On the Work of Dynamic Constant-Time Parallel Algorithms for Regular Tree Languages and Context-Free Languages

Jonas Schmidt
TU Dortmund University, Germany

Thomas Schwentick
TU Dortmund University, Germany

Jennifer Todtenhoefer
TU Dortmund University, Germany

Abstract

Previous work on Dynamic Complexity has established that there exist dynamic constant-time parallel algorithms for regular tree languages and context-free languages under label or symbol changes. However, these algorithms were not developed with the goal to minimise work (or, equivalently, the number of processors). In fact, their inspection yields the work bounds $\mathcal{O}(n^2)$ and $\mathcal{O}(n^7)$ per change operation, respectively.

In this paper, dynamic algorithms for regular tree languages are proposed that generalise the previous algorithms in that they allow unbounded node rank and leaf insertions, while improving the work bound from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^\epsilon)$, for arbitrary $\epsilon > 0$.

For context-free languages, algorithms with better work bounds (compared with $\mathcal{O}(n^7)$) for restricted classes are proposed: for every $\epsilon > 0$ there are such algorithms for deterministic context-free languages with work bound $\mathcal{O}(n^{3+\epsilon})$ and for visibly pushdown languages with work bound $\mathcal{O}(n^{2+\epsilon})$.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Parallel algorithms

Keywords and phrases Dynamic complexity, work, parallel constant time

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.81

Related Version *Full Version*: `http://arxiv.org/abs/2307.10131`

Acknowledgements We are grateful to Jens Keppeler and Christopher Spinrath for careful proof reading.

1 Introduction

It has been known for many years that regular and context-free string languages and regular tree languages are maintainable under symbol changes by means of dynamic algorithms that are specified by formulas of first-order logic, that is, in the dynamic class DynFO [10, 7]. It is also well-known that such specifications can be turned into parallel algorithms for the CRCW PRAM model that require only constant time [8] and polynomially many processors.

However, an "automatic" translation of a "dynamic program" of the DynFO setting usually yields a parallel algorithm with large work, i.e., overall number of operations performed by all processors.[1] In the case of regular languages, the dynamic program sketched in [10] has a polynomial work bound, in which the exponent of the polynomial depends on the number of states of a DFA for the language at hand. The dynamic program given in [7] has quadratic work.

[1] We note that in the context of constant-time parallel algorithms work is within a constant factor of the number of processors.

© Jonas Schmidt, Thomas Schwentick, and Jennifer Todtenhoefer;
licensed under Creative Commons License CC-BY 4.0

Only recently a line of research has started that tries to determine, how efficient such constant-time dynamic algorithms can be made with respect to their work. It turned out that regular languages can be maintained with work $\mathcal{O}(n^\epsilon)$, for every $\epsilon > 0$ [11], even under polylogarithmic numbers of changes [12], and even with logarithmic work for star-free languages under single changes [11] and polylogarithmic work under polylogarithmic changes [12].

For context-free languages the situation is much less clear. The dynamic algorithms resulting from [7] have an $\mathcal{O}(n^7)$ upper work bound. In [11] it was shown that the Dyck-1 language, i.e., the set of well-bracketed strings with one bracket type, can be maintained with work $\mathcal{O}((\log n)^3)$ and that Dyck-k languages can be maintained with work $\mathcal{O}(n \log n)$. Here, the factor n is due to the problem to test equality of two substrings of a string.

Most of these results also hold for the query that asks for membership of a substring in the given language. For Dyck languages the upper bounds for substring queries are worse than the bounds for membership queries: for every $\epsilon > 0$ there exist algorithms for Dyck-1 and Dyck-k languages with work bounds $\mathcal{O}(n^\epsilon)$ and $\mathcal{O}(n^{1+\epsilon})$, respectively.

It was also shown in [11] that there is some context-free language that can be maintained in constant time with work $\mathcal{O}(n^{\omega-1-\epsilon})$, for any $\epsilon > 0$, only if the k-Clique conjecture [1] fails. Here, ω is the matrix multiplication exponent, which is known to be smaller than 2.373 and conjectured by some to be exactly two according to [14].

In this paper, we pursue two natural research directions.

Regular tree languages. We first extend the results on regular string languages to regular tree languages. On one hand, this requires to adapt techniques from strings to trees. On the other hand, trees offer additional types of change operations beyond label changes that might change the structure of the tree. More concretely, besides label changes we study insertions of new leaves and show that the favourable bounds of [11] for regular string languages still hold. This is the main contribution of this paper. Our algorithms rely on a hierarchical partition of the tree of constant depth. The main technical challenge is to maintain such a partition hierarchy under insertion[2] of leaves.

Subclasses of context-free languages. We tried to improve on the $\mathcal{O}(n^7)$ upper work bound for context-free languages, but did not succeed yet. The other goal of this paper is thus to find better bounds for important subclasses of the context-free languages: deterministic context-free languages and visibly pushdown languages. We show that, for each $\epsilon > 0$, there are constant-time dynamic algorithms with work $\mathcal{O}(n^{3+\epsilon})$ for deterministic context-free languages and $\mathcal{O}(n^{2+\epsilon})$ for visibly pushdown languages. Here, the main challenge is to carefully apply the technique from [11] that allows to store information for only $\mathcal{O}(n^\epsilon)$ as opposed to n different values for some parameters. For more restricted change operations, the algorithm for regular tree languages yields an $\mathcal{O}(n^\epsilon)$ work algorithm for visibly pushdown languages.

Structure of the paper. We explain the framework in Section 2, and present the results on regular tree languages and context-free string languages in Sections 3 and 4, respectively. Almost all proofs are delegated to the full version of this paper.

[2] For simplicity, we only consider insertions of leaves, but deletions can be handled in a straightforward manner, as discussed in Section 2.

Related work. In [12], parallel dynamic algorithms for regular string languages under bulk changes were studied. It was shown that membership in a regular language can be maintained, for every $\epsilon > 0$, in constant time with work $\mathcal{O}(n^\epsilon)$, even if a polylogarithmic number of changes can be applied in one change operation. If the language is star-free, polylogarithmic work suffices. The paper also shows that for regular languages that are not star-free, polylogarithmic work does *not* suffice.

Maintaining regular languages of trees under label changes has also been studied in the context of enumeration algorithms (for non-Boolean queries) [3]. The dynamic parallel algorithms of [11] partially rely on dynamic sequential algorithms, especially [6].

2 Preliminaries

Trees and regular tree languages. We consider ordered, unranked trees t, which we represent as tuples (V, r, c, label), where V is a finite set of nodes, $r \in V$ is the root, $c : V \times \mathbb{N} \to V$ is a function, such that $c(u, i)$ yields the i-th child of u, and $\text{label} : V \to \Sigma$ is a function that assigns a label to every node.

We denote the set of unranked trees over an alphabet Σ as $T(\Sigma)$. The terms *subtree*, *subforest*, *sibling*, *ancestor*, *descendant*, *depth* and *height* of nodes are defined as usual. A node that has no child is called a *leaf*. A *forest* is a sequence of trees.

Let $\preceq$ denote the order on siblings, i.e., $u \prec v$ denotes that u is a sibling to the left of v. We write $u \preceq v$ if $u \prec v$ or $u = v$ holds.

By t^v we denote the subtree of t induced by node v. For sibling nodes $u \prec v$, we write ${}^u t^v$ for the subforest of the tree t, induced by the sequence $u, \dots, v$. If w is a node in t^v, then t^v_w denotes the subtree consisting of t^v without t^w. Analogously, for ${}^u t^v_w$.

Our definition of tree automata is inspired from hedge automata in the TaTa book [5], slightly adapted for our needs.

▶ **Definition 2.1.** *A* deterministic finite (bottom-up) tree automaton (DTA) *over an alphabet Σ is a tuple $\mathcal{B} = (Q_\mathcal{B}, \Sigma, Q_f, \delta, \mathcal{A})$ where $Q_\mathcal{B}$ is a finite set of states, $Q_f \subseteq Q_\mathcal{B}$ is a set of final states, $\mathcal{A} = (Q_\mathcal{A}, Q_\mathcal{B}, \delta_\mathcal{A}, s)$ is a DFA over alphabet $Q_\mathcal{B}$ (without final states) and $\delta : Q_\mathcal{A} \times \Sigma \to Q_\mathcal{B}$ maps pairs (p, σ), where p is a state of $\mathcal{A}$ and $\sigma \in \Sigma$, to states of $\mathcal{B}$.*

We refer to states from $Q_\mathcal{B}$ as $\mathcal{B}$*-states* and typically denote them by the letter q. Likewise states from $Q_\mathcal{A}$ are called $\mathcal{A}$*-states* and denoted by p. We note that we do not need a set of accepting states for $\mathcal{A}$, since its final states are fed into δ.

The semantics of DTAs is defined as follows.

For each tree $t \in T(\Sigma)$, there is a unique *run* of $\mathcal{B}$ on t, that is, a unary function ρ_t that assigns a $\mathcal{B}$-state to each node in V. It can be defined in a bottom-up fashion, as follows. For each node $v \in V$ with label σ and children $u_1, \dots, u_\ell$, $\rho_t(v)$ is the $\mathcal{B}$-state $\delta(\delta^*_\mathcal{A}(s, \rho_t(u_1) \cdots \rho_t(u_\ell)), \sigma)$. That is, the state of a node v with label σ is determined by $\delta(p, \sigma)$, where p is the final $\mathcal{A}$-state that $\mathcal{A}$ assumes when reading the sequence of states of v's children, starting from the initial state s. In particular, if v is a leaf with label σ, its $\mathcal{B}$-state is $\delta(s, \sigma)$.

A tree t is accepted by the DTA $\mathcal{B}$ if $\rho_t(r) \in Q_f$ holds for the root r of t. We denote the language of all trees accepted by $\mathcal{B}$ as $L(\mathcal{B})$. We call the languages decided by DTAs *regular*.

Strings and context-free languages. Strings w are finite sequences of symbols from an alphabet Σ. By $w[i]$ we denote the i-th symbol of w and by $w[i, j]$ we denote the substring from position i to j. We denote the empty string by λ, since ϵ has a different purpose in this paper. We use standard notation for context-free languages and pushdown automata, to be found in the full version of this paper.

Dynamic algorithmic problems. In this paper, we view a dynamic (algorithmic) problem basically as the interface of a data type: that is, there is a collection of operations by which some object can be initialised, changed, and queried. A *dynamic algorithm* is then a collection of algorithms, one for each operation. We consider two main dynamic problems in this paper, for regular tree languages and context-free languages.

For each regular tree language L, the algorithmic problem $\textsc{RegTree}(L)$ maintains a labelled tree T and has the following operations.

- `Init`(T, r, σ) yields an initial labelled tree object T and returns in r a node id for its root, which is labelled by σ;
- `Relabel`(T, u, σ) changes the label of node u in T into σ;
- `AddChild`(T, u, v, σ) adds a new child with label σ behind the last child of node u and returns its id in v;
- `Query`(T, v) returns true if and only if the subtree of T rooted at v is in L.

We refer to the restricted problem without the operation `AddChild` as $\textsc{RegTree}^{-}$. For this data type, we assume that the computation starts from an initial non-trivial tree and that the auxiliary data for that tree is given initially.

For each context-free language L, the algorithmic problem $\mathrm{CFL}(L)$ maintains a string w and has the following operations.

- `Init`(w) yields an initial string object w with an empty string;
- `Relabel`(w, i, σ) changes the label at position i of w into σ;
- `InsertPositionBefore`(w, i, σ) and `InsertPositionAfter`(w, i, σ) insert a new position with symbol σ before or after the current position i, respectively;
- `Query`(w, i, j) returns true if and only if the substring $w[i, j]$ is in L.

Readers may wonder, why these dynamic problems do not have operations that delete nodes of a tree or positions in a string. This is partially to keep the setting simple and partially because node labels and symbols offer easy ways to simulate deletion by extending the alphabet with a symbol $\sqcup$ that indicates an object that should be ignored. E.g., if $\delta_{\mathcal{A}}(p, \sqcup) = p$, for every state p of the horizontal DFA of a DTA, then the label $\sqcup$ at a node u effectively deletes the whole subtree induced by u for the purpose of membership in $L(\mathcal{B})$. Similarly, a CFL might have a neutral symbol or even a pair $(_{\sqcup},)_{\sqcup}$ of "erasing" brackets that make the PDA ignore the substring between $(_{\sqcup}$ and $)_{\sqcup}$.

For $\textsc{RegTree}(L)$ and $\mathrm{CFL}(L)$, the `Init` operation is possible in constant sequential time and will not be considered in detail.

Throughout this paper, n will denote an upper bound of the size of the structure at hand (number of nodes of a tree or positions of a string) that is linear in that size, but changes only infrequently. More precisely, the number of nodes of a tree or the length of the string will always be between $\frac{1}{4}n$ and n. Whenever the size of the structure grows beyond $\frac{1}{2}n$, the data structure will be prepared for structures of size up to $2n$ and, once this is done, n will be doubled. Since the size of the structure is always $\theta(n)$ all bounds in n also hold with respect to the size of the structure.

Parallel Random Access Machines (PRAMs). A *parallel random access machine* (PRAM) consists of a number of processors that work in parallel and use a shared memory. The memory is comprised of memory cells which can be accessed by a processor in $\mathcal{O}(1)$ time. Furthermore, we assume that simple arithmetic and bitwise operations, including addition, can be done in $\mathcal{O}(1)$ time by a processor. We mostly use the Concurrent-Read Concurrent-Write model (CRCW PRAM), i.e. processors are allowed to read and write concurrently from and to the same memory location. More precisely, we assume the *common* PRAM model:

several processors can concurrently write into the same memory location, only if all of them write the same value. We also mention the Exclusive-Read Exclusive-Write model (EREW PRAM), where concurrent access is not allowed. The work of a PRAM computation is the sum of the number of all computation steps of all processors made during the computation. We define the space s required by a PRAM computation as the maximal index of any memory cell accessed during the computation. We refer to [9] for more details on PRAMs and to [13, Section 2.2.3] for a discussion of alternative space measures.

The main feature of the common CRCW model relevant for our algorithms that separates it from the EREW model is that it allows to compute the minimum or maximum value of an array of size n in constant time (with work $\mathcal{O}(n^{1+\epsilon})$) which is shown in another paper at MFCS 2023.[3]

For simplicity, we assume that even if the size bound n grows, a number in the range $[0, n]$ can still be stored in one memory cell. This assumption is justified, since addition of larger numbers N can still be done in constant time and polylogarithmic work on a CRCW PRAM. Additionally, we assume that the number of processors always depends on the current size bound n. Hence, the number of processors increases with growing n which allows us to use the PRAM model with growing structures.

We describe our PRAM algorithms on an abstract level and do not exactly specify how processors are assigned to data. Whenever an algorithm does something in parallel for a set of objects, these objects can be assigned to a bunch of processors with the help of some underlying array. This is relatively straightforward for strings and substrings and the data structures used in Section 4. In Section 3, it is usually based on zone records and their underlying partition records.

3 Maintaining regular tree languages

In this section, we present our results on maintaining regular tree languages under various change operations. We will first consider only operations that change node labels, but do not change the shape of the given tree. A very simple dynamic algorithm with work $\mathcal{O}(n^2)$ is presented in the full version of this paper. We sketch its main idea and how it can be improved to $\mathcal{O}(n^\epsilon)$ work per change operation by using a *partition hierarchy* in Subsection 3.1. These algorithms even work on the EREW PRAM model.

Afterwards, in Subsection 3.2, we also consider an operation that can change the tree structure: adding a leaf to the tree. Here, the challenge is to maintain the hierarchical structure that we used before to achieve work $\mathcal{O}(n^\epsilon)$ per change operation. It turns out that maintaining this structure is possible without a significant increase of work, that is, maintaining membership under these additional operations is still possible with work $\mathcal{O}(n^\epsilon)$ per change operation.

3.1 Label changes: a work-efficient dynamic program

In this section, we describe how membership in a regular tree language can be maintained under label changes, in a work efficient way.

[3] Jonas Schmidt, Thomas Schwentick. Dynamic constant time parallel graph algorithms with sub-linear work.

▶ **Proposition 3.1.** *For each $\epsilon > 0$ and each regular tree language L, there is a parallel constant time dynamic algorithm for* $\textsc{RegTree}^-(L)$ *with work $\mathcal{O}(n^\epsilon)$ on an EREW PRAM. The* `Query` *operation can actually be answered with constant work.*

We start by briefly sketching the $\mathcal{O}(n^2)$ work algorithm that is given in the full version of this paper. The algorithm basically combines the dynamic programs for regular string languages and binary regular tree languages from [7]. For regular string languages, the program from [7] stores the behaviour of a DFA for the input word w by maintaining information of the form "if the run of the DFA starts at position i of w and state p, then it reaches state q at position j" for all states p, q and substrings $w[i, j]$. After a label change at a position ℓ, this information can be constructed by combining the behaviour of the DFA on the intervals $w[i, \ell - 1]$ and $w[\ell + 1, j]$ with the transitions induced by the new label at position ℓ.

The dynamic program for (binary) regular tree languages from [7] follows a similar idea and stores the behaviour of a (binary) bottom-up tree automaton by maintaining information of the form "if v gets assigned state q, then u gets assigned state p by the tree automaton" for all states p, q and all nodes v, u, where v is a descendant of u.

Both programs induce algorithms with $\mathcal{O}(n^2)$ work bounds. Towards a $\mathcal{O}(n^2)$ work algorithm for unranked tree languages, the two dynamic programs can be combined into an algorithm that mainly stores the following *automata functions* for a fixed DTA $\mathcal{B} = (Q_\mathcal{B}, \Sigma, Q_f, \delta, \mathcal{A})$ for L, with DFA $\mathcal{A} = (Q_\mathcal{A}, Q_\mathcal{B}, \delta_\mathcal{A}, s)$:

- The ternary function $\mathcal{B}_t : Q_\mathcal{B} \times V \times V \mapsto Q_\mathcal{B}$ maps each triple (q, u, v) of a state $q \in Q_\mathcal{B}$ and nodes of t, where u is a proper ancestor of v, to the state that the run of $\mathcal{B}$ on t_v^u takes at u, with the provision that the state at v is q.
- The ternary function $\mathcal{A}_t : Q_\mathcal{A} \times V \times V \mapsto Q_\mathcal{A}$ maps each triple (p, u, v) of a state $p \in Q_\mathcal{A}$ and nodes of t, where $u \prec v$ are siblings, to the state that the run of $\mathcal{A}$ on $u, \ldots, v$, starting from state p, takes after v.

Every single function value can be updated in constant sequential time, as stated in the following lemma. This leads to a quadratic work bound since there are quadratically many tuples to be updated in parallel.

▶ **Lemma 3.2.** *After a* `Relabel` *operation, single values $\mathcal{A}_t(p, u, x)$ and $\mathcal{B}_t(q, u, x)$ can be updated by a sequential algorithm in constant time.*

Some information about the shape of the tree is required, which we refer to as *basic tree functions*. For more details we refer to the full version of this paper. However, as label changes cannot change the shape of the tree, this information does not need to be updated und can be assumed as precomputed.

To lower the work bound the basic idea now is to store the automata functions not for *all* possible arguments, but for a small subset of *special* arguments that allow the computation of function values for *arbitrary* arguments in constant time with constant work.

In [11], this idea was applied to the $\mathcal{O}(n^2)$ work program for regular string languages. A constant-depth hierarchy of intervals was defined by repeatedly partitioning intervals into $\mathcal{O}(n^\theta)$ subintervals, for some $\theta > 0$. This hierarchy allowed to define *special* intervals such that any update only affects $\mathcal{O}(n^\epsilon)$ intervals and function values of arbitrary intervals can be computed in constant time with constant work.

We transfer this idea to the case of unranked tree languages by partitioning the tree into $\mathcal{O}(n^\theta)$ *zones*, each of which is partitioned into further $\mathcal{O}(n^\theta)$ zones and so on until, after a constant number of refinements, we arrive at zones of size $\mathcal{O}(n^\theta)$. Here, $\theta > 0$ is a constant that will be chosen later. It will always be chosen such that $h = \frac{1}{\theta}$ is an integer.

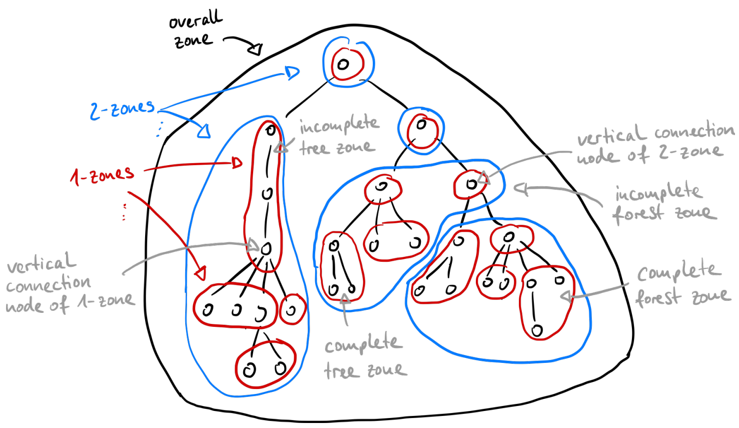

Figure 1 Example of a $(1, \frac{1}{3})$-bounded partition hierarchy.

Before we define this partition hierarchy more precisely, we first define zones and show that they can always be partitioned in a way that guarantees certain number and size constraints.

▶ **Definition 3.3.** A *zone* is a set S of nodes with the following properties:

- S is a proper subforest of t,
- for every $v \in S$ it holds that either no or all children are in S, and
- there exists at most one node v_S in S, whose children are not in S. The node v_S is called the *vertical connection node* of S.

We call a zone a *tree zone* if it consists of only one sub-tree of t and a *non-tree zone* otherwise. We call a zone *incomplete* if it has a vertical connection node and *complete*, otherwise. There are thus four different types of zones which can be written, with the notation introduced in Section 2, as follows: complete tree zones t^v, complete non-tree zones ${}^u t^v$, incomplete tree zones t^v_w, and incomplete non-tree zones ${}^u t^v_w$. Depending on the type, zones can therefore be represented by one to three "important nodes". The overall tree can be seen as the zone t^r, where r is its root.

From now on, we always assume that n is as in Section 2, some $\theta > 0$ is fixed, and that $h = \frac{1}{\theta}$ is an integer.

We call a zone of t with at most $n^{\theta\ell}$ nodes an ℓ*-zone*. The tree t itself constitutes a h-zone, to which we will refer to as the *overall zone*.

We next define *partition hierarchies* formally. More precisely, for every $\ell \geq 2$, we define partition hierarchies of height ℓ for ℓ-zones as follows. If S is a 2-zone and $S_1, \dots, S_k$ are 1-zones that constitute a partition of S, then $(S, \{S_1, \dots, S_k\})$ is a partition hierarchy of height 2 for S. If S is an $(\ell + 1)$-zone, $\{S_1, \dots, S_k\}$ is a partition of S into ℓ-zones, and for each j, H_j is a partition hierarchy of height i for S_j, then $(S, \{H_1, \dots, H_k\})$ is a partition hierarchy of height $\ell + 1$ for S. A partition hierarchy of height h of the zone consisting of t is called a partition hierarchy of t.

An example of a $(1, \frac{1}{3})$-bounded partition hierarchy is given in Figure 1.

We often call a zone S' that occurs at some level $i < \ell$ within the partition hierarchy of a zone S of some level ℓ a *component zone*. If S' has level $\ell - 1$ we also call it a *sub-zone* of S.

We call a partition hierarchy H (c, θ)*-bounded*, constants c and $\theta > 0$, if each partition of a zone consists of at most cn^θ nodes.

Our next aim is to prove that $(10, \theta)$-bounded partition hierarchies actually exist. To this end, we prove the following lemma. It is similar to [4, Lemma 3], but adapted to our context, which requires a hierarchy of constant depth and a certain homogeneity regarding children of vertical connection nodes.

▶ **Lemma 3.4.** *Let $m \geq 2$ be a number and S a zone with more than m nodes. Then S can be partitioned into at most five zones, one of which has at least $\frac{1}{2}m$ and at most m nodes.*

This lemma immediately yields the existence of $(10, \theta)$-bounded partition hierarchies.

▶ **Proposition 3.5.** *For each $\theta > 0$, each tree t has some $(10, \theta)$-bounded partition hierarchy.*

We now explain in more detail, which information about the behaviour of $\mathcal{A}$ and $\mathcal{B}$ is stored by the work-efficient algorithm.

Function values for the ternary functions are stored only for so-called special pairs of nodes, which we define next. Special pairs of nodes are always defined in the context of some zone S of a partition hierarchy. In the following, we denote, for a zone S of a level $\ell \geq 2$ its set of sub-zones of level $\ell - 1$ by T.

- Any pair of siblings $u \prec v$ in a zone S of level 1 is a *special horizontal pair.* A pair of siblings $u \prec v$ in a complete zone S of level $\ell \geq 2$ is a *special horizontal pair*, if u is a left boundary of some zone in T and v is a right boundary of some zone in T. However, if S is incomplete and there is an ancestor w' of the lower boundary w with $u \preceq w' \preceq v$, then, instead of (u, v), there are two special pairs: $(u, \textsc{left-sibling}(w'))$ and $(\textsc{right-sibling}(w'), v)$.
- Any pair of nodes u, v in some zone S of level 1 is a *special vertical pair*, if v is an ancestor of u. A pair of nodes u, v in some zone S of level $\ell \geq 2$ is a *special vertical pair*, if v is an ancestor of u, v is an upper or lower boundary of some zone in T and u is a lower boundary of some zone in T. However, if S is incomplete with lower boundary w and $w' := \textsc{lca}(w, u)$ is strictly above u and below or equal to v, then, instead of (u, v), there are two special pairs: $(u, \textsc{anc-child}(w', u))$ and (w', v). Here $\textsc{lca}$ determines the least common ancestor and $\textsc{anc-child}$ the child of w' that is an ancestor of u.

The algorithm stores $\mathcal{A}_t(p, u, v)$ for each state p of $\mathcal{A}$ and each special horizontal pair u, v. Furthermore, it stores $\mathcal{B}_t(q, u, v)$, for each state q of $\mathcal{B}$ and each special vertical pair u, v.

We note, that in all cases $\mathcal{A}_t(p, u, v)$ and $\mathcal{B}_t(q, u, v)$ only depend on the labels of the nodes in the zone, for which (u, v) is special.

▶ **Lemma 3.6.** *From the stored values for functions $\mathcal{A}_t$ and $\mathcal{B}_t$ for special pairs, it is possible to compute $\rho_t(v)$, for arbitrary nodes v, $\mathcal{A}_t(p, u, u')$ for arbitrary pairs $u \prec u'$ of siblings of t and $\mathcal{B}_t(q, u, u')$ for arbitrary pairs u, u' of nodes, where u' is an ancestor of u, sequentially in constant time.*

This enables us to show the $\mathcal{O}(n^\epsilon)$ work bound for label changes.

Proof of Proposition 3.1. To achieve the stated bound, we use the above algorithm with work parameter $\theta = \frac{\epsilon}{2}$. The algorithm uses a $(\theta, 10)$-bounded partition hierarchy, which exists thanks to Proposition 3.5.

As indicated before, the algorithm stores $\mathcal{A}_t(\cdot, u, v)$ and $\mathcal{B}_t(\cdot, u, v)$, for all special pairs (u, v). As already observed before, these values only depend on the labels of the nodes of the zone relative to which (u, v) is special. Therefore, if a node label is changed for some node x, values $\mathcal{A}_t(\cdot, u, v)$ and $\mathcal{B}_t(\cdot, u, v)$ need only be updated for special pairs of zones in which x occurs. Since each node occurs in exactly h zones and each zone has $\mathcal{O}(n^{2\theta}) = \mathcal{O}(n^\epsilon)$ special pairs, $h \cdot \mathcal{O}(n^\epsilon)$ processors can be used, where every processor updates a single value in constant time and work, as is possible thanks to Lemma 3.2 and Lemma 3.6. Since the shape of the tree does not change we can assume a mapping from the updated node and the processor number to the special tuple that the respective processor recomputes. ◀

3.2 Structural Changes

In Proposition 3.1 only label changes were allowed, so the structure of the underlying tree did not change. In particular, there was no need to update any of the basic tree functions.

In this subsection, we consider structural changes of the tree. We show that the work bounds of Proposition 3.1 can still be met for the full data types RegTree(L).

▶ **Theorem 3.7.** *For each regular tree language L and each $\epsilon > 0$, there is a dynamic constant time parallel algorithm for* RegTree(L) *that handles change operations with work $\mathcal{O}(n^\epsilon)$ and answers query operations with constant work.*

In the next subsection, we describe the general strategy of the algorithm, define some notions that will be used and present its proof. Then, in a second subsection, we give some more detailed information about the data that is stored and how it can be maintained.

3.2.1 High-level description of the dynamic algorithm

Our approach generalises the algorithm of Subsection 3.1. It makes sure that, at any point in time, there is a valid partition hierarchy together with corresponding tree and automata functions. The general strategy of the dynamic algorithm is to add new leaves to their nearest zone. In principle, this is not hard to handle – unless it leads to a violation of a size constraint of some zone. As soon as zones exceed a certain size bound the affected parts of the hierarchy will thus be recomputed to ensure the size constraints.

For reasons that will become clearer below, we need to slightly modify the definition of partition hierarchies, basically by omitting the lowest two levels. To this end, we define 3-pruned partition hierarchies just like we defined partition hierarchies, but the lowest level is at height 3. More precisely, *a 3-pruned partition hierarchy of height* 3 is just a 3-zone, and 3-*pruned partition hierarchies of height* $\ell > 3$ are inductively defined just like partition hierarchies of height ℓ. It is clear that a 3-pruned partition hierarchy exists for each tree by ommiting the two lowest levels in the partition hierarchy computed in Proposition 3.5. Moreover, using a 3-pruned partition hierarchy as basis for our efficient label change approach still ensures the sequential constant time computation of arbitrary automaton function values from the stored values for special pairs. However, zones on the lowest level have size $\mathcal{O}(n^{3\theta})$ leading to a work bound of $\mathcal{O}(n^{6\theta})$ per change operation.

To ensure that at each point in time, a usable partition hierarchy is available, the general strategy is as follows: the algorithm starts from a *strong partition hierarchy* in which zones at level ℓ have size at most $\frac{1}{4}n^{\ell\theta}$, well below the maximum allowed size of such a zone of $n^{\ell\theta}$. As soon as the size of a zone S at level ℓ reaches its *warning limit* $\frac{1}{2}n^{\ell\theta}$, the algorithm starts to compute a new partition hierarchy for the parent zone S' of S at level $\ell + 1$. This computation is orchestrated in a way that makes sure that the new partition hierarchy for S' is ready (together with all required function values) before S reaches its size limit $n^{\ell\theta}$, at which point the old partition hierarchy for S' becomes useless.

Since a partition hierarchy of the whole tree together with the required function values has size $\Omega(n)$, its computation inherently requires that amount of work and it can probably not be done in constant time. Furthermore, since we aim at work $\mathcal{O}(n^\epsilon)$ per operation, the algorithm cannot afford to do the re-computation "as fast as possible" but rather needs to stretch over at least $n^{1-\epsilon}$ steps. However, the fact that the tree can change during a re-computation poses a challenge: if many change operations happen with respect to a particular zone in a low level of the new partition hierarchy, this new zone might reach its warning limit and then its hard limit, before the overall re-computation of the hierarchy has finished. This challenge can be met by a careful orchestration of the re-computation.

We will next describe the data structure that the dynamic algorithm uses to orchestrate re-computations of partition hierarchies. As mentioned before, there will always be a valid partition hierarchy. However, for some zones, re-computations might be underway. The algorithm will always manage to complete the re-computation of a partition hierarchy for a zone of level ℓ, before any of the subzones of level $(\ell-1)$ of the new partition reaches its warning limit. Therefore, for each zone within the data structure, there is always at most one partition hierarchy under construction, and therefore each zone has at any time at most two partition records. If a zone actually has two partition records, one of them contains a usable partition hierarchy. We formalise usability of a partition hierarchy by the term *operable* and tie the whole data structure together through the following notion of zone records. It is defined in an inductive fashion, together with the concept of partition records.

▶ **Definition 3.8.** A *zone record* of level 3 is a 3-zone. A *zone record* of level $\ell > 3$ consists of an ℓ-zone S and up to two partition records P_1, P_2 of level ℓ for S. If it has two partition records then P_1 is complete and P_2 is incomplete.

A *partition record* (Z, M) of level $\ell > 3$ for an ℓ-zone S consists of a set Z of zone records of level $\ell - 1$ and a set M of zones, such that the zones from Z and the zones from M together constitute a partition of S. A partition record Z of level ℓ is *valid*, if all zones of its zone records are actual $(\ell-1)$-zones.

A zone record of level 3 is *operable*.

A partition record at level $\ell > 3$ is *operable*, if it is valid and all its zone records are operable. A zone record of level $\ell > 3$ is *operable*, if its first partition record is operable.

We refer to the hierarchical structure constituting the overall zone record as the *extended partition hierarchy*. Within the extended partition hierarchy, we are particularly interested in "operable substructures". To this end, we associate with an operable zone record, the *primary partition hierarchy* that results from recursively picking the operable partition record from each zone record.

Altogether, the algorithm maintains an extended partition hierarchy for t.

Before we describe how the algorithm stores the extended partition hierarchy, we need two more concepts. For each zone record R of a level ℓ there is a sequence $R_h, \dots, R_\ell = R$ of zone records such that, for each $i \geq \ell$, R_i is a zone record that occurs in a partition record of R_{i+1}. This sequence can be viewed as the *address* of R in the extended partition hierarchy. Furthermore, this address induces a *finger print* for R: the sequence $\text{status}(R_h), \dots, \text{status}(R_\ell)$, where $\text{status}(R_i)$ is either *operable* or *in progress*. It is a simple but useful observation that if a tree node v occurs in two zones with zone records $R \neq R'$ within the extended partition hierarchy, then the finger prints of R and R' are different. Consequently a tree node occurs in at most 2^h and thus, a constant number of zones in the extended partition hierarchy.

Now we can describe, how the algorithm stores t and the extended partition hierarchy.

- A zone record of level 3 is represented as an array of $\mathcal{O}(n^\theta)$ nodes.
- A zone record of a level $\ell > 3$ consists of up to four boundary nodes and up to two pointers to partition records. The operable partition record is flagged.

- Each zone record of level $\ell \geq 3$ with finger print pa, also stores a pointer to its zone on level $\ell + 1$ with finger print p, and three pointers to the zone records of its parent, first child and right sibling zones.
- A partition record (Z, M) is represented as an array of zone records (some of which may be zones of M). The zones records from Z are flagged.
- The nodes of t are stored in an array (in no particular order) together with pointers for the functions PARENT, LEFT-SIBLING, RIGHT-SIBLING, FIRST-CHILD, and LAST-CHILD.
- For each node v, and each possible finger print p, a pointer $Z^p(v)$ to its zone with finger print p.

Now we are prepared to outline the proof of Theorem 3.7.

Proof (of Theorem 3.7). Let $\mathcal{B}$ be a DTA for the regular tree language L and let $\theta = \frac{\epsilon}{7}$. The dynamic algorithm stores t and an extended partition hierarchy as described above. It also stores some additional function values, including values for the automata functions, that will be specified in Subsubsection 3.2.2.

Some functions are independent from zones and are stored for all nodes. Some other functions are independent from zones but are only stored for particular node tuples that are induced from zones (like it was already the case for the automata functions in Subsection 3.1) and some functions are actually defined for (tuples of) zones.

After each change operation, the algorithm updates function values, pursues re-computations of hierarchies and computes function values that are needed for newly established zones. It starts a re-computation for a zone S, whenever one of its subzones reaches its warning limit. It starts a re-computation of the overall zone, whenever the number of nodes of t reaches $\frac{1}{2}n$.

The algorithm has one thread for each zone with an ongoing re-computation, that is, for each zone whose zone record is not yet operable.

A re-computation for a zone at level ℓ requires the computation of $\mathcal{O}(n^{\theta})$ zones of level $\ell - 1$, each of which yields re-computations of $\mathcal{O}(n^{\theta})$ zones of level $\ell - 2$ and so forth, down to level 3. It is easy to see that the overall number of zones that needs to be computed during a re-computation of a zone at level ℓ is bounded by $\mathcal{O}(n^{(\ell-3)\theta})$. The re-computation of the overall zone requires the computation of at most $\mathcal{O}(n^{1-3\theta})$ zones. We show in Lemma 3.9 that, in the presence of a primary partition hierarchy for the overall zone, the computation of a new zone is possible in constant time with work $\mathcal{O}(n^{6\theta})$.

The thread for the re-computation of a zone at level ℓ thus (first) consists of $\mathcal{O}(n^{(\ell-3)\theta})$ computations of component zones, each of which is carried out in constant time with work $\mathcal{O}(n^{6\theta})$. We refer to such a re-computation as a round. A thread thus consists of $\mathcal{O}(n^{(\ell-3)\theta})$ rounds of zone computations. The thread follows a breadth-first strategy, that is, it first computes all zones of level $\ell - 1$ then the sub-zones of those zones at level $\ell - 2$ and so forth. Once the zone record of a zone S is operable, the thread computes in its second phase all function values associated to S. This can be done in constant time with work $\mathcal{O}(n^{7\theta})$ per sub-zone of S, as is shown in the full version of this paper. That is, it requires at most $\mathcal{O}(n^{(\ell-3)\theta})$ additional rounds.

We note that it does not matter if the primary partition hierarchy H required for Lemma 3.9 changes during the computation of a thread, since H is only used to make the identification of a new zone more efficient.

To address the above mentioned challenge, the algorithm starts a separate thread for each zone that is newly created during this process. That is, for each zone at level $\ell - 1$, an additional re-computation thread is started, as soon as the zone is created.

Now we can state the orchestration strategy for re-computations. This strategy is actually very simple:

> **Re-computation strategy:** After each change operation affecting some node v, the algorithm performs one computation round, for all threads of zones S, at any level, with $v \in S$.

That is, thanks to the above observation, after a change operation, there are at most 2^h threads for which one computation round is performed. Since 2^h is a constant, these computations together require work at most $\mathcal{O}(n^{7\theta})$.

On the other hand, the whole re-computation for a zone S at level ℓ, including the computation of the relevant function values, is finished after at most $\mathcal{O}(n^{(\ell-3)\theta})$ change operations that affect S. Since $\frac{1}{2}n^{(\ell-1)\theta}$ leaf additions are needed to let a sub-zone S' grow from the warning limit $\frac{1}{2}n^{(\ell-1)\theta}$ to the hard limit $n^{(\ell-1)\theta}$, it is guaranteed that the re-computation thread for S is completed, before S' grows too large. In fact, this is exactly, why partition hierarchies are 3-pruned. When a re-computation of the overall zone was triggered by the size of t, n is doubled as soon as this re-computation is completed.

Thanks to Lemma 3.11 the overall work to update the stored function values for all affected zones (in constant time) after a change operation is $\mathcal{O}(n^{3\theta})$.

Altogether, the statement of the theorem follows by choosing $\theta = \frac{\epsilon}{7}$. ◀

We state the lemma about the computation of new zones next. The partition hierarchy is used as a means to assign evenly distributed nodes to processors and to do parallel search for nodes with a particular property regarding the number of their descendants.

▶ **Lemma 3.9.** *Given a tree t, a 3-pruned partition hierarchy H of t, and a zone S with at least m nodes, S can be partitioned into at most five zones, one of which has at least $\frac{1}{2}m$ and at most m nodes, in constant time with work $\mathcal{O}(n^{6\theta})$.*

3.2.2 Maintaining functions

In Subsection 3.1, the tree functions were static and given by the initialisation. Only the automata functions needed to be updated. However, if leaf insertions are allowed, the tree functions can change. To keep the algorithm efficient, the special pairs need to be adapted to the evolution of the partition hierarchy, and tree functions can no longer be stored for all possible arguments. Furthermore, additional tree functions and functions defined for zones will be used.

The stored information suffices to compute all required functions in constant time, and almost all of them with constant work.

▶ **Lemma 3.10.** *Given a tree t, a 3-pruned partition hierarchy H of t, and the stored information as described above, for each $\theta > 0$, the* CHILD *function can be evaluated in $\mathcal{O}(1)$ time with work $\mathcal{O}(n^{\theta})$. All other functions can be evaluated for all tuples with constant work.*

Furthermore, all stored information can be efficiently updated, with the help of and in accordance with the current primary partition hierarchy.

▶ **Lemma 3.11.** *Let $\theta > 0$ and H be a 3-pruned partition hierarchy of t with automata and tree functions. The stored information described above can be maintained after each* `Relabel` *and* `AddChild` *operation in constant time with $\mathcal{O}(n^{6\theta})$ work per operation.*

4 Maintaining context-free languages

As mentioned in the introduction, an analysis of the dynamic program that was used in [7] to show that context-free languages can be maintained in DynFO yields the following result.

▶ **Theorem 4.1** ([7, Proposition 5.3]). *For each context-free language L, there is a dynamic constant-time parallel algorithm on a CRCW PRAM for* $\mathrm{CFL}(L)$ *with* $\mathcal{O}(n^7)$ *work.*

There is a huge gap between this upper bound and the conditional lower bound of $\mathcal{O}(n^{\omega-1-\epsilon})$, for any $\epsilon > 0$, derived from the k-Clique conjecture [1], where $\omega < 2.373$ [11]. Our attempts to make this gap significantly smaller, have not been successful yet. However, for realtime deterministic context-free languages and visibly pushdown languages, more efficient dynamic algorithms are possible, as shown in the following two subsections.

4.1 Deterministic context-free languages

Realtime deterministic context-free languages are decided by deterministic PDAs without λ-transitions (RDPDAs).

▶ **Theorem 4.2.** *For each realtime deterministic context-free language L and each $\epsilon > 0$, there is a dynamic constant-time parallel algorithm on a CRCW PRAM for* $\mathrm{CFL}(L)$ *with* $\mathcal{O}(n^{3+\epsilon})$ *work.*

Given an RDPDA $\mathcal{A}$ for L, a configuration $C = (p, u, s)$ consists of a state p, a string u that is supposed to be read by $\mathcal{A}$ and a string s, the initial stack content. We use the following functions δ_{state}, δ_{stack}, and δ_{empty} to describe the behaviour of $\mathcal{A}$ on configurations.

- $\delta_{\text{state}}(C)$ yields the last state of $\mathrm{run}(C)$.
- $\delta_{\text{stack}}(C)$ yields the stack content at the end of $\mathrm{run}(C)$.
- $\delta_{\text{empty}}(C)$ is the position in u, after which $\mathrm{run}(C)$ empties its stack. It is zero, if this does not happen at all.

The algorithm maintains the following information, for each simple configuration $C = (p, u, \tau)$, where $u = w[i, j]$, for some $i \leq j$, for each suffix $v = w[m, n]$ of w, where $j < m$, each state q, and some $k \leq n$.

- $\hat{\delta}(C)$ defined as the tuple $(\delta_{\text{state}}(C), |\delta_{\text{stack}}(C)|, \mathrm{top}_1(\delta_{\text{stack}}(C)), \delta_{\text{empty}}(C),)$, consisting of the state, the height of the stack, the top symbol of the stack, at the end of the run on C and the position where the run ends. If the run empties the stack prematurely or at the end of u, then $\mathrm{top}_1(\delta_{\text{stack}}(C))$ is undefined;
- $\text{push-pos}(C, k)$, defined as the length of the longest prefix x of u, such that $|\delta_{\text{stack}}(p, x, \tau)| = k$. Informally this is the position of u at which the k-th symbol of $\delta_{\text{stack}}(C)$, counted from the bottom, is written;
- $\text{pop-pos}(C, q, v, k)$, defined as the pair (o, r), where o is the length of the prefix v' of v, for which $\mathrm{run}(q, v, \mathrm{top}_k(\delta_{\text{stack}}(C)))$ empties its stack at the last symbol of v', and r is the state it enters.

However, tuples for push-pos and pop-pos are only stored for values k of the form $an^{b\theta}$, for integers $b < \frac{1}{\theta}$ and $a \leq n^{\theta}$, for some fixed $\theta > 0$. A more detailed account is given in the full version of this paper.

4.2 Visibly pushdown languages

Visibly pushdown languages are a subclass of realtime deterministic CFLs. They use *pushdown alphabets* of the form $\tilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_{\text{int}})$ and deterministic PDA that always push a symbol when reading a symbol from Σ_c, pop a symbol when reading a symbol from Σ_r and leave the stack unchanged otherwise. We refer to [2] for more information.

There is a correspondence between wellformed strings over a pushdown alphabet and labelled trees, where each matching pair (a, b) of a call symbol from Σ_c and a return symbol from Σ_r is represented by an inner node with label (a, b) and each other symbol by a leaf. From Theorem 3.7 and this correspondence the following can be concluded.

▶ **Proposition 4.3.** *For each visibly pushdown language L and each $\epsilon > 0$, there is a dynamic constant-time parallel algorithm on a CRCW PRAM for $\text{VPL}^-(L)$ with $\mathcal{O}(n^\epsilon)$ work.*

Here, $\text{VPL}^-(L)$ only allows the following change operations:

- Replacement of a symbol by a symbol of the same type;
- Insertion of an internal symbol from Σ_{int} before a return symbol;
- Replacement of an internal symbol by two symbols ab, where $a \in \Sigma_c$ and $b \in \Sigma_r$.

For arbitrary symbol replacements and insertions, there is a much less work-efficient algorithm which, however, is still considerably more efficient than the algorithm for DCFLs.

▶ **Theorem 4.4.** *For each visibly pushdown language L and each $\epsilon > 0$, there is a dynamic constant-time parallel algorithm on a CRCW PRAM for $\text{VPL}(L)$ with $\mathcal{O}(n^{2+\epsilon})$ work.*

The work improvement mainly relies on the fact that how the height of the stack evolves during a computation only depends on the types of symbols.

5 Conclusion

We have shown that the good work bounds for regular string languages from [11] carry over to regular tree languages, even under some structural changes of the tree. In turn they also hold for visibly pushdown languages under limited change operations. For realtime deterministic context-free languages and visibly pushdown languages under more general change operations better work bounds than for context-free languages could be shown.

There are plenty of questions for further research, including the following: are there other relevant change operations for trees that can be handled with work $\mathcal{O}(n^\epsilon)$? What are good bounds for further operations? Can the bounds for context-free languages be improved? Can the $\mathcal{O}(n^{3+\epsilon})$ be shown for arbitrary (not necessarily realtime) DCFLs? And the most challenging: are there further lower bound results that complement our upper bounds?

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the current clique algorithms are optimal, so is valiant's parser. *SIAM J. Comput.*, 47(6):2527–2555, 2018.

2 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 202–211, 2004. doi:10.1145/1007352.1007390.

3 Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory (ICDT)*, volume 98 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICDT.2018.5.

4 Mikołaj Bojańczyk. Algorithms for regular languages that use algebra. *SIGMOD Rec.*, 41(2):5–14, 2012. doi:10.1145/2350036.2350038.

5 Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. hal-03367725, 2008.

6 Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997. doi:10.1145/256303.256309.

7 Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19, 2012. doi:10.1145/2287718.2287719.

8 Neil Immerman. *Descriptive complexity*. Springer Science & Business Media, 2012.

9 Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

10 Sushant Patnaik and Neil Immerman. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.

11 Jonas Schmidt, Thomas Schwentick, Till Tantau, Nils Vortmeier, and Thomas Zeume. Work-sensitive dynamic complexity of formal languages. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures – 24th International Conference (FOSSACS)*, pages 490–509, 2021. doi:10.1007/978-3-030-71995-1_25.

12 Felix Tschirbs, Nils Vortmeier, and Thomas Zeume. Dynamic complexity of regular languages: Big changes, small work. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL)*, pages 35:1–35:19, 2023. doi:10.4230/LIPIcs.CSL.2023.35.

13 Peter van Emde Boas. Machine models and simulation. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1990.

14 Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference (STOC)*, pages 887–898, 2012. doi:10.1145/2213977.2214056.