	1153.45
	ORG	0.04	0.31	2.46	28.22	361.46
8	Baseline	0.13	0.96	10.61	142.69	2308.99
	ORG	0.09	0.57	5.78	79.71	1361.91
32	Baseline	0.15	1.09	12.99	182.52	2963.47
	ORG	0.11	0.81	8.84	129.04	2358.22

Table 11: Execution times (in seconds) of our original CPU implementation (ORG) and another implementation (baseline) from the literature (Roman & Szykula, 2015).

8. Conclusion

The synchronization problem has several applications in practice, and although finding a synchronizing word is easy, it is hard to find the shortest one. Even a short word is hard to find, especially when the automaton is large. There exist many synchronizing heuristics in the literature. However, high-quality heuristics such as SynchroP producing relatively shorter sequences are very expensive and can take hours, especially when the automaton has tens of thousands of states. In this work, we focus on boosting high-quality but slow synchronizing heuristics in the literature and propose a GPU implementation, which is more than 1000× faster compared to the original proposal implemented on CPU.

Acknowledgements

This work was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) [grant number 114E569]. This research was supported in part by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award). We would like to thank the authors of (Roman & Szykula, 2015) for providing their heuristics implementations, which we used to compare our SynchroP implementation as given in Table 11.

References

Ananichev, D. S., Volkov, M. V., & Zaks, Y. I. (2006). Synchronizing automata with a letter of deficiency 2. In O. H. Ibarra, & Z. Dang (Eds.), *Developments in Language Theory* (pp. 433–442). Berlin, Heidelberg: Springer Berlin Heidelberg.

Broy, M., Jonsson, B., Katoen, J., Leucker, M., & Pretschner, A. (Eds.) (2005). *Model-Based Testing of Reactive Systems, Advanced Lectures* volume 3472 of *Lecture Notes in Computer Science*. Springer.

Černỳ, J. (1964). Poznámka k homogénnym experimentom s konečnỳmi automatmi. *Matematicko-fyzikálny časopis*, *14*, 208–216.

Černỳ, J., Pirická, A., & Rosenauerová, B. (1971). On directable automata. *Kybernetika*, *7*, 289–298.

Cho, H., Jeong, S.-W., Somenzi, F., & Pixley, C. (1993). Multiple observation time single reference test generation using synchronizing sequences. In *Design Automation, 1993, with the European Event in ASIC Design. Proceedings.[4th] European Conference on* (pp. 494–498). IEEE.

Don, H., Zantema, H., & de Bondt, M. (2020). Slowly synchronizing automata with fixed alphabet size. *Information and Computation*, (p. 104614).

Eppstein, D. (1990). Reset sequences for monotonic automata. *SIAM J. Comput.*, *19*, 500–510.

Jourdan, G., Ural, H., & Yenigün, H. (2015). Reduced checking sequences using unreliable reset. *Inf. Process. Lett.*, *115*, 532–535.

Karahoda, S., Erenay, O. T., Kaya, K., Türker, U. C., & Yenigün, H. (2016). Parallelizing heuristics for generating synchronizing sequences. In *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings* (pp. 106–122). Cham: Springer International Publishing.

Karahoda, S., Erenay, O. T., Kaya, K., Türker, U. C., & Yenigün, H. (2020). Multicore and manycore parallelization of cheap synchronizing sequence heuristics. *J. Parallel Distributed Comput.*, *140*, 13–24.

Karahoda, S., Kaya, K., & Yenigün, H. (2018). Synchronizing heuristics: Speeding up the fastest. *Expert Systems with Applications*, *94*, 265 – 275.

Kowalski, J., & Szykula, M. (2013). A new heuristic synchronizing algorithm. *CoRR*, *abs/1308.1978*.

Kudlacik, R., Roman, A., & Wagner, H. (2012). Effective synchronizing algorithms. *Expert Systems with Applications*, *39*, 11746–11757.

Lee, D., & Yannakakis, M. (1996). Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, *84*, 1090–1123.

Natarajan, B. K. (1986). An algorithmic approach to the automated design of parts orienters. In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986* (pp. 132–142). IEEE Computer Society.

Neider, D., Smetsers, R., Vaandrager, F., & Kuppens, H. (2019). Benchmarks for automata learning and conformance testing. In T. Margaria, S. Graf, & K. G. Larsen (Eds.), *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday* (pp. 390–416). Cham: Springer International Publishing.

Olschewski, J., & Ummels, M. (2010). The complexity of finding reset words in finite automata. In *International Symposium on Mathematical Foundations of Computer Science* (pp. 568–579). Springer.

Roman, A. (2009). Synchronizing finite automata with short reset words. *Applied Mathematics and Computation*, *209*, 125–136.

Roman, A., & Szykula, M. (2015). Forward and backward synchronizing algorithms. *Expert Systems with Applications*, *42*, 9512–9527.

Szykula, M. (2018). Improving the upper bound on the length of the shortest reset word. In R. Niedermeier, & B. Vallée (Eds.), *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France* (pp. 56:1–56:13). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik volume 96 of *LIPIcs*.

Volkov, M. V. (2008). Synchronizing automata and the Černỳ conjecture. In *In-*

ternational Conference on Language and Automata Theory and Applications (pp. 11–27). Springer.

Volkov, M. V. (2019). Slowly synchronizing automata with idempotent letters of low rank. *J. Autom. Lang. Comb.*, *24*, 375–386.