FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Implementation of parallel algorithm for run of k-local tree automata
Student:	Milan Borový
Supervisor:	Ing. Štěpán Plachý
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

Study the PRAM algorithm for a parallel run of k-local finite tree automata. [1]
Explore existing libraries and frameworks for parallelization.
After an agreement with the supervisor implement the algorithm with the use of suitable technology.
Test your implementation and compare the speed of the calculation with sequential algorithm 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 4 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor’s thesis

Implementation of parallel algorithm for run of k-local tree automata

Department of Theoretical Computer Science

Supervisor: Ing. Štěpán Plachý

May 13, 2021

Acknowledgements

I would like to thank my parents for the tremendous support they are giving to me during my study.

I would also like to thank my thesis supervisor Ing. Štěpán Plachý for all the help and valuable feedback he gave me during creation of this bachelor's thesis.

Declaration

Citation of this thesis

Borový, Milan. *Implementation of parallel algorithm for run of k-local tree automata.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstract

This thesis deals with k-local deterministic finite tree automata (DFTA) which are important for tree pattern matching. There exists a work-optimal parallel algorithm for a run of k-local DFTA on EREW PRAM. This algorithm will be implemented, experimentally measured and compared with the sequential algorithm in this thesis.

Keywords k-locality, deterministic finite tree automaton, parallel run implementation, EREW PRAM, OpenMP

Abstrakt

Tato práce se zabývá k-lokálními deterministickými konečnými stromovými automaty (DKSA), které hrají důležitou roli při hledání vzorů ve stromových strukturách. Existuje pracovně optimální paralelní algoritmus pro běh k-lokálních DKSA na výpočetním modelu EREW PRAM. Tento algoritmus bude implementován, experimentálně změřen a porovnán se sekvenčním algoritmem v této práci.

Klíčová slova k-lokálnost, deterministický konečný stromový automat, implementace paralelního běhu, EREW PRAM, OpenMP

Contents

List of Figures

Introduction

There are many problems that, even though effective algorithms are known to solve them, are incomputable for large enough inputs. That is where parallel algorithms comes in play since increasing performance of computers is unsustainable.

One of such problems is problem of evaluating a tree. Effective algorithm for this runs in linear time, but running time could be improved asymptotically presuming enough processors are available without pushing their physical limits.

Standard computation model for trees and tree languages is finite tree automaton (FTA). This thesis aims to implement parallel run of k-local deterministic finite tree automaton (DFTA), which is special kind of FTA, that is especially important for tree pattern matching since pattern of depth k can be matched by k-local DFTA.

Work-optimal algorithm for parallel run of k-local DFTA was created and theoretically described on computation model EREW PRAM[1]. This thesis will implement this algorithm alongside with all support functions needed and execution time of this implementation will be experimentally measured and compared to the sequential algorithm for run of k-local DFTA.

Goals

The first goal of this thesis is to study all the needed theory and existing algorithms for run of k-local DFTA and all other needed algorithms, their analysis and analysis of their existing implementations.

The second goal of this thesis is to implement parallel run of k-local DFTA and all the needed algorithms.

The third and last goal of this thesis is to measure and compare execution times of the created implementations.

Chapter 1

Theory

In this chapter all the needed theory will be presented. Starting with basics of graph theory[2], tree languages[1] and algorithm complexity[3] through computation models[3] to definition of individual problems that are needed to be solved to run k-local DFTA in parallel.

All problems defined in this chapter will be analysed in chapter 2.

Notation in this chapter will be similar to the notation in [3] 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
- *$E = \{\{u, v\} : u, v \in V \wedge u \neq v\}$ is a set of* edges

Definition 1.2 *Let $G = (V, E)$ be a graph. Degree of a vertex $u \in V$ is*

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

Definition 1.3 Path *$x_1 - x_n$ in the graph $G = (V, E)$ is an unempty sequence of vertices $x_1 \dots x_n$, where*

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

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

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

Definition 1.4 Connected graph *is a graph $G = (V, E)$ where for each pair of vertices $u, v \in V, u \neq v$ exists a path $u - v$.*

Definition 1.5 Cycle *in the graph $G = (V, E)$ is a 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 1.6 Acyclic graph *is a graph that doesn't contain any cycle in it.*

Definition 1.7 Oriented graph *is a pair $G = (V, E)$, where*

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

Definition 1.8 *Let $G = (V, E)$ be an oriented graph. In-degree of a vertex $u \in V$ is*

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

Out-degree of a vertex $u \in V$ is

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

Definition 1.9 Oriented path $x_1 - x_n$ *in the oriented graph $G = (V, E)$ is an unempty sequence of vertices $x_1 \dots x_n$, where*

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

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

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

Definition 1.10 Weakly connected graph *is an oriented graph $G = (V, E)$ where for each pair of vertices $u, v \in V$, $u \neq v$ exists a path $u - v$ (ignoring edges orientation).*

Definition 1.11 Strongly connected graph *is an oriented graph $G = (V, E)$ where for each pair of vertices $u, v \in V, u \neq v$ exists an oriented path $u - v$.*

1.1.2 Tree

Definition 1.12 Tree *is acyclic and connected graph.*

Definition 1.13 Rooted tree *is a tree, where one vertex has been designated the* root *of the tree.*

Definition 1.14 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 1.15 *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 the vertex u is called* parent *of the vertex v denoted by $parent(v)$ and the vertex v is called* child *of the vertex u. Set of childs of vertex u is denoted by $childs(u)$.*

Definition 1.16 *Let $G = (V, E)$ be a tree and u any vertex of that tree. Then $\forall v, w \in childs(u)$ such that $v \neq w$ the vertex v is called* sibling *of vertex w. Set of siblings of vertex v is denoted by $siblings(v)$.*

Definition 1.17 *Let $G = (V, E)$ be a tree and $u, v \in V$ two vertices of that tree. Vertex u is called an* ancestor *of the vertex v if*

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

Set of ancestors of the vertex v is denoted by $ancestors(v)$.

Definition 1.18 *Let $G = (V, E)$ be a tree and $u, v \in V$ such that $u \in ancestors(v)$. Then v is called* descendant *of the vertex u. Set of descendants of the vertex u is denoted by $descendants(u)$.*

Definition 1.19 Subtree *$G' = (V', E')$ of the tree $G = (V, E)$ is a tree such that $V' \subseteq V$ and*

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

Definition 1.20 *Let $G = (V, E)$ be a tree and $u \in V$ vertex of that tree.* Arity *of vertex u is*

$$arity(u) = |childs(u)|$$

Definition 1.21 *Let $G = (V, E)$ be a tree and $u \in V$ vertex of that tree. u is called a* leaf *if $arity(u) = 0$. Vertex that is not a leaf is called* inner vertex. *Set of leaves is denoted by $leaves(G)$.*

Definition 1.22 Ordered tree *$G = (V, E)$ is a tree such that $\forall u \in V$ $childs(u)$ is ordered set.*

Definition 1.23 *Let $G = (V, E)$ be an ordered tree and $u, v \in V$ vertices of that tree such that $v \in childs(u)$. v is called* i-th child *of vertex u if v is on i-th position in ordered set $childs(u)$ denoted by $child_i(u)$.*

1.1.3 Tree language

Definition 1.24 Alphabet *Σ is a finite set of symbols.*

Definition 1.25 String *over the alphabet Σ is sequence of symbols $a_1 \ldots a_n$ from alphabet Σ.*
ϵ denotes empty string.
$|a_1 \ldots a_n| = n$ denotes length of string $a_1 \ldots a_n$.
Set of strings *over the alphabet Σ is denoted by Σ^*.*
$|w|_a$ denotes number of occurences of symbol $a \in \Sigma$ in a string $w \in \Sigma^*$.
w_i denotes i-th symbol of string $w \in \Sigma^*$

Definition 1.26 Concatenation *of strings over the alphabet Σ is mapping $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that*

$$(a_1 \ldots a_n) \cdot (b_1 \ldots b_n) = (a_1 \ldots a_n \; b_1 \ldots b_n)$$

Definition 1.27 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 1.28 Prefix *of string w over the alphabet Σ is such string p that*

$$(\exists t \in \Sigma^*) w = p \cdot t$$

Definition 1.29 Postfix *of string w over the alphabet Σ is such string p that*

$$(\exists t \in \Sigma^*) w = t \cdot p$$

Definition 1.30 Ranked alphabet *is a pair* $\mathcal{F} = (\Sigma, rank)$*, where*

- Σ *is alphabet.*
- *rank is function* $\Sigma \to \mathbb{N}_0$ *which for each symbol from the alphabet* Σ *assigns a natural number as its* rank*.*

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

Definition 1.31 *Set of terms* $T(\mathcal{F}, \mathcal{X})$ *over the ranked alphabet* $\mathcal{F}$ *and set of constants called* variables $\mathcal{X}$*, where* $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$*, is the smallest set defined by*

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

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

Definition 1.32 *Term* $t \in T(\mathcal{F}, \mathcal{X})$ *where* $\mathcal{X} = \emptyset$ *is called a* ground term *over the ranked alphabet* $\mathcal{F}$*. Set of ground terms over the ranked alphabet* $\mathcal{F}$ *is denoted by* $T(\mathcal{F})$*.*

Theorem 1.1 *Term* $t = f(t_1, \ldots, t_r) \in T(\mathcal{F}, \mathcal{X})$ *is equivalent to an ordered tree* $G = (V, E)$ *such that*

$$V = \{node(t)\} \cup \left(\bigcup_{i=1}^{r} V'_i \right)$$

$$E = \{\{node(t), root(G'_i)\} : \forall i \in \widehat{r}\} \cup \left(\bigcup_{i=1}^{r} E'_i \right)$$

where $node(t)$ *denotes node representing term* t*,* $G'_i = (V'_i, E'_i)$ *is a tree equivalent to the term* t_i *and set of childs of* $node(t)$ *is ordered with respect to* i *(i.e.* $child_i(node(t)) = node(t_i)$*).* $label(node(t))$ *denotes symbol* f *(i.e. top-level symbol).*

Definition 1.33 Ground substition *sigma over the set of the variables $\mathcal{X}$ and the ranked alphabet $\mathcal{F}$ is mapping $\mathcal{X} \to T(\mathcal{F})$ which for each variable $x \in \mathcal{X}$ assigns a ground term $t \in T(\mathcal{F})$.*

Definition 1.34 Subterm $t|_p$ *of a term $t \in T(\mathcal{F}, \mathcal{X})$ at position $p \in \mathbb{N}_0^*$ is defined by the following:*

- $t|_\epsilon = t$
- *if* $t = f(t_1, \ldots, t_r)$ *then* $t|_{ip'} = t_i|_{p'}$ *for* $i \leq r$

Set of subterms of term t is denoted by $subterms(t)$.

Definition 1.35 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 1.36 Deterministic finite tree automaton *(DFTA) over a ranked alphabet $\mathcal{F}$ is a quadruple $A = (Q, \mathcal{F}, Q_f, \Delta)$, where*

- *Q is a finite set of states.*
- *$\mathcal{F}$ is a ranked alphabet.*
- *$Q_f \subseteq Q$ is a set of final states.*
- *Δ is a transition function of type $f(q_1, \ldots, q_r) \to q$, where*
  - $f \in \mathcal{F}_r$
  - $q_1, \ldots, q_r, q \in Q$

Definition 1.37 Extended transition function *of DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ is a mapping $\widehat{\Delta} : T(\mathcal{F}) \to Q$ defined as follows:*

- $(\forall f \in \mathcal{F}_0)\widehat{\Delta}(f) = \Delta(f)$
- $(\forall r > 0)(\forall f \in \mathcal{F}_r)(\forall t_1, \ldots, t_r \in T(\mathcal{F}))$

$$\widehat{\Delta}(f(t_1, \ldots, t_r)) = \Delta(f(\widehat{\Delta}(t_1), \ldots, \widehat{\Delta}(t_r)))$$

Definition 1.38 *A ground term $t \in T(\mathcal{F})$ is* accepted *by the DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ if $\widehat{\Delta}(t) \in Q_f$.*

1.1.5 k-local tree automaton

Definition 1.39 *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 if*

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

Definition 1.40 Minimal variable depth *is a function $MVD : T(\mathcal{F}, \mathcal{X}) \to \mathbb{N}_0$ such that*

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

Definition 1.41 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 1.42 Time complexity *$T_A^K(n)$ of algorithm A solving problem K for input of size n is a computer time required to run that algorithm.*

Definition 1.43 Sequential lower bound *$SL^K(n)$ of problem K is function such that*

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

Definition 1.44 *Algorithm A is the* best known *sequential algorithm for solving problem K if there's not any known algorithm B such that*

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

Definition 1.45 Sequential upper bound $SU^K(n)$ *of problem K worst-case time complexity of the best known sequential algorithm solving K.*

Definition 1.46 *Algorithm A is called* optimal *sequential algorithm for solving problem K if*

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

1.2.2 Parallel Complexity

Definition 1.47 Parallel time complexity $T_A^K(n,p)$ *of parallel algorithm A solving problem K for input of size n using p processors is a total time elapsed from the beginning of execution until the last processor finishes.*

Definition 1.48 Parallel speedup *of parallel algorithm A solving problem K for input of size n using p processors is*

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

Definition 1.49 Parallel cost *of algorithm A solving problem K for input of size n using p processors is*

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

Definition 1.50 *Algorithm A is called* cost-optimal *if*

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

Definition 1.51 Synchronous parallel work *of a synchronous algorithm A solving problem K for input of size n using p processors in τ parallel steps where p_i denotes number of active processors in step i is*

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

Definition 1.52 Asynchronous parallel work *of an asynchronous algorithm A solving problem K for input of size n using p processors where T_i denotes number of steps executed by i-th processor is*

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

Definition 1.53 *Algorithm A is called* work-optimal *if*

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

Definition 1.54 Parallel efficiency *of algorithm A solving problem K for input of 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 in 2 groups.

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

This thesis is focused on shared-memory models, specifically on PRAM model. For more insights [3] is recommended.

Definition 1.55 Random Access Machine *(RAM) model is computation model consisting of a single processor with bounded number of registers, unbounded number of local memory cells with a user-defined program, read-only input tape and write-only output tape.*

Instruction set of processor contains instructions for simple data manipulation, comparisons, branching and basic arithmetic operations. Program is executed from first instruction until HALT instruction.

Definition 1.56 Parallel Random Access Machine *(PRAM) model is computation model consisting of multiple RAM processors $p_2, p_2, \ldots$ without input and output tapes and without local memory, all processors are connected to a shared memory with unbounded number of cells $M_1, M_2, \ldots$. Each processors p_i knows its index i. Each processor have constant-time access to any M_j unless there are access conflicts. All processors work synchronously and can communicate with each other only through writing to and reading from shared memory. p_1 has control role and starts execution of other processors. p_1 can halt only when other processors halted.*

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

Definition 1.57 Exclusive Read Exclusive Write *(EREW) PRAM model is PRAM submodel that doesn't allow 2 processors to access the same memory cell simultaneously.*

Definition 1.58 Concurrent Read Exclusive Write *(CREW) PRAM model is PRAM submodel that allows reading from a single memory cell to multiple processors simultaneously but only 1 processor may attempt to write on given cell at a time.*

Definition 1.59 Concurrent Read Concurrent Write *(CRCW) PRAM model is PRAM submodel that allows multiple processors to read simultaneously single cell and multiple processors may attempt to write on given cell at a time.*

Concurrent read operations don't affect each other but concurrent write operations don't have clear semantics and thus those must be defined.

Definition 1.60 Priority CRCW PRAM *model is CRCW PRAM submodel that has fixed distinct priorities and the processor with highest priority is allowed to complete write operation.*

Definition 1.61 Arbitrary CRCW PRAM *model is CRCW PRAM submodel that allows to 1 randomly chosen processor to complete write operation.*

Definition 1.62 Common CRCW PRAM *model is CRCW PRAM submodel that allows all processors to complete write operation but all processors must write the same value to the given memory cell and Common CRCW PRAM algorithms must ensure that this condition is satisfied.*

1.4 Reduction and Scan

Definition 1.63 *Let* $\mathbb{X} = \{x_1, \ldots, x_n\}$ *be a finite set of values and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.
Problem of finding $x_1 \oplus \cdots \oplus x_n$ *is called* reduction *and* $\oplus$ *is called* reduction operator.

Definition 1.64 *Let* $(x_i)_{i=1}^{n}$ *be a finite sequence of values from* $\mathbb{X}$ *and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.
Problem of finding a sequence $(y_i)_{i=1}^{n}$ *such that*

$$(\forall i \in \widehat{n}) y_i = x_1 \oplus \cdots \oplus x_i$$

is called inclusive scan.

Definition 1.65 *Let* $(x_i)_{i=1}^{n}$ *be a finite sequence of values from* $\mathbb{X}$ *and* $\oplus$ *an associative binary operator* $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.
Problem of finding a sequence $(y_i)_{i=1}^{n}$ *such that*

$$(\forall i \in \widehat{n}) y_i = x_1 \oplus \cdots \oplus x_{i-1}$$

is called exclusive scan.

Definition 1.66 *Let $(x_i)_{i=1}^n$ be a finite sequence of values from $\mathbb{X}$ and $\oplus$ an associative binary operator $\mathbb{X} \times \mathbb{X} \to \mathbb{X}$.*
Let $\mathcal{X}$ be a set of subsequences of $(x_i)_{i=1}^n$ where each subsequence contains consecutive run of elements from $(x_i)_{i=1}^n$, each 2 subsequences are disjunct and concatenation of all subsequences forms $(x_i)_{i=1}^n$.
Problem of finding an inclusive (or exclusive) scan of each subsequence from $\mathcal{X}$ is called segmented inclusive (or exlusive) scan.

1.5 Lists

Definition 1.67 Linked list *is a pair $L = (X, S)$ where*

- *X is an unempty set of* nodes
- *S is an injective* successor *function $X \to X$ such that*
  - *$(\exists! h \in X)(\forall x \in X) S(x) \neq h$, node h is called* head *and is denoted by $head(L)$.*
  - *$(\exists! t \in X) S(t)$ $is\ undefined$, node t is called* tail *and is denoted by $tail(L)$.*

Lemma 1.1 *Let $L = (X, S)$ be a linked list. Then*

$$X = \bigcup_{i=0}^{|X|-1} \{S^i(head(L))\}$$

Proof 1.1 *Let $L = (X, S)$ be a linked list.*
If $|X| = 1$ then $head(L) = S^0(head(L)) = tail(L)$ and

$$X = \bigcup_{i=0}^{1-1} \{S^i(head(L))\} = \{head(L)\}$$

If $|X| \geq 2$ then exists list $L' = (X', S')$ such that

$$X \setminus X' = \{tail(L)\}$$

and

$$(\forall x \in X')S'(x) = \begin{cases} S(x), \text{iff } S(x) \neq tail(L) \\ undefined, \ otherwise \end{cases}$$

and then

$$X = X' \cup \{S(tail(L'))\}$$

Recursive application of this results in

$$X = \{head(L'^{\cdots\prime\prime})\} \cup \{S(tail(L'^{\cdots\prime\prime}))\} \cup \{S(tail(L'^{\cdots\prime}))\} \cup \cdots \cup \{tail(L')\}$$

and if a linked list $L'^{\cdots\prime\prime} = (X'^{\cdots\prime\prime}, S'^{\cdots\prime\prime})$ has size $|X'^{\cdots\prime\prime}| = 1$ then

$$\begin{aligned} X = \{head(L'^{\cdots\prime\prime})\} & \cup \{S(head(L'^{\cdots\prime\prime}))\} \\ & \cup \{S(S(head(L'^{\cdots\prime\prime})))\} \\ & \cup \ldots \\ & \cup \{S^{|X|-1}(head(L'^{\cdots\prime\prime}))\} \end{aligned}$$

$$= \bigcup_{i=0}^{|X|-1} \{S^i(head(L))\}$$

$\square$

Definition 1.68 *Let $L = (X, S)$ be a linked list.* Independent set $I \subset X$ *of a linked list L is such subset of X that*

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

Lemma 1.2 *Independent set I of linked list L can be removed from L in parallel on EREW PRAM.*

Proof 1.2 *Let $L = (X, S)$ be a linked list and $I \subset X$ its independent set. Since for each pair of nodes $i, j \in I$ $S(i) \neq j$ there are no neighbouring nodes in the independent set I. Thus each node can be removed from L by relinking its predecessor to its successor (i.e.* iff *$S(i) \in I$ then $S(i) \leftarrow S(S(i))$) without any conflicts.* $\square$

Definition 1.69 *Let $L = (X, S)$ be a linked list and C a set of* colors *of size k. $X \cap C = \emptyset$. Problem of finding a mapping color : $X \rightarrow C$ such that*

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

is called list k-coloring.

Lemma 1.3 *Let color be a k-coloring of a linked list $L = (X, S)$.
The set of local minima of the k-coloring*

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

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

Proof 1.3 *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.
Since the k-coloring assigns different colors to adjacent nodes for each pair of nodes u, v such that $S(u) = v$ $color(u) < color(v)$ or $color(u) > color(v)$ thus x and y cannot be adjacent thus set of local minima forms an independent set.
Since there are no local minima in 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 at least $\frac{n}{2 \cdot k - 2} \in \Omega(\frac{n}{k})$.* $\square$

Definition 1.70 *Let $L = (X, S)$ be a linked list. Problem of finding a mapping $rank : X \rightarrow \mathbb{N}_0$ such that*

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

is called list ranking.

[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.

1.6 Euler Tour Technique

Definition 1.71 (Oriented) Euler tour *of a (oriented) graph $G = (V, E)$ is a sequence of consecutive (oriented) edges in the graph G that traverses every (oriented) edge in E exactly once.*
(Oriented) Graph G that contains Euler tour is called (oriented) Euler *graph.*

Theorem 1.2 *(Euler's theorem[2]) A connected graph $G = (V, E)$ is Euler if and only if*

$$(\forall u \in V) deg(u) \textit{ is even}$$

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

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

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 1.4 *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*

$$(\forall u \in V)(\forall u' \in V') u = u' \Rightarrow (deg(u) = deg_{in}(u') = deg_{out}(u'))$$

Hence G' is oriented Euler graph. □

Definition 1.72 Euler tour technique *is a problem of finding of an Euler tour of an ordered tree.*

1.7 Parentheses Matching

Definition 1.73 *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 1.74 *Let* $w \in \{(,)\}^*$ *be a well-formed string of parentheses. Problem of finding a mapping* $match : \mathbb{N} \to \mathbb{N}_0$ *such that* $(\forall i, j \in \widehat{|w|})$

$$match(i) = j \Leftrightarrow match(j) = i \Leftrightarrow (i < j \wedge w_i\ \ldots\ w_j\ is\ well-formed)$$

is called parentheses matching.

Chapter 2

Analysis and Design

In this chapter used data structures will be designed first, then algorithms solving problems defined in chapter 1 will be analysed.

All those algorithms are needed to run k-local DFTA in parallel work-optimally and are used as support algorithms in the main algorithm which will be analysed at the end of this chapter.

2.1 Structures

2.1.1 Array

An array is a consecutive memory block of a specific type that provides random access to its elements.

There are several implementations for arrays. The most simple one is the C-like array which is just an allocated block of memory.

Standard C++ libraries include multiple implementations of the array. The most important are *std::vector* and *std::array.*

std::array is a statically allocated array that cannot be resized during run-time. It is simple wrapper around C-like arrays that adds boundary checks, iterators and other C++ functionalities that satisfies the requirements of *Container*, *ReversibleContainer*, *ContiguousContainer* and partially *SequenceContainer.*

std::vector is a dynamically allocated array wrapping a C-like array that allows resizing of the array on run-time. This occurs when capacity of the vector is insufficient for stored elements. Resizing must move all elements of the vector thus insertion has $O\left(()\, n\right)$ time complexity but this is just in the case of increasing capacity which is very infrequent. Amortized time of insert is thus $\Theta^*\left(1\right)$.

Both of those implementations are however designed for sequential use only and are slow for use in parallel programming.

Boost library contains parallel implementation of the array *boost::compute::vector* that stores values in OpenCL buffer for fast computations.

2.1.2 Tree

Tree in this thesis 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:

- *labels* including symbols of ranked alphabet $\mathcal{F}$ stored in each vertex
- *childs* pointing to (including indices of) childs of each vertex

and a pointer to (index of) a root of the tree.

2.1.3 Arc

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

- pointer to (index of) source vertex
- pointer to (index of) target vertex
- pointer to (index of) opposite arc
- type of the arc (upgoing/downgoing)

Since Arc is a very specific struct with specific usage there are no implementations of it in C++ standard libraries nor Boost libraries.

2.1.4 DFTA

DFTA as defined in definition 1.36 is a 4-tuple $A = (Q, \mathcal{F}, Q_f, \Delta)$. Since the ranked alphabet $\mathcal{F}$ and transition function Δ must contain the same set of symbols DFTA may be represented by a 3-tuple consisting of

- a finite set of states.
- a finite set of final states.
- a transition function.

Assuming that states are numbers $0, \dots, n$ finite set of states can be represented by the state with the greatest number (i.e. n).

Since transition function has different arity for different symbols, this can be represented by a table associating symbol to corresponding transition function of symbol-specific arity.

2.2 Reduction and Scan

In the following algorithms, for purpose of simplicity, the size of the input is assumed to be a power of two. There are as many processors as needed available and in the parallel sections of the algorithms, all processors execute the same statement in synchrony.

2.2.1 Reduction

Problem of reduction is defined by definition 1.63.

2.2.1.1 Algorithms

Since the operator $\oplus$ is associative the problem

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \cdots \oplus x_{n-3} \oplus x_{n-2} \oplus x_{n-1} \oplus x_n$$

can be reformulated into a *linear order*

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

or a *tree-like order*

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

Sequential solution of this problem can be easily made from the linear order because except for the first pair of values reduction operator is always applied to a cumulative intermediate result and a value $x_i, 3 \leq i \leq n$. Let $\mathbb{0}$ be a left-identity with respect to the reduction operator $\oplus$. Then linear order can be written as

$$((((\dots((((\mathbb{0} \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus \dots \oplus x_{n-3}) \oplus x_{n-2}) \oplus x_{n-1}) \oplus x_n)$$

and if we consider $\mathbb{0}$ to be a first cumulative intermediate result then reduction operator is applied to a cumulative intermediate result and value $x_i, \forall i \in \widehat{n}$. Hence the algorithm 1.

Algorithm 1: Sequential reduction

```
Input: values $x_1, \dots, x_n$
Result: reduction of values
$r \leftarrow \mathbb{0}$;
for $i \leftarrow 1$ to $n$ do
    $r \leftarrow r \oplus x_i$;
end
return $r$;
```

Each value x_i is on the right side of the reduction operator $\oplus$ only once in algorithm 1. Hence the time complexity

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

And because each value x_i must appear at least once on left or right side of the reduction operator problem cannot be solved faster than in linear time.

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

Parallel solution of reduction is achievable with usage of the tree-like order of the problem

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

because application of reduction operator on pairs of values colored in red depicted above doesn't contain any read/write conflicts and thus can run

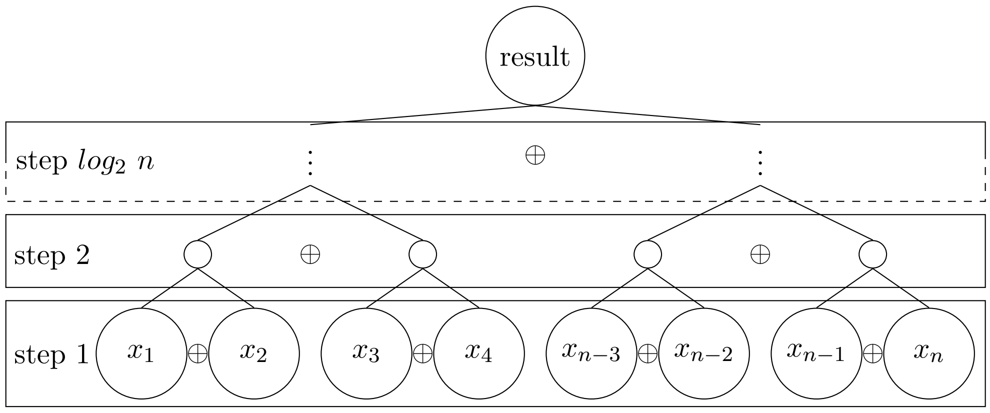


Figure 2.1: Parallel reduction computation

independently on each other in parallel. Solving those independent pairs leads to

$$(\dots(x_{1,2} \oplus x_{3,4}) \oplus \cdots \oplus (x_{n-3,n-2} \oplus x_{n-1,n})\dots)$$

which is the problem of reduction too and can be solved in the same way. This can be repeated until a single value (result) remains. Figure 2.1 depicts how parallel reduction is computed.

This can be formulated as algorithm 2.

Algorithm 2: Parallel reduction (EREW PRAM)

```
Input: values $\{x_i : i \in \widehat{n}\}$ and $n$ is power of 2
Result: reduction of values
Auxiliary: intermediate results $r_1, \dots, r_{\frac{n}{2}}$, left and right indices
           $left_i, right_i, \forall i \in \widehat{\frac{n}{2}}$
for $i \leftarrow 1$ to $log_2\ n$ do
    for $j \leftarrow 1$ to $\frac{n}{2^i}$ do in parallel
        $left_j \leftarrow 1 + (j-1) \cdot 2^i$;
        $right_j \leftarrow left + 2^{i-1}$;
        $r_{left_j} \leftarrow r_{left_j} \oplus r_{right_j}$;
    end
end
return $r_1$;
```

Each thread in inner cycle executes $O\left(1\right)$ arithmetic operations and thus runs in $O\left(\frac{n}{p}\right)$ time using p processors. Outer cycle has $O\left(log_2\ n\right)$ iterations taking $O\left(\frac{n}{p}\right)$ time, hence the parallel time

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

Speedup is

$$S\left(n,p\right)=\frac{n\cdot p}{n\cdot log_2\ n}=\frac{p}{log_2\ n}$$

Parallel cost of algorithm 2 is

$$C\left(n,p\right)=p\cdot O\left(log_2\ n\right)=O\left(n\cdot log_2\ n\right)=\omega\left(SU\left(n\right)\right)$$

thus the algorithm is not cost-optimal.

A source of non-cost-optimality is the fact that in each step half of the threads are active than in the previous step.

A parallel work of the algorithm is

$$W\left(n,p\right)=\sum_{i=1}^{log_2\ n}\frac{n}{2^i}=\sum_{i=1}^{log_2\ n}\Theta\left(n\right)=log_2\ n\cdot\Theta\left(n\right)=O\left(n\cdot log_2\ n\right)=\omega\left(SU\left(n\right)\right)$$

hence the algorithm is not work-optimal.

As shown before there are no read/write conflicts during execution thus those complexities apply on EREW PRAM.

This algorithm can be made cost and work-optimal by splitting the input into smaller portions that are precomputed sequentially by individual processors and then reducing the results of the sequential reductions. This trick is described in more detail in the inclusive scan section.

2.2.1.2 Implementations

There exists multiple implementation of reduction. For sequential reduction there exists function *std::reduce* in C++17 *numeric* library. This implementation works in $O(n)$ time and is similar to algorithm 1.

For parallel reduction overriden function *std::reduce* can be used that uses the work-optimal modification of the algorithm described above.

Boost library includes its own implementation of reduction *boost::compute::reduce* that runs in logarithmic time too, but is implemented using OpenCL (computation using graphic cards) and thus is much faster than (std::reduce).

OpenMP comes with implementation of reduction too that is implemented similarly to algorithm 2 but uses preprocessor directives instead of function.

2.2.2 Inclusive scan

Problem of inclusive scan is defined by definition 1.64.

Problem of inclusive scan is very similar to the problem of reduction. The only difference is that reduction aims to obtain result only for the whole set of values $\{x_1, \ldots, x_n\}$ but inclusive scan aims to obtain results of all subsets $\{x_1, \ldots, x_i\}$ such that $i \leq n$. That means to get all intermediate results of linear order described in reduction analysis 2.2.1.

2.2.2.1 Algorithms

Sequential solution of this problem is a simple modification of the sequential solution of the reduction. It is just needed to store all the intermediate results.

Algorithm 3: Sequential inclusive scan

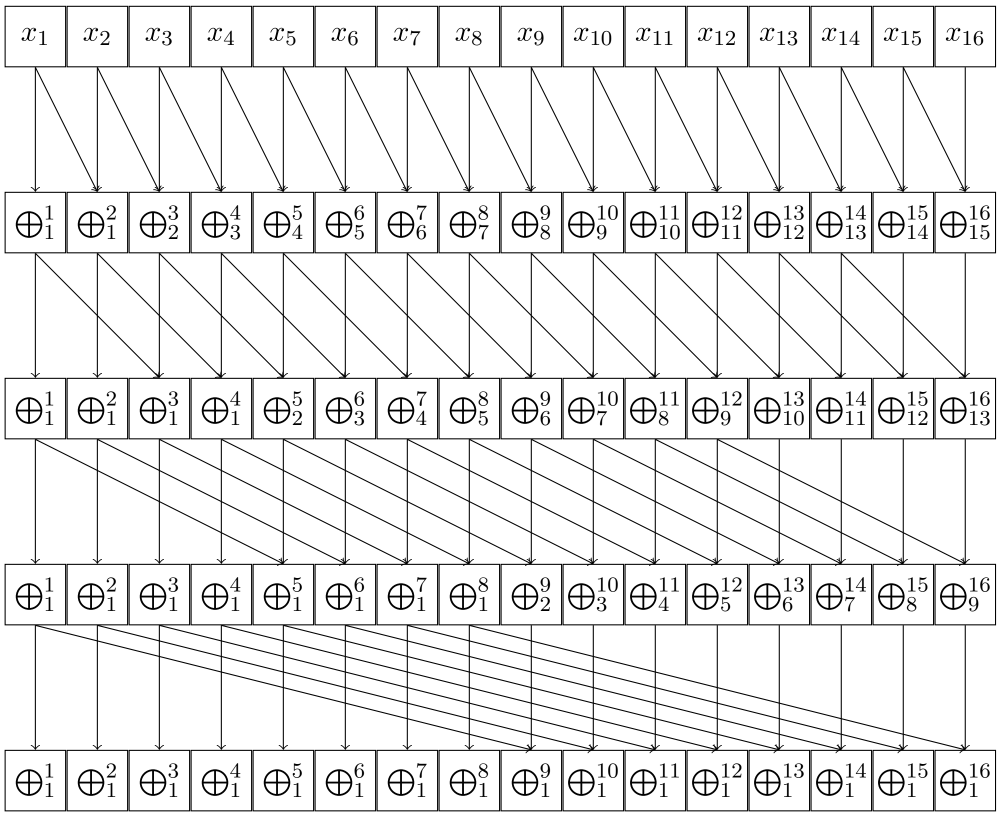
```
**Input:** values $x_1, \ldots, x_n$
**Output:** inclusive scan $s_1, \ldots, s_n$
$r \leftarrow \mathbb{0}$;
**for** $i \leftarrow 1$ **to** $n$ **do**
    $r \leftarrow r \oplus x_i$;
    $s_i \leftarrow r$;
**end**
```

Since the algorithm 3 is the same as the algorithm 1 complexities are the same too.

The same applies to the lower bound of the problem.

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

Since to achieve parallelism for the reduction tree-like order of problem must be used but intermediate results of linear order are needed, parallel solution of inclusive scan cannot be similarly simple modification of parallel reduction as in case of the sequential solution.



x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16}
$\oplus_1^1$ $\oplus_1^2$ $\oplus_2^3$ $\oplus_3^4$ $\oplus_4^5$ $\oplus_5^6$ $\oplus_6^7$ $\oplus_7^8$ $\oplus_8^9$ $\oplus_9^{10}$ $\oplus_{10}^{11}$ $\oplus_{11}^{12}$ $\oplus_{12}^{13}$ $\oplus_{13}^{14}$ $\oplus_{14}^{15}$ $\oplus_{15}^{16}$
$\oplus_1^1$ $\oplus_1^2$ $\oplus_1^3$ $\oplus_1^4$ $\oplus_2^5$ $\oplus_3^6$ $\oplus_4^7$ $\oplus_5^8$ $\oplus_6^9$ $\oplus_7^{10}$ $\oplus_8^{11}$ $\oplus_9^{12}$ $\oplus_{10}^{13}$ $\oplus_{11}^{14}$ $\oplus_{12}^{15}$ $\oplus_{13}^{16}$
$\oplus_1^1$ $\oplus_1^2$ $\oplus_1^3$ $\oplus_1^4$ $\oplus_1^5$ $\oplus_1^6$ $\oplus_1^7$ $\oplus_1^8$ $\oplus_2^9$ $\oplus_3^{10}$ $\oplus_4^{11}$ $\oplus_5^{12}$ $\oplus_6^{13}$ $\oplus_7^{14}$ $\oplus_8^{15}$ $\oplus_9^{16}$
$\oplus_1^1$ $\oplus_1^2$ $\oplus_1^3$ $\oplus_1^4$ $\oplus_1^5$ $\oplus_1^6$ $\oplus_1^7$ $\oplus_1^8$ $\oplus_1^9$ $\oplus_1^{10}$ $\oplus_1^{11}$ $\oplus_1^{12}$ $\oplus_1^{13}$ $\oplus_1^{14}$ $\oplus_1^{15}$ $\oplus_1^{16}$

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

To achieve parallel solution logically redundant applications of $\oplus$ operator must be added to the parallel solution of the reduction. This solution was firstly presented by Hillis and Steele. Figure 2.2 depicts how is inclusive scan of an array of size 16 computed using Hillis-Steele algorithm[4].

The algorithm 4 applies $\oplus$ operator to each pair of consecutive elements in the first iteration and in every following iteration it applies $\oplus$ operator to each pair of elements that are double the distance from previous iteration far away from each other.

Hillis-Steele algorithm 4 contains read/write conflict but could be easily solved by using auxiliary array to temporarily store values from previous iteration thus is applicable for EREW PRAM.

Copying input to output on the first line could be executed in $O\left(\frac{n}{p}\right)$ time in parallel using p processors.

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

```
Input: values $x_1, \ldots, x_n$
Output: inclusive scan $s_1, \ldots, s_n$
$s_i \leftarrow x_i, \forall i \in \widehat{n}$;
for $i \leftarrow 1$ to $log_2\ n$ do
    for $j \leftarrow 1$ to $n - 2^{i-1}$ do in parallel
        $s_{j+2^{i-1}} \leftarrow s_j \oplus s_{j+2^{i-1}}$;
    end
end
return $r_1$;
```

The inner loop of the algorithm contains $O(1)$ arithmetic operations and is executed in parallel thus outer loop with $O(log_2\ n)$ steps execute in $O(1 \cdot log_2\ n)$.

Hence the complexities

$$T(n,p) = O\left(\frac{n}{p}\right) + O(log_2\ n) \cdot O(1) = O\left(\frac{n}{p} + log_2\ n\right)$$

$$T(n,n) = O(log_2\ n)$$

$$S(n,p) = \frac{n}{\frac{n}{p} + log_2\ n}$$

$$S(n,n) = \frac{n}{log_2\ n}$$

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

$$C(n,n) = O(n + n \cdot log_2\ n) = O(n \cdot log_2\ n) = \omega(SU(n))$$

thus Hillis-Steele algorithm is not cost-optimal. It has the same reason as in the case of parallel reduction.

A parallel work of the algorithm is

$$W(n,p) = \sum_{i=1}^{log_2\ n} n - 2^{i-1} = n \cdot log_2\ n - \sum_{i=1}^{log_2\ n} 2^{i-1} = O(n \cdot log_2\ n) = \omega(SU(n))$$

since

$$\sum_{i=1}^{log_2\ n} 2^{i-1} \leq \sum_{i=1}^{log_2\ n} 2^{log_2\ n} = \sum_{i=1}^{log_2\ n} n = n \cdot log_2\ n$$

thus the algorithm is not work-optimal.

To achieve work-optimality of the algorithm simple trick could be used. The input is split to same-sized smaller portions that are precomputed sequentially.

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

```
Input: values $x_1, \dots, x_n$
Output: inclusive scan $s_1, \dots, s_n$
Auxiliary: intermediate results $r_1, \dots, r_p$ where $p$ is the number of
           processors
split $x_1, \dots, x_n$ to $p$ consecutive sections of size $\frac{n}{p}$;
foreach section $x_i, \dots, x_j$ do in parallel
    sequential inclusive scan (in: $x_i, \dots, x_j$, out: $x_i, \dots, x_j$);
    $r_{tid} \leftarrow s_j$;
end
Hillis-Steele algorithm (in: $r$, out: $r$);
foreach section $x_i, \dots, x_j$ do in parallel
    for $k \leftarrow i$ to $j$ do
        if $tid \neq 1$ then
            $s_k \leftarrow r_{tid-1} \oplus s_k$;
        end
    end
end
```

Results of those precomputation are then used in Hillis-Steele algorithm and then applied back to the smaller portions.

The precomputation of the portion takes $\frac{n}{p}$ time. Then Hillis-steele is executed in $log_2\ p$ time. Total parallel time is

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

and using $\frac{n}{log_2\ n}$ processors this is

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

and since

$$log_2\ \frac{n}{log_2\ n} = O(log_2\ n)$$

the parallel time results in

$$T\left(n, \frac{n}{log_2\ n}\right) = O(log_2\ n)$$

The parallel time and speedup are unafffected by the modification. But the parallel cost and work are now

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

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

thus the modified algorithm is cost-optimal and work-optimal.

Another parallel solution of this was presented by Blelloch[5]. The algorithm consists of 2 steps. The first one, called up-sweep step, is almost identical to the parallel reduction.

Algorithm 6: Up-Sweep step of 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
           $right_1, \dots, right_{\frac{n}{2}}$
$r_i \leftarrow x_i, \forall i \in \widehat{n}$;
for $i \leftarrow 1$ to $log_2\ n$ do
    for $j \leftarrow 1$ to $\frac{n}{2^i}$ do in parallel
        $left_j \leftarrow 1 + (j-1) \cdot 2^i$;
        $right_j \leftarrow left + 2^{i-1}$;
        $r_{left_j} \leftarrow r_{left_j} \oplus r_{right_j}$;
    end
end
```

The first step formulated by algorithm 6 can be depicted as a tree where the computation proceeds from the leaves to the root and thus is called *Up Sweep step*. Result of the Up Sweep step can be used to compute scan of the input.

The second step is called *down-sweep step* because computation proceeds from the root to the leaves. At the beginning identity $\mathbb{0}$ is set to the root. At each step (level of the tree) the algorithm passes value from parent to its left child and $\oplus$ applied to the parent and left child is passed to the right child.

The figures 2.3 and 2.4 depicts up-sweep and down-sweep step respectively for input size of 8 elements. As can be seen from figure 2.4 this step is natively exclusive, but it will prove useful in the algorithm as a whole.

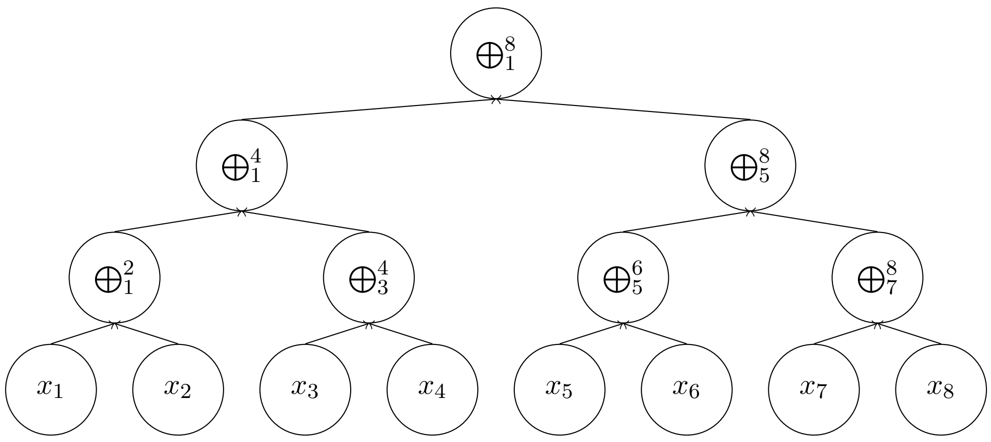


Figure 2.3: Up Sweep step for the input of the size 8

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

```
Input: values $x_1, \dots, x_n$ and $n$ is power of 2
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 $t_1, \dots, t_{\frac{n}{2}}$
$s_i \leftarrow x_i, \forall i \in \widehat{n}$;
for $i \leftarrow log_2\ n$ downto 1 do
    for $j \leftarrow 1$ to $\frac{n}{2^i}$ do in parallel
        $left_j \leftarrow 1 + (j-1) \cdot 2^i$;
        $right_j \leftarrow left + 2^{i-1}$;
        $t_j \leftarrow s_{left_j} \oplus s_{right_j}$;
        $s_{left_j} \leftarrow s_{right_j}$;
        $s_{right_j} \leftarrow t_j$;
    end
end
```

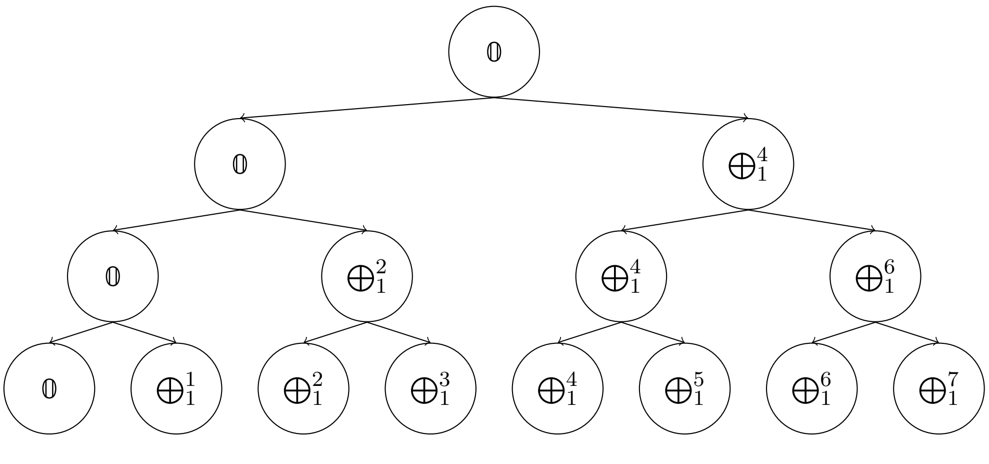


Figure 2.4: Down Sweep step for the input of the size 8

Inner loop in both steps contains $O(1)$ arithmetic operations and thus both outer loops executes in $O\left(\frac{n}{p} \cdot log_2\ n\right)$ time using p processors. Copying input to output values at the beginning of both steps takes $O\left(\frac{n}{p}\right)$ time.

If there's a fixed number of processors $p \leq n$ the input can be split into p almost equally-sized sections, each scanned sequentially by single processor. Since the sequential algorithm runs in $O(n)$ time and size of each section is about $\frac{n}{p}$ this will take $O\left(\frac{n}{p}\right)$ time.

Then the values $\bigoplus_i^j$ from each section starting with x_i and ending with x_j will together form an input for a parallel scan. This means for parallel scan there will be the input of the size p and thus will run in $O(log_2\ p)$ time.

Results of parallel scan will be used by processors as offset for values to apply to its section. This takes $O\left(\frac{n}{p}\right)$ time.

In the algorithm 8 $tid \in \widehat{p}$ denotes id of executing thread and for each two sections $x_i, \ldots, x_j$ and $x_k, \ldots, x_l$ where $i < j < k < l$ section $x_i, \ldots, x_j$ is executed by a thread with lower id than the other section.

The total parallel time of the Blelloch algorithm is

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

Algorithm 8: Blelloch scan alorithm (EREW PRAM)

```
Input: values $x_1, \dots, x_n$
Output: inclusive scan $s_1, \dots, s_n$
Auxiliary: intermediate results $r_1, \dots, r_p$ where $p$ is the number of
           processors
split $x_1, \dots, x_n$ to $p$ consecutive sections of size $\frac{n}{p}$;
foreach section $x_i, \dots, x_j$ do in parallel
    sequential inclusive scan (in: $x_i, \dots, x_j$, out: $s_i, \dots, s_j$);
    $r_{tid} \leftarrow s_j$;
end
up-sweep (in: $r$, out: $r$);
down-sweep (in: $r$, out: $r$);
foreach section $x_i, \dots, x_j$ do in parallel
    for $k \leftarrow i$ to $j$ do
        $s_k \leftarrow r_{tid} \oplus s_k$;
    end
end
```

Speedup of the algorithm is

$$S(n,p) = \frac{n}{\frac{n}{p} + log_2\ p}$$

$$S\left(n, \frac{n}{log_2\ n}\right) = \frac{n}{log_2\ n}$$

which is the same as in the case of Hillis-Steele algorithm. Despite the fact that Blelloch and Hillis-Steele algorithm have asymptotically same parallel time, Hillis-Steele algorithm should be faster and when unmodified is applicable for linked lists.

Cost of the Blelloch algorithm is

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

$$C\left(n, \frac{n}{log_2\ n}\right) = O\left(n + \frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) = O(n) = \Theta(SU(n))$$

thus Blelloch algorithm is cost-optimal. The same applies to parallel work

$$W(n,p) = n + O(p \cdot log_2\ p) + O(p \cdot log_2\ p) + n$$

$$= O(n + p \cdot log_2\ p)$$

$$\begin{aligned} W\left(n, \frac{n}{log_2\ n}\right) =& O\left(\frac{n}{log_2\ n} + \frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) \\ =& O\left(\frac{n}{log_2\ n} + \frac{n}{log_2\ n} \cdot log_2\ n\right) \\ =& O\left(n\right) = \Theta\left(SU\left(n\right)\right) \end{aligned}$$

since

$$log_2\ \frac{n}{log_2\ n} = O\left(log_2\ n\right)$$

Parallel work of up and down-sweep steps are derived from parallel reduction work.

2.2.2.2 Implementations

As in the case of reduction there exist multiple implementations of inclusive scan. The most important are implementation in C++17 standard *numeric* library *std::inclusive_scan* for both, sequential and parallel, scans and implementation in Boost *compute* library *boost::compute::inclusive_scan* that is parallel only.

The standard implementation uses a modified Hillis-Steele algorithm for the purpose of speed. The implementation in Boost library does the same but instead of processors is employing graphic cards for the computation to achieve an even better time.

2.2.3 Exclusive scan

Problem of exclusive scan is defined by definition 1.65.

The problem of exclusive scan is almost the same as inclusive scan. The only difference is that result of the exclusive scan is shifted relative to the result of the inclusive scan.

2.2.3.1 Algorithms

Thus algorithms for the inclusive scan are valid algorithms for the exclusive scan with an added shifting of the result.

2.2.3.2 Implementations

There are implementations of exclusive scan in both standard C++17 *numeric* library and Boost *compute* library. Those are *std::exclusive_scan* and *boost::compute::exclusive_scan* respectively implemented in the same way as the inclusive scan.

2.2.4 Segmented scan

Problem of segmented scan is defined by definition 1.66.

2.2.4.1 Algorithms

Since segmented scan (inclusive or exclusive) is the same problem as its non-segmented variant but has only modified $\oplus$ operator to take into account a segment indicator, all algorithm for non-segmented variant applies to the segmented scan too.

2.2.4.2 Implementations

There are no implementations of segmented scans in C++17 standard *numeric* library nor Boost *compute* library.

2.3 Lists

2.3.1 Linked list

Linked lists are defined by definition 1.67.

Linked lists could be implemented as

- a dynamic linked structure where each element (node) of the list is represented by a structure consisting of a node value and a pointer to its succesor.

$$\begin{tikzcd}
4 \arrow[r] & 2 \arrow[r] & 7 \arrow[r] & 1 \arrow[r] & 5 \arrow[r] & 6 \arrow[r] & 8 \arrow[r] & 3 \arrow[loop right]
\end{tikzcd}$$

Figure 2.5: Dynamic linked list

- an array (successor array) where each element of the array represents one element (node) of the list and contains an index of its successor and may contain a value of the node.

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

Figure 2.6: Successor array representation of a linked list

2.3.1.1 Implementations

C++ standard *list* library contains a class list which is an implementation of a linked list. This implementation uses the first approach to represent the linked list, that is as a dynamic linked structure.

2.3.2 List Ranking

The problem of list ranking is defined by definition 1.70.

The rank of the node in the linked list is equal to the number of nodes preceding it in the list. This can be acquired using inclusive scan on the list, where initial value of each node is set to 1. The inclusive scan as described in analysis 2.2.2 is inclusive scan on arrays.

Sequential solution of a list ranking is simply counting the number of node already traversed and assigning this count to the current node as its rank. Hence the algorithm 9.

Algorithm 9: Sequential list ranking

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

This algorithm visits each node exactly once hence the time complexity

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

Since to assign the rank to the node each node must be visited at least once the lower bound must be at least linear

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

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

In the Hillis-Steele algorithm distance between two elements on which $\oplus$ operator is applied is doubled each iteration. This can be easily achieved on linked lists by assigning $s_i \leftarrow s_{s_i}$ in each iteration. This act is called *pointer jumping*.

Figure 2.7 depicts how list ranking is computed by pointer jumping. Value inside each node is the resulting rank.

Inner loop of the algorithm 10 executes $O(1)$ arithmetic operations. The inner loop is executed in parallel using p processors. The outer loop has $log_2\ n$ steps hence the parallel time

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

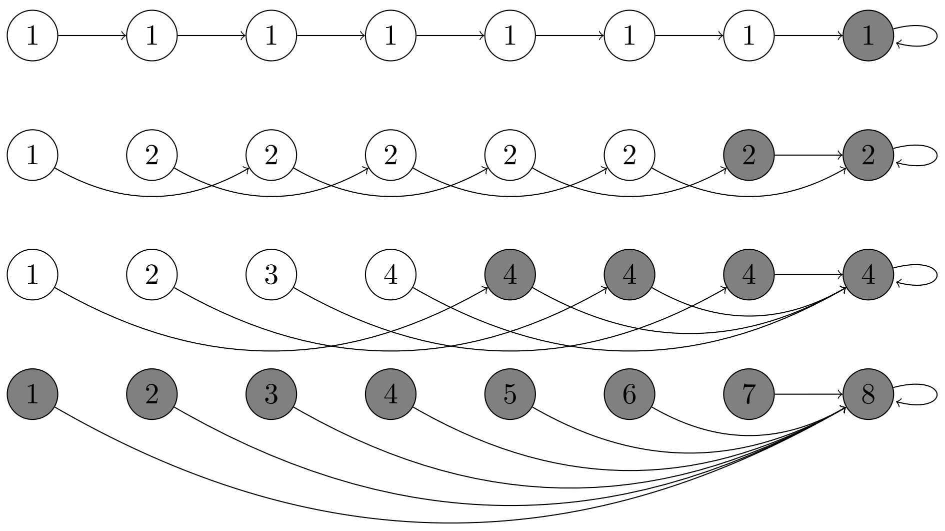


Figure 2.7: Parallel list ranking by pointer jumping

Algorithm 10: List ranking by pointer jumping (CREW PRAM)

```
**Input:** linked list given as successor array $s_1, \ldots, s_n$
**Output:** list ranking $r_1, \ldots, r_n$
**Auxiliary:** node active indicators $a_1, \ldots, a_n$
$a_i \leftarrow active, \forall i \in \widehat{n}$;
$r_i \leftarrow 1, \forall i \in \widehat{n}$;
for $i \leftarrow 1$ to $log_2\ n$ do
    for $j \leftarrow 1$ to $n$ do in parallel
        if $a_i = active$ then
            $r_{s_i} \leftarrow r_i + r_{s_i}$;
            if $s_i = s_{s_i}$ then
                $a_i \leftarrow inactive$;
            else
                $s_i \leftarrow s_{s_i}$;
            end
        end
    end
end
```

$$T(n,n) = O(log_2\ n)$$

and parallel speedup

$$S(n,p) = \frac{n \cdot p}{n \cdot log_2\ n} = \frac{p}{log_2\ n}$$

Parallel cost of this algorithm is

$$C(n,p) = p \cdot O\left(\frac{n}{p} \cdot log_2\ n\right) = O(n \cdot log_2\ n) = \omega(SU(n))$$

and parallel work is

$$W(n,p) = \sum_{i=1}^{log_2\ n} n = O(n \cdot log_2\ n) = \omega(SU(n))$$

thus algorithm is neither cost-optimal nor work-optimal.

There doesn't exist any algorithm that is able to identify same-sized consecutive portions of the linked list in logarithmic time thus this modification of Hillis-Steele algorithm cannot be made optimal by splitting the input into smaller portions.

The idea of cost and work-optimal list ranking comes from the following observation.

Observation 2.1 *Let there be a linked list of size $n' = \frac{n}{log_2\ n}$ then using $p = n' = \frac{n}{log_2\ n}$ processors list ranking by pointer jumping of this linked list is executed in parallel time*

$$T(n',n') = O(log_2\ n') = O\left(log_2\ \frac{n}{log_2\ n}\right) = O(log_2\ n)$$

and has parallel cost

$$C(n',n') = O(n' \cdot log_2\ n') = O\left(\frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) =$$

$$= O\left(\frac{n}{log_2\ n} \cdot log_2\ n\right) = O(n) = \Theta(SU(n))$$

The general idea is formulated in algorithm 11.

This idea demands from shrinking and restoring to run in $T\left(n, \frac{n}{log_2\ n}\right) = O(log_2\ n)$ time with parallel cost $C\left(n, \frac{n}{log_2\ n}\right) = O(n)$.

Algorithm 11: Idea of work-optimal list ranking

```
**Input:** linked list $L$ with $n$ elements
**Output:** list ranking $r$
shrink $L$ to $L'$ of size $n' = O\left(\frac{n}{log_2\ n}\right)$ using $\Theta\left(\frac{n}{log_2\ n}\right)$ processors;
apply pointer jumping on $L'$;
restore $L$ from $L'$ and finish ranking for elements in $L \setminus L'$ with the
  same complexity as in the case of shrinking;
```

If those demands are met then this algorithm runs in parallel time

$$T\left(n, \frac{n}{log_2\ n}\right) = O\left(n\right)$$

with speedup

$$S\left(n, \frac{n}{log_2\ n}\right) = \frac{n}{log_2\ n}$$

Has cost

$$C\left(n, \frac{n}{log_2\ n}\right) = O\left(n + n + n\right) = O\left(n\right) = \Theta\left(SU\left(n\right)\right)$$

and work

$$W\left(n, \frac{n}{log_2\ n}\right) = O\left(n + n + n\right) = O\left(n\right) = \Theta\left(SU\left(n\right)\right)$$

thus is cost and work-optimal.

There exists a way to achieve a work-optimality but not a cost-optimality. This approach utilizes an independent sets of a linked list defined by definition 1.68. As mentioned in lemma 1.3 the set of local minima in k-coloring forms an independent set of size $\Omega(\frac{n}{k})$.

The best k-coloring achievable in parallel on EREW PRAM is a 3-coloring.

Size of the independent set formed from 3-coloring is at least $\frac{n}{2\cdot 3-2} = \frac{n}{4}$. Thus size of the linked list $L' = L \setminus I$ is $n - \frac{n}{4} = \frac{3}{4} \cdot n$. Linked list L' could be shrinked in the same way.

After s removals the size of the shrinked linked list is $\left(\frac{3}{4}\right)^s \cdot n$. To achieve the required size of at most $\frac{n}{log_2\ n}$ $\Theta(log_2^2\ n)$ removals must be applied to the linked list L because

$$\begin{aligned}\left(\frac{3}{4}\right)^s \cdot n \leq \frac{n}{log_2\ n} \Rightarrow & \left(\frac{3}{4}\right)^s \leq \frac{1}{log_2 n}\\ \Rightarrow & \left(\frac{4}{3}\right)^s \geq log_2\ n\\ \Rightarrow & s \geq log_{\frac{4}{3}}\ log_2\ n = \frac{log_2^2\ n}{log_2\ \frac{4}{3}} = \Theta\left(log_2^2\ n\right)\end{aligned}$$

2.3.2.1 6-coloring

Problem of k-coloring is defined by definition 1.69.

The first step in obtaining a 3-coloring of a linked list is to obtain it's 6-coloring. This can be achieved by using technique called *Deterministi Coin Tossing* (DCT) to reduce n-coloring to 6-coloring.

Let x_{bin} be a binary representation of x and let $diff(x, y)$ be the least bit number in which x_{bin} and y_{bin} differ. $x_{[i]}$ denotes i-th bit of x. Let log_b^* be a function defined as

$$log_b^*\ x := min\{i : log_b^i\ x \leq 1\}$$

Algorithm 12: 6-coloring (CREW PRAM)

```
Input: linked list given as successor array $s_1, \ldots, s_n$
Output: 6-coloring $c_1, \ldots, c_n$
Auxiliary: diff result storage $\pi_1, \ldots, \pi_n$
(* Generate $n - coloring$ *)
for $i \leftarrow 1$ to $n$ do in parallel
    $c_i \leftarrow i$;
end
(* Reduce to $6 - coloring$ *)
for $i \leftarrow 1$ to $log_2^*\ n$ do
    for $j \leftarrow 1$ to $n$ do in parallel
        $\pi_j \leftarrow diff(c_j, c_{s_j})$;
        $c_j \leftarrow 2 \cdot \pi_j + c_{j[\pi_j]}$;
    end
end
```

The inner loop produces a valid coloring c' from a valid coloring c. Since c is a valid coloring for all adjacent pairs of nodes $c_i \neq c_{s_i}$. Thus π_i is well defined for each such pair. If $\pi_i \neq \pi_{s_i}$ then

$$c'_i = 2 \cdot \pi_i + c_{i[\pi_i]} \neq 2 \cdot \pi_{s_i} + c_{s_i\left[\pi_{s_i}\right]} = c_i$$

because $c_i, c_{s_i} \in \{0, 1\}$.

If c' is not valid coloring then π_i must be equal to π_{s_i} but then

$$2 \cdot \pi_i + c_{i[\pi_i]} = 2 \cdot \pi_i + c_{s_i[pi_i]}$$

$$c_{i[\pi_i]} = c_{s_i[\pi_i]}$$

And that is in contradiction with the fact that $c_i \neq c_{s_i}$ and π_i is the least bit numbere where c_i and c_{s_i} differ. Thus c' must be valid coloring.

Let n be the greatest color number in coloring c, then $\lfloor log_2\ n \rfloor$ is the greates value of π_i in such coloring and thus the greatest color number in coloring c' is $2 \cdot \lfloor log_2\ n \rfloor + 1$.

DCT can reduce n-coloring to 6-coloring only. If $\lfloor log_2\ n \rfloor = 2$ then the greatest color number reached by further reduction is $2 \cdot 2 + 1 = 5$ but $\lfloor log_2\ 5 \rfloor = 2$ thus DCT cannot further reduce coloring.

Since both parallel loops execute $O(1)$ arithmetic operations they can run in $O(1)$ time using n processors. The outer loop that has $log_2^*\ n$ steps won't exceed 6 steps for any realistic value since $log_2^*\ n = 6 \Rightarrow 2^{65536} < 2^{2^{65536}}$ thus the outer loop can be approximated to have $O(1)$ steps thus this can be approximated to $O(1)$ steps.

The parallel time (considering the mentioned approximation) of 6-coloring is

$$T\left(n, p\right) = O\left(\frac{n}{p}\right)$$

Parallel cost and work of this algorithm are

$$C\left(n, p\right) = p \cdot O\left(\frac{n}{p}\right) = O\left(n\right)$$

$$W\left(n, p\right) = n + n \cdot log_2^*\ n = O\left(n\right)$$

considering approximation that $log_2^*\ n = O\left(1\right)$.

Algorithm 13: 3-coloring (EREW PRAM)

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

2.3.2.2 3-coloring

The 3-coloring is obtainable from 6-coloring simply by replacing colors greater than 2 with color $\in \{0, 1, 2\}$ that is not assigned to its neighbours.

Loops execute $O(1)$ arithmetic operations each. The parallel time of this algorithm is

$$T(n, p) = 4 \cdot O\left(\frac{n}{p}\right) = O\left(\frac{n}{p}\right)$$

The parallel cost and work are then

$$C(n, p) = p \cdot O\left(\frac{n}{p}\right) = O(n)$$

$$W(n, p) = O(n) + 3 \cdot n = O(n)$$

2.3.2.3 Work-optimal list ranking

The following algorithm is application of idea in algorithm 11 using 3-coloring to identify independent sets.[3][6]

Algorithm 14: Work-optimal list ranking (CREW PRAM)

```
Input: linked list given as successor array $s_1, \dots, s_n$
Output: list ranking $r_1, \dots, r_n$
Auxiliary: indicators $f_1, \dots, f_n$, results $n_1, \dots, n_n$, coloring
           $c_1, \dots, c_n$, predecessors $p_1, \dots, p_n$, stack $S$, temporary
           $t_1, \dots, t_n$
$S \leftarrow \emptyset$, $n' \leftarrow n$, $r_i \leftarrow 1, \forall i \in \widehat{n}$;
reverse successor list $s$ to obtain predecessor list $p$;
while $n' > \frac{n}{log_2\ n}$ do
    $t_i \leftarrow \emptyset, \forall i$;
    3-coloring(in: $s_1, \dots, s_{n'}$, out: $c_1, \dots, c_{n'}$);
    (* Identify $I$ *)
    for $i \leftarrow 1$ to $n'$ do in parallel
        if $c_i < min(c_{p_i}, c_{s_i})$ then
            $f_i \leftarrow \top$, $n_i \leftarrow 1$;
        else
            $f_i \leftarrow \bot$, $n_i \leftarrow 0$;
        end
    end
    (* ! Remove $I$ from $L$ *)
    inclusive scan(in: $n_1, \dots, n_{n'}$, out: $n_1, \dots, n_{n'}$);
    for $\imath \leftarrow 1$ 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 *)
    $n_i \leftarrow 0$ $iff$ $f_i$ $else$ $1, \forall i \in \widehat{n}$;
    inclusive scan(in: $n_1, \dots, n_{n'}$, out: $n_1, \dots, n_{n'}$);
    for $\imath \leftarrow 1$ 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
list ranking by pointer jumping (in: $s_1, \dots, s_{n'}$, out: $r_1, \dots, r_{n'}$);
restore $L$ and rank removed nodes by emptying stack $S$ and reversing
  steps in while loop starting with ! in comment;
```

The first line of this algorithm can be executed in $O\left(\frac{n}{p}\right)$ time with $O\left(n\right)$ work using p processors since there is assignment to $O\left(n\right)$ variables that can be executed in parallel.

Reversing list is a simple operation

$$(\forall i \in \widehat{n})(i \neq s_i \wedge j = s_i) \Rightarrow i = p_j$$

$$(\forall i \in \widehat{n})((\forall j \in \widehat{n}) i \neq s_j) \Rightarrow i = p_i$$

and since there aren't 2 nodes that are successor of single node, this operation can run in parallel with time $O\left(\frac{n}{p}\right)$ and work $O\left(n\right)$ using p processors.

Asigning to n different variables on the first line inside the loop takes $O\left(\frac{n}{p}\right)$ time and $O\left(n\right)$ work.

3-coloring has time and work complexity of $O\left(\frac{n}{p}\right)$ and $O\left(n\right)$ respectively considering approximation of $log_2^* \; n = O\left(1\right)$.

Identifying I_k is a loop of $O\left(n\right)$ steps executed in parallel. Since there are $O\left(1\right)$ arithmetic operations inside the loop this loop executes in $O\left(\frac{n}{p}\right)$ time with $O\left(n\right)$ work.

Inclusive scan has parallel time $O\left(\frac{n}{p} + log_2 \; p\right)$ and work $O\left(n\right)$

Removal of I from L consists of inclusive scan and a loop of $O\left(n\right)$ steps executed in parallel with $O\left(1\right)$ arithmetic operations inside. Thus takes $O\left(\frac{n}{p} + log_2 \; p\right)$ time and $O\left(n\right)$ work using p processors.

Compacting L' to consecutive memory consist of parallel assignment to n different variables, inclusive scan and sligtly modified loop from previous step, thus has parallel time $O\left(2 \cdot \frac{n}{p}\right) + O\left(\frac{n}{p} + log_2 \; p\right) = O\left(\frac{n}{p} + log_2 \; p\right)$ and work $O\left(3 \cdot n\right) = O\left(n\right)$.

Last 2 rows inside the loop are simple assignments that take $O\left(1\right)$ time and work.

Since the size of I created by 3-coloring is of size $\Omega\left(\frac{n}{3}\right)$ the total number of steps k of the loop is

$$\begin{aligned}\left(\frac{2}{3}\right)^k \cdot n > \frac{n}{log_2\ n} \Rightarrow & \left(\frac{2}{3}\right)^k > \frac{1}{log_2\ n}\\ \Rightarrow & \left(\frac{3}{2}\right)^k < log_2\ n\\ \Rightarrow & log_{\frac{3}{2}}\ log_2\ n < k\\ \Rightarrow & k = O\left(log_2\ log_2\ n\right)\end{aligned}$$

Whole loop executes in parallel time

$$\begin{aligned}T\left(n,p\right) =& O\left(log_2\ log_2\ n\right) \cdot \left(7 \cdot O\left(\frac{n}{p}\right) + 2 \cdot O\left(\frac{n}{p} + log_2\ p\right) + 2 \cdot O\left(1\right)\right)\\ =& O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right)\end{aligned}$$

with parallel work

$$W\left(n,p\right) = 9 \cdot O\left(n\right) + 2 \cdot O\left(1\right) = O\left(n\right)$$

Pointer jumping is applied to a list of size $O\left(\frac{n}{log_2\ n}\right)$ with parallel time

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

and parallel work

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

The last row has the same complexity as the code in while loop starting with exclamation mark in comment (including the surrounding loop) thus

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

$$W\left(n,p\right)=O\left(n\right)$$

Overall, the algorithm executes in parallel time

$$\begin{aligned}T\left(n,p\right)=&3\cdot O\left(\frac{n}{p}\right)+2\cdot O\left(\left(\frac{n}{p}+log_2\ p\right)\cdot log_2\ log_2\ n\right)\\=&O\left(\left(\frac{n}{p}+log_2\ p\right)\cdot log_2\ log_2\ n\right)\end{aligned}$$

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

Has speedup

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

$$S\left(\left(,n\right),\frac{n}{log_2\ n}\right)=\frac{n}{log_2\ n\cdot log_2\ log_2\ n}$$

and cost

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

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

thus is not cost-optimal but

$$W\left(n,p\right)=5\cdot O\left(n\right)=O\left(n\right)=\Theta\left(SU\left(n\right)\right)$$

is work-optimal.

This algorithm is more efficient but is slower than pointer jumping.

2.3.2.4 Implementations

There's no implementation of list ranking neither in C++ standard libraries, Boost libraries nor any other popular C++ libraries. For sequential solution standard C++ inclusive scan together with standard C++ list could be used but there's no sufficient substitute for parallel solution.

2.4 Euler Tour Technique

Euler Tour Technique is defined by definition 1.72.

This is not a single algorithm but technique used by multiple algorithms. All of the algorithms use Euler tour of a tree, which starts and ends at its root.

2.4.1 Algorithms

Each edge could be represented by a pair of arcs. One arc is downgoing and the other one is upgoing. If there is a tree of size n on the input, that tree has $n - 1$ edges and thus this tree can be represented by $2 \cdot n - 2$ arcs.

Note 2.1 *The sequential solution of the Euler Tour construction will not be described in this thesis since there will be no sequential implementation but Euler tour can be sequentially constructed using simple modification of DFS. Thus the time complexity is $T(n) = SU(n) = O(n)$ and since each arc must be visited at least once, the lower bound is $SL(n) = O(n)$.*

For parallel solution those arcs can be arranged into an array where all arcs representing edges originated in single vertex form a consecutive portion of the array and pairs of opposite arcs are adjacent.

$origin(a)$, $target(a)$, $type(a)$ and $opposite(a)$ denotes an source node, target node, type of the arc (upgoing/downgoing) and opposite arc respectively.

Let there be a function $next(a)$ that assings to each arc such arc that

$$next(arc\,x{-}y) = \begin{cases} arc\ x - child_{i+1}(x),\ \text{iff}\ y = child_i(x),\ where\ i \neq arity(x) \\ arc\ x - parent(x),\ \text{iff}\ y = child_{arity(x)}(x) \wedge parent(x) \neq root \\ arc\ x - child_1(x),\ \text{iff}\ a\ is\ arc\ x - parent(x) \wedge arity(x) > 0 \\ arc\ x - y,\ otherwise \end{cases}$$

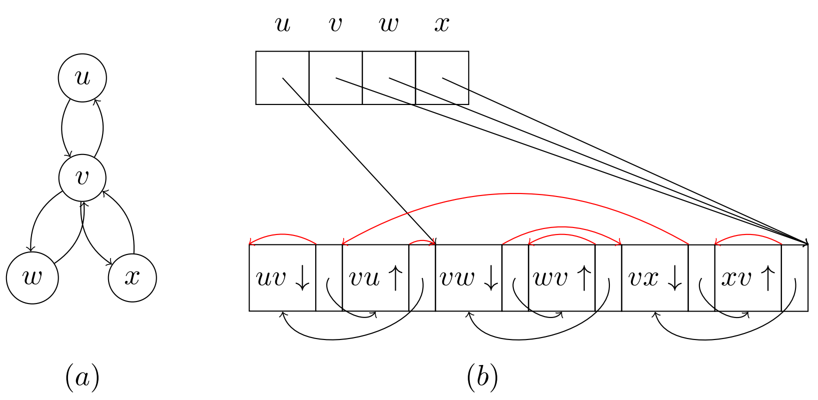


Figure 2.8: (a) Euler circit of tree (b) Array representation of arcs

Figure 2.8 (b) depicts the mentioned representation of a tree. The red arrow shows what is assigned to each arc by function $next(a)$.

The path is then constructed as follows

$$(\forall a \in arcs) path(a) := next(opposite(a))$$

with exception of the arc $f = child_{arity(root)}(root) - root$ for this arc path is defined as

$$path(f) := f$$

The path together with arcs form a linked list where arcs are elements of that linked list and path is a successor function.

If arcs are reordered with respect to the list ranking of that linked list then array of arcs forms an Euler tour of the input tree. Hence the algorithm 15.[7]

Arcs can be created in $O(1)$ for each edge. Since those arcs are opposite to each other there's no need to search opposite arcs.

To arrange those arcs into consecutive location of an array based on node of origin of the edge, prefix computations to prepare indices of individual arcs can be used. Thus preparation of this indices takes $O\left(\frac{n}{p}\right)$ time and those 2 arcs can be then inside the loop assigned in $O(1)$ time to the right position.

Thus the first loop considering the preparation step takes $O\left(\frac{n}{p}\right)$ time since it has $O(n)$ steps executed in parallel.

Path computation can be made in $O\left(\frac{n}{p}\right)$ time using p processors on EREW PRAM since there are no read-write conflicts and loop runs in parallel.

Algorithm 15: Euler Tour construction (EREW PRAM)

```
Input: tree $T = (V, E)$ of $n$ nodes
Output: Euler tour $a_1, \ldots, a_{2 \cdot n-2}$
Auxiliary: path $p_1, \ldots, p_{2 \cdot n-2}$, ranks $r_1, \ldots, r_{2 \cdot n-2}$
foreach $(u, v) \in E$ do in parallel
    create 2 arcs $u - v$ (downgoing) and $v - u$ (upgoing) and arrange
      them into the array $a$ as mentioned above;
end
for $i \leftarrow 1$ to $2 \cdot n - 2$ do in parallel
    $p_i \leftarrow index\ of\ next(opposite(a_i))$;
end
$p_i \leftarrow i$ where $i$ is index of arc $child_{arity(root)}(root) - root$;
list ranking (in: $p$, out: $r$);
reorder $a$ with respect to ranking $r$;
```

List ranking takes $O\left(\frac{n}{p} \cdot log_2\ n\right)$ time using pointer jumping or $O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right)$ using work-optimal algorithm.

Reordering of array with respect to the result of list ranking takes $\frac{n}{p}$ time if executed in parallel using p processors.

The total parallel time is

$$T\left(n, p\right) = 3 \cdot O\left(\frac{n}{p}\right) + O\left(1\right) + O\left(\frac{n}{p} \cdot log_2\ n\right) = O\left(\frac{n}{p} \cdot log_2\ n\right)$$

using pointer jumping or

$$\begin{aligned} T\left(n, p\right) =& 3 \cdot O\left(\frac{n}{p}\right) + O\left(1\right) + O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) \\ =& O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) \end{aligned}$$

using work-optimal list ranking.

The speedup of this algorithm is

$$S\left(n, p\right) = \frac{n}{\frac{n}{p} \cdot log_2\ n} = \frac{p}{log_2\ n}$$

using pointer jumping or

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

using work-optimal list ranking.

Since time complexity is determined by time complexity of list ranking, cost will be the same for algorithm 15 as for the used list ranking algorith. This means

$$C(n,p) = O(n \cdot log_2\ n) = \omega(SU(n))$$

in the case of pointer jumping and

$$C(n,p) = O((n + p \cdot log_2\ p) \cdot log_2\ log_2\ n) = \omega(SU(n))$$

and thus cost-optimality is dependent on cost-optimality of the list ranking algorithm.

Parallel work is

$$W(n,p) = 4 \cdot O(n) + W_{list\ ranking}(n,p)$$

thus the work-optimality is fully dependent on the used list-ranking algorithm.

If pointer jumping is used then

$$W(n,p) = O(n) + O(n \cdot log_2\ n) = O(n \cdot log_2\ n) = \omega(SU(n))$$

this algorithm is not work-optimal but in the case that work-optimal list ranking is used

$$W(n,p) = O(n) + O(n) = O(n) = \Theta(SU(n))$$

the Euler tour construction is work-optimal too.

2.4.2 Implementations

There are no implementations of the Euler Tour construction in the C++ standard libraries, Boost libraries nor any other popular libraries. Though for the sequential solution Boost *boost::depth_first_search* or any other implementation of DFS that allows modification of behaviour of this function can be used.

2.4.3 Applications

There are many application for Euler Tour technique.

The most important one is computation of depths of each node of the tree in parallel. This application is formulated in algorithm 16.

Algorithm 16: Get depths of each node of the tree (EREW PRAM)

```
Input: tree $T = (V, E)$ of $n$ nodes
Output: array of depths $d_1, \ldots, d_n$
Auxiliary: array of arcs $a_1, \ldots, a_{2 \cdot n-2}$, temporary array $t_1, \ldots, t_{2 \cdot n-2}$
Euler tour construction (in: $T$, out: $a$);
for $i \leftarrow 1$to $2 \cdot n - 2$ do in parallel
    if $a_i$ is downgoing then
        $t_i \leftarrow 1$;
    else
        $t_i \leftarrow -1$;
    end
end
inclusive scan (in: $t$, out: $t$);
for $i \leftarrow 1$ to $2 \cdot n - 2$ do in parallel
    if $a_i$ is downgoing then
        $d_{target(a_i)} \leftarrow t_i$;
    end
end
```

The last loop of this algorithm has potentially a lot of write-write conflicts. But all arcs that targets the same node x will hold the same value. Thus this conflict is not really a problem and this algorithm could be applied to EREW PRAM computation model.

Similar loops (in terms of complexity) and inclusive scan are used inside the Euler tour construction thus this algorithm has the same complexities as the Euler tour construction itself.

2.5 Parentheses matching

Problem of parentheses matching is defined by definition 1.74.

For the purpose of simplicity only well-formed strings of parentheses are taken into account. But all algorithms can be modified to be able to work with not well-formed strings.

2.5.1 Algorithms

The sequential solution of parentheses matching problem can be found in single pass utilising stack.

String of parentheses is read from left (lower indices) to right (higher indices), each left parenthesis is pushed to the stack and each right parenthesis is matched with left parenthesis on top of the stack, left parentheses is popped from the stack when it is matched with right parenthesis.

Since the string on the input is well-formed each prefix of this string contains at least the same amount of left parentheses as right parentheses. Thus stack will never be empty when right parenthesis occurs in the input string and because there is the same amount of left and right parentheses in the well-formed string stack will be empty at the end of the string. Thus all parentheses will be matched.

Algorithm 17: Sequential parentheses matching

```
Input: well-formed string of parentheses $p_1, \dots, p_n$
Output: match array $match_1, \dots, match_n$
Auxiliary: stack $S$
$S \leftarrow \emptyset$;
for $i \leftarrow 1$ to $n$ do
    if $p_i$ is left parenthesis then
        $S \leftarrow (S, i)$;
    else
        $t \leftarrow top$, where $S = (S', top)$;
        $S \leftarrow S'$, where $S = (S', top)$;
        $match_i \leftarrow t$;
        $match_t \leftarrow i$;
    end
end
```

This algorithm evaluates string of parentheses in a single pass in which it visits each parenthesis exactly once. Hence the time complexity

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

and because to match each parenthesis, each parenthesis must be visited at least once the lower and thus the upper bound will be

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

too.

There are multiple algorithms to solve parentheses matchin in parallel. Two of them will be presented in this thesis.

The first one utilises properties of parenthesis depth defined as follows. [8]

Definition 2.1 *Parenthesis p is* nested *in parentheses pair l, r if l and r are matching parentheses in a string of parentheses that is in the form*

$$\ldots\ l\ \ldots\ p\ \ldots\ r\ \ldots$$

Depth *of the parenthesis p is the number of parentheses pairs l_i, r_i in which p is nested.*

Observation 2.2 *In a well-formed string a depth of the parenthesis p is equal to the number of unmatched left parentheses in its prefix ending with parentheses p (excluding p).*

Observation 2.3 *Left parenthesis l with depth d_l matches the right parenthesis r with depth d_r if and only if $d_l = d_r$, string is in the form*

$$\ldots\ l\ \ldots\ r\ \ldots$$

and there doesn't exist any parenthesis p with depth d_p such that $d_l = d_p = d_r$ and the string is in the form

$$\ldots\ l\ \ldots\ p\ \ldots\ r\ \ldots$$

Based on the observation 2.3 if stable sort is applied to the array of parentheses with respect to depth, matching parentheses will be adjacent to each other.

Permutation π in the following algorithm denotes function that for each index i assigns index j such that stable sort applied on array $\{a_1, \ldots, a_k\}$ moves element a_i, that is on i-th position, to j-th position.

Inverse permutation π^{-1} in the following algorithm denotes inverse function of π thus $\pi^{-1}(i) = j$ denotes that element that after application of stable sort ends up on i-th position was on j-th position in the original array. Hence the algorithm 18.

All three loops in this algorithm execute $O(1)$ arithmetic operations in each step and runs in parallel thus has time complexity $O\left(\frac{n}{p}\right)$. Since the total number of steps is $\frac{n}{p}$ and in each step p processors are active parallel work of those loops is $O(n)$.

Algorithm 18: Parallel parentheses matching using depth array (EREW PRAM)

```
Input: well-formed string of parentheses $p_1, \ldots, p_n$
Output: match array $match_1, \ldots, match_n$
Auxiliary: depth array $d_1, \ldots, d_n$
for $i \leftarrow 1$ to $n$ do in parallel
    if $p_i$ is left parenthesis then
        $d_i \leftarrow 1$;
    else
        $d_i \leftarrow -1$;
    end
end
inclusive scan (in: $d$, out: $d$);
for $i \leftarrow 1$ to $n$ do in parallel
    if $p_i$ is right parenthesis then
        $d_i \leftarrow d_i + 1$;
    end
end
acquire permutation $\pi$ from stable sort applied to $d$;
for $i \leftarrow 1$ to $\frac{n}{2}$ do in parallel
    $match_{\pi^{-1}(2 \cdot i - 1)} \leftarrow \pi^{-1}(2 \cdot i)$;
    $match_{\pi^{-1}(2 \cdot i)} \leftarrow \pi^{-1}(2 \cdot i - 1)$;
end
```

Inclusive scan has parallel time $O\left(\frac{n}{p} + log_2\ p\right)$ and work $O\left(n\right)$.

Since lower bound of sorting is $\Omega\left(n \cdot log_2\ n\right)$ any parallel stable sort cannot run faster than $\Omega\left(\frac{n \cdot log_2\ n}{p}\right)$ using p processors. Parallel work of such sorting algorithm must be at least $\Omega\left(n \cdot log_2\ n\right)$.

The overall parallel time is

$$T\left(n, p\right) = 3 \cdot O\left(\frac{n}{p}\right) + O\left(\frac{n}{p} + log_2\ p\right) + \Omega\left(\frac{n \cdot log_2\ n}{p}\right) = \Omega\left(\frac{n \cdot log_2\ n}{p}\right)$$

and speedup is

$$S\left(n, p\right) = \frac{n \cdot p}{n \cdot log_2\ n} = \frac{p}{log_2\ n}$$

Since parallel cost of this algorithm is

$$C\left(n, p\right) = p \cdot \Omega\left(\frac{n \cdot log_2\ n}{p}\right) = \Omega\left(n \cdot log_2\ n\right) = \omega\left(SU\left(n\right)\right)$$

this algorithm isn't cost-optimal independently on which stable sorting algorithm is used.

The same applies to the parallel work

$$W(n,p) = 4 \cdot O(n) + \Omega(n \cdot log_2\ n) = \Omega(n \cdot log_2\ n) = \omega(SU(n))$$

Another approach to parallel parentheses matching is to modify algorithm 5.

Input is split into p consecutive segments that are matched sequentially. Unmatched parentheses in each segment forms sequence of right parentheses followed by sequence of left parentheses. Each processor creates list of its unmatched right parentheses and list of its unmatched left parentheses.

Processors then split into pairs such that their segments combined form a consecutive segment of the input. In each pair unmatched left parentheses in the left segment are matched with unmatched right parentheses in the right segment. Three situations can occur

- All these parentheses are matched.
- Some of the left parentheses remain unmatched. Then they're prepended to the unmatched left parentheses of the right segment.
- Some of the right parentheses remain unmatched. Then they're appended to the unmatched right parentheses of the left segment.

Then one of the processors in that pair are no longer needed and total number of segments is halved.

This can be repeated until single segment remains (i.e. input string). This way all parentheses are matched.

In the algorithm 19 the number of processors available p is assumed to be power of 2 for the purpose of simplicity.

Splitting of input to p consecutive segments can be done in $O(1)$ time with $O(1)$ work.

Sequential parentheses matching, arranging unmatched parentheses and counting unmatched parentheses can all be realised in a simple pass of the string, thus their time complexity is $O(n)$ and work $O(n)$.

Algorithm 19: Work-optimal parallel parentheses matching (EREW PRAM)

```
Input: well-formed string of parentheses $s_1, \dots, s_n$
Output: match array $match_1, \dots, match_n$
Auxiliary: unmatched parentheses indices $t_1, \dots, t_n$, # of unmatched
           parentheses $left_1, \dots, left_p$, $right_1, \dots, right_p$ where $p$ is
           number of processors
split $s_1, \dots, s_n$ to $p$ consecutive sections of size $\frac{n}{p}$;
foreach section $s_i, \dots, s_j$ do in parallel
    sequential parentheses matching (in: $s_i, \dots, s_j$, out:
      $match_i, \dots, match_j$);
    arrange unmatched parentheses indices in $s_i, \dots, s_j$ without
      reordering into array $t_i, \dots, t_j$ where unmatched left (resp. right)
      parentheses are in consecutive memory locations and ends at
      index $j$ (resp. starts at index $i$);
    $left_{tid} \leftarrow$ # of unmatched left paretheses;
    $right_{tid} \leftarrow$ # of unmatched right paretheses;
end
for $i \leftarrow 1$ to $log_2\ p$ do
    for $j \leftarrow 1$ to $\frac{n}{2^i}$ 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 $t_{lbase-left_{llo}+1}, \dots, t_{lbase-matched}$ to
              $t_{rbase-rem+1}, \dots, t_{rbase}$;
        else if $right_{rlo} > matched$ then
            $rem \leftarrow right_{llo} - matched$;
            $lbase \leftarrow n \cdot \frac{llo}{p} + left_{llo}$;
            move $t_{rbase+matched}, \dots, t_{rbase+right_{rlo}-1}$ to
              $t_{lbase}, \dots, t_{lbase+rem-1}$;
        end
        $left_{llo} \leftarrow left_{llo} + left_{rlo} - matched$;
        $right_{llo} \leftarrow right_{llo} + right_{rlo} - matched$;
    end
end
```

The first loop hase p parallel steps each of the steps is executing $O\left(\frac{n}{p}\right)$ operations. Hence the parallel time of the first loop $O\left(\frac{n}{p}\right)$. First loop uses $O(n)$ work.

Moving segments of memory in the second loop takes $O(n)$ time with $O(n)$ work.

The inner-most loop has $matched = O(n)$ steps executing $O(1)$ arithmetic operations.

The parallel loop executes $3 \cdot O(n) = O(n)$ operations and since it runs in parallel overall time complexity of the loop is $O(n)$. Parallel work of this loop is also $O(n)$.

Second outer loop has $atworst log_2\ p$ steps and time complexity of each step is $O(n)$ thus the loop takes $O(n \cdot log_2\ p)$ time and also $O(n \cdot log_2\ p)$ work.

Overall time complexity of this algorithm is

$$T(n,p) = O(1) + O\left(\frac{n}{p}\right) + O(n \cdot log_2\ p) = O(n \cdot log_2\ p)$$

and parallel speedup

$$S(n,p) = \frac{n}{n \cdot log_2\ p} = \frac{1}{log_2\ p}$$

Parallel cost of this algorithm is

$$C(n,p) = p \cdot O(n \cdot log_2\ p) = O(p \cdot n \cdot log_2\ p)$$

and parallel work is

$$W(n,p) = O(1) + O(n) + O(n \cdot log_2\ p) = O(n \cdot log_2\ p)$$

This algorithm as is cost and work-optimal only if $p = 1$ (i.e. it work sequentially). Problem of this algorithm is moving array and matching parentheses in linear time inside the parallel loop.

The overall time can be improved by employing unused processors to speedup move and match operations.

This algorithm is in worst-case slower than the sequential algorithm and is not work-optimal but in most cases the move and match operation will run nearly in constant time and the parallel time will get close to $O\left(\frac{n}{p}\right)$ thus this algorithm runs for most of the input strings faster than the sequential algorithm.

This algorithm is also much more efficient in using processors and is closer to work-optimality than the algorithm 18. It will also run faster when less processors are available than the algorithm 18.

2.5.2 Implementations

There are no implementations of parentheses matching in C++ standard library nor Boost library.

2.6 Run of k-local DFTA

To run a k-local DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$ for ground term $t \in T(\mathcal{F})$ means to evaluate expression $\widehat{\Delta}(t)$, where $\widehat{\Delta}$ is extended transition function of DFTA A as defined by definition 1.37.

To evaluate a node, all its childs must be evaluated first. This can be achieved by modifying DFS. Each node is evaluated right before it is closed in DFS traversal of the tree.

Even though this approach is applicable for both sequential and parallel solution, the parallel solution wouldn't be optimal and better parallel solution will be presented in this thesis.

Tree in the algorithm 20 is represented as mentioned in analysis 2.1.2 as a 3-tuple consisting of *labels* array, *children* array and *root* node.

Algorithm 20: Sequential run of k-local DFTA

```
Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, Tree $t = (labels, children, root)$ of
       size $n$
Output: state array $state_1, \dots, state_n$
foreach $child \in children_{root}$ do
    run (in: $A$, $t' = (labels, children, child)$, out: $state$);
end
$state_{root} \leftarrow \Delta_{labels_{root}}(state_{i_1}, \dots, state_{i_{arity(root)}}), \forall i_j$ $i_j$ is index of
  $j$-th child of $root$;
```

Since this algorithm is just a simple modification of DFS with addition of state assignment that takes $O(1)$ time, the overall time complexity is the same as for DFS, that is

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

and since to assign state to each node each node must be visited at least once the lower bound of the run of k-local DFTA is

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

The sequential algorithm above is applicable for any DFTA not only k-local DFTA.

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 Main algorithm

Even though the layers separated by at least $k-1$ layers don't affect each other, to compute correctly each layer $k-1$ layers below are needed.

That means to compute state of nodes in layer i layers $i+1, \dots i+k-1$ are neeeded and even if layers $i+k, \dots$ don't affect results of layer i directly, they may affect results of layers below i that are needed.

This problem can be solved by computing the states in 2 passes. In the first pass states are set to arbitrary value. This is a synchronization pass, that is used to obtain correct initial state for each node. (This is possible thanks to the k-locality.) The second pass has already the correct initial states and thus computes all the states correctly.

To ease those computations the input tree is linearized. To benefit from k-locality as described above the order of nodes in the linear representation (array) is determined by *depth mod k* and is in ascending order.

All nodes with same *depth mod k* may be computed fully in parallel, but there will typically be lesser processors available. To use those processors effectively and to ease synchronization before computation of states step array will be precomputed expressing in which step will be each node computed.

Since complexity analysis of the algorithm 21 depends on used subroutines that are analysed below, the overall analysis is located in subsection 2.6.5.

Algorithm 21: Parallel run of k-local DFTA (EREW PRAM)

```
Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, Tree $t = (labels, children, root)$ of
       size $n$
Output: state array $state_1, \dots, state_n$
Auxiliary: depth array $depth_1, \dots, depth_n$, step array
           $step_1, \dots, step_n$, depth-mod-k order $dmk_1, \dots, dmk_n$
get depths using ETT (in: $t$, out: $depth$);
depthModKSort (in: $t$, $depth$, out: $dmk$);
computeStep (in: $dmk$, $depth$, out: $step$);
$state_i \leftarrow 0, \forall i \in \widehat{n}$;
computeState (in: $A$, $t$, $dmk$, $step$, out: $state$);
computeState (in: $A$, $t$, $dmk$, $step$, out: $state$);
```

2.6.2 Depth-mod-k sort

To acquire depth-mod-k sort order is to obtain order of a modified BFS traversal of the tree. Instead of traversing layer by layer, layer $i + k$ follows after layer i or if the $i + k$ is not a layer then layer $(i \ mod \ k) + 1$ follows instead.

The order starts on layer $k-1$ and ends with deepest layer such that $i \ mod \ k = k - 1$.

This order forms a linked list. This linked list can be obtained by modifying a parentheses matching.

Each arc in euler tour which is downgoing could be represented as right parenthesis and each which is upgoing as left parenthesis. This representation will form a sequence of parentheses starting with sequence of right parentheses and ending of left parentheses.

To obtain a well-formed string this sequence must be wrapped with parentheses. Number of these wrapping parentheses is equal to the total height of the tree. This wrapped sequence will be a well-formed string of paretntheses.

This is thanks to the fact that each edge is represented by a pair of opposite arcs. Depth of the parentheses is equal to the depth of the arc and thus on each layer there will be only one unmatched left and one unmatched right parenthesis. Those are matched by wrapping parentheses.

Those wrapping paretheses also helps to identify first and last arc of the layer since they are matched with them.

Once those parentheses are matched, right parentheses that are representing arcs (i.e. not a wrapping parentheses) are linked to the opposite arc instead of the matching parenthesis. This way a linked list for each layer is created, going from the left-most arc to the right-most arc.

The final step is to relink arcs on the end of the layer to the next layer in depth-mod-k order as described before.

Finally list ranking of this linked list can be used to create a depth-mod-k order array.

```
Algorithm 22: depthModKSort (EREW PRAM)
  Input: Tree $t = (labels, children, root)$ of size $n$, depths array
         $depth_1, \dots, depth_n$
  Output: depth-mod-k order $dmk_1, \dots, dmk_n$
  Auxiliary: euler tour $e_1, \dots, e_{2 \cdot n-2}$, parentheses string
             $par_{-max(depth)}, \dots, par_{2 \cdot n-2+max(depth)-1}$, successor array
             of arcs $next_{-max(depth)}, \dots, next_{2 \cdot n-2+max(depth)-1}$,
             ranking $r_{-max(depth)}, \dots, r_{2 \cdot n-2+max(depth)-1}$
  Euler tour construction (in: $t$, out: $e$);
  $height \leftarrow$ reduce (in: $depth$);
  foreach $e_i \in e$ do in parallel
      $par_{i-1} \leftarrow \begin{cases} left, & \text{iff } e_i \text{ is downgoing} \\ right, & otherwise \end{cases}$ ;
  end
  for $i \leftarrow 1$ to $height$ do in parallel
      $par_{-i} \leftarrow left$;
      $par_{|e|-1+i} \leftarrow right$;
  end
  parentheses matching (in: $par$, out: $next$);
  foreach $e_i \in e$ do in parallel
      if $e_i$ is downgoing then
          $next_{i-1} \leftarrow$ index of $opposite(e_i) - 1$;
      end
  end
  for $i \leftarrow 1$ to $height$ do in parallel
      $next_{|e|-1+i} \leftarrow \begin{cases} next_{-i-k}, & \text{iff } i < height - k \\ next_{-i \bmod k} - 2, & \text{iff } i \bmod k \neq k-1 \\ |e| - 1 + i, & otherwise \end{cases}$ ;
  end
  list ranking (in: $next$, out: $r$);
  Using $r$ construct $dmk$ as array of lower nodes of corresponding arcs
    with $root(t)$ added to the beginning of that array;
```

Euler tour construction runs in $O\left(\frac{n}{p} \cdot log_2\ n\right)$ with $O\left(n \cdot log_2\ n\right)$ work using non work-optimal list ranking or $O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right)$ time and $O\left(n\right)$ work using work-optimal list ranking.

Parallel reduction has parallel time $O\left(\frac{n}{p} + log_2\ p\right)$ and parallel work $O\left(n + p \cdot log_2\ p\right)$ if is modified to be work-optimal as described in analysis 2.2.1.

Parentheses matching could be done in $O\left(\frac{n}{p} + log_2\ p\right)$ with work $O\left(n + p \cdot log_2\ p\right)$.

List ranking can be done in $O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right)$ with parallel work $O\left(n\right)$. Or if the non work-optimal algorithm is used the parallel time is $O\left(\frac{n}{p} \cdot log_2\ n\right)$ and work is $O\left(n \cdot log_2\ n\right)$.

All the other lines are (or can be implemented as) simple loops with assignments without conflicts and thus can be done in $O\left(\frac{n}{p}\right)$ with $O\left(n\right)$ work.

For work-optimal list ranking it has parallel time

$$\begin{aligned} T\left(n,p\right) =& 2 \cdot O\left(\frac{n}{p} + log_2\ p\right) + 2 \cdot O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) + 5 \cdot O\left(\frac{n}{p}\right) \\ =& O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) \end{aligned}$$

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

parallel cost is

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

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

The parallel work of this algorithm is

$$W(n,p) = 7 \cdot O(n) + 3 \cdot O(n + p \cdot log_2\ p) = O(n + p \cdot log_2\ p)$$
$$W\left(n, \frac{n}{log_2\ n}\right) = O\left(n + \frac{n}{log_2\ n} \cdot log_2\ \frac{n}{log_2\ n}\right) = O(n+n) = O(n)$$

Alternative to this is using the nom work-optimal list ranking, then the parallel time is

$$T(n,p) = 2 \cdot O\left(\frac{n}{p} \cdot log_2\ n\right) + 2 \cdot O\left(\frac{n}{p} + log_2\ p\right) + 5 \cdot O\left(\frac{n}{p}\right) = O\left(\frac{n}{p} \cdot log_2\ n\right)$$
$$T\left(n, \frac{n}{log_2\ n}\right) = O\left(\frac{n \cdot log_2\ n}{n} \cdot log_2\ n\right) = O\left((log_2\ n)^2\right)$$
$$T(n,n) = O\left(\frac{n}{n} \cdot log_2\ n\right) = O(log_2\ n)$$

Using non work-optimal list ranking is faster, but only if more processors are available. The parallel cost is

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

which is worse than in case of work-optimal list ranking.

This approach results in parallel work

$$W(n,p) = 5 \cdot O(n) + 3 \cdot O(n + p \cdot log_2\ p) + 2 \cdot O(n \cdot log_2\ n) = O(n \cdot log_2\ n)$$

Non work-optimal list ranking may be faster, but is not optimal and needs more processors to be actually faster. If speed is priority and $\Omega\left(\frac{n}{log_2\ log_2\ n}\right)$ processors are available, than non work-optimal list ranking is better choice, otherwise work-optimal list ranking is better.

2.6.3 Step computation

Step computation is realised using depth-mod-k order of the nodes and their depths.

At first nodes in depth-mod-k order are split into k groups based on their *depth mod k*. These groups are consecutive in depth-mod-k order.

Those groups are split into consecutive subgroups of size p (excluding the left-most subgroup which may have size in range 1 to p). Those subgroups are numbered from right to left beginning from 1. Number of the subgroup represents the step in which the group is computed.

The inclusive suffix scan in the algorithm 23 is inclusive scan that performs scan in reverse order (i.e. for input $x_1, \ldots, x_n$ is it the same as performing inclusive (prefix) scan for input $x_n, \ldots, x_1$).

p in the algorithm 23 denotes number of processors available.

Algorithm 23: computeStep (EREW PRAM)

```
Input: depth-mod-k order $dmk_1, \ldots, dmk_n$, depths array
       $depth_1, \ldots, depth_n$
Output: step array $step_1, \ldots, step_n$
Auxiliary: group $group_1, \ldots, group_n$, group end index $gei_1, \ldots, gei_n$
for $i \leftarrow 1$ to $n$ do in parallel
    $group_i \leftarrow depth_{dmk_i}\ mod\ k$;
    if $i = n - 1 \lor group_i \neq group_{i+1}$ then
        $gei_i \leftarrow i$;
    else
        $gei_i \leftarrow \infty$;
    end
end
inclusive suffix scan using $min$ operator (in: $gei$, out: $gei$);
for $i \leftarrow 1$ to $n$ do in parallel
    if $(gei_i - i)\ mod\ p = 0$ then
        $step_i \leftarrow 1$;
    else
        $step_i \leftarrow 0$;
    end
end
inclusive suffix scan (in: $step$, out: $step$);
```

Algorithm composes of 2 inclusive scans with parallel time $O\left(\frac{n}{p} + log_2\ p\right)$ and work $O\left(n\right)$ and 2 simple loops with assignments which have parallel time $O\left(\frac{n}{p}\right)$ and work $O\left(n\right)$.

Thus the parallel time of the step computation has parallel time

$$T\left(n, p\right) = 2 \cdot O\left(\frac{n}{p} + log_2\ p\right) + 2 \cdot O\left(\frac{n}{p}\right) = O\left(\frac{n}{p} + log_2\ p\right)$$

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

Parallel cost of this algorithm is

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

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

And parallel work is

$$W(n,p) = 4 \cdot O(n) = O(n)$$

2.6.4 State computation

State computation traverses the nodes in depth-mod-k order from right to left. Processors are used effectively, only on boundary between 2 groups some processors may stay.

If p processors are used maximal number of processors that will stall is $p - 1$. There are $k - 1$ places where processors may stall thus maximal wasted work will be $(p-1) \cdot (k-1) = O(p \cdot k)$.

pid in the algorithm 24 denotes id of executing processor starting with 0 and p denotes total number of processors.

Algorithm 24: computeState (EREW PRAM)

```
Input: DFTA $A = (Q, \mathcal{F}, Q_f, \Delta)$, Tree $t = (labels, children, root)$ of
       size $n$, depth-mod-k order $dmk_1, \ldots, dmk_n$, step array
       $step_1, \ldots, step_n$
Output: state array $state_1, \ldots, state_n$
do in parallel
    $s \leftarrow 1$;
    $i \leftarrow n - pid$;
    while $i \geq 1$ do
        if $step_i = s$ then
            $state_i \leftarrow \Delta_{labels_i}(state_{child_1(i)}, \ldots, state_{child_{arity(i)}(i)})$;
            $i \leftarrow i - p$;
        end
        $s \leftarrow s + 1$;
    end
end
```

The evaluation of Δ is required to take $O(1)$ time, that is ensured by transition function (For more info see implementation 3.2.4).

The while loop executes $O\left(1\right)$ operations in each step. Total number of steps is $O\left(\frac{n}{p}\right)$ since each processors starts at $\Theta\left(n\right)$ index, each step decreases index value by $O\left(p\right)$ and stalls maximaly $O\left(k\right) = O\left(1\right)$ times.

Since everything is executed in parallel the total parallel time is

$$T\left(n, p\right) = O\left(\frac{n}{p}\right)$$

the cost is

$$C\left(n, p\right) = p \cdot O\left(\frac{n}{p}\right) = O\left(n\right)$$

and parallel work is

$$W\left(n, p\right) = p \cdot \frac{n}{p} = O\left(n\right)$$

since there are $O\left(\frac{n}{p}\right)$ steps and in each step p processors are active.

2.6.5 Complexity analysis

ETT and depthModKSort are dependent on used list ranking algorithm. They can run work-optimally but potentially slower, or faster but not work-optimally. Both variants will be analysed. For non work-optimal variant both those algorithms run in $O\left(\frac{n}{p} \cdot log_2\ n\right)$ time with $O\left(n \cdot log_2\ n\right)$ work. And for work-optimal variant those algorithms have parallel time
$O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right)$ with $O\left(n\right)$ work.

Compute step has parallel time $O\left(\frac{n}{p} + log_2\ p\right)$ and parallel work $O\left(n\right)$.

Compute state runs with $O\left(\frac{n}{p}\right)$ time and $O\left(n\right)$ work.

Assignment of arbitrary state can be achieved in $O\left(\frac{n}{p}\right)$ parallel time with $O\left(n\right)$ work.

Overall total parallel time of this algorithm is

$$\begin{aligned} T\left(n,p\right) &= 2 \cdot O\left(\frac{n}{p} \cdot log_2\ n\right) + O\left(\frac{n}{p} + log_2\ p\right) + 3 \cdot O\left(\frac{n}{p}\right) \\ &= O\left(\frac{n}{p} \cdot log_2\ n\right) \end{aligned}$$

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

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

using non work-optimal list ranking, or

$$\begin{aligned} T\left(n,p\right) &= 2 \cdot O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) + O\left(\frac{n}{p} + log_2\ p\right) + 3 \cdot O\left(\frac{n}{p}\right) \\ &= O\left(\left(\frac{n}{p} + log_2\ p\right) \cdot log_2\ log_2\ n\right) \end{aligned}$$

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

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

thus the non work-optimal list ranking could run faster than the work-optimal variant, but needs more processors and as will be shown below it is wasting work of processors. It also runs slower when there are not enough processors. There needs to be at least $\Omega\left(\frac{n}{log_2\ log_2\ n}\right)$ processors to run faster than work-optimal variant.

The parallel cost for non work-optimal variant is

$$C\left(n,p\right) = p \cdot O\left(\frac{n}{p} \cdot log_2\ n\right) = O\left(n \cdot log_2\ n\right) = \omega\left(SU\left(n\right)\right)$$

and for the cost-optimal variant

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

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

thus neither of those two variants are cost-optimal, but the work-optimal variant has better cost than the other one.

Tha parallel work for non work-optimal variant is

$$W\left(n,p\right) = 2 \cdot O\left(n \cdot log_2\ n\right) + 4 \cdot O\left(n\right) = O\left(n \cdot log_2\ n\right) = \omega\left(SU\left(n\right)\right)$$

thus using non work-optimal list ranking causes that the whole algorithm is not work-optimal. For the work-optimal list ranking the parallel work is

$$W\left(n,p\right) = 6 \cdot O\left(n\right) = O\left(n\right) = \Theta\left(SU\left(n\right)\right)$$

and the algorithm is work-optimal then.

The complexity of parallel run of k-local DFTA thus depends on the selection of list ranking algorithm. For most of the cases the work-optimal list ranking is better choice but if speed is the priority and $\Omega\left(\frac{n}{log_2\ log_2\ n}\right)$ processors are available the non work-optimal list ranking could have better results.

2.6.6 Implementations

Since this algorithm is very specific and new algorithm there are no implementations of libraries that author of this thesis is aware of.

Chapter 3

Implementation

In this chapter available libraries for parallelisation will be briefly described and library used by this work will be compared to others. Then implementation notes will follow.

3.1 Libraries

Support for parallelism could be ensured using POSIX threads (*pthread*) standard. This standard offers basic thread manipulation and synchronization primitives.

Standard C++ library offers its own interface for *pthread* that is spread through multiple classes and functions (e.g. *std::thread*,*std::mutex*, etc.).

This brings total control of how is threading implemented and those simple building blocks (thread, mutex, etc.) are fully sufficient to build any parallel application. But on the other hand this simple implementation imposes responsibility for synchronization and threads creation and joining on the programmer and every simple task must be solved by the programmer.

Another library that supports parallelism is Threading Building Blocks (TBB) that implements all that is implemented in pthread and on top of it some of the parallel algorithms (e.g. reduction, scan, sort, ...). The most important feature of TBB is implemented workload balancing that distributes work amongst processors to optimize run of implemented algorithms.

Another possibility is to use OpenMP[10] library, that offers simple interface for parallel programming. OpenMP unlike the others uses preprocessor directives instead of template functions. Programmer doesn't have full control over final code but OpenMP manages threads and partially synchronization on its own as described by the directives. This library offers most comfort thanks to the fact it solves trivial problems on its own and offers straightforward parallelization of existing sequential code using directives.

OpenMP is also very useful on distributed systems when combined with OpenMPI.

Since there's no need to have full control over compiled code and OpenMP offers most comfort and for potential future extension for distributed system all parallel algorithms in this thesis will be implemented using OpenMP library.

3.2 Structures

3.2.1 Array

Since the implementations in standard C++ libraries aren't designed for parallel use and Boost implementation is designed especially for Boost function array will be implemented in this thesis as class *borovmi5::types::ParallelArray*.

This will be a wrapper around C-like arrays conforming to *Container* C++ named requirement.

ParallelArray will be similar to *std::vector* and *std::array* but most operations (e.g., initialization, assignment, comparation, ...) will be executed in parallel in contrast with standard C++ implementations.

The ParallelArray will be dynamically allocated but won't allow (for the purpose of simplicity) adding or removing elements after construction.

Since all basic operations over the array are without any read/write conflicts, they can be implemented in $O(n/p)$ time using $p < n$ processors or $O(1)$ time using $p \geq n$ processors on EREW PRAM.

3.2.2 Tree

Tree will be implemented in class *borovmi5::types::Tree.*

To represent arrays in the tree *borovmi5::types::ParallelArray* will be used.

3.2.3 Arc

Arc will be implemented in class *borovmi5::types::Arc*.

3.2.4 DFTA

DFTA will be implemented in class *borovmi5::dfta::DFTA*.

A finite set of states will be represented by it's state with greatest number.

A finite set of final states is subset of the set of states with no other limitations. Thus can be implemented using array of bools indexed by state numbers. Value stored under state number represents whether the given state is final or not. This method allows $O(1)$ resolution whether the state is final.

The second possibility is to represent the set of final states with *std::set*, assuming that most of the sets of final states will be relatively small, this is a more space-efficient solution but provides worse resolution time $O(log_2\ n)$. But because this resolution takes place only once during the whole run of the DFTA and assuming the final state set is small this fact is negligible.

Implementation in this work will use *std::set* to represent set of final states.

To represent transition function *std::unordered_map* will be used that will assign symbol-specific transition function to the symbols. The unordered map (hash map) is used to conform the requirements of algorithm 24 that evaluation of transition function has time complexity $O(1)$.

Symbol-specific transition function will be represented by class *borovmi5::dfta::DFTA::TransitionTable* that will be implemented as an array indexed by child states. To neglect the fact that each TransitionTable may have different arity, the transition table is flattened and member function *borovmi5::dfta::DFTA::TransitionTable::transition* accepting vector of children states handles the indexation.

3.3 Reduction and Scan

3.3.1 Reduction

Despite the fact there exist many sufficient implementations of reduction, both sequential and parallel reduction will be implemented in this thesis as function *borovmi5::scans::reduce* that will have an interface corresponding to other implemented functions. But this implementation will be very similar to the implementation in standard C++17 libraries and could be replaced with it.

The algorithm 2 is not work-optimal itself but could be made work-optimal in the same manner as algorithms 5 and 8. Using $\frac{n}{\log_2 n}$ processors and splitting input into smaller portions precomputed sequentially by individual processors.

Parallel variant of *borovmi5::scans::reduction* will be implemented using the work-optimal modification of the parallel algorithm mentioned above.

3.3.2 Inclusive scan

Implementation included in standard and Boost libraries would be sufficient for this thesis but despite that, an inclusive scan will be implemented in this thesis as a function *borovmi5::scans::inclusiveScan.* All three variants (sequential, modified Hillis-Steele and Blelloch) will be implemented and compared in chapter Testing.

The algorithm 4 is designed for CREW PRAM and includes read/write conflict. This problem could be easily solved using simple synchronization. The implementation will read current values in the first place, then synchronize threads using synchronization barrier and after that will write new values. This modification makes the Hillis-Steele algorithm EREW PRAM algorithm.

3.3.3 Exclusive scan

The exclusive scan will be implemented as function *borovmi5::scans::exclusiveScan* in the same way as inclusive scan using the same algorithms.

3.3.4 Segmented scan

There doesn't exist implementation of segmented scan in standard nor boost library. Thus inclusive and exclusive segmented scan will be implemented in this thesis as functions *borovmi5::scans::segmentedInclusiveScan* and *borovmi5::scans::segmentedExclusiveScan*. Of parallel variants of the algorithm, only Hillis-Steele will be implemented because it is as good as the Blelloch algorithm.

3.4 Lists

3.4.1 Linked list

Since a dynamic linked structure is not an optimal approach for a parallel implementation of algorithms over a linked lists the standard implementation *std::list* won't be used. Instead a *borovmi5::ParallelArray* will be used to represent linked lists where values stored in this array will be indices of successors.

3.4.2 k-coloring

k-coloring will be implemented in this thesis. There will be 3 functions.

The first one will be *borovmi5::ranking::coloring::nColoring* that will assign unique color to each node of the list.

The second one will be *borovmi5::ranking::coloring::sixColoring* that will use the first function to generate n-coloring and then reduce it to 6-coloring using DCT as described in algorithm 12.

The last one will be *borovmi5::ranking::coloring::threeColorint* that will use the second function to generate 6-coloring and then reduce it to 3-coloring using algorithm 13.

3.4.3 List Ranking

List ranking will be implemented in this thesis as a function
borovmi5::ranking::listRanking using all three mentioned alorithms
(i.e. 9, 10, 14).

All three algorithms in analysis assigns ranks in range $[1, n]$ but since arrays in C++ are indexed starting with 0, algorithms implemented in this thesis will assign ranks in range $[0, n-1]$.

In the case of sequential algorithm it means to assign the rank to the node before incrementing it thus using exclusive scan instead of inclusive scan.

In the case of parallel algorithms initial rank of the head will be set to 0 instead of 1.

None of the functions will take head index as its argument. Implementations of the list ranking in this thesis will find the head on their own. Sequential solution will scan the linked list first to find the head in $O(n)$ time. Parallel functions will do the same but in parallel in $O\left(\frac{n}{p}\right)$ time.

This can be slower than just accepting head as parameter of the function, but will ensure that head will be really head of the list and doesn't need that user of the function finds head first.

3.5 Euler Tour Technique

Parallel solution of Euler tour construction will be implemented in this thesis as function *borovmi5::algorithm::EulerTour*.

To arrange arcs into the array as described in analysis the indices of starts of portions of arcs originated in each node will be precomputed applying exclusive scan on array of child counts of each node. Result of this scan will be an array that will assign base index to each node.

The fuction *next* mentioned in analysis will be replaced with an array *next* that will assign each arc an arc corresponding to the behaviour of function *next*. This will be achieved in 2 steps.

The first one will take care of downgoing arcs only. For each arc $x - child_i(x)$ that will be stored at index $baseIndex_x + 2 \cdot (i - 1)$ the value $baseIndex_x + 2 \cdot (i)$ will be assigned in the case that $child_i(x)$ isn't the right-most child of x or $baseIndex_x$ (index of the 1st (left-most) childs) otherwise.

The second step will take care of upgoing arcs. For each upgoing arc $x - parent(x)$ that will be stored at index $baseIndex_{parent(x)} + 2 \cdot (i - 1) + 1$ (if $x = child_i(parent(x))$) the *next* value for the arc $x - child_{arity(x)}(x)$ will be set to the index of this arc and *next* value for this arc will be set to the index of the arc $x - child_1(x)$. If x has no childs, then *next* value for this arc will be set to the index of this arc.

The algorithm 16 will be impemented as function *borovmi5::dfta::getDepths* since it will be needed for run of DFTA.

3.6 Parentheses matching

Parentheses matching will be implemented as function *borovmi5::algorithm::matchParentheses*.

All three presented algorithms will be implemented

Each segment of the input string will be processed in a single pass instead of multiple passes as in the algorithm 19.

Move and match in parallel loop of the algorithm 19 will be implemented to run in parallel and use free processors to speedup overall runtime of the program.

Parallel merge sort is used as stable sort mentioned in algorithm 18. This is implemented as simple parallelisation of merge sort. When input is split into two parts that are then processed in merge sort they are processed in parallel.

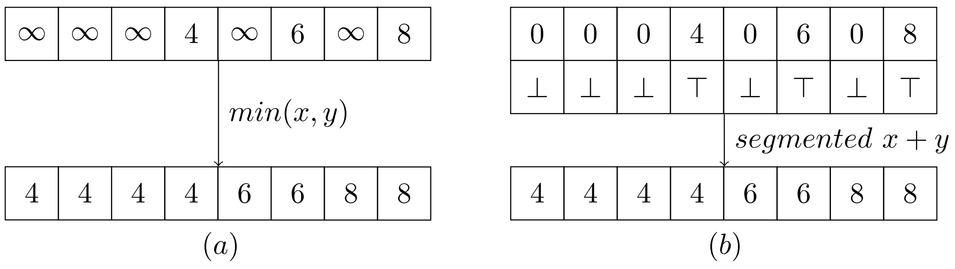


Figure 3.1: (a) inclusive scan with min operator (b) segmented inclusive scan with + operator

3.7 Run of k-local DFTA

The run of k-local DFTA will be implemented in both its sequential and parallel form.

3.7.1 Main algorithm

The main algorithm will be represented by function *borovmi5::dfta::run*.

The main algorithm will construct Euler tour, use it to compute depths and pass it to the depthModKSort and thus will avoid redundant Euler tour construction.

3.7.2 Depth-mod-k sort

The depthModKSort will be implemented as function *borovmi5::dfta::depthModKSort*. It will be implemented with a little modification mentioned above.

3.7.3 Step computation

Step array computation will be realised as function *borovmi5::dfta::computeSteps*. Due to difficulties with using *min* function in inclusive scan as is implemented, segmented inclusive scan will be used instead. The segments will be regions with the same group end index.

Figure 3.1(a) depicts how is the group end index distributed in algorithm 24 and figure 3.1(b) depicts how is it realised in implementation.

3.7.4 State computation

State computation traversal is implemented as function *borovmi5::dfta::computeState*.

Chapter 4

Testing

In this chapter unit testing of individual support functions will be described followed by a system test of the run of k-local DFTA.

4.1 Unit Tests

Unit tests will be used to test the functionality of support functions in this thesis. Unit tests will be implemented using the GoogleTest framework.

4.1.1 Reduction and Scans

Unit tests for reduction and all scans are implemented in folder *test/functionality/scans* as files *reduce_test.cpp*, *inclusive_scan_test.cpp*, *exclusive_scan_test.cpp*, *segmented_inclusive_scan_test.cpp* and *segmented_exclusive_scan_test.cpp*. Each file includes corresponding Test Suite. Each Test Suite consist of 9 tests. Individual tests are described in the table 4.1.

Test	Input size	Description
empty	0	empty input
zeroes	100	input is sequence of 0 (i.e. $0 \ \dots \ 0$)
simple	100	input is sequence of 1 (i.e. $1 \ \dots \ 1$)
negative	100	input is sequence of -1 (i.e. $-1 \ \dots \ -1$)
alternating	100	input is alternating sequence of -1 and 1 (i.e. $1 \ -1 \ \dots \ 1 \ -1$)
randomSmall	100	random input sequence
randomMedium	10.000	random input sequence
randomLarge	1.000.000	random input sequence
fullRandom	random from $[100, 1.000.000]$	random input sequence, 10 repetitions

Figure 4.1: Reduction and scans unit tests

Test	Input size	Description
empty	0	empty input
simple	100	list $1 \rightarrow 2 \rightarrow \cdots \rightarrow 99 \rightarrow 100$
reverse	100	list $100 \rightarrow 99 \rightarrow \cdots \rightarrow 2 \rightarrow 1$
randomSmall	100	random input sequence
randomMedium	10.000	random list
randomLarge	100.000	random list
fullRandom	random from $[100, 50.000]$	random list, 10 repetitions

Figure 4.2: Coloring and list ranking unit tests

Test	Input size	Description
empty	0	empty tree
singleNode	1	tree with root node only
simple	100	pre-defined tree
randomSmall	100	random tree
randomMedium	10.000	random tree
randomLarge	1.000.000	random tree
fullRandom	random from $[100, 1.000.000]$	random tree, 10 repetitions

Figure 4.3: Euler tour technique unit tests

4.1.2 Coloring and List ranking

Unit tests for coloring and list ranking are implemented in folder *test/functionality/ranking* as files *coloring_test.cpp* and *list_ranking_test.cpp*. Each file includes corresponding Test Suite. Each Test Suite consist of 7 tests. Individual tests are described in the table 4.2.

Random linked lists together with their correct ranking are acquired by a function *generateLinkedList* in generators.

4.1.3 Euler Tour Technique

Unit tests for Euler tour construction are implemented in file *test/functionality/algorithm/euler_tour_test.cpp*. Test Suite consists of 7 tests. Individual tests are described in the table 4.3.

Random trees together with their correct Euler tour are acquired by a function *generateTreeWithTour* in generators.

Test	Input size	Description
empty	0	empty string
invalid	2	)(
partiallyValid	4	)()(
valid	2	()
simple	24	(((())())()(()))(()(()))
randomSmall	100	random well-formed string
randomMedium	10.000	random well-formed string
randomLarge	1.000.000	random well-formed string
fullRandom	random even from $[100, 1.000.000]$	random well-formed string, 10 repetitions

Figure 4.4: Parentheses matching unit tests

4.1.4 Parentheses matching

Unit tests for Parentheses matching are implemented in file *test/functionality/algorithm/parentheses_matching_test.cpp*. Test Suite consists of 9 tests. Individual tests are described in the table 4.4.

Random well-formed strings together with their correct matching are acquired by a function *generatePars* in generators.

4.1.5 Other

Other functions implemented in this thesis such as log_2^* or merge sort have their unit tests too but will not be described further.

4.2 System test

Parallel and sequential run of k-local DFTA will be tested by system tests using GoogleTest. They will be similar to the unit tests but will be applied on the whole parallel run, not individual parts.

System test is implemented in file *test/functionality/dfta/run_dfta_test.cpp* and includes several subtests described in the table 4.5.

Figure 4.6 depicts pre-defined DFTA used in testing. Figures 4.7 and 4.8 depicts pre-defined trees used in testing. Numbers in nodes of pre-defined trees show states of the nodes after run of pre-defined DFTA.

Test	Tree size	k	Description
empty	0	-	empty tree and DFTA
trivialV	5, 6, 7	3	pre-defined DFTA and pre-defined trees A, B and C
trivialX	6, 7	3	pre-defined DFTA and pre-defined trees D and E
trivial	11	3	pre-defined DFTA and pre-defined tree F
basic	32	3	pre-defined DFTA and pre-defined tree G
randomSmall	100	random from $[3, 5]$	random DFTA and random Tree
randomMedium	10.000	random from $[3, 6]$	random DFTA and random Tree
randomLarge	1.000.000	random from $[3, 8]$	random DFTA and random Tree (may take a long time)

Figure 4.5: Parentheses matching unit tests

DFTA $A = (\{0,1,2,3,4\}, \{a_2, b_1, c_0\}, \{4\}, \Delta$:

$$\begin{aligned} &a(2,3) \to 4 && b(1) \to 3 \qquad c \to 1 \\ &a(3,3) \to 4 && b(q) \to 2,\ q \neq 1 \\ &a(q_1,q_2) \to 0,\ q_1 \notin \{2,3\} \vee q_2 \neq 3 && \end{aligned}$$

)

Figure 4.6: Pre-defined 3-local DFTA

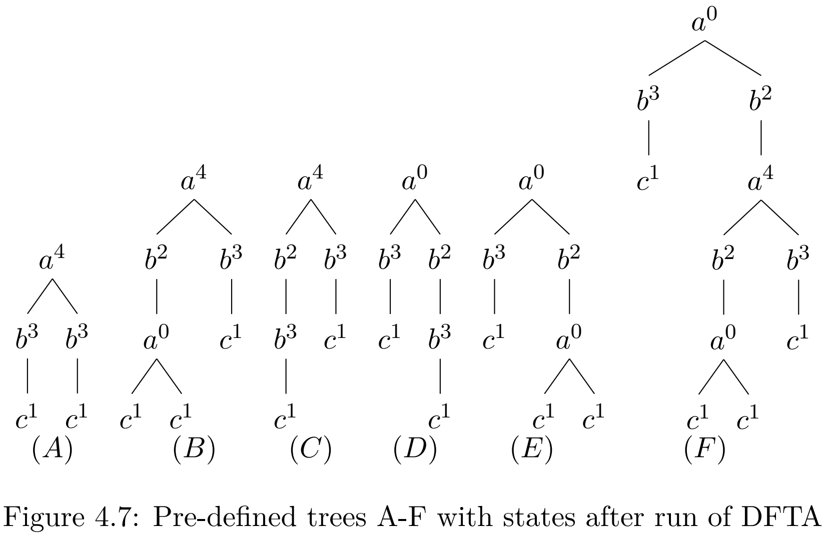


Figure 4.7: Pre-defined trees A-F with states after run of DFTA

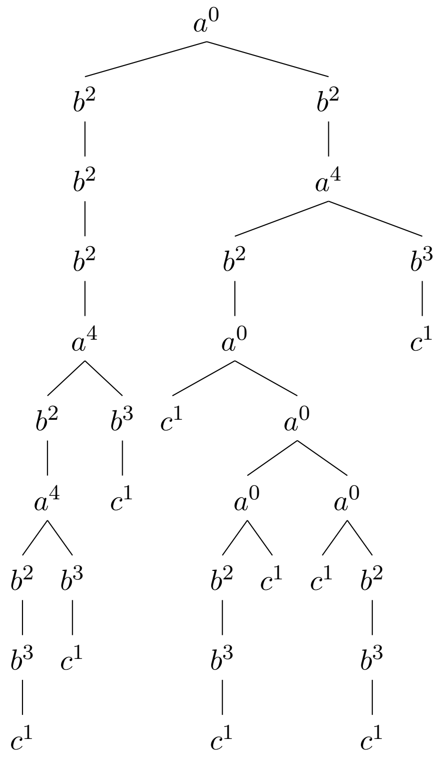


Figure 4.8: Pre-defined tree G with states after run of DFTA

Chapter 5

Time measurements

In this chapter methodology of measuring the execution time of algorithms will be described followed by measurements of execution times of run of k-local DFTA. Execution times of sequential and parallel run will be compared.

5.1 Methodology

2 different times will be measured.

- Real Time - Elapsed time from the beginning to the end of execution of the master thread.
- CPU time - Elapsed time during execution of the program on all processors in all threads.

Real time should be analogue to parallel time and CPU time should be analogue to parallel work.

Three algorithms will be tested in total. Sequential algorithm 20 which will be denoted as S in the tables and figures. Parallel algorithm 21 which will be using algorithm 10 as list ranking algorithm, this one will be denoted by P_F in the tables and figures. And parallel algorithm 21 which will be using work-optimal list ranking algorithm 14, P_W denotes this algorithm in the tables and figures.

Multiple data sets will be tested with each algorithm. Each of those data sets will consist of a single k-local DFTA and a single tree of size n.

Parallel algorithms will be tested for a various number of processors p.

Each test will be repeated multiple times and results will be averaged to mitigate potential measurement error.

5.2 Hardware

All tests will be executed on Intel Xeon scalable processor (Cascade Lake) with 6 vCPUs, base frequency of 3.1 GHz and 32 GB of RAM available.

5.2.1 Test Data

There will be pre-generated k-local DFTAs for $k = 2$, 3, 4, 8, 16 and 24. For each DFTA there will be pre-generated trees of size $n = 256$, 1024, 4096, 16384, 32768, 65536, 131072 and 262144. Each such data set will be tested 5-times. Parallel algorithms will be measured using $p = 2$, 3, 4 and 6 processors.

5.2.1.1 Data Generation

Test data are generated using Algorithms Library Toolkit[9] (ALT).

To generate k-local DFTA, a tree pattern of depth k is generated first using ALT. FTA accepting such pattern is k-local thus NFTA accepting this generated pattern is generated using ALT. This NFTA is then determinized, minimalized and normalized using ALT. This produces k-local DFTA of the desired k.

To generate a ranked tree of size n, an unranked tree of this size is generated using ALT first. Then based on the alphabet of previously generated DFTA this tree is labelled to be valid input for this DFTA.

5.3 Results

Results of time measurements are presented in table 5.1. Column n is for input tree size and column p for processor count.

For each algorithm and data set 2 times are presented. Real time and CPU time.

As shown in the analysis only place where k may affect the time is in the state computation step, where high k may cause slowdown based on the number of processors. This slowdown is maximally by $O(p \cdot k)$ time and since p is constant in individual tests, the slowdown caused by different values of k is $O(k)$. This slowdown caused by the higher value of k is negligible and thus values for the same input tree size are almost the same for different values of k tested in this thesis.

Thus times presented in the table 5.1 and in the charts 5.2 and 5.3 are average of all times across all values of k tested. Each test is repeated 5-times thus each value represents average of $30 = 5 \cdot 6$ tests for given input size n, processor count n and algorithm.

As can be seen from the charts 5.2 and 5.3, all three algorithms seem to have $O(n)$ Real time (which should be equivalent to the parallel time) and $O(n)$ CPU time (which should be equivalent of parallel work).

Both parallel implementations seem to have improving Real time at the cost of CPU time with the increasing number of processors. Increasing work with the increasing number of processors may be caused by the fact that not all algorithms are implemented work-optimally and by the loops where in each iteration more and more processors are becoming inactive and are waiting for other processors to finish their work. This possibly may be solved by refactoring those loops to yield processors that it no longer needs.

The processor count of 6 seems to be worse than 4, it has worse Real time and much worse CPU time. This probably has the same cause as the increasing work and the fact that processor count is not a power of 2 may harm the performance in algorithms that are designed with the presumption of such processor count. It is probable that with the further increasing number of processors the Real time will be improving further and parallel work may be better for 8 processors than for 6 processors.

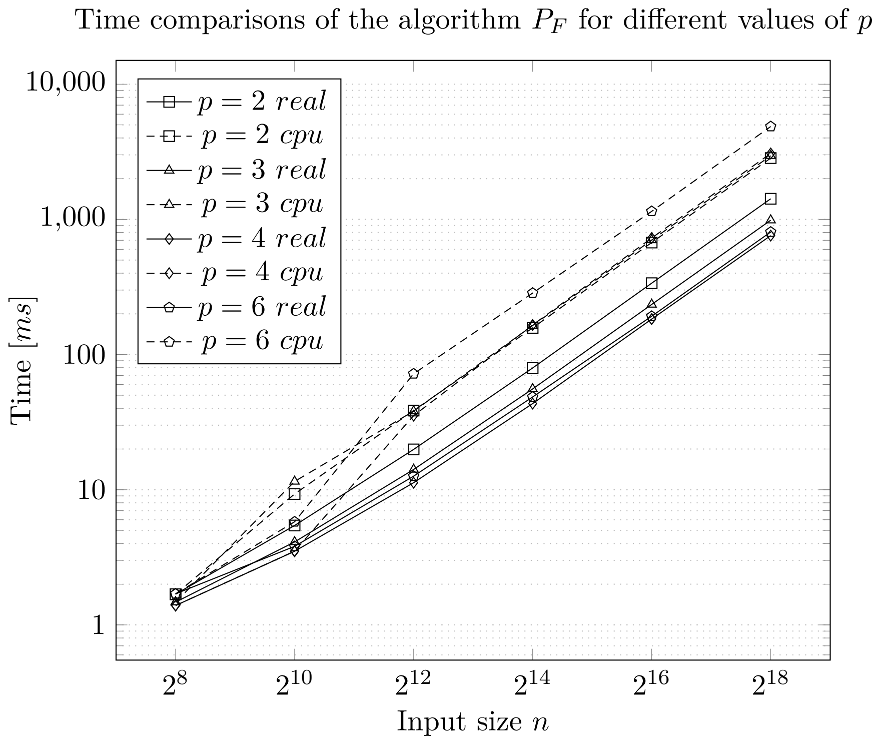
The real time of parallel algorithms seems to be linear instead of logarithmic. This is probably caused by the low number of processors. Both parallel implementations expect a number of processors to be scaling together with input size. But processor count in all tests is fixed.

Since the increasing number of processors seems to lower Real time, it is probable that if the number of processors is scaling together with input size the resulting Real time will be closer to the expected parallel time.

The results look promising that the expected parallel time possibly could be achieved with this implementation, but further testing must be performed to make any conclusions.

n	p	S		P_F		P_W	
		real	*cpu*	*real*	*cpu*	*real*	*cpu*
256	1	$294\mu s$	$294\mu s$	–	–	–	–
	2	–	–	$1.69ms$	$1.69ms$	$2.43ms$	$2.43ms$
	3	–	–	$1.47ms$	$1.47ms$	$2.19ms$	$2.19ms$
	4	–	–	$1.39ms$	$1.39ms$	$2.11ms$	$2.11ms$
	6	–	–	$1.70ms$	$1.70ms$	$2.55ms$	$2.55ms$
1024	1	$1.18ms$	$1.18ms$	–	–	–	–
	2	–	–	$5.43ms$	$9.30ms$	$7.76ms$	$12.5ms$
	3	–	–	$4.11ms$	$11.5ms$	$6.00ms$	$14.0ms$
	4	–	–	$3.50ms$	$3.51ms$	$5.22ms$	$17.2ms$
	6	–	–	$3.81ms$	$5.80ms$	$5.73ms$	$25.7ms$
4096	1	$4.72ms$	$4.72ms$	–	–	–	–
	2	–	–	$19.9ms$	$38.5ms$	$27.5ms$	$52.3ms$
	3	–	–	$14.1ms$	$38.1ms$	$19.9ms$	$57.5ms$
	4	–	–	$11.3ms$	$35.3ms$	$16.1ms$	$64.1ms$
	6	–	–	$12.6ms$	$72.0ms$	$17.9ms$	$97.9ms$
16384	1	$19.0ms$	$19.0ms$	–	–	–	–
	2	–	–	$79.6ms$	$157ms$	$108ms$	$214ms$
	3	–	–	$55.6ms$	$164ms$	$76ms$	$227ms$
	4	–	–	$43.4ms$	$165ms$	$60.4ms$	$240ms$
	6	–	–	$48.5ms$	$285ms$	$66.6ms$	$390ms$
65536	1	$77.4ms$	$77.4ms$	–	–	–	–
	2	–	–	$337ms$	$674ms$	$455ms$	$910ms$
	3	–	–	$235ms$	$703ms$	$321ms$	$961ms$
	4	–	–	$183ms$	$728ms$	$254ms$	$1.01s$
	6	–	–	$192ms$	$1.15s$	$267ms$	$1.60s$
262144	1	$313ms$	$313ms$	–	–	–	–
	2	–	–	$1.42s$	$2.84s$	$1.97s$	$3.95s$
	3	–	–	$986ms$	$2.96s$	$1.39s$	$4.17s$
	4	–	–	$768ms$	$3.07s$	$1.10s$	$4.40s$
	6	–	–	$807ms$	$4.86s$	$1.15s$	$6.95s$

Figure 5.1: Time measurement results



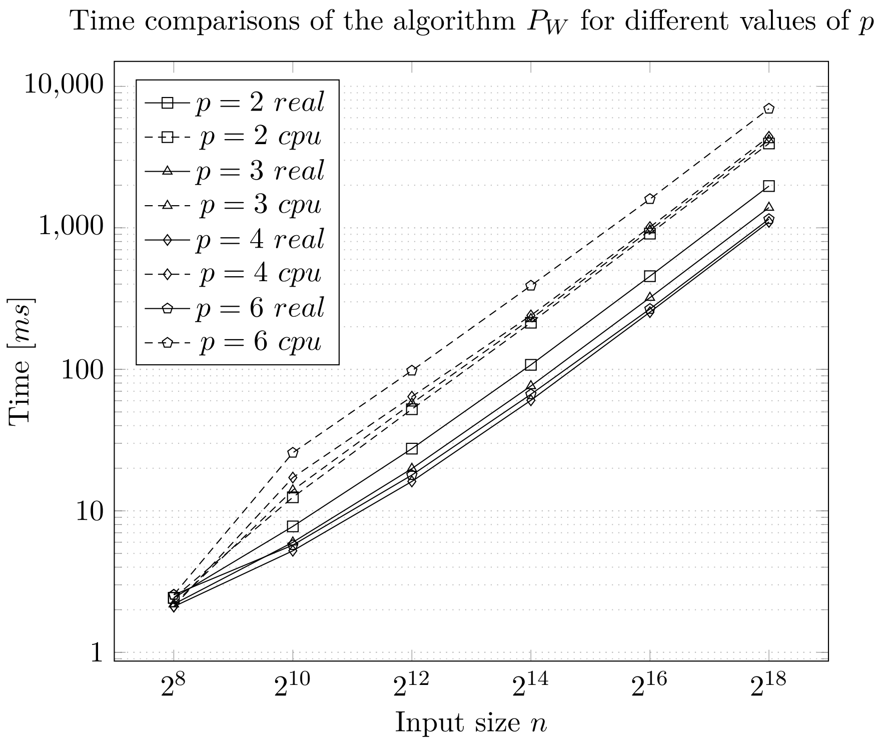


Figure 5.2: Parallel algorithms time comparisons based on number of processors

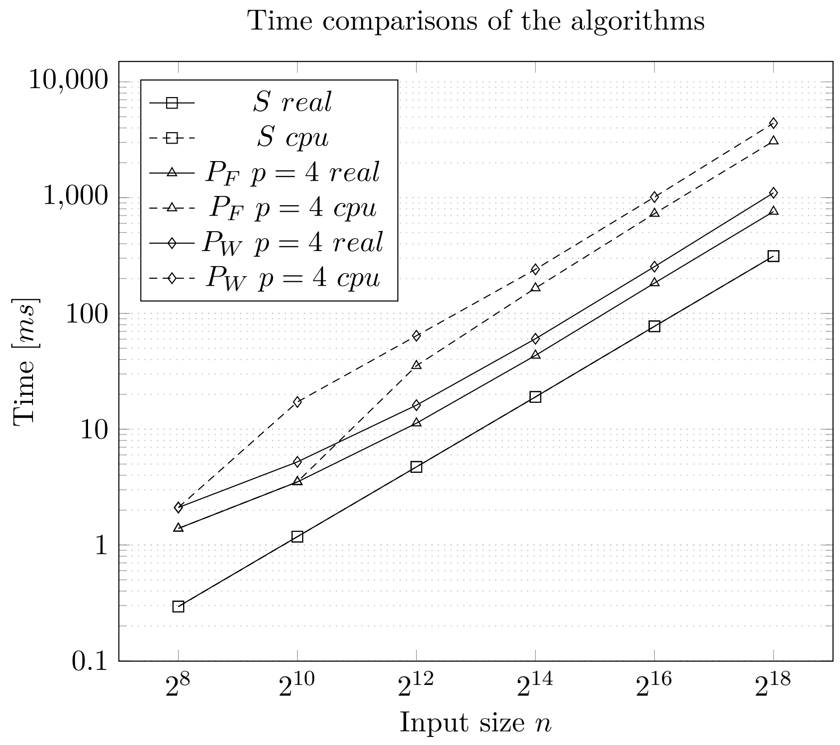


Figure 5.3: Time comparisons based on algorithm

Conclusions and Future work

This thesis was implementation of the parallel run of k-local DFTA.

Work-optimal parallel run of k-local DFTA was implemented successfully alongside a sequential run of k-local DFTA and all needed support algorithms.

Most of the algorithms were implemented work-optimally. Solutions for some problems were implemented using different algorithms (e.g. list ranking which was implemented sequentially, by pointer jumping and using Cole's algorithm).

A parallel run of k-local DFTA was implemented with the ability to select which list ranking algorithm is used. This was because list ranking by pointer jumping is potentially faster if enough processors are available but is not work-optimal. Work-optimal list ranking is also faster when there are fewer processors available.

Execution times of all three implementations of a run of k-local DFTA (i.e. sequential, parallel with pointer jumping and parallel with work-optimal list ranking) were experimentally measured and compared.

Testing was performed with up to 6 processor cores. Results of the measurement seem to be promising that the expected parallel time may be possibly achieved with this implementation, but further testing with more processor cores and possibly bigger input trees is needed to be performed to make any conclusions since provided number of processor cores during tests seems insufficient for this algorithm to obtain some speedup.

Future work

Tests with bigger data and a higher number of processors could be performed in the future on hardware with more cores.

This algorithm was described for the EREW PRAM computation model but possibly may be modified for other computation models in the future. There's potential for migrating to distributed memory systems due to the need for high parallelism. This algorithm could be modified for usage on GPGPU or other SIMD architectures in the future.

All those future implementations possibly may serve as proof of concept for this algorithm outside the PRAM computation model.

Bibliography

[1] Plachý, Š.; Janoušek, J. On Synchronizing Tree Automata and Their Work–Optimal Parallel Run, Usable for Parallel Tree Pattern Matching. In *SOFSEM 2020: Theory and Practice of Computer Science*, edited by A. Chatzigeorgiou; R. Dondi; H. Herodotou; C. Kapoutsis; Y. Manolopoulos; G. A. Papadopoulos; F. Sikora, Cham: Springer International Publishing, 2020, ISBN 978-3-030-38919-2, pp. 576–586.

[2] Rahman, M. S. *Basic graph theory*. Cham, Switzerland: Springer, 2017, ISBN 978-3-319-49475-3.

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

[4] Hillis, W. D.; Steele, G. L. Data parallel algorithms. *Communications of the ACM*, volume 29, no. 12, Dec. 1986: pp. 1170–1183, doi:10.1145/7902.7903. Available from: `https://doi.org/10.1145/7902.7903`

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

[6] Cole, R.; Vishkin, U. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, volume 81, no. 3, 1989: pp. 334–352, ISSN 0890-5401, doi:https://doi.org/10.1016/0890-5401(89)90036-9. Available from: `https://www.sciencedirect.com/science/article/pii/0890540189900369`

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

[8] Levcopoulos, C.; Petersson, O. Matching parentheses in parallel. *Discrete Applied Mathematics*, volume 40, no. 3, 1992: pp. 423–431, ISSN 0166-218X, doi:https://doi.org/10.1016/0166-218X(92)90011-X. Available from: `https://www.sciencedirect.com/science/article/pii/0166218X9290011X`

[9] 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/`

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

Appendix A

Acronyms

BFS	Breadth first search
CRCW	Concurrent Read Concurrent Write
CREW	Concurrent Read Exclusive Write
DCT	Deterministic Coin Tossing
DFS	Depth first search
DFTA	Deterministic finite tree automaton
EREW	Exclusive Read Exclusive Write
ETT	Euler's Tour Technique
PRAM	Parallel Random Access Machine
pthread	POSIX threads
RAM	Random Access Machine
TBB	Thread Building Blocks

Appendix B

Symbols

$\mathbb{N}_0$	set of natural numbers
$\mathbb{R}^+$	set of positive real numbers
$\widehat{x}$	set $\{1, 2, \ldots, x\}$

Appendix C

User manual

C.1 Prerequisities

- CMake v3.13 or newer
- OpenMP v5.1 or newer
- Algorithms Library Toolkit v0.0.0.r1109 or newer

C.2 Compilation

To compile this thesis follow these steps:

1. Navigate to the desired directory (hereinafter referred to as <BUILD_DIR>) where the compiled files should be created.
2. Type *cmake <SRC_DIR>* in the terminal, replace *<SRC_DIR>* with the path to the root sources directory.
3. Type *cmake –build <BUILD_DIR>* in the terminal.

C.3 Usage

The compilation of the source codes produces multiple files:

- *<BUILD_DIR>/src/libparallel_run_dfta_lib.a* library containing the parallel run of k-local DFTA and all the other support functions and structures.
- *<BUILD_DIR>/test/generators/libparallel_run_dfta_generators.a* library containing generators of test data.
- *<BUILD_DIR>/test/functionality/parallel_run_dfta_test* executable that runs unit tests of the implemented functions.
- *<BUILD_DIR>/test/measurements/parallel_run_dfta_measure* executable that runs time measurement tests of the run of k-local DFTA.
- *<BUILD_DIR>/parallel_run_ dfta_demo* executable that runs simple demonstration program that runs k-local DFTA in parallel for pre-defined input tree G from figure 4.8 and 3-local DFTA from figure 4.6.

To run the demo program type *./<BUILD_DIR>/parallel_run_dfta_demo* in the terminal.

Appendix D

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
data........................ the directory of the pre-generated test data
results .................. the directory of the time measurement results
  results.csv ........... the time measurement results in CSV format
src...................................... the directory of source codes
  dfta .................................. the implementation sources
  thesis .............. the directory of LaTeX source codes of the thesis
text ........................................ the thesis text directory
  thesis.pdf ......................... the thesis text in PDF format
```