Assignment of master's thesis

Title:	Paralelní běh k-lokálních stromových automatů na GPU
Student:	Bc. Milan Borový
Supervisor:	Ing. Štěpán Plachý
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Explore frameworks and libraries for parallelization on GPU.
Adapt the PRAM algorithm for a parallel run of k-local finite tree automata [1] to run on GPU.
Implement your algorithm, test your implementation and compare the speed of the calculation with previously implemented algorithms for the problem.

[1] Plachý Š., Janoušek J. (2020) On Synchronizing Tree Automata and Their Work–Optimal Parallel Run, Usable for Parallel Tree Pattern Matching. In: Chatzigeorgiou A. et al. (eds) SOFSEM 2020: Theory and Practice of Computer Science. SOFSEM 2020. Lecture Notes in Computer Science, vol 12011. Springer, Cham. https://doi.org/10.1007/978-3-030-38919-2_47

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 5 March 2023 in Prague.

Master's thesis

Parallel run of k-local tree automata on GPUs

Bc. Milan Borový

Department of Theoretical Computer Science

Supervisor: Ing. Štěpán Plachý

May 8, 2024

Acknowledgements

I would like to thank my family for the tremendous support during my studies.

I would also like to thank my thesis supervisor Ing. Štěpán Plachý for all the help and patience during the creation of this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 8, 2024 . . . .

Czech Technical University in Prague

Faculty of Information Technology

© 2024 Milan Borový. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Borový, Milan. *Implementation of parallel algorithm for run of k-local tree automata.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

This thesis implements tree pattern matching using deterministic finite tree automata. A work-optimal parallel algorithm exists and has been implemented for EREW PRAM. This algorithm will be optimized for APRAM and ported to SIMT. Then, both new implementations will be measured experimentally and compared.

Keywords k-locality, deterministic finite tree automaton, parallel run implementation, APRAM, SIMT, CUDA, GPGPU

Abstrakt

Tato práce implementuje vyhledávání vzorøu ve stromech za použití deterministických konečných stromových automatøu. Existuje popis a implementace pracovně optimálního algoritmu pro EREW PRAM. Tento algoritmus bude optimalizován pro APRAM a portován na SIMT. Poté budou tyto dvě nové implementace experimentálně změřeny a porovnány.

Klíčová slova k-lokálnost, deterministický konečný stromový automat, implementace paralelního běhu, APRAM, SIMT, CUDA, GPGPU

Contents

Introduction **1**
Goals 1

1 Theory **3**
1.1 Basic definitions 3
1.1.1 Graph 3
1.1.2 Tree 5
1.1.3 Tree language 7
1.1.4 Tree automaton 9
1.1.5 k-local tree automaton 10
1.2 Algorithm Complexity 10
1.2.1 Sequential Complexity 10
1.2.2 Parallel Complexity 11
1.3 Parallel Computation Models 12
1.4 Reduction and Scan 15
1.5 Lists 16
1.6 Euler Tour Technique 17
1.7 Parentheses Matching 19

2 Analysis and Design **21**
2.1 Structures 21
2.1.1 Array 21
2.1.1.1 CPU 22
2.1.1.2 GPU 22
2.1.2 Tree 22
2.1.2.1 CPU 22
2.1.2.2 GPU 23
2.1.3 Arc 23
2.1.4 DFTA 23

2.1.4.1 CPU . . . 23
2.1.4.2 GPU . . . 24
2.2 Reduction and Scan . . . 24
2.2.1 Reduction . . . 24
2.2.1.1 Algorithms . . . 24
2.2.1.2 Implementations . . . 26
2.2.2 Inclusive scan . . . 27
2.2.2.1 Algorithms . . . 27
2.2.2.2 Implementations . . . 33
2.2.3 Exclusive scan . . . 33
2.2.3.1 Implementations . . . 33
2.3 Lists . . . 33
2.3.1 Linked list . . . 33
2.3.1.1 Implementations . . . 34
2.3.2 List ranking . . . 34
2.3.2.1 6-coloring . . . 38
2.3.2.2 3-coloring . . . 39
2.3.2.3 Work-optimal list ranking . . . 39
2.3.2.4 Implementations . . . 41
2.4 Euler Tour Technique . . . 42
2.4.1 Algorithms . . . 42
2.4.2 Implementations . . . 44
2.4.3 Applications . . . 44
2.5 Parentheses matching . . . 45
2.5.1 Algorithms . . . 45
2.5.2 Implementations . . . 49
2.6 DFTA run . . . 49
2.6.1 The main algorithm . . . 50
2.6.2 Depth-mod-k sort . . . 51
2.6.3 Step computation . . . 54
2.6.4 State computation . . . 55
2.6.5 Complexity analysis . . . 55
2.6.6 APRAM modification . . . 56
2.6.7 GPGPU modification . . . 57
2.6.8 Implementations . . . 57

3 Implementation **59**
3.1 Libraries . . . 59
3.1.1 CPU . . . 59
3.1.2 GPU . . . 60
3.2 Structures . . . 60
3.2.1 Array . . . 60
3.2.1.1 CPU . . . 60
3.2.1.2 GPU . . . 60

3.2.2 Tree . . . 61
3.2.2.1 CPU . . . 61
3.2.2.2 GPU . . . 61
3.2.3 Arc . . . 61
3.2.3.1 CPU . . . 61
3.2.3.2 GPU . . . 61
3.2.4 DFTA . . . 61
3.2.4.1 CPU . . . 61
3.2.4.2 GPU . . . 62
3.3 Reduction and Scan . . . 62
3.3.1 Reduction . . . 62
3.3.1.1 CPU . . . 62
3.3.1.2 GPU . . . 62
3.3.2 Inclusive scan . . . 62
3.3.2.1 CPU . . . 62
3.3.2.2 GPU . . . 62
3.3.3 Exclusive scan . . . 63
3.3.3.1 CPU . . . 63
3.3.3.2 GPU . . . 63
3.4 Lists . . . 63
3.4.1 Linked list . . . 63
3.4.1.1 CPU . . . 63
3.4.1.2 GPU . . . 63
3.4.2 List ranking . . . 63
3.4.2.1 CPU . . . 63
3.4.2.2 GPU . . . 64
3.5 Euler Tour Technique . . . 64
3.5.1 CPU . . . 64
3.5.2 GPU . . . 64
3.6 Parentheses matching . . . 65
3.6.1 CPU . . . 65
3.6.2 GPU . . . 65
3.7 Run of k-local DFTA . . . 65
3.7.1 Run . . . 65
3.7.1.1 CPU . . . 65
3.7.1.2 GPU . . . 65
3.7.2 Preprocess . . . 66
3.7.2.1 CPU . . . 66
3.7.2.2 GPU . . . 66

4 Testing **67**
4.1 System test . . . 67
4.2 Time measurements . . . 68
4.2.1 Methodology . . . 68

4.2.2 Hardware . . . 68
4.2.3 Data . . . 69
4.2.4 Results . . . 69

Conclusions and Future work **73**
Future work . . . 73

Bibliography **75**

A Acronyms **77**

B Symbols **79**

C User manual **81**
C.1 Prerequisities . . . 81
C.1.1 CPU . . . 81
C.1.2 GPU . . . 81
C.2 Compilation . . . 81
C.3 Usage . . . 82

D Contents of the enclosed medium **83**

List of Figures

2.1 Parallel reduction computation 25
2.2 Hillis-Steele algorithm for input of size 16 28
2.3 Up-Sweep step for the input of the size 8 30
2.4 Down-Sweep step for the input of the size 8 30
2.5 Linked structure 34
2.6 Successor array 34
2.7 Pointer jumping 35
2.8 (a) Euler circuit of the tree (b) Array representation of the arcs . . 43

4.1 Example pattern 67
4.2 Evaluated example tree 68
4.3 Results 69
4.4 Time comparisons 70

Introduction

Even with the rapid development of processor operation speed, overcoming issues with big data computations is impossible and not feasible. The solution is parallelization, as potentially slower processors (and cheaper processors) may achieve a greater speed by splitting the work needed.

One of the problems that may greatly benefit from that is tree pattern matching. Linear time complexity is feasible only in the case of relatively small trees. When enough processors are supplied, the computation time can be reduced substantially.

Patterns of height k can be represented by a k-local deterministic finite tree automaton. Thus, running such an automaton for some trees can be used to match patterns in trees. The work-optimal algorithm for that problem was presented described[1] and implemented[2]. The original algorithm was designed for EREW PRAM architecture, which is not usual in its pure form.

Instead, modern CPUs are close to APRAM architecture, which is more relaxed than EREW PRAM, offering possible optimizations. But, for mass parallelization, GPUs seem more suitable. Modern GPUs are of the SIMT architecture, which is similar to PRAM in some regards but has its own challenges.

Goals

There are several goals for this thesis.

The first one is to analyze the algorithm in general. Including possibilities to implement it.

The second one is to implement a run of the k-local DFTA for CPUs and apply possible optimizations enabled by the APRAM architecture.

The third one is to implement the run for GPUs, adapting the algorithm to the new architecture.

The fourth one is to compare mass-parallelized GPU implementation with the CPU implementation.

Chapter 1

Theory

In this Chapter, all the needed theory will be presented. Starting with basics of graph theory[3], tree languages[1] and algorithm complexity[4] through computation models[4] to the definition of individual problems that are needed to be solved to run k-local DFTA in parallel on APRAM and SIMT architectures.

All problems defined in this chapter will be analyzed in Chapter 2.

Notation in this chapter will be similar to the notation in [4] and [1] for tree languages.

1.1 Basic definitions

1.1.1 Graph

Definition 1.1 Graph *is a pair* $G = (V, E)$*, where*

- V *is a set of elements called* vertices *or* nodes*,*
- $E \subseteq \{\{u, v\} : u, v \in V \wedge u \neq v\}$ *is a set of* edges*.*

Definition 1.2 Induced subgraph $G' = (V', E')$ *of the graph* $G = (V, E)$ *is a graph such that* $V' \subseteq V$ *and*

$$\forall u, v \in V' : \ \{u, v\} \in E' \Leftrightarrow \{u, v\} \in E$$

Definition 1.3 *Let $G = (V, E)$ be a graph. The degree of a vertex $u \in V$ is*

$$deg(u) := |\{\{u, v\} : \ \{u, v\} \in E\}| .$$

Definition 1.4 Path $x_1 - x_n$ *in the graph $G = (V, E)$ is an nonempty sequence of vertices $x_1 \dots x_n$, where*

- $\forall i \leq n : \ x_i \in V$,
- $\forall i < n : \ \{x_x`i, x_{i+1}\} \in E$,
- $\forall i, j \leq n : \ i = j \vee x_i \neq x_j$.

Definition 1.5 Length *of the path $x_1 - x_n$ is*

$$length\ of\ x_1 - x_n = n - 1.$$

Definition 1.6 *A graph in which a path exists for each pair of vertices $u, v \in V, u \neq v$ is called* connected.

Definition 1.7 Cycle *in the graph $G = (V, E)$ is the sequence of vertices $x_1 \dots x_n$, where*

- $n \geq 3$,
- $\forall i \leq n : \ x_i \in V$,
- $\forall i < n : \ \{x_i, x_{i+1}\} \in E$,
- $\forall i, j < n : \ i = j \vee x_i \neq x_j$,
- $\{x_1, x_n\} \in E$.

Definition 1.8 *def:agraph A graph in which no cycle exists is called* acyclic.

Definition 1.9 Directed graph *is a pair $DG = (V, E)$, where*

- *V is a set of elements called* vertices *or* nodes,
- *$E \subseteq \{(u, v) : u, v \in V \wedge u \neq v\}$ is a set of* directed edges.

Definition 1.10 *Let $DG = (V, E)$ be a directed graph. The in-degree of a vertex $u \in V$ is*

$$deg_{in}(u) := |\{(v, u) : \ (v, u) \in E\}| \, .$$

The out-degree of a vertex $u \in V$ is

$$deg_{out}(u) := |\{(u, v) : \ (u, v) \in E\}| \, .$$

Definition 1.11 *The* directed path $x_1 - x_n$ *in the directed graph $DG = (V, E)$ is an unempty sequence of vertices $x_1 \dots x_n$, where*

- $\forall i \leq n : \ x_i \in V$,
- $\forall i < n : \ (x_i, x_{i+1}) \in E$,
- $\forall i, j \leq n : \ i = j \vee x_i \neq x_j$.

Definition 1.12 Length *of the directed path $x_1 - x_n$ is*

$$length\ of\ x_1 - x_n = n - 1.$$

Definition 1.13 *A directed graph with a directed path for each pair of vertices $u, v \in V, u \neq v$ is called* strongly connected*.*

Definition 1.14 *If the directed graph is not strongly connected, but an undirected path exists for each pair of vertices, the graph is called* weakly connected*.*

1.1.2 Tree

Definition 1.15 Tree *is an acyclic and connected graph.*

Definition 1.16 Rooted tree *is a tree where one vertex has been designated as the* root*.*

Definition 1.17 Depth *of the vertex u in a tree with the root r is length of the path $r - u$ and is denoted $depth(u)$.*

Definition 1.18 *Let $G = (V, E)$ be a tree and u, v any two vertices of G such that $\{u, v\} \in E$ and $depth(u) < depth(v)$. Then u is called* parent *of v denoted by $parent(v)$, and v is called* child *of u. The set of children of u is denoted by $children(u)$.*

Definition 1.19 *Let $G = (V, E)$ be a tree and u any vertex of that tree. Then $\forall v, w \in children(u), v \neq w : v$ is called* sibling *of w. The set of siblings of v is denoted by $siblings(v)$.*

Definition 1.20 *Let $G = (V, E)$ be a tree and $u, v \in V$ two vertices of that tree. Then u is called an* ancestor *of v if*

- *$u = parent(v)$, or*
- *$\exists w \in V$ such that u is an ancestor of w and w is an ancestor of the vertex v.*

The set of ancestors of v is denoted by $ancestors(v)$.

Definition 1.21 *Let $G = (V, E)$ be a tree and $u, v \in V$ two vertices of that tree such that $u \in ancestors(v)$. Then v is called* descendant *of u. The set of descendants of the vertex u is denoted by $descendants(u)$.*

Definition 1.22 *Tree G' that is an induced subgraph of another tree G is called its* subtree.

Definition 1.23 *Let $G = (V, E)$ be a tree and $u \in V$ a vertex of that tree. The* arity *of u is*

$$arity(u) = |children(u)| .$$

Definition 1.24 *Let $G = (V, E)$ be a tree and $u \in V$ vertex of that tree. u is called a* leaf *iff $arity(u) = 0$. u is called an* inner vertex *iff it's not a leaf. The set of leaves is denoted by $leaves(G)$.*

Definition 1.25 Ordered tree *$G = (V, E)$ is a tree such that $\forall u \in V :$ $children(u)$ is an ordered set.*

Definition 1.26 *Let $G = (V, E)$ be an ordered tree and $u, v \in V$ vertices of that tree such that $v \in children(u)$. v is called* i-th child *of vertex u if v is on the i-th position of the ordered set $children(u)$ denoted by $child_i(u)$.*

1.1.3 Tree language

Definition 1.27 Alphabet Σ *is a finite set of symbols.*

Definition 1.28 String w *over an alphabet Σ is a sequence of symbols $w_1 \dots w_n$ from the alphabet Σ.*

ϵ denotes an empty string.

$|w| = |w_1 \dots w_n| = n$ denotes length of string w.

The set of all strings over an alphabet Σ is denoted by Σ^*.

$|w|_a$ denotes the number of occurrences of the symbol $a \in \Sigma$ in the string $w \in \Sigma^*$.

w_i denotes the i-th symbol of the string $w \in \Sigma^*$.

Definition 1.29 Concatenation *of strings over an alphabet Σ is mapping $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that*

$$a_1 \dots a_n \ \cdot \ b_1 \dots b_n = a_1 \dots a_n b_1 \dots b_n.$$

Definition 1.30 Substring *of string w over the alphabet Σ is such string s that*

$$\exists t, u \in \Sigma^* : \ w = t \cdot s \cdot u.$$

Definition 1.31 Prefix *of string w over the alphabet Σ is such string p that*

$$\exists t \in \Sigma^* : \ w = p \cdot t.$$

Definition 1.32 Suffix *of string w over the alphabet Σ is such string s that*

$$\exists t \in \Sigma^* : \ w = t \cdot s.$$

Definition 1.33 Ranked alphabet *is a pair* $\mathcal{F} = (\Sigma, rank)$, *where*

- Σ *is an alphabet,*
- $rank$ *is a function* $\Sigma \to \mathbb{N}_0$ *assigning a natural number to each symbol of the alphabet* Σ.

$\mathcal{F}_r = \{a : \ a \in \Sigma \wedge rank(a) = r\}$ *denotes subset of symbols with rank* r.

Definition 1.34 *The set of terms* $T(\mathcal{F}, \mathcal{X})$ *over the ranked alphabet* $\mathcal{F}$ *and the set of constants* $\mathcal{X}$ *called* variables, *where* $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$, *is the smallest set satisfying*

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \mathcal{X})$,
- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$,
- $\forall r \geq 1, \forall f \in \mathcal{F}_r : \ t_1, \ldots, t_r \in T \Rightarrow f(t_1, \ldots, t_r) \in T$.

Each $t \in T$ *is called a* term *over the ranked alphabet* $\mathcal{F}$.

Definition 1.35 *The term* $t \in T(\mathcal{F}, \mathcal{X})$ *where* $\mathcal{X} = \emptyset$ *is called a* ground term *over the ranked alphabet* $\mathcal{F}$. *The set of ground terms over the ranked alphabet* $\mathcal{F}$ *is denoted by* $T(\mathcal{F})$.

Theorem 1.1 *Let* $t = f(t_1, \ldots, t_r) \in T(\mathcal{F}, \mathcal{X})$ *be a term over the ranked alphabed* $\mathcal{F}$. *Then there exists an equivalent tree* $G = (V, E)$ *such that*

- $V = \{node(t)\} \cup (\bigcup_{i=1}^{r} V'_i)$,
- $E = \{node(t), root(G'_i) : \ \forall i \in \hat{r}\} \cup (\bigcup_{i=1}^{r} E'_i)$,
- $node(t)$ *denotes node representing term* t,
- $label(root(G))$ *denotes symbol* f,
- $G'_i = (V'_i, E'_i)$ *is an equivalent tree of the term* t_i.

Such tree can be used to represent t.

Definition 1.36 Ground substition σ *over the set of the variables* $\mathcal{X}$ *and the ranked alphabet* $\mathcal{F}$ *is a mapping* $\mathcal{X} \to T(\mathcal{F})$ *assigning a ground term* $t \in T(\mathcal{F})$ *to each variable* $x \in \mathcal{X}$*.*

Definition 1.37 Subterm $t|_p$ *of the term* $t = f(t_1, \ldots, t_r) \in T(\mathcal{F}, \mathcal{X})$ *at the position* $p \in \mathbb{N}_0^*$ *is*

- t*, iff* $p = \epsilon$*,*
- $t_i|_{p'}$*, iff* $p = ip'$*.*

The set of subterms of the term t *is denoted by* $subterms(t)$*.*

Definition 1.38 Tree language *over the ranked alphabet* $\mathcal{F}$ *is a set of ground terms* $L \subseteq T(\mathcal{F})$*.*

1.1.4 Tree automaton

Definition 1.39 Deterministic finite tree automaton *(DFTA) over the ranked alphabet* $\mathcal{F}$ *is a quadruple* $A = (Q, \mathcal{F}, Q_f, \Delta)$ *such that*

- Q *is a finite set of states,*
- $\mathcal{F}$ *is a ranked alphabet,*
- $Q_f \subseteq Q$ *is a set of final states,*
- $\Delta = \bigcup_r \Delta_r : \mathcal{F}_r \times Q^r \to Q$ *is a transition function.*

Definition 1.40 Extended transition function *of the DFTA* $A = (Q, \mathcal{F}, Q_f, \Delta)$ *is a mapping* $\widehat{\Delta} : T(\mathcal{F}) \to Q$ *such that*

- $\forall f \in \mathcal{F}_0 : \widehat{\Delta}(f) = \Delta(f)$
- $\forall r \geq 1, \forall f \in \mathcal{F}_r, \forall t_1, \ldots, t_r \in T(\mathcal{F}) : \widehat{\Delta}(f(t_1, \ldots, t_r)) = \Delta(f(\widehat{\Delta}(t_1), \ldots, \widehat{\Delta}(t_r)))$

Definition 1.41 *A ground term* $t \in T(\mathcal{F})$ *is* accepted *by the DFTA* $A = (Q, \mathcal{F}, Q_f, \Delta)$ *iff* $\widehat{\Delta}(t) \in Q_f$*.*

1.1.5 k-local tree automaton

Definition 1.42 *Let $A = (Q, \mathcal{F}, Q_f, \Delta)$ be a DFTA and $t \in T(\mathcal{F}, \mathcal{X})$ be a term over the ranked alphabet $\mathcal{F}$. Term t is called* synchronizing *for A iff*

$$\exists q \in Q, \forall \sigma : \ \widehat{\Delta}(\sigma(t)) = q.$$

Definition 1.43 Minimal variable depth *is a function $MVD : \ T(\mathcal{F}, \mathcal{X}) \ \rightarrow \ \mathbb{N}_0$ such that*

- $\forall f \in \mathcal{F}_0 : \ MVD(f) = +\infty,$
- $\forall x \in \mathcal{X} : \ MVD(x) = 0,$
- $\forall p > 0, \forall f \in \mathcal{F}_r, \forall t_1, \ldots, t_r \in T(\mathcal{F}, \mathcal{X}) :$
  $MVD(f(t_1, \ldots, t_r)) = 1 + min_{i=1}^{r} MVD(t_i).$

Definition 1.44 k-local DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ *is a DFTA such that*

$$\forall t \in T(\mathcal{F}, \mathcal{X}) : \ MVD(t) \geq k \Rightarrow t \textit{ is synchronizing}.$$

1.2 Algorithm Complexity

1.2.1 Sequential Complexity

Definition 1.45 Time complexity $T_A^K(n)$ *of the algorithm A solving the problem K for an input of the size n is a computer time required to run that algorithm.*

Definition 1.46 Sequential lower bound $SL^K(n)$ *of the problem K is a function such that*

$$\forall A : \ T_A^K(n) \in \Omega\left(SL^K(n)\right).$$

Definition 1.47 Best known *sequential algorithm solving the problem K is such algorithm A that there's no known algorithm B such that*

$$T_A^K(n) \in \omega\left(T_B^K(n)\right).$$

Definition 1.48 Sequential upper bound $SU^K(n)$ *of the problem* K *is the worst-case time complexity of the best known sequential algorithm solving* K*.*

Definition 1.49 Optimal *sequential algorithm solving the problem* K *is such algorithm* A *that*

$$T_A^K(n) \in \Theta\left(SL^K(n)\right).$$

1.2.2 Parallel Complexity

Definition 1.50 Parallel time complexity $T_A^K(n,p)$ *of the parallel algorithm* A *solving the problem* K *for an input of then size* n *using* p *processors is a total time elapsed from the beginning of execution until the last processor finishes.*

Definition 1.51 Parallel speedup *of the parallel algorithm* A *solving the problem* K *for an input of the size* n *using* p *processors is*

$$S_A^K(n,p) = \frac{SU^K(n)}{T_A^K(n,p)}.$$

Definition 1.52 Parallel cost *of the algorithm* A *solving the problem* K *for an input of the size* n *using* p *processors is*

$$C_A^K(n,p) = p \cdot T_A^K(n,p).$$

Definition 1.53 Cost-optimal *algorithm* A *is such algorithm that*

$$C_A^K(n,p) \in \Theta\left(SU^K(n)\right).$$

Definition 1.54 Synchronous parallel work *of the synchronous algorithm* A *solving the problem* K *for an input of the size* n *using* p *processors in* τ *parallel steps where* p_i *denotes the number of active processors in the step* i *is*

$$W_A^K(n,p) = \sum_{i=1}^{\tau} p_i.$$

Definition 1.55 Asynchronous parallel work *of the asynchronous algorithm A solving problem K for an input of the size n using p processors where T_i denotes the number of steps executed by the i-th processor is*

$$W_A^K(n,p) = \sum_{i=1}^{p} T_i.$$

Definition 1.56 Work-optimal *algorithm A is such algorirthm that*

$$W_A^K(n,p) \in \Theta\left(SU^K(n)\right).$$

Definition 1.57 Parallel efficiency *of the algorithm A solving the problem K for an input of the size n using p processors is*

$$E_A^K(n,p) = \frac{SU^K(n)}{C_A^K(n,p)}.$$

1.3 Parallel Computation Models

Parallel computation models are split into 2 groups.

- *Shared-memory* models where all processors share one common memory.
- *Distributed-memory* models where each processor (or group of processors) has private memories and passes data through messages.

This thesis is focused on shared-memory models. Specifically MIMD APRAM (de facto multi-core CPUs) and SIMD PRAM (de facto many-core GPUs). For more insights, [4] is recommended.

Definition 1.58 Random Access Machine *(RAM) model is a computation model consisting of a single processor with*

- *a bounded number of registers,*
- *an unbounded number of local memory cells with a user-defined program,*
- *a read-only input tape,*
- *a write-only output tape.*

The processor's instruction set contains instructions for simple data manipulation, comparisons, branching, and basic arithmetic operations. The program is executed from the first instruction until the HALT instruction.

Definition 1.59 Parallel Random Access Machine *(PRAM) model is a computation model consisting of multiple RAM processors $p_1, p_2, \ldots$ with no input nor output tape and without a local memory.*

All processors are connected to a shared memory with an unbounded number of cells $M_1, M_2, \ldots$. Each processor p_i knows its index i. Each processor has constant-time access to any M_j unless access conflicts exist.

All processors work synchronously and can communicate with each other only through writing to or reading from the shared memory. p_1 has a control role and starts execution of other processors. p_1 can halt only in case other processors halted as well.

Access conflicts mentioned in the previous definition are handled based on the conflict-handling strategy of specific PRAM submodels.

Definition 1.60 Exclusive Read Exclusive Write *(EREW) PRAM submodel doesn't allow 2 processors to access the same memory cell simultaneously.*

Definition 1.61 Concurrent Read Exclusive Write *(CREW) PRAM submodel allows reading single memory cell from multiple processors simultaneously. Still, only one processor can write a single cell at a time.*

Definition 1.62 Concurrent Read Concurrent Write *(CRCW) PRAM submodel allows multiple processors to read or write a single cell simultaneously.*

The concurrent read operations don't affect each other, but the concurrent write operations may be ambiguous, and thus, their semantics have to be defined.

Definition 1.63 Priority CRCW PRAM *submodel has fixly prioritized processors. Only the processor with the highest priority can complete the write operation in case of conflict.*

Definition 1.64 Arbitrary CRCW PRAM *submodel allows to single randomly chosen processor to complete write operation.*

Definition 1.65 Common CRCW PRAM *submodel allows all processors to complete write operation, but all processors have to write the same value (algorithm responsibility).*

Another way to split computation models is based on the number of instruction streams and input streams being processed simultaneously[5].

- *Single-Instruction* (SI) architectures can process only one instruction at a time.
- *Multiple-Instruction* (MI) architectures can process multiple instructions simultaneously.
- *Single-Data* (SD) architectures can process data from only one input stream at a time.
- *Multiple-Data* (MD) architectures can process data from multiple input streams simultaneously.

Given those SISD, SIMD, MISD and MIMD architectures are defined.

Note 1.1 *Modern multi-core CPUs represent the MIMD APRAM model.*

Modern many-core GPUs represent the SIMD PRAM model.

Even though this model still applies to program design, GPUs are not strictly SIMD PRAM. Threads are split into groups. Threads in a single group operate as a SIMD PRAM (not strictly true), but multiple groups operate as a MIMD APRAM. To avoid confusion by calling modern GPUs SIMD even though they aren't, new term SIMT (single-instruction multiple-threads) was created for them.

1.4 Reduction and Scan

Definition 1.66 *Let* $\mathbb{X} = \{x_1, \ldots, x_n\}$ *be a finite set of values and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.

The problem of finding $\bigoplus_{i=1}^{n} x_i$ *is called a* reduction *and* $\oplus$ *is called a* reduction operator.

Definition 1.67 *Let* $(x_i)_{i=1}^{n}$ *be a finite sequence of values from* $\mathbb{X}$ *and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.

The problem of finding a sequence $(y_i)_{i=1}^{n}$ *such that*

$$\forall i \in \widehat{n} : \; y_i = \bigoplus_{j=1}^{i} x_j$$

is called an inclusive scan.

Definition 1.68 *Let* $(x_i)_{i=1}^{n}$ *be a finite sequence of values from* $\mathbb{X}$ *and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.

The problem of finding a sequence $(y_i)_{i=1}^{n}$ *such that*

$$\forall i \in \widehat{n} : \; y_i = \bigoplus_{j=1}^{i-1} x_j$$

is called an exclusive scan.

1.5 Lists

Definition 1.69 Linked list *is a pair* $L = (X, S)$ *where*

- X *is an unempty set of* nodes*,*
- S *is a* successor *function* $X \to X$ *such that*
  - $\exists! h \in X, \forall x \in X: \; S(x) \neq h$*,*
  - $\exists! t \in X: \; S(t)$ *is undefined,*
  - $S^{|X|-1}(h) = t$*.*

The node h *is called a* head *and is denoted by* $head(L)$*. The node* t *is called a* tail *and is denoted by* $tail(L)$*.*

Definition 1.70 *Let* $L = (X, S)$ *be a linked list.* Independent set $I \subset X$ *of a linked list* L *is such set that*

$$\forall i \in I: \; S(i) \notin I \vee S(i) \; is \; undefined.$$

Lemma 1.1 *Independent set* I *of linked list* L *can be removed from* L *in parallel on an EREW PRAM.*

Proof 1.1 *Let* $L = (X, S)$ *be a linked list and* $I \subset X$ *its independent set.*

Since $\forall i, j \in I: \; S(i) \neq j$*, there are no neighbouring nodes in the independent set* I*. Thus each node can be removed from* L *by re-linking its predecessor to its successor (i.e.* iff $S(i) \in I$ *then* $S(i) \leftarrow S(S(i))$*) without any conflicts.* □

Definition 1.71 *Let* $L = (X, S)$ *be a linked list and* C *a set of* k colors *(*$X \cap C = \emptyset$*). Problem of finding a mapping* $color: \; X \to C$ *such that*

$$\forall x, y \in X: \; S(x) = y \Rightarrow color(x) \neq color(y)$$

is called list k-coloring.

Lemma 1.2 *Let color be a k-coloring of a linked list $L = (X, S)$.*

The set of local minima of the k-coloring

$$\{x : x \in X \wedge (\forall y \in X : \; (S(x) = y \vee S(y) = x) \Rightarrow color(x) < color(y))\}$$

is an independent set of the linked list L of a size $O\left(\frac{n}{k}\right)$.

Proof 1.2 *Let x, y be 2 local minima of a k-colouring color of a linked list $L = (X, S)$ with no other local minima in between.*

As local minima is defined to be strictly smaller than its neighbours, there can't be 2 neighboring local minima.

Since there are no local minima between x and y, colours of nodes between x and y must form a bitonic sequence[1] that has at most $2k - 3$ colours. Thus the size of the set of the local minima is $O\left(\frac{n}{2 \cdot k - 2}\right) \subseteq O\left(\frac{n}{k}\right)$. □

Definition 1.72 *Let $L = (X, S)$ be a linked list. Problem of finding a mapping $rank : X \to \mathbb{N}_0$ such that*

$$\forall x \in X : \; S^{rank(x)}(head(L)) = x$$

is called list ranking.

1.6 Euler Tour Technique

Definition 1.73 Euler tour *of a graph $G = (V, E)$ is a sequence of consecutive edges in the graph G that traverses every edge in E exactly once.*

Graph G that contains Euler tour is called Euler *graph.*

Theorem 1.2 *(Euler's theorem[3]) A connected graph $G = (V, E)$ is Euler iff*

$$\forall u \in V : \; deg(u) \text{ is even.}$$

[1] sequence $(a)_1^n$ such that $\exists k : 1 < k < n$ for which $(a)_1^k$ is monotonic increasing and $(a)_k^n$ is monotonic decreasing or vice versa.

Theorem 1.3 *A connected oriented graph $G = (V, E)$ is Euler iff*

$$\forall u \in V : \; deg_{in}(u) = deg_{out}(u).$$

Theorem 1.4 *Let $G = (V, E)$ be a tree. An oriented graph $G' = (V, E')$ such that*

$$\forall u, v \in V : ((u, v) \in E' \wedge (v, u) \in E') \Leftrightarrow \{u, v\} \in E$$

is an oriented Euler graph.

Proof 1.3 *Since $G = (V, E)$ is connected and each edge in E was replaced with pair of edges in both directions, $G' = (V, E')$ must be strongly connected and*

$$\forall u \in V : \; deg(u) = deg'_{in}(u) = deg'_{out}(u).$$

Hence G' is an oriented Euler graph. □

Definition 1.74 Euler tour technique *is a problem of finding an Euler tour of an ordered tree.*

1.7 Parentheses Matching

Definition 1.75 *String of parentheses* $w \in \{(,)\}^*$ *is* well-formed *when*

- $w = (\ )$*, or*
- $w = u \cdot v$*, where* u, v *are well-formed, or*
- $w = (\ v\ )$*, where* v *is well-formed.*

Definition 1.76 *Let* $w \in \{(,)\}^*$ *be a well-formed string of parentheses.*

Problem of finding a mapping $match : \mathbb{N} \rightarrow \mathbb{N}_0$ *such that*

$$\begin{array}{rrl} \forall i, j \in \widehat{|w|)} : & & match(i) = j \\ & \Leftrightarrow & match(j) = i \\ & \Leftrightarrow & w_{min\{i,j\}} \ \ldots \ w_{max\{i,j\}} \ is\ well-formed \end{array}$$

is called parentheses matching.

Chapter **2**

Analysis and Design

In this chapter, the data structures used will be designed first. Secondly, algorithms solving problems defined in the previous Chapter will be analyzed. All those algorithms are required for the main algorithm.

Next, the main algorithm and possible optimizations for APRAM will be analyzed. The last part of this Chapter will analyze the main algorithm modification for SIMT architecture.

For the sake of simplicity, all the algorithms present in this chapter assume that the size of the input is a power of two, there are always enough processors, and all processors are synchronous (PRAM model).

2.1 Structures

2.1.1 Array

An array is a contiguous block of memory providing random access to the individual elements. Arrays are a core feature of C++. Stack-based arrays would require limiting the size of trees and DFTAs to some arbitrary number, which is unfavorable. Heap-based arrays have no limit, but they have to be managed manually.

STL includes auto-managed variants of those. *std::array* for stack-based array and *std::vector* for heap-based array. Even though they're implemented efficiently, for the purpose of this thesis, they're slow.

The first problem is that they can't be allocated without initialization the

same way as C-like arrays. The second problem is initialization/copy, which is implemented sequentially, slowing down the parallel applications.

2.1.1.1 CPU

A custom container similar to *std::vector* will be implemented to overcome those issues. This container will have a minimal subset of capabilities required for this thesis.

2.1.1.2 GPU

As *thrust* (CUDA) will be used for SIMT variant, *thrust::host_vector* and *thrust::device_vector* may be used to represent array. Those are designed specifically to be used in GPGPU and handle data transfer between CPU and GPU.

2.1.2 Tree

Tree will represent term $t \in T(\mathcal{F}, \mathcal{X})$ as described in theorem 1.1 and could be represented as a pair of arrays:

- *symbols* of ranked alphabet $\mathcal{F}$ for each vertex,
- *children* pointing to children of each vertex,

and a pointer to the root of the tree.

As most tree implementations available are usually based on linked structures rather then contiguous memory blocks, custom tree will have to be implemented as well.

2.1.2.1 CPU

children array can be represented as a 2-dimensional array (i.e., array of arrays). The first dimension is the vertex, which owns its children's list. The second dimension is individual children.

2.1.2.2 GPU

As 2-dimensional arrays are impractical for GPGPU as they may present an unnecessary level of indirection, *children array* should rather be linearized, requiring an additional array specifying *first* index of children list for each vertex.

2.1.3 Arc

Arc of the Euler tour of the tree is a 4-tuple consisting of:

- pointer to *source vertex*,
- pointer to *target vertex*,
- pointer to *opposite arc*,
- and a *type* of the arc.

2.1.4 DFTA

A DFTA is defined (see definition 1.39) as a 4-tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$. As Δ has to contain all symbols from $\mathcal{F}$ with correct arity, there's no need to explicitly specify ranked alphabet. Thus, the DFTA may be a 3-tuple consisting of

- a set of states, represented by n assuming states are numbers $0, \dots, n$,
- a set of final states, represented by an actual set,
- and a transition function.

The transition function can be represented as an array associating states to a new state.

2.1.4.1 CPU

The set of final states may be represented by *std::unordered_set*. It has similar issues as *std::vector* regarding multi-threading, but as those sets will be small, this is negligible.

For ease of use, the transition function may be a 2-dimensional array, with the first dimension associating the symbol of the alphabet to a transition function for that symbol. The second dimension associating the states of the children to a new state.

2.1.4.2 GPU

As the *std::unordered_set* cannot be used in GPGPU, an array of booleans should be used instead, denoting whether the set is final for each vertex.

As with arrays (2.1.1), the transition function array will have to be linearized, introducing an array for *first* indices.

2.2 Reduction and Scan

2.2.1 Reduction

The reduction is defined in definition 1.66.

2.2.1.1 Algorithms

As the operator $\oplus$ is associative, the reduction can be performed in arbitrary order.

For the sequential solution, linear order can be used, i.e.

$$((((\dots(((x_1 \oplus x_2) \oplus x_3) \oplus x_4) \oplus \dots \oplus x_{n-3}) \oplus x_{n-2}) \oplus x_{n-1}) \oplus x_n)\,.$$

Each value is then consecutively accumulated to an accumulator. Such accumulator can be initialized to a left-identity $\mathbb{0}$ with respect to $\oplus$. Hence, the algorithm 1.

As the whole input is iterated once the time complexity is

$$T(n) = O(n)\,.$$

For the same reason, it's impossible to do it any faster as the input must be read at least once.

$$SL(n) = SU(n) = O(n)$$

Algorithm 1: Sequential reduction

```
Input: values x_1, ..., x_n
Result: reduction of values
r ← 𝟘;
for i ← 1 to n do
  | r ← r ⊕ x_i;
end
return r;
```

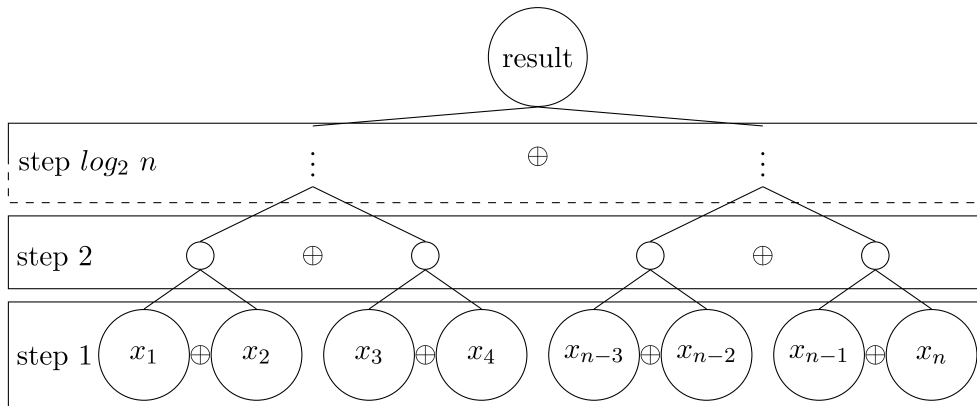

Figure 2.1: Parallel reduction computation

For the parallel solution, tree-like order is more beneficial, i.e.

$$(\dots((x_1 \oplus x_2) \oplus (x_3 \oplus x_4)) \oplus \dots \oplus ((x_{n-3} \oplus x_{n-2}) \oplus (x_{n-1} \oplus x_n)) \dots) .$$

Reduction is performed in several steps. Each step consists of a computation of results for every application of $\oplus$ that hasn't been computed yet and does not require any uncomputed intermediate result. Every computation done in a single step is performed by a different processor. Figure 2.1 depicts parallel reduction.

This is formulated in algorithm 2.

The inner loop has $O(n)$ iterations split among p processors. The outer loop has $O(log_2\ n)$ iterations performed sequentially. This combines to

Algorithm 2: Parallel reduction (EREW PRAM)

Input: values $x_1, \ldots, x_n$
Result: reduction of values
Auxiliary: intermediate results $r_1, \ldots, r_{\frac{n}{2}}$, $\forall i \in \widehat{\frac{n}{2}}$: indices $left_i, right_i$

```
for i ← 1 to log2 n do
    for j ← 1 to n/2^i do in parallel
        left_j ← 1 + (j − 1) · 2^i;
        right_j ← left + 2^(i−1);
        r_left_j ← r_left_j ⊕ r_right_j;
    end
end
return r_1;
```

$$
\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) \cdot O(log_2\ n) = O\left(\frac{n \cdot log_2\ n}{p}\right), \\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right), \\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n) = \omega(SU(n)), \\
W(n,p) =& \sum_{i=1}^{log_2\ n} \frac{n}{2^i} = n \cdot \sum_{i=1}^{log_2\ n} \frac{1}{2^i} = n \cdot 1 = O(n) = \Theta(SU(n)).
\end{aligned}
$$

As the number of processors is halved each step, this algorithm is not cost-optimal. No read/write conflicts exist, so all complexities apply to EREW PRAM.

2.2.1.2 Implementations

C++17 numeric library provides *std::reduce*, which implements both sequential and parallel reduction with the same complexity as described above. It provides a convenient and flexible interface supporting containers but lacks control over the number of threads used.

OpenMP provides its own reduction implementation using pragma directives. This provides full control over the number of threads but lacks the flexibility of STL.

For GPU implementation, *thrust::reduce* can be used, similar to *std::reduce* but designed for GPU computation.

2.2.2 Inclusive scan

The inclusive scan is defined in definition 1.67.

The inclusive scan is the same as reduction, but all intermediate results of linear order (see 2.2.1) must be returned.

2.2.2.1 Algorithms

The sequential solution is a modification of sequential reduction, storing all intermediate results to output.

Algorithm 3: Sequential inclusive scan

Input: values $x_1, \ldots, x_n$
Output: inclusive scan $s_1, \ldots, s_n$
$r \leftarrow \mathbb{0}$;
for $i \leftarrow 1$ **to** n **do**
  $r \leftarrow r \oplus x_i$;
  $s_i \leftarrow r$;
end

Complexities and bounds are the same as in the case of reduction.

As linear order results are required, but the parallel reduction has a tree-like order, a similar modification cannot be used for the parallel inclusive scan.

There are two dominant algorithms for scans.

The first one was presented by Hillis & Steele[6] and is depicted in figure 2.2.

The algorithm 4 is divided into several steps. In each step $\oplus$ is applied to values with distance 2^{step-1}.

The auxiliary array is necessary for EREW as without it, there will be read/write conflicts, resulting in the algorithm being CREW.

The copy has complexity $O\left(\frac{n}{p}\right)$. The inner loop has $O(n)$ steps split among p processors. The outer loop has $O(log_2\ n)$ steps performed sequentially. This

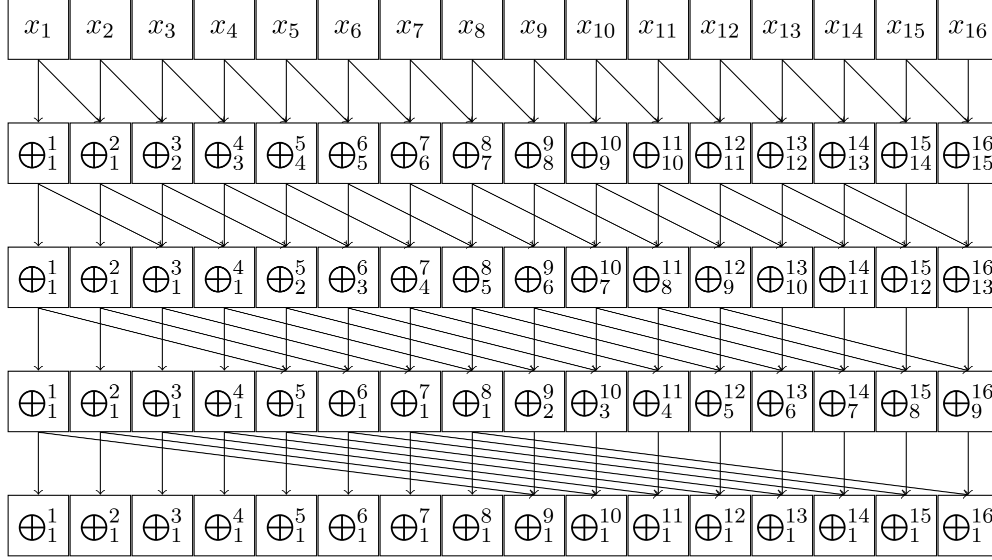

Figure 2.2: Hillis-Steele algorithm for input of size 16

Algorithm 4: Hillis-Steele scan algorithm (EREW PRAM)

Input: values $x_1, \ldots, x_n$
Output: inclusive scan $s_1, \ldots, s_n$
Auxiliary: intermediate results $r_1, \ldots, r_n$
$\forall i \in \hat{n} : \; s_i \leftarrow x_i$;
for $i \leftarrow 1$ **to** $log_2\ n$ **do**
  $\forall i \in \hat{n} : \; r_i \leftarrow s_i$;
  for $j \leftarrow 1$ **to** $n - 2^{i-1}$ **do in parallel**
    $s_{j+2^{i-1}} \leftarrow r_j \oplus r_{j+2^{i-1}}$;
  end
end
return s_1;

combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} \cdot log_2\ n\right) = O\left(\frac{n \cdot log_2\ n}{p}\right), \\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right), \\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n) = \omega(SU(n)), \\
W(n,p) =& \sum_{i=1}^{log_2\ n} n - 2^{i-1} = O(n \cdot log_2\ n) = \omega(SU(n)).
\end{aligned}$$

The number of the utilized processors is reduced each step, resulting in cost non-optimality. Also, the algorithm is not work-optimal because there are redundant applications of $\oplus$.

To achieve work optimality, the input could be split into p portions that are precomputed sequentially. Hillis-Steele is then applied to the results, and the result of that is then distributed back to the portions.

Algorithm 5: Modified Hillis-Steele scan algorithm (EREW PRAM)

```
Input: values x_1, ..., x_n
Output: inclusive scan s_1, ..., s_n
Auxiliary: intermediate results r_1, ..., r_p
equipartition x_1, ..., x_n to p contiguous sections;
foreach section x_i, ..., x_j do in parallel
    sequential inclusive scan (in: x_i, ..., x_j, out: x_i, ..., x_j);
    r_tid ← s_j;
end
Hillis-Steele algorithm (in: r, out: r);
foreach section x_i, ..., x_j do in parallel
    for k ← i to j do
        if tid ≠ 1 then
            s_k ← r_{tid-1} ⊕ s_k;
        end
    end
end
```

Both the sequential scan and partial result redistribution take $O\left(\frac{n}{p}\right)$. Hillis-Steele for input of size p and p processors has complexity $O\left(\frac{p \cdot log_2\ p}{p}\right)$. This combines to

$$
\begin{aligned}
T(n,p) &= O\left(\frac{n}{p}\right) + O\left(\frac{p \cdot log_2\ p}{p}\right) = O\left(\frac{n}{p} + log_2\ p\right), \\
S(n,p) &= \frac{O(n)}{O\left(\frac{n}{p} + log_2\ p\right)} = O\left(\frac{n \cdot p}{n + p \cdot log_2 p}\right), \\
C(n,p) &= p \cdot O\left(\frac{n}{p} + log_2\ p\right) = O(n + p \cdot log_2\ p) = \omega(SU(n)), \\
W(n,p) &= O(n) + O(p \cdot log_2\ p) = O(n + p \cdot log_2\ p) = \omega(SU(n)), \\
C\left(n, \frac{n}{log_2\ n}\right) &= O\left(n + \frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) = O\left(n + \frac{n}{log_2\ n} \cdot log_2\ n\right) = O(n) = \Theta(SU(n)), \\
W\left(n, \frac{n}{log_2\ n}\right) &= O\left(n + \frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) = O(n) = \Theta(SU(n)).
\end{aligned}
$$

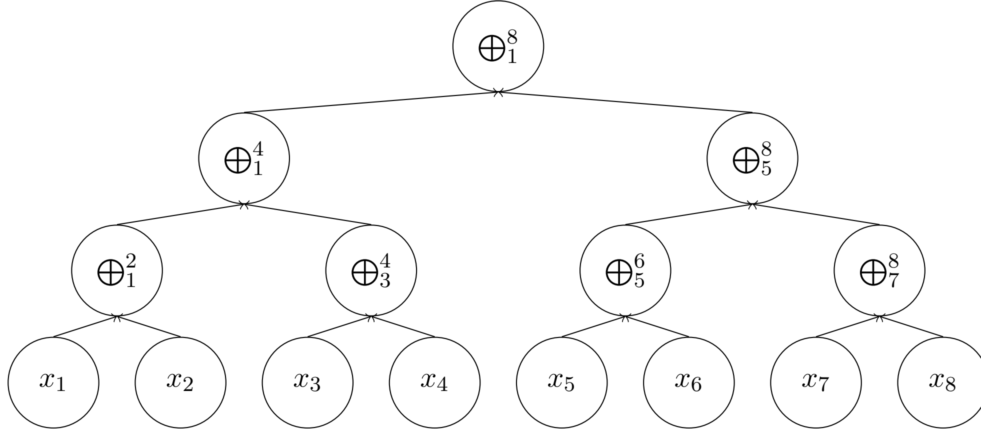

Figure 2.3: Up-Sweep step for the input of the size 8

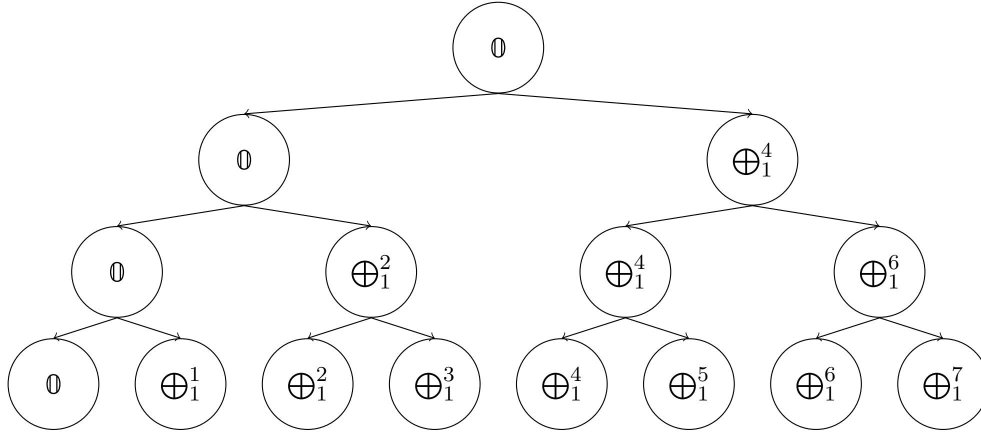

Figure 2.4: Down-Sweep step for the input of the size 8

Thus, this modification made Hillis-Steele cost and work optimal in case $\frac{n}{log_2\ n}$ processors are used.

The second algorithm was presented by Blelloch[7]. The algorithm works with a tree structure similar to the one used by reduction (see Figure 2.1) in 2 steps. The first step, which is almost identical with the parallel reduction, is called up-sweep and computes partial results traversing the tree from leaves towards the root (see Figure 2.3). The second step is called down-sweep and redistributes partial results from the root towards leaves (see Figure 2.4).

As the figures show, this algorithm is natively exclusive but can be modified to be inclusive. In the same spirit, Hillis-Steele is natively inclusive but can be modified to be exclusive.

Both steps consist of 2 nested loops. The inner one performs $O(n)$ steps split

Algorithm 6: Up-Sweep step of the Blelloch scan algorithm (EREW PRAM)

Input: values $x_1, \dots, x_n$
Output: intermediate results $r_1, \dots, r_n$
Auxiliary: left and right indices $left_1, \dots, left_{\frac{n}{2}}$ and $right_1, \dots, right_{\frac{n}{2}}$

```
∀i ∈ n̂ : r_i ← x_i;
for i ← 1 to log_2 n do
    for j ← 1 to n/2^i do in parallel
        left_j ← 1 + (j − 1) · 2^i;
        right_j ← left + 2^(i−1);
        r_{left_j} ← r_{left_j} ⊕ r_{right_j};
    end
end
```

Algorithm 7: Down-Sweep step of Blelloch scan algorithm (EREW PRAM)

Input: values $x_1, \dots, x_n$
Output: scan $s_1, \dots, s_n$
Auxiliary: left and right indices $left_1, \dots, left_{\frac{n}{2}}$ and $right_1, \dots, right_{\frac{n}{2}}$, temporary values $t_1, \dots, t_{\frac{n}{2}}$

```
∀i ∈ n̂ : s_i ← x_i;
for i ← log_2 n downto 1 do
    for j ← 1 to n/2^i do in parallel
        left_j ← 1 + (j − 1) · 2^i;
        right_j ← left + 2^(i−1);
        t_j ← s_{left_j} ⊕ s_{right_j};
        s_{left_j} ← s_{right_j};
        s_{right_j} ← t_j;
    end
end
```

among p processors. The outer one performs $O\left(log_2\ n\right)$ steps sequentially. Both steps also contain a copy that has a complexity of $O\left(\frac{n}{p}\right)$. This combines to

$$T\left(n,p\right) = O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} \cdot log_2\ n\right) = O\left(\frac{n \cdot log_2\ n}{p}\right).$$

Similarly to Hillis-Steele, input is equipartitioned first, and a sequential scan is applied to each partition first. Up-sweep and down-sweep are applied to intermediate results of those partitions. The results of those steps are then applied back to the partitions. The offset at which those intermediates are applied determines whether the algorithm will be inclusive or exclusive (this also applies to Hillis-Steele).

Algorithm 8: Blelloch scan alorithm (EREW PRAM)

```
Input: values x_1, ..., x_n
Output: inclusive scan s_1, ..., s_n
Auxiliary: intermediate results r_1, ..., r_p
equipartition x_1, ..., x_n to p contiguous sections;
foreach section x_i, ..., x_j do in parallel
|   sequential inclusive scan (in: x_i, ..., x_j, out: s_i, ..., s_j);
|   r_tid <- s_j;
end
up-sweep (in: r, out: r);
down-sweep (in: r, out: r);
foreach section x_i, ..., x_j do in parallel
|   for k <- i to j do
|   |   s_k <- r_tid ⊕ s_k;
|   end
end
```

Complexities of the Blelloch algorithm are

$$\begin{aligned}
T\left(n,p\right) =& O\left(\frac{n}{p}\right) + O\left(\frac{p \cdot log_2\ p}{p}\right) = O\left(\frac{n}{p} + log_2\ p\right), \\
S\left(n,p\right) =& \frac{O\left(n\right)}{O\left(\frac{n}{p} + log_2\ p\right)} = O\left(\frac{n \cdot p}{n + p \cdot log_2\ p}\right), \\
C\left(n,p\right) =& p \cdot O\left(\frac{n}{p} + log_2\ p\right) = O\left(n + p \cdot log_2\ p\right) = \omega\left(SU\left(n\right)\right), \\
W\left(n,p\right) =& O\left(n\right) + O\left(\sum_{i=1}^{log_2\ p} \frac{p}{2^i}\right) = O\left(n\right) + O\left(p \cdot \sum_{i=1}^{log_2\ p} \frac{1}{2^i}\right) = O\left(n+p\right) = O\left(n\right) = \Theta\left(SU\left(n\right)\right).
\end{aligned}$$

As for both steps, the number of utilized processors is halved each iteration, the algorithm is not cost-optimal.

2.2.2.2 Implementations

Similar to reduction, *C++17 numeric* library provides *std::inclusive_scan* that lacks control over the number of processors used.

As *OpenMP* does not offer sufficiently flexible implementation, custom inclusive scan based on Blelloch must be implemented for CPU.

For GPU implementations *thrust::inclusive_scan* could be used.

2.2.3 Exclusive scan

The exclusive scan is defined in definition 1.68.

The exclusive scan is almost identical to the inclusive scan. The result is just shifted.

2.2.3.1 Implementations

In *C++17 numeric* library, it's implemented as *std::exclusive_scan*.

In *thrust*, it's implemented as *thrust::exclusive_scan*.

2.3 Lists

2.3.1 Linked list

The linked list is defined in definition 1.69.

Two basic strategies exist for their implementation.

1. Using a linked structure. Each node holds a pointer to its successor. Nodes are dynamically allocated independently from each other.

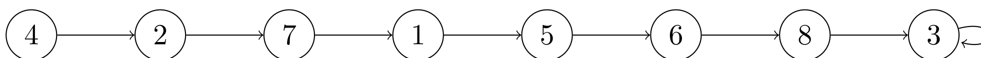

Figure 2.5: Linked structure

2. Using a successor array. Each element holds an index of its successor. Nodes are allocated en masse in a contagious memory block.

5	7	3	2	6	8	1	3
1	2	3	4	5	6	7	8

Figure 2.6: Successor array

For the purpose of this thesis, the successor array is more beneficial as it allows for quick allocation and random access. Quick insertion/removal, which is a strong side of the linked structure, is not required.

2.3.1.1 Implementations

C++ list library implements *std::list* utilizing the linked structure strategy.

2.3.2 List ranking

The list ranking is defined in definition 1.72.

The rank of a node is equal to the number of its predecessors. This can be achieved by using an inclusive scan, respecting order given by the list.

The sequential solution consists of a simple counter and traversal of the list. Hence, the algorithm 9.

As each node is visited once, the complexity is

$$T(n) = O(n).$$

As each node has to be visited to set its rank, the bounds must be the same.

$$SL(n) = SU(n) = O(n)$$

The parallel solution of the list ranking can be achieved by modifying the Hillis-Steele algorithm 4 for the scan.

Algorithm 9: Sequential list ranking

Input: successor array $s_1, \ldots, s_n$, head index h
Output: ranking $r_1, \ldots, r_n$
$r \leftarrow \mathbb{0}$;
while $s_h \neq h$ **do**
    $r_h \leftarrow r \leftarrow r + 1$;
    $h \leftarrow s_h$;
end
$r_h \leftarrow r + 1$;

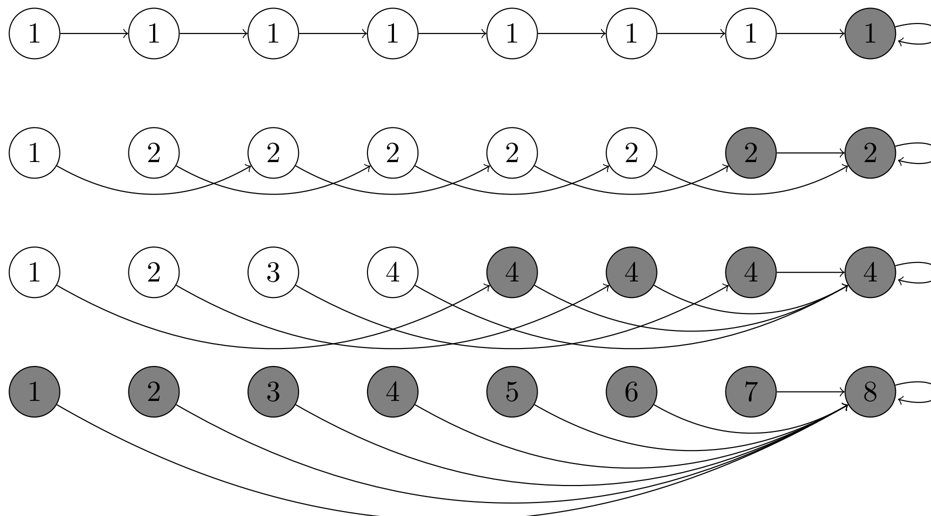

Figure 2.7: Pointer jumping

The order given by the list has to be respected. Thus instead of direct indexing and distance, the successor array has to be used. To emulate the distance, its value is set to $s_i \leftarrow s_{s_i}$ in each iteration. This is called *pointer jumping* and is depicted by figure 2.7. This formulates to the algorithm 10.

The inner loop has $O\left(n\right)$ iterations split among p processors. The outer loop has $O\left(log_2\ n\right)$ iterations. The initialization of arrays has $O\left(\frac{n}{p}\right)$ complexity.

Algorithm 10: Pointer jumping (CREW PRAM)

```
Input: successor array s_1, ..., s_n
Output: ranking r_1, ..., r_n
Auxiliary: active indicators a_1, ..., a_n
∀i ∈ n̂ : a_i ← active, r_i ← 1;
for i ← 1 to log_2 n do
    for j ← 1 to n do in parallel
        if a_i = active then
            r_{s_i} ← r_i + r_{s_i};
            if s_i = s_{s_i} then
                a_i ← inactive;
            else
                s_i ← s_{s_i};
            end
        end
    end
end
```

This combines to

$$
\begin{aligned}
T(n,p) &= O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} \cdot log_2\ n\right) = O\left(\frac{n \cdot log_2\ n}{p}\right), \\
S(n,p) &= \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right), \\
C(n,p) &= p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n) = \omega(SU(n)), \\
W(n,p) &= \sum_{i=1}^{log_2\ n} n - 2^i = O(n \cdot log_2\ n) = \omega(SU(n)).
\end{aligned}
$$

The number of utilized processors is reduced each iteration, this is the source of the cost non-optimality. Also, the addition is performed redundantly, resulting in work non-optimality. As it's not possible to identify equal-sized partitions of the list, modification of the Hillis-Steele algorithm won't work.

The observation 2.1 describes an idea of a cost/work-optimal list ranking.

Observation 2.1 *Let there be a linked list of size* $n' = \frac{n}{log_2\ n} = O(n)$*. Then*

$$\begin{aligned}
T(n', n') =& O(log_2\ n') = O(log_2\ n),\\
C(n', n') =& O(n' \cdot log_2\ n')\\
=& O\left(\frac{n}{log_2\ n} \cdot log_2\ n\right)\\
=& O(n) = \qquad \Theta(SU(n)).
\end{aligned}$$

The idea requires the list to be shrunk to the size n' using n' processors in $T(n, n') = O(log_2\ n)$ and with $C(n, n') = O(n)$. Then, pointer jumping may be applied to that shrunken list, and the original list has to be restored with the same complexity.

Meeting all those requirements results in

$$\begin{aligned}
T(n, n') =& O(log_2\ n),\\
S(n, n') =& \frac{O(n)}{O(log_2\ n)} = O\left(\frac{n}{log_2\ n}\right),\\
C(n, n') =& O(n) + n' \cdot O(log_2\ n) = O\left(\frac{n}{log_2\ n} \cdot log_2\ n\right) = O(n) = \Theta(SU(n)).
\end{aligned}$$

Thus, the algorithm would be cost and potentially work-optimal.

To achieve work-optimality (without cost-optimality), the independent sets (defined in definition 1.70) could be used for shrinking the list.

To get such an independent set, 3-coloring could be used, as it's the best coloring achievable in EREW PRAM. Such independent set would have size $O\left(\frac{n}{2\cdot 3-2}\right) = O\left(\frac{n}{4}\right)$ (see proof 1.2). Thus size of the shrunk list $L' = L \setminus I$ is $\Omega\left(\frac{3\cdot n}{4}\right)$.

Shrinking can be repeated until the required size is achieved. To achieve that, $\Theta(log_2^2\ n)$ shrinkings has to be done, as

$$\begin{aligned}
\left(\frac{3}{4}\right)^s \cdot n \leq& \frac{n}{log_2\ n},\\
\left(\frac{4}{3}\right)^s \geq& log_2\ n,\\
s \geq& log_{\frac{4}{3}}\ log_2\ n = \Theta\left(log_2^2\ n\right).
\end{aligned}$$

2.3.2.1 6-coloring

The k-coloring is defined in definition 1.71.

The 6-coloring is used as an intermediate step to achieve 3-coloring. *Deterministic Coin Tossing* technique is used to get it and is formulated in algorithm 11.

Function $diff(x, y)$ return least bit index at which binary representations of x and y differ. $x_{[i]}$ denotes i-th bit of x. Function $log_b^* \ x$ is defined as $min\{i : \ log_b^i \ x \leq 1\}$.

Algorithm 11: 6-coloring (CREW PRAM)

```
Input: successor array s_1, ..., s_n
Output: coloring c_1, ..., c_n
Auxiliary: difference π_1, ..., π_n
(* Generate n-coloring *)
for i ← 1 to n do in parallel
  | c_i ← i;
end
(* Reduce to 6-coloring *)
for i ← 1 to log*_2 n do
  | for j ← 1 to n do in parallel
  |   | π_j ← diff(c_j, c_{s_j});
  |   | c_j ← 2 · π_j + c_{j[π_j]};
  | end
end
```

The inner loop preserves the validity of the coloring and thus π is always well defined. The best coloring that can be achieved by DCT is 6-coloring.

The n-coloring generation has n iterations split among p processors. The inner loop of the 6-coloring reduction has n iterations split among p processors as well. The outer loop of the 6-coloring reduction has $log_2^* \ n$ iterations. As the value of log_2^* won't exceed 6 for any representable value (as $log_2^* \ 2^{2^{65536}} - 1 = 6$), it could be neglected and considered constant. This combines to

$$\begin{aligned} T(n, p) &= O\left(\frac{n}{p}\right), \\ C(n, p) &= p \cdot O\left(\frac{n}{p}\right) = O(n), \\ W(n, p) &= n + n \cdot log_2^* \ n = O(n). \end{aligned}$$

2.3.2.2 3-coloring

The 3-coloring is obtained by reduction of 6-coloring by gradual elimination of colors.

Algorithm 12: 3-coloring (EREW PRAM)

```
Input: successor array s_1, ..., s_n
Output: coloring c_1, ..., c_n
6-coloring (in: s, out: c);
for i ← 1 to n do in parallel
    if c_i = 5 then
        c_i ← any of {0,1,2} \ {c_{s_i}, c_j}, where s_j = i;
    end
end
for i ← 1 to n do in parallel
    if c_i = 4 then
        c_i ← any of {0,1,2} \ {c_{s_i}, c_j}, where s_j = i;
    end
end
for i ← 1 to n do in parallel
    if c_i = 3 then
        c_i ← any of {0,1,2} \ {c_{s_i}, c_j}, where s_j = i;
    end
end
```

To get a predecessor, the list can be reversed, which takes $O\left(\frac{n}{p}\right)$.

Each loop has n iterations split among p processors. This results in

$$\begin{aligned} T(n,p) &= O\left(\frac{n}{p}\right), \\ C(n,p) &= p \cdot O\left(\frac{n}{p}\right) = O(n), \\ W(n,p) &= O(n). \end{aligned}$$

2.3.2.3 Work-optimal list ranking

This all can be used to formalize an algorithm 13 for a work-optimal list ranking.[4][8]

Algorithm 13: Work-optimal list ranking (CREW PRAM)

Input: successor array $s_1, \ldots, s_n$
Output: ranking $r_1, \ldots, r_n$
Auxiliary: indicators $f_1, \ldots, f_n$, results $n_1, \ldots, n_n$, coloring $c_1, \ldots, c_n$, predecessors $p_1, \ldots, p_n$, stack S, temporary $t_1, \ldots, t_n$

$\forall i \in \widehat{n}: \ S \leftarrow \emptyset,\ n' \leftarrow n,\ r_i \leftarrow 1$;
reverse(**in:** s, **out:** p);
while $n' > \frac{n}{log_2\ n}$ **do**
  $\forall i: \ t_i \leftarrow \emptyset$;
  3-coloring(**in:** $s_1, \ldots, s_{n'}$, **out:** $c_1, \ldots, c_{n'}$);
  (* Identify I *)
  for $i \leftarrow 1$ **to** n' **do in parallel**
    $f_i \leftarrow c_i < min(c_{p_i}, c_{s_i})$;
    $n_i \leftarrow 1$ *iff* f_i *else* 0;
  end
  (* Remove I from L *)
  inclusive scan(**in:** $n_1, \ldots, n_{n'}$, **out:** $n_1, \ldots, n_{n'}$);
  for $i \leftarrow 1$ **to** n' **do in parallel**
    if f_i **then**
      $t_{n_i} \leftarrow (i, s_i, p_i, r_i)$;
      $r_{p_i} \leftarrow r_{p_i} + r_i$;
      $s_{p_i} \leftarrow s_i,\ p_{s_i} \leftarrow p_i$;
    end
  end
  (* Compact $L' = L \setminus I$ to consecutive memory locations *)
  $\forall i \in \widehat{n}: \ n_i \leftarrow 0$ *iff* f_i *else* 1;
  inclusive scan(**in:** $n_1, \ldots, n_{n'}$, **out:** $n_1, \ldots, n_{n'}$);
  for $i \leftarrow 1$ **to** n' **do in parallel**
    if $\neg f_i$ **then**
      $s_{n_i} \leftarrow n_{s_i},\ p_{n_i} \leftarrow n_{p_i}$;
      $r_{n_i} \leftarrow r_i$;
    end
  end
  $S \leftarrow (S, \{t_i : \ t_i \neq \emptyset\})$;
  $n' \leftarrow n' - |\{t_i : \ t_i \neq \emptyset\}|$;
end
pointer jumping (**in:** $s_1, \ldots, s_{n'}$, **out:** $r_1, \ldots, r_{n'}$);
restore L and rank removed nodes by emptying stack S and reversing steps in removal and compaction;

All initializations can be executed in $O\left(\frac{n}{p}\right)$ time and $O\left(n\right)$ work. List reverse has the same complexities.

The big loop has $O\left(log_2^2\ n\right)$ iterations. Both removal and compaction have n' iterations split among p processors. There's a 3-coloring and inclusive scan as a part of that loop.

Overall, the big loop has combined complexities of

$$\begin{aligned}
T\left(n,p\right) =& O\left(log_2^2\ n\right)\cdot O\left(\frac{n}{p}+log_2\ p\right) = O\left(\frac{\left(n+p\cdot log_2\ p\right)\cdot log_2^2\ n}{p}\right),\\
W\left(n,p\right) =& O\left(n\right).
\end{aligned}$$

The pointer jumping used there is the same pointer jumping as work non-optimal list ranking. Restoring the original list has identical complexities as it's shrinking.

All of this combines to

$$\begin{aligned}
T\left(n,p\right) =& O\left(\frac{\left(n+p\cdot log_2\ p\right)\cdot log_2^2\ n}{p}\right),\\
S\left(n,p\right) =& O\left(\frac{n\cdot p}{\left(n+p\cdot log_2\ p\right)\cdot log_2^2\ n}\right),\\
C\left(n,p\right) =& O\left(\left(n+p\cdot log_2\ p\right)\cdot log_2^2\ n\right) = \omega\left(SU\left(n\right)\right),\\
W\left(n,p\right) =& O\left(n\right) = \Theta\left(SU\left(n\right)\right).
\end{aligned}$$

Thus the algorithm is work-optimal. The cost non-optimality is caused by the fact that not all processors are utilized in the shrunk list in case there are enough of them for the whole list, and const non-optimality of used algorithms even in case there's just enough of them for the shrunk list. Even though efficiency is higher, overall speed is lower than pointer jumping.

2.3.2.4 Implementations

There's no proper implementation of list ranking. *C++ std::inclusive_scan* together with *std::list* could be used for sequential ranking, but there's no such substitute for parallel ranking.

2.4 Euler Tour Technique

The Euler Tour Technique is defined in definition 1.74.

2.4.1 Algorithms

Note 2.1 *The sequential solution is omitted as there's no use for it in this thesis. It's complexity is* $T(n) = SU(n) = SL(n) = O(n)$*.*

The parallel solution utilizes an arc representation of the tree. Each edge is represented by a pair of arcs (1 downgoing, 1 upgoing). Those arcs are arranged into an array where arcs originating in a single vertex form a contagious block and opposite arcs are adjacent to each other.

Let there be the following notation.

- $origin(xy \updownarrow) = x$
- $target(xy \updownarrow) = y$
- $type(xy \downarrow) = \downarrow, type(yx \uparrow) = \uparrow$
- $opposite(xy \downarrow) = yx \uparrow$
- $next(xy \updownarrow) = \begin{cases} x(child_{i+1}(x)) \downarrow, \text{ iff } y = child_i(x) \wedge i < arity(x), \\ x(parent(x)) \uparrow, \text{ iff } y = child_{arity(x)}(x) \wedge parent(x) \neq root, \\ x(child_1(x)) \downarrow, \text{ iff } xy \updownarrow \ is\ x(parent(x)) \uparrow \wedge arity(x) > 0, \\ xy \updownarrow, otherwise. \end{cases}$

This representation is depicted in figure 2.8. The *next* function is depicted by the red arrow. The *opposite* function is depicted by the blue arrow.

The path is defined as

$$path(a) := \begin{cases} a, \text{ iff } a\ is\ (child_{arity(root)}(root))(root) \uparrow, \\ next(opposite(a)), otherwise \end{cases}$$

The *path* serves as a successor function for a linked list of arcs. Using list ranking to determine the index, arcs could be reordered in order of the Euler tour. Hence the algorithm 14.[9]

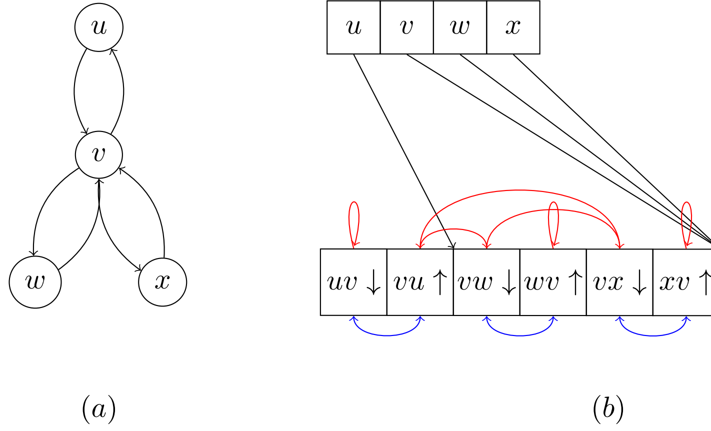

Figure 2.8: (a) Euler circuit of the tree (b) Array representation of the arcs

---

Algorithm 14: Euler Tour construction (EREW PRAM)

Input: tree $T = (V, E)$
Output: Euler tour $a_1, \dots, a_{2\cdot|V|-2}$
Auxiliary: path $p_1, \dots, p_{2\cdot n-2}$, ranks $r_1, \dots, r_{2\cdot n-2}$
foreach $\{u, v\} \in E$ **do in parallel**
  create $uv \downarrow$ & $vu \uparrow$ and arrange them into the array a;
end
for $i \leftarrow 1$ **to** $2 \cdot n - 2$ **do in parallel**
  $p_i \leftarrow next(opposite(a_i))$;
end
$last \leftarrow (child_{arity(root)}(root))(root) \uparrow p_{last} \leftarrow last$;
list ranking (**in:** p, **out:** r);
reorder a with respect to the ranking r;

Arc creation has $n - 1$ iterations split among p processors. Path computation has $2 \cdot n - 2$ iterations split among p processors. Ordering with list ranking available has complexity $O\left(\frac{n}{p}\right)$.

Total complexity depends on the list ranking algorithm used.

For pointer jumping, it combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) + O\left(\frac{n \cdot log_2\ n}{p}\right) = O\left(\frac{n \cdot log_2\ n}{p}\right),\\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right),\\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n) = \omega(SU(n)),\\
W(n,p) =& O(n) + O(n \cdot log_2\ n) = O(n \cdot log_2\ n) = \omega(SU(n)).
\end{aligned}$$

For work-optimal ranking, it combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) + O\left(\frac{(n + p \cdot log_2\ p) \cdot log_2^2\ n}{p}\right) = O\left(\frac{(n + p \cdot log_2\ p) \cdot log_2^2\ n}{p}\right),\\
S(n,p) =& \frac{O(n)}{O\left(\frac{(n + p \cdot log_2\ p) \cdot log_2^2\ n}{p}\right)} = O\left(\frac{n \cdot p}{(n + p \cdot log_2\ p) \cdot log_2^2\ n}\right),\\
C(n,p) =& p \cdot O\left(\frac{(n + p \cdot log_2\ p) \cdot log_2^2\ n}{p}\right) = O\left((n + p \cdot log_2\ p) \cdot log_2^2\ n\right) = \omega(SU(n)),\\
W(n,p) =& O(n) = \Theta(SU(n)).
\end{aligned}$$

Thus, optimality is fully dependent on the list ranking used.

2.4.2 Implementations

No implementations of the Euler Tour construction as described above exist in any popular libraries. *Boost boost:depth_first_search* can be used as a sequential substitute. No such parallel substitute exists.

2.4.3 Applications

ETT has many applications. The most important one (and the one required in this thesis) is tree nodes depths parallel computation. This is formulated in algorithm 15.

Even though there are potentially a lot of write-write conflicts in the last loop, the same value will be written to the conflicting ones. Thus, this algorithm can still be used for EREW PRAM, even though it's a Common CREW PRAM.

Algorithm 15: Get depths of nodes of the tree (EREW PRAM)

```
Input: tree T = (V, E)
Output: array of depths d_1, ..., d_n
Auxiliary: array of arcs a_1, ..., a_{2·|V|−2}, temporaries t_1, ..., t_{2·|V|−2}
Euler tour construction (in: T, out: a);
for i ← 1 to 2 · |V| − 2 do in parallel
    t_i ← 1 iff a is ↓ else −1;
end
inclusive scan (in: t, out: t);
for i ← 1 to 2 · n − 2 do in parallel
    if a_i is downgoing then
        d_{target(a_i)} ← t_i;
    end
end
```

The complexities are the same as in case of Euler Tour construction.

2.5 Parentheses matching

The parentheses matching is defined in definition 1.76.

2.5.1 Algorithms

In all algorithms presented, only well-formed strings are allowed for the sake of simplicity.

The sequential solution formulated in the algorithm 16 stores unmatched opening parentheses on the stack and matches them with encountered closing parentheses.

As this algorithm performs a single pass of the string, it has a complexity

$$T(n) = O(n).$$

As matching has to write a matched parenthesis for each parenthesis, it has to perform at least a single full pass. Thus,

$$SL(n) = SU(n) = O(n).$$

Algorithm 16: Sequential parentheses matching

Input: string $p_1, \ldots, p_n$
Output: matches $m_1, \ldots, m_n$
Auxiliary: stack S
$S \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** n **do**
  if p_i *is* (**then**
    $S \leftarrow (S, i)$;
  else
    $S, t \leftarrow S', top$, where $S = (S', top)$;
    $m_i \leftarrow t,\ m_t \leftarrow i$;
  end
end

There are multiple parallel solutions. Two of them will be presented as they will be used later.

The first one utilizes parentheses depths.[10] Matching parentheses have the same depths, and there's no parenthesis with the same depth between them. Thus, parentheses depth is computed first, and then parentheses are stable sorted. This way, matching parentheses will be adjacent and may be quickly paired. Hence, the algorithm 17.

All loops have n iterations split among p processors. Any parallel stable sort has time complexity $T(n,p) = \Omega\left(\frac{n \cdot log_2\ n}{p}\right)$ with parallel work $W(n,p) = \Omega(n \cdot log_2\ n)$. This together with inclusive scan combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) + O\left(\frac{n \cdot log_2\ n}{p}\right) = O\left(\frac{n \cdot log_2\ n}{p}\right),\\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right),\\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n) = \omega(SU(n)),\\
W(n,p) =& O(n) + O(n \cdot log_2\ n) = O(n \cdot log_2\ n) = \omega(SU(n)).
\end{aligned}$$

Sorting prevents cost and work optimality as it's not possible to do it quicker/more efficiently.

The second one is more similar to the sequential solution. Instead of using a stack, the stack is replaced by an array, thus allowing random access and parallelization.

Algorithm 17: Depth-based parallel parentheses matching (EREW PRAM)

Input: string $p_1, \dots, p_n$
Output: matches $m_1, \dots, m_n$
Auxiliary: depths $d_1, \dots, d_n$, permutation $\pi_1, \dots, \pi_n$
for $i \leftarrow 1$ **to** n **do in parallel**
  $d_i \leftarrow 1$ iff p_i *is* (*else* -1;
end
inclusive scan (**in:** d, **out:** d);
for $i \leftarrow 1$ **to** n **do in parallel**
  if p_i *is*) **then**
    $d_i \leftarrow d_i + 1$;
  end
end
stable sort(**in:** d, **out:** π);
for $i \leftarrow 1$ **to** $\frac{n}{2}$ **do in parallel**
  $m_{\pi^{-1}(2 \cdot i - 1)} \leftarrow \pi^{-1}(2 \cdot i)$;
  $m_{\pi^{-1}(2 \cdot i)} \leftarrow \pi^{-1}(2 \cdot i - 1)$;
end

First, every processor matches its partition of the input string. It forms a sequence of unmatched closing and opening parentheses $(x_), x_()$. All unmatched closing parentheses will precede all unmatched opening parentheses.

The result of the previous step is then reduced. Processors are matching parentheses in pairs $((l_), l_(), (r_), r_())$. First, $l_($ is matched with $r_)$. One of the three situations may occur.

- All parentheses are matched.
- Some parentheses from $l_($ are unmatched. Then they're prepended to $r_($.
- Some parentheses from $r_)$ are unmatched. Then they're appended to $l_)$.

This is formulated in algorithm 18.

Partition-based sequential preprocessing takes $O\left(\frac{n}{p}\right)$ time and $O(n)$ work. The inner loop has $\frac{n}{2^i} = O(n)$ iterations split among p processors. Each such iteration has $O(n)$ matches/moves. But across all processors, it totals with $O(n)$ operations split among p processors. The outer loop has $log_2\ p$

Algorithm 18: Work-optimal parallel parentheses matching (EREW PRAM)

Input: string $p_1, \ldots, p_n$
Output: matches $m_1, \ldots, m_n$
Auxiliary: unmatched parentheses $u_1, \ldots, u_n$, # of unmatched parentheses $l_1, \ldots, l_P$, $r_1, \ldots, r_P$ where P is the # of processors

equipartition $p_1, \ldots, p_n$ to P contiguous sections;
foreach *section* $p_i, \ldots, p_j$ **do in parallel**
  sequential parentheses matching (**in:** $p_i, \ldots, p_j$, **out:** $m_i, \ldots, m_j$);
  arrange unmatched parentheses to $t_i, \ldots, t_j$ with closing parentheses start at i and opening parentheses end at j;
  $l_{tid} \leftarrow$ # of unmatched opening parentheses;
  $r_{tid} \leftarrow$ # of unmatched closing parentheses;
end
for $i \leftarrow 1$ **to** $log_2\ P$ **do**
  for $j \leftarrow 1$ **to** $\frac{P}{2^i}$ **do in parallel**
    $llo \leftarrow tid \cdot 2^i,\ rlo \leftarrow llo + 2^{i-1}$;
    $rbase \leftarrow n \cdot \frac{rlo}{p},\ lbase \leftarrow rbase - 1$;
    $matched \leftarrow min\{left_{llo}, right_{rlo}\}$;
    for $k \leftarrow 1$ **to** $matched$ **do**
      match $t_{lbase-k+1}$ with $t_{rbase+k-1}$;
    end
    if $left_{llo} > matched$ **then**
      $rem \leftarrow left_{llo} - matched$;
      $rbase \leftarrow n \cdot \frac{rlo+2^{i-1}}{p} - left_{rlo} - 1$;
      move $t_{lbase-left_{llo}+1}, \ldots, t_{lbase-matched}$ to $t_{rbase-rem+1}, \ldots, t_{rbase}$;
    else if $right_{rlo} > matched$ **then**
      $rem \leftarrow right_{llo} - matched$;
      $lbase \leftarrow n \cdot \frac{llo}{p} + left_{llo}$;
      move $t_{rbase+matched}, \ldots, t_{rbase+right_{rlo}-1}$ to $t_{lbase}, \ldots, t_{lbase+rem-1}$;
    end
    $left_{llo} \leftarrow left_{llo} + left_{rlo} - matched$;
    $right_{llo} \leftarrow right_{llo} + right_{rlo} - matched$;
  end
end

---

iterations. This combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} \cdot log_2\ p\right) = O\left(\frac{n \cdot log_2\ p}{p}\right),\\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ p}{p}\right)} = O\left(\frac{p}{log_2\ p}\right),\\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ p}{p}\right) = O(n \cdot log_2\ p) = \omega(SU(n)),\\
W(n,p) =& O(n) + O(n \cdot log_2\ p) = O(n \cdot log_2\ p).
\end{aligned}$$

As processors become gradually inactive, this results in cost non-optimality. As some of the moves are redundant, the algorithm is not work-optimal either but it is very close to it with a lower number of processors. Also, the case when there's substantial redundant work done is rare, and thus, it has an average case time of $\frac{n}{p}$.

2.5.2 Implementations

There are no proper implementations of parentheses matching in a *C++ STL* nor *Boost*.

2.6 DFTA run

To run the DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ for ground term $t \in T(\mathcal{F})$ means to evaluate the extended transitition function $\widehat{\Delta}(t)$. The extended transition function is defined in definition 1.40.

The sequential solution can be achieved by traversing the tree (for example by using DFS) and evaluating $\widehat{\Delta}$ from leaves towards the root. Hence, the algorithm 19.

As this algorithm consists of a single simple pass of the tree, the complexities are

$$T(n) = SL(n) = SU(n) = O(n).$$

Even though this algorithm can be parallelized as any other DFS, it's far from optimal.

Algorithm 19: Sequential run of the DFTA

Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, tree $t = (labels, children, root)$ of the size n

Output: states $state_1, \dots, state_n$

foreach $child \in children_{root}$ **do**

  run (**in:** A, $t' \leftarrow (labels, children, child)$, **out:** $state$);

end

$state_{root} \leftarrow \Delta_{labels_{root}}(state_{children_{root,1}}, \dots, state_{children_{root,arity(root)}})$;

Parallel run of k-local DFTA is achievable utilising the fact that any term of MVD at least k is synchronizing (i.e. subtrees below this depth of k don't affect the resulting state), thus states of nodes at layers separated by at least $k-1$ another layers can be computed in parallel without affecting each other.

2.6.1 The main algorithm

To achieve better parallelization for k-local DFTAs, the fact that states of nodes separated by at least $k-1$ layers, or laying on the same layer, are not affecting each other, is utilized.

Even though two such layers don't affect each other directly, the result of the upper layer is dependent on a correct state of the intermediate layers, that are dependent on the lower layer. To ensure that, the algorithm runs in two passes. The first pass (also called synchronization pass) is used to initialize state of each node correctly. The second pass is used to obtain the correct states of all nodes.

The nodes are split to groups based on the layer they lie in. Each group contains nodes that are independent on each other (i.e. they lie on ith and $(i + l \cdot k)$th layer, where l is any integer). All the groups are arranged into an linear array to ease the computation.

As the groups will be bigger than number of processors available in most cases, the computation step is precomputed for each node.

The complexity will be analyzed later after all functions are analyzed (see 2.6.5.

Algorithm 20: Parallel run of the k-local DFTA (EREW PRAM)

Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, tree $t = (labels, children, root)$ of the size n
Output: states $state_1, \ldots, state_n$
Auxiliary: depths $depth_1, \ldots, depth_n$, steps $step_1, \ldots, step_n$, depth-mod-k order $dmk_1, \ldots, dmk_n$
get depths using ETT (**in:** t, **out:** $depth$);
depth-mod-k sort (**in:** t, $depth$, **out:** dmk);
compute step (**in:** dmk, $depth$, **out:** $step$);
$\forall i \in \widehat{n} : \; state_i \leftarrow 0$;
compute state (**in:** A, t, dmk, $step$, **out:** $state$);
compute state (**in:** A, t, dmk, $step$, **out:** $state$);

2.6.2 Depth-mod-k sort

The depth-mod-k order is a modified BFS traversal of the tree, where all layers in the same group as described in the previous section are merged. The order is ascending with respect to *depth mod k*.

To achieve such traversal ETT with parentheses matching could be used. The Euler tour of the tree is acquired first. Then each downgoing arc is replaced by a closing parenthesis, and each upgoing arc is replaced by an opening parenthesis.

Whole such string is wrapped into parentheses, whose count is equal to the depth of the tree, to ensure that the string is well-formed. Those wrapping parentheses will match unmatched parentheses in the Euler tour. In addition to that, they can be used to identify individual layers.

To acquire a linked list from the matching, downgoing arcs are re-linked to the opposite arcs first. Then arcs at the end of each layer are re-linked to appropriate layer.

The depth-mod-k order is then obtained by ranking such list.

The algorithm consists of an ETT, reduction, parentheses matching, and list ranking, which are all analyzed in previous sections. Besides those, the rest of the algorithm is simple conflict-less assignments.

Algorithm 21: depth-mod-k sort (EREW PRAM)

Input: tree $t = (labels, children, root)$ of size n, depths array $depth_1, \ldots, depth_n$

Output: depth-mod-k order $dmk_1, \ldots, dmk_n$

Auxiliary: Euler tour $e_1, \ldots, e_{2 \cdot n-2}$, parentheses $par_{-max(depth)}, \ldots, par_{2 \cdot n-2+max(depth)-1}$, successor array $next_{-max(depth)}, \ldots, next_{2 \cdot n-2+max(depth)-1}$, ranking $r_{-max(depth)}, \ldots, r_{2 \cdot n-2+max(depth)-1}$

```
create Euler tour (in: t, out: e);
height ← reduce (in: depth);
foreach e_i ∈ e do in parallel
    par_{i-1} ← { ( iff e_i is downgoing
                { ) otherwise                ;
end
for i ← 1 to height do in parallel
    par_{-i} ← (;
    par_{|e|-1+i} ←);
end
match parentheses (in: par, out: next);
foreach e_i ∈ e do in parallel
    if e_i is downgoing then
        next_{i-1} ← index of opposite(e_i) − 1;
    end
end
for i ← 1 to height do in parallel
    next_{|e|-1+i} ← { next_{-i-k} iff i < height − k
                     { next_{-i mod k} − 2 iff i mod k ≠ k − 1    ;
                     { |e| − 1 + i otherwise
end
rank list (in: next, out: r);
construct dmk from r as its inversion;
```

Using non-optimal algorithms, this combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{n \cdot log_2\ n}{p}\right) + O(n)\,p = O\left(\frac{n \cdot log_2\ n}{p}\right),\\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n),\\
W(n,p) =& O(n \cdot log_2\ n) + O(n) = O(n \cdot log_2\ n).
\end{aligned}$$

Using optimal algorithms, this combines to

$$\begin{aligned}
T(n,p) =& O\left(\frac{(n + p \cdot log_2\ p) \cdot log_2^2\ n}{p}\right) + O\left(\frac{n \cdot log_2\ n}{p}\right) + O\left(\frac{n}{p}\right)\\
=& O\left(\frac{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}{p}\right),\\
C(n,p) =& p \cdot O\left(\frac{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}{p}\right)\\
=& O\left(p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n\right),\\
W(n,p) =& O(n) + O(n \cdot log_2\ p) = O(n \cdot log_2\ p).
\end{aligned}$$

The non-work-optimal variant is better suited for cases where the number of processors is high and close to n. The work-optimal variant is better when only a few processors are available.

This is obvious when p is assumed to be $O(n)$ (many processors) or $O(1)$ (few processors). For the non-work-optimal variant, it is

$$\begin{aligned}
T(n, O(1)) =& O\left(\frac{n \cdot log_2\ n}{1}\right) = O(n \cdot log_2\ n),\\
W(n, O(1)) =& O(n \cdot log_2\ n),\\
T(n, O(n)) =& O\left(\frac{n \cdot log_2\ n}{n}\right) = O(log_2\ n),\\
W(n, O(n)) =& O(n \cdot log_2\ n).
\end{aligned}$$

For the work-optimal variant, it is

$$
\begin{aligned}
T\left(n, O\left(1\right)\right) =& O\left(\frac{1 \cdot log_2\ 1 \cdot log_2^2\ n + n \cdot log_2\ n}{1}\right) = O\left(n \cdot log_2\ n\right), \\
W\left(n, O\left(1\right)\right) =& O\left(n \cdot log_2\ 1\right) = O\left(n\right), \\
T\left(n, O\left(n\right)\right) =& O\left(\frac{n \cdot log_2\ n \cdot log_2^2\ n + n \cdot log_2\ n}{n}\right) = O\left(log_2\ n \cdot log_2^2\ n\right), \\
W\left(n, O\left(n\right)\right) =& O\left(n \cdot log_2\ n\right).
\end{aligned}
$$

2.6.3 Step computation

Steps are pre-computed based on depth-mod-k order. Nodes are split into groups based on depth mod k first. Those groups are then split into sub-groups of size p. Each such sub-group corresponds to a single step of computation. Hence, the algorithm 22.

Algorithm 22: compute step (EREW PRAM)

Input: depth-mod-k order $dmk_1, \ldots, dmk_n$, depths $depth_1, \ldots, depth_n$
Output: steps $step_1, \ldots, step_n$
Auxiliary: groups $group_1, \ldots, group_n$, group end indices $gei_1, \ldots, gei_n$
for $i \leftarrow 1$ **to** n **do in parallel**
  $group_i \leftarrow depth_{dmk_i}\ mod\ k$;
  $gei_i \leftarrow i$ iff $n - 1 \vee group_i \neq group_{i+1}$ $else$ ∞;
end
inclusive suffix min scan (**in:** gei, **out:** gei);
for $i \leftarrow 1$ **to** n **do in parallel**
  $step_i \leftarrow 1$ iff $(gei_i - 1)\ mod\ p\ else\ 0$;
end
inclusive suffix scan (**in:** $step$, **out:** $step$);

As the algorithm consists of 2 pairs of simple loops and scans, the complexities are

$$
\begin{aligned}
T\left(n, p\right) =& O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} + log_2\ p\right) = O\left(\frac{n}{p} + log_2\ p\right), \\
C\left(n, p\right) =& p \cdot O\left(\frac{n}{p} + log_2\ p\right) = O\left(n + p \cdot log_2\ p\right), \\
W\left(n, p\right) =& O\left(n\right).
\end{aligned}
$$

2.6.4 State computation

Node states are computed in depth-mod-k order. Besides group boundaries, there's no processor stalling. There are $k - 1$ such boundaries, and at most $p - 1$ processors will stall. This means that $O(p \cdot k)$ stalls occur throughout the run. Hence, the algorithm 23.

Algorithm 23: compute state (EREW PRAM)

```
Input: DFTA A = (Q, F, Q_f, Δ), tree t = (labels, children, root) of
       the size n, depth-mod-k order dmk_1, ..., dmk_n, steps
       step_1, ..., step_n
Output: states state_1, ..., state_n
do in parallel
    s ← 1;
    i ← n − pid;
    while i ≥ 1 do
        if step_i = s then
            state_i ← Δ_{labels_i}(state_{child_1(i)}, ..., state_{child_{arity(i)}(i)});
            i ← i − p;
        end
        s ← s + 1;
    end
end
```

As it's just a simple loop, assuming that Δ evaluates in $O(1)$, the complexities are

$$
\begin{aligned}
T(n,p) =& O\left(\frac{n}{p}\right), \\
C(n,p) =& p \cdot O\left(\frac{n}{p}\right) = O(n), \\
W(n,p) =& O(n).
\end{aligned}
$$

2.6.5 Complexity analysis

The algorithm consists of ETT, dept-mod-k sort, step computation, and state computation. All of those were analyzed above. The complexities depend on

the ranking algorithm used. For the non-work-optimal ranking, they're

$$\begin{aligned}
T(n,p) =& O\left(\frac{n \cdot log_2\ n}{p}\right) + O\left(\frac{n}{p} + log_2\ p\right) + O(n)\,p = O\left(\frac{n \cdot log_2\ n}{p}\right),\\
S(n,p) =& \frac{O(n)}{O\left(\frac{n \cdot log_2\ n}{p}\right)} = O\left(\frac{p}{log_2\ n}\right),\\
C(n,p) =& p \cdot O\left(\frac{n \cdot log_2\ n}{p}\right) = O(n \cdot log_2\ n),\\
W(n,p) =& O(n \cdot log_2\ n) + O(n) = O(n \cdot log_2\ n).
\end{aligned}$$

For the work-optimal ranking, they're

$$\begin{aligned}
T(n,p) =& O\left(\frac{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}{p}\right) + O\left(\frac{n}{p} + log_2\ p\right) + O\left(\frac{n}{p}\right)\\
=& O\left(\frac{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}{p}\right),\\
S(n,p) =& O\left(\frac{n \cdot p}{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}\right),\\
C(n,p) =& p \cdot O\left(\frac{p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n}{p}\right)\\
=& O\left(p \cdot log_2\ p \cdot log_2^2\ n + n \cdot log_2\ n\right),\\
W(n,p) =& O(n \cdot log_2\ p) + O(n) = O(n \cdot log_2\ p).
\end{aligned}$$

2.6.6 APRAM modification

As in APRAM, the processors are not working in synchrony, all of the parallel regions have to be synchronized during potential conflicts. The algorithm is designed for EREW PRAM, i.e. it's designed to be conflict-less. There are some conflicts present in the implementation, all are described in appropriate sections. All of them are negligible and do not conflict with the nature of EREW PRAM.

The only reason for step pre-computation is to be able to tell when the synchronous processors should stall. For asynchronous processors, a synchronization barrier could be used instead. Then, the whole step of computation can be skipped. Hence the modified algorithms for DFTA run 25 and state computation 24.

Algorithm 24: compute state (APRAM)

Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, tree $t = (labels, children, root)$ of the size n, depth-mod-k order $dmk_1, \ldots, dmk_n$

Output: states $state_1, \ldots, state_n$

do in parallel
- $s \leftarrow 1$;
- $i \leftarrow n - pid$;
- **for** $g \leftarrow dmk_n$ **downto** 1 **do**
  - **while** $dmk_i = g$ **do**
    - $state_i \leftarrow \Delta_{labels_i}(state_{child_1(i)}, \ldots, state_{child_{arity(i)}(i)})$;
    - $i \leftarrow i - p$;
  - **end**
  - synchronize processors;
- **end**

end

Algorithm 25: Parallel run of the k-local DFTA (APRAM)

Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, tree $t = (labels, children, root)$ of the size n

Output: states $state_1, \ldots, state_n$

Auxiliary: depths $depth_1, \ldots, depth_n$, depth-mod-k order $dmk_1, \ldots, dmk_n$

get depths using ETT (**in:** t, **out:** *depth*);
depth-mod-k sort (**in:** t, *depth*, **out:** *dmk*);
$\forall i \in \hat{n} : \ state_i \leftarrow 0$;
compute state (**in:** A, t, *dmk*, **out:** *state*);
compute state (**in:** A, t, *dmk*, **out:** *state*);

2.6.7 GPGPU modification

As SIMT architecture is closer to the true nature of EREW PRAM, the steps cannot be omitted completely. Instead of step computation, the boundaries of each group are computed. Then, the individual groups are processed one by one without any additional stalling.

2.6.8 Implementations

As the algorithm is still relatively new, there are no practical implementations. The only existing implementation the author of this thesis is aware of, is

implementation done by the author himself. [2]

Chapter 3

Implementation

This chapter starts with a short description of parallelization libraries. The implementation notes of individual structures and algorithms follow. The CPU and GPU implementation of individual parts is next to each other for easier comparison.

3.1 Libraries

3.1.1 CPU

POSIX threads are considered a parallelization standard. Most of the other libraries use it as a reference or even wrap it, providing a convenient interface. It offers basic thread manipulation and synchronization primitives.

STL provides its OO interface for POSIX threads. Even though it's powerful and convenient, it's not sufficient for the purpose of this thesis as it lacks control over number of processors utilized.

TBB is considered an industry standard for parallelization. Besides parity with POSIX, an implementation of some algorithms is provided as well. Workload balancing is part of the implementation.

OpenMP has a similar, if not the same, status as TBB. Rather than classes and functions, preprocessor directives and core language features are utilized to interface with the user. It provides control over the number of used processors. This is the main strength of this library and the main reason why implementation will use that.

3.1.2 GPU

OpenCL provides all the required functionality to create GPGPU applications. It's an open standard for GPGPU computation. It's considered to be an industry standard for multiplatform GPGPU.

CUDA is a proprietary NVIDIA GPGPU API. It's on par with OpenCL regarding the provided functionalities. As it's a proprietary standard of NVIDIA, it's well-optimized for NVIDIA GPUs.

Thrust is a C++ extension of CUDA, that implements many algorithms and structures for GPGPU computing. As the NVIDIA GPUs are expected to be used with the implementation, this library will be used for it.

3.2 Structures

3.2.1 Array

3.2.1.1 CPU

As most implementations of arrays have sequential initialization & copy, custom implementation is implemented. The implementation can be found in *b5:type.par_vector* module, named *b5::type::par_vector*. It conforms to *Container* named requirement.

As the addition and removal of the elements after initialization is not required, the implementation doesn't support that.

All the operations are implemented with either $T\left(n,p\right)=O\left(\frac{n}{p}\right)$ or $T\left(n,p\right)=O\left(1\right)$.

3.2.1.2 GPU

The Thrust library provides *thrust::host_vector* & *thrust::device_vector* as CPU and GPU counterparts of each other. It supports the transition of the data from CPU to GPU and vice versa.

3.2.2 Tree

3.2.2.1 CPU

The implementation can be found in *b5:type.tree*, named *b5::type::tree*. *b5::type::par_vector* is used as array implementation.

3.2.2.2 GPU

The implementation is split into classes *b5::type::tree::host* & *b5::type::tree::device*. Those use *thrust::host_vector* & *thrust::device_vector* as array implementation.

3.2.3 Arc

3.2.3.1 CPU

The implementation can be found in *b5:algorithm.ett*, named *b5::algorithm::arc*.

3.2.3.2 GPU

As it's POD, there's only one implementation common for both, host and device. It's implemented in class *b5::algorithm::arc*.

3.2.4 DFTA

3.2.4.1 CPU

The implementation can be found in *b5:type.dfta*, named *b5::type::dfta*. The states are represented by the maximal state. The final states are represented by an *std::unordered_set* to be more memory-efficient than using *b5::type::par_vector<bool>*. The transition function is represented by *std::unordered_map*, which conforms to 23 assumption that it's evaluated in $O(1)$.

The symbol-specific sections of the transition functions are represented by *b5::type::dfta::transitions*. To ease the implementation with various supported

arities, the table is flattened.

3.2.4.2 GPU

The implementation is split into classes *b5::type::dfta::host* & *b5::type::dfta::device*. The transition function is flattened fully.

3.3 Reduction and Scan

3.3.1 Reduction

3.3.1.1 CPU

The OpenMP implementation of reduction is utilized.

3.3.1.2 GPU

As the reduction is not used in GPU implementation, it's not present.

3.3.2 Inclusive scan

3.3.2.1 CPU

As the implementation provided by the STL doesn't provide control over the number of processors used, there's a custom implementation. It can be found in *b5:algorithm.scan.inclusive*, named *b5::algorithm::scan::inclusive*. It implements the Blelloch algorithm 8.

3.3.2.2 GPU

The *thrust::inclusive_scan* is used in GPU implementation.

3.3.3 Exclusive scan

3.3.3.1 CPU

The exclusive counterpart of the inclusive scan can be found in *b5::algorithm.scan.exclusive*, named *b5::algorithm::scan::exclusive*.

3.3.3.2 GPU

The exclusive counterpart is implemented as *thrust::exclusive_scan*.

3.4 Lists

3.4.1 Linked list

3.4.1.1 CPU

The implementation of the linked list using the successor array can be found in *b5:type.linked_list*, named *b5::type::linked_list*.

3.4.1.2 GPU

The implementation is split into classes *b5::type::ll::host* & *b5::type::ll::device*. Both of them use the successor array.

3.4.2 List ranking

3.4.2.1 CPU

The algorithm 13 is implemented.

The n-coloring can be found in *b5:algorithm.ranking.n_coloring* and is named *b5::algorithm::ranking::n_coloring*.

The 6-coloring using DCT (algorithm 11) can be found in *b5:algorithm.ranking.six_coloring* and is named *b5::algorithm::ranking::six_coloring*.

The 3-coloring (algorithm 12) can be found in *b5:algorithm.ranking.three_coloring* and is named *b5::algorithm::ranking::three_coloring*.

The ranking itself can be found in *b5:algorithm.ranking* and is named *b5::algorithm::list_ranking*. 0 is used instead of 1 as initial rank as it's used to index arrays.

3.4.2.2 GPU

As the algorithm 13 won't work well with GPUs, due to its nature, the algorithm 10 is implemented instead. It's named *b5::algorithm::list_ranking*.

3.5 Euler Tour Technique

3.5.1 CPU

The implementation of the algorithm 14 can be found in *b5:algorithm.ett*, named *b5::algorithm::euler_tour*.

The index of the first arc for each node is precomputed first. This allows for a parallel arc creation.

The creation of the downgoing and upgoing arcs is split for simplification.

3.5.2 GPU

The implementation of the algorithm 14 is implemented as *b5::algorithm::euler_tour*. It's similar to the CPU implementation.

3.6 Parentheses matching

3.6.1 CPU

The implementation can be found in *b5:algorithm.par_match*, named *b5::algorithm::par_match*. It implements work-optimal parentheses matching 18.

3.6.2 GPU

The algorithm 17 is used instead as it's more SIMT-friendly. *thrust::stable_sort_by_key* is used as a stable sort implementation. It's implemented as *b5::algorithm::par_match*.

3.7 Run of k-local DFTA

3.7.1 Run

3.7.1.1 CPU

The run can be found in *b5:algorithm.dfta.run* as *b5::algorithm::dfta::run*.

It's associated with the state computation as its dependency. It's named *b5::algorithm::dfta::update_states* but is not exposed.

3.7.1.2 GPU

The run is implemented as *b5::algorithm::run*.

It's associated with the state computation as its dependency. It's named *b5::algorithm::dfta::update_states* but is not exposed.

3.7.2 Preprocess

3.7.2.1 CPU

The preprocess combines the get depths algorithm 15 with the depth-mod-k algorithm 21.

They're implemented as *b5::algorithm::dfta::get_depths* & *b5::algorithm::dfta::depth_mod_k_sort* but are not exposed. Instead, *b5::algorithm::dfta::preprocess* could be used. All of them may be found in *b5:algorithm.dfta.preprocess*.

3.7.2.2 GPU

The preprocess is similar to the CPU implementation, including names and exposition.

Chapter 4

Testing

This chapter starts with a system test of created implementations. The time measurement of those implementations follows. It's closed by a comparison of both implementations.

4.1 System test

The system test is not using any standardized framework. Instead, the set of pre-defined trees and DFTAs with known results is used. One of those trees and DFTAs is provided with this thesis as an example.

Figure 4.1 depicts a pattern described by example DFTA and 4.2 example tree with states after evaluation.

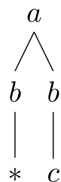

Figure 4.1: Example pattern

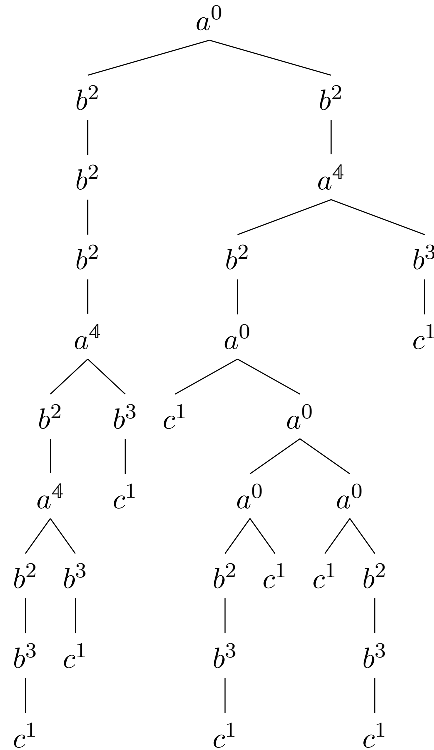

Figure 4.2: Evaluated example tree

4.2 Time measurements

4.2.1 Methodology

The preprocessing time was measured separately from the time of the run, as it's a common use case to preprocess a tree for multiple patterns.

The real-time was measured as an approximation of the execution time, neglecting time scheduling.

Multiple data sets were tested with each algorithm described in 4.2.3.

4.2.2 Hardware

All tests were executed with the following hardware.

- Intel Core i7 12850HX with 24 vCPUs,

n	CPU			GPU		
	preproc.	*run*	$\sum$	*preproc.*	*run*	$\sum$
2^{10}	6.20 ms	220 μs	6.43 ms	597 ms	1.60 ms	599 ms
2^{12}	8.49 ms	825 μs	9.31 ms	668 ms	2.05 ms	670 ms
2^{14}	11.6 ms	3.46 ms	15.1 ms	672 ms	2.56 ms	674 ms
2^{16}	34.6 ms	8.75 ms	43.4 ms	674 ms	2.97 ms	677 ms
2^{18}	133 ms	34.2 ms	168 ms	676 ms	3.01 ms	679 ms
2^{20}	482 ms	144 ms	625 ms	693 ms	6.51 ms	700 ms
2^{22}	3.51 s	687 ms	4.19 s	913 ms	15.7 ms	928 ms
2^{24}	11.6 s	4.06 s	15.6 s	1.95 s	53.8 ms	2.01 s
2^{26}	47.0 s	14.7 s	1.03 min	6.53 s	241 ms	6.77 s
2^{28}	3.57 min	1.12 min	4.70 min	24.0 s	994 ms	25.0 s
2^{30}	14.8 min	5.24 min	20.0 min	1.54 min	3.93 s	1.61 min

Figure 4.3: Results

- 64 GiB of RAM, 4800 MHz
- NVIDIA GeForce RTX 3080 Ti

4.2.3 Data

The testing data consisted of randomly generated trees and DFTAs.

The actual k-locality of DFTA was neglected as it does not affect the measured time. Various DFTAs were used mainly to mitigate variance in results. ks were chosen in range 2 through 24.

The processor count wasn't limited in any way for CPU implementation, as it doesn't make sense to limit CPUs when competing with GPUs.

The tested tree sizes were in the range of 2^10 through 2^30. Each tree's run time was measured multiple times, and multiple trees of the same size were tested to mitigate variance.

Random test data were generated using Algorithms Library Toolkit.[11]

4.2.4 Results

Results are presented in the figure 4.3. The results presented are average of individual results. Charts 4.4 present results visually.

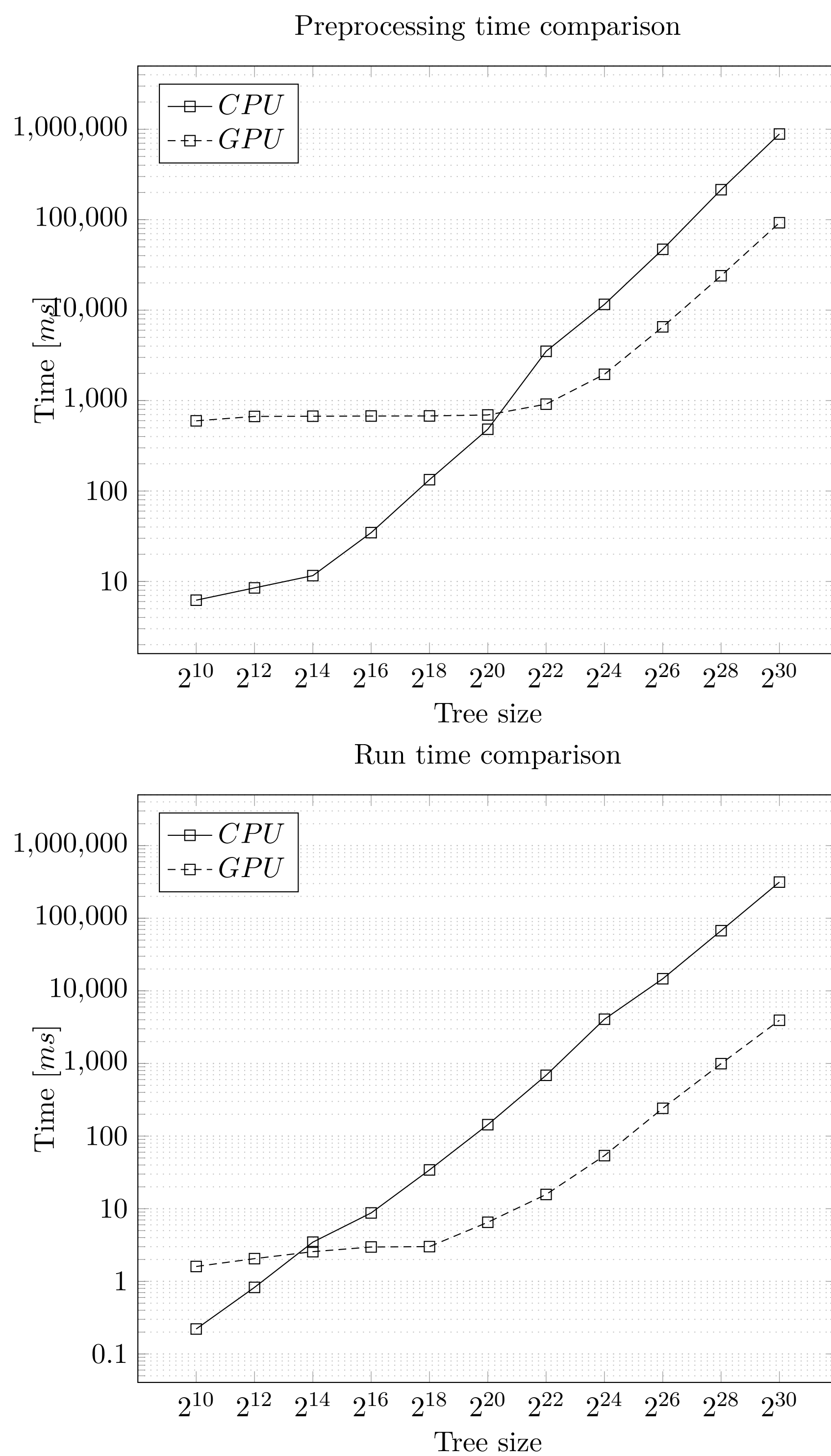

Figure 4.4: Time comparisons

As expected, the charts show that GPU implementation achieves significantly higher speed than CPU implementation. This occurs only with bigger trees, as the GPU synchronization's relatively big overhead causes the GPU implementation to be slower with smaller inputs. As it's somewhat expensive to pass data between CPU and GPU, the benefit of GPU is expected to be mitigated by the overhead with small trees.

The size of approximately 1.5 million of nodes seems to be the boundary at which GPU speedup overcomes its overhead. The CPU implementation can be up to 100 times faster for smaller trees. With bigger trees, the GPU needs approximately $\frac{1}{10}$ of execution time required by the CPU implementation.

The linear increase of execution time with tree size is most probably caused by the full utilization of threads. As there are not enough threads to split all the work, the waiting time that grows linearly with tree size is defining the overall trend. The threads are depleted much sooner for CPU implementation than the GPU implementation. That is caused by the fact that GPUs can have thousands of threads available, while typical CPUs have less than a hundred threads available. The point at which GPU threads are depleted can be seen in the chart as a point where the constant trend of the time complexity changes to a linear trend.

The main issue of the measurement is the limited number of threads available. The algorithm expects the number of processors to scale with the input size. That's why the declared complexity is not visible in charts.

The speed-up of the GPU is much more significant in a pure run with preprocessed data. The GPU outperforms the CPU up to 100 times. As the preprocessing is much more expensive than running itself, it's worth preprocessing the tree only once when possible. The tree should be preprocessed for the biggest k of all DFTAs it should be used with. This way, it can be run with all of the DFTAs without further processing, causing much more significant speed-up than parallelization itself.

As is the results seem very promising, as the GPU implementation was able to keep great execution time until the threads were depleted. Further testing may prove the feasibility of the implemented algorithm for pattern matching.

Conclusions and Future work

This thesis was about implementing a parallel run of k-local DFTA for two different architectures.

For APRAM, the original algorithm was optimized and implemented work optimally. The implementation was tested and measured experimentally.

Besides APRAM, the original algorithm was adapted to SIMT architecture, which is similar to the original EREW PRAM, yet different. The implementation for SIMT isn't work-optimal. Some of the work-optimal algorithms presented are not compatible with SIMT architecture and thus cannot be used. The algorithm was implemented, favoring the overall speed, sacrificing some of the optimality. The implementation was tested and measured experimentally as well.

Both implementations were compared to each other. As expected, the GPU implementation outperformed the CPU one. The testing was performed with all processors available to achieve maximal speed.

The implementation proves the applicability and feasibility of the algorithm for other architectures than the originally intended one.

Even though the results are very promising, further testing is required to reach any conclusions regarding overall performance and scalability.

Future work

Further tests with many more processors will be required to test scalability properly. To achieve that, modification for a NUMA model would be required

as even testing performed in this thesis is hitting the limits of usual UMA models.

Besides migrating to distributed memory, there are many possible optimizations for current implementations.

The APRAM implementation was made to optimize/modify the original EREW PRAM algorithm slightly. Further relaxation of requirements given by that architecture can be applied to optimize the algorithm for asynchronous systems.

The SIMT implementation currently doesn't solve data locality to the depth. There's a lot of space for improvement in that regard. By improving data locality and optimizing data transfers between RAM, VRAM, and caches, a lot of additional acceleration can be achieved, benefiting from GPGPU much more than in the current thesis.

Even though the NUMA architectures should be prioritized for further work, it's possible to adapt the algorithm to other UMA architectures to prove its applicability regardless of the architecture.

Bibliography

[1] Plachý, v.; Janoušek, J. *On Synchronizing Tree Automata and Their Work–Optimal Parallel Run Usable for Parallel Tree Pattern Matching*. Springer International Publishing, 2020, ISBN 9783030389192, p. 576–586, doi:10.1007/978-3-030-38919-2_47. Available from: http://dx.doi.org/10.1007/978-3-030-38919-2_47

[2] Borový, M. Implementation of parallel algorithm for run of k-local tree automata. *Czech Technical University in Prague*, 2021. Available from: https://dspace.cvut.cz/handle/10467/95439

[3] Rahman, M. S. *Basic Graph Theory*. Springer International Publishing, 2017, ISBN 9783319494753, doi:10.1007/978-3-319-49475-3. Available from: http://dx.doi.org/10.1007/978-3-319-49475-3

[4] Tvrdík, P. *Parallel algorithms and computing*. Praha: Vydavatelství ČVUT, 2003, ISBN 80-01-02824-0.

[5] Flynn, M. Very high-speed computing systems. *Proceedings of the IEEE*, volume 54, no. 12, 1966: p. 1901–1909, ISSN 0018-9219, doi:10.1109/proc.1966.5273. Available from: http://dx.doi.org/10.1109/PROC.1966.5273

[6] Hillis, W. D.; Steele, G. L. Data parallel algorithms. *Communications of the ACM*, volume 29, no. 12, Dec. 1986: p. 1170–1183, ISSN 1557-7317, doi:10.1145/7902.7903. Available from: http://dx.doi.org/10.1145/7902.7903

[7] Blelloch, G. E. Prefix sums and their applications. Technical report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[8] Cole, R.; Vishkin, U. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, volume 81, no. 3, June 1989: p. 334–352, ISSN 0890-5401, doi:10.1016/0890-5401(89)90036-9. Available from: http://dx.doi.org/10.1016/0890-5401(89)90036-9

[9] Tarjan, R.; Vishkin, U. Finding biconnected componemts and computing tree functions in logarithmic parallel time. In *25th Annual Symposium on Foundations of Computer Science, 1984.*, IEEE, 1984, doi:10.1109/sfcs.1984.715896. Available from: http://dx.doi.org/10.1109/sfcs.1984.715896

[10] Levcopoulos, C.; Petersson, O. Matching parentheses in parallel. *Discrete Applied Mathematics*, volume 40, no. 3, Dec. 1992: p. 423–431, ISSN 0166-218X, doi:10.1016/0166-218x(92)90011-x. Available from: http://dx.doi.org/10.1016/0166-218X(92)90011-X

[11] Faculty of Information Technology, Czech Technical University in Prague. Algorithms Library Toolkit. 2021-04-03, version 0.0.0.r1109.gc0ac370eb. Available from: https://alt.fit.cvut.cz/

[12] OpenMP. 2020-11-17, version 5.1. Available from: https://www.openmp.org/

Appendix A

Acronyms

APRAM Asynchronous Parallel Random Access Machine

BFS Breadth-First Search

CRCW Concurrent Read Concurrent Write

CREW Concurrent Read Exclusive Write

CUDA Compute Unified Device Architecture

DCT Deterministic Coin Tossing

DFS Depth-First Search

DFTA Deterministic finite tree automaton

EREW Exclusive Read Exclusive Write

ETT Euler's Tour Technique

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

NUMA Non-Uniform Memory Access

PRAM Parallel Random Access Machine

RAM Random Access Machine

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SISD Single Instruction Single Data

TBB Thread Building Blocks

UMA Uniform Memory Access

Appendix B

Symbols

$\mathbb{N}_0$ set of natural numbers

$\mathbb{R}^+$ set of positive real numbers

$\widehat{x}$ set $\{1, 2, \ldots, x\}$

$\bigoplus_{i=a}^{b} x_i$ shorthand for $x_a \oplus \cdots \oplus x_b$.

Appendix C

User manual

C.1 Prerequisitities

C.1.1 CPU

- CMake v3.28 or newer
- OpenMP v4.5 or newer

C.1.2 GPU

- CMake v3.28 or newer
- CUDA v12.4 or newer
- Compatible thrust version

C.2 Compilation

To compile this thesis, follow these steps:

1. Navigate to the desired build directory $< BUILD_DIR >$.
2. Type *cmake* $< SRC_DIR >$ in the terminal.
3. Type *cmake –build* $< BUILD_DIR >$ in the terminal.

C.3 Usage

The compilation of the source codes produced $< BUILD_DIR > /lib/dftaOmp_lib.a$ and $< BUILD_DIR > /bin/dftaOmp_demo$.

To run the demo program type $< BUILD_DIR > /bin/dftaOmp_demo < TREE_BINARY > < DFTA_BINARY > k$. To run the time measuring type $< BUILD_DIR > /bin/dftaOmp_measure < TREE_BINARY > < DFTA_BINARY > k$.

For the GPU implementation just replace $dftaOmp$ with $dftaCuda$.

Appendix D

Contents of the enclosed medium

```
 readme.txt ....................... the file with CD contents description
 cuda ................................ the GPU implementation sources
 omp .................................. the CPU implementation sources
   bin
     example.dfta ........................... example DFTA binary
     example.dfta.desc .................. example DFTA description
     example.out.desc .................. example output description
     example.tree ............................. example tree binary
     example.tree.desc .................... example tree description
 text ......................................... the thesis text directory
   thesis.pdf ......................... the thesis text in PDF format
```