# ALGORITHMIC OPTIMIZATION AND PARALLELIZATION OF EPPSTEIN'S SYNCHRONIZING HEURISTIC

by

SERTAÇ KARAHODA

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University
July, 2018

# ALGORITHMIC OPTIMIZATION AND PARALLELIZATION OF EPPSTEIN'S SYNCHRONIZING HEURISTIC

APPROVED BY:

Assoc. Prof. Dr. Hüsnü Yenigün
(Thesis Supervisor)

Asst. Prof. Dr. Kamer Kaya
(Thesis Co-Advisor)

Assoc. Prof. Dr. Esra Erdem Patoğlu

Assoc. Prof. Dr. Cemal Yılmaz

Asst. Prof. Dr. Zafeirakis Zafeirakopoulos

DATE OF APPROVAL: 31/07/2018

**ABSTRACT**

# ALGORITHMIC OPTIMIZATION AND PARALLELIZATION OF EPPSTEIN'S SYNCHRONIZING HEURISTIC

SERTAÇ KARAHODA

Computer Science and Engineering, Master's Thesis, 2018

Thesis Supervisor: Hüsnü Yenigün

Thesis Co–Supervisor: Kamer Kaya

Keywords: Finite state automata, Synchronizing words, Synchronizing heuristics, CPU parallelization, GPU parallelization

Testing is the most expensive and time consuming phase in the development of complex systems. Model–based testing is an approach that can be used to automate the generation of high quality test suites, which is the most challenging part of testing. Formal models, such as finite state machines or automata, have been used as specifications from which the test suites can be automatically generated. The tests are applied after the system is synchronized to a particular state, which can be accomplished by using a synchronizing word. Computing a shortest synchronizing word is of interest for practical purposes, e.g. for a shorter testing time. However, computing a shortest synchronizing word is an NP–hard problem. Therefore, heuristics are used to compute short synchronizing words. GREEDY is one of the fastest synchronizing heuristics currently known. In this thesis, we present approaches to accelerate GREEDY algorithm. Firstly, we focus on parallelization of GREEDY. Second, we propose a lazy execution of the preprocessing phase of the algorithm, by postponing the preparation of the required information until it is to be used in the reset word generation phase. We suggest other algorithmic enhancements as well for the implementation of the heuristics. Our experimental results show that depending on the automata size, GREEDY can be made $500\times$ faster. The suggested improvements become more effective as the size of the automaton increases.

# ÖZET

## EPPSTEIN'IN SIFIRLAMA SEZGİSELİNİN ALGORİTMİK ENİYİLEMESİ VE PARALELLEŞTİRİLMESİ

SERTAÇ KARAHODA

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2018

Tez Danışmanı: Hüsnü Yenigün

Tez Eşdanışmanı: Kamer Kaya

Anahtar Kelimeler: Sonlu durum otomatları, Sıfırlama kelimeleri, Sıfırlama Sezgiselleri, AİÜ paralelleştirilmesi, GİÜ paralelleştirilmesi

Karmaşık sistemlerin geliştirilmesinde, test etme en pahalı ve en çok zaman alan evredir. Model tabanlı testler yüksek kaliteli deney kurgusunu otomatik üretmede kullanılan yaklaşımlardan birisidir. Deney kurgusunu otomatik üretme test etmenin en zorlu parçalarından biridir. Sonlu durum makineleri ya da özdevinimler gibi biçimsel modeller, otomatik deney grubunu üretmek için kullanılmaktadır. Sistem belirli bir duruma senkronize edildikten sonra testler uygulanır ve bu belirli duruma gelebilmek için sıfırlama kelimeleri kullanılmaktadır. Daha kısa deney süreleri için en kısa sıfırlama kelimesini hesaplamak önemlidir, ancak en kısa sıfırlama kelimesini hesaplamak NP–hard bir problemdir. Bu nedenle kısa sıfırlama kelimelerini hesaplamak için sezgisel yöntemler kullanılmaktadır. GREEDY algoritması bu alanda bilinen en hızlı sezgisel algoritmadır. Bu tezde, GREEDY algoritmasını hızlandıran yaklaşımlar sunulmaktadır. İlk olarak GREEDY algoritmasının paralelleştirilmesine odaklanılmaktadır. İkinci olarak ise tembel bir yaklaşım önererek sıfırlama kelimesinin üretilmesi için gerekli bilgilerin hazırlanma süreci ertelenmektedir. Aynı zamanda, GREEDY algoritması için benzer algoritmik iyileştirilmeler önerilmektedir. Deney sonuçlarımız özdevinim büyüklüğüne bağlı olarak GREEDY algoritmasının 500 kat daha hızlı hale getirilebileceğini göstermektedir. önerilen geliştirmeler özdevinim büyüklüğü arttıkça daha etkili hale gelmektedir.

## ACKNOWLEDGMENTS

I would like to state my gratitude to my supervisors, Hüsnü Yenigün and Kamer Kaya for everything they have done for me, especially for their invaluable guidance, limitless support and understanding.

The financial support of Sabanci University is gratefully acknowledged.

I would like to thank TUBITAK 114E569 project for the financial support provided.

## CONTENTS


**1 INTRODUCTION** **1**

**2 PRELIMINARIES** **4**
2.1 Graphics Processing Units and CUDA . . . . . 6

**3 EPPSTEIN'S GREEDY ALGORITHM** **8**
3.1 Analysis on GREEDY . . . . . 11

**4 PARALLELIZATION ON GREEDY** **14**
4.1 Frontier to Remaining in Parallel . . . . . 14
4.2 Remaining to Frontier . . . . . 16
4.3 Hybrid Approach . . . . . 17
4.4 Searching from the Entire Set . . . . . 19
4.5 Parallelization of the Second Phase . . . . . 20
4.6 Implementation Details . . . . . 21

**5 SPEEDING UP THE FASTEST** **23**
5.1 Lazy PMF Construction . . . . . 23
5.2 Looking Ahead from the Current Pair . . . . . 25
5.3 Reverse Intersection of the Active Pairs and PMF . . . . . 28

**6 EXPERIMENTAL RESULTS** **29**
6.1 Multicore Parallelization of PMF Construction . . . . . 29
6.2 Second Phase Parallelization . . . . . 35
6.3 Speeding up the Fastest . . . . . 36

**7 CONCLUSION AND FUTURE WORK** **39**

## LIST OF FIGURES


2.1 A synchronizing automaton $\mathcal{A}$ (left), and the data structures to store and process the transition function $\delta^{-1}$ in memory (right). For each symbol $x \in \Sigma$, we used two arrays `ptrs` and `ids` where the former is of size $n+1$ and the latter is of size $n$. For each state $s \in S$, `ptrs`$[s]$ and `ptrs`$[s+1]$ are the start (inclusive) and end (exclusive) pointers to two `ids` entries. The array `ids` stores the ids of the states $\delta^{-1}(s, x)$ in between `ids[ptrs[`$s$`]]` and `ids[ptrs[`$s+1$`] - 1]`. . . . . 4

2.2 The pair automaton $\mathcal{A}^{\langle 2 \rangle}$ of the automaton in Figure 2.1. . . . . 5

3.1 The percentage of nodes at each level in PMF . . . . . 13

4.1 The number of frontier and remaining vertices at each BFS level and the corresponding execution times of F2R and R2F while constructing the PMF $\tau$ for $n = 2000$ and $p = 8$ (top) and $p = 128$ (bottom). . . . . 18

4.2 Indexing and placement of the state pair arrays. A simple placement of the pairs (on the left) uses redundant places for state pairs $\{s_i, s_j\}$, $i \neq j$, e.g., $\{s_1, s_2\}$ and $\{s_2, s_1\}$ in the figure. On the right, the indexing mechanism we used is shown. . . . . 22

5.1 The figure summarizes the lookahead process: The BFS forest (the top part of the figure) is being constructed via $\delta^{-1}$ in a lazy way. However, $P = \{\{s_i, s_j\} | \tau(\{s_i, s_j\})$ *is defined*$\}$ and $C^{\langle 2 \rangle}$ are disconnected. The process tries to find a shortest path from $C^{\langle 2 \rangle}$ to the queue $Q$ (the green colored BFS frontier). As an example, the $\sigma$ path passing through the blue $Q$ pair on the left is not the shortest one since there is a red $Q$ pair on the right which is reachable from the same purple lookahead pair. When the blue node is found, the current lookahead level (consisting of the nodes in $Q_L$) shall be completed to guarantee that the red node does (or does not) exist. . . . . 26

6.1 Speedups obtained with parallel F2R over the sequential PMF construction baseline. . . . . 31

6.2 The speedups of the Hybrid PMF construction algorithms with $p = 2$(a), $8$(b), $32$(c), $128$(d) and $n \in \{2000, 4000, 8000\}$. The $x$-axis shows the number of threads used for the Hybrid execution. The values are computed based on the average sequential PMF construction time over $100$ different automata for each $(n, p)$ pair. . . . . 34

6.3 The speedup values normalized w.r.t. the naive baseline. For each additional improvement, the cumulative speedup is given with stacked columns. 37

## LIST OF TABLES


3.1 Sequential PMF construction time ($t_{PMF}$), and overall time ($t_{ALL}$) in seconds . . . . . 12
3.2 The length of the longest merging sequence in PMF($h_{PMF}$) constructed in the first phase for random automata; maximum ($h_{max}$), and average ($h_{mean}$) lengths for merging sequences, used in the second phase of Algorithm 3. . . . . 12
3.3 The length of the longest merging sequence in PMF($h_{PMF}$) constructed in the first phase for the Černý automata; maximum ($h_{max}$), and average ($h_{mean}$) lengths for merging sequences, used in the second phase of GREEDY. . . . . 12

4.1 Comparison of the run time of Algorithm 4 ($t_{FIND\_MIN}$), i.e., the first sub-phase, and the second phase ($t_{SECOND\_PHASE}$). . . . . 21

6.1 Comparison of the parallel execution times (in seconds) of the PMF construction algorithms. . . . . 32
6.2 The speedups obtained on GREEDY when the memory optimized CUDA implementation of Hybrid PMF construction algorithm is used. . . . . 33
6.3 The execution times (in seconds) of Algorithms 4 and 10. . . . . 35
6.4 The percentage of processed edges . . . . . 38

## LIST OF ALGORITHMS


1 Computing a PMF $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$ . . . . . 9
2 BFS_step (F2R) . . . . . 10
3 Eppstein's GREEDY algorithm . . . . . 11
4 Find_Min . . . . . 11

5 BFS_step_F2R (in parallel) . . . . . 15
6 BFS_step_R2F (in parallel) . . . . . 17
7 Computing a function $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$ (Hybrid) . . . . . 19
8 BFS_step_S2R (in parallel) . . . . . 20
9 BFS_step_S2F (in parallel) . . . . . 20
10 Find_Min (in parallel) . . . . . 21

11 GREEDY algorithm with lazy PMF construction . . . . . 24
12 Looking ahead from $C^{\langle 2 \rangle}$ . . . . . 27

# CHAPTER 1

# INTRODUCTION

A *synchronizing word* $w$ for an automaton $\mathcal{A}$ is a sequence of inputs such that no matter at which state $\mathcal{A}$ currently is, if $w$ is applied, $\mathcal{A}$ is brought to a particular state. Such words do not necessarily exist for every automaton. An automaton with a synchronizing word is called *synchronizing*.

Synchronizing automata have practical applications in many areas. For example in model based testing [3] and in particular, for finite state machine based testing [13], test sequences are designed to be applied at a designated state. The implementation under test can be brought to the desired state by using a synchronizing word. Similarly, synchronizing words are used to generate test cases for synchronous circuits with no reset feature [6]. Even when a reset feature is available, there are cases where reset operations are too costly to be applied. In these cases, a synchronizing word can be used as a compound reset operation [8]. Natarajan [14] puts forward another surprising application area, part orienters, where a part moving on a conveyor belt is oriented into a particular orientation by the obstacles placed along the conveyor belt. The part is in some unknown orientation initially, and the obstacles should be placed in such a way that, regardless of the initial orientation of the part, the sequence of pushes performed by the obstacles along the way makes sure that the part is in a unique orientation at the end. Volkov [25] presents more examples for the applications of synchronizing words together with a survey of theoretical results related to synchronizing automata.

As noted above, not every automaton is synchronizing. As shown by Eppstein [7], checking if an automaton with $n$ states and $p$ letters is synchronizing can be performed in time $O(pn^2)$. For a synchronizing automaton, finding a shortest synchronizing word (which is not necessarily unique) is of interest from a practical point of view for obvious reasons (e.g. shorter test sequences in testing applications, or fewer number of obstacles for parts orienters, etc.).

The problem of finding the length of a shortest synchronizing word for a synchronizing automaton has been a very interesting problem from a theoretical point of view as well. This problem is known to be `NP-hard` [7], and `coNP-hard` [16]. The methods to find shortest synchronizing words scale up to a couple of hundreds of states in practice at most [11]. Another interesting aspect of this problem is the following. 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 [4, 5]. Posed half a century ago, the conjecture is still open and claimed to be one of the longest standing open problems in automata theory. Until recently, the best upper bound known for the length of a synchronizing word is $(n^3-n)/6$ by Pin [17]. Currently, the best bound is slightly better than $\frac{114}{685}n^3 + O(n^2)$ as provided by Szykuła [21].

Due to the hardness results given above for finding shortest synchronizing words, there exist heuristics in the literature, known as *synchronizing heuristics*, to compute short synchronizing words. Among such heuristics are GREEDY by Eppstein [7], CYCLE by Trahtman [22], SYNCHROP by Roman [18], SYNCHROPL by Roman [18], FASTSYNCHRO by Kudłacik et al. [12], and forward and backward synchronization heuristics by Roman and Szykuła [19]. In terms of complexity, these heuristics are ordered as follows: GREEDY/CYCLE with time complexity $O(n^3 + pn^2)$, FASTSYNCHRO with time complexity $O(pn^4)$, and finally SYNCHROP/SYNCHROPL with time complexity $O(n^5 + pn^2)$ [18, 12], where $n$ is the number of states and $p$ is the size of the alphabet. This ordering with respect to the worst case time complexity is the same if the actual performance of the algorithms are considered (see for example [12, 19] for experimental comparison of the performance of these algorithms).

The fastest synchronizing heuristics, GREEDY and CYCLE, are also the earliest heuristics that appeared in the literature. Therefore GREEDY and CYCLE are usually considered as a baseline to evaluate the quality and the performance of new heuristics. Newer heuristics do generate shorter synchronizing words, but by performing a more complex analysis, which implies a substantial increase on the runtime. The time performance of GREEDY and CYCLE are unmatched to date.

All synchronizing heuristics consist of a preprocessing phase, followed by reset word generation phase. As presented in this thesis, our initial experiments revealed that the preprocessing phase dominates the runtime of the overall algorithm for GREEDY. We also discovered that the preprocessing computes more information than reset word generation phase needs. To speed up GREEDY without sacrificing the quality of the synchronizing words generated by the heuristic, we propose two main techniques that speedup GREEDY. First, we focused on parallelization of GREEDY. Second, we propose a lazy execution of the preprocessing, by postponing the preparation of the required information until it is to be used in the reset word generation phase. We suggest other algorithmic enhancements as well for the implementation of the heuristics.

To the best of our knowledge, this is the first work towards parallelization of synchronizing heuristics. Although, a parallel approach for constructing a synchronizing sequence for partial automata[1] has been proposed in [23], it is not exact (in the sense that it may fail to find a synchronizing sequence even if at least one exists). Furthermore, it is not a polynomial time algorithm.

The rest of the thesis is organized as follows: in Chapter 2, the notation used in the thesis is introduced, and synchronizing sequences are formally defined. We give the details of Eppstein's GREEDY construction algorithm in Chapter 3. The parallelization approach together with the implementation details are described in Chapter 4. Chapter 5, algorithmic optimizations which avoid most of the redundant computations in the original heuristic are introduced. The results in these two chapters are published in [9] and [10], respectively. Chapter 6 presents the experimental results and Chapter 7 concludes the thesis.

[1] Please see Chapter 2 for the definition of a partial automaton.

# CHAPTER 2

# PRELIMINARIES

FSMs are mathematical abstractions for real word systems. When an FSM gets an input, it moves from one state to another with an output. Since synchronizing sequences consider only the destination state without making any observation on the system, the output is not in the scope of this work. Therefore, we can consider an FSM as an automaton with a simple transition function and without an output.

When an automaton is complete and deterministic, it is defined by a triplet $\mathcal{A} = (S, \Sigma, \delta)$ where $S = \{1, 2, \ldots, n\}$ is a finite set of $n$ states, $\Sigma$ is a finite alphabet consisting of $p$ input symbols (or simply *letters*), and $\delta : S \times \Sigma \to S$ is a total transition function. When the transition function is a partial function, then the automaton is said to be a *partial* automaton.

If the automaton $\mathcal{A}$ is at a state $s$ and if an input $x$ is applied, then $\mathcal{A}$ moves to the state $\delta(s, x)$. Figure 2.1 (left) shows an example automaton $\mathcal{A}$ with 4 states and 2 input.

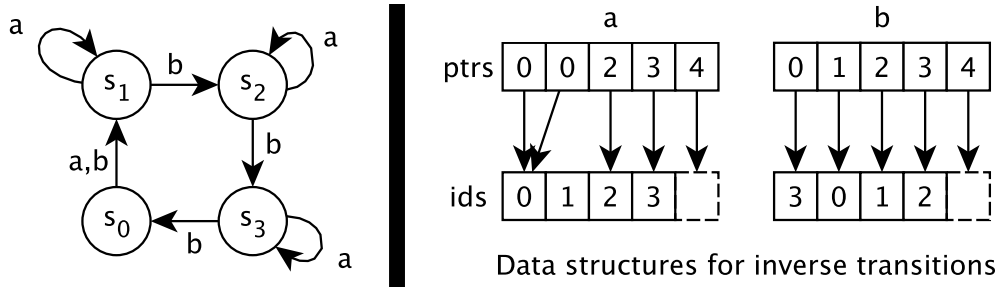



Figure 2.1: A synchronizing automaton $\mathcal{A}$ (left), and the data structures to store and process the transition function $\delta^{-1}$ in memory (right). For each symbol $x \in \Sigma$, we used two arrays `ptrs` and `ids` where the former is of size $n + 1$ and the latter is of size $n$. For each state $s \in S$, `ptrs`$[s]$ and `ptrs`$[s + 1]$ are the start (inclusive) and end (exclusive) pointers to two `ids` entries. The array `ids` stores the ids of the states $\delta^{-1}(s, x)$ in between `ids[ptrs[`$s$`]]` and `ids[ptrs[`$s + 1$`] - 1]`.

An element of the set $\Sigma^\star$ is called an *input sequence* (or simply a *word*). $|w|$ denotes the length of $w$, and $\varepsilon$ expresses the empty word. The transition function $\delta$ can be extended to a set of states and to a word in the usual way. Assuming $\delta(s, \varepsilon) = s$, for a word $w \in \Sigma^\star$ and a letter $x \in \Sigma$, $\delta(s, xw) = \delta(\delta(s, x), w)$. Likewise, for a set of states $S' \subseteq S$, $\delta(S', w) = \{\delta(s, w) | s \in S'\}$.

The inverse $\delta^{-1} : S \times X \to 2^S$ of the transition function $\delta$ is also a well defined function; $\delta^{-1}(s, x)$ denotes the set of states with a transition to state $s$ with input $x$. Formally, $\delta^{-1}(s, x) = \{s' \in S | \delta(s', x) = s\}$. Figure 2.1 (right) shows the data structure used to store the inverse transition function for the example automaton.

Let $\mathcal{A} = (S, \Sigma, \delta)$, $C \subseteq S$ and $C^{\langle 2 \rangle} = \{\{s_i, s_j\} | s_i, s_j \in C\}$ be set of multisets with cardinality two. For $\{s_i, s_j\} \in C^{\langle 2 \rangle}$, if $s_i = s_j$ then it is called a *singleton*, otherwise called a *pair*.

An automaton which is produced from the set of pairs $S^{\langle 2 \rangle}$; $\mathcal{A}^{\langle 2 \rangle} = (S^{\langle 2 \rangle}, \Sigma, \delta^{\langle 2 \rangle})$ is called the *pair automaton*. For a pair automaton, the set of inputs is the same and the transition function of the pair automaton is $\delta^{\langle 2 \rangle}(\{s_i, s_j\}, x) = \{\delta(s_i, x), \delta(s_j, x)\}$.

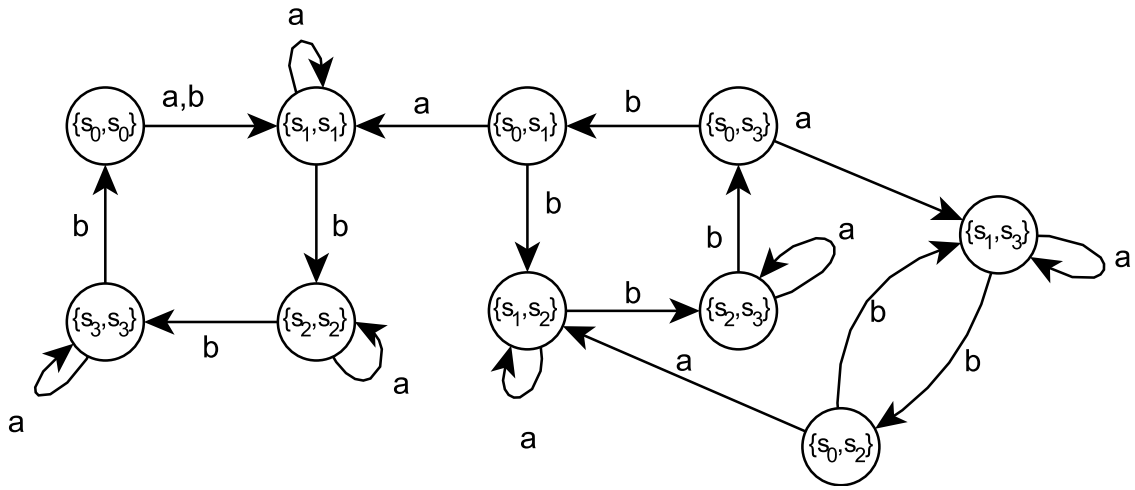



Figure 2.2: The pair automaton $\mathcal{A}^{\langle 2 \rangle}$ of the automaton in Figure 2.1.

Let $C \subseteq S$ be a set of states and $w \in \Sigma^*$ be an input sequence. If the cardinality of $\delta(C, w)$ is one then $w$ is said to be a *merging sequence* for $C$. If there exists a merging sequence for a set of states $C$, then $C$ is called *mergeable*. If there exists a merging sequence $w$ for $S$ (i.e. for all states), $w$ is called a *reset word*[1] of the automaton, and the automaton is called *synchronizable* or *synchronizing*. As shown by [7], deciding if an

[1]In the literature, reset word is also called as both synchronizing word and synchronizing sequence. In this thesis, these three terms are used interchangably.

automaton is synchronizing can be performed in time $O(pn^2)$ by checking if there exists a merging word for $\{s_i, s_j\}$, for all $\{s_i, s_j\} \in S^{\langle 2 \rangle}$. Recently, Berlinkov [2] showed that there exists an algorithm that decides on synchronizability in linear expected time in $n$.

Černý has conjectured that the length of the shortest synchronizing word of an automaton with $n$ states is at most $(n-1)^2$ [24]. Černý has also provided the following class of automata $\mathcal{A}_c$, called *Černý automata*, which hits to this conjectured upper bound.

Let $\mathcal{A}_c = (S, \Sigma_c, \delta_c)$, $\Sigma_c = \{a, b\}$, $|S| = n$, and

$$\delta_c(s_i, x) = \begin{cases} s_{(i+1) \bmod n}, & x = b \text{ or } s_i = s_0 \\ s_i, & \text{otherwise} \end{cases}$$

An example of a Černý automaton is given in Figure 2.1.

## 2.1 Graphics Processing Units and CUDA

At the hardware level, a CUDA capable Graphics Processing Units (GPU) processor is a collection of *multiprocessors* (SMX), each having a number of *processors*. Each multiprocessor has its own shared memory which is common to all its processors. 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. In any given cycle, each processor in the multiprocessor executes the same instruction on different data. Communication between multiprocessors can be achieved through the *global device memory*, which is available to all the processors in all multiprocessors [15].

In 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 collection of threads, called a *warp*, run simultaneously (on a multiprocessor). If the number of threads is more than the warp size then these threads are time-shared internally on the multiprocessor. At any given time, a *block* of threads runs on a multiprocessor. Therefore threads in a block may be bundled into several warps. Each thread executes a piece of code called a *kernel*. The kernel is the core code to be executed on a multiprocessor. During its execution, a thread $t_i$ is given a unique ID and during execution thread $t_i$ can access data residing in the GPU by using its ID. Since the GPU memory is available to all the threads, a thread can access any memory location. During GPU computation the CPU can continue to operate.

Therefore the CUDA programming model is a hybrid computing model in which a GPU is referred as a co-processor (*device*) for the CPU (*host*).

# CHAPTER 3

# EPPSTEIN'S GREEDY ALGORITHM

The GREEDY algorithm is one of the fastest algorithms among the reset word generation heuristics in the literature. The correctness of the algorithm is based on the following proposition (see Theorem 1.14 in the book [3], [20]).

**Proposition 3.0.1** *An automaton $\mathcal{A} = (S, \Sigma, \delta)$ is synchronizing iff $\forall s_i, s_j \in S$, there exists a merging sequence for $\{s_i, s_j\}$.*

GREEDY uses the shortest merging sequences of pairs to find a short reset word. Like most of the algorithms mentioned in Chapter 1, GREEDY has two phases. In the first phase, it finds the shortest merging sequences for all pairs. If there is a pair which is not mergeable, due to Proposition 3.0.1, the automaton is not synchronizing. Otherwise, the algorithm continues with the second phase.

The merging sequences of pairs are stored in a function $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$, which is called the *pairwise merging function* (PMF) for $\mathcal{A}$. If $\{s_i, s_j\}$ is mergeable, then $\tau(\{s_i, s_j\})$ is the merging sequence, otherwise it is undefined. Note that PMF does not have to be unique, i.e., $\tau(\{s_i, s_j\})$ may differ, however $|\tau(\{s_i, s_j\})|$ is unique and the shortest possible. To find all the shortest merging sequences, a breadth first search (BFS) can be initiated over the pair automata. By using the inverse of transition function and starting from $\{s_i, s_i\}$ singletons, all mergeable pairs and their shortest merging sequences can be found. Let $p = |\Sigma|$ and $n = |S|$; in worst case, the algorithm traverses all edges, i.e., $p$ letters of each $n(n-1)$ pairs and $n$ singletons should be checked. Therefore the complexity of the first phase is $O(pn^2)$.

Algorithm 1 keeps track of most recently computed mergeable pairs via a list, which is called *frontier set* ($F$). The level of a frontier set refers to the length of the corresponding merging sequences inside. Since $\tau(\{s_i, s_i\}) = \epsilon$, singletons are placed in the root level, level 0, of BFS. The *remaining set* ($R$) is the set of pairs whose merging sequences are not computed yet. At each iteration of Algorithm 1, new frontier and remaining sets are computed for the next level.

**Algorithm 1:** Computing a PMF $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$

**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$
**output:** A PMF $\tau : S^{\langle 2 \rangle} \to \Sigma^\star$
1 **foreach** *singleton* $\{s, s\} \in S^{\langle 2 \rangle}$ **do** $\tau(\{s, s\}) = \varepsilon$;
2 **foreach** *pair* $\{s_i, s_j\} \in S^{\langle 2 \rangle}$ **do** $\tau(\{s_i, s_j\}) =$ *undefined*;
3 $F \longleftarrow \{\{s, s\} | s \in S\}$; `// all singletons of` $S^{\langle 2 \rangle}$
4 $R \longleftarrow \{\{s_i, s_j\} | s_i, s_j \in S \wedge s_i \neq s_j\}$; `// all pairs of` $S^{\langle 2 \rangle}$
5 **while** $R$ *is not empty and* $F$ *is not empty* **do**
6 $\quad F, R, \tau \longleftarrow$ BFS_step$(A, F, R, \tau)$;

**Proposition 3.0.2** *Let* $\{s_i, s_j\}$ *be a pair in* $S^{\langle 2 \rangle}$. *If* $w \in \Sigma^*$ *is a merging sequence for* $\delta(\{s_i, s_j\}, x)$ *then* $xw$ *is a merging sequence for* $\{s_i, s_j\}$.

Thanks to the inverse of transition function and Proposition 3.0.2, Algorithm 2 constructs PMF from the most recent frontier set. At lines 3-4, the algorithm searches the pairs which can reach the frontier set pairs by applying a single letter. When the algorithm finds such a pair whose merging sequence has not been defined yet, it marks the pair as the next frontier set's pair for the next iteration and sets its merging sequence. Since the algorithm computes the PMF of the remaining set by using the frontier set, it is called *frontier to remaining* (F2R).

When the first phase is completed, Algorithm 3 first checks if the automaton is synchronizing or not in $O(n^2)$ (lines 2-3). It then initializes the *set of active states* ($C$) as the set of all states and the initial reset word as empty. After that, iteratively, it selects the shortest merging sequence of all active pairs, appends it to reset word, and finally updates the set of active states by applying the selected merging sequence. This operation is repeated until only a single active state is left.

**Algorithm 2:** BFS_step (F2R)

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

```
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 let R' be R \ F';
```

At each iteration, the merging sequence is applied, so the cardinality of $C$ decreases. Therefore, at most $n-1$ iterations are performed. At line 7, the algorithm finds the active pair with the shortest merging sequence which takes $O(n^2)$ per iteration. Line 8 takes constant time. The length of each merging sequence can be at most $n^2$. Therefore the time complexity of line 9 is $O(n^3)$ for a single iteration. Overall, the second phase takes $O(n^4)$ and Algorithm 3 requires $O(pn^2 + n^4)$ time.

The upper bounds of the phases can be computed in a slightly different way. For a synchronizing automaton, the first phase is $\Omega(n^2)$ since it finds a merging sequence for all pairs. At best, phase two takes a merging sequence with length of one, which is also a reset word. Then the algorithm applies the merging sequence to all states. Therefore, the lower bound of the second phase is $\Omega(n)$. Thus Algorithm 3 has $O(pn^2 + n^4)$ and $\Omega(n^2)$ time complexity. Since there is a huge gap between the best and the worst case complexities, we extended our observations with the empirical results. In the next subsection, the bottleneck of the algorithm is introduced with a thorough experimental analysis.

**Algorithm 3:** Eppstein's GREEDY algorithm

**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$
**output:** A reset word $\Gamma$ for $\mathcal{A}$ (or fail if $\mathcal{A}$ is not synchronizable)

compute a PMF $\tau$ using Algorithm 1;
**if** *there exists a pair* $\{s_i, s_j\}$ *such that* $\tau(\{s_i, s_j\})$ *is undefined* **then**
    report that $\mathcal{A}$ is not synchronizable and exit;
$C = S$; `// C will keep track of the current set of states`
$\Gamma = \varepsilon$; `// Γ is the synchronizing sequence to be constructed`
**while** $|C| > 1$ **do** `// we have two or more states yet to be merged`
    $\{s_i, s_j\} = Find\_Min(C, \tau)$;
    $\Gamma = \Gamma\ \tau(\{s_i, s_j\})$;
    $C = \delta(C, \tau(\{s_i, s_j\}))$;

**Algorithm 4:** Find_Min

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

$\{s_i, s_j\}$ = undefined;
**foreach** $\{s_k, s_\ell\} \in C^{\langle 2 \rangle}$ **do**
    **if** $\{s_i, s_j\}$ *is undefined or* $|\tau(\{s_k, s_\ell\})| < |\tau(\{s_i, s_j\})|$ **then**
        $\{s_i, s_j\} = \{s_k, s_\ell\}$

### 3.1 Analysis on GREEDY

As discussed in Chapter 3, the time complexity of GREEDY is $O(pn^2 + n^4)$. For most of the cases, $p$ is too small when compared to $n$. Hence, the complexity of the second phase, $O(n^4)$, dominates the first phase in theory. To analyze the algorithm, we performed experiments on 100 randomly generated automata for each $p \in \{2, 8, 32, 128\}$ letters and $n \in \{2000, 4000, 8000\}$ states. To generate a random automaton, for each state $s$ and input $x$, $\delta(s, x)$ is randomly assigned to a state $s' \in S$. In addition, we used Černý automata [24] for $n \in \{2000, 4000, 8000\}$ states. All the experiments are excuted on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). In Table 3.1, experiments from 1200 randomly generated automata show that the execution time of the second phase does not dominate the overall time of the algorithm for random automata.

| | $n = 2000$ | | | $n = 4000$ | | | $n = 8000$ | | |
|---|---|---|---|---|---|---|---|---|---|
| $p$ | $t_{PMF}$ | $t_{ALL}$ | $\frac{t_{PMF}}{t_{ALL}}$ | $t_{PMF}$ | $t_{ALL}$ | $\frac{t_{PMF}}{t_{ALL}}$ | $t_{PMF}$ | $t_{ALL}$ | $\frac{t_{PMF}}{t_{ALL}}$ |
| 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 |

Table 3.1: Sequential PMF construction time ($t_{PMF}$), and overall time ($t_{ALL}$) in seconds

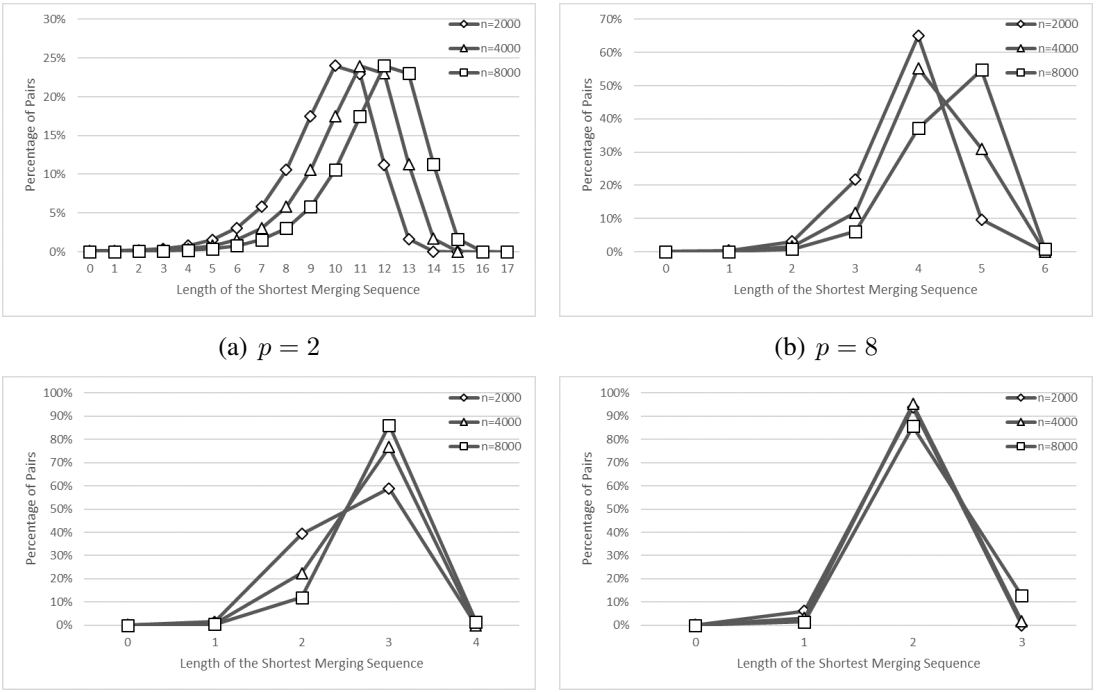
To understand the behavior of the algorithm, we extended our experiments by analyzing the structure of PMF. While computing time complexity of the algorithm, the length of the merging sequence is at most $n^2$. However, Table 3.2 shows that $n^2$ is loose bound for the length of merging sequence. For instance, when automata with 8000 states and 128 letters are considered, the lengths of merging sequences in PMF are at most 3, not 64000000. Another observation is that the second phase tends to pick shorter length of merging sequences. For example, when we take an automaton with 8000 states and 2 letters, the longest merging sequence in PMF has the length 16.9. The second phase uses only merging sequences with length 12.1 and less. Thus, the merging sequences of almost 30% of the nodes are unnecessarily computed (see Figure 3.1).

| | n=2000 | | | n=4000 | | | n=8000 | | |
|---|---|---|---|---|---|---|---|---|---|
| p | $h_{PMF}$ | $h_{max}$ | $h_{mean}$ | $h_{PMF}$ | $h_{max}$ | $h_{mean}$ | $h_{PMF}$ | $h_{max}$ | $h_{mean}$ |
| 2 | 14.2 | 10.0 | 1.9 | 15.5 | 11.2 | 1.9 | 16.9 | 12.1 | 1.9 |
| 8 | 5.0 | 4.0 | 1.3 | 6.0 | 4.2 | 1.3 | 6.0 | 4.6 | 1.3 |
| 32 | 3.1 | 2.7 | 1.1 | 4.0 | 2.9 | 1.1 | 4.0 | 3.0 | 1.1 |
| 128 | 3.0 | 2.0 | 1.0 | 3.0 | 2.0 | 1.0 | 3.0 | 2.1 | 1.0 |

Table 3.2: The length of the longest merging sequence in PMF($h_{PMF}$) constructed in the first phase for random automata; maximum ($h_{max}$), and average ($h_{mean}$) lengths for merging sequences, used in the second phase of Algorithm 3.

| n | $h_{PMF}$ | $h_{max}$ | $h_{mean}$ |
|---|---|---|---|
| 2000 | 1999000.0 | 1952000.0 | 8884.8 |
| 4000 | 7998000.0 | 7808000.0 | 19750.8 |
| 8000 | 31996000.0 | 31232000.0 | 43480.8 |

Table 3.3: The length of the longest merging sequence in PMF($h_{PMF}$) constructed in the first phase for the Černý automata; maximum ($h_{max}$), and average ($h_{mean}$) lengths for merging sequences, used in the second phase of GREEDY.




(a) $p = 2$ (b) $p = 8$

(c) $p = 32$ (d) $p = 128$

Figure 3.1: The percentage of nodes at each level in PMF

With these experiments, we observed that the execution time of PMF construction phase in general dominates the GREEDY algorithm except some special automata classes such as Černý. Therefore, we focused on parallelization of PMF construction, which is explained in Chapter 4. We also noticed that not all information from the first phase is used in the second phase. Based on these observations, various algorithmic improvements that make GREEDY much faster are presented in Chapter 5. But first, we will focus on its parallelization in the next section.

# CHAPTER 4

# PARALLELIZATION ON GREEDY

Our preliminary experimental results show that in general, the PMF construction phase is the bottleneck of GREEDY. The first approach we took to reduce its cost is using parallel algorithms.

Algorithm 1 is a BFS algorithm which starts from singletons and searches the shortest merging sequences of all pairs. The length of the merging sequence for a pair represents the level of the pair in BFS tree. Since the merging sequence of each singleton is $\epsilon$, the algorithm initially sets singletons as level 0 nodes. To find the $k^{th}$ level nodes, Algorithm 2 uses the $k-1^{st}$ level as the frontier set. The cost of processing each pair in the frontier set depends on the cost of inverse transition function $\delta^{-1}$. Likewise, the cost of each iteration depends on the number of pairs in frontier set. Therefore, the cost in each iteration vary.

## 4.1 Frontier to Remaining in Parallel

While finding the $k^{th}$ level pairs (in the next frontier set $F'$), the algorithm has to ensure that all pairs from the $(k-1)^{st}$ level are found. Likewise, for correctness, it needs to process all $k^{th}$ level pairs before processing a pair from the $(k+1)^{st}$ level. Hence, a FIFO-based data structure satisfies these requirements. Since the sequential implementation picks a single pair at a time, a simple queue is more than enough to schedule processing of pairs.

Indeed, using a queue is a flawless method to maintain the dependency between the pairs. However, implementing a parallel version of the algorithm is not that straightforward. Each thread needs to process the pairs from the same level; otherwise, a pair from the next frontier can be processed before another pair in the current frontier and an incorrect PMF can be computed. The problem can be solved if the queue is implemented in a thread-safe manner; that is concurrent insertions and deletions cannot disrupt the

integrated FIFO strategy. However, such an implementation requires expensive synchronization mechanisms such as atomic operations and locks. Since there can be millions of enqueue and dequeue operations to be performed, the queue itself will be the bottleneck. Fortunately, we do not have any restriction on the processing order of the pairs in the same level and a cheaper parallelization approach exists.

**Algorithm 5:** BFS_step_F2R (in parallel)

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

1 **foreach** *thread t* **do** $F'_t \longleftarrow \emptyset$ ;
2 **foreach** $\{s_i, s_j\} \in F$ **in parallel do**
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  $\tau(\{s'_i, s'_j\}) \longleftarrow x\tau(\{s_i, s_j\})$;
7  $F'_t = F'_t \cup \{\{s'_i, s'_j\}\}$;
8 $F' \longleftarrow \emptyset$;
9 **foreach** *thread t* **do** $F' = F' \cup F'_t$ ;
10 let $R'$ be $R \setminus F'$;

The parallel implementation is presented in Algorithm 5. In this algorithm, each pair in $F$ is assigned to a single thread. When a thread finds a new pair whose merging sequence is not decided yet, it pushes it to the new frontier set. Since pushing an item to a set is not an atomic operation, we need to change the process of insertions to the next frontier set. The easiest way is considering the process as a critical region (which can be executed only a single thread at a time). However, as mentioned before, this is not time efficient. Here we implemented a lock-free mechanism. Instead of global $F'$, each thread stores a local $F'$. When all pairs from $F$ are processed, a thread merges local sets $F'$ in a sequential manner. Yet, this lock-free mechanism comes with a drawback. If two threads find the same pair at the same time, which is possible due to concurrency, both threads push it to $F'$ (lines 5-6 of Algorithm 5). Hence, the same pair can exist multiple times in the combined frontier. One can solve this problem with a separate duplicate pair removal process which can be a burden on the performance. For CPU parallelization, our preliminary experiments revealed that at most one in a thousand extra pairs are inserted to $|F'|$. Since duplicate pairs do not effect the correctness of the algorithm, we decided not to perform a costly duplicate pair elimination. Instead, the algorithm processes them

more than once whose time cost is negligible.

Due to duplicate pairs, updating the remaining pair set $R$ becomes a costly operation. In the sequential implementation of Algorithm 1, we were just counting the number of remaining pairs, i.e., $|R|$. However, in the parallel version, correctly counting the number of remaining pairs while allowing duplicate pairs is not possible. A careless implementation can think that all the pairs are processed even if some are still existing. Therefore, in parallelization of Algorithm 1, we do not maintain $R$. Instead, we allow the implementation perform one more iteration in which no updates are detected. Although this approach requires an extra iteration, its cost is also negligible compared to the cost of maintaining $R$.

### 4.2 Remaining to Frontier

Using the frontier set $F$ to construct PMF, as in Algorithms 2 and 5, is the most natural and probably the most common BFS implementation. Another approach, which we call *remaining to frontier* (R2F), is processing the remaining set $R$ for PMF. The main difference is that R2F uses the transition function $\delta$ to iterate the edges of the pair automaton whereas F2R uses $\delta^{-1}$. Thanks to Proposition 3.0.2, this version, presented in Algorithm 6, correctly searches all pairs $\{s_i, s_j\} \in R$ and applies all possible letters $x \in \Sigma$. If the algorithm finds a merging sequence $\delta(\{s_i, s_j\}, x) = w$ (lines 4-6), then it sets $\tau(\{s_i, s_j\}) = xw$ (lines 7-9). Otherwise, the pair is pushed to $R'$ (lines 10-11). Similar to Algorithm 5, Algorithm 6 also uses local sets. Each thread $t$ uses its local remaining set $R'_t$ for a lock-free parallelization. Since each thread processes different pair sets, there is no duplicate pairs in $R'$. Therefore, Algorithm 1 performs one less iteration compared to parallel F2R implementation.

**Algorithm 6:** BFS_step_R2F (in parallel)

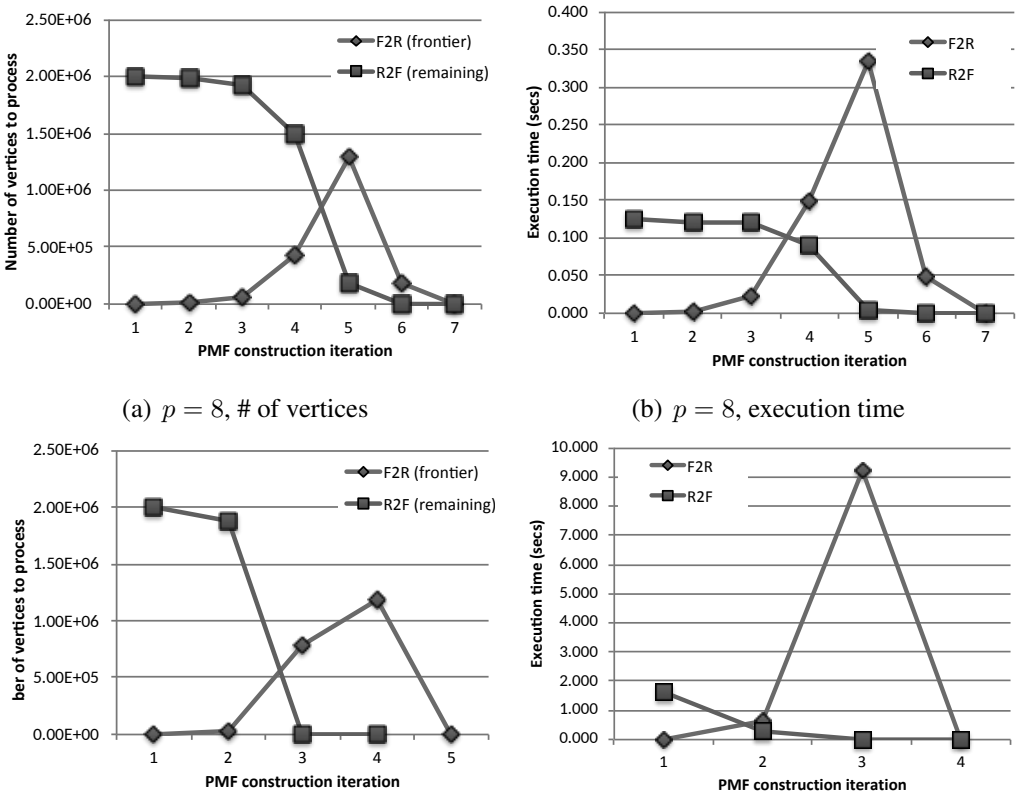
**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$, the frontier $F$, the remaining set $R$, $\tau$
**output:** The new frontier $F'$, the new remaining set $R'$, and updated function $\tau$

```
1  foreach thread t do R'_t ⟵ ∅;
2  foreach {s_i, s_j} ∈ R in parallel do
3      connected ⟵ false;
4      foreach x ∈ Σ do
5          {s'_i, s'_j} ⟵ {δ(s_i, x), δ(s_j, x)};
6          if τ({s'_i, s'_j}) is defined then //  {s'_i, s'_j} ∈ F
7              τ({s_i, s_j}) ⟵ xτ({s'_i, s'_j});
8              connected ⟵ true;
9              break;
10     if not connected then
11         R'_t = R'_t ∪ {{s_i, s_j}};
12 R' ⟵ ∅;
13 foreach thread t do  R' = R' ∪ R'_t ;
14 let F' be R \ R';
```

### 4.3 Hybrid Approach

Per-iteration costs of Algorithms 5 and 6 are closely related to the frontier set $F$ and remaining set $R$ cardinality, respectively. Initially, $R$ is the set of all pairs and $F$ is the set of all singletons. Hence, $|F|$ is much smaller than $|R|$ for the first iteration. In addition, the cardinality of $R$ decreases by $|F|$ at each iteration. In our preliminary experiments, we measured $|F|$ and $|R|$ for each iteration of F2R and R2F, respectively, as well as the execution time per iteration. Figure 4.1 shows the results of these experiments. The figure verifies our predictions; the $R$'s cardinality is larger than $F$'s cardinality for the first few iterations. However, for the later iterations, it is exactly the opposite. Fortunately, at each iteration of PMF construction, it is possible to predict the costs of F2R and R2F variants which allows us to choose the variant with less cost. This is what we call the hybrid approach.

The hybrid approach idea for traditional BFS-based graph traversal is introduced by Beamer et al. [1]. Their algorithm checks all the edges, so determining the cost of each iteration by the number of edges is the most precise technique as in [1]. In our work, the BFS algorithm is applied to a pair automaton $\mathcal{A}^{\langle 2 \rangle}$ which is created in a lazy-manner. That is, we do not have the edges at the beginning and we do not generate them unless we really need them. For each pair, it requires $O(p)$ time to count the number of edges




(a) $p = 8$, # of vertices

(b) $p = 8$, execution time

(c) $p = 128$, # of vertices

(d) $p = 128$, execution time

Figure 4.1: The number of frontier and remaining vertices at each BFS level and the corresponding execution times of F2R and R2F while constructing the PMF $\tau$ for $n = 2000$ and $p = 8$ (top) and $p = 128$ (bottom).

for F2R. Accordingly, estimating the cost of F2R and R2F from the edges takes $O(pn^2)$ which is the same time complexity of the BFS algorithm itself.

For R2F, the number of edges per vertex is fixed to the alphabet size. For F2R, the number of edge per vertex varies. However, the average value is equal to alphabet size, like in R2F. Therefore, assuming the number of edges per vertex approximately equals to the alphabet size is an acceptable approximation. Thus, to simplify the cost estimation, one can use the number of pairs instead of the number of possible transitions to predict the cost of each variant.

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

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

```
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};
4 R ⟵ {{s_i, s_j}|s_i, s_j ∈ S ∧ s_i ≠ s_j};
5 while F is not empty do
6     if |F| < |R| then
7         F, R, τ ⟵ BFS_step_F2R(A, F, R, τ);
8     else
9         F, R, τ ⟵ BFS_step_R2F(A, F, R, τ);
```

### 4.4 Searching from the Entire Set

In both Algorithms 5 and 6, each thread uses a local set. At the end of each BFS step, the algorithm merges the local sets to construct the global set. One 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 synchronization constructs, for each local frontier set, 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. Hence, the previous approach we took 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 visited by more than a single thread increases with the number of threads. Therefore, we need another approach instead of the local set mechanism.

For GPU parallelization, the algorithm processes the entire pair set $S^{\langle 2 \rangle}$, instead of $R$ or $F$. We call this approach S2R and S2F, respectively. At each iteration of S2R, $S^{\langle 2 \rangle}$ 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.

**Algorithm 8:** BFS_step_S2R (in parallel)

**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$, the frontier level $f$, $\tau$
**output:** updated function $\tau$

```
1 foreach {s_i, s_j} ∈ S^2 in parallel do
2     if |τ({s_i, s_j})| = f then
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});
```

**Algorithm 9:** BFS_step_S2F (in parallel)

**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$, $\tau$
**output:** updated function $\tau$

```
1 foreach {s_i, s_j} ∈ S^2 in parallel do
2     if τ({s_i, s_j}) is undefined then
3         foreach x ∈ Σ do
4             {s'_i, s'_j} ⟵ {δ(s_i, x), δ(s_j, x)};
5             if τ({s'_i, s'_j}) is defined then //  {s'_i, s'_j} ∈ F
6                 τ({s_i, s_j}) ⟵ xτ({s'_i, s'_j});
7                 break;
```

### 4.5 Parallelization of the Second Phase

As Table 3.1 demonstrates, the execution time of the second phase is negligible for random automata. However, it is not the case for slowly synchronizing automata. Our experiments indicate that the execution time for the second phase dominates the overall time for Černý automata. Hence, parallelizing the first phase is not sufficient to obtain significant speedups. In this section, the parallelization of the second phase is introduced.

The second phase of the algorithm has two major sub-phases: 1) finding a pair having the minimum length merging sequence (Algorithm 4) and 2) 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 4.1 presents the averages of these executions.

| n | $t_{FIND\_MIN}$ | $t_{SECOND\_PHASE}$ | $\frac{t_{FIND\_MIN}}{t_{SECOND\_PHASE}}$ |
|---|---|---|---|
| 2000 | 4.729 | 4.741 | 0.997 |
| 4000 | 41.034 | 41.098 | 0.998 |
| 8000 | 1035.093 | 1035.48 | 1.000 |

Table 4.1: Comparison of the run time of Algorithm 4 ($t_{FIND\_MIN}$), i.e., the first sub-phase, and the second phase ($t_{SECOND\_PHASE}$).

The table shows that Algorithm 4 dominates the execution time of the second phase. Fortunately, the sub-phase is pleasingly parallelizable. The algorithm distributes the set $C^{\langle 2\rangle}$ to the threads. Each thread finds a local minimum in parallel which are then sequentially merged to obtain a global minimum.

**Algorithm 10:** Find_Min (in parallel)

**input :** Current set of state $C$ and the PMF function $\tau$
**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 **foreach** *thread t* **do** $\{s_{i_t}, s_{j_t}\} =$ undefined ;
2 **foreach** $\{s_k, s_\ell\} \in C^{\langle 2\rangle}$ **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  $\{s_{i_t}, s_{j_t}\} = \{s_k, s_\ell\}$
5 $\{s_i, s_j\} =$ undefined;
6 **foreach** *thread t* **do**
7  **if** $\{s_i, s_j\}$ *is undefined or* $|\tau(\{s_{i_t}, s_{j_t}\})| < |\tau(\{s_i, s_j\})|$ **then**
8  $\{s_i, s_j\} = \{s_{i_t}, s_{j_t}\}$

### 4.6 Implementation Details

To store and utilize the $\delta^{-1}(s, x)$ for all $x \in \Sigma$ and $s \in S$, we employ the data structures in Fig. 2.1 (right). For each symbol $x \in \Sigma$, we used two arrays `ptrs` and `ids` where the former is of size $n+1$ and the latter is of size $n$. For each state $s \in S$, `ptrs`$[s]$ and `ptrs`$[s+1]$ are the start (inclusive) and end (exclusive) pointers to two `ids` entries. The array `ids` stores the ids of the states $\delta^{-1}(s, x)$ in between `ids[ptrs[`$s$`]]` and `ids[ptrs[`$s+1$`] - 1]`. This representation has a low memory footprint. Furthermore, we access the entries in the order of their array placement in our implementation hence, it is also good for spatial locality. We also sorted the set of current pairs by their indexes before line 2 of Algorithm 10 since it improves spatial locality further.

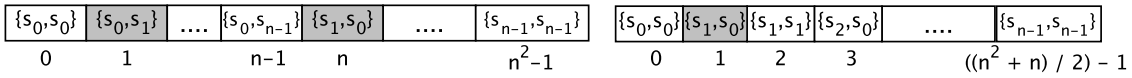



Figure 4.2: Indexing and placement of the state pair arrays. A simple placement of the pairs (on the left) uses redundant places for state pairs $\{s_i, s_j\}$, $i \neq j$, e.g., $\{s_1, s_2\}$ and $\{s_2, s_1\}$ in the figure. On the right, the indexing mechanism we used is shown.

The memory complexity of the algorithms investigated in this study is $O(n^2)$. For each pair of states, we need to employ an array to store the length of the shortest merging sequence. To do that one can allocate an array of size $n^2$, Fig. 4.2 (left), and given the array index $\ell = (i-1) \times n + j$ for a state pair $\{s_i, s_j\}$ where $1 \leq i \leq j \leq n$, she can obtain the state ids by $i = \lceil \frac{\ell}{n} \rceil$ and $j = \ell - ((i-1) \times n)$. This simple approach effectively uses only the half of the array since for a state pair $\{s_i, s_j\}$, a redundant entry for $\{s_j, s_i\}$ is also stored. In our implementation, Fig. 4.2 (right), we do not use redundant locations. For an index $\ell = \frac{i \times (i+1)}{2} + j$ the state ids can be obtained by $i = \lfloor \sqrt{1 + 2\ell} - 0.5 \rfloor$ and $j = \ell - \frac{i \times (i+1)}{2}$. Preliminary experiments show that this approach, which does not suffer from the redundancy, also have a positive impact on the execution time. That being said, all of our algorithms use it and this improvement has no effect to their relative performance.

Due to architecture of GPU, algorithms that require less synchronization are more efficient. Since the number of threads is too high, creating frontier and remaining sets is an inefficient operation. Therefore we implemented S2R and S2F algorithms. For the CUDA version, each thread checks only one pair. To match pairs and threads, the above memory indexing formula is used. Algorithm 10 uses constant number of threads. Each thread finds its local minimum, as in S2R and S2F. When a thread is done, it uses CUDA's `atomicMin` operation to update the global minimum instead of sequential synchronization.

# CHAPTER 5

# SPEEDING UP THE FASTEST

As mentioned in Chapter 3, GREEDY has two main phases: PMF construction and reset word generation. The observations from Section 3.1 shows that in general, i.e., if the automata is not slowly synchronizing, the first phase dominates the execution time of the algorithm. However, to construct the reset word, the second phase does not use all the merging sequences obtained in the first phase. Therefore, the second phase can use a partial PMF. This observation establishes the base of our first optimization.

In this chapter, we propose three algorithmic enhancements for GREEDY algorithm. For the first improvement, the PMF construction is performed in a lazy manner, which is introduced in Section 5.1. Section 5.2 explains the second optimization on searching the merging sequences from a pair and is useful in the later stages. The last optimization, presented in Section 5.3, is minor and uses a basic idea to compute the intersection of the active pair set and a partial PMF.

## 5.1 Lazy PMF Construction

GREEDY algorithm uses PMF to pick a shortest merging sequence among the set of current pairs ($C^{\langle 2 \rangle}$). However, Table 3.2 shows that the algorithm does not need to construct the whole PMF. It is redundant to compute the merging sequences whose length is longer than $h_{max}$. As the first improvement, we generated PMF in a lazy way and combined the two phases into a single one. Algorithm 11 searches a shortest merging sequence in PMF which is also a shortest merging sequence of a pair in $C^{\langle 2 \rangle}$. GREEDY uses the partial PMF which initially contains only the merging sequences of the singletons. At each iteration, a new part of PMF is computed when it is needed. The algorithm checks all pairs in $C^{\langle 2 \rangle}$ to find a shortest merging sequence. If it does not find a merging sequence a PMF construction phase is initiated. This lazy process continues until a pair in $C^{\langle 2 \rangle}$ is

found. After that it applies the merging sequence to all active pairs and continues with the next iteration. Note that the PMF construction is performed in a BFS-manner. Hence, the length of unidentified merging sequences cannot be shorter than the identified merging sequence in PMF.

**Algorithm 11:** GREEDY algorithm with lazy PMF construction

**input :** An automaton $\mathcal{A} = (S, \Sigma, \delta)$
**output:** A synchronizing word $\Gamma$ for $\mathcal{A}$

1 **foreach** *singleton* $\{s, s\} \in S^{\langle 2 \rangle}$ **do** $\tau(\{s, s\}) = \varepsilon$;
2 **foreach** *pair* $\{s_i, s_j\} \in S^{\langle 2 \rangle}$ **do** $\tau(\{s_i, s_j\})$*undefined*;
3 $Q \longleftarrow \{\{s, s\} | s \in S\}$; `// Q is a queue which will store unprocessed pair from frontier set and found pair from next frontier set.`
4 $C = S$; `// C will keep track of the current set of states`
5 $\Gamma = \varepsilon$; `// Γ is the synchronizing sequence to be constructed`
6 **while** $|C| > 1$ **do**
7  **while** $\forall \{s_i, s_j\} \in C^{\langle 2 \rangle} : \tau(\{s_i, s_j\})$ *is undefined* **do**
8  $\{s_i, s_j\}$ = dequeue the next item from $Q$;
9  **foreach** $x \in \Sigma$ **do**
10  **foreach** $\{s_k, s_\ell\} \in \delta^{-1}(\{s_i, s_j\}, x)$ **do**
11  **if** $\tau(\{s_k, s_\ell\})$ *is undefined* **then**
12  $\tau(\{s_k, s_\ell\}) = x\ \tau(\{s_i, s_j\})$;
13  enqueue $\{s_k, s_\ell\}$ onto $Q$;
14  find a pair $\{s_i, s_j\} \in C^{\langle 2 \rangle}$ with a minimum $|\tau(\{s_i, s_j\})|$ among all pairs in $C^{\langle 2 \rangle}$;
15  $\Gamma = \Gamma\ \tau(\{s_i, s_j\})$;
16  $C = \delta(C, \tau(\{s_i, s_j\}))$;

In theory, Algorithm 11 has the same upper bound with Algorithm 3. For lines 7-13, the time complexity depends on the number of edges which is $O(pn^2)$. The number of iterations does not have an impact on the complexity of lines 7-13. Also there is no lower bound for PMF construction. Picking a shortest merging sequence (lines 14-16) is performed in the same way as in Algorithm 3. Therefore, the overall complexity is $O(n^4 + pn^2)$ and $\Omega(n)$. Thus the upper bound does not change but the lower bound decreases.

## 5.2 Looking Ahead from the Current Pair

As Section 5.1 describes, GREEDY does not need a full PMF to find a shortest merging sequence at each iteration. Furthermore, Table 3.2 shows that the average length of a selected merging sequence is less than two. Hence, for most of the iterations, the algorithm picks a length-one merging sequence and it occasionally picks a merging sequence with a longer length, like $h_{max}$. Therefore, although such longer merging sequences are rare, Algorithm 11 still needs to construct PMF up to level $h_{max}$. In this section, a *lookahead* technique is introduced to avoid constructing the deeper levels of PMF. The lookahead approach tries to find a shortest path from $C^{\langle 2 \rangle}$ to $Q$ via $\delta$ (instead of $\delta^{-1}$). Algorithm 12 and Figure 5.1 summarize the basic lookahead process.

Let $\sigma_{\{s_i,s_j\}}$ be the shortest path from $C^{\langle 2 \rangle}$ to $\{s_i, s_j\}$ found by the lookahead process. When $\tau(\{s_i, s_j\})$ is undefined for all $\{s_i, s_j\} \in C^{\langle 2 \rangle}$, performing a lookahead fully and every time will be costly. To reduce the overhead, two constraints are defined; lookahead is allowed only when there are less than MAXSTATES (line 9) states in $C$ and it is allowed to traverse at most MAXPAIRS (line 16) pairs. We did not fine tune these parameters and use MAXSTATES $= \log n$ and MAXPAIRS $= n$.

Although some bookkeeping can be applied, we never reuse the lookahead paths from the previous iteration (see lines 15-16 in Algorithm 12). We forget all the lookahead information once the process ends, because we want to continue exploring the forest level-by-level in a BFS fashion to keep the paths being shortest during the course of the overall heuristic. Furthermore, the current set of pairs changes in every iteration. The algorithm checks different pairs in each iteration. Therefore, the bookkeeping mechanism may not give a remarkable speed up. We decided not to implement the bookkeeping mechanism.

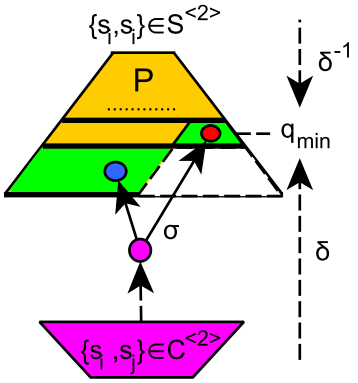



Figure 5.1: The figure summarizes the lookahead process: The BFS forest (the top part of the figure) is being constructed via $\delta^{-1}$ in a lazy way. However, $P = \{\{s_i, s_j\}|\tau(\{s_i, s_j\})$ *is defined*$\}$ and $C^{\langle 2\rangle}$ are disconnected. The process tries to find a shortest path from $C^{\langle 2\rangle}$ to the queue $Q$ (the green colored BFS frontier). As an example, the $\sigma$ path passing through the blue $Q$ pair on the left is not the shortest one since there is a red $Q$ pair on the right which is reachable from the same purple lookahead pair. When the blue node is found, the current lookahead level (consisting of the nodes in $Q_L$) shall be completed to guarantee that the red node does (or does not) exist.

**Algorithm 12:** Looking ahead from $C^{\langle 2 \rangle}$

1..6 `... // same as Algorithm 11`
7 **while** $|C| > 1$ **do**
8  **if** $\forall \{s_i, s_j\} \in C^{\langle 2 \rangle} : \tau(\{s_i, s_j\})$ *is undefined* **then**
9  **if** $|C| <$MAXSTATES **then**
10  let $Q_L = C^{\langle 2 \rangle}$ and $Q_{L_2} = \emptyset$ be two queues;
11  $q_{min} = \min_{\{s_i, s_j\} \in Q} |\tau(\{s_i, s_j\})|$;
12  $cnt = 0$;
13  $\{s_{k'}, s_{\ell'}\} =$ *undefined*;
14  $\{s_{k'_{root}}, s_{\ell'_{root}}\} =$ *undefined*;
15  **foreach** $\{s_i, s_j\} \in S^{\langle 2 \rangle}$ **do** $\sigma(\{s_i, s_j\}) =$ *undefined*;
16  **foreach** $\{s_i, s_j\} \in C^{\langle 2 \rangle}$ **do** $\sigma(\{s_i, s_j\}) = \varepsilon$ and $\{s_{i_{root}}, s_{j_{root}}\} = \{s_i, s_j\}$;
17  **while** $Q_L$ *is not empty* **do**
18  $\{s_i, s_j\} =$ pop an item from $Q_L$;
19  **for** $x \in \Sigma$ **do**
20  $\{s_k, s_\ell\} = \{\delta(s_i, x), \delta(s_j, x)\}$;
21  **if** $\sigma(\{s_k, s_\ell\})$ *is undefined* **then**
22  push $\{k, \ell\}$ to $Q_{L_2}$;
23  $\sigma(\{s_k, s_\ell\}) = x\sigma(\{s_i, s_j\})$;
24  $\{s_{k_{root}}, s_{\ell_{root}}\} = \{s_{i_{root}}, s_{j_{root}}\}$;
25  $cnt++$;
26  **if** $\tau(\{s_k, s_\ell\})$ *is defined and* $|\tau(\{s_k, s_\ell\})| < |\tau(\{s_{k'}, s_{\ell'}\})|$ **then**
27  $\{s_{k'}, s_{\ell'}\} = \{s_k, s_\ell\}$
28  **if** $Q_L$ *is empty and* $cnt <$MAXPAIRS *and* $\{s_{k'}, s_{\ell'}\}$ *is undefined* **then**
29  $Q_L = Q_{L_2}$ and $Q_{L_2} = \emptyset$;
30  **if** $\{s_{k'}, s_{\ell'}\}$ *is defined* **then**
31  use $\{s_{k'_{root}}, s_{\ell'_{root}}\}$ as $\{s_i, s_j\}$ and $\sigma(\{s_{k'}, s_{\ell'}\})\tau(\{s_{k'}, s_{\ell'}\})$ as $\tau(\{s_i, s_j\})$ for line 32-41
32  **while** $\forall \langle s_i, s_j \rangle \in C^{\langle 2 \rangle} : \tau(\{s_i, s_j\})$ *is undefined* **do**
33..38  `... // same as lines 8-13 of Algo. 11`
39..41  `... // same as lines 14-16 of Algo. 11`

### 5.3 Reverse Intersection of the Active Pairs and PMF

Algorithms 11 and 12 frequently check if $\tau(\{s_i, s_j\})$ is undefined or not for all the pairs in $C^{\langle 2 \rangle}$. Our baseline implementation traverses the pairs in $C^{\langle 2 \rangle}$ one by one and checks if they are in $P = \{\{s_i, s_j\} \mid \tau(\{s_i, s_j\})$ *is defined*$\}$. Although this is efficient when $|C^{\langle 2 \rangle}|$ is small, $|C^{\langle 2 \rangle}|$ is not small enough at the early stages of the algorithm. Hence, we reverse the search when $|C^{\langle 2 \rangle}| > |P|$ and traverse the pairs in $P$ and check if they are in $C^{\langle 2 \rangle}$.

# CHAPTER 6

# EXPERIMENTAL RESULTS

All the experiments in the thesis are performed on a single machine running on 64 bit CentOS 6.5 equipped with 64GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz where each socket has 15 cores (30 in total). The machine also has a NVIDIA K40 GPU with 12GB of global memory and 15 SMs each having 192 cores. For the multicore implementations, we used OpenMP and all the codes are compiled with `gcc 4.9.2` with the `-O3` optimization flag enabled and GPU parallelization is achieved with CUDA.

In order to measure the efficiency of the proposed algorithms, we used randomly generated automata with $n \in \{2000, 4000, 8000\}$ states and $p \in \{2, 8, 32, 128\}$ inputs.To generate a random automaton, for each state $s$ and input $x$, $\delta(s, x)$ is randomly assigned to a state $s' \in S$. For each $(n, p)$ pair, we randomly generated $100$ different automata and executed each algorithm on these automata. The values in the figures and the tables are the averages of these $100$ executions for each configuration, i.e., algorithm, $n$ and $p$.

As slowly synchronizing automata, Černý automata with $n \in \{2000, 4000, 8000\}$ states are used for experiments. Since there is only one automaton for each $n$, to reduce the variance on the measured execution times, we execute the algorithms five times for each $n$ value and report the averages.

## 6.1 Multicore Parallelization of PMF Construction

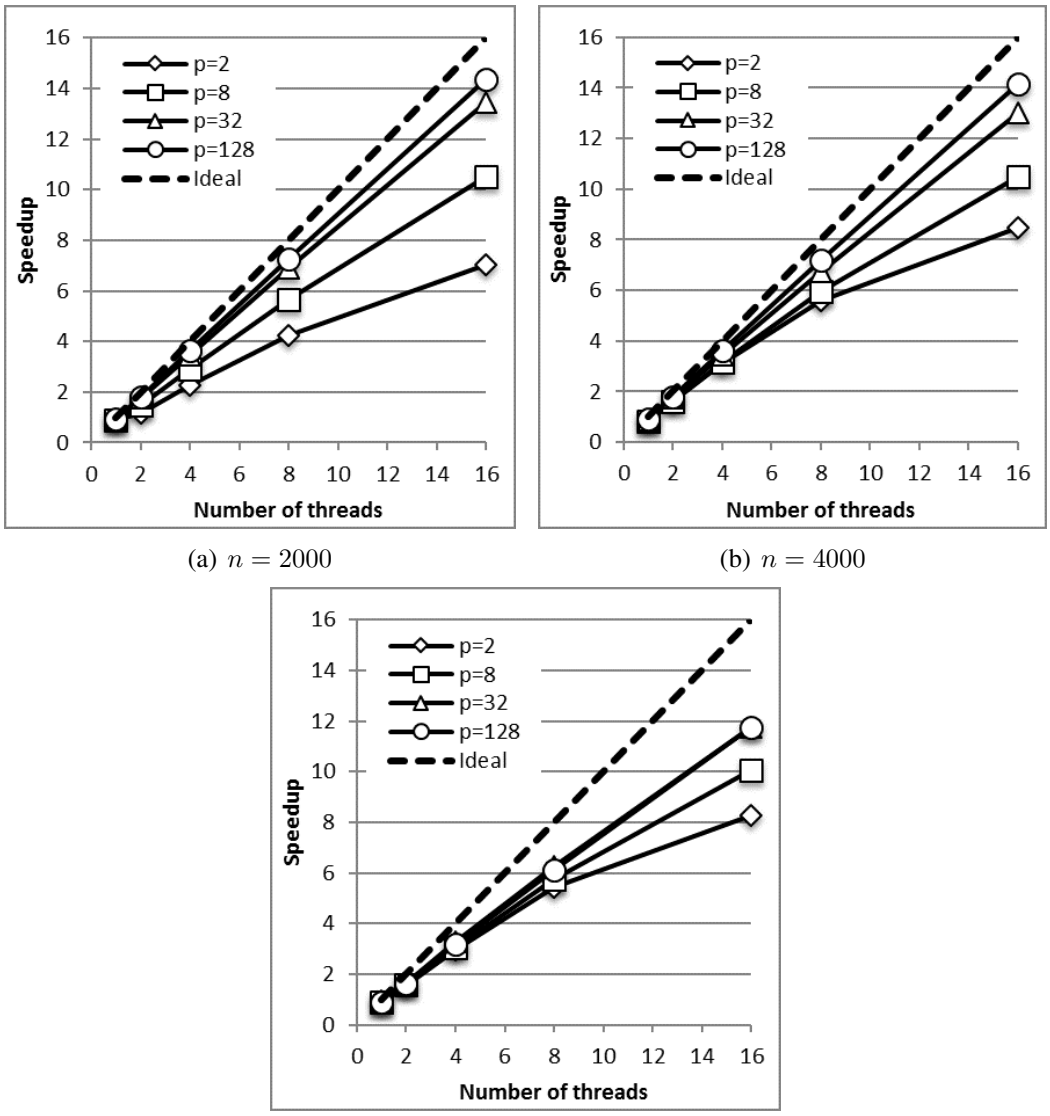
We sequentially implemented Algorithm 3, which is called *sequential* in this section. For the CPU parallelization, we employed the algorithms in Sections 4.1, 4.2 and 4.3. We used 1, 2, 4, 8 and 16 threads for the experiments. To measure the impact of the proposed indexing formula, we implemented GPU parallelization with basic and memory optimized indexing functions. We called these two implementations as CUDA and CUDA

M.O. respectively.

Figure 6.1 shows the speedups of our parallel F2R implementation over the sequential baseline (that has no parallelism). Since F2R uses the same frontier extension mechanism with the sequential baseline, whereas R2F, S2R and S2F employ completely different ones, here we only present the speedup values of F2R. As the figure shows, when $p$ is large, the parallel F2R presents good speedups, e.g., for $p = 128$, the average speedup is $13.4$ with $16$ threads. Furthermore, when compared to the single-thread F2R, the average speedup is $14.9$ with $16$ threads. A performance difference between sequential baseline and single-threaded F2R exists because of the parallelization overhead during the local queue management. Overall, we observed $10\%$ parallelization penalty for F2R on the average over the sequential baseline for all $(n, p)$ pairs.

For $p$ values smaller than $128$, i.e., $2$, $8$, and $32$, the average speedups are $7.9$, $10.4$, and $12.7$, respectively, with $16$ threads. The impact of the parallelization overhead is more for such cases since the amount of the local-queue overhead is proportional to the number of states but not to the number of edges. Consequently, when $p$ decreases the amount of total work decreases and hence, the impact of the overhead increases. Furthermore, since the number of iterations for PMF construction increases with decreasing $p$, the local queues are merged more for smaller $p$ values. Therefore, one can expect more overhead, and hence, less efficiency for smaller $p$ values as the experiments confirm.

Table 6.1 compares the execution times of F2R, R2F, S2R, S2F and Hybrid algorithm for $n \in 2000, 4000, 8000$, $p \in \{2, 8, 32, 128\}$ and $\{1, 2, 4, 8, 16\}$ threads. An interesting observation is that F2R is consistently faster than R2F for $p = 2$, however, it is slower otherwise. This can be explained by the difference in the number of required iterations to construct PMF: when $p$ is large, the frontier expands very quickly and the PMF is constructed in less iterations, e.g., for $n = 2000$, the PMF is generated in $16$ iterations for $p = 2$, whereas only $7$ iterations are required for $p = 8$. Since each edge will be processed once, the runtime of F2R always increases with $p$, i.e., with the number of edges. However, since the frontier expands much faster, the total number of remaining (R-)pairs processed by the R2F throughout the process will probably decrease. Furthermore, since the frontier is large, while traversing the edge list of an R-pair, it is more probable to early terminate the traversal and add the R-pair to the next frontier earlier. Surprisingly, when $p$ increases, these may yield a decrease in the R2F runtime (observe the change




(a) $n = 2000$

(b) $n = 4000$

(c) $n = 8000$

Figure 6.1: Speedups obtained with parallel F2R over the sequential PMF construction baseline.

from $p = 2$ to $p = 8$ in Table 6.1). However, once the performance benefits of early termination are fully exploited, an increase on the R2F runtime with increasing $p$ is more probable since the overall BFS work, i.e., the total number of edges, also increases with $p$ (observe the change from $p = 8$ to $p = 32$ in Table 6.1).

Observing such performance differences for R2F and F2R on automata with different characteristics, the potential benefit of a Hybrid algorithm in practice is more clear. As Table 6.1 shows, the hybrid approach, which is just a combination of F2R and R2F, is almost always faster than employing a pure F2R or a pure R2F BFS-level expansion. Furthermore, we do not need parallelism to observe these performance benefits: the Hybrid

| | | 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 |
| | 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 |
| 1 | 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** |
| | 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 |
| 2 | 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** |
| | 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 |
| 4 | 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** |
| | 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 |
| 8 | 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** |
| | 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 |
| 16 | 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** |
| | 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 |
| CUDA | 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** |
| | 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 |
| CUDA M.O. | 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 |
| | 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 6.1: Comparison of the parallel execution times (in seconds) of the PMF construction algorithms.

approach works better even when a single thread is used at runtime[1]. For example, when $n = 8000$ and $p = 128$, the Hybrid algorithm is $38$ and $15$ times faster than F2R and R2F, respectively. For the same automaton set, the average speedups due to hybridization are $29$ and $16$ with $16$ threads.

When the Hybrid algorithm is used, the speedups on the PMF generation phase are given in Figure 6.2. As the figure shows, thanks to parallelism and good scaling of Hybrid (for large $p$ values), the speedups increase when the number of threads increases. With the CUDA implementation, the PMF generation process becomes even much faster: the memory optimized version obtains $25$, $51$, $143$, and $494$ speedup for $2$, $8$, $32$, and $128$ letter automata, respectively, with $n = 8000$ states.

[1] Although one can implement F2R, R2F, and Hybrid sequentially, we do not have their sequential variants. With their OpenMP-based implementations, we expect $10\%$ parallelization overhead for both R2F and Hybrid as in F2R since the parallelization techniques employed in the implementations are similar.

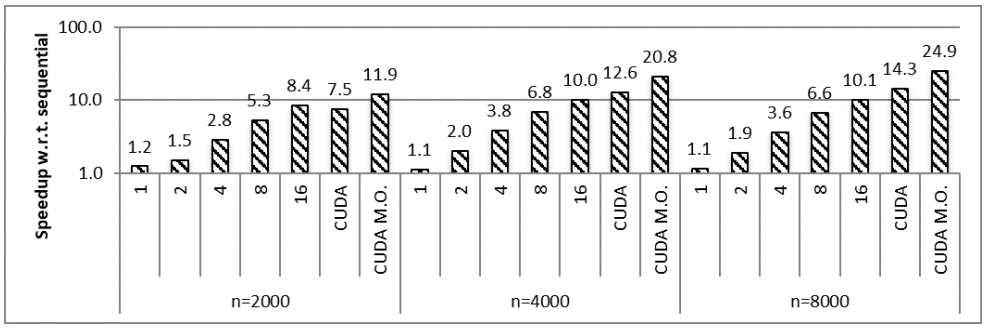
Since we generate the PMF to find a synchronizing sequence, a more realistic evaluation metric would be the performance improvement over the whole sequential reset sequence construction process. As Table 3.1 shows, for Eppstein's GREEDY heuristic (also for some other heuristics such as CYCLE [22]), the PMF generation phase dominates the overall runtime. For this reason, we simply conducted an experiment where the Hybrid approach is used to construct the PMF and no further parallelization is applied during the synchronizing sequence construction phase. Table 6.2 shows the speedups for this experiment for all parallel implementations of Hybrid executions. As the results show, when we compared sequential F2R implementation and hybrid implementation with single thread, more than 2x and more than 14x improvement is possible for $p = 32$ and $p = 128$, respectively.

| | Speedup | | | | $\frac{t_{PMF}}{t_{ALL}}$ | | | |
|---|---|---|---|---|---|---|---|---|
| n\p | 2 | 8 | 32 | 128 | 2 | 8 | 32 | 128 |
| 2000 | 11.90 | 34.20 | 72.99 | 206.84 | 0.52 | 0.53 | 0.69 | 0.77 |
| 4000 | 20.84 | 54.27 | 97.60 | 315.45 | 0.50 | 0.46 | 0.62 | 0.68 |
| 8000 | 24.92 | 51.00 | 143.01 | 493.97 | 0.36 | 0.39 | 0.45 | 0.47 |

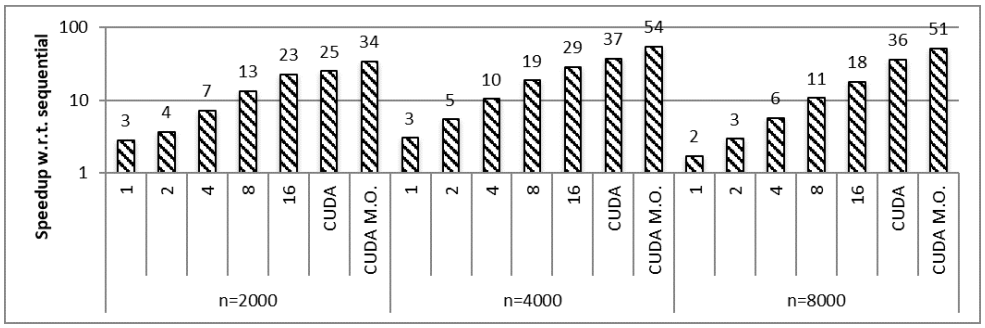
Table 6.2: The speedups obtained on GREEDY when the memory optimized CUDA implementation of Hybrid PMF construction algorithm is used.

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

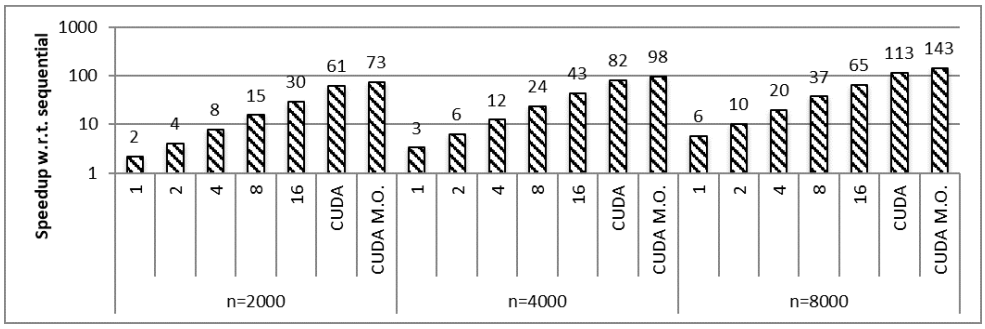
We did not perform an extensive study on larger automata since it takes too long with the sequential baseline implementation. For example, sequential PMF generation takes around 2 hours and 30 minutes for an automaton with 40,000 states and 128 letters, whereas our Hybrid implementation generates a sequence in 2 minutes and 30 seconds.



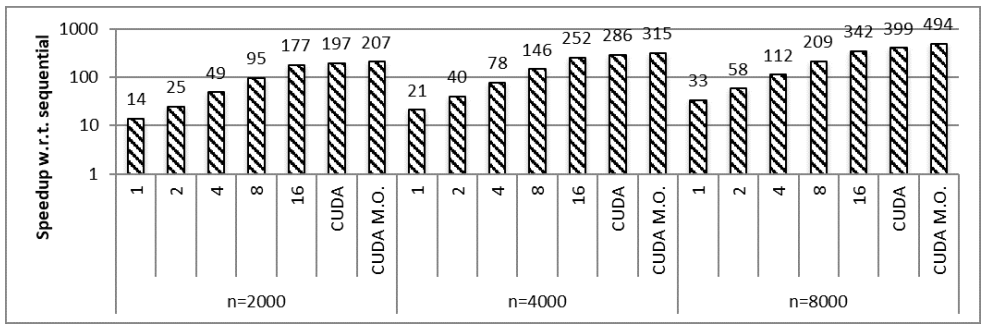

(a) $p = 2$




(b) $p = 8$




(c) $p = 32$




(d) $p = 128$

Figure 6.2: The speedups of the Hybrid PMF construction algorithms with $p = 2$(a), $8$(b), $32$(c), $128$(d) and $n \in \{2000, 4000, 8000\}$. The $x$-axis shows the number of threads used for the Hybrid execution. The values are computed based on the average sequential PMF construction time over $100$ different automata for each $(n, p)$ pair.

### 6.2 Second Phase Parallelization

As mentioned in Section 4.5, the execution time of second phase is worthy to take into account for the slowly synchronizing automata. Hence, we used Černý automata with $n \in \{2000, 4000, 8000\}$ for completeness of the experiments. For CPU parallelization of 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 4.6.

| | n | 2000 | 4000 | 8000 |
|---|---|---|---|---|
| sequential | unsorted | 4.729 | 41.034 | 1035.093 |
| | sorted | 1.604 | 12.701 | 109.758 |
| 1 | unsorted | 5.098 | 47.021 | 896.384 |
| | sorted | 2.553 | 20.274 | 168.116 |
| 2 | unsorted | 3.869 | 37.352 | 874.253 |
| | sorted | 1.935 | 15.085 | 132.770 |
| 4 | unsorted | 2.308 | 22.705 | 522.930 |
| | sorted | 1.178 | 8.946 | 75.673 |
| 8 | unsorted | 1.259 | 13.131 | 289.750 |
| | sorted | 0.719 | 5.044 | 40.842 |
| 16 | unsorted | 0.694 | 6.674 | 154.796 |
| | sorted | 0.723 | 3.350 | 22.403 |
| CUDA | unsorted | 0.684 | 5.514 | 51.280 |
| | sorted | 0.391 | 1.556 | 9.613 |

Table 6.3: The execution times (in seconds) of Algorithms 4 and 10.

Table 6.3 shows that sorting the set of current pairs has a remarkable impact on the performance. Even in sequential implementation, we observed $3\times$ to $9.5\times$ speedups. When both the implementation improvement and GPU parallelization is applied, we observed between $12\times$ and $107\times$ speedups over naive and sequential implementation of the second phase.

### 6.3 Speeding up the Fastest

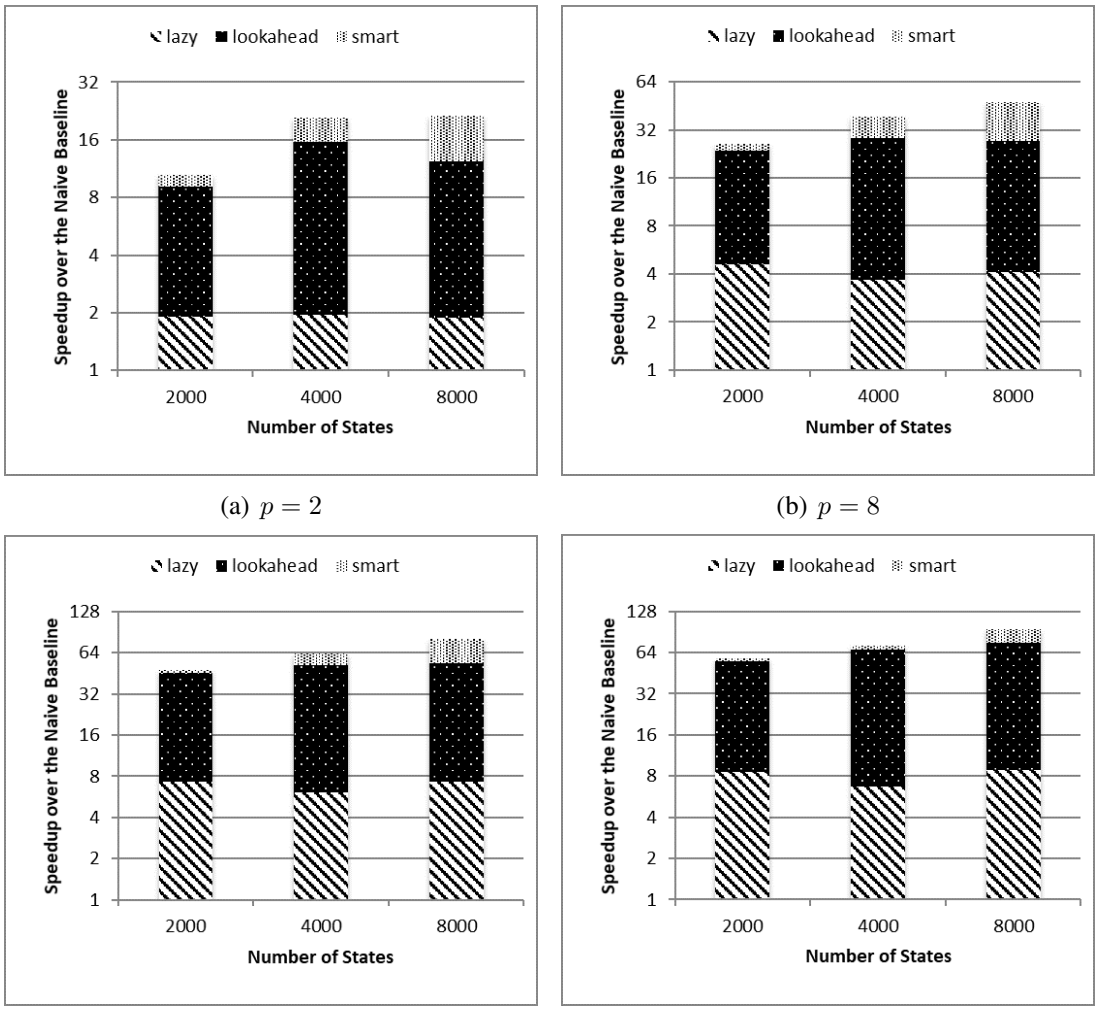
We call the implementation of GREEDY below as the naive baseline implementation. We also implemented the improvements over naive baseline as suggested in Section 5.1, Section 5.2, and Section 5.3. Below, we call these implementations as "Lazy", "Lookahead", and "Smart", respectively. Note that Lookahead is implemented on top of Lazy, and Smart is implemented on top of Lazy and Lookahead.

Figure 6.3 presents the speedups of each improvement (i.e. Lazy, Lookahead, and Smart) over the naive baseline for GREEDY. Lazy computation provides a stable improvement, which is more sensitive to alphabet size $p$. For a given alphabet size, we observe a somewhat constant speed up values. However, the average speedup values increase with the alphabet size. The lookahead also provides a stable improvement and it is more effective. For small alphabet sizes, the speedup provided by the lookahead is almost constant for different number of states. For larger alphabet sizes, on the other hand, the speedup provided by the lookahead increases with the number of states. The effect of the lookahead also increases with the alphabet size. The smart intersection check similarly improves the performance, but not to the same extent as lazy and lookahead.

Overall, when all the improvements are considered together, the average speedup values shows an increasing trend as the size of the automaton increases, and as we apply more aggressive improvements. We start with $2\times$ speedup for the automata with $n = 2000$ and $p = 2$ when only the lazy computation is enabled, and we reach to $95\times$ (almost two orders of magnitude) speedup for the automata with $n = 8000$ and $p = 128$ when all the improvements are enabled.

To validate effectiveness of our methods, we compare the time spent for BFS forest construction in each method. Note that, the time spent in PMF construction is proportional to the number of edges processed. The total number of edges to be processed for an automaton with $n$ states and $p$ letters is $pn(n-1)/2$. Table 6.4 reports the percentage of these edges processed by each method.

First of all, note that even Algorithm 1 does not process all edges. This is caused by the termination condition of the while–loop at line 5. In Algorithm 1, another possible termination is given at line 5 as a comment: "$F$ is not empty". When this alternative termination condition is used, Algorithm 1 processes 100% of the edges.




(a) $p = 2$ (b) $p = 8$

(c) $p = 32$ (d) $p = 128$

Figure 6.3: The speedup values normalized w.r.t. the naive baseline. For each additional improvement, the cumulative speedup is given with stacked columns.

Recall that the initial motivation for this work is based on the observation that PMF construction time being too high compared to the overall running time of GREEDY. We aimed at reducing this high cost, by modifying GREEDY so that they can run without full PMF construction. The figures in Table 6.4 show that we succeed. Only a very small part of the PMF is constructed in the modified version of GREEDY Algorithm. As the size of the automata increases, the percentage of the PMF constructed decreases.

| $n$ | $p$ | Algorithm 1 | Lazy | Lookahaed | Smart |
|---|---|---|---|---|---|
| 2000 | 2 | 99.99% | 40.31% | 1.71% | 1.71% |
| | 8 | 84.83% | 12.18% | 0.56% | 0.57% |
| | 32 | 37.41% | 3.64% | 0.25% | 0.25% |
| | 128 | 11.19% | 1.00% | 0.11% | 0.09% |
| 4000 | 2 | 99.99% | 36.92% | 1.42% | 1.52% |
| | 8 | 87.11% | 11.72% | 0.42% | 0.45% |
| | 32 | 40.12% | 2.93% | 0.16% | 0.17% |
| | 128 | 12.08% | 0.94% | 0.07% | 0.08% |
| 8000 | 2 | 100.00% | 37.56% | 1.31% | 1.24% |
| | 8 | 89.16% | 12.09% | 0.38% | 0.33% |
| | 32 | 42.78% | 3.08% | 0.11% | 0.06% |
| | 128 | 13.02% | 0.83% | 0.06% | 0.05% |

Table 6.4: The percentage of processed edges

# CHAPTER 7

# CONCLUSION AND FUTURE WORK

GREEDY is widely used as a performance baseline for the new heuristics in the literature. None of the newer heuristics could actually match the time performance of GREEDY and CYCLE so far. With the new improved approaches suggested in this thesis, GREEDY and CYCLE have now become more competitive than they already are.

We investigated the efficient implementation and use of modern multicore CPUs and GPUs to scale the performance of synchronizing sequence generation heuristics. We mainly focused on the PMF generation phase (which is employed by almost all the heuristics in the literature), since it is the most time consuming part of GREEDY, which is probably the same with CYCLE. Even with no parallelization, our algorithmic improvements yield 33x speedup on GREEDY for automata with 8000 states and 128 inputs. Furthermore, around 494x speedup has been obtained with GPU parallelization for the same automata class.

First phase of GREEDY is common for the well known synchronizing heuristics, such as SYNCHROP, SYNCHROPL, and FASTSYNCHRO. Since they are more expensive, their first phase does not dominate as much as it does for GREEDY and CYCLE. However, the parallelization techniques can be applied to other heuristics. One can also parallelize second phase of the slower heuristics to make them more competitive. Furthermore, we proposed techniques to speedup GREEDY without additional parallelization. With these optimizations, we obtained order(s) of magnitude faster heuristics. The techniques suggested in this thesis become more effective as the size of the automata increases. Due to the increased speeds of GREEDY, the heuristic will now scale more.

Note that other synchronizing heuristics existing in the literature such as SYNCHROP, SYNCHROPL, and FASTSYNCHRO cannot directly benefit from the lazy computation and further techniques. These heuristics require shortest merging sequence information for many pairs of states, if not for all. Due to this property, a large part of, or the entire, BFS forest will have to be constructed. Therefore, the underlying intuition for SYNCHROP, SYNCHROPL, and FASTSYNCHRO will need to be changed in order for them to take advantage of the algorithmic improvements that the thesis proposed. Modifying the underlying intuition for SYNCHROP, SYNCHROPL, and FASTSYNCHRO so that they can also benefit from the techniques suggested in this thesis can be considered as a future work.

## BIBLIOGRAPHY


[1] S. BEAMER, K. ASANOVIĆ, AND D. PATTERSON, *Direction-optimizing breadth-first search*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Los Alamitos, CA, USA, 2012, IEEE Computer Society Press, pp. 12:1–12:10.

[2] M. V. BERLINKOV, *On the probability of being synchronizable*, in Algorithms and Discrete Applied Mathematics: Second International Conference, CALDAM 2016, Thiruvananthapuram, India, February 18-20, 2016, Proceedings, S. Govindarajan and A. Maheshwari, eds., Springer International Publishing, 2016, pp. 73–84.

[3] M. BROY, B. JONSSON, J. KATOEN, M. LEUCKER, AND A. PRETSCHNER, eds., *Model-Based Testing of Reactive Systems, Advanced Lectures*, vol. 3472 of Lecture Notes in Computer Science, Springer, 2005.

[4] J. ČERNÝ, *Poznámka k homogénnym experimentom s konečnỳmi automatmi*, Matematicko-fyzikálny časopis, 14 (1964), pp. 208–216.

[5] J. ČERNÝ, A. PIRICKÁ, AND B. ROSENAUEROVÁ, *On directable automata*, Kybernetika, 7 (1971), pp. 289–298.

[6] H. CHO, S.-W. JEONG, F. SOMENZI, AND C. PIXLEY, *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, IEEE, 1993, pp. 494–498.

[7] D. EPPSTEIN, *Reset sequences for monotonic automata*, SIAM J. Comput., 19 (1990), pp. 500–510.

[8] G. JOURDAN, H. URAL, AND H. YENIGÜN, *Reduced checking sequences using unreliable reset*, Inf. Process. Lett., 115 (2015), pp. 532–535.

[9] S. KARAHODA, O. T. ERENAY, K. KAYA, U. C. TÜRKER, AND H. YENIGÜN, *Parallelizing heuristics for generating synchronizing sequences*, in IFIP International Conference on Testing Software and Systems, Springer International Publishing, 2016, pp. 106–122.

[10] S. KARAHODA, K. KAYA, AND H. YENIGÜN, *Synchronizing heuristics: Speeding up the fastest*, Expert Systems with Applications, 94 (2018), pp. 265 – 275.

[11] A. KISIELEWICZ, J. KOWALSKI, AND M. SZYKUŁA, *Computing the shortest reset words of synchronizing automata*, Journal of Combinatorial Optimization, 29 (2015), pp. 88–124.

[12] R. KUDŁACIK, A. ROMAN, AND H. WAGNER, *Effective synchronizing algorithms*, Expert Systems with Applications, 39 (2012), pp. 11746–11757.

[13] D. LEE AND M. YANNAKAKIS, *Principles and methods of testing finite state machines-a survey*, Proceedings of the IEEE, 84 (1996), pp. 1090–1123.

[14] B. K. NATARAJAN, *An algorithmic approach to the automated design of parts orienters*, in 27th Annual Symposium on Foundations of Computer Science (sfcs 1986), Oct 1986, pp. 132–142.

[15] NVIDIA, *NVIDIA CUDA C Programming Guide*, (2015).

[16] J. OLSCHEWSKI AND M. UMMELS, *The complexity of finding reset words in finite automata*, in International Symposium on Mathematical Foundations of Computer Science, Springer, 2010, pp. 568–579.

[17] J.-E. PIN, *On two combinatorial problems arising from automata theory*, North-Holland Mathematics Studies, 75 (1983), pp. 535–548.

[18] A. ROMAN, *Synchronizing finite automata with short reset words*, Applied Mathematics and Computation, 209 (2009), pp. 125–136.

[19] A. ROMAN AND M. SZYKUŁA, *Forward and backward synchronizing algorithms*, Expert Systems with Applications, 42 (2015), pp. 9512–9527.

[20] P. STARKE, *Abstract automata*, North-Holland Pub. Co., 1972.

[21] M. SZYKUŁA, *Improving the upper bound the length of the shortest reset words*, CoRR, abs/1702.05455 (2017).

[22] A. N. TRAHTMAN, *Some results of implemented algorithms of synchronization*, in 10th Journees Montoises d'Inform, 2004.

[23] U. C. TÜRKER, *Parallel algorithm for deriving reset sequences from deterministic incomplete finite automata* - submitted, IJFCS International Journal of Foundations of Computer Science.

[24] ČERNÝ JÁN, *A note on homogeneous experiments with finite automata English*, Matematicko-fyzikálny časopis, 14 (1964), pp. 208–216.

[25] M. V. VOLKOV, *Synchronizing automata and the Černý conjecture*, in International Conference on Language and Automata Theory and Applications, Springer, 2008, pp. 11–27.