An Evaluation of Massively Parallel Algorithms for DFA Minimization

Jan Martens
Leiden University
The Netherlands
j.j.m.martens@liacs.leidenuniv.nl

Anton Wijs
Eindhoven University of Technology
The Netherlands
a.j.wijs@tue.nl

We study parallel algorithms for the minimization of Deterministic Finite Automata (DFAs). In particular, we implement four different massively parallel algorithms for DFA minimization on Graphics Processing Units (GPUs). Our results confirm the expectations that the algorithm with the theoretically best time complexity is not practically suitable to run on GPUs due to the large amount of resources needed. We empirically verify that parallel partition refinement algorithms from the literature perform better in practice, even though their time complexity is worse. Lastly, we introduce a novel algorithm based on partition refinement with an extra parallel partial transitive closure step and show that on specific benchmarks it has better run-time complexity and performs better in practice.

1 Introduction

In contrast to sequential chips, the processing power of parallel devices keeps increasing. Graphics Processing Units, or GPUs, are examples of such devices. Originating from the need to do simple computations for many (independent) pixels to generate graphics, GPUs have also shown useful as computational powerhouses, and led to general-purpose computing on GPUs (GPGPU). Most convincingly, GPUs have become indispensable in training models for artificial intelligence. Because of the enormous potential of GPUs, it is important to investigate how computational problem solving can be accelerated with them.

Deterministic Finite Automata (DFAs) are one of the simplest computational formalisms. The natural problem of computing a minimal machine that is equivalent to a given machine w.r.t. the input is omnipresent in the field of theoretical computer science. In the case of DFAs the problem has a rich history. The first method that computes a minimal DFA dates back to Moore's framework [13], and is a *partition refinement* algorithm. Later, this algorithm was adapted by Hopcroft [8] to a quasi-linear time algorithm.

The complexity class known as Nick's Class (*NC*) consists of the problems that can be solved in polylogarithmic time with a parallel machine using a polynomial number of parallel processors. It is an open question whether $NC \stackrel{?}{=} P$, but it is widely believed that this is not the case. Similar to the assumption that decision problems not in *P* are inherently difficult (known as Cobham's thesis), we can think of *P*-complete problems as being inherently sequential.

The problem of minimizing DFAs is known to be in *NC* [4], which intuitively means it can be efficiently computed in parallel. In contrast, the problem of computing bisimilarity on non-deterministic structures is known to be *P*-complete [1]. Interestingly, the most efficient sequential algorithms for these two problems, i.e., Hopcroft's algorithm [8] and an algorithm based on Paige-Tarjan [14], respectively, are very similar. In particular, these algorithms are both *partition refinement* algorithms.

The parallel algorithms studied for computing bisimilarity on non-deterministic structures and DFA minimization are also partition refinement algorithms [12, 16, 18, 21]. Since DFA minimization is in

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 138–153, doi:10.4204/EPTCS.409.13

© J.J.M. Martens & A.J. Wijs
This work is licensed under the
Creative Commons Attribution License.

NC, there is a parallel sublinear time algorithm. However, none of these partition refinement algorithms studied have a sublinear run-time. A linear lower bound for the parallel run-time was proven in [6] for any parallel partition refinement algorithm deciding bisimilarity, and this result also directly applies to deterministic structures such as DFA minimization. This means that no partition refinement algorithm can achieve the theoretically optimal run-time on parallel machines. It is therefore interesting to investigate whether there is an algorithm that is not a partition refinement algorithm that performs better in parallel than partition refinement algorithms.

The algorithm introduced in [4] runs in logarithmic time. However, the work is mainly theoretical and the large amount of parallel processors and memory required makes it unlikely to scale well in practice. The main constraint here is the need to compute the transitive closure for the underlying graph of the DFA. It seems hard to find a significant improvement in the number of parallel processors needed.

In this paper we compare implementations of different parallel algorithms for DFA minimization on GPUs, using the various parallel algorithms proposed in the literature as a basis. We establish that the logarithmic runtime complexity with the construction from [4] is not feasible due to the large amount of processors needed. Additionally, we find that on our benchmarks the partition refinement algorithm that uses sorting in each iteration performs better than the naive splitting strategy on the more diverse benchmarks from the VLTS benchmark set,[1] but worse for benchmarks that are known to be hard for partition refinement algorithms. Finally, we show a method of adding a partial transitive closure as a preprocessing step that can significantly increase the performance on benchmarks with a very specific shape.

2 Preliminaries

We write $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ for the set of booleans, $\mathbb{N}$ for the set of natural numbers, and for numbers $i, j \in \mathbb{N}$ we define $[i, j] = \{c \in \mathbb{N} \mid i \leq c \leq j\} \subseteq \mathbb{N}$, the closed interval from i to j. Given an alphabet Σ, a sequence $a_1 a_2 \ldots a_n$ of symbols from Σ is called a word. We write Σ^* for the set containing all finite sequences of letters in Σ. The empty-sequence consisting of no symbols is written as ε.

Definition 1. (Deterministic Finite Automaton) A deterministic finite automaton (DFA) $A = (Q, \Sigma, \delta, F, q_0)$ is a five-tuple consisting of:

- a finite set of states Q,
- a finite alphabet Σ,
- a transition function $\delta : Q \times \Sigma \to Q$,
- a set of accepting states $F \subseteq Q$, and
- an initial state $q_0 \in Q$.

We sometimes write $q \xrightarrow{a} q'$ if $\delta(q, a) = q'$. The function $\delta^* : Q \times \Sigma^* \to Q$ extends the transition function to words and is defined inductively for all words in Σ^* as follows:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

Given a DFA $A = (Q, \Sigma, \delta, F, q_0)$, a word $w \in \Sigma^*$ is *accepted* iff $\delta^*(q_0, w) \in F$. The language of a DFA A, notation $\mathscr{L}(A)$, is the set of all words $w \in \Sigma^*$ that are accepted by A.

[1] https://cadp.inria.fr/resources/vlts (visited on: 19-04-2024).

We consider the problem of computing the minimal DFA, i.e., given a DFA A, identifying the DFA A' with the smallest number of states such that $\mathscr{L}(A') = \mathscr{L}(A)$.

Minimizing DFAs consists of combining undistinguishable states and deleting unreachable states. The main part of the problem consists of combining states, and removing unreachable states can be seen as a simple pre-processsing step. For the remainder of the paper, we assume that all the states in a DFA are reachable from its initial state. The algorithms can be seen as computing bisimilarity, or the coarsest set partition problem, without the preprocessing step that removes unreachable states.

Representation. For an input automaton $A = (Q, \Sigma, \delta, F, q_0)$, we assume that the states in Q and letters in Σ are represented by unique indices, i.e., $Q = \{0, \ldots, |Q|\}$ and $\Sigma = \{0, \ldots |\Sigma|\}$. The transition function δ is represented by $|\Sigma|$ arrays of length $|Q|$, such that for state $q \in Q$ and letter $a \in \Sigma$, $\delta[a][q] = \delta(q, a)$.

The PRAM Model. The complexities we mention assume the model of the *Parallel Random Access Machine (PRAM)*. The PRAM is a natural extension of the RAM model, where parallel processors have access to a shared memory. A PRAM consists of a sequence of processors $P_0, P_1, \ldots$ and a function $\mathscr{P}$ that given the size of the input defines a bound on the number of processors used.

Each processor P_i has the natural instructions of a normal RAM and in addition has an instruction to retrieve its unique index i. All processors run the same program in lock-step, using their index to identify the data they need to access. This parallel processing is called single-instruction multiple data (SIMD).

There are many different ways to handle data-races. We assume the concurrent read, concurrent write (CRCW) model following [17], where processors are allowed to read from and write to the same memory location concurrently. After multiple concurrent writes to the same memory location, that location contains the result of one of those writes.

GPUs. While in reality, no device completely adheres to the PRAM model, recent hardware advancements has led to devices that are getting better and better at approximating this model. The GPU, in particular, is a very suitable target platform for PRAM algorithms, as it has been specifically designed for SIMD processing. The performance of GPU programs typically relies on launching tens to hundreds of thousands of threads, as the performance of these programs is often memory-bound: accessing the input data in the GPU's *global* memory, in NVIDIA CUDA terminology, is relatively slow. This latency can be hidden by a GPU via fast context switching between threads. As one thread is waiting for data to be retrieved, another thread is executed in the meantime on the same processor. It is this fast context switching between threads that allows GPUs, typically equipped with several thousands of cores, to virtually execute hundreds of thousands of threads concurrently. In the current work, we employ NVIDIA GPUs, programs for which can be written in CUDA C++.

3 The algorithms

3.1 Transitive closure

The first algorithm we discuss has theoretical polylogarithmic runtime [4]. However, the large amount of memory and parallel processors it uses makes it unlikely to work in practice. Here we confirm this fact.

The idea is rather simple; build a graph with nodes $V = Q \times Q$ and edges E containing $(q, q') \rightarrow (p, p')$ iff there is a letter $a \in \Sigma$ such that $\delta(q, a) = p$ and $\delta(q', a) = p'$. Initially, in the array `Apart` we label the nodes $(q, q') \in V$ to be inequivalent if $q_1 \in F \iff q_2 \notin F$. Any two states $q_1, q_2 \in Q$ are not equivalent iff they were initially labelled in `Apart` or there is a path to a labelled node. Computing this reachability of false nodes can be seen as computing the transitive closure in the directed graph (V, E) containing n^2

Algorithm 1 Transitive DFA minimization `trans`.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$ where $|Q| = n$
Output: The minimal quotient automaton represented in the matrix *Apart*

```
1:  V :: Q × Q                                      ▷ Nodes of graph consisting of pair of states
2:  Apart :: Array[n^2] of type 𝔹
3:  Reach :: Array[n^2][n^2] of type 𝔹             ▷ Represents reachability in V, initially false
4:  do in parallel for (q,q') ∈ V                    ▷ Initializes data structures in parallel.
5:      Apart[(q,q')] := (q ∈ F ⟺ q' ∉ F)            ▷ State initially unequal
6:      for all a ∈ Σ do
7:          Reach[(q,q')][(δ(q,a), δ(q',a))] := true
8:  stable := false
9:  while ¬stable do
10:     stable := true
11:     do in parallel for s,t,u ∈ V
12:         if Reach[s][t] and Reach[t][u] and ¬Reach[s][u] then
13:             Reach[s][u] := true
14:     do in parallel for s,t ∈ V
15:         if Reach[s][t] and Apart[t] and ¬Apart[s] then
16:             Apart[s] := true
17:             stable := false
```

nodes. In parallel this computation can be done in polylogarithmic running time using $O(|V|^3)$ parallel processors [9, Chapter 5.5.].

Algorithm 1, which we refer to as `trans`, implements this idea. First, at lines 4–7 (l.4–7), the graph is constructed, inequivalent nodes labelled in the *Apart* data structure and the edges stored in the adjacency matrix *Reach*. Next, the parallel transitive closure of *Reach* is computed and *Apart* updated accordingly. If in an iteration there is no new pair of states labelled *Apart* the algorithm is finished. The minimal automaton is represented in the graph where states $q, q' \in Q$ can be combined if $\neg Apart(q, q')$.

Computing the transitive closure for a directed graph in logarithmic time requires many processors. Our naive implementation requires $|V|^3$ parallel processors. Given a DFA with n states, this means that since $|V| = n^2$, we require n^6 processors. Theoretically, more efficient methods are known for computing the transitive closure, which uses matrix multiplication. Matrix multiplication can be computed with $O(n^\omega)$ operations, where currently $\omega \leq 2.372\ldots$, this means we can compute our transitive closure with $O(|V|^\omega)$ processors. Since these algorithms are non-trivial and already $|V| = n^2$, we believe these improvements would not significantly change the results mentioned here.

3.2 Naive partition refinement

The next algorithm, `naivePR`, is an adaptation of the parallel algorithm for bisimilarity checking of Kripke structures from [12]. The program runs on a PRAM with $\max(n, m)$ processes, where n is the number of states and $m = |\Sigma| * n$ is the number of transitions in the input DFA.

The algorithm applies *partition refinement*: states are initially partitioned into *blocks*, and the algorithm repeatedly splits blocks into smaller blocks until a fix-point is reached. Once this has happened, each block represents one state of the minimized DFA.

When splitting blocks in parallel, one particular challenge is how to identify newly created blocks,

as each new block requires a unique identifier. Algorithm 2 does this by means of a leader election procedure: for each block, one of its states is elected leader, meaning that it is used as an identifier to refer to the block. In this way each iteration of the algorithm takes constant time if performed on a parallel machine that has concurrent writes.

In Algorithm 2, at l.1, an array *block* is initialized that defines for every state in Q its current block (as represented by a leader in Q). An array *new_leader* is defined at l.2 that is used to elect new leaders. At l.3, the initial leaders are selected: one state $q_f \in F$ for the block consisting of all the accepting states $q \in F$, and one state $q_n \in Q \setminus F$ for all the non-accepting states $q' \in Q \setminus F$. The array *block* is subsequently initialised using these leaders (l.4–5).

Next, partition refinement is applied inside the **while**-loop at l.7. The variable *stable* is used to monitor whether a fix-point has been reached, which has happened as soon as no blocks can be split any further. At the start of each iteration through the **while**-loop, *stable* is set to `true` (l.8). Next, all transitions of the DFA are processed in parallel (l.9), and for each transition $q \xrightarrow{a} q'$, it is checked whether $block[q']$ differs from the block that the leader $block[q]$ can reach via an a-transition. If it does, then q should be separated from its leader. At l.11, q is assigned to $new_leader[block[q]]$, the latter being the position where the leader for the new block will be elected. Here, the result of concurrent writes, as allowed by the PRAM CRCW model, is used for leader election.

Subsequently, when l.12 is reached, *new_leader* contains the newly elected leaders: specifically, at $new_leader[block[q]]$, the leader for the new block created by splitting off states from $block[q]$ is stored. In the parallel loop of l.12, the transitions are once more processed in parallel, and whenever a state turns out to differ from its leader regarding block reachability (l.13), the leader of that state is updated (l.14). Finally, since a block has been split, *stable* is set to `false` at l.15.

The largest difference between Algorithm 2 and the original algorithm [12] is that Algorithm 2 splits blocks directly w.r.t. the leader, as opposed to first selecting one particular block as *splitter*, and splitting those blocks in which some states differ w.r.t. their leader concerning the ability to reach the splitter. The reason for this difference is that for DFAs, comparing the outgoing transitions of two states is much more straightforward, as each state has exactly one outgoing transition for every $a \in \Sigma$. In the setting of LTSs, due to non-determinism it is not possible to directly compare the behaviour of a state with the leader state, and hence a fixed splitter is chosen before.

In Algorithm 2, leader election is performed in two phases: in the first phase (l.9–11), states are written to *new_leader* to elect new leaders, and the results are subsequently read at l.12–15. One could argue that this is inefficient, and that it would perhaps be better to combine these two phases. This is possible, but it requires the use of atomic *compare-and-swap* (CAS) operations. This is illustrated in Algorithm 3. In the single loop at l.11–17, new leaders are written to *and* read from *new_leader*. At l.14–15, the use of a compare-and-swap operation is described: in one atomic operation, the current value stored at $new_leader[leader]$ is stored in *new_block*, and it is checked whether $new_leader[leader]$ is equal to the initial value $\perp$, and if it is, it is set to q. Next, at l.16, if *new_block* is equal to $\perp$, it means q has been elected as leader. Otherwise, *new_block* indicates which state is the new leader. Note that for this to work, after execution of the loop at l.11, the values of *new_leader* have to be reset to $\perp$.

In practice, we experienced that a GPU implementation (in CUDA 12.2) of Algorithm 3 exhibits similar runtimes compared to a GPU implementation of Algorithm 2. The benefit of merging the loops seems to be negated by the use of atomic operations. For this reason, when discussing the experiments in Section 4.2, we do not involve Algorithm 3.

Algorithm 2 Parallel leader-election-based algorithm `naivePR`.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$ where $|Q| = n$
Output: The minimal quotient automaton represented in the array *block*

1: *block* :: Array$[n]$ of type Q
2: *new_leader* :: Array$[n]$ of type Q
3: Select initial leader states $q_f \in F$ and $q_n \in Q \setminus F$
4: **do in parallel for** $q \in Q$
5: $\quad$ $block[q] := (q \in F\ ?\ q_f : q_n)$ ▷ Initialize
6: $stable :=$ `false`
7: **while** $\neg stable$ **do**
8: $\quad$ $stable :=$ `true`
9: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
10: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
11: $\quad\quad\quad$ $new_leader[block[q]] := q$ ▷ Leader election
12: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
13: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
14: $\quad\quad\quad$ $block[q] := new_leader[block[q]]$ ▷ Split from leader
15: $\quad\quad\quad$ $stable :=$ `false`

Algorithm 3 Parallel leader-election based `naivePR` with atomics.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$, where $|Q| = n$
Output: The minimal quotient automaton represented in the array *block*

1: *block* :: Array$[n]$ of type Q
2: *new_block* :: Q
3: *leader* :: Q
4: *new_leader* :: Array$[n]$ of type Q
5: Select initial leader states $q_f \in F$ and $q_n \in Q \setminus F$
6: **do in parallel for** $q \in Q$
7: $\quad$ $new_leader[q] := \perp$ ▷ Initialize
8: $\quad$ $block[q] := (q \in F\ ?\ q_f : q_n)$
9: **while** $\neg stable$ **do**
10: $\quad$ $stable :=$ `true`
11: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
12: $\quad\quad$ $leader := block[q]$
13: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(leader,a)]$ **then**
14: $\quad\quad\quad$ $\{new_block := new_leader[leader];$ ▷ Leader election with CAS
15: $\quad\quad\quad$ $new_leader[leader] = \perp\ ?\ new_leader[leader] := q\}$
16: $\quad\quad\quad$ $block[q] := new_block = \perp\ ?\ q : new_block$ ▷ Split from leader
17: $\quad\quad\quad$ $stable :=$ `false`
18: $\quad$ **do in parallel for** $q \in Q$
19: $\quad\quad$ $new_leader[q] := \perp$

3.3 Sorting arrays

The next algorithm is an algorithm inspired by [16, 18]. Similar to Algorithm 2, this algorithm also performs partition refinement, but instead of doing so using leader elections, it repeatedly computes a *signature* for every state, and sorts the states w.r.t. their signatures. This method allows splitting a block in more than two subblocks with the downside that each iteration takes more than constant parallel time.

The algorithm from [18] uses hashing to construct and compare signatures. Since sorting arrays is a very native operation on GPUs, we follow [16] and use a sorting approach to construct the new blocks.

Algorithm 4 presents this approach as `sortPR`. Again, an array *block* is created (l.1). In addition, an array *state* is used for sorting the states (l.2). The signature of a state consists of a list of block IDs, one for each $a \in \Sigma$: $signature[q][a]$ is equal to q' iff $q \xrightarrow{a} q''$ and $block[q''] = q'$.

The array *new_block* is used to store the results of assigning new blocks to states (l.4). Finally, the current number of blocks is stored at l.5 in *num_blocks*.

Next, at l.6–8, *block* and *state* are initialised. The block consisting of all accepting states is given ID 0, while the other states are assigned to block 1 (l.7). All the states are added to *state* at l.8.

In the loop at l.9–19, the partition refinement is performed until a fix-point has been reached, i.e., the number of blocks has not increased (l.19). In each iteration of this loop, the following is performed. First, in parallel, the signatures are updated (l.11–12). After that, *state* is sorted in parallel, using *signature* to compare states. The comparison function is given at l.20–26. First, states are compared based on the block they reside in. If they reside in the same block, then the blocks they can reach via outgoing transitions are compared. Note that the for loop starting in (l.23) is sequential and requires iterating over the alphabet letters in a fixed order.

Once *state* has been sorted, the parallel *adjacent difference* is computed and stored in *new_block*. The result of this is that $new_block[0] = state[0]$ and for all $0 < i < n$, $new_block[i] = are_neq(state[i], state[i-1])$, with *are_neq* as defined at l.27–31. Once $new_block[0]$ has been reset to 0 (l.15), *new_block* contains only 0's and 1's, with each 1 identifying the start of a new block. At l.16, an *inclusive scan* is performed in parallel, resulting in *new_block* having been updated in such a way that for each $0 \leq i < n$, $new_block[i] = \sum_{0 \leq j \leq i} new_block'[j]$, with new_block' referring to *new_block* at the start of executing l.16.

Now, for all $0 \leq i < n$, $new_block[i]$ contains the new block of state $state[i]$. At l.17–18, *block* is updated in parallel to reflect this. As the largest new block ID can be found at $new_block[n-1]$, this location can be used to determine the new number of blocks at l.19.

In [16] it is shown that on average this algorithm has polylogarithmic run-time complexity. The argument given uses the fact that on uniformly sampled DFAs almost all pairs of states have a shortest distinguishing word of polylogarithmic depth. This fact is attributed to [19]. Although this is true for uniformly sampled DFAs, we like to stress that for many use cases and real-life applications this bound does not apply. For example, in the Fibonacci automata presented in Section 4.2 this is not the case. In the automaton $\texttt{Fib}_i$ containing n states, there is a pair of states for which the shortest distinguishing word, and thus also the number of iterations, has length $n-2$.

3.4 Partition refinement using partial transitive closure

In this section, we present a new algorithm `transPR`. The main idea of the algorithm is to perform partition refinement like the algorithms before, but in the initialization compute the transitive closure on each distinct letter. After this initialization step, we use `naivePR` to complete the minimization.

This approach is presented in Algorithm 5. This is done in a data-parallel way which is also used for prefix sum and finding the end of a linked list [7]. On some DFAs, with a rather specific structure, this

Algorithm 4 Parallel sorting-based algorithm `sortPR`

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$, where $|Q| = n$ and $|\Sigma| = k$

Output: The minimal quotient automaton represented in the array *block*

```
 1: block :: Array[n] of type ℕ
 2: state :: Array[n] of type Q
 3: signature :: Array[n][k] of type Q
 4: new_block :: Array[n] of type ℕ
 5: num_blocks := 2
 6: do in parallel for q ∈ Q
 7:     block[q] := (q ∈ F ? 0 : 1)                                ▷ Initialize
 8:     state[q] := q
 9: repeat
10:     num_blocks := new_block[n − 1] + 1            ▷ Number of blocks before iteration
11:     do in parallel for q, a ∈ Q × Σ
12:         signature[q][a] := block[δ(q, a)]
13:     sort(state, COMPARE)
14:     new_block := adjacent_diff(state, ARE_NEQ)           ▷ Place 1 for each change
15:     new_block[0] := 0
16:     new_block := inclusive_scan(new_block)              ▷ Compute new block labels
17:     do in parallel for q ∈ Q
18:         block[state[q]] = new_block[q]
19: until new_block[n − 1] + 1 = num_blocks

20: function COMPARE(q1, q2)
21:     if block[q1] > block[q2] then return false
22:     if block[q1] < block[q2] then return true
23:     for all a ∈ Σ do
24:         if signature[q1][a] > signature[q2][a] then return false
25:         if signature[q1][a] < signature[q2][a] then return true
26:     return false

27: function ARE_NEQ(q1, q2)
28:     if block[q1] ≠ block[q2] then return true
29:     for all a ∈ Σ do
30:         if signature[q1][a] ≠ signature[q2][a] then return true
31:     return false
```

method exponentially improves the runtime compared to the other partition refinement algorithms.

Algorithm 5 Parallel partition refinement with transitive closure `transPR`

1: $\Sigma^T := \{a^{2^i} \mid a \in \Sigma, i \in [0, \lfloor \log n \rfloor]\}$
2: $\delta^T :: Q \times \Sigma^T \mapsto Q$
3: $\delta^T(q,a) := \delta(q,a)$ for all $a \in \Sigma$
4: **for all** $i \in [1, \lfloor \log n \rfloor]$ **do**
5:     **for all** $a \in \Sigma$ **do**
6:         **do in parallel for** $q \in Q$
7:             $\delta^T(q, a^{2^i}) := \delta^T(\delta^T(q, a^{2^{i-1}}), a^{2^{i-1}})$
8: Perform `naivePR` on the DFA $A' = (Q, \Sigma^T, \delta^T, F, q_0)$

The algorithm works by adding letters for increasingly large words of the same letters. Given an input DFA $A = (Q, \Sigma, \delta, F, q_0)$, we construct a DFA $A' = (Q, \Sigma^T, \delta^T, F, q_0)$ which has the same set of states and final states, but a larger alphabet Σ^T. The alphabet contains the letters $a^{2^0}, a^{2^1}, \ldots, a^{2^{\lfloor \log n \rfloor}}$ for each original letter $a \in \Sigma$. The transition function is computed such that for each new symbol $a^k \in \Sigma^T$ the transition function $\delta^T(q_1, a^k) = q_k$ if in the original DFA Q there are states $q_1, \ldots q_k \in Q$ such that $\delta(q_i, a) = q_{i+1}$ for each $i \in [1,k]$. This can be computed in a logarithmic number of parallel steps, by using the previously computed transitions, as is done at l.7 of Algorithm 5.

The correctness of this algorithm relies on the fact that equality on states is invariant under the partial closure that is added. Indeed, we can see that the DFA A' obtained in Algorithm 5 is language equivalent to the input DFA A if we consider the alphabet letters added as words. If $\delta^T(q, a_T) = q'$ for some $a_T \in \Sigma^T$, then $a_T = a^{2^j}$ for some $a \in \Sigma$ and $j \in [0, \lfloor \log n \rfloor]$. By construction there is a sequence $q_0, \ldots q_k$ such that $q_0 = q$, $q_{i+1} = \delta(q_i, a)$ and $q_k = q'$.

This approach helps in the case of long paths with the same letter. Consider the DFA A from Figure 1. This DFA accepts all words a^j with $j > 8$. Any parallel partition refinement algorithm would need at least 8 iterations to conclude that q_0 is not the same as q_1. However, building the partial transitive closure only requires a logarithmic number of parallel iterations. With this partial transitive closure added, a partition refinement algorithm can in the first iteration conclude that q_0 is different from q_1 since the transition with a^8 leads to different states.

4 Experiments

In this section, we discuss the results of our implementations. We benchmarked the implementations with respect to three families of DFAs: Fibonacci DFAs from [3], bit-splitters $\mathscr{B}_k$ derived from [6], and DFAs derived from a subset of the VLTS benchmark set.

4.1 Benchmarks

Fibonacci DFAs: The first family of DFAs we use for benchmarking consists of so-called Fibonacci automata. These are simple automata with only a unary alphabet. However, they exhibit very particular behaviour. As witnessed in [3], these automata are notoriously hard for partition refinement and the number of iterations of any partition refinement algorithm is n. The automata are called Fibonacci automata due to the close correspondence with Fibonacci words over the binary alphabet, which are

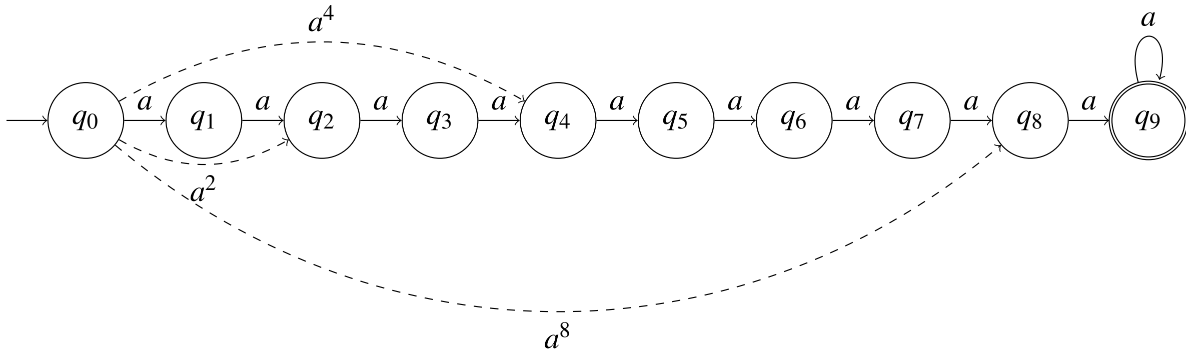

Figure 1: The DFA $A = (\{q_0, \ldots, q_9\}, \{a\}, \delta, \{q_9\}, q_0)$ with the extra partial transitive closure from q_0 added in dashed arrows.

defined inductively as follows: the base cases are $w_0 = 1$, and $w_1 = 0$, and for every $i \in \mathbb{N}$, $w_{n+1} = w_n w_{n-1}$. This gives the following sequence:

$$
\begin{aligned}
w_2 &= 01 \\
w_3 &= 010 \\
w_4 &= 01001 \\
w_5 &= 01001010 \\
&\ldots
\end{aligned}
$$

For every $n \in \mathbb{N}$, we define the automaton $\texttt{Fib}_n = (Q, \{a\}, \delta, q_0, F)$ as follows, with $w_n[i]$ referring to the i-th bit in the bit sequence w_n:

- the set of states is $Q = \{q_i \mid i \in [0, |w_n|]\}$;
- the transition function is $\delta(q_i, a) = q_{i+1 \mod |w_n|}$;
- the set of final states is $F = \{q_i \mid q_i \in Q \text{ and } w_n[i] = 1\}$.

Bit-splitters: The second family of automata consists of the so-called *bit-splitters* $\mathscr{B}_n$. For $n \in \mathbb{N}$, the bit-splitter $\mathscr{B}_n$ is a deterministic automaton with 2^n states and an alphabet consisting of $n-1$ symbols. By construction, during partition refinement, every time a block can be split, it is split in two blocks of equal size. This property makes the family inherently hard for partition refinement algorithms. However, the parallel algorithm requires only a logarithmic number of iterations to compute the minimal DFA. The bit splitter $\mathscr{B}_3$ is given in Figure 2.

The family does not contain an initial state, and comes from the setting of Labelled Transition Systems (LTSs). An LTS is a graph structure with a finite number of states and transitions between states, with each transition having an action label.

Since it is a hard example for partition refinement, we use it for this purpose and allow the absence of an initial state. In the following, states σ represent bit sequences of length n, i.e., $\sigma \in \{0,1\}^n$. We define $\mathscr{B}_1 = (Q_1, \Sigma_1, \delta_1, F_1)$, where $Q_1 = \{\texttt{0}, \texttt{1}\}$, $\Sigma_1 = \emptyset$ and $F_1 = \{\texttt{1}\}$. Given the automaton $\mathscr{B}_n = (Q_n, \Sigma_n, \delta_n, F_n)$ for some $n \in \mathbb{N}$, we define $\mathscr{B}_{n+1} = (Q_{n+1}, \Sigma_{n+1}, \delta_{n+1}, F_{n+1})$, such that:

- The set of states contains two copies of Q_n, i.e., $Q_{n+1} = \{\texttt{0}\sigma, \texttt{1}\sigma \mid \sigma \in Q_n\}$,

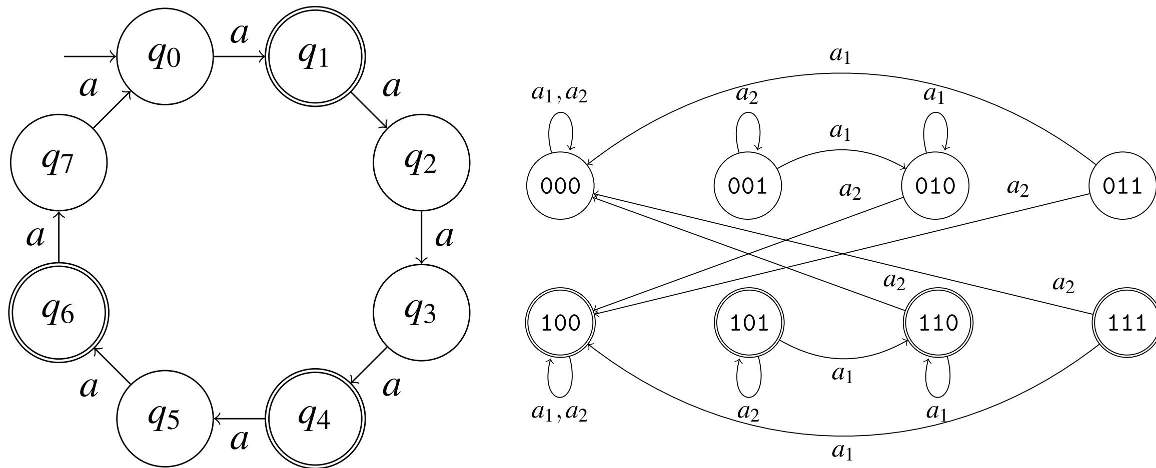

Figure 2: The DFA $\texttt{Fib}_5$ on the left, and the DFA $\mathscr{B}_3$ on the right.

- One fresh symbol $a_n \notin \Sigma_n$ is added to the alphabet: $\Sigma_{n+1} = \Sigma_n \cup \{a_n\}$,
- The transition function δ_{n+1} is defined such that for each $a_m \in \Sigma_n$, and state $\texttt{b}\sigma \in Q_{n+1}$, it maintains the behaviour of $\mathscr{B}_n$, i.e., $\delta_{n+1}(a_m, \texttt{b}\sigma) = \texttt{b}\delta_n(a_m, \sigma)$. For the fresh symbol $a_n \in \Sigma_{n+1} \setminus \Sigma_n$, δ_{n+1} is extended as follows, with $\bar{\texttt{b}}$ being the bit flipped version of $\texttt{b}$, i.e., $\texttt{b} = 0 \iff \bar{\texttt{b}} = 1$:

$$\delta_{n+1}(\texttt{b}\sigma, a_n) = \begin{cases} \bar{\texttt{b}}\texttt{0}^n & \text{If } \sigma[0] = 1, \\ \texttt{b}\sigma & \text{otherwise.} \end{cases}$$

- the set of accepting states is $F_{n+1} = \{\texttt{1}\sigma \mid \sigma \in Q_n\}$.

As previously mentioned, bit-splitter DFAs are constructed in such a way that they are inherently hard to minimize by partition refinement. Each bit-splitter $\mathscr{B}_{n+1}$ combines two copies of the bit-splitter $\mathscr{B}_n$, with the transition function defined in such a way that each possible split divides an existing block in two blocks of the same size. This results in a DFA in which the amount of required work for splitting is large, since each split involves moving many states to the new block. However, because in each split a block is split in two parts of the same size, the number of sequential splits needed is smaller than for the Fibonacci automata.

VLTSs: Lastly, we benchmark our implementations against the VLTS benchmark suite.[2] The VLTS acronym stands for *Very Large Transition Systems*. This suite consists of LTSs that originate from modelling protocols and concurrent systems. Some of the benchmarks are from case studies from industrial systems.

The transition relation of an LTS does not need to be deterministic, nor complete. We turn an LTS into a DFA by first making the LTS deterministic such that each state has at most one outgoing transition for every label, using the powerset construction algorithm [15]. To convert the deterministic LTS to a DFA, we need to complete the transition function and define which states are accepting. We define all the states as accepting and add one new non-accepting state $\perp$. For each state q and label a for which there exists no transition with that label from q, we add a new transition labelled a to $\perp$, i.e. $\delta(q, a) = \perp$.

[2] `https://cadp.inria.fr/resources/vlts` (visited on: 04-2024).

Name	N	Iterations	Time (ms)	Memory(Mb)	#threads
$\texttt{Fib}_4$	8	3	0.3	0	589,824
$\texttt{Fib}_5$	13	4	0.7	0	6,230,016
$\texttt{Fib}_6$	21	5	7.8	0	88,510,464
$\texttt{Fib}_7$	34	5	159.9	0	1,620,545,536
$\texttt{Fib}_8$	55	6	3,034.9	10	27,955,840,000
$\texttt{Fib}_9$	89	7	66,846.7	60	498,865,340,416
$\texttt{Fib}_{10}$	144	t/o	t/o	412	8,943,640,510,464

Table 1: Results of running the algorithm `trans` on the Fibonacci automata.

This completes the transition function and creates a DFA accepting all the words corresponding with a path through the original LTS.

Due to state-space explosion we were not able to make all VLTS benchmarks deterministic. We used all benchmarks for which the computation to make them deterministic took less than ten minutes.

4.2 Results

The algorithms were implemented in CUDA C++ and compiled using the CUDA toolkit 12.2, with the implementation of `sortPR` using the Thrust library for sorting and computing the adjacent differences and inclusive scans [2]. Experiments were conducted on a device running Linux Mint 20, equipped with an NVIDIA TITAN RTX GPU with 24 GB of memory and 4,608 cores. Such a GPU can manage trillions of light-weight threads. Thanks to fast context switching between threads, a GPU can typically handle a few hundred thousand threads as if they execute in parallel.

The reported times are the average of five separate runs. Benchmarks that did not finish within five minutes were aborted, in which case we registered a timeout 't/o'. Benchmarks for which there was not enough memory are indicated by 'OoM'.

Transitive approach: The results of running Algorithm 1 on the Fibonacci automata are given in Table 1. As expected, the number of threads used to compute the transitive closure in parallel grows very quickly. Although the number of iterations of the algorithm is indeed logarithmic, the available parallelism is not sufficient to lead to logarithmic run times. We only use this set of small Fibonacci automata for this algorithm. It already suggests that for a relatively small amount of states (~ 100), obtaining the required resources is already infeasible. The other benchmarks are almost completely out of range of the algorithm.

Partition refinement algorithms: The results of running the parallel partition refinement algorithms, `naivePR` (Algorithm 2), `sortPR` (Algorithm 4), and `transPR` (Algorithm 5) are given in Table 2 and Table 3 for the different benchmarks.

First, we observe in Table 2 that on the Fibonacci automata the `naivePR` performs better than `sortPR`. This can be explained by the fact that the number of iterations is n for both algorithms, while each iteration in `sortPR` is slower than in `naivePR`. Another interesting observation here is that for all benchmarks $\texttt{Fib}_{18},\ldots,\texttt{Fib}_{28}$ the run time of the algorithm `naivePR` scales linearly with the number of states n. Since the number of parallel iterations is $n-2$ for all these benchmarks, each parallel iteration processing up to $\sim 500k$ states took a similar amount of time. In other words, the GPU was able to run around $500k$ threads as if they ran in parallel. This confirms the statement about fast context switching at the beginning of Section 4.2.

	Benchmark metrics			Times (ms)			Iterations		
Name	N	$\|\Sigma\|$	Size output	`naivePR`	`sortPR`	`transPR`	`naivePR`	`sortPR`	`transPR`
$\texttt{Fib}_{20}$	17,711	1	17,711	308.8	3,909.2	1.7	17,710	17,710	14
$\texttt{Fib}_{21}$	28,657	1	28,657	494.2	6,374.2	2.4	28,656	28,656	25
$\texttt{Fib}_{22}$	46,368	1	46,368	778.7	11,712.1	4.1	46,367	46,367	61
$\texttt{Fib}_{23}$	75,025	1	75,025	1,241.3	21,366.6	8.0	75,024	75,024	101
$\texttt{Fib}_{24}$	121,393	1	121,393	2,006.7	34,793.1	12.5	121,392	121,392	104
$\texttt{Fib}_{25}$	196,418	1	196,418	3,251.3	64,411.7	18.3	196,417	196,417	138
$\texttt{Fib}_{26}$	317,811	1	317,811	5,277.8	178,367.4	49.8	317,810	317,810	102
$\texttt{Fib}_{27}$	514,229	1	514,229	8,607.7	t/o	96.1	514,228	t/o	268
$\texttt{Fib}_{28}$	832,040	1	832,040	22,723.0	t/o	178.4	832,039	t/o	299
$\texttt{Fib}_{29}$	1,346,269	1	1,346,269	59,510.8	t/o	726.9	1,346,268	t/o	755
$\texttt{Fib}_{30}$	2,178,309	1	2,178,309	141,601.0	t/o	1,109.3	2,178,308	t/o	914
$\mathscr{B}_{15}$	32,768	14	32,768	0.8	25.8	1.7	14	14	2
$\mathscr{B}_{16}$	65,536	15	65,536	1.4	29.7	3.7	15	15	2
$\mathscr{B}_{17}$	131,072	16	131,072	2.6	54.3	9.4	16	16	2
$\mathscr{B}_{18}$	262,144	17	262,144	5.0	107.2	25.6	17	17	2
$\mathscr{B}_{19}$	524,288	18	524,288	9.6	235.7	60.9	18	18	2
$\mathscr{B}_{20}$	1,048,576	19	1,048,576	19.3	520.2	139.8	19	19	2
$\mathscr{B}_{21}$	2,097,152	20	2,097,152	39.8	1,148.6	312.2	20	20	2
$\mathscr{B}_{22}$	4,194,304	21	4,194,304	82.6	2,538.5	728.7	21	21	2
$\mathscr{B}_{23}$	8,388,608	22	8,388,608	170.3	5,612.7	1,612.1	22	22	2
$\mathscr{B}_{24}$	16,777,216	23	16,777,216	352.6	12,351.8	OoM	23	23	OoM
$\mathscr{B}_{25}$	33,554,432	24	33,554,432	737.4	27,092.2	OoM	24	24	OoM
$\mathscr{B}_{26}$	67,108,864	25	67,108,864	1,541.5	59,203.8	OoM	25	25	OoM

Table 2: Results of running the partition refinement algorithms on the `Fib` and $\mathscr{B}$ benchmark set.

Finally, for the Fibonacci automata, we see that `transPR` performs significantly better on this benchmark. This can be explained by the fact that the partial transitive closure reduces the number of iterations of the algorithm significantly.

The results on the bit-splitter automata in Table 2 show that the improvement of `transPR` does not work on all automata. The high number of alphabet letters together with the structure of the automata make the transitive closure less effective, making `naivePR` much faster.

For the VLTS benchmark set we see the power of `sortPR` in Table 3. In some benchmarks, like 'vasy_69_520' the algorithm performs significantly better. In these examples, it helps that in `sortPR`, in each iteration a block can be split into many subblocks, which is not the case in the other algorithms.

Since the VLTS benchmarks originate from communication protocols and concurrent systems, the success of `sortPR` suggests that for DFAs that represent 'real' systems, this algorithm is a solid choice for efficient DFA minimization. However, the experiments with the Fibonacci and bit-splitter families of DFAs demonstrate room for improvement.

5 Conclusions & Future work

We implemented and compared different parallel algorithms for DFA minimization on GPUs. We find that the *NC* algorithm `trans` with parallel logarithmic run-time does not scale well because of the large number of resources needed. Instead, we find that the partition refinement algorithms perform better.

	Benchmark metrics			**Times (ms)**			**Iterations**		
Name	N	$\|\Sigma\|$	Size output	`naivePR`	`sortPR`	`transPR`	`naivePR`	`sortPR`	`transPR`
cwi_1_2	4,448	26	2,416	5.4	66.7	25.1	308	38	621
cwi_2416_17605	503	15	58	0.8	38.2	0.4	40	40	8
cwi_3_14	63	2	63	1.2	9.1	0.4	61	61	8
vasy_0_1	92	2	10	0.2	3.9	0.4	6	5	5
vasy_1_4	6,087	6	29	0.4	8.5	0.9	15	7	20
vasy_10_56	10,850	12	2113	8.7	40.2	30.9	519	33	791
vasy_1112_5290	1,112,491	23	266	135.4	386.8	2,049.2	246	4	231
vasy_157_297	157,605	235	4,290	455.1	1,736.3	11,312.0	1,049	27	1,306
vasy_164_1619	109,911	37	1,025	69.9	50.5	823.4	770	4	766
vasy_166_651	393,147	211	392,175	159,265.6	1,070.6	t/o	175,764	19	t/o
vasy_18_73	419,664	17	31,952	1,586.1	305.2	34,055.2	13,343	27	18,444
vasy_25_25	25,218	25,216	25,218	262,878.6	3,502.7	t/o	25,217	2	t/o
vasy_386_1171	355,790	73	114	36.9	489.4	766.0	58	8	113
vasy_40_60	40,007	3	40,007	331.6	8,391.5	845.2	20,004	20002	20,004
vasy_5_9	5,088	31	138	2.2	14.3	7.0	113	5	124
vasy_574_13561	574,058	141	3,578	2,332.2	976.5	64,312.6	2,351	5	2,634
vasy_6120_11031	3,190,785	125	5,216	13,186.6	21,886.0	t/o	2,373	21	t/o
vasy_65_2621	65,538	72	65,537	2,591.8	38.3	47,568.0	36,575	4	38,999
vasy_66_1302	209,791	81	208,419	42,864.9	96.0	t/o	179,861	8	t/o
vasy_69_520	74,958	135	74,958	7,223.0	124.2	181,611.4	49,723	12	74,667
vasy_720_390	87,741	49	3,279	176.0	57.1	2,961.7	2,936	5	2,950
vasy_8_24	20,306	11	560	5.9	26.8	22.1	282	17	348
vasy_8_38	8,922	81	220	5.7	44.1	31.5	174	5	215
vasy_83_325	393,147	211	392,175	162,495.0	1,074.4	t/o	173,218	19	t/o

Table 3: Results of running the partition refinement algorithms on the VLTS benchmark set.

This might be seen as contradictory since these partition refinement algorithms have inherently linear parallel run-times.

When comparing the different partition refinement algorithms, the structure of the input DFA is of high influence. The trade-off is that in `sortPR` each iteration takes more time than in `naivePR`, but in `sortPR`, an iteration has the potential to lead to a block being split into more than two subblocks. When this happens sufficiently often, fewer iterations are needed. This leads to `sortPR` being slower in cases where the number of iterations is high. In other benchmarks, it leads to fewer iterations and thereby a significant speed-up.

Finally, we showed a way to incorporate a partial transitive closure in partition refinement algorithms. We showed that for a specific class of DFAs this approach leads to logarithmic run-times, where every partition refinement algorithm is inherently linear.

As future work it would be interesting to further investigate sublinear time parallel algorithms for DFA minimization. Specifically, there are two key questions that come to mind. The first question is: what is a reasonable number of parallel processes necessary for a poly-logarithmic time parallel algorithm? It seems feasible to use a similar argument as in [11] to get a superlinear lower bound. However, the gap between the $O(n^{2\omega})$ processors[3] used in Algorithm 1 remains large. The second question is: can a method such as the one presented in this paper using the partial transitive closure be implemented in such a way that the run time will be sublinear with high probability, i.e. any parallel run-time $O(n^{1-\varepsilon})$ for some $\varepsilon \geq 0$. A good starting point would be the recent work on parallel reachability algorithms [20, 5, 10].

[3] If matrix multiplication can be computed in time $O(n^{\omega})$, currently best known bounds $\omega \leq 2.372\ldots$.

References

[1] J. Balcázar, J. Gabarro & M. Santha (1992): *Deciding bisimilarity is P-complete*. Formal aspects of computing 4(1), pp. 638–648, doi:10.1007/BF03180566.

[2] N. Bell & J. Hoberock (2012): *Thrust: A Productivity-Oriented Library for CUDA*. In: GPU Computing Gems Jade Edition, chapter 26, Morgan Kaufmann Publishers Inc., pp. 359–371, doi:10.1016/C2010-0-68654-8.

[3] G. Castiglione, A. Restivo & M. Sciortino (2008): *Hopcroft's Algorithm and Cyclic Automata*. In C. Martín-Vide, F. Otto & H. Fernau, editors: Proc. of LATA 2008, LNCS 5196, Springer, pp. 172–183, doi:10.1007/978-3-540-88282-4_17.

[4] S. Cho & D.T. Huynh (1992): *The parallel complexity of coarsest set partition problems*. Information Processing Letters 42(2), pp. 89–94, doi:10.1016/0020-0190(92)90095-D.

[5] J.T. Fineman (2018): *Nearly Work-Efficient Parallel Algorithm for Digraph Reachability*. In: Proc. of STOC 2018, ACM, p. 457–470, doi:10.1145/3188745.3188926.

[6] J.F. Groote, J.J.M. Martens & E.P. de Vink (2023): *Lowerbounds for bisimulation by partition refinement*. Logical Methods in Computer Science Volume 19, Issue 2, doi:10.46298/lmcs-19(2:10)2023.

[7] W.D. Hillis & G.L. Steele Jr. (1986): *Data Parallel Algorithms*. Communications of the ACM 29(12), pp. 1170–1183, doi:10.1145/7902.7903.

[8] J. Hopcroft (1971): *An $n \log n$ algorithm for minimizing states in a finite automaton*. In Z. Kohavi & A. Paz, editors: Theory of Machines and Computations, Academic Press, pp. 189–196, doi:10.1016/b978-0-12-417750-5.50022-1.

[9] J. JáJá (1992): *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[10] A. Jambulapati, Y.P. Liu & A. Sidford (2019): *Parallel reachability in almost linear work and square root depth*. In: Proc. of FOCS 2019, IEEE, pp. 1664–1686, doi:10.1109/FOCS.2019.00098.

[11] S. Khuller & U. Vishkin (1994): *On the parallel complexity of digraph reachability*. Information Processing Letters 52(5), pp. 239–241, doi:10.1016/0020-0190(94)00153-7.

[12] J.J.M. Martens, J.F. Groote, L.B. Haak, P. Hijma & A.J. Wijs (2022): *Linear parallel algorithms to compute strong and branching bisimilarity*. Software and Systems Modeling, pp. 1–25, doi:10.1007/s10270-022-01060-7.

[13] E.F. Moore (1956): *Gedanken-Experiments on Sequential Machines*. In Claude Shannon & John McCarthy, editors: Automata Studies, Princeton University Press, Princeton, NJ, pp. 129–153, doi:10.1515/9781400882618-006.

[14] R. Paige & R. E. Tarjan (1987): *Three partition refinement algorithms*. SIAM Journal on Computing 16(6), pp. 973–989, doi:10.1137/0216062.

[15] M.O. Rabin & D. Scott (1959): *Finite automata and their decision problems*. IBM Journal of Research and Development 3(2), pp. 114–125, doi:10.1147/rd.32.0114.

[16] B. Ravikumar & X. Xiong (1996): *A Parallel Algorithm for Minimization of Finite Automata*. In: Proceedings of the 10th International Parallel Processing Symposium, IPPS '96, IEEE Computer Society, USA, pp. 187–191, doi:10.1109/IPPS.1996.508056.

[17] L. Stockmeyer & U. Vishkin (1984): *Simulation of parallel random access machines by circuits*. SIAM Journal on Computing 13(2), pp. 409–422, doi:10.1137/0213027.

[18] A. Tewari, U. Srivastava & P. Gupta (2002): *A Parallel DFA Minimization Algorithm*. In: Proc. of HiPC, LNCS 2552, Springer, pp. 34–40, doi:10.1007/3-540-36265-7_4.

[19] B.A. Trakhtenbrot & J.M. Barzdin (1973): *Finite automata: behavior and synthesis*. North-Holland Publishing.

[20] J. Ullman & M. Yannakakis (1990): *High-Probability Parallel Transitive Closure Algorithms*. In: Proc. of SPAA 1990, pp. 200–209, doi:10.1145/97444.97686.

[21] A.J. Wijs (2015): *GPU Accelerated Strong and Branching Bisimilarity Checking*. In C. Baier & C. Tinelli, editors: *Proc. of TACAS*, *LNCS* 9035, Springer, pp. 368–383, doi:10.1007/978-3-662-46681-0_29.