WS 2014/15

Parallel Algorithms

Harald Räcke

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2014WS/pa/

Winter Term 2014/15

Part I

Organizational Matters

Part I

Organizational Matters

- Modul: IN2011
- Name: "Parallel Algorithms"
  "Parallele Algorithmen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 14:00–16:00 (Room 00.08.038)
    Fri 8:30–10:00 (Room 00.08.038)
- Webpage: `http://www14.in.tum.de/lehre/2014WS/pa/`

Part I

Organizational Matters

- Modul: IN2011
- Name: "Parallel Algorithms"
  "Parallele Algorithmen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 14:00–16:00 (Room 00.08.038)
    Fri 8:30–10:00 (Room 00.08.038)
- Webpage: `http://www14.in.tum.de/lehre/2014WS/pa/`

Part I

Organizational Matters

- Modul: IN2011
- Name: “Parallel Algorithms”
  “Parallele Algorithmen”
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 14:00–16:00 (Room 00.08.038)
    Fri 8:30–10:00 (Room 00.08.038)
- Webpage: `http://www14.in.tum.de/lehre/2014WS/pa/`

Part I

Organizational Matters

- Modul: IN2011
- Name: “Parallel Algorithms”
  “Parallele Algorithmen”
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 14:00–16:00 (Room 00.08.038)
    Fri 8:30–10:00 (Room 00.08.038)
- Webpage: http://www14.in.tum.de/lehre/2014WS/pa/

Part I

Organizational Matters

- Modul: IN2011
- Name: "Parallel Algorithms"
  "Parallele Algorithmen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 14:00–16:00 (Room 00.08.038)
    Fri 8:30–10:00 (Room 00.08.038)
- Webpage: `http://www14.in.tum.de/lehre/2014WS/pa/`

- Required knowledge:
  - IN0001, IN0003
    "Introduction to Informatics 1/2"
    "Einführung in die Informatik 1/2"
  - IN0007
    "Fundamentals of Algorithms and Data Structures"
    "Grundlagen: Algorithmen und Datenstrukturen" (GAD)
  - IN0011
    "Basic Theoretic Informatics"
    "Einführung in die Theoretische Informatik" (THEO)
  - IN0015
    "Discrete Structures"
    "Diskrete Strukturen" (DS)
  - IN0018
    "Discrete Probability Theory"
    "Diskrete Wahrscheinlichkeitstheorie" (DWT)
  - IN2003
    "Efficient Algorithms and Data Structures"
    "Effiziente Algorithmen und Datenstrukturen"

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “Einführung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “Einführung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)
  - IN2003
    “Efficient Algorithms and Data Structures”
    “Effiziente Algorithmen und Datenstrukturen”

The Lecturer

- Harald Räcke
- Email: raecke@in.tum.de
- Room: 03.09.044
- Office hours: (per appointment)

Tutorials

- Tutors:
  - Chris Pinkau
  - pinkau@in.tum.de
  - Room: 03.09.037
  - Office hours: Tue 13:00–14:00
- Room: 03.11.018
- Time: Tue 14:00–16:00

Assignment sheets

- In order to pass the module you need to pass a 3 hour exam

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Monday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Monday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.019A.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Monday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Monday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.019A.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Monday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Monday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.019A.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Monday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Monday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.019A.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Monday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Monday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.019A.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.

1 Contents

1 Contents

- PRAM algorithms
  - Parallel Models
  - PRAM Model
  - Basic PRAM Algorithms
  - Sorting
  - Lower Bounds
- Networks of Workstations
  - Offline Permutation Routing on the Mesh
  - Oblivious Routing in the Butterfly
  - Greedy Routing
  - Sorting on the Mesh
  - ASCEND/DESCEND Programs
  - Embeddings between Networks

2 Literatur

- Tom Leighton:
  Introduction to Parallel Algorithms and Architecture: Arrays, Trees, Hypercubes,
  Morgan Kaufmann: San Mateo, CA, 1992
- Joseph JaJa:
  An Introduction to Parallel Algorithms,
  Addison-Wesley: Reading, MA, 1997
- Jeffrey D. Ullman:
  Computational Aspects of VLSI,
  Computer Science Press: Rockville, USA, 1984
- Selim G. Akl.:
  The Design and Analysis of Parallel Algorithms,
  Prentice Hall: Englewood Cliffs, NJ, 1989

Part II

Foundations

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

3 Introduction

Parallel Computing

A parallel computer is a collection of processors usually of the same type, interconnected to allow coordination and exchange of data.

The processors are primarily used to jointly solve a given problem.

Distributed Systems

A set of possibly many different types of processors are distributed over a larger geographic area.

Processors do not work on a single problem.

Some processors may act in a malicous way.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate sequential algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- ...

Asymptotic bounds (e.g., for running time) often give a good indication on the algorithms performance on a wide variety of machines.

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted classes of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n and on number of processors p
- statements usually only hold for restricted classes of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n **and** on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Cost measures

How do we evaluate parallel algorithms?

- time efficiency
- space utilization
- energy consumption
- programmability
- communication requirement
- ...

Problems

- performance (e.g. runtime) depends on problem size n **and** on number of processors p
- statements usually only hold for restricted types of parallel machine as parallel computers may have vastly different characteristics (in particular w.r.t. communication)

Speedup

Suppose a problem P has sequential complexity $T^*(n)$, i.e., there is no algorithm that solves P in time $o(T^*(n))$.

Definition 1

The speedup $S_p(n)$ of a parallel algorithm A that requires time $T_p(n)$ for solving P with p processors is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \ .$$

Clearly, $S_p(n) \leq p$. **Goal:** obtain $S_p(n) \approx p$.

It is common to replace $T^*(n)$ by the time bound of the best **known** sequential algorithm for P!

Speedup

Suppose a problem P has sequential complexity $T^*(n)$, i.e., there is no algorithm that solves P in time $o(T^*(n))$.

Definition 1

The speedup $S_p(n)$ of a parallel algorithm A that requires time $T_p(n)$ for solving P with p processors is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \ .$$

Clearly, $S_p(n) \leq p$. **Goal:** obtain $S_p(n) \approx p$.

It is common to replace $T^*(n)$ by the time bound of the best **known** sequential algorithm for P!

Speedup

Suppose a problem P has sequential complexity $T^*(n)$, i.e., there is no algorithm that solves P in time $o(T^*(n))$.

Definition 1

The speedup $S_p(n)$ of a parallel algorithm A that requires time $T_p(n)$ for solving P with p processors is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \ .$$

Clearly, $S_p(n) \le p$. **Goal:** obtain $S_p(n) \approx p$.

It is common to replace $T^*(n)$ by the time bound of the best **known** sequential algorithm for P!

Speedup

Suppose a problem P has sequential complexity $T^*(n)$, i.e., there is no algorithm that solves P in time $o(T^*(n))$.

Definition 1

The speedup $S_p(n)$ of a parallel algorithm A that requires time $T_p(n)$ for solving P with p processors is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \ .$$

Clearly, $S_p(n) \leq p$. **Goal:** obtain $S_p(n) \approx p$.

It is common to replace $T^*(n)$ by the time bound of the best **known** sequential algorithm for P!

Efficiency

Definition 2

The efficiency of a parallel algorithm A that requires time $T_p(n)$ when using p processors on a problem of size n is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \quad .$$

$E_p(n) \approx 1$ indicates that the algorithm is running roughly p times faster with p processors than with one processor.

Note that $E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$. Hence, the efficiency goes down rapidly if $p \geq T_1(n)/T_\infty(n)$.

Disadvantage: cost-measure does not relate to the optimum sequential algorithm.

Efficiency

Definition 2

The efficiency of a parallel algorithm A that requires time $T_p(n)$ when using p processors on a problem of size n is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \quad .$$

$E_p(n) \approx 1$ indicates that the algorithm is running roughly p times faster with p processors than with one processor.

Note that $E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$. Hence, the efficiency goes down rapidly if $p \geq T_1(n)/T_\infty(n)$.

Disadvantage: cost-measure does not relate to the optimum sequential algorithm.

Efficiency

Definition 2

The efficiency of a parallel algorithm A that requires time $T_p(n)$ when using p processors on a problem of size n is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \quad .$$

$E_p(n) \approx 1$ indicates that the algorithm is running roughly p times faster with p processors than with one processor.

Note that $E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$. Hence, the efficiency goes down rapidly if $p \geq T_1(n)/T_\infty(n)$.

Disadvantage: cost-measure does not relate to the optimum sequential algorithm.

Efficiency

Definition 2

The efficiency of a parallel algorithm A that requires time $T_p(n)$ when using p processors on a problem of size n is

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \enspace .$$

$E_p(n) \approx 1$ indicates that the algorithm is running roughly p times faster with p processors than with one processor.

Note that $E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$. Hence, the efficiency goes down rapidly if $p \geq T_1(n)/T_\infty(n)$.

Disadvantage: cost-measure does not relate to the optimum sequential algorithm.

Parallel Models — Requirements

Simplicity
A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability
Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

Parallel Models — Requirements

Simplicity

A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability

Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

Parallel Models — Requirements

Simplicity

A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability

Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

Parallel Models — Requirements

Simplicity

A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability

Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

Parallel Models — Requirements

Simplicity

A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability

Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

Parallel Models — Requirements

Simplicity
A model should allow to easily analyze various performance measures (speed, communication, memory utilization etc.).

Results should be as hardware-independent as possible.

Implementability
Parallel algorithms developed in a model should be easily implementable on a parallel machine.

Theoretical analysis should carry over and give meaningful performance estimates.

A real satisfactory model does not exist!

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

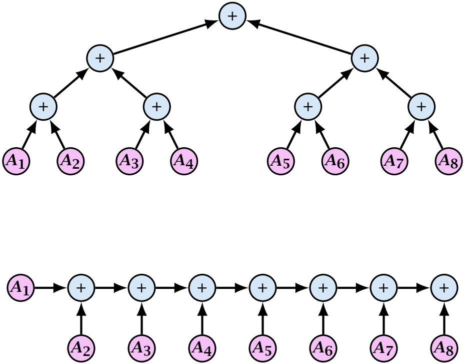
Often used for automatically parallelizing numerical computations.

DAG model — computation graph

- nodes represent operations (single instructions or larger blocks)
- edges represent dependencies (precedence constraints)
- closely related to circuits; however there exist many different variants
- branching instructions cannot be modelled
- completely hardware independent
- scheduling is not defined

Often used for automatically parallelizing numerical computations.

Example: Addition



Here, vertices without incoming edges correspond to input data.

The graph can be viewed as a data flow graph.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3

A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \ldots, p\}$; $t_v \in \{1, \ldots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.
T is the length of the schedule.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3

A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \dots, p\}$; $t_v \in \{1, \dots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.

T is the length of the schedule.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3

A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \ldots, p\}$; $t_v \in \{1, \ldots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.
T is the length of the schedule.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3

A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \ldots, p\}$; $t_v \in \{1, \ldots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.

T is the length of the schedule.

DAG model — computation graph

The DAG itself is not a complete algorithm. A scheduling implements the algorithm on a parallel machine, by assigning a time-step t_v and a processor p_v to every node.

Definition 3

A scheduling of a DAG $G = (V, E)$ on p processors is an assignment of pairs (t_v, p_v) to every internal node $v \in V$, s.t.,

- $p_v \in \{1, \ldots, p\}$; $t_v \in \{1, \ldots, T\}$
- $t_u = t_v \Rightarrow p_u \neq p_v$
- $(u, v) \in E \Rightarrow t_v \geq t_u + 1$

where a non-internal node x (an input node) has $t_x = 0$.
T is the length of the schedule.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$
$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$

$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$

$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG

$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$

$$T_p(n) \geq T_1(n)/p$$

Lemma 4

A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5

Finding an optimal schedule is in general NP-complete.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$
$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

DAG model — computation graph

The parallel complexity of a DAG is defined as

$$T_p(n) = \min_{\text{schedule } S} \{T(S)\} \ .$$

$T_1(n)$: #internal nodes in DAG
$T_\infty(n)$: diameter of DAG

Clearly,

$$T_p(n) \geq T_\infty(n)$$
$$T_p(n) \geq T_1(n)/p$$

Lemma 4
A schedule with length $\mathcal{O}(T_1(n)/p + T_\infty(n))$ can be found easily.

Lemma 5
Finding an optimal schedule is in general NP-complete.

Note that the DAG model as defined is a non-uniform model of computation.

In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

Note that the DAG model as defined is a non-uniform model of computation.

In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

Note that the DAG model as defined is a non-uniform model of computation.

In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

Note that the DAG model as defined is a non-uniform model of computation.

In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

Note that the DAG model as defined is a non-uniform model of computation.

In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

Note that the DAG model as defined is a non-uniform model of computation.

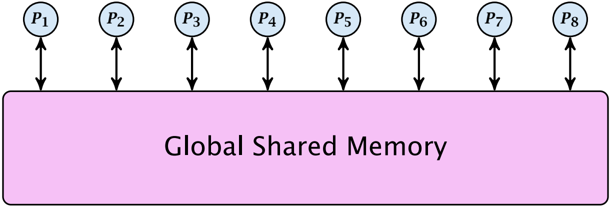
In principle, there could be a different DAG for every input size n.

An algorithm (e.g. for a RAM) must work for every input size and must be of finite description length.

Hence, specifying a different DAG for every n has more expressive power.

Also, this is not really a complete model, as the operations allowed in a DAG node are not clearly defined.

PRAM Model

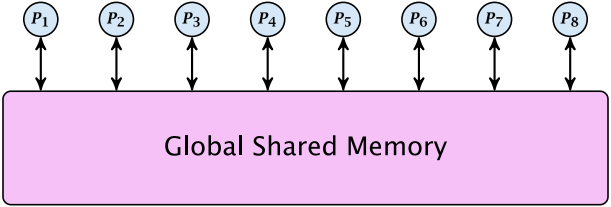


All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model

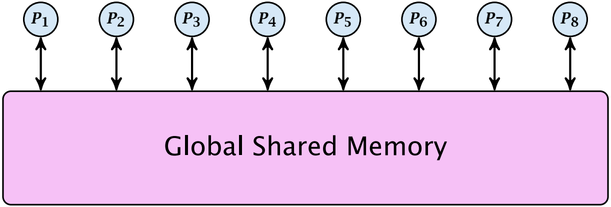


All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model

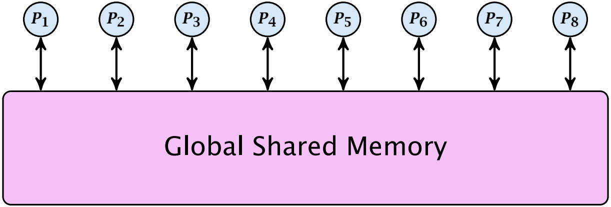


All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model

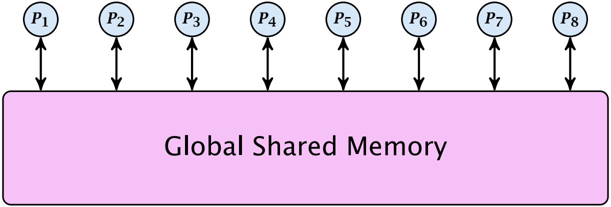


All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model



All processors are synchronized.

In every round a processor can:

- read a register from global memory into local memory
- do a local computation à la RAM
- write a local register into global memory

PRAM Model

Every processor executes the same program.

However, the program has access to two special variables:

- p: total number of processors
- $id \in \{1, \dots, p\}$: the id of the current processor

The following (stupid) program copies the content of the global register $x[1]$ to registers $x[2] \dots x[p]$.

Algorithm 1 copy

```
if id = 1 then round ← 1
while round ≤ p and id = round do
    x[id + 1] ← x[id]
    round ← round + 1
```

PRAM Model

Every processor executes the same program.

However, the program has access to two special variables:

- p: total number of processors
- $id \in \{1, \dots, p\}$: the id of the current processor

The following (stupid) program copies the content of the global register $x[1]$ to registers $x[2] \dots x[p]$.

Algorithm 1 copy

if $id = 1$ **then** $round \leftarrow 1$
while $round \leq p$ **and** $id = round$ **do**
    $x[id + 1] \leftarrow x[id]$
    $round \leftarrow round + 1$

PRAM Model

Every processor executes the same program.

However, the program has access to two special variables:

- p: total number of processors
- $id \in \{1, \dots, p\}$: the id of the current processor

The following (stupid) program copies the content of the global register $x[1]$ to registers $x[2] \dots x[p]$.

Algorithm 1 copy

if $id = 1$ **then** $round \leftarrow 1$
while $round \leq p$ **and** $id = round$ **do**
    $x[id + 1] \leftarrow x[id]$
    $round \leftarrow round + 1$

PRAM Model

Every processor executes the same program.

However, the program has access to two special variables:

- p: total number of processors
- $id \in \{1, \ldots, p\}$: the id of the current processor

The following (stupid) program copies the content of the global register $x[1]$ to registers $x[2] \ldots x[p]$.

Algorithm 1 copy

if $id = 1$ **then** $round \leftarrow 1$
while $round \le p$ **and** $id = round$ **do**
    $x[id + 1] \leftarrow x[id]$
    $round \leftarrow round + 1$

PRAM Model

- processors can effectively execute different code because of branching according to *id*
- however, not arbitrarily; still uniform model of computation

Often it is easier to explicitly define which parts of a program are executed in parallel:

Algorithm 2 sum

```
// computes sum of x[1] ... x[p]
// red part is executed only by processor 1
r ← 1
while 2^r ≤ p do
    for id mod 2^r = 1 pardo
    // only executed by processors whose id matches
        x[id] = x[id] + x[id + 2^(r−1)]
    r ← r + 1
return x[1]
```

PRAM Model

- processors can effectively execute different code because of branching according to *id*
- however, not arbitrarily; still uniform model of computation

Often it is easier to explicitly define which parts of a program are executed in parallel:

Algorithm 2 sum

```
// computes sum of x[1] ... x[p]
// red part is executed only by processor 1
r ← 1
while 2^r ≤ p do
        for id mod 2^r = 1 pardo
        // only executed by processors whose id matches
                x[id] = x[id] + x[id + 2^(r-1)]
        r ← r + 1
return x[1]
```

PRAM Model

- processors can effectively execute different code because of branching according to id
- however, not arbitrarily; still uniform model of computation

Often it is easier to explicitly define which parts of a program are executed in parallel:

Algorithm 2 sum

// computes sum of $x[1] \ldots x[p]$
// red part is executed only by processor 1
$r \leftarrow 1$
while $2^r \le p$ **do**
    for $id \bmod 2^r = 1$ **pardo**
    // only executed by processors whose id matches
        $x[id] = x[id] + x[id + 2^{r-1}]$
    $r \leftarrow r + 1$
return $x[1]$

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed;
  write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Different Types of PRAMs

Simultaneous Access to Shared Memory:

- EREW PRAM:
  simultaneous access is not allowed
- CREW PRAM:
  concurrent read accesses to the same location are allowed; write accesses have to be exclusive
- CRCW PRAM:
  concurrent read and write accesses allowed
  - commom CRCW PRAM
    all processors writing to $x[i]$ must write same value
  - arbitrary CRCW PRAM
    values may be different; an arbitrary processor succeeds
  - priority CRCW PRAM
    values may be different; processor with smallest id succeeds

Algorithm 3 sum

```
1: // computes sum of x[1]...x[p]
2: r ← 1
3: while 2^r ≤ p do
4:     for id mod 2^r = 1 pardo
5:         x[id] = x[id] + x[id + 2^(r−1)]
6:     r ← r + 1
7: return x[1]
```

The above is an EREW PRAM algorithm.

On a CREW PRAM we could replace Line 4 by
for $1 \leq id \leq p$ **pardo**

Algorithm 3 sum

```
1: // computes sum of x[1]...x[p]
2: r ← 1
3: while 2^r ≤ p do
4:     for id mod 2^r = 1 pardo
5:         x[id] = x[id] + x[id + 2^(r-1)]
6:     r ← r + 1
7: return x[1]
```

The above is an EREW PRAM algorithm.

On a CREW PRAM we could replace Line 4 by

for $1 \leq id \leq p$ **pardo**

Algorithm 3 sum

```
1: // computes sum of x[1]...x[p]
2: r ← 1
3: while 2^r ≤ p do
4:     for id mod 2^r = 1 pardo
5:         x[id] = x[id] + x[id + 2^(r−1)]
6:     r ← r + 1
7: return x[1]
```

The above is an EREW PRAM algorithm.

On a CREW PRAM we could replace Line 4 by
for $1 \leq id \leq p$ **pardo**

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

PRAM Model — remarks

- similar to a RAM we either need to restrict the size of values that can be stored in registers, or we need to have a non-uniform cost model for doing a register manipulation (cost for manipulating $x[i]$ is proportional to the bit-length of the largest number that is ever being stored in $x[i]$)
  - in this lecture: uniform cost model but we are not exploiting the model
- global shared memory is very unrealistic in practise as uniform access to all memory locations does not exist
- global synchronziation is very unrealistic; in real parallel machines a global synchronization is very costly
- model is good for understanding basic parallel mechanisms/techniques but **not** for algorithm development
- model is good for lower bounds

Network of Workstations — NOWs

- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

Network of Workstations — NOWs

- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

Network of Workstations — NOWs

- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

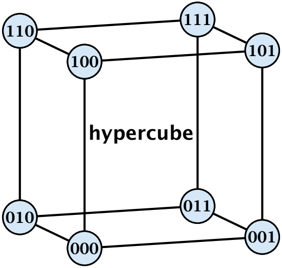
Network of Workstations — NOWs

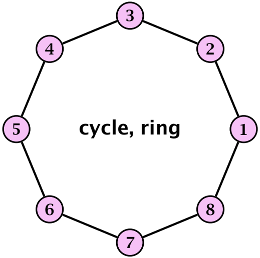
- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

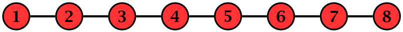
Network of Workstations — NOWs

- interconnection network represented by a graph $G = (V, E)$
- each $v \in V$ represents a processor
- an edge $\{u, v\} \in E$ represents a two-way communication link between processors u and v
- network is asynchronous
- all coordination/communiation has to be done by explicit message passing

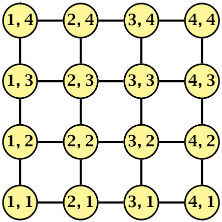
Typical Topologies







linear array



mesh/grid

Network of Workstations — NOWs

Computing the sum on a d-dimensional hypercube. Note that $x[0] \ldots x[2^d - 1]$ are stored at the individual nodes.

Processors are numbered consecutively starting from 0

Algorithm 4 sum

```
// computes sum of x[0]...x[2^d − 1]
r ← 1
while 2^r ≤ 2^d do // p = 2^d
    if id mod 2^r = 0 then
        temp ← receive(id + 2^(r−1))
        x[id] = x[id] + temp
    if id mod 2^r = 2^(r−1) then
        send(x[id], id − 2^(r−1))
    r ← r + 1
if id = 0 then return x[id]
```

Network of Workstations — NOWs

Computing the sum on a d-dimensional hypercube. Note that $x[0] \dots x[2^d - 1]$ are stored at the individual nodes.

Processors are numbered consecutively starting from 0

Algorithm 4 sum

// computes sum of $x[0] \dots x[2^d - 1]$
$r \leftarrow 1$
while $2^r \le 2^d$ **do** // $p = 2^d$
    if $id \bmod 2^r = 0$ **then**
        $temp \leftarrow$ receive$(id + 2^{r-1})$
        $x[id] = x[id] + temp$
    if $id \bmod 2^r = 2^{r-1}$ **then**
        send$(x[id], id - 2^{r-1})$
    $r \leftarrow r + 1$
if $id = 0$ **then return** $x[id]$

Network of Workstations — NOWs

Computing the sum on a d-dimensional hypercube. Note that $x[0]\dots x[2^d-1]$ are stored at the individual nodes.

Processors are numbered consecutively starting from 0

Algorithm 4 sum

// computes sum of $x[0]\dots x[2^d-1]$
$r \leftarrow 1$
while $2^r \le 2^d$ **do** // $p = 2^d$
    if $id \bmod 2^r = 0$ **then**
        $temp \leftarrow$ receive$(id + 2^{r-1})$
        $x[id] = x[id] + temp$
    if $id \bmod 2^r = 2^{r-1}$ **then**
        send$(x[id], id - 2^{r-1})$
    $r \leftarrow r + 1$
if $id = 0$ **then return** $x[id]$

Network of Workstations — NOWs

Computing the sum on a d-dimensional hypercube. Note that $x[0] \dots x[2^d - 1]$ are stored at the individual nodes.

Processors are numbered consecutively starting from 0

Algorithm 4 sum

```
// computes sum of x[0]...x[2^d − 1]
r ← 1
while 2^r ≤ 2^d do // p = 2^d
    if id mod 2^r = 0 then
        temp ← receive(id + 2^(r−1))
        x[id] = x[id] + temp
    if id mod 2^r = 2^(r−1) then
        send(x[id], id − 2^(r−1))
    r ← r + 1
if id = 0 then return x[id]
```

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Network of Workstations — NOWs

Remarks

- One has to ensure that at any point in time there is at most one active communication along a link
- There also exist synchronized versions of the model, where in every round each link can be used once for communication
- In particular the asynchronous model is quite realistic
- Difficult to develop and analyze algorithms as a lot of low level communication has to be dealt with
- Results only hold for one specific topology and cannot be generalized easily

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \le P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \le P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \leq P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \leq P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \le P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \leq P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose that we can solve an instance of a problem with size n with $P(n)$ processors and time $T(n)$.

We call $C(n) = T(n) \cdot P(n)$ the time-processor product or the cost of the algorithm.

The following statements are equivalent

- $P(n)$ processors and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n))$ cost and time $\mathcal{O}(T(n))$
- $\mathcal{O}(C(n)/p)$ time for any number $p \leq P(n)$ processors
- $\mathcal{O}(C(n)/p + T(n))$ for any number p of processors

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \le i \le T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \le \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \le \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \le i \le T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \le \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \le \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \le i \le T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \le \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \le \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \le i \le T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \le \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \le \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \le i \le T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \le \sum_i \left(\lfloor W_i(n)/p \rfloor + 1\right) \le \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Suppose we have a PRAM algorithm that takes time $T(n)$ and work $W(n)$, where work is the total number of operations.

We can nearly always obtain a PRAM algorithm that uses time at most

$$\lfloor W(n)/p \rfloor + T(n)$$

parallel steps on p processors.

Idea:

- $W_i(n)$ denotes operations in parallel step i, $1 \leq i \leq T(n)$
- simulate each step in $\lceil W_i(n)/p \rceil$ parallel steps
- then we have

$$\sum_i \lceil W_i(n)/p \rceil \leq \sum_i \left(\lfloor W_i(n)/p \rfloor + 1 \right) \leq \lfloor W(n)/p \rfloor + T(n)$$

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Performance of PRAM algorithms

Why nearly always?

We need to assign processors to operations.

- every processor p_i needs to know whether it should be active
- in case it is active it needs to know which operations to perform

design algorithms for an arbitrary number of processors; keep total time and work low

Optimal PRAM algorithms

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$S_p(n)$

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$$S_p(n) = \Omega\left(\frac{T^*(n)}{T^*(n)/p + T(n)}\right)$$

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$$S_p(n) = \Omega\left(\frac{T^*(n)}{T^*(n)/p + T(n)}\right) = \Omega\left(\frac{pT^*(n)}{T^*(n) + pT(n)}\right)$$

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$$S_p(n) = \Omega\left(\frac{T^*(n)}{T^*(n)/p + T(n)}\right) = \Omega\left(\frac{pT^*(n)}{T^*(n) + pT(n)}\right) = \Omega(p)$$

Optimal PRAM algorithms

Suppose the optimal sequential running time for a problem is $T^*(n)$.

We call a PRAM algorithm for the same problem work optimal if its work $W(n)$ fulfills

$$W(n) = \Theta(T^*(n))$$

If such an algorithm has running time $T(n)$ it has speedup

$$S_p(n) = \Omega\left(\frac{T^*(n)}{T^*(n)/p + T(n)}\right) = \Omega\left(\frac{pT^*(n)}{T^*(n) + pT(n)}\right) = \Omega(p)$$

for $p = \mathcal{O}(T^*(n)/T(n))$.

This means by improving the time $T(n)$, (while using same work) we improve the range of p, for which we obtain optimal speedup.

We call an algorithm worktime (WT) optimal if $T(n)$ cannot be asymptotically improved by any work optimal algorithm.

This means by improving the time $T(n)$, (while using same work) we improve the range of p, for which we obtain optimal speedup.

We call an algorithm worktime (WT) optimal if $T(n)$ cannot be asymptotically improved by any work optimal algorithm.

Example

Algorithm for computing the sum has work $W(n) = \mathcal{O}(n)$.
optimal

$T(n) = \mathcal{O}(\log n)$. Hence, we achieve an optimal speedup for $p = \mathcal{O}(n/\log n)$.

One can show that any CREW PRAM requires $\Omega(\log n)$ time to compute the sum.

Example

Algorithm for computing the sum has work $W(n) = \mathcal{O}(n)$.
optimal

$T(n) = \mathcal{O}(\log n)$. Hence, we achieve an optimal speedup for $p = \mathcal{O}(n/\log n)$.

One can show that any CREW PRAM requires $\Omega(\log n)$ time to compute the sum.

Example

Algorithm for computing the sum has work $W(n) = \mathcal{O}(n)$.
optimal

$T(n) = \mathcal{O}(\log n)$. Hence, we achieve an optimal speedup for $p = \mathcal{O}(n/\log n)$.

One can show that any CREW PRAM requires $\Omega(\log n)$ time to compute the sum.

Example

Algorithm for computing the sum has work $W(n) = \mathcal{O}(n)$.
optimal

$T(n) = \mathcal{O}(\log n)$. Hence, we achieve an optimal speedup for $p = \mathcal{O}(n/\log n)$.

One can show that any CREW PRAM requires $\Omega(\log n)$ time to compute the sum.

Communication Cost

When we differentiate between local and global memory we can analyze communication cost.

We define the communication cost of a PRAM algorithm as the worst-case traffic between the local memory of a processor and the global shared memory.

Important criterion as communication is usually a major bottleneck.

Communication Cost

When we differentiate between local and global memory we can analyze communication cost.

We define the communication cost of a PRAM algorithm as the worst-case traffic between the local memory of a processor and the global shared memory.

Important criterion as communication is usually a major bottleneck.

Communication Cost

When we differentiate between local and global memory we can analyze communication cost.

We define the communication cost of a PRAM algorithm as the worst-case traffic between the local memory of a processor and the global shared memory.

Important criterion as communication is usually a major bottleneck.

Communication Cost

When we differentiate between local and global memory we can analyze communication cost.

We define the communication cost of a PRAM algorithm as the worst-case traffic between the local memory of a processor and the global shared memory.

Important criterion as communication is usually a major bottleneck.

Communication Cost

Algorithm 5 MatrixMult(A, B, n)

Input: $n \times n$ matrix A and B; $n = 2^k$
Output: $C = AB$
for $1 \le i, j, \ell \le n$ **pardo**
    $X[i, j, \ell] \leftarrow A[i, \ell] \cdot B[\ell, j]$
for $r \leftarrow 1$ **to** $\log n$
    for $1 \le i, j \le n$; $\ell \bmod 2^r = 1$ **pardo**
        $X[i, j, \ell] \leftarrow X[i, j, \ell] + X[i, j, \ell + 2^{r-1}]$
for $1 \le i, j \le n$ **pardo**
    $C[i, j] \leftarrow X[i, j, 1]$

On n^3 processors this algorithm runs in time $\mathcal{O}(\log n)$.
It uses n^3 multiplications and $\mathcal{O}(n^3)$ additions.

Communication Cost

Algorithm 5 MatrixMult(A, B, n)

Input: $n \times n$ matrix A and B; $n = 2^k$
Output: $C = AB$
for $1 \le i, j, \ell \le n$ **pardo**
    $X[i, j, \ell] \leftarrow A[i, \ell] \cdot B[\ell, j]$
for $r \leftarrow 1$ **to** $\log n$
    for $1 \le i, j \le n$; $\ell \bmod 2^r = 1$ **pardo**
        $X[i, j, \ell] \leftarrow X[i, j, \ell] + X[i, j, \ell + 2^{r-1}]$
for $1 \le i, j \le n$ **pardo**
    $C[i, j] \leftarrow X[i, j, 1]$

On n^3 processors this algorithm runs in time $\mathcal{O}(\log n)$.
It uses n^3 multiplications and $\mathcal{O}(n^3)$ additions.

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j,\ell \le n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j, \ell \le n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \leq j,\ell \leq n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \leq j \leq n; 1 \leq \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j,\ell \le n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \leq j, \ell \leq n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \leq j \leq n; 1 \leq \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j,\ell \le n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \leq j,\ell \leq n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \leq j \leq n; 1 \leq \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j,\ell \le n$
n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$
$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.
n time; n communication

What happens if we have n processors?

Phase 1

p_i computes $X[i,j,\ell] = A[i,\ell] \cdot B[\ell,j]$ for all $1 \le j,\ell \le n$

n^2 time; n^2 communication for every processor

Phase 2 (round r)

p_i updates $X[i,j,\ell]$ for all $1 \le j \le n; 1 \le \ell \bmod 2^r = 1$

$\mathcal{O}(n \cdot n/2^r)$ time; no communication

Phase 3

p_i writes i-th row into $C[i,j]$'s.

n time; n communication

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underbrace{\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{bmatrix}}_{C}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$
work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$
time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underbrace{\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{pmatrix}}_{C}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$
work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$
time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underbrace{\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{bmatrix}}_{C}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$

work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$

time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underset{A}{\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}} \cdot \underset{B}{\begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix}} = \underset{C}{\begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{pmatrix}}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$

work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$

time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underbrace{\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}}_{A} \cdot \underbrace{\begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix}}_{B} = \underbrace{\begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{pmatrix}}_{C}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$
work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$
time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

Split matrix into blocks of size $n^{2/3} \times n^{2/3}$.

$$\underbrace{\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{pmatrix}}_{\mathsf{A}} \cdot \underbrace{\begin{pmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{pmatrix}}_{\mathsf{B}} = \underbrace{\begin{pmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{pmatrix}}_{\mathsf{C}}$$

Note that $C_{i,j} = \sum_\ell A_{i,\ell} B_{\ell,j}$.

Now we have the same problem as before but $n' = n^{1/3}$ and a single multiplication costs time $\mathcal{O}((n^{2/3})^3) = \mathcal{O}(n^2)$. An addition costs $n^{4/3}$.

work for multiplications: $\mathcal{O}(n^2 \cdot (n')^3) = \mathcal{O}(n^3)$
work for additions: $\mathcal{O}(n^{4/3} \cdot (n')^3) = \mathcal{O}(n^3)$
time: $\mathcal{O}(n^2) + \log n' \cdot \mathcal{O}(n^{4/3}) = \mathcal{O}(n^2)$

Alternative Algorithm

The communication cost is only $\mathcal{O}(n^{4/3}\log n')$ as a processor in the original problem touches at most $\log n$ entries of the matrix.

Each entry has size $\mathcal{O}(n^{4/3})$.

The algorithm exhibits less parallelism but still has optimum work/runtime for just n processors.

much, much better in practise

Alternative Algorithm

The communication cost is only $\mathcal{O}(n^{4/3}\log n')$ as a processor in the original problem touches at most $\log n$ entries of the matrix.

Each entry has size $\mathcal{O}(n^{4/3})$.

The algorithm exhibits less parallelism but still has optimum work/runtime for just n processors.

much, much better in practise

Alternative Algorithm

The communication cost is only $\mathcal{O}(n^{4/3}\log n')$ as a processor in the original problem touches at most $\log n$ entries of the matrix.

Each entry has size $\mathcal{O}(n^{4/3})$.

The algorithm exhibits less parallelism but still has optimum work/runtime for just n processors.

much, much better in practise

Alternative Algorithm

The communication cost is only $\mathcal{O}(n^{4/3}\log n')$ as a processor in the original problem touches at most $\log n$ entries of the matrix.

Each entry has size $\mathcal{O}(n^{4/3})$.

The algorithm exhibits less parallelism but still has optimum work/runtime for just n processors.

much, much better in practise

Part III

PRAM Algorithms

Prefix Sum

input: $x[1]\dots x[n]$

output: $s[1]\dots s[n]$ with $s[i] = \sum_{j=1}^{i} x[i]$ (w.r.t. operator $*$)

Algorithm 6 PrefixSum$(n, x[1]\dots x[n])$

```
// compute prefixsums; n = 2^k
if n = 1 then s[1] ← x[1]; return
for 1 ≤ i ≤ n/2 pardo
    a[i] ← x[2i − 1] ∗ x[2i]
z[1],...,z[n/2] ← PrefixSum(n/2, a[1]...a[n/2])
for 1 ≤ i ≤ n pardo
    i even   : s[i] ← z[i/2]
    i = 1    : s[1] = x[1]
    i odd    : s[i] ← z[(i − 1)/2] ∗ x[i]
```

Prefix Sum

input: $x[1] \ldots x[n]$

output: $s[1] \ldots s[n]$ with $s[i] = \sum_{j=1}^{i} x[i]$ (w.r.t. operator $*$)

Algorithm 6 PrefixSum$(n, x[1] \ldots x[n])$

// compute prefixsums; $n = 2^k$
if $n = 1$ **then** $s[1] \leftarrow x[1]$; **return**
for $1 \le i \le n/2$ **pardo**
    $a[i] \leftarrow x[2i-1] * x[2i]$
$z[1], \ldots, z[n/2] \leftarrow$ PrefixSum$(n/2, a[1] \ldots a[n/2])$
for $1 \le i \le n$ **pardo**
    i even : $s[i] \leftarrow z[i/2]$
    $i = 1$: $s[1] = x[1]$
    i odd : $s[i] \leftarrow z[(i-1)/2] * x[i]$

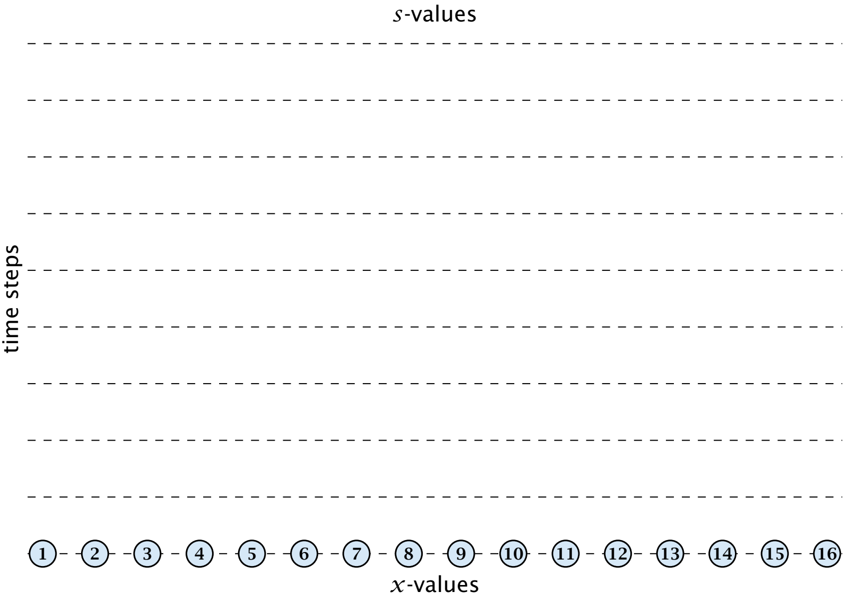
Prefix Sum

s-values

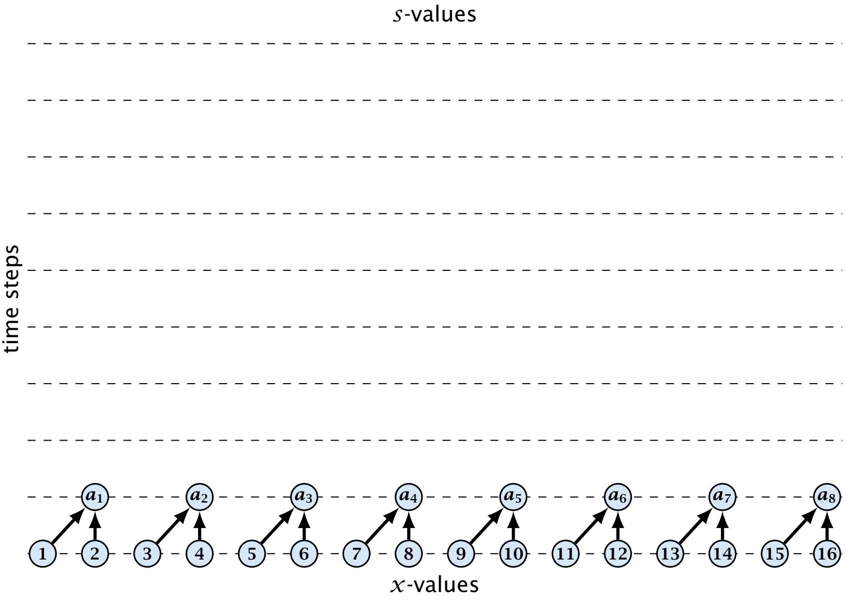
time steps

x-values

Prefix Sum

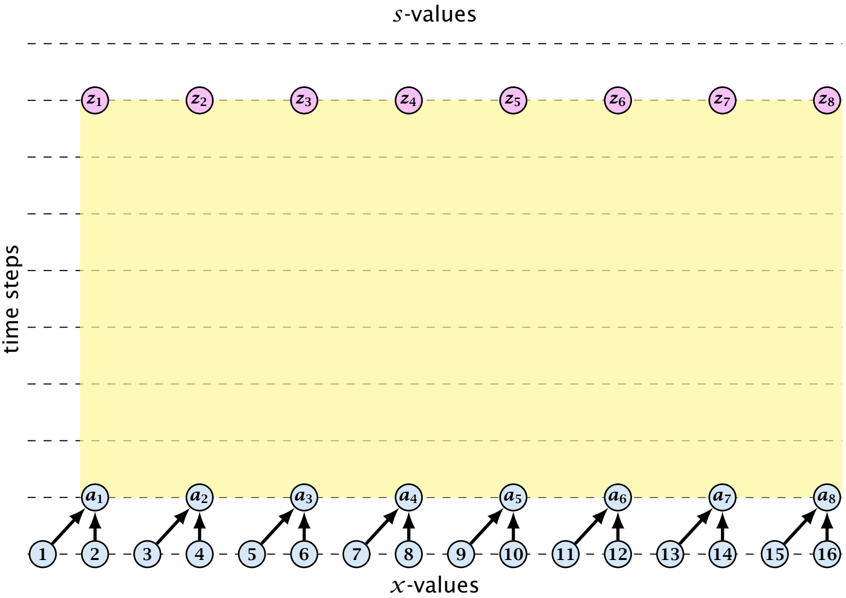


Prefix Sum



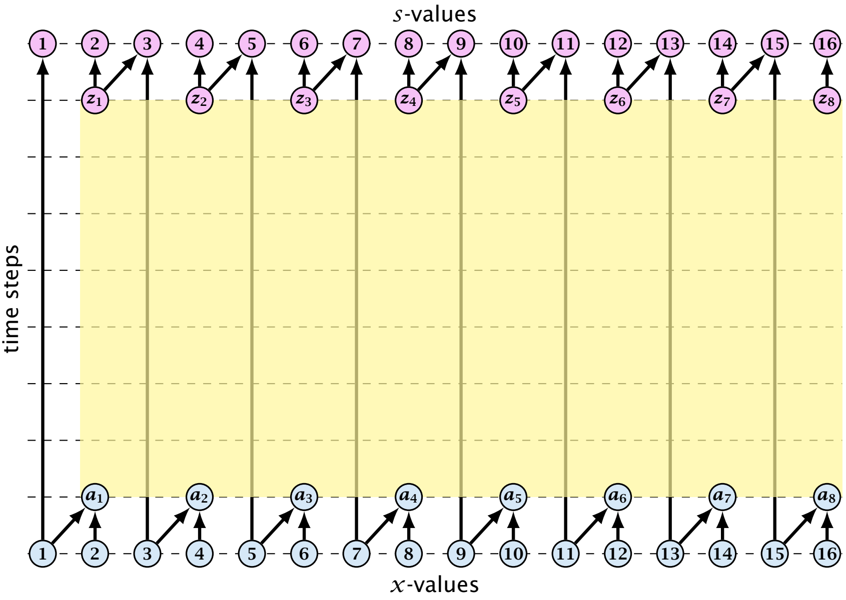
s-values
time steps
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum



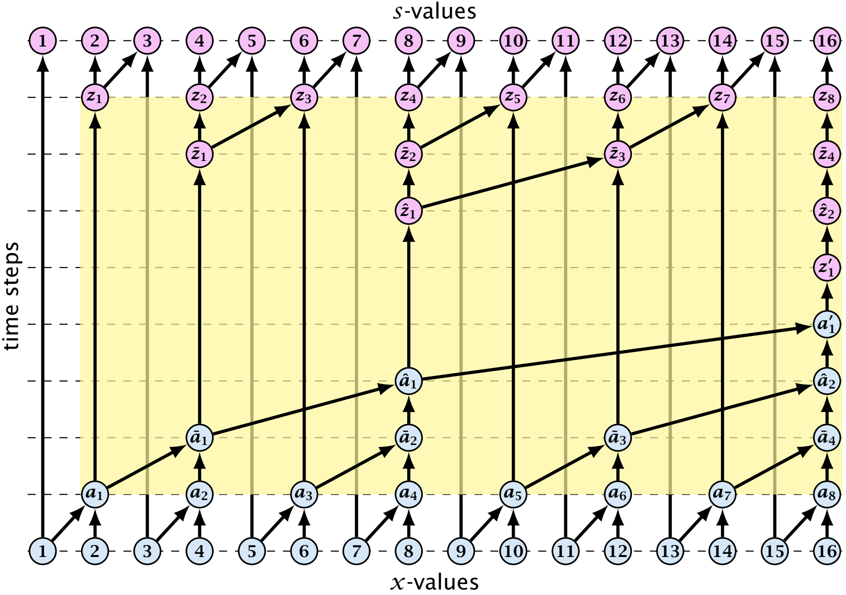
s-values
z_1
z_2
z_3
z_4
z_5
z_6
z_7
z_8
time steps
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
x-values

Prefix Sum



s-values
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
z1
z2
z3
z4
z5
z6
z7
z8
time steps
a1
a2
a3
a4
a5
a6
a7
a8
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum



s-values
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
z_1
z_2
z_3
z_4
z_5
z_6
z_7
z_8
$\bar{z}_1$
$\bar{z}_2$
$\bar{z}_3$
$\bar{z}_4$
$\hat{z}_1$
$\hat{z}_2$
z_1'
a_1'
$\hat{a}_1$
$\hat{a}_2$
$\bar{a}_1$
$\bar{a}_2$
$\bar{a}_3$
$\bar{a}_4$
a_1
a_2
a_3
a_4
a_5
a_6
a_7
a_8
time steps
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
x-values

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

It is clearly work-optimal.

Theorem 6

On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

It is clearly work-optimal.

Theorem 6

On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Prefix Sum

The algorithm uses work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ for solving Prefix Sum on an EREW-PRAM with n processors.

It is clearly work-optimal.

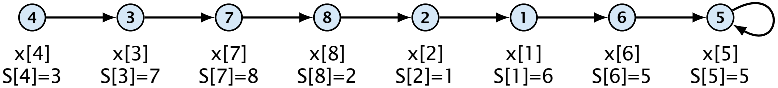
Theorem 6

On a CREW PRAM a Prefix Sum requires running time $\Omega(\log n)$ regardless of the number of processors.

Parallel Prefix

Input: a linked list given by successor pointers; a value $x[i]$ for every list element; an operator $*$;

Output: for every list position ℓ the sum (w.r.t. $*$) of elements after ℓ in the list (including ℓ)



Parallel Prefix

Algorithm 7 ParallelPrefix

for $1 \le i \le n$ **pardo**
    $P[i] \leftarrow S[i]$
    while $S[i] \ne S[S[i]]$ **do**
        $x[i] \leftarrow x[i] * x[S[i]]$
        $S[i] \leftarrow S[S[i]]$
    if $P[i] \ne i$ **then** $x[i] \leftarrow x[i] * x[S(i)]$

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

for $1 \leq i \leq n$ **pardo**
  $P[i] \leftarrow S[i]$
  while $S[i] \neq S[S[i]]$ **do**
    $x[i] \leftarrow x[i] * x[S[i]]$
    $S[i] \leftarrow S[S[i]]$
  if $P[i] \neq i$ **then** $x[i] \leftarrow x[i] * x[S(i)]$

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

```
for 1 ≤ i ≤ n pardo
    P[i] ← S[i]
    while S[i] ≠ S[S[i]] do
        x[i] ← x[i] ∗ x[S[i]]
        S[i] ← S[S[i]]
    if P[i] ≠ i then x[i] ← x[i] ∗ x[S(i)]
```

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

Parallel Prefix

Algorithm 7 ParallelPrefix

```
for 1 ≤ i ≤ n pardo
    P[i] ← S[i]
    while S[i] ≠ S[S[i]] do
        x[i] ← x[i] * x[S[i]]
        S[i] ← S[S[i]]
    if P[i] ≠ i then x[i] ← x[i] * x[S(i)]
```

The algorithm runs in time $\mathcal{O}(\log n)$.

It has work requirement $\mathcal{O}(n \log n)$. non-optimal

This technique is also known as pointer jumping

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 7

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\operatorname{rank}(y : X)$ of y in X is

$$\operatorname{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\operatorname{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \operatorname{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_n)$, compute the sorted squence $C = (c_1, \dots, c_n)$.

Definition 7

Let $X = (x_1, \dots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \dots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \dots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 7

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 7

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \mathrm{rank}(c_i : A \cup B)$.

This means we just need to determine rank($x : A \cup B$) for all elements!

Observe, that $\mathrm{rank}(x : A \cup B) = \mathrm{rank}(x : A) + \mathrm{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\mathrm{rank}(x : A)$, and for $x \in B$ we know $\mathrm{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine rank$(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \mathrm{rank}(c_i : A \cup B)$.

This means we just need to determine $\mathrm{rank}(x : A \cup B)$ for all elements!

Observe, that $\mathrm{rank}(x : A \cup B) = \mathrm{rank}(x : A) + \mathrm{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\mathrm{rank}(x : A)$, and for $x \in B$ we know $\mathrm{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \text{rank}(c_i : A \cup B)$.

This means we just need to determine $\text{rank}(x : A \cup B)$ for all elements!

Observe, that $\text{rank}(x : A \cup B) = \text{rank}(x : A) + \text{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\text{rank}(x : A)$, and for $x \in B$ we know $\text{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \ldots a_n)$ and $B = (b_1 \ldots b_n)$, compute the sorted squence $C = (c_1 \ldots c_n)$.

Observation:
We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \mathrm{rank}(c_i : A \cup B)$.

This means we just need to determine $\mathrm{rank}(x : A \cup B)$ for all elements!

Observe, that $\mathrm{rank}(x : A \cup B) = \mathrm{rank}(x : A) + \mathrm{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\mathrm{rank}(x : A)$, and for $x \in B$ we know $\mathrm{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Given two sorted sequences $A = (a_1 \dots a_n)$ and $B = (b_1 \dots b_n)$, compute the sorted squence $C = (c_1 \dots c_n)$.

Observation:

We can assume wlog. that elements in A and B are different.

Then for $c_i \in C$ we have $i = \mathrm{rank}(c_i : A \cup B)$.

This means we just need to determine $\mathrm{rank}(x : A \cup B)$ for all elements!

Observe, that $\mathrm{rank}(x : A \cup B) = \mathrm{rank}(x : A) + \mathrm{rank}(x : B)$.

Clearly, for $x \in A$ we already know $\mathrm{rank}(x : A)$, and for $x \in B$ we know $\mathrm{rank}(x : B)$.

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 8
On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.

can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 8

On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 8

On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

Compute $\text{rank}(x : A)$ for all $x \in B$ and $\text{rank}(x : B)$ for all $x \in A$.
can be done in $\mathcal{O}(\log n)$ time with $2n$ processors by binary search

Lemma 8
On a CREW PRAM, Merging can be done in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n \log n)$ work.

not optimal

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;

$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

1: $j_0 \leftarrow 0$
2: $j_k \leftarrow n$
3: **for** $1 \le i \le k-1$ **pardo**
4: $\quad j_i \leftarrow \mathrm{rank}(b_{i\log n} : A)$
5: **for** $0 \le i \le k-1$ **pardo**
6: $\quad B_i \leftarrow (b_{i\log n+1}, \ldots, b_{(i+1)\log n})$
7: $\quad A_i \leftarrow (a_{j_i+1}, \ldots, a_{j_{i+1}})$

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

```
j_0 ← 0
j_k ← n
for 1 ≤ i ≤ k − 1 pardo
    j_i ← rank(b_{i log n} : A)
for 0 ≤ i ≤ k − 1 pardo
    B_i ← (b_{i log n+1}, ..., b_{(i+1) log n})
    A_i ← (a_{j_i+1}, ..., a_{j_{i+1}})
```

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \dots, a_n)$; $B = (b_1, \dots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

```
j_0 ← 0
j_k ← n
for 1 ≤ i ≤ k − 1 pardo
    j_i ← rank(b_{i log n} : A)
for 0 ≤ i ≤ k − 1 pardo
    B_i ← (b_{i log n+1}, ..., b_{(i+1) log n})
    A_i ← (a_{j_i+1}, ..., a_{j_{i+1}})
```

If C_i is the merging of A_i and B_i then the sequence $C_0 \dots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

$A = (a_1, \ldots, a_n)$; $B = (b_1, \ldots, b_n)$;
$\log n$ integral; $k := n/\log n$ integral;

Algorithm 8 GenerateSubproblems

1: $j_0 \leftarrow 0$
2: $j_k \leftarrow n$
3: **for** $1 \le i \le k-1$ **pardo**
4: $\quad j_i \leftarrow \mathrm{rank}(b_{i\log n} : A)$
5: **for** $0 \le i \le k-1$ **pardo**
6: $\quad B_i \leftarrow (b_{i\log n+1}, \ldots, b_{(i+1)\log n})$
7: $\quad A_i \leftarrow (a_{j_i+1}, \ldots, a_{j_{i+1}})$

If C_i is the merging of A_i and B_i then the sequence $C_0 \ldots C_{k-1}$ is a sorted sequence.

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.3 Divide & Conquer — Merging

We can generate the subproblems in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Note that in a sub-problem B_i has length $\log n$.

If we run the algorithm again for every subproblem, (where A_i takes the role of B) we can in time $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n)$ generate subproblems where A_j and B_j have both length at most $\log n$.

Such a subproblem can be solved by a single processor in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(|A_i| + |B_i|)$.

Parallelizing the last step gives total work $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$.

the resulting algorithm is work optimal

4.4 Maximum Computation

Lemma 9

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(1)$ with n^2 processors.

proof on board...

4.4 Maximum Computation

Lemma 9

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(1)$ with n^2 processors.

proof on board...

4.4 Maximum Computation

Lemma 10

On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n\log\log n)$.

proof on board...

4.4 Maximum Computation

Lemma 10
On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n\log\log n)$.

proof on board...

4.4 Maximum Computation

Lemma 11
On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n)$.

proof on board...

4.4 Maximum Computation

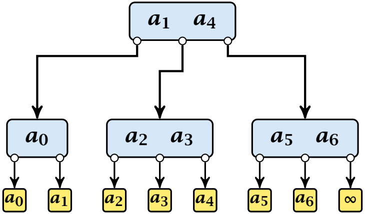
Lemma 11
On a CRCW PRAM the maximum of n numbers can be computed in time $\mathcal{O}(\log\log n)$ with n processors and work $\mathcal{O}(n)$.

proof on board...

4.5 Inserting into a $(2,3)$-tree

Given a $(2,3)$-tree with n elements, and a sequence $x_0 < x_1 < x_2 < \cdots < x_k$ of elements. We want to insert elements $x_1, \ldots, x_k$ into the tree ($k \ll n$).

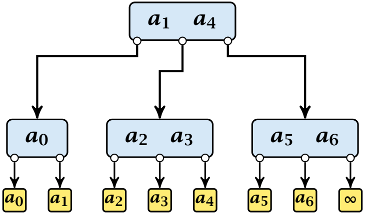
time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$



4.5 Inserting into a $(2,3)$-tree

Given a $(2,3)$-tree with n elements, and a sequence $x_0 < x_1 < x_2 < \dots < x_k$ of elements. We want to insert elements $x_1, \dots, x_k$ into the tree ($k \ll n$).

time: $\mathcal{O}(\log n)$; **work:** $\mathcal{O}(k \log n)$



4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(\log k)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n + k)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(\log k)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n + k)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(\log k)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n + k)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

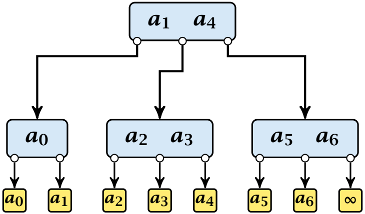
   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(\log k)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n + k)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

4.5 Inserting into a $(2,3)$-tree

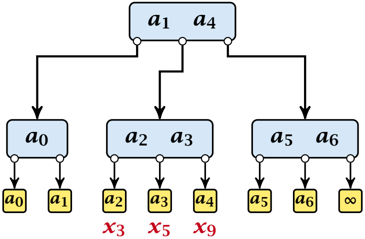
1. determine for every x_i the leaf element before which it has to be inserted
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k \log n)$; CREW PRAM

   all x_i's that have to be inserted before the same element form a chain
2. determine the largest/smallest/middle element of every chain
   time: $\mathcal{O}(\log k)$; work: $\mathcal{O}(k)$;
3. insert the middle element of every chain
   compute new chains
   time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(k_i \log n + k)$; k_i= #inserted elements
   (computing new chains is constant time)
4. repeat Step 3 for logarithmically many rounds
   time: $\mathcal{O}(\log n \log k)$; work: $\mathcal{O}(k \log n)$;

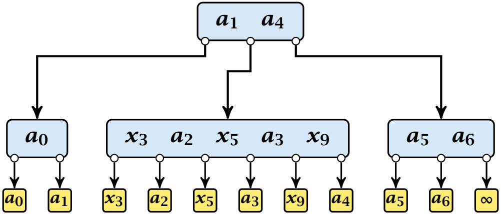
Step 3



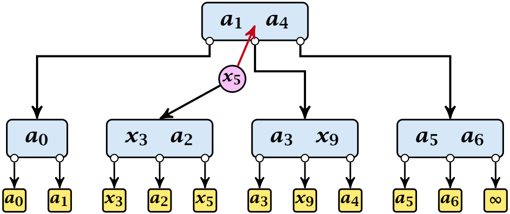
Step 3



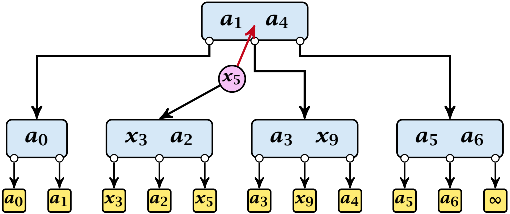
Step 3



Step 3

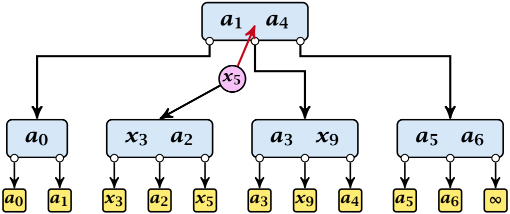


Step 3



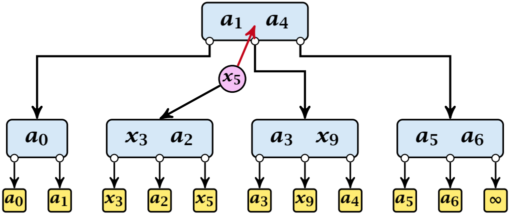
- each internal node is split into at most two parts

Step 3



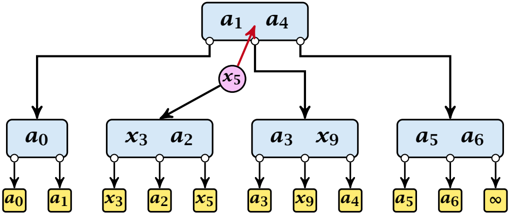
- each internal node is split into at most two parts
- each split operation promotes at most one element

Step 3



- each internal node is split into at most two parts
- each split operation promotes at most one element
- hence, on every level we want to insert at most one element per successor pointer

Step 3



- each internal node is split into at most two parts
- each split operation promotes at most one element
- hence, on every level we want to insert at most one element per successor pointer
- we can use the same routine for every level

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.5 Inserting into a $(2,3)$-tree

- Step 3, works in phases; one phase for every level of the tree
- Step 4, works in rounds; in each round a different set of elements is inserted

Observation

We can start with phase i of round r as long as phase i of round $r-1$ and (of course), phase $i-1$ of round r has finished.

This is called Pipelining. Using this technique we can perform all rounds in Step 4 in just $\mathcal{O}(\log k + \log n)$ many parallel steps.

4.6 Symmetry Breaking

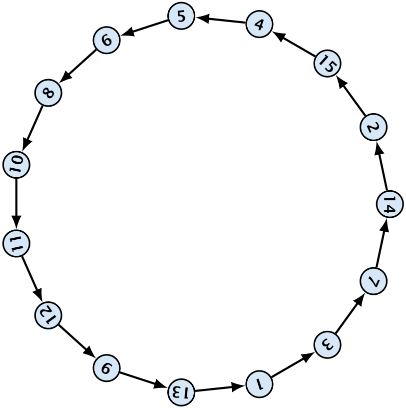
The following algorithm colors an n-node cycle with $\lceil \log n \rceil$ colors.

Algorithm 9 BasicColoring

```
1: for 1 ≤ i ≤ n pardo
2:     col(i) ← i
3:     k_i ← smallest bitpos where col(i) and col(S(i)) differ
4:     col'(i) ← 2k_i + col(i)_{k_i}
```

1: **for** $1 \le i \le n$ **pardo**
2: $\quad \mathrm{col}(i) \leftarrow i$
3: $\quad k_i \leftarrow$ smallest bitpos where $\mathrm{col}(i)$ and $\mathrm{col}(S(i))$ differ
4: $\quad \mathrm{col}'(i) \leftarrow 2k_i + \mathrm{col}(i)_{k_i}$

(bit positions are numbered starting with 0)



v	col	k	col′
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \leq \lceil \log_2(2t) \rceil = \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\mathcal{O}(\log^* n)$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \leq \lceil \log_2(2t) \rceil = \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\mathcal{O}(\log^* n)$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1) \rceil \le \lceil \log_2(2t) \rceil = \lceil \log_2(t) \rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\mathcal{O}(\log^* n)$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1)\rceil \le \lceil \log_2(2t)\rceil = \lceil \log_2(t)\rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\mathcal{O}(\log^* n)$ operations.

4.6 Symmetry Breaking

Applying the algorithm to a coloring with bit-length t generates a coloring with largest color at most

$$2(t-1)+1$$

and bit-length at most

$$\lceil \log_2(2(t-1)+1)\rceil \le \lceil \log_2(2t)\rceil = \lceil \log_2(t)\rceil + 1$$

Applying the algorithm repeatedly generates a constant number of colors after $\mathcal{O}(\log^* n)$ operations.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

for $\ell \leftarrow 5$ **to** 3
    for $1 \leq i \leq n$ **pardo**
        if $\mathrm{col}(i) = \ell$ **then**
        $\mathrm{col}(i) \leftarrow \min\{\{0,1,2\} \setminus \{\mathrm{col}(P[i]), \mathrm{col}(S[i])\}\}$

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
            col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

for $\ell \leftarrow 5$ **to** 3
  for $1 \leq i \leq n$ **pardo**
    if $\mathrm{col}(i) = \ell$ **then**
      $\mathrm{col}(i) \leftarrow \min\{\{0, 1, 2\} \setminus \{\mathrm{col}(P[i]), \mathrm{col}(S[i])\}\}$

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

As long as the bit-length $t \geq 4$ the bit-length decreases.

Applying the algorithm with bit-length 3 gives a coloring with colors in the range $0, \ldots, 5 = 2t - 1$.

We can improve to a 3-coloring by successively re-coloring nodes from a color-class:

Algorithm 10 ReColor

```
for ℓ ← 5 to 3
    for 1 ≤ i ≤ n pardo
        if col(i) = ℓ then
            col(i) ← min{{0,1,2} \ {col(P[i]), col(S[i])}}
```

This requires time $\mathcal{O}(1)$ and work $\mathcal{O}(n)$.

4.6 Symmetry Breaking

Lemma 12
We can color vertices in a ring with three colors in $\mathcal{O}(\log^* n)$ *time and with* $\mathcal{O}(n \log^* n)$ *work.*

not work optimal

4.6 Symmetry Breaking

Lemma 13

Given n integers in the range $0, \ldots, \mathcal{O}(\log n)$, there is an algorithm that sorts these numbers in $\mathcal{O}(\log n)$ time using a linear number of operations.

Proof: Exercise!

4.6 Symmetry Breaking

Algorithm 11 OptColor

for $1 \le i \le n$ **pardo**
    $\mathrm{col}(i) \leftarrow i$
apply BasicColoring once
sort vertices by colors
for $\ell = 2\lceil \log n \rceil$ **to** 3 **do**
    for all vertices i of color ℓ **pardo**
        $\mathrm{col}(i) \leftarrow \min\{\{0,1,2\} \setminus \{\mathrm{col}(P[i]), \mathrm{col}(S[i])\}\}$

Lemma 14

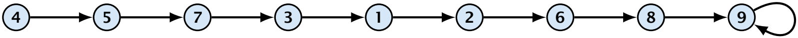
A ring can be colored with 3 colors in time $\mathcal{O}(\log n)$ and with work $\mathcal{O}(n)$.

work optimal but not too fast

List Ranking

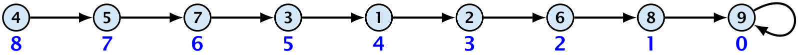
Input:

A list given by successor pointers;



Output:

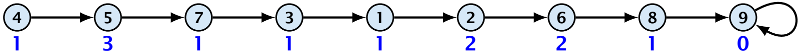
For every node number of hops to end of the list;



Observation:

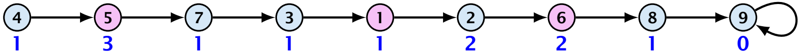
Special case of parallel prefix

List Ranking



1. Given a list with values; perhaps from previous iterations.
   The list is given via predecessor pointers $P(i)$ and successor pointers $S(i)$.
   $S(4) = 5$, $S(2) = 6$, $P(3) = 7$, etc.

List Ranking

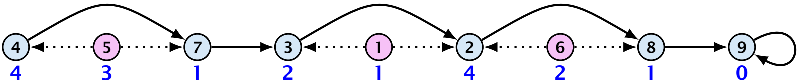


2. Find an independent set; time: $\mathcal{O}(\log n)$; work: $\mathcal{O}(n)$.

   The independent set should contain a constant fraction of the vertices.

   Color vertices; take local minima

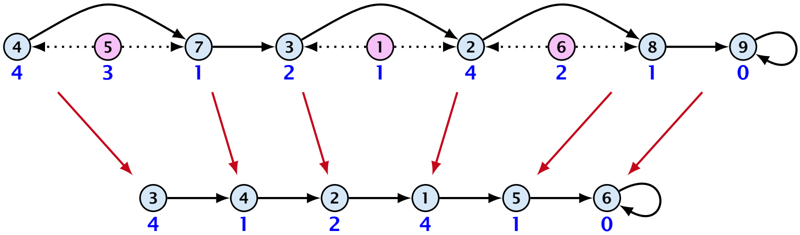
List Ranking



3. Splice the independent set out of the list;

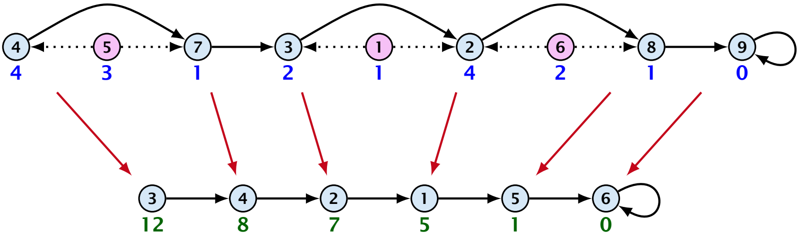
   At the independent set vertices the array still contains old values for $P(i)$ and $S(i)$;

List Ranking



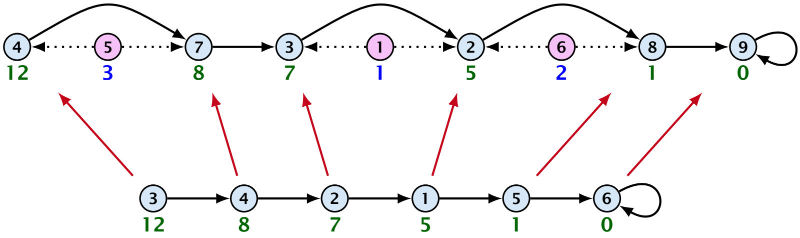
4. Compress remaining n' nodes into a new array of n' entries.
   The index positions can be computed by a prefix sum in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$
   Pointers can then be adjusted in time $\mathcal{O}(1)$.

List Ranking



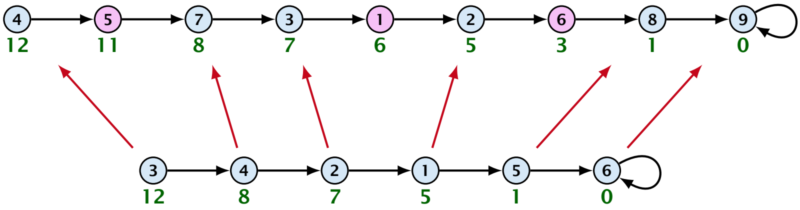
5. Solve the problem on the remaining list.
   If current size is less than $n/\log n$ do pointer jumping: time $\mathcal{O}(\log n)$; work $\mathcal{O}(n)$.
   Otherwise continue shrinking the list by finding an independent set

List Ranking



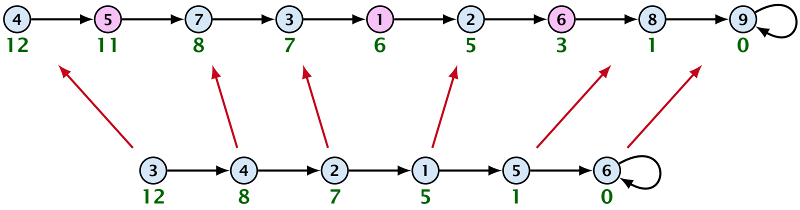
6. Map the values back into the larger list. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(n)$

List Ranking



7. Compute values for independent set nodes. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(1)$.
8. Splice nodes back into list. Time: $\mathcal{O}(1)$; Work: $\mathcal{O}(1)$.

List Ranking



We need $\mathcal{O}(\log\log n)$ shrinking iterations until the size of the remaining list reaches $\mathcal{O}(n/\log n)$.

Each shrinking iteration takes time $\mathcal{O}(\log n)$.

The work for all shrinking operations is just $\mathcal{O}(n)$, as the size of the list goes down by a constant factor in each round.

List Ranking can be solved in time $\mathcal{O}(\log n \log\log n)$ and work $\mathcal{O}(n)$ on an EREW-PRAM.

We need $\mathcal{O}(\log\log n)$ shrinking iterations until the size of the remaining list reaches $\mathcal{O}(n/\log n)$.

Each shrinking iteration takes time $\mathcal{O}(\log n)$.

The work for all shrinking operations is just $\mathcal{O}(n)$, as the size of the list goes down by a constant factor in each round.

List Ranking can be solved in time $\mathcal{O}(\log n \log\log n)$ and work $\mathcal{O}(n)$ on an EREW-PRAM.

We need $\mathcal{O}(\log\log n)$ shrinking iterations until the size of the remaining list reaches $\mathcal{O}(n/\log n)$.

Each shrinking iteration takes time $\mathcal{O}(\log n)$.

The work for all shrinking operations is just $\mathcal{O}(n)$, as the size of the list goes down by a constant factor in each round.

List Ranking can be solved in time $\mathcal{O}(\log n \log\log n)$ and work $\mathcal{O}(n)$ on an EREW-PRAM.

We need $\mathcal{O}(\log\log n)$ shrinking iterations until the size of the remaining list reaches $\mathcal{O}(n/\log n)$.

Each shrinking iteration takes time $\mathcal{O}(\log n)$.

The work for all shrinking operations is just $\mathcal{O}(n)$, as the size of the list goes down by a constant factor in each round.

List Ranking can be solved in time $\mathcal{O}(\log n \log\log n)$ and work $\mathcal{O}(n)$ on an EREW-PRAM.

Optimal List Ranking

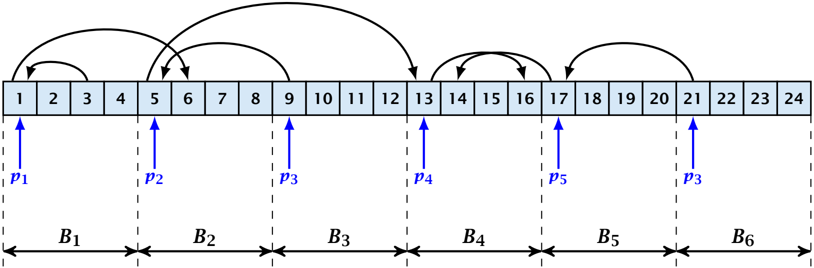
In order to reduce the work we have to improve the shrinking of the list to $\mathcal{O}(n/\log n)$ nodes.

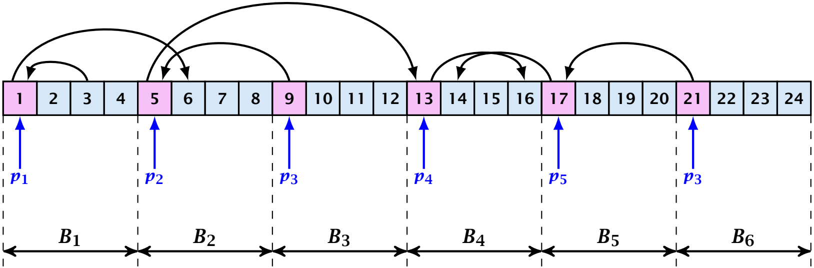
After this we apply pointer jumping

Optimal List Ranking

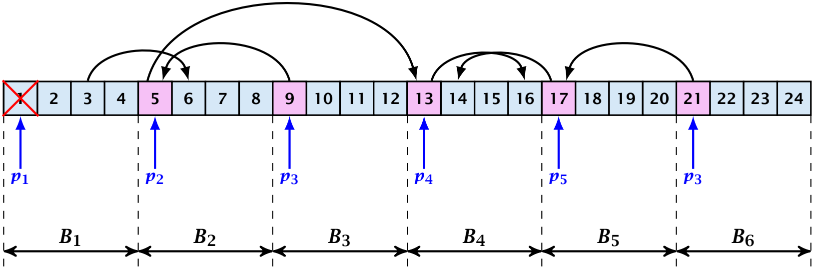
In order to reduce the work we have to improve the shrinking of the list to $\mathcal{O}(n/\log n)$ nodes.

After this we apply pointer jumping

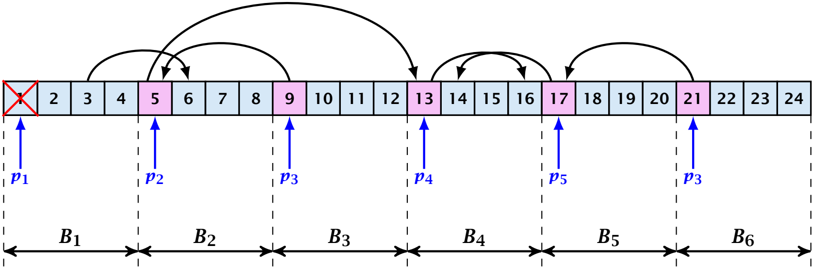

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
p_1
p_2
p_3
p_4
p_5
p_3
B_1
B_2
B_3
B_4
B_5
B_6



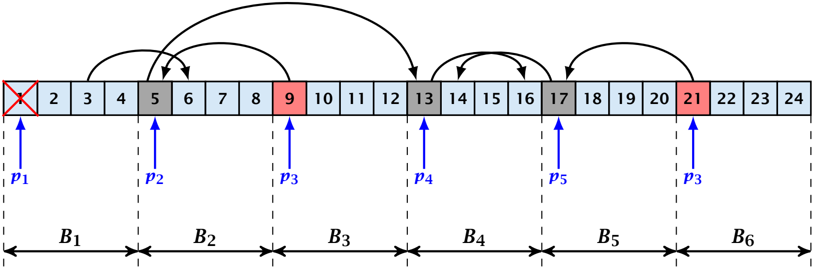
- some nodes are active;
- active nodes without neighbouring active nodes are isolated;
- the others form sublists;



1 delete isolated nodes from the list;

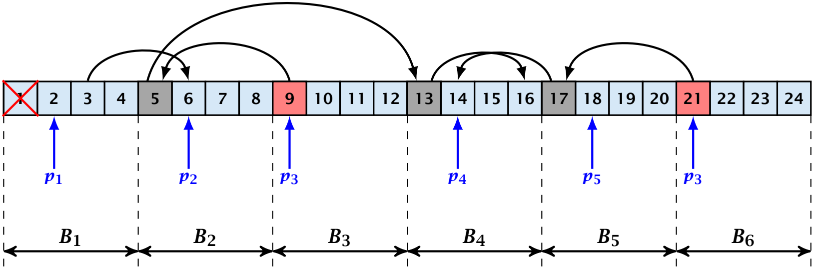


1. delete isolated nodes from the list;
2. color each sublist with $\mathcal{O}(\log\log n)$ colors; time: $\mathcal{O}(1)$; work: $\mathcal{O}(n)$;

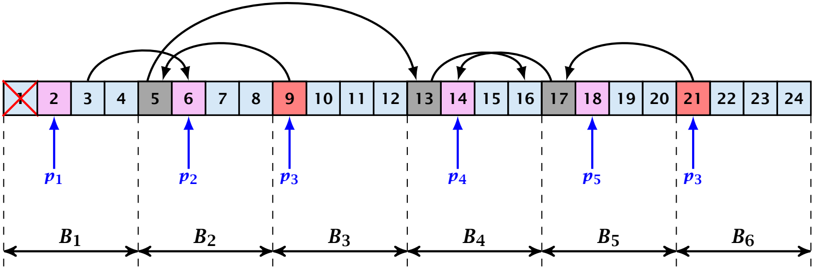


1 delete isolated nodes from the list;
2 color each sublist with $\mathcal{O}(\log\log n)$ colors; time: $\mathcal{O}(1)$; work: $\mathcal{O}(n)$;

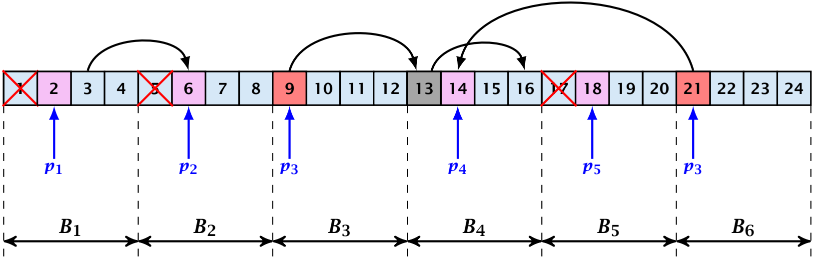
label local minima w.r.t. color as ruler; others as subject
first node of sublist is ruler; needs to be changed!!!



3 advance pointers of removed nodes and of subjects;



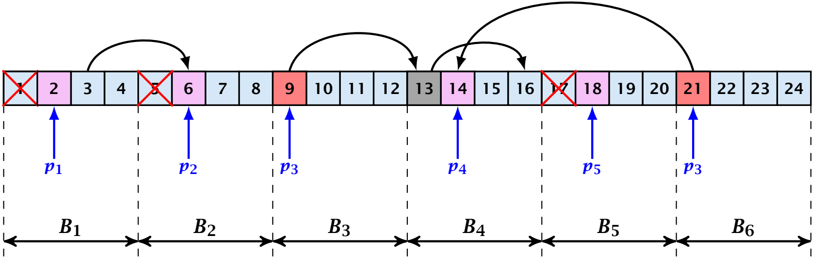
3 advance pointers of removed nodes and of subjects; make new nodes active



New Iteration

0. every ruler deletes its next subject;

   rulers without a subject become active


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
p_1
p_2
p_3
p_4
p_5
p_3
B_1
B_2
B_3
B_4
B_5
B_6

Optimal List Ranking

Each iteration requires constant time and work $\mathcal{O}(n/\log n)$, because we just work on one node in every block.

We need to prove that we just require $\mathcal{O}(\log n)$ iterations to reduce the size of the list to $\mathcal{O}(n/\log n)$.

Optimal List Ranking

Each iteration requires constant time and work $\mathcal{O}(n/\log n)$, because we just work on one node in every block.

We need to prove that we just require $\mathcal{O}(\log n)$ iterations to reduce the size of the list to $\mathcal{O}(n/\log n)$.

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

a ruler will delete a subject

an active node either

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either
  - becomes a ruler (with a subject)
  - becomes a subject
  - is isolated and therefore gets deleted

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either
  - becomes a ruler (with a subject)
  - becomes a subject
  - is isolated and therefore gets deleted

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either
  - becomes a ruler (with a subject)
  - becomes a subject
  - is isolated and therefore gets deleted

Observations/Remarks:

- If the p-pointer of a block cannot be advanced without leaving the block, the processor responsible for this block simply stops working; all other blocks continue.
- The p-node of a block (the node p_i is pointing to) at the beginning of a round is either a ruler with a living subject or the node will become active during the round.
- The subject nodes always lie to the left of the p-node of the respective block (if it exists).

Measure of Progress:

- a ruler will delete a subject
- an active node either
  - becomes a ruler (with a subject)
  - becomes a subject
  - is isolated and therefore gets deleted

Analysis

For the analysis we assign a weight to every node in every block as follows.

Definition 15

The weight of the i-th node in a block is

$$(1-q)^i$$

with $q = \frac{1}{\log\log n}$, where the node-numbering starts from 0. Hence, a block has nodes $\{0,\ldots,\log n - 1\}$.

Analysis

For the analysis we assign a weight to every node in every block as follows.

Definition 15
The weight of the i-th node in a block is

$$(1-q)^i$$

with $q = \frac{1}{\log\log n}$, where the node-numbering starts from 0. Hence, a block has nodes $\{0, \dots, \log n - 1\}$.

Definition of Rulers

Properties:

- A ruler should have at most $\log\log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Definition of Rulers

Properties:

- A ruler should have at most $\log \log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Definition of Rulers

Properties:

- A ruler should have at most $\log\log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Definition of Rulers

Properties:

- A ruler should have at most $\log\log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Definition of Rulers

Properties:

- A ruler should have at most $\log\log n$ subjects.
- The weight of a ruler should be at most the weight of any of its subjects.
- Each ruler must have at least one subject.
- We must be able to remove the next subject in constant time.
- We need to make the ruler/subject decision in constant time.

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler

Given a sublist of active nodes.

Color the sublist with $\mathcal{O}(\log\log n)$ colors. Take the local minima w.r.t. this coloring.

If the first node is not a ruler

- if the second node is a ruler switch ruler status between first and second
- otw. just make the first node into a ruler

This partitions the sub-list into chains of length at most $\log\log n$ each starting with a ruler

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Now we change the ruler definition.

Consider some chain.

We make all local minima w.r.t. the weight function into a ruler; ties are broken according to block-id (so that comparing weights gives a strict inequality).

A ruler gets as subjects the nodes left of it until the next local maximum (or the start of the chain) (including the local maximum) and the nodes right of it until the next local maximum (or the end of the chain) (excluding the local maximum).

In case the first node is a ruler the above definition could leave it without a subject. We use constant time to fix this in some arbitrary manner

Set $q = \frac{1}{\log\log n}$.

The i-th node in a block is assigned a weight of $(1-q)^i$, $0 \le i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q\log n}$.

to show:
After $\mathcal{O}(\log n)$ iterations the weight is at most $(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

Set $q = \frac{1}{\log\log n}$.

The i-th node in a block is assigned a weight of $(1-q)^i$, $0 \le i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q\log n}$.

to show:
After $\mathcal{O}(\log n)$ iterations the weight is at most $(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

Set $q = \frac{1}{\log\log n}$.

The i-th node in a block is assigned a weight of $(1-q)^i$, $0 \leq i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q \log n}$.

to show:

After $\mathcal{O}(\log n)$ iterations the weight is at most $(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

Set $q = \frac{1}{\log\log n}$.

The i-th node in a block is assigned a weight of $(1-q)^i$, $0 \le i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q\log n}$.

to show:

After $\mathcal{O}(\log n)$ iterations the weight is at most $(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

Set $q = \frac{1}{\log\log n}$.

The i-th node in a block is assigned a weight of $(1-q)^i$, $0 \le i < \log n$

The total weight of a block is at most $1/q$ and the total weight of all items is at most $\frac{n}{q \log n}$.

to show:
After $\mathcal{O}(\log n)$ iterations the weight is at most $(n/\log n)(1-q)^{\log n}$

This means at most $n/\log n$ nodes remain because the smallest weight a node can have is $(1-q)^{\log n - 1}$.

In every iteration the weight drops by a factor of

$$(1 - q/4) \ .$$

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

We consider subject nodes to just have half their weight.

We can view the step of becoming a subject as a precursor to deletion.

Hence, a node looses half its weight when becoming a subject and the remaining half when deleted.

Note that subject nodes will be deleted after just an additional $\mathcal{O}(\log\log n)$ iterations.

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from p-nodes (ruler/active).

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from p-nodes (ruler/active).

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from p-nodes (ruler/active).

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from p-nodes (ruler/active).

The weight is reduced because

- An isolated node is removed.
- A node is labelled as ruler, and the corresponding subjects reduce their weight by a factor of 1/2.
- A node is a ruler and deletes one of its subjects.

Hence, the weight reduction comes from p-nodes (ruler/active).

Each p-node is responsible for some other nodes; it has to generate a weight reduction large enough so that the weight of all nodes it is responsible for decreases by the desired factor.

An active node is responsible for all nodes that come after it in its block.

A ruler is responsible for all nodes that come after it in its block **and** for all its subjects.

Note that by this definition every node remaining in the list is covered.

Each p-node is responsible for some other nodes; it has to generate a weight reduction large enough so that the weight of all nodes it is responsible for decreases by the desired factor.

An active node is responsible for all nodes that come after it in its block.

A ruler is responsible for all nodes that come after it in its block **and** for all its subjects.

Note that by this definition every node remaining in the list is covered.

Each p-node is responsible for some other nodes; it has to generate a weight reduction large enough so that the weight of all nodes it is responsible for decreases by the desired factor.

An active node is responsible for all nodes that come after it in its block.

A ruler is responsible for all nodes that come after it in its block **and** for all its subjects.

Note that by this definition every node remaining in the list is covered.

Each p-node is responsible for some other nodes; it has to generate a weight reduction large enough so that the weight of all nodes it is responsible for decreases by the desired factor.

An active node is responsible for all nodes that come after it in its block.

A ruler is responsible for all nodes that come after it in its block **and** for all its subjects.

Note that by this definition every node remaining in the list is covered.

Case 1: Isolated Node

Case 1: Isolated Node

Suppose we delete an isolated node v that is the i-th node in its block.

Case 1: Isolated Node

Suppose we delete an isolated node v that is the i-th node in its block.

The weight of all node that v is responsible for is

$$\sum_{i \leq j < \log n} (1-q)^j$$

Case 1: Isolated Node

Suppose we delete an isolated node v that is the i-th node in its block.

The weight of all node that v is responsible for is

$$\sum_{i \le j < \log n} (1-q)^j$$

This weight reduces to

$$\sum_{i < j < \log n} (1-q)^j$$

Case 1: Isolated Node

Suppose we delete an isolated node v that is the i-th node in its block.

The weight of all node that v is responsible for is

$$\sum_{i \leq j < \log n} (1-q)^j$$

This weight reduces to

$$\sum_{i < j < \log n} (1-q)^j \leq (1-q) \sum_{i \leq j < \log n} (1-q)^j$$

Case 1: Isolated Node

Suppose we delete an isolated node v that is the i-th node in its block.

The weight of all node that v is responsible for is

$$\sum_{i \le j < \log n} (1-q)^j$$

This weight reduces to

$$\sum_{i < j < \log n} (1-q)^j \le (1-q) \sum_{i \le j < \log n} (1-q)^j$$

Hence, weight reduces by a factor $(1-q) \le (1-q/4)$.

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

New weight:

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le i \le c \log n} (1-q)^{i} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le \left(1 - \frac{q}{4}\right) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le \left(1 - \frac{q}{4}\right) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le (1 - \frac{q}{4}) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j}$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le \left(1 - \frac{q}{4}\right) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le (1 - \frac{q}{4}) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le (1 - \frac{q}{4}) Q$$

Case 2: Creating Subjects

Suppose we generate a ruler with at least one subject.

Weight of ruler: $(1-q)^{i_1}$.
Weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$.

Initial weight:

$$Q = \sum_{j=1}^{k} \sum_{i_j \le \ell < \log n} (1-q)^{\ell} \le \frac{1}{q} \sum_{j=1}^{k} (1-q)^{i_j} \le \frac{2}{q} \sum_{j=2}^{k} (1-q)^{i_j}$$

New weight:

$$Q' = Q - \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \le (1 - \frac{q}{4}) Q$$

Case 3: Removing Subjects

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \leq j \leq k$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$Q$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$Q = \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j}$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$
\begin{aligned}
Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2} \sum_{j=2}^{k} (1-q)^{i_j} \\
&\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2}
\end{aligned}
$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$
\begin{aligned}
Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2}\sum_{j=2}^{k}(1-q)^{i_j} \\
&\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2} \\
&\le \frac{1}{q}(1-q)^{i_2} + \frac{1}{2q}(1-q)^{i_2}
\end{aligned}
$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$
\begin{aligned}
Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2}\sum_{j=2}^{k}(1-q)^{i_j} \\
&\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2} \\
&\le \frac{1}{q}(1-q)^{i_2} + \frac{1}{2q}(1-q)^{i_2}
\end{aligned}
$$

New weight:

$$Q'$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$
\begin{aligned}
Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2}\sum_{j=2}^{k}(1-q)^{i_j} \\
&\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2} \\
&\le \frac{1}{q}(1-q)^{i_2} + \frac{1}{2q}(1-q)^{i_2}
\end{aligned}
$$

New weight:

$$
Q' = Q - \frac{1}{2}(1-q)^{i_2}
$$

Case 3: Removing Subjects

weight of ruler: $(1-q)^{i_1}$; weight of subjects: $(1-q)^{i_j}$, $2 \le j \le k$

Assume ruler removes subject with largest weight say i_2 (why?).

Initial weight:

$$
\begin{aligned}
Q &= \sum_{i_1 \le \ell < \log n} (1-q)^{\ell} + \frac{1}{2}\sum_{j=2}^{k}(1-q)^{i_j} \\
&\le \frac{1}{q}(1-q)^{i_1} + \frac{k}{2}(1-q)^{i_2} \\
&\le \frac{1}{q}(1-q)^{i_2} + \frac{1}{2q}(1-q)^{i_2}
\end{aligned}
$$

New weight:

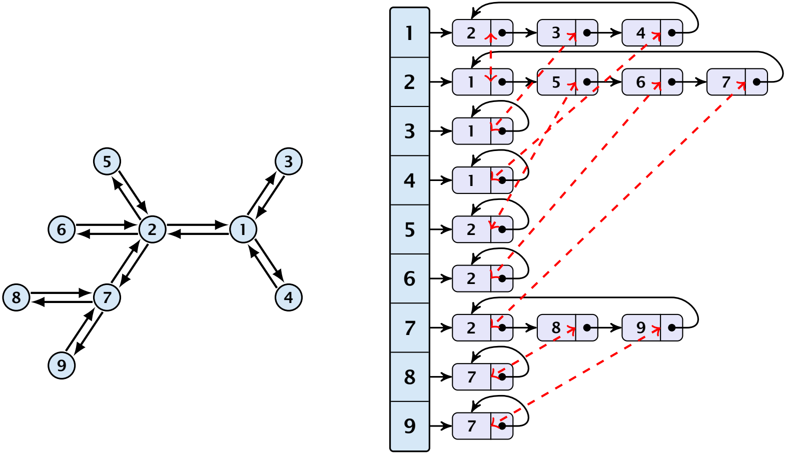
$$
Q' = Q - \frac{1}{2}(1-q)^{i_2} \le (1-\frac{q}{3})Q
$$

After s iterations the weight is at most

$$\frac{n}{q \log n}\left(1 - \frac{q}{4}\right)^s \stackrel{!}{\leq} \frac{n}{\log n}(1-q)^{\log n}$$

Choosing $i = 5 \log n$ the inequality holds for sufficiently large n.

Tree Algorithms



Euler Circuits

Every node v fixes an arbitrary ordering among its adjacent nodes:

$$u_0, u_1, \dots, u_{d-1}$$

We obtain an Euler tour by setting

$$\mathrm{succ}((u_i, v)) = (v, u_{(i+1) \bmod d})$$

Euler Circuits

Lemma 16
An Euler circuit can be computed in constant time $\mathcal{O}(1)$ with $\mathcal{O}(n)$ operations.

Euler Circuits — Applications

Rooting a tree

- split the Euler tour at node r
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 1$ for every edge;
- perform parallel prefix; let $s[\cdot]$ be the result array
- if $s[(u,v)] < s[(v,u)]$ then u is parent of v;

Euler Circuits — Applications

Postorder Numbering

- split the Euler tour at node r
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 1$ for every edge $(v, \mathrm{parent}(v))$
- assign $x[e] = 0$ for every edge $(\mathrm{parent}(v), v)$
- perform parallel prefix
- $\mathrm{post}(v) = s[(v, \mathrm{parent}(v))]$; $\mathrm{post}(r) = n$

Euler Circuits — Applications

Level of nodes

- split the Euler tour at node r
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = -1$ for every edge $(v, \text{parent}(v))$
- assign $x[e] = 1$ for every edge $(\text{parent}(v), v)$
- perform parallel prefix
- $\text{level}(v) = s[(\text{parent}(v), v)]$; $\text{level}(r) = 0$

Euler Circuits — Applications

Number of descendants

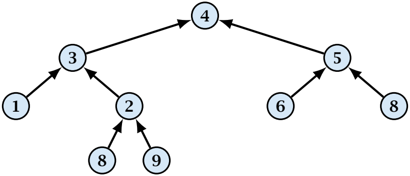
- split the Euler tour at node r
- this gives a list on the set of directed edges (Euler path)
- assign $x[e] = 0$ for every edge $(\text{parent}(v), v)$
- assign $x[e] = 1$ for every edge $(v, \text{parent}(v))$, $v \neq r$
- perform parallel prefix
- $\text{size}(v) = s[(v, \text{parent}(v))] - s[(\text{parent}(v), v)]$

Rake Operation

Given a binary tree T.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove u and $p(u)$
- attach sibling of u to $p(p(u))$

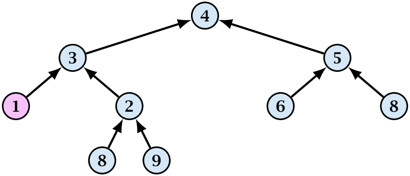


Rake Operation

Given a binary tree T.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove u and $p(u)$
- attach sibling of u to $p(p(u))$

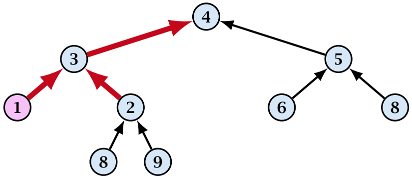


Rake Operation

Given a binary tree T.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove u and $p(u)$
- attach sibling of u to $p(p(u))$

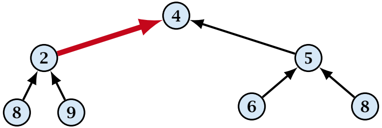


Rake Operation

Given a binary tree T.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove u and $p(u)$
- attach sibling of u to $p(p(u))$

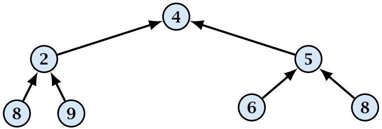


Rake Operation

Given a binary tree T.

Given a leaf $u \in T$ with $p(u) \neq r$ the rake-operation does the following

- remove u and $p(u)$
- attach sibling of u to $p(p(u))$



We want to apply rake operations to a binary tree T until T just consists of the root with two children.

Possible Problems:

- we could concurrently apply the rake-operation to two siblings
- we could concurrently apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

We want to apply rake operations to a binary tree T until T just consists of the root with two children.

Possible Problems:

1. we could concurrently apply the rake-operation to two siblings
2. we could concurrently apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

We want to apply rake operations to a binary tree T until T just consists of the root with two children.

Possible Problems:

1. we could concurrently apply the rake-operation to two siblings
2. we could concurrently apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

We want to apply rake operations to a binary tree T until T just consists of the root with two children.

Possible Problems:

1. we could concurrently apply the rake-operation to two siblings
2. we could concurrently apply the rake-operation to two leaves u and v such that $p(u)$ and $p(v)$ are connected

By choosing leaves carefully we ensure that none of the above cases occurs

Algorithm:

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations

Algorithm:

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even leaves

Algorithm:

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even leaves

Algorithm:

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even leaves

Algorithm:

- label leaves consecutively from left to right (excluding left-most and right-most leaf), and store them in an array A
- for $\lceil \log(n+1) \rceil$ iterations
  - apply rake to all odd leaves that are left children
  - apply rake operation to remaining odd leaves (odd at start of round!!!)
  - A=even leaves

Observations

- the rake operation does not change the order of leaves
- two leaves that are siblings do not perform a rake operation in the same round because one is even and one odd at the start of the round
- two leaves that have adjacent parents either have different parity (even/odd) or they differ in the type of child (left/right)

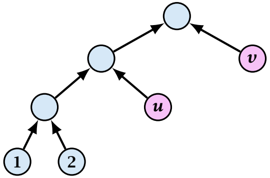
Observations

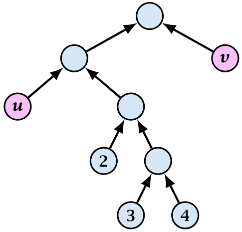
- the rake operation does not change the order of leaves
- two leaves that are siblings do not perform a rake operation in the same round because one is even and one odd at the start of the round
- two leaves that have adjacent parents either have different parity (even/odd) or they differ in the type of child (left/right)

Observations

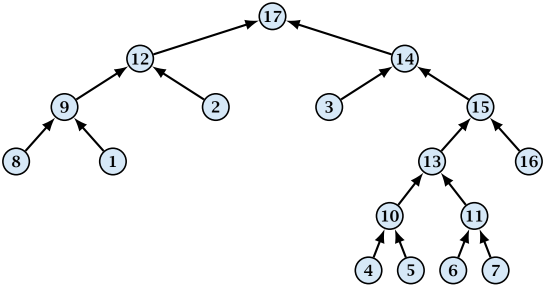
- the rake operation does not change the order of leaves
- two leaves that are siblings do not perform a rake operation in the same round because one is even and one odd at the start of the round
- two leaves that have adjacent parents either have different parity (even/odd) or they differ in the type of child (left/right)

Cases, when the left edge btw. $p(u)$ and $p(v)$ is a left-child edge.

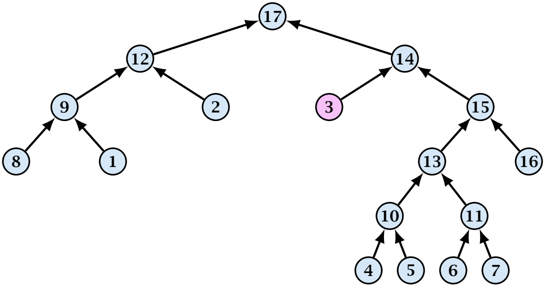




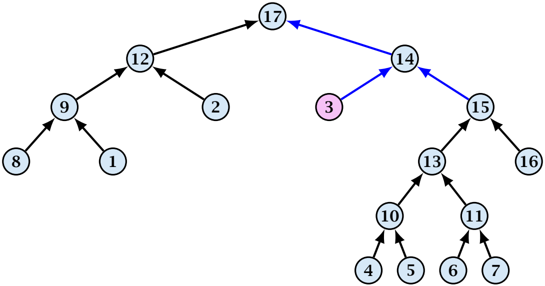
Example



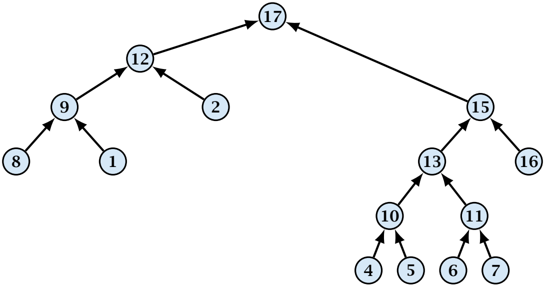
Example



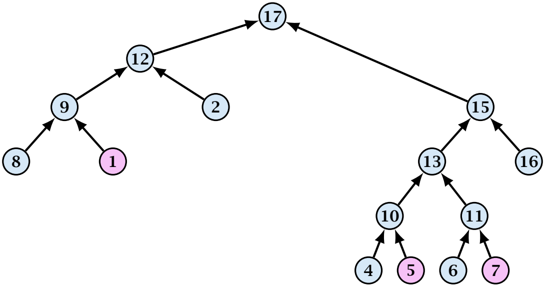
Example



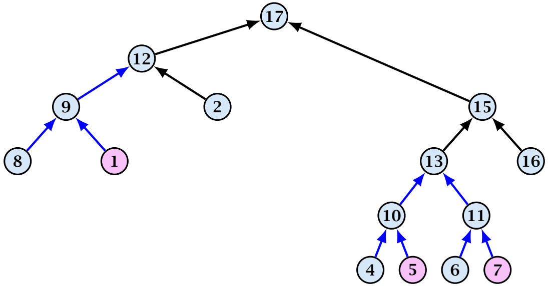
Example



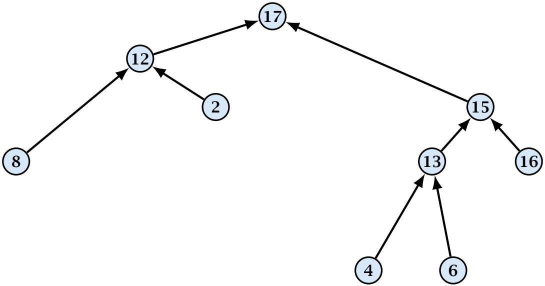
Example



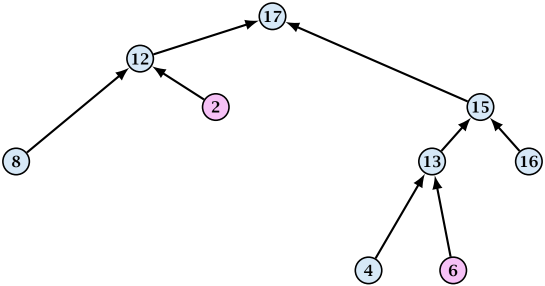
Example



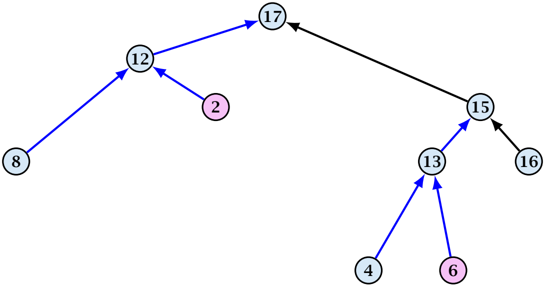
Example



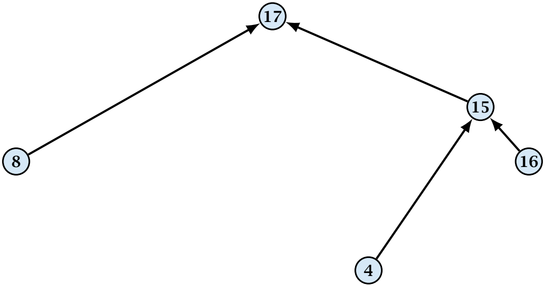
Example



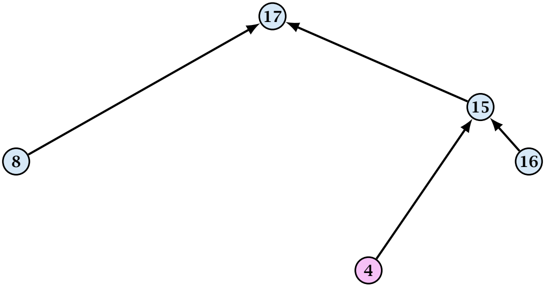
Example



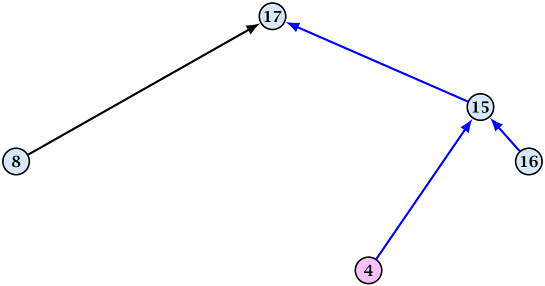
Example



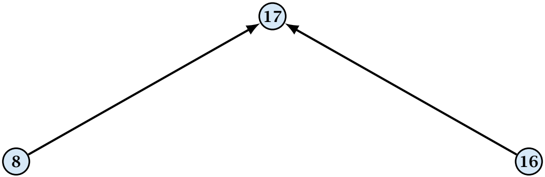
Example



Example



Example



- one iteration can be performed in constant time with $\mathcal{O}(|A|)$ processors, where A is the array of leaves;
- hence, **all** iterations can be performed in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ work;
- the intial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

- one iteration can be performed in constant time with $\mathcal{O}(|A|)$ processors, where A is the array of leaves;
- hence, **all** iterations can be performed in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ work;
- the intial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

- one iteration can be performed in constant time with $\mathcal{O}(|A|)$ processors, where A is the array of leaves;
- hence, **all** iterations can be performed in $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ work;
- the intial parallel prefix also requires time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$

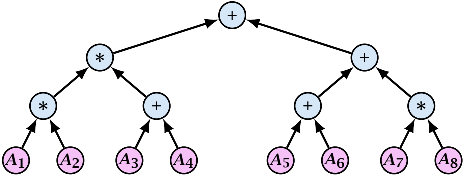
Evaluating Expressions

Suppose that we want to evaluate an expression tree, containing additions and multiplications.

If the tree is not balanced this may be time-consuming.

Evaluating Expressions

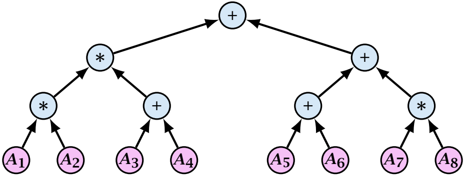
Suppose that we want to evaluate an expression tree, containing additions and multiplications.

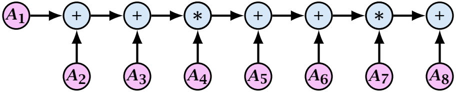


If the tree is not balanced this may be time-consuming.

Evaluating Expressions

Suppose that we want to evaluate an expression tree, containing additions and multiplications.

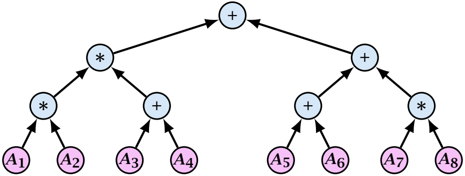


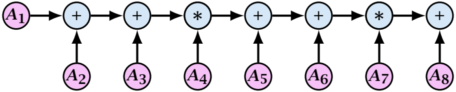


If the tree is not balanced this may be time-consuming.

Evaluating Expressions

Suppose that we want to evaluate an expression tree, containing additions and multiplications.





If the tree is not balanced this may be time-consuming.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:
Let u be internal node with children v and w. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:

Let u be internal node with children v and w. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:
Let u be internal node with children v and w. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:
Let u be internal node with children v and w. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

We can use the rake-operation to do this quickly.

Applying the rake-operation changes the tree.

In order to maintain the value we introduce parameters a_v and b_v for every node that still allows to compute the value of a node based on the value of its children.

Invariant:

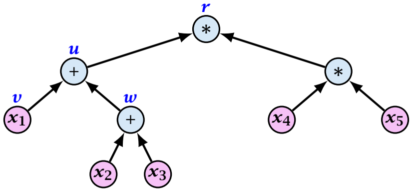
Let u be internal node with children v and w. Then

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) \otimes (a_w \cdot \mathrm{val}(w) + b_w)$$

where $\otimes \in \{*, +\}$ is the operation at node u.

Initially, we can choose $a_v = 1$ and $b_v = 0$ for every node.

Rake Operation



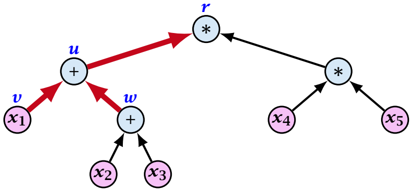
Currently the value at u is

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w)$$
$$= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)$$

In the expression for r this goes in as

$$a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u$$
$$= \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}$$

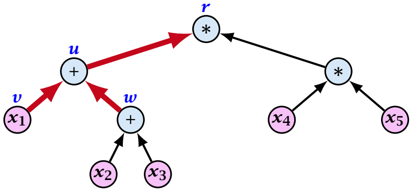
Rake Operation



Currently the value at u is

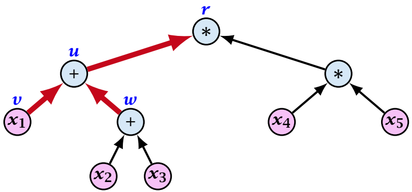
In the expression for r this goes in as

Rake Operation



Currently the value at u is

Rake Operation



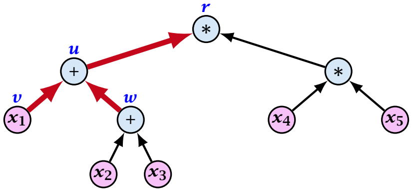
Currently the value at u is

$$\begin{aligned}\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)\end{aligned}$$

In the expression for r this goes in as

$$\begin{aligned}&a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u \\ &= \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}\end{aligned}$$

Rake Operation



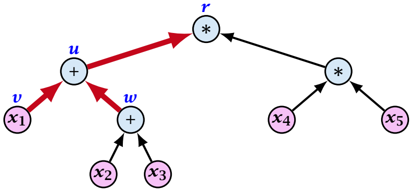
Currently the value at u is

$$\mathrm{val}(u) = (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w)$$
$$= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)$$

In the expression for r this goes in as

$$a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u$$
$$= \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}$$

Rake Operation

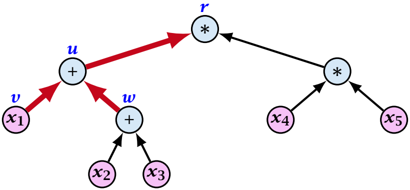


Currently the value at u is

$$
\begin{aligned}
\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\
&= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)
\end{aligned}
$$

In the expression for r this goes in as

Rake Operation



Currently the value at u is

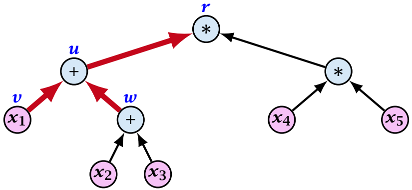
$$\begin{aligned}\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)\end{aligned}$$

In the expression for r this goes in as

$$a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u$$

$$= \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}$$

Rake Operation



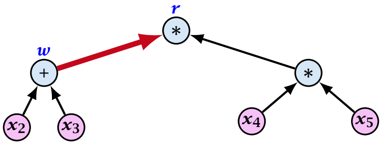
Currently the value at u is

$$
\begin{aligned}
\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\
&= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)
\end{aligned}
$$

In the expression for r this goes in as

$$
\begin{aligned}
&a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u \\
&\quad = \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}
\end{aligned}
$$

Rake Operation



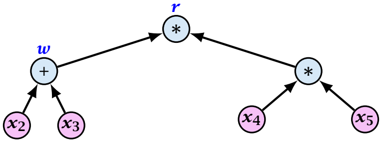
Currently the value at u is

$$\begin{aligned}\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)\end{aligned}$$

In the expression for r this goes in as

$$\begin{aligned}&a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u \\ &\quad = \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}\end{aligned}$$

Rake Operation



Currently the value at u is

$$\begin{aligned}\mathrm{val}(u) &= (a_v \cdot \mathrm{val}(v) + b_v) + (a_w \cdot \mathrm{val}(w) + b_w) \\ &= x_1 + (a_w \cdot \mathrm{val}(w) + b_w)\end{aligned}$$

In the expression for r this goes in as

$$\begin{aligned}&a_u \cdot [x_1 + (a_w \cdot \mathrm{val}(w) + b_w)] + b_u \\ &\quad = \underbrace{a_u a_w}_{a'_w} \cdot \mathrm{val}(w) + \underbrace{a_u x_1 + a_u b_w + b_u}_{b'_w}\end{aligned}$$

If we change the a and b-values during a rake-operation according to the previous slide we can calculate the value of the root in the end.

Lemma 17

We can evaluate an arithmetic expression tree in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$ regardless of the height or depth of the tree.

By performing the rake-operation in the reverse order we can also compute the value at each node in the tree.

If we change the a and b-values during a rake-operation according to the previous slide we can calculate the value of the root in the end.

Lemma 17

We can evaluate an arithmetic expression tree in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$ regardless of the height or depth of the tree.

By performing the rake-operation in the reverse order we can also compute the value at each node in the tree.

If we change the a and b-values during a rake-operation according to the previous slide we can calculate the value of the root in the end.

Lemma 17

We can evaluate an arithmetic expression tree in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$ regardless of the height or depth of the tree.

By performing the rake-operation in the reverse order we can also compute the value at each node in the tree.

Lemma 18

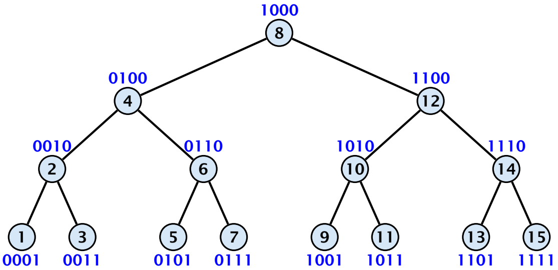
We compute *tree functions* *for arbitrary trees in time* $\mathcal{O}(\log n)$ *and a linear number of operations.*

proof on board...

In the LCA (least common ancestor) problem we are given a tree and the goal is to design a data-structure that answers LCA-queries in constant time.

Least Common Ancestor

LCAs on complete binary trees (inorder numbering):

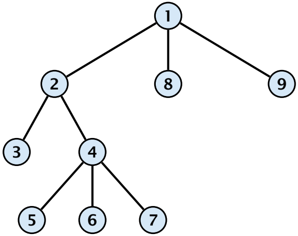


The least common ancestor of u and v is

$$z_1 z_2 \dots z_i 1 0 \dots 0$$

where z_{i+1} is the first bit-position in which u and v differ.

Least Common Ancestor



nodes	1	2	3	2	4	5	4	6	4	7	4	2	1	8	1	9	1
levels	0	1	2	1	2	3	2	3	2	3	2	1	0	1	0	1	0

$\ell(v)$ is index of first appearance of v in node-sequence.

$r(v)$ is index of last appearance of v in node-squence.

$\ell(v)$ and $r(v)$ can be computed in constant time, given the node- and level-sequence.

Least Common Ancestor

Lemma 19

1. *u is ancestor of v iff $\ell(u) < \ell(v) < r(u)$*
2. *u and v are not related iff either $r(u) < \ell(v)$ or $r(v) < \ell(u)$*
3. *suppose $r(u) < \ell(v)$ then $\mathrm{LCA}(u, v)$ is vertex with minimum level over interval $[r(u), \ell(v)]$.*

Range Minima Problem

Given an array $A[1 \dots n]$, a range minimum query (ℓ, r) consists of a left index $\ell \in \{1, \dots, n\}$ and a right index $r \in \{1, \dots, n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell \dots r]$.

The goal in the range minima problem is to preprocess the array such that range minima queries can be answered quickly (constant time).

Range Minima Problem

Given an array $A[1 \dots n]$, a range minimum query (ℓ, r) consists of a left index $\ell \in \{1, \dots, n\}$ and a right index $r \in \{1, \dots, n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell \dots r]$.

The goal in the range minima problem is to preprocess the array such that range minima queries can be answered quickly (constant time).

Range Minima Problem

Given an array $A[1\ldots n]$, a range minimum query (ℓ, r) consists of a left index $\ell \in \{1,\ldots,n\}$ and a right index $r \in \{1,\ldots,n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell\ldots r]$.

The goal in the range minima problem is to preprocess the array such that range minima queries can be answered quickly (constant time).

Range Minima Problem

Given an array $A[1\ldots n]$, a range minimum query (ℓ, r) consists of a left index $\ell \in \{1, \ldots, n\}$ and a right index $r \in \{1, \ldots, n\}$.

The answer has to return the index of the minimum element in the subsequence $A[\ell \ldots r]$.

The goal in the range minima problem is to preprocess the array such that range minima queries can be answered quickly (constant time).

Observation

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

Remark

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the restricted range minima problem where two successive elements in the array just differ by $+1$ or -1.

Observation

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

Remark

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the restricted range minima problem where two successive elements in the array just differ by $+1$ or -1.

Observation

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

Remark

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the restricted range minima problem where two successive elements in the array just differ by $+1$ or -1.

Observation

Given an algorithm for solving the range minima problem in time $T(n)$ and work $W(n)$ we can obtain an algorithm that solves the LCA-problem in time $\mathcal{O}(T(n) + \log n)$ and work $\mathcal{O}(n + W(n))$.

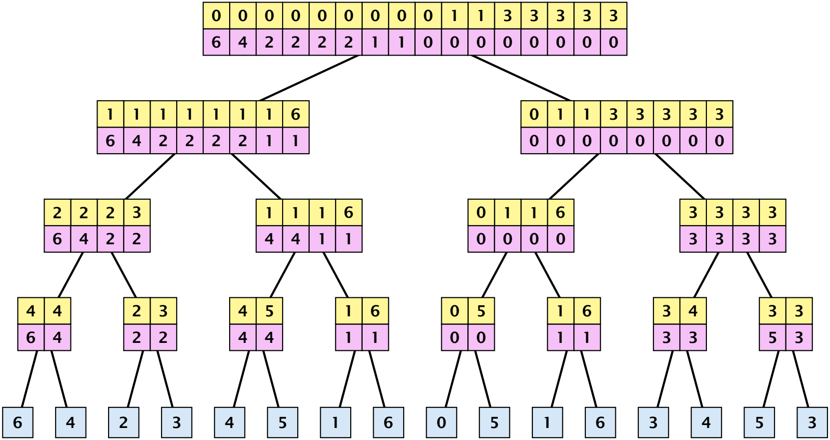
Remark

In the sequential setting the LCA-problem and the range minima problem are equivalent. This is not necessarily true in the parallel setting.

For solving the LCA-problem it is sufficient to solve the restricted range minima problem where two successive elements in the array just differ by $+1$ or -1.

Prefix and Suffix Minima

Tree with prefix-minima and suffix-minima:



- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x.
- The minimum of these two values is the result.

- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x.
- The minimum of these two values is the result.

- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x.
- The minimum of these two values is the result.

- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x.
- The minimum of these two values is the result.

- Suppose we have an array A of length $n = 2^k$
- We compute a complete binary tree T with n leaves.
- Each internal node corresponds to a subsequence of A. It contains an array with the prefix and suffix minima of this subsequence.

Given the tree T we can answer a range minimum query (ℓ, r) in constant time.

- we can determine the LCA x of ℓ and r in constant time since T is a complete binary tree
- Then we consider the suffix minimum of ℓ in the left child of x and the prefix minimum of r in the right child of x.
- The minimum of these two values is the result.

Lemma 20

We can solve the range minima problem in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n \log n)$.

Reducing the Work

Partition A into blocks B_i of length $\log n$

Preprocess each B_i block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block B_i compute the minimum x_i and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \ldots, x_{n/\log n})$.

Reducing the Work

Partition A into blocks B_i of length $\log n$

Preprocess each B_i block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block B_i compute the minimum x_i and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \dots, x_{n/\log n})$.

Reducing the Work

Partition A into blocks B_i of length $\log n$

Preprocess each B_i block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block B_i compute the minimum x_i and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \ldots, x_{n/\log n})$.

Reducing the Work

Partition A into blocks B_i of length $\log n$

Preprocess each B_i block separately by a sequential algorithm so that range-minima queries within the block can be answered in constant time. (**how?**)

For each block B_i compute the minimum x_i and its prefix and suffix minima.

Use the previous algorithm on the array $(x_1, \ldots, x_{n/\log n})$.

Answering a query (ℓ, r):

- if ℓ and r are from the same block the data-structure for this block gives us the result in constant time
- if ℓ and r are from different blocks the result is a minimum of three elements:
  - the suffix minmum of entry ℓ in ℓ's block
  - the minimum among $x_{\ell+1}, \ldots, x_{r-1}$
  - the prefix minimum of entry r in r's block

Answering a query (ℓ, r):

- if ℓ and r are from the same block the data-structure for this block gives us the result in constant time
- if ℓ and r are from different blocks the result is a minimum of three elements:
  - the suffix minmum of entry ℓ in ℓ's block
  - the minimum among $x_{\ell+1}, \ldots, x_{r-1}$
  - the prefix minimum of entry r in r's block

Searching

An extension of binary search with p processors gives that one can find the rank of an element in

$$\log_{p+1}(n) = \frac{\log n}{\log(p+1)}$$

many parallel steps with p processors. (not work-optimal)

This requires a CREW PRAM model. For the EREW model searching cannot be done faster than $\mathcal{O}(\log n - \log p)$ with p processors even if there are p copies of the search key.

Searching

An extension of binary search with p processors gives that one can find the rank of an element in

$$\log_{p+1}(n) = \frac{\log n}{\log(p+1)}$$

many parallel steps with p processors. (not work-optimal)

This requires a CREW PRAM model. For the EREW model searching cannot be done faster than $\mathcal{O}(\log n - \log p)$ with p processors even if there are p copies of the search key.

Searching

An extension of binary search with p processors gives that one can find the rank of an element in

$$\log_{p+1}(n) = \frac{\log n}{\log(p+1)}$$

many parallel steps with p processors. (not work-optimal)

This requires a CREW PRAM model. For the EREW model searching cannot be done faster than $\mathcal{O}(\log n - \log p)$ with p processors even if there are p copies of the search key.

Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 21

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \leq y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 21

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \leq y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 21

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \le y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

Merging

Given two sorted sequences $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, compute the sorted squence $C = (c_1, \ldots, c_n)$.

Definition 21

Let $X = (x_1, \ldots, x_t)$ be a sequence. The rank $\mathrm{rank}(y : X)$ of y in X is

$$\mathrm{rank}(y : X) = |\{x \in X \mid x \leq y\}|$$

For a sequence $Y = (y_1, \ldots, y_s)$ we define $\mathrm{rank}(Y : X) := (r_1, \ldots, r_s)$ with $r_i = \mathrm{rank}(y_i : X)$.

Merging

We have already seen a merging-algorithm that runs in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Using the fast search algorithm we can improve this to a running time of $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n\log\log n)$.

Merging

We have already seen a merging-algorithm that runs in time $\mathcal{O}(\log n)$ and work $\mathcal{O}(n)$.

Using the fast search algorithm we can improve this to a running time of $\mathcal{O}(\log\log n)$ and work $\mathcal{O}(n\log\log n)$.

Merging

Input: $A = a_1, \ldots, a_n$; $B = b_1, \ldots, b_m$; $m \le n$

1. if $m < 4$ then rank elements of B, using the parallel search algorithm with p processors. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(n)$.
2. Concurrently rank elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_m$ in A using the parallel search algorithm with $p = \sqrt{n}$. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(\sqrt{m} \cdot \sqrt{n}) = \mathcal{O}(n)$

   $j(i) := \mathrm{rank}(b_{i\sqrt{m}} : A)$
3. Let $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$; and $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$.

   Recursively compute $\mathrm{rank}(B_i : A_i)$.
4. Let k be index not a multiple of $\sqrt{m}$. $i = \lceil \frac{k}{\sqrt{m}} \rceil$. Then $\mathrm{rank}(b_k : A) = j(i) + \mathrm{rank}(b_k : A_i)$.

Merging

Input: $A = a_1, \dots, a_n$; $B = b_1, \dots, b_m$; $m \le n$

1. if $m < 4$ then rank elements of B, using the parallel search algorithm with p processors. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(n)$.
2. Concurrently rank elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \dots, b_m$ in A using the parallel search algorithm with $p = \sqrt{n}$. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(\sqrt{m} \cdot \sqrt{n}) = \mathcal{O}(n)$

   $j(i) := \mathrm{rank}(b_{i\sqrt{m}} : A)$

3. Let $B_i = (b_{i\sqrt{m}+1}, \dots, b_{(i+1)\sqrt{m}-1})$; and $A_i = (a_{j(i)+1}, \dots, a_{j(i+1)})$.

   Recursively compute $\mathrm{rank}(B_i : A_i)$.

4. Let k be index not a multiple of $\sqrt{m}$. $i = \lceil \frac{k}{\sqrt{m}} \rceil$. Then $\mathrm{rank}(b_k : A) = j(i) + \mathrm{rank}(b_k : A_i)$.

Merging

Input: $A = a_1, \ldots, a_n$; $B = b_1, \ldots, b_m$; $m \le n$

1. if $m < 4$ then rank elements of B, using the parallel search algorithm with p processors. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(n)$.
2. Concurrently rank elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_m$ in A using the parallel search algorithm with $p = \sqrt{n}$. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(\sqrt{m} \cdot \sqrt{n}) = \mathcal{O}(n)$

   $$j(i) := \mathrm{rank}(b_{i\sqrt{m}} : A)$$

3. Let $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$; and $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$.

   Recursively compute $\mathrm{rank}(B_i : A_i)$.

4. Let k be index not a multiple of $\sqrt{m}$. $i = \lceil \frac{k}{\sqrt{m}} \rceil$. Then $\mathrm{rank}(b_k : A) = j(i) + \mathrm{rank}(b_k : A_i)$.

Merging

Input: $A = a_1, \ldots, a_n$; $B = b_1, \ldots, b_m$; $m \leq n$

1. if $m < 4$ then rank elements of B, using the parallel search algorithm with p processors. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(n)$.
2. Concurrently rank elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_m$ in A using the parallel search algorithm with $p = \sqrt{n}$. Time: $\mathcal{O}(1)$. Work: $\mathcal{O}(\sqrt{m} \cdot \sqrt{n}) = \mathcal{O}(n)$

   $j(i) := \mathrm{rank}(b_{i\sqrt{m}} : A)$
3. Let $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$; and $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$.

   Recursively compute $\mathrm{rank}(B_i : A_i)$.
4. Let k be index not a multiple of $\sqrt{m}$. $i = \lceil \frac{k}{\sqrt{m}} \rceil$. Then $\mathrm{rank}(b_k : A) = j(i) + \mathrm{rank}(b_k : A_i)$.

The algorithm can be made work-optimal by standard techniques.

proof on board...

Mergesort

Lemma 22

A straightforward parallelization of Mergesort can be implemented in time $\mathcal{O}(\log n \log\log n)$ and with work $\mathcal{O}(n \log n)$.

This assumes the CREW-PRAM model.

Mergesort

Lemma 22
A straightforward parallelization of Mergesort can be implemented in time $\mathcal{O}(\log n \log\log n)$ *and with work* $\mathcal{O}(n \log n)$*.*

This assumes the CREW-PRAM model.

Mergesort

Let $L[v]$ denote the (sorted) sublist of elements stored at the leaf nodes rooted at v.

We can view Mergesort as computing $L[v]$ for a complete binary tree where the leaf nodes correspond to nodes in the given array.

Since the merge-operations on one level of the complete binary tree can be performed in parallel we obtain time $\mathcal{O}(h \log\log n)$ and work $\mathcal{O}(hn)$, where $h = \mathcal{O}(\log n)$ is the height of the tree.

Mergesort

Let $L[v]$ denote the (sorted) sublist of elements stored at the leaf nodes rooted at v.

We can view Mergesort as computing $L[v]$ for a complete binary tree where the leaf nodes correspond to nodes in the given array.

Since the merge-operations on one level of the complete binary tree can be performed in parallel we obtain time $\mathcal{O}(h \log\log n)$ and work $\mathcal{O}(hn)$, where $h = \mathcal{O}(\log n)$ is the height of the tree.

Mergesort

Let $L[v]$ denote the (sorted) sublist of elements stored at the leaf nodes rooted at v.

We can view Mergesort as computing $L[v]$ for a complete binary tree where the leaf nodes correspond to nodes in the given array.

Since the merge-operations on one level of the complete binary tree can be performed in parallel we obtain time $\mathcal{O}(h \log \log n)$ and work $\mathcal{O}(hn)$, where $h = \mathcal{O}(\log n)$ is the height of the tree.

Pipelined Mergesort

We again compute $L[v]$ for every node in the complete binary tree.

After round s, $L_s[v]$ is an **approximation** of $L[v]$ that will be improved in future rounds.

For $s \geq 3\,\mathrm{height}(v)$, $L_s[v] = L[v]$.

Pipelined Mergesort

We again compute $L[v]$ for every node in the complete binary tree.

After round s, $L_s[v]$ is an **approximation** of $L[v]$ that will be improved in future rounds.

For $s \geq 3\,\mathrm{height}(v)$, $L_s[v] = L[v]$.

Pipelined Mergesort

We again compute $L[v]$ for every node in the complete binary tree.

After round s, $L_s[v]$ is an **approximation** of $L[v]$ that will be improved in future rounds.

For $s \geq 3\,\mathrm{height}(v)$, $L_s[v] = L[v]$.

Pipelined Mergesort

In every round, a node v sends $\mathrm{sample}(L_s[v])$ (an approximation of its current list) upwards, and receives approximations of the lists of its children.

It then computes a new approximation of its list.

A node is called active in round s if $s \le 3\,\mathrm{height}(v)$ (this means its list is not yet complete at the start of the round, i.e., $L_{s-1}[v] \ne L[v]$).

Pipelined Mergesort

In every round, a node v sends $\text{sample}(L_s[v])$ (an approximation of its current list) upwards, and receives approximations of the lists of its children.

It then computes a new approximation of its list.

A node is called active in round s if $s \le 3\,\text{height}(v)$ (this means its list is not yet complete at the start of the round, i.e., $L_{s-1}[v] \neq L[v]$).

Pipelined Mergesort

In every round, a node v sends $\text{sample}(L_s[v])$ (an approximation of its current list) upwards, and receives approximations of the lists of its children.

It then computes a new approximation of its list.

A node is called active in round s if $s \leq 3\,\text{height}(v)$ (this means its list is not yet complete at the start of the round, i.e., $L_{s-1}[v] \neq L[v]$).

Pipelined Mergesort

Algorithm 11 ColeSort()

initialize $L_0[v] = A_v$ for leaf nodes; $L_0[v] = \emptyset$ otw.
for $s \leftarrow 1$ **to** $3 \cdot \text{height}(T)$ **do**
  for all active nodes v **do**
    // u and w children of v
    $L'_s[u] \leftarrow \text{sample}(L_{s-1}[u])$
    $L'_s[w] \leftarrow \text{sample}(L_{s-1}[w])$
    $L_s[v] \leftarrow \text{merge}(L'_s[u], L'_s[w])$

$$\text{sample}(L_s[v]) = \begin{cases} \text{sample}_4(L_s[v]) & s \le 3\,\text{height}(v) \\ \text{sample}_2(L_s[v]) & s = 3\,\text{height}(v) + 1 \\ \text{sample}_1(L_s[v]) & s = 3\,\text{height}(v) + 2 \end{cases}$$

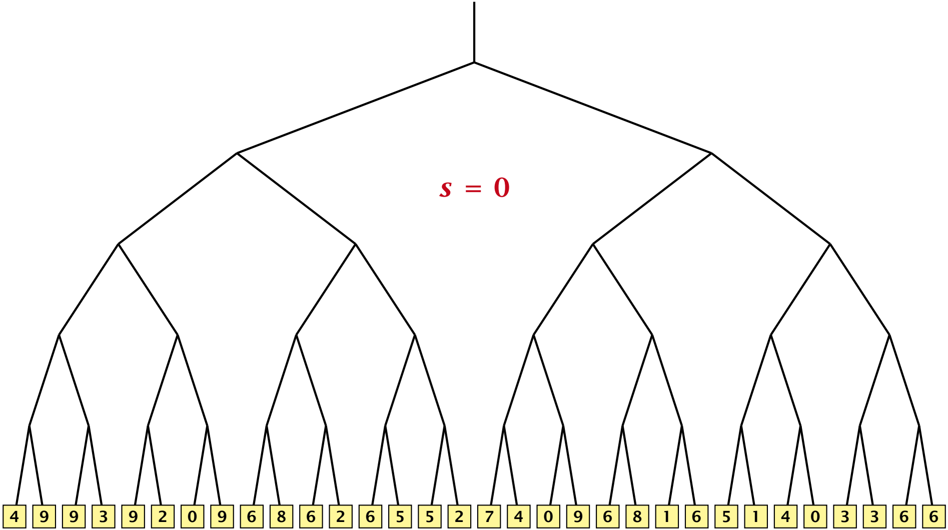
Pipelined Mergesort

Algorithm 11 ColeSort()

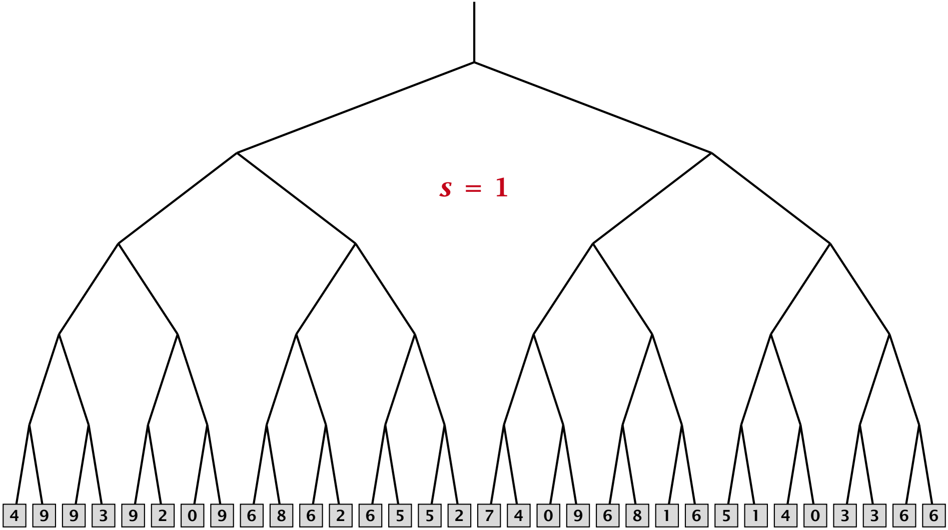
```
initialize L_0[v] = A_v for leaf nodes; L_0[v] = ∅ otw.
for s ← 1 to 3 · height(T) do
    for all active nodes v do
        // u and w children of v
        L'_s[u] ← sample(L_{s-1}[u])
        L'_s[w] ← sample(L_{s-1}[w])
        L_s[v] ← merge(L'_s[u], L'_s[w])
```

$$\text{sample}(L_s[v]) = \begin{cases} \text{sample}_4(L_s[v]) & s \leq 3\,\text{height}(v) \\ \text{sample}_2(L_s[v]) & s = 3\,\text{height}(v) + 1 \\ \text{sample}_1(L_s[v]) & s = 3\,\text{height}(v) + 2 \end{cases}$$

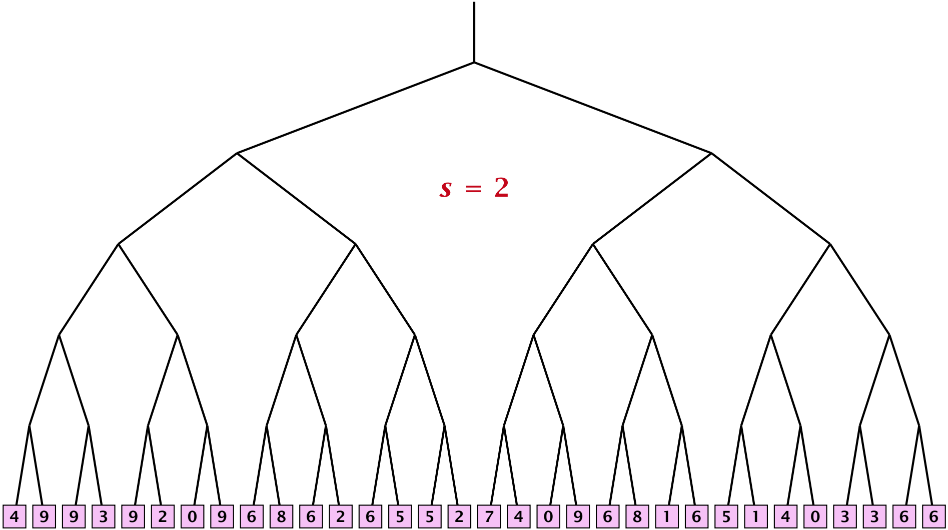
Colesort



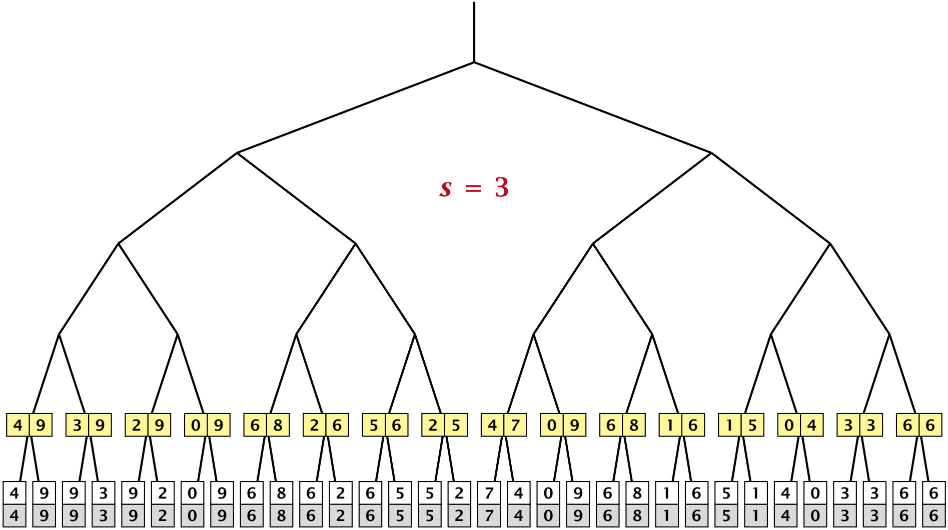
Colesort



Colesort

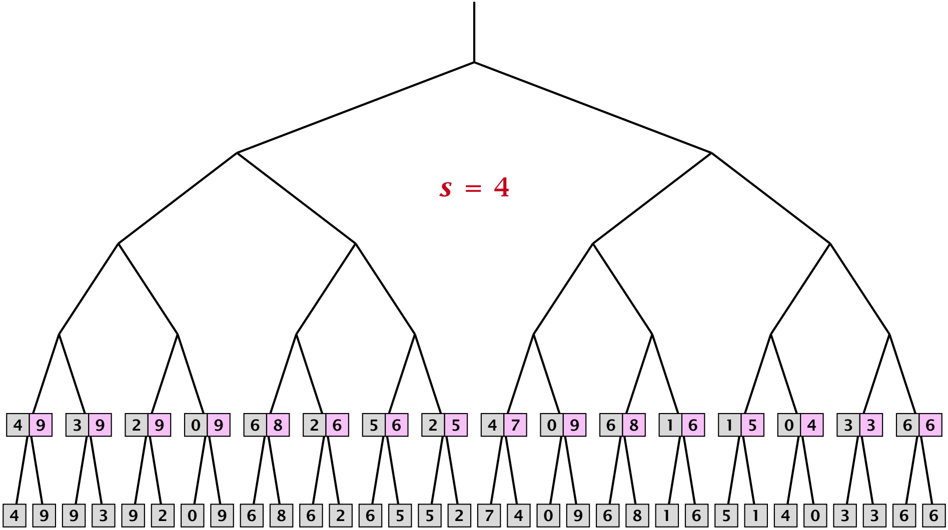


Colesort



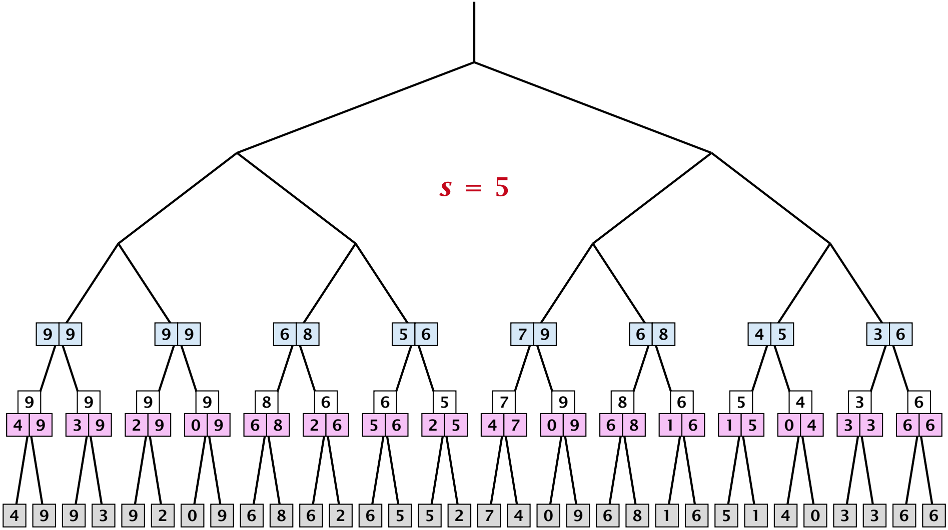
$s = 3$
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort



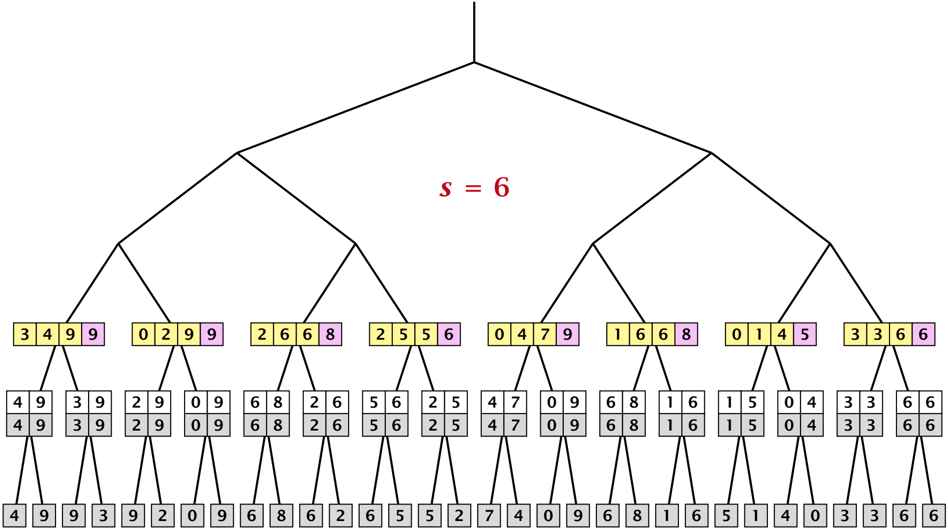
s = 4
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

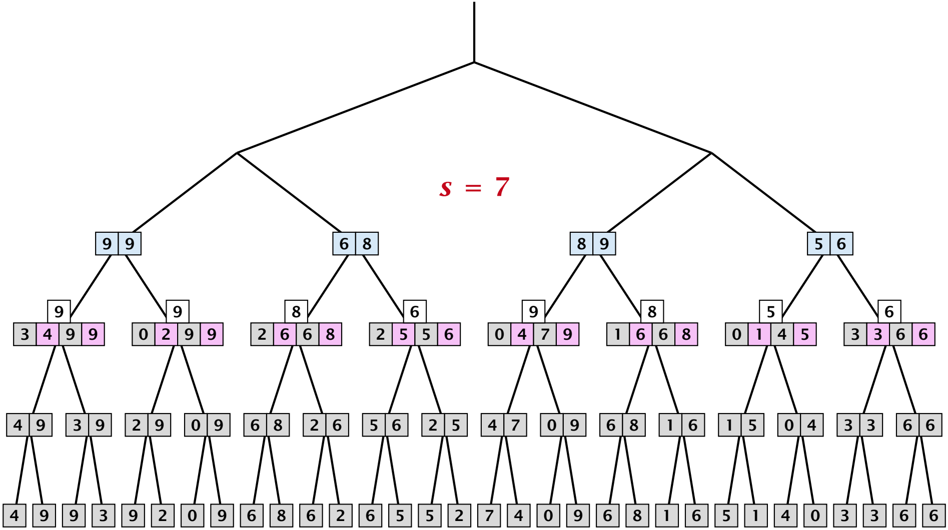


$s = 5$
9 9
9 9
6 8
5 6
7 9
6 8
4 5
3 6
9 9 9 9 8 6 6 5 7 9 8 6 5 4 3 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

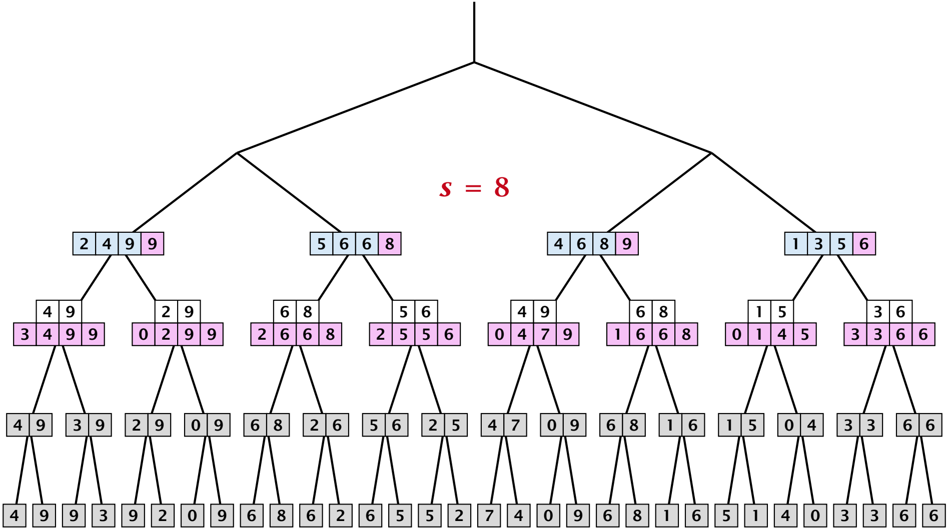


Colesort

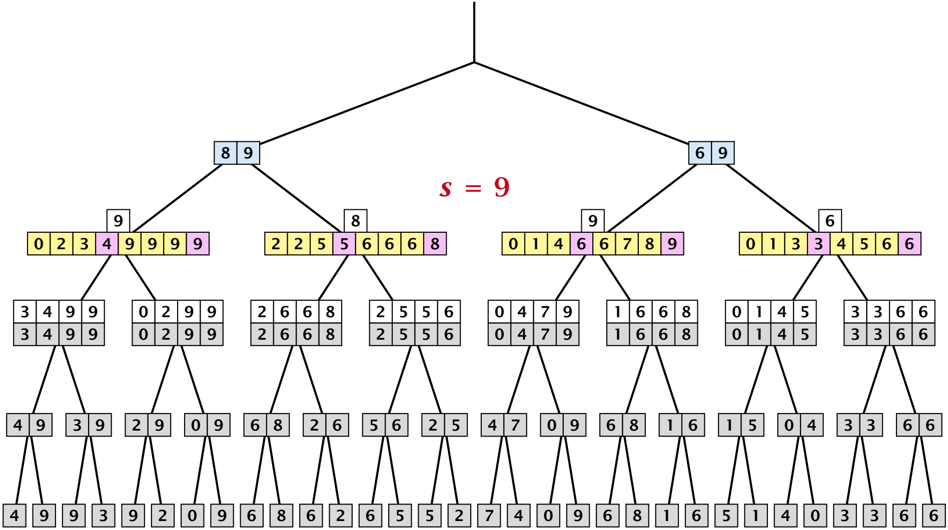


$s = 7$
9 9
6 8
8 9
5 6
9
9
8
6
9
8
5
6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

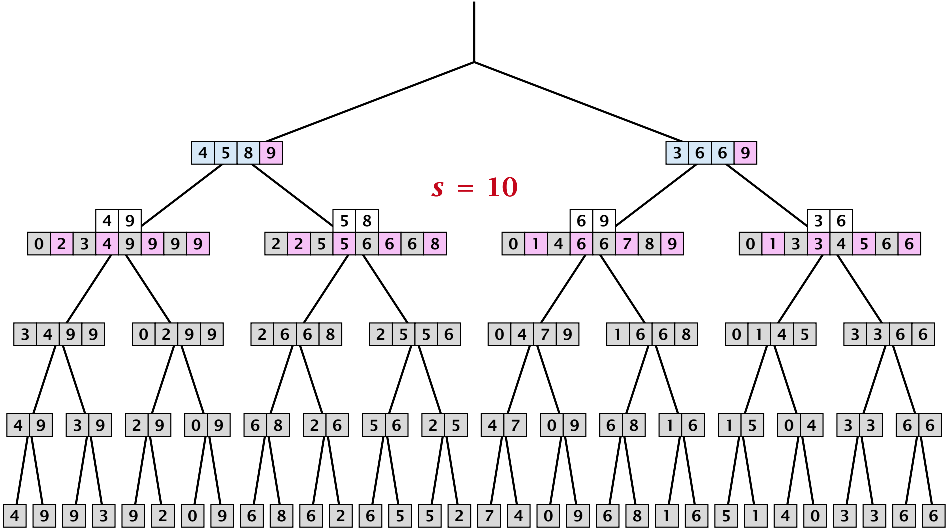


Colesort

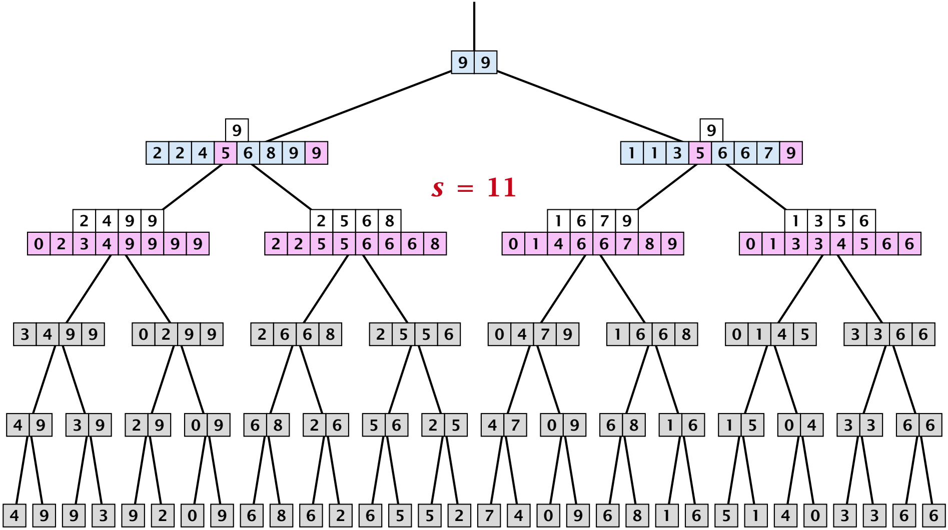


8 9
6 9
$s = 9$
9
8
9
6
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

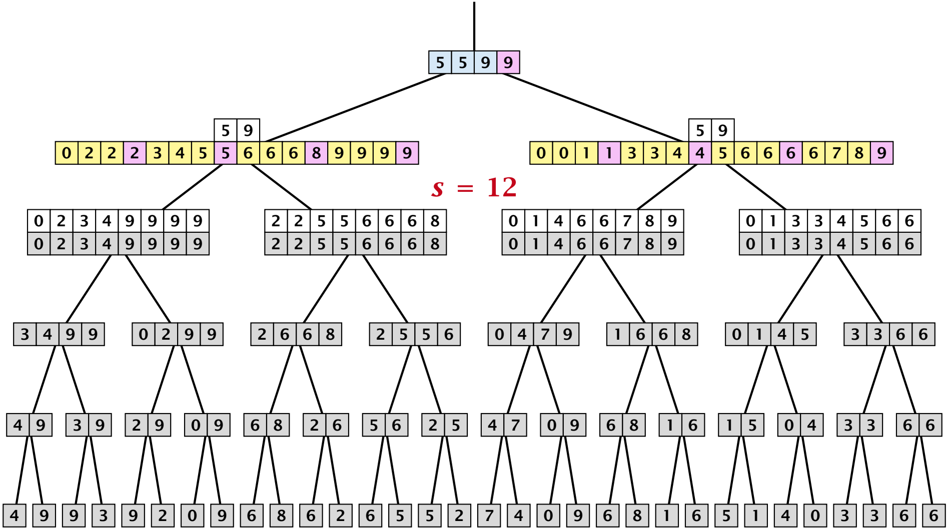


Colesort



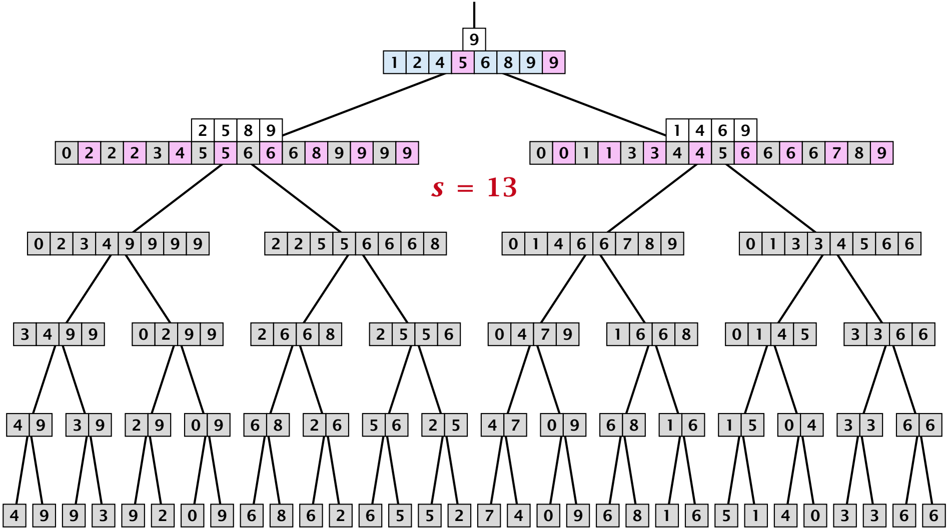
9 9
9
2 2 4 5 6 8 9 9
9
1 1 3 5 6 6 7 9
s = 11
2 4 9 9
0 2 3 4 9 9 9 9
2 5 6 8
2 2 5 5 6 6 6 8
1 6 7 9
0 1 4 6 6 7 8 9
1 3 5 6
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort



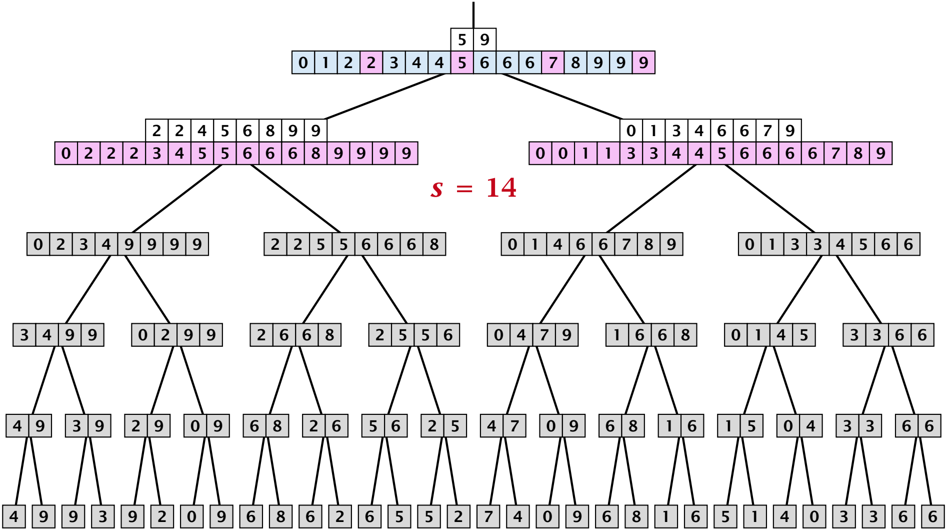
5 5 9 9
5 9
0 2 2 2 3 4 5 5 6 6 6 8 9 9 9 9
5 9
0 0 1 1 3 3 4 4 5 6 6 6 6 7 8 9
s = 12
0 2 3 4 9 9 9 9
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort



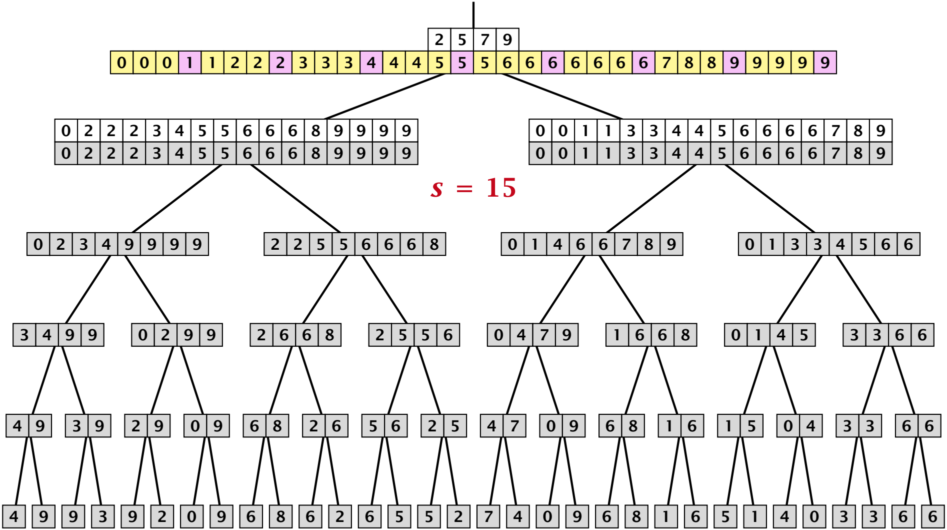
9
1 2 4 5 6 8 9 9
2 5 8 9
0 2 2 2 3 4 5 5 6 6 6 8 9 9 9 9
1 4 6 9
0 0 1 1 3 3 4 4 5 6 6 6 6 7 8 9
$s = 13$
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

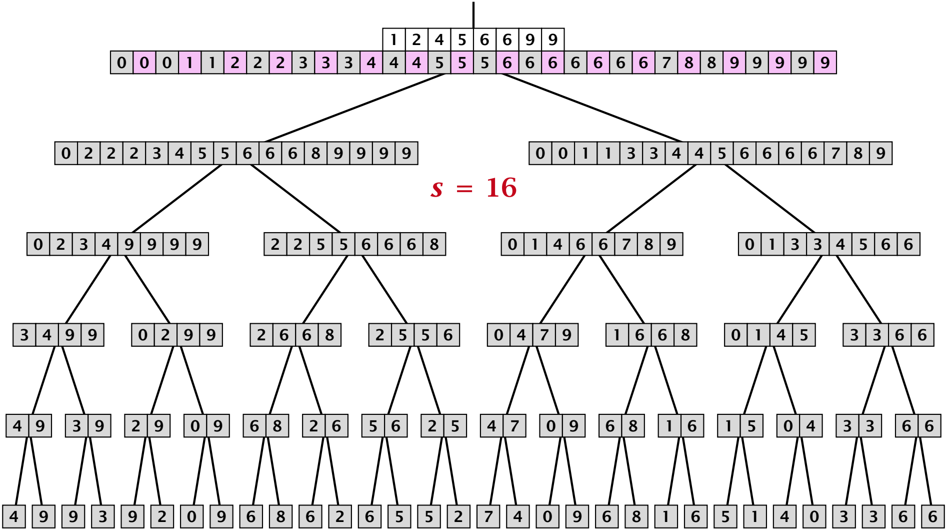


5 9
0 1 2 2 3 4 4 5 6 6 6 7 8 9 9 9
2 2 4 5 6 8 9 9
0 2 2 2 3 4 5 5 6 6 6 8 9 9 9 9
0 1 3 4 6 6 7 9
0 0 1 1 3 3 4 4 5 6 6 6 6 7 8 9
s = 14
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort

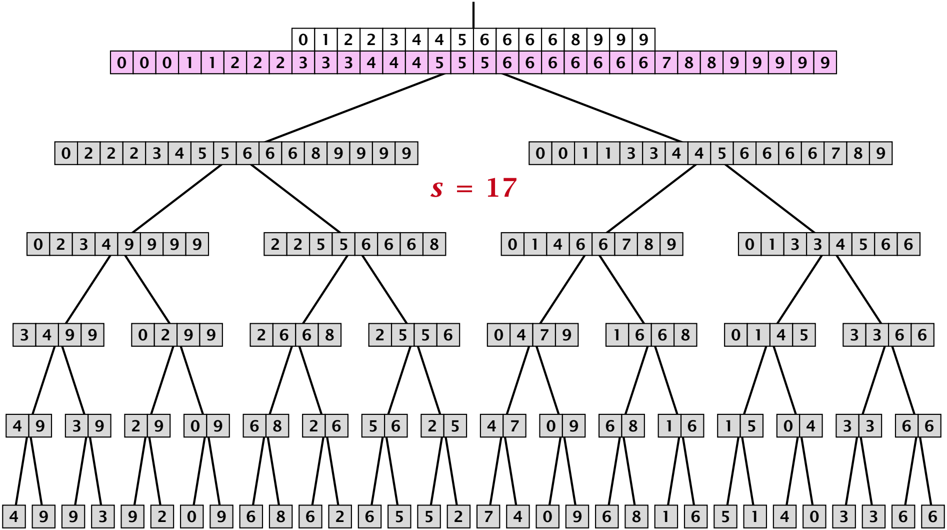


Colesort



1 2 4 5 6 6 9 9
0 0 0 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 6 6 6 6 7 8 8 9 9 9 9 9
0 2 2 2 3 4 5 5 6 6 6 8 9 9 9 9
0 0 1 1 3 3 4 4 5 6 6 6 6 7 8 9
$s = 16$
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Colesort



0 1 2 2 3 4 4 5 6 6 6 6 8 9 9 9
0 0 0 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 6 6 6 6 7 8 8 9 9 9 9 9
0 2 2 2 3 4 5 5 6 6 6 8 9 9 9 9
0 0 1 1 3 3 4 4 5 6 6 6 6 7 8 9
s = 17
0 2 3 4 9 9 9 9
2 2 5 5 6 6 6 8
0 1 4 6 6 7 8 9
0 1 3 3 4 5 6 6
3 4 9 9
0 2 9 9
2 6 6 8
2 5 5 6
0 4 7 9
1 6 6 8
0 1 4 5
3 3 6 6
4 9
3 9
2 9
0 9
6 8
2 6
5 6
2 5
4 7
0 9
6 8
1 6
1 5
0 4
3 3
6 6
4 9 9 3 9 2 0 9 6 8 6 2 6 5 5 2 7 4 0 9 6 8 1 6 5 1 4 0 3 3 6 6

Pipelined Mergesort

Lemma 23

After round $s = 3\,\mathrm{height}(v)$, *the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23
After round $s = 3\,\mathrm{height}(v)$, *the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23

After round $s = 3\,\mathrm{height}(v)$, *the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23
After round $s = 3\,\text{height}(v)$*, the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\text{sample}(L_{3h+2}[u]) = L[u]$ and $\text{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23
After round $s = 3\,\mathrm{height}(v)$*, the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23
After round $s = 3\,\mathrm{height}(v)$, the list $L_s[v]$ is complete.

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 23

After round $s = 3\,\mathrm{height}(v)$*, the list* $L_s[v]$ *is complete.*

Proof:

- clearly true for leaf nodes
- suppose it is true for all nodes up to height h;
- fix a node v on level $h+1$ with children u and w
- $L_{3h}[u]$ and $L_{3h}[w]$ are complete by induction hypothesis
- further $\mathrm{sample}(L_{3h+2}[u]) = L[u]$ and $\mathrm{sample}(L_{3h+2}[v]) = L[v]$
- hence in round $3h+3$ node v will merge the complete list of its children; after the round $L[v]$ will be complete

Pipelined Mergesort

Lemma 24
The number of elements in lists $L_s[v]$ for active nodes v is at most $\mathcal{O}(n)$.

proof on board...

Definition 25

A sequence X is a c-cover of a sequence Y if for any two consecutive elements α, β from $(-\infty, X, \infty)$ the set $|\{y_i \mid \alpha \le y_i \le \beta\}| \le c$.

Pipelined Mergesort

Lemma 26

$L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.

If $[a,b]$ fulfills $|[a,b] \cap (A \cup \{-\infty, \infty\})| = k$ we say $[a,b]$ intersects $(-\infty, A, +\infty)$ in k items.

Lemma 27

If $[a,b]$ with $a, b \in L'_s[v] \cup \{-\infty, \infty\}$ intersects $(-\infty, L'_s[v], \infty)$ in $k \geq 2$ items, then $[a,b]$ intersects $(-\infty, L'_{s+1}, \infty)$ in at most $2k$ items.

Pipelined Mergesort

Lemma 26
$L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.

If $[a,b]$ fulfills $|[a,b] \cap (A \cup \{-\infty,\infty\})| = k$ we say $[a,b]$ intersects $(-\infty, A, +\infty)$ in k items.

Lemma 27
If $[a,b]$ with $a,b \in L'_s[v] \cup \{-\infty,\infty\}$ intersects $(-\infty, L'_s[v], \infty)$ in $k \geq 2$ items, then $[a,b]$ intersects $(-\infty, L'_{s+1}, \infty)$ in at most $2k$ items.

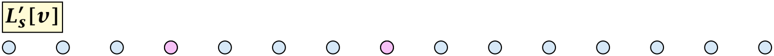
Pipelined Mergesort

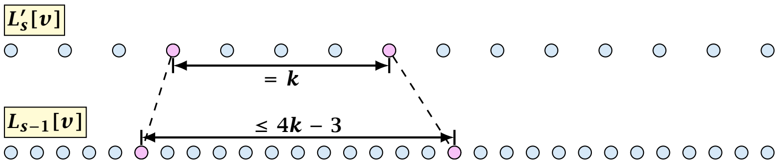
Lemma 26
$L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.

If $[a,b]$ fulfills $|[a,b] \cap (A \cup \{-\infty, \infty\})| = k$ we say $[a,b]$ intersects $(-\infty, A, +\infty)$ in k items.

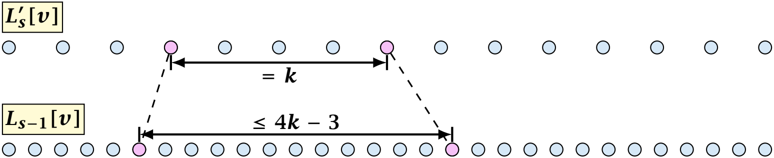
Lemma 27
If $[a,b]$ with $a, b \in L'_s[v] \cup \{-\infty, \infty\}$ intersects $(-\infty, L'_s[v], \infty)$ in $k \geq 2$ items, then $[a,b]$ intersects $(-\infty, L'_{s+1}, \infty)$ in at most $2k$ items.

$\boxed{L'_s[v]}$

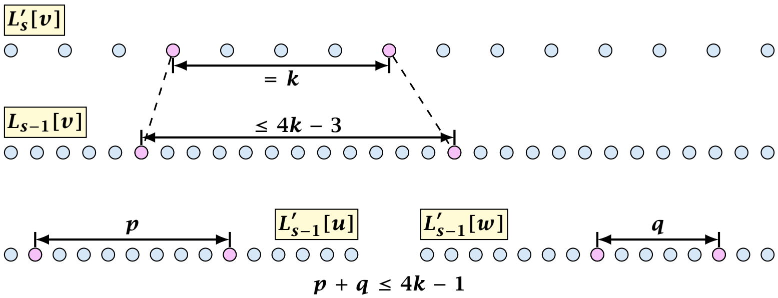

$L'_s[v]$



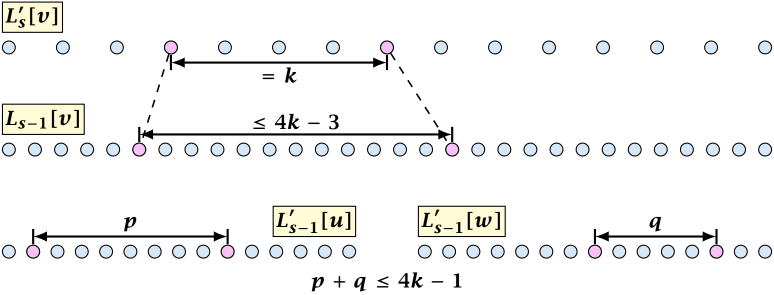
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$



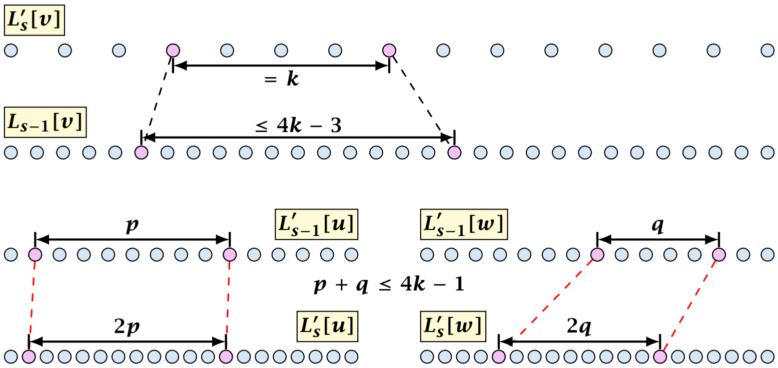
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$



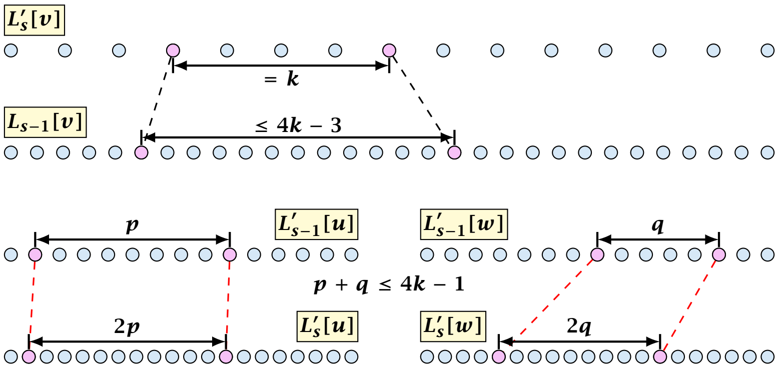
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\le 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \le 4k - 1$



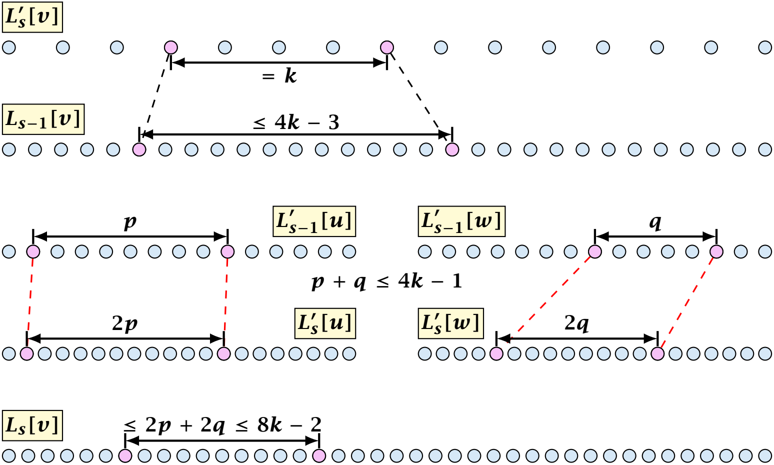
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$

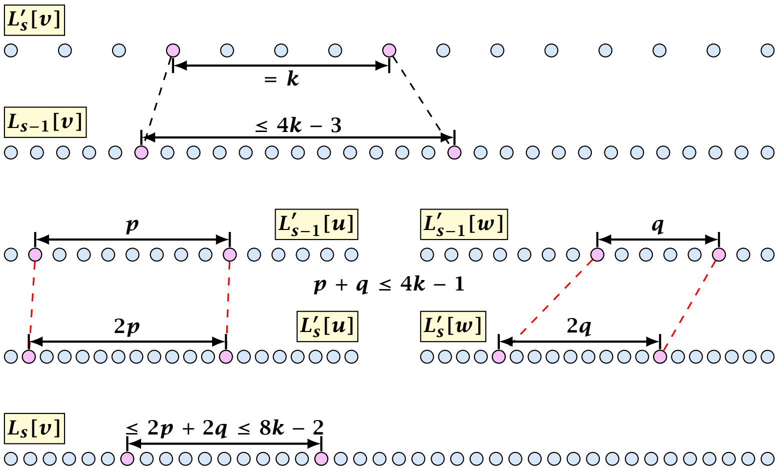


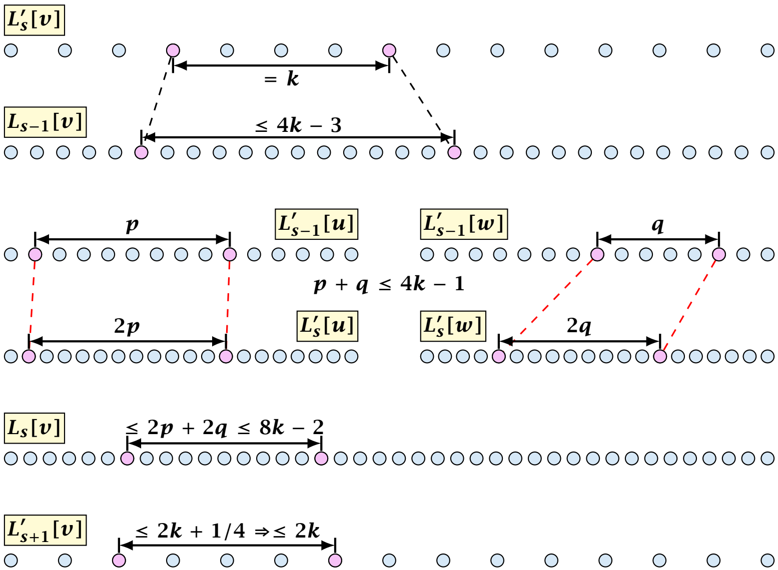
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$



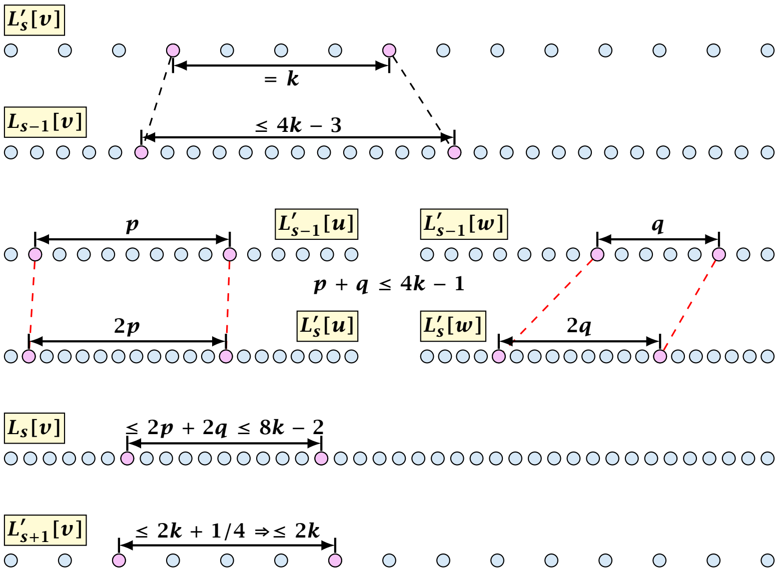
$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$


$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$
$L_s[v]$
$\leq 2p + 2q \leq 8k - 2$

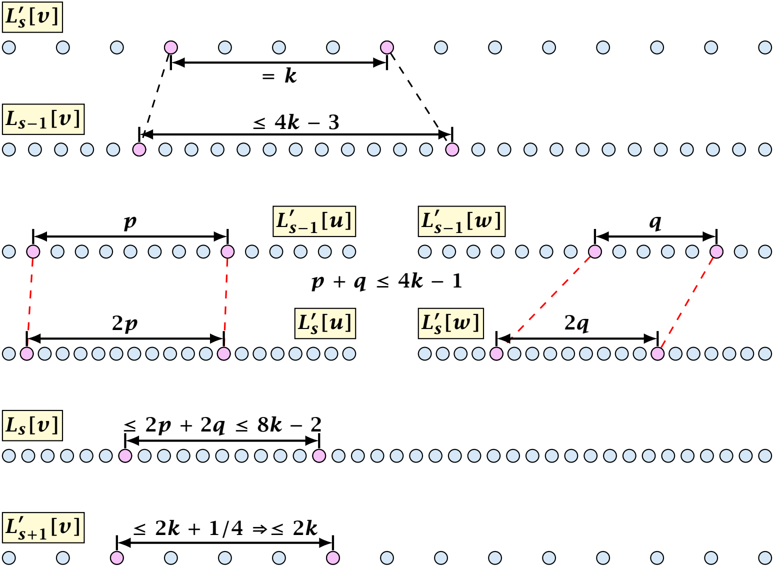

$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$
$L_s[v]$
$\leq 2p + 2q \leq 8k - 2$



$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$
$L_s[v]$
$\leq 2p + 2q \leq 8k - 2$
$L'_{s+1}[v]$
$\leq 2k + 1/4 \Rightarrow \leq 2k$



$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$
$L_s[v]$
$\leq 2p + 2q \leq 8k - 2$
$L'_{s+1}[v]$
$\leq 2k + 1/4 \Rightarrow \leq 2k$



$L'_s[v]$
$= k$
$L_{s-1}[v]$
$\leq 4k - 3$
p
$L'_{s-1}[u]$
$L'_{s-1}[w]$
q
$p + q \leq 4k - 1$
$2p$
$L'_s[u]$
$L'_s[w]$
$2q$
$L_s[v]$
$\leq 2p + 2q \leq 8k - 2$
$L'_{s+1}[v]$
$\leq 2k + 1/4 \Rightarrow \leq 2k$

Merging with a Cover

Lemma 28

Given two sorted sequences A and B. Let X be a c-cover of A and B for constant c, and let $\operatorname{rank}(X : A)$ and $\operatorname{rank}(X : B)$ be known.

We can merge A and B in time $\mathcal{O}(1)$ using $\mathcal{O}(|X|)$ operations.

Merging with a Cover

Lemma 29

Given two sorted sequences A *and* B. *Let* X *be a* c*-cover of* B *for constant* c, *and let* $\mathrm{rank}(A : X)$ *and* $\mathrm{rank}(X : B)$ *be known.*

We can compute $\mathrm{rank}(A : B)$ *using* $\mathcal{O}(|X| + |A|)$ *operations.*

Merging with a Cover

Lemma 30

Given two sorted sequences A and B. Let X be a c-cover of B for constant c, and let $\mathrm{rank}(A:X)$ and $\mathrm{rank}(X:B)$ be known.

We can compute $\mathrm{rank}(B:A)$ using $\mathcal{O}(|X| + |A|)$ operations.

Easy to do with concurrent read. Can also be done with exclusive read but non-trivial.

In order to do the merge in iteration $s+1$ in constant time we need to know

$$\operatorname{rank}(L_s[v] : L'_{s+1}[u]) \text{ and } \operatorname{rank}(L_s[v] : L'_{s+1}[w])$$

and we need to know that $L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$.

Lemma 31

$L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$.

Lemma 31
$L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$.

- $L_s[v] \supseteq L'_s[u], L'_s[w]$
- $L'_s[u]$ is 4-cover of $L'_{s+1}[u]$
- Hence, $L_s[v]$ is 4-cover of $L'_{s+1}[u]$ as adding more elements cannot destroy the cover-property.

Lemma 31

$L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$.

- $L_s[v] \supseteq L'_s[u], L'_s[w]$
- $L'_s[u]$ is 4-cover of $L'_{s+1}[u]$
- Hence, $L_s[v]$ is 4-cover of $L'_{s+1}[u]$ as adding more elements cannot destroy the cover-property.

Lemma 31

$L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$.

- $L_s[v] \supseteq L'_s[u], L'_s[w]$
- $L'_s[u]$ is 4-cover of $L'_{s+1}[u]$
- Hence, $L_s[v]$ is 4-cover of $L'_{s+1}[u]$ as adding more elements cannot destroy the cover-property.

Analysis

Lemma 32

Suppose we know for every internal node v with children u and w

- $\mathrm{rank}(L'_s[v] : L'_{s+1}[v])$
- $\mathrm{rank}(L'_s[u] : L'_s[w])$
- $\mathrm{rank}(L'_s[w] : L'_s[u])$

We can compute

- $\mathrm{rank}(L'_{s+1}[v] : L'_{s+2}[v])$
- $\mathrm{rank}(L'_{s+1}[u] : L'_{s+1}[w])$
- $\mathrm{rank}(L'_{s+1}[w] : L'_{s+1}[u])$

in constant time and $\mathcal{O}(|L_{s+1}[v]|)$ operations, where v is the parent of u and w.

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ **(4-cover)**
- $\text{rank}(L'_s[w] : L'_s[u])$
- $\text{rank}(L'_s[u] : L'_s[w])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ **(4-cover)**

Compute

- $\text{rank}(L'_{s+1}[w] : L'_s[u])$
- $\text{rank}(L'_{s+1}[u] : L'_s[w])$

Compute

- $\text{rank}(L'_{s+1}[w] : L'_{s+1}[u])$
- $\text{rank}(L'_{s+1}[u] : L'_{s+1}[w])$

ranks between siblings can be computed easily

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ **(4-cover)**
- $\text{rank}(L'_s[w] : L'_s[u])$
- $\text{rank}(L'_s[u] : L'_s[w])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ **(4-cover)**

Compute

- $\text{rank}(L'_{s+1}[w] : L'_s[u])$
- $\text{rank}(L'_{s+1}[u] : L'_s[w])$

Compute

- $\text{rank}(L'_{s+1}[w] : L'_{s+1}[u])$
- $\text{rank}(L'_{s+1}[u] : L'_{s+1}[w])$

ranks between siblings can be computed easily

Given

- $\operatorname{rank}(L'_s[u] : L'_{s+1}[u])$ **(4-cover)**
- $\operatorname{rank}(L'_s[w] : L'_s[u])$
- $\operatorname{rank}(L'_s[u] : L'_s[w])$
- $\operatorname{rank}(L'_s[w] : L'_{s+1}[w])$ **(4-cover)**

Compute

- $\operatorname{rank}(L'_{s+1}[w] : L'_s[u])$
- $\operatorname{rank}(L'_{s+1}[u] : L'_s[w])$

Compute

- $\operatorname{rank}(L'_{s+1}[w] : L'_{s+1}[u])$
- $\operatorname{rank}(L'_{s+1}[u] : L'_{s+1}[w])$

ranks between siblings can be computed easily

Given

- $\mathrm{rank}(L'_s[u] : L'_{s+1}[u])$ (**4-cover** → $\mathrm{rank}(L'_{s+1}[u] : L'_s[u])$)
- $\mathrm{rank}(L'_s[w] : L'_{s+1}[u])$
- $\mathrm{rank}(L'_s[u] : L'_{s+1}[w])$
- $\mathrm{rank}(L'_s[w] : L'_{s+1}[w])$ (**4-cover** → $\mathrm{rank}(L'_{s+1}[w] : L'_s[w])$)

Compute (recall that $L_s[v] = \mathrm{merge}(L'_s[u], L'_s[w])$)

- $\mathrm{rank}(L_s[v] : L'_{s+1}[u])$
- $\mathrm{rank}(L_s[v] : L'_{s+1}[w])$

Compute

- $\mathrm{rank}(L_s[v] : L_{s+1}[v])$ (by adding)
- $\mathrm{rank}(L'_{s+1}[v] : L'_{s+2}[v])$ (by sampling)

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ (**4-cover** → $\text{rank}(L'_{s+1}[u] : L'_s[u])$)
- $\text{rank}(L'_s[w] : L'_{s+1}[u])$
- $\text{rank}(L'_s[u] : L'_{s+1}[w])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ (**4-cover** → $\text{rank}(L'_{s+1}[w] : L'_s[w])$)

Compute (recall that $L_s[v] = \text{merge}(L'_s[u], L'_s[w])$)

- $\text{rank}(L_s[v] : L'_{s+1}[u])$
- $\text{rank}(L_s[v] : L'_{s+1}[w])$

Compute

- $\text{rank}(L_s[v] : L_{s+1}[v])$ (by adding)
- $\text{rank}(L'_{s+1}[v] : L'_{s+2}[v])$ (by sampling)

Given

- $\text{rank}(L'_s[u] : L'_{s+1}[u])$ (**4-cover** → $\text{rank}(L'_{s+1}[u] : L'_s[u])$)
- $\text{rank}(L'_s[w] : L'_{s+1}[u])$
- $\text{rank}(L'_s[u] : L'_{s+1}[w])$
- $\text{rank}(L'_s[w] : L'_{s+1}[w])$ (**4-cover** → $\text{rank}(L'_{s+1}[w] : L'_s[w])$)

Compute (recall that $L_s[v] = \text{merge}(L'_s[u], L'_s[w])$)

- $\text{rank}(L_s[v] : L'_{s+1}[u])$
- $\text{rank}(L_s[v] : L'_{s+1}[w])$

Compute

- $\text{rank}(L_s[v] : L_{s+1}[v])$ (by adding)
- $\text{rank}(L'_{s+1}[v] : L'_{s+2}[v])$ (by sampling)

Definition 33

A 0-1 sequence S is bitonic if it can be written as the concatenation of subsequences S_1 and S_2 such that either

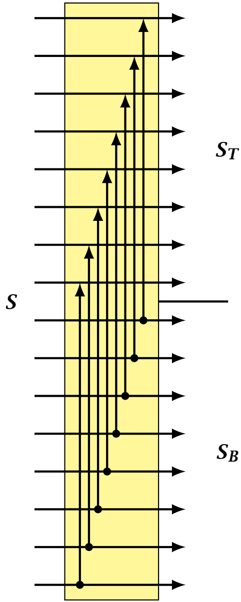
- S_1 is monotonically increasing and S_2 monotonically decreasing, or
- S_1 is monotonically decreasing and S_2 monotonically increasing.

Note, that this just defines bitonic 0-1 sequences. Bitonic sequences are defined differently.

Bitonic Merger

If we feed a bitonic 0-1 sequence S into the network on the right we obtain two bitonic sequences S_T and S_B s.t.

1. $S_B \leq S_T$ (element-wise)
2. S_B and S_T are bitonic



Proof:

- assume wlog. S more 1's than 0's.
- assume for contradiction two 0s at same comparator $(i, j = i + 2^d)$
  - everything 0 btw. i and j means we have more than 50% zeros (↯).
  - all 1s btw. i and j means we have less than 50% ones (↯).
  - 1 btw. i and j and elsewhere means S is not bitonic (↯).

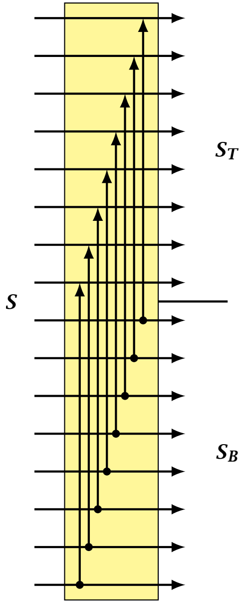
Bitonic Merger

If we feed a bitonic 0-1 sequence S into the network on the right we obtain two bitonic sequences S_T and S_B s.t.

1. $S_B \leq S_T$ (element-wise)
2. S_B and S_T are bitonic

Proof:

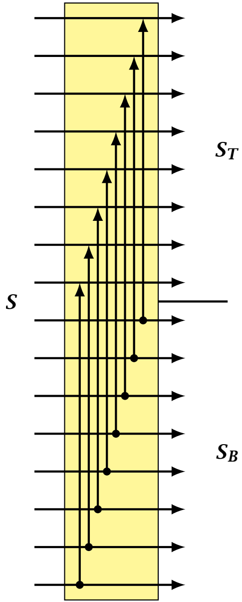
- assume wlog. S more 1's than 0's.
- assume for contradiction two 0s at same comparator $(i, j = i + 2^d)$
  - everything 0 btw i and j means we have more than 50% zeros (↯).
  - all 1s btw. i and j means we have less than 50% ones (↯).
  - 1 btw. i and j and elsewhere means S is not bitonic (↯).



Bitonic Merger

If we feed a bitonic 0-1 sequence S into the network on the right we obtain two bitonic sequences S_T and S_B s.t.

1. $S_B \leq S_T$ (element-wise)
2. S_B and S_T are bitonic



Proof:

- assume wlog. S more 1's than 0's.
- assume for contradiction two 0s at same comparator $(i, j = i + 2^d)$
  - everything 0 btw i and j means we have more than 50% zeros (⚡).
  - all 1s btw. i and j means we have less than 50% ones (⚡).
  - 1 btw. i and j and elsewhere means S is not bitonic (⚡).

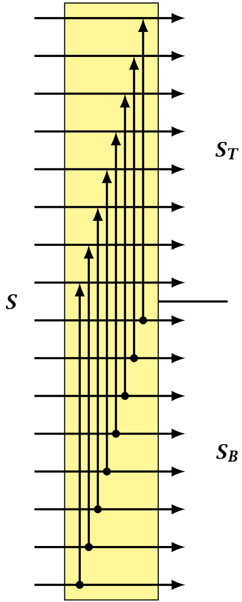
Bitonic Merger

If we feed a bitonic 0-1 sequence S into the network on the right we obtain two bitonic sequences S_T and S_B s.t.

1. $S_B \le S_T$ (element-wise)
2. S_B and S_T are bitonic

Proof:

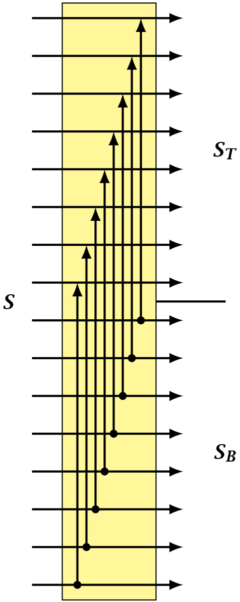
- assume wlog. S more 1's than 0's.
- assume for contradiction two 0s at same comparator $(i, j = i + 2^d)$
  - everything 0 btw i and j means we have more than 50% zeros (↯).
  - all 1s btw. i and j means we have less than 50% ones (↯).
  - 1 btw. i and j and elsewhere means S is not bitonic (↯).



Bitonic Merger

If we feed a bitonic 0-1 sequence S into the network on the right we obtain two bitonic sequences S_T and S_B s.t.

1. $S_B \le S_T$ (element-wise)
2. S_B and S_T are bitonic



Proof:

- assume wlog. S more 1's than 0's.
- assume for contradiction two 0s at same comparator $(i, j = i + 2^d)$
  - everything 0 btw i and j means we have more than 50% zeros (↯).
  - all 1s btw. i and j means we have less than 50% ones (↯).
  - 1 btw. i and j and elsewhere means S is not bitonic (↯).

Bitonic Merger

Bitonic Merger B_d

The bitonic merger B_d of dimension d is constructed by combining two bitonic mergers of dimension $d-1$.

If we feed a bitonic 0-1 sequence into this, the sequence will be sorted.

(actually, any bitonic sequence will be sorted, but we do not prove this)

Bitonic Merger

Bitonic Merger B_d
The bitonic merger B_d of dimension d is constructed by combining two bitonic mergers of dimension $d-1$.

If we feed a bitonic 0-1 sequence into this, the sequence will be sorted.

(actually, any bitonic sequence will be sorted, but we do not prove this)

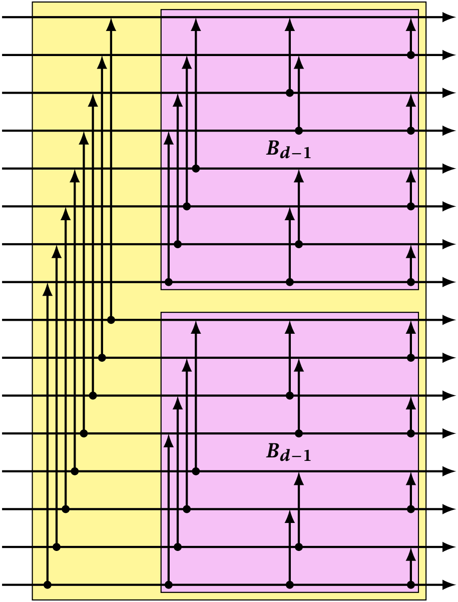
Bitonic Merger

Bitonic Merger B_d

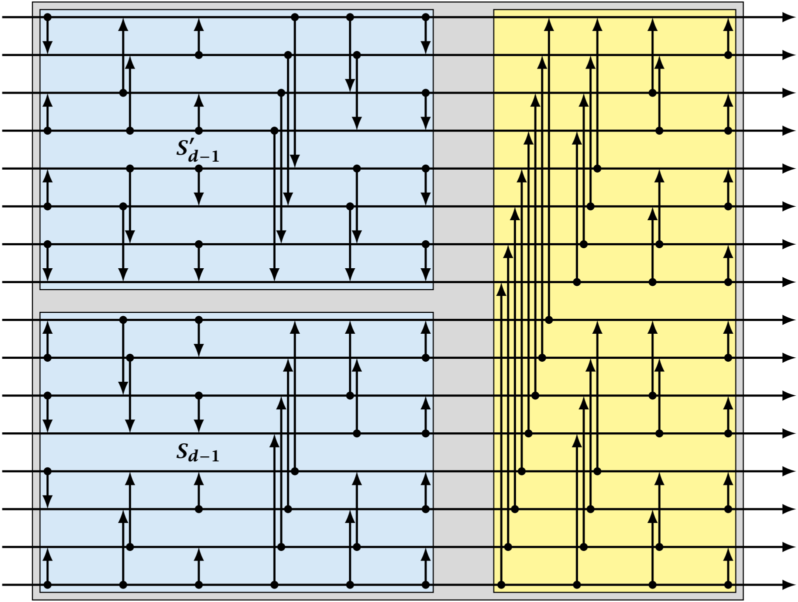
The bitonic merger B_d of dimension d is constructed by combining two bitonic mergers of dimension $d-1$.

If we feed a bitonic 0-1 sequence into this, the sequence will be sorted.

(actually, any bitonic sequence will be sorted, but we do not prove this)



Bitonic Sorter S_d



Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2$

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2$

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow$ $C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \mathcal{O}(\log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \mathcal{O}(\log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \mathcal{O}(\log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow C(n) = \mathcal{O}(n \log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow$
  $C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \Theta(\log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow$
  $C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \Theta(\log^2 n)$.

Bitonic Merger: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + n/2 \Rightarrow C(n) = \mathcal{O}(n \log n)$.
- depth: $D(n) = D(n/2) + 1 \Rightarrow D(d) = \mathcal{O}(\log n)$.

Bitonic Sorter: ($n = 2^d$)

- comparators: $C(n) = 2C(n/2) + \mathcal{O}(n \log n) \Rightarrow$
  $C(n) = \mathcal{O}(n \log^2 n)$.
- depth: $D(n) = D(n/2) + \log n \Rightarrow D(n) = \Theta(\log^2 n)$.

Odd-Even Merge

How to merge two sorted sequences?

$A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, n even.

Split into odd and even sequences:

$A_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, $A_{\text{even}} = (a_2, a_4, a_6, \ldots a_n)$

$B_{\text{odd}} = (b_1, b_3, b_5, \ldots, b_{n-1})$, $B_{\text{even}} = (b_2, b_4, b_6, \ldots, b_n)$

Let

$$X = \text{merge}(A_{\text{odd}}, B_{\text{odd}}) \text{ and } Y = \text{merge}(A_{\text{even}}, B_{\text{even}})$$

Then

$$S = (x_1, \min\{x_2, y_1\}, \max\{x_2, y_1\}, \min\{x_3, y_2\}, \ldots, y_n)$$

Odd-Even Merge

How to merge two sorted sequences?

$A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, n even.

Split into odd and even sequences:

$A_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, $A_{\text{even}} = (a_2, a_4, a_6, \ldots a_n)$

$B_{\text{odd}} = (b_1, b_3, b_5, \ldots, b_{n-1})$, $B_{\text{even}} = (b_2, b_4, b_6, \ldots, b_n)$

Let

$X = \text{merge}(A_{\text{odd}}, B_{\text{odd}})$ and $Y = \text{merge}(A_{\text{even}}, B_{\text{even}})$

Then

$$S = (x_1, \min\{x_2, y_1\}, \max\{x_2, y_1\}, \min\{x_3, y_2\}, \ldots, y_n)$$

Odd-Even Merge

How to merge two sorted sequences?

$A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, n even.

Split into odd and even sequences:

$A_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, $A_{\text{even}} = (a_2, a_4, a_6, \ldots a_n)$

$B_{\text{odd}} = (b_1, b_3, b_5, \ldots, b_{n-1})$, $B_{\text{even}} = (b_2, b_4, b_6, \ldots, b_n)$

Let

$$X = \text{merge}(A_{\text{odd}}, B_{\text{odd}}) \text{ and } Y = \text{merge}(A_{\text{even}}, B_{\text{even}})$$

Then

$$S = (x_1, \min\{x_2, y_1\}, \max\{x_2, y_1\}, \min\{x_3, y_2\}, \ldots, y_n)$$

Odd-Even Merge

How to merge two sorted sequences?

$A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$, n even.

Split into odd and even sequences:

$A_{\text{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$, $A_{\text{even}} = (a_2, a_4, a_6, \ldots a_n)$

$B_{\text{odd}} = (b_1, b_3, b_5, \ldots, b_{n-1})$, $B_{\text{even}} = (b_2, b_4, b_6, \ldots, b_n)$

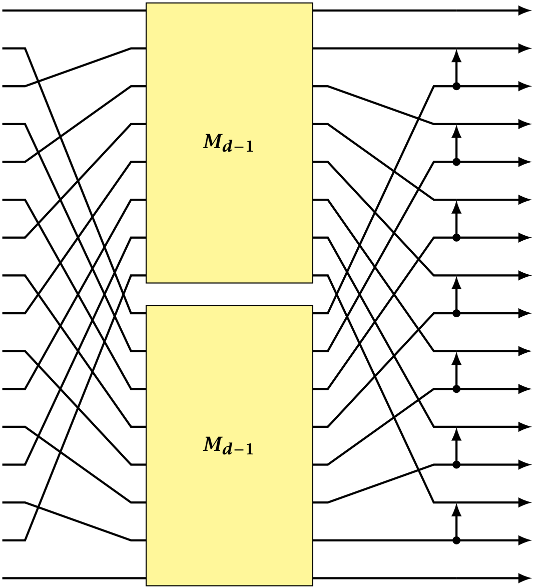
Let

$$X = \text{merge}(A_{\text{odd}}, B_{\text{odd}}) \text{ and } Y = \text{merge}(A_{\text{even}}, B_{\text{even}})$$

Then

$$S = (x_1, \min\{x_2, y_1\}, \max\{x_2, y_1\}, \min\{x_3, y_2\}, \ldots, y_n)$$

Odd-Even Merge



Theorem 34
There exists a sorting network with depth $\mathcal{O}(\log n)$ *and* $\mathcal{O}(n \log n)$ *comparators.*

Parallel Comparison Tree Model

A parallel comparison tree (with parallelism p) is a 3^p-ary tree.

- each internal node represents a set of p comparisons btw. p pairs (not necessarily distinct)
- a leaf v corresponds to a unique permutation that is valid for all the comparisons on the path from the root to v
- the number of parallel steps is the height of the tree

Parallel Comparison Tree Model

A parallel comparison tree (with parallelism p) is a 3^p-ary tree.

- each internal node represents a set of p comparisons btw. p pairs (not necessarily distinct)
- a leaf v corresponds to a unique permutation that is valid for all the comparisons on the path from the root to v
- the number of parallel steps is the height of the tree

Parallel Comparison Tree Model

A parallel comparison tree (with parallelism p) is a 3^p-ary tree.

- each internal node represents a set of p comparisons btw. p pairs (not necessarily distinct)
- a leaf v corresponds to a unique permutation that is valid for all the comparisons on the path from the root to v
- the number of parallel steps is the height of the tree

Comparison PRAM

A comparison PRAM is a PRAM where we can only compare the input elements;

- we cannot view them as strings
- we cannot do calculations on them

A lower bound for the comparison tree with parallelism p directly carries over to the comparison PRAM with p processors.

Comparison PRAM

A comparison PRAM is a PRAM where we can only compare the input elements;

- we cannot view them as strings
- we cannot do calculations on them

A lower bound for the comparison tree with parallelism p directly carries over to the comparison PRAM with p processors.

Comparison PRAM

A comparison PRAM is a PRAM where we can only compare the input elements;

- we cannot view them as strings
- we cannot do calculations on them

A lower bound for the comparison tree with parallelism p directly carries over to the comparison PRAM with p processors.

Comparison PRAM

A comparison PRAM is a PRAM where we can only compare the input elements;

- we cannot view them as strings
- we cannot do calculations on them

A lower bound for the comparison tree with parallelism p directly carries over to the comparison PRAM with p processors.

A Lower Bound for Searching

Theorem 35
Given a sorted table X of n elements and an element y. Searching for y in X requires $\Omega(\frac{\log n}{\log(p+1)})$ steps in the parallel comparsion tree with parallelism $p < n$.

A Lower Bound for Maximum

Theorem 36
A graph G with m edges and n vertices has an independent set on at least $\frac{n^2}{2m+n}$ vertices.

base case ($n = 1$)

- The only graph with one vertex has $m = 0$, and an independent set of size 1.

A Lower Bound for Maximum

Theorem 36
A graph G with m edges and n vertices has an independent set on at least $\frac{n^2}{2m+n}$ vertices.

base case ($n = 1$)

- The only graph with one vertex has $m = 0$, and an independent set of size 1.

A Lower Bound for Maximum

Theorem 36

A graph G with m edges and n vertices has an independent set on at least $\frac{n^2}{2m+n}$ vertices.

base case ($n = 1$)

- The only graph with one vertex has $m = 0$, and an independent set of size 1.

induction step ($1, \ldots, n \to n+1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \leq m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \geq \frac{n^2}{2m+n}$$

induction step ($1, \dots, n \to n+1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \le m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \ge \frac{n^2}{2m+n}$$

induction step ($1, \dots, n \to n + 1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \le m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \ge \frac{n^2}{2m+n}$$

induction step ($1, \dots, n \to n+1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \le m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \ge \frac{n^2}{2m+n}$$

induction step ($1, \dots, n \to n+1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \le m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \ge \frac{n^2}{2m+n}$$

induction step ($1, \dots, n \to n+1$)

- Let G be a graph with $n+1$ vertices, and v a node with minimum degree (d).
- Let G' be the graph after deleting v and its adjacent vertices in G.
- $n' = n - (d+1)$
- $m' \leq m - \frac{d}{2}(d+1)$ as we remove $d+1$ vertices, each with degree at least d
- In G' there is an independent set of size $((n')^2/(2m'+n'))$.
- By adding v we obtain an indepent set of size

$$1 + \frac{(n')^2}{2m'+n'} \geq \frac{n^2}{2m+n}$$

A Lower Bound for Maximum

Theorem 37

Computing the maximum of n elements in the comparison tree requires $\Omega(\log\log n)$ steps whenever the degree of parallelism is $p \le n$.

Theorem 38

Computing the maximum of n elements requires $\Omega(\log\log n)$ steps on the comparison PRAM with n processors.

A Lower Bound for Maximum

Theorem 37

Computing the maximum of n elements in the comparison tree requires $\Omega(\log\log n)$ steps whenever the degree of parallelism is $p \le n$.

Theorem 38

Computing the maximum of n elements requires $\Omega(\log\log n)$ steps on the comparison PRAM with n processors.

An adversary can specify the input such that at the end of the $(i+1)$-st step the maximum lies in a set C_{i+1} of size s_{i+1} such that

- no two elements of C_{i+1} have been compared
- $s_{i+1} \geq \frac{s_i^2}{2p+c_i}$

An adversary can specify the input such that at the end of the $(i+1)$-st step the maximum lies in a set C_{i+1} of size s_{i+1} such that

- no two elements of C_{i+1} have been compared
- $s_{i+1} \geq \frac{s_i^2}{2p+c_i}$

Theorem 39

The selection problem requires $\Omega(\log n/\log\log n)$ steps on a comparison PRAM.

not proven yet

A Lower Bound for Merging

The (k, s)-merging problem, asks to merge k pairs of subsequences $A^1, \dots, A^k$ and $B^1, \dots, B^k$ where we know that all elements in $A^i \cup B^i$ are smaller than elements in $A^j \cup B^j$ for $(i < j)$. Further $|A_i|, |B_i| \geq s$.

A Lower Bound for Merging

Lemma 40

Suppose we are given a parallel comparison tree with parallelism p to solve the (k, s) merging problem. After the first step an adversary can specify the input such that an arbitrary (k', s') merging problem has to be solved, where

$$k' = \frac{3}{4}\sqrt{pk}$$

$$s' = \frac{s}{4}\sqrt{\frac{k}{p}}$$

A Lower Bound for Merging

Partition $A^i s$ and $B^i s$ into blocks of length roughly s/ℓ; hence ℓ blocks.

Define an $\ell \times \ell$ binary matrix M^i, where M^i_{xy} is 0 iff the parallel step **did not** compare an element from A^i_x with an element from B^i_y.

The matrix has $2\ell - 1$ diagonals.

A Lower Bound for Merging

Partition A^is and B^is into blocks of length roughly s/ℓ; hence ℓ blocks.

Define an $\ell \times \ell$ binary matrix M^i, where M^i_{xy} is 0 iff the parallel step **did not** compare an element from A^i_x with an element from B^i_y.

The matrix has $2\ell - 1$ diagonals.

A Lower Bound for Merging

Partition A^is and B^is into blocks of length roughly s/ℓ; hence ℓ blocks.

Define an $\ell \times \ell$ binary matrix M^i, where M^i_{xy} is 0 iff the parallel step **did not** compare an element from A^i_x with an element from B^i_y.

The matrix has $2\ell - 1$ diagonals.

Choose for every i the diagonal of M^i that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \dots, \ell-1\}$ specifies the chosen diagonal) for which the entry in M^i is zero.

We can choose value s.t. elements for the j-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a (k', s') problem.

Choose for every i the diagonal of M^i that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \dots, \ell-1\}$ specifies the chosen diagonal) for which the entry in M^i is zero.

We can choose value s.t. elements for the j-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a (k', s') problem.

Choose for every i the diagonal of M^i that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \ldots, \ell-1\}$ specifies the chosen diagonal) for which the entry in M^i is zero.

We can choose value s.t. elements for the j-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a (k', s') problem.

Choose for every i the diagonal of M^i that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \dots, \ell-1\}$ specifies the chosen diagonal) for which the entry in M^i is zero.

We can choose value s.t. elements for the j-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a (k', s') problem.

Choose for every i the diagonal of M^i that has most zeros.

Pair all $A^i_{j+d_i}, B^i_j$, (where $d_i \in \{-(\ell-1), \dots, \ell-1\}$ specifies the chosen diagonal) for which the entry in M^i is zero.

We can choose value s.t. elements for the j-th pair along the diagonal are **all** smaller than for the $(j+1)$-th pair.

Hence, we get a (k', s') problem.

How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix M^i) hits at least

$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$

  zeroes.
- Choosing $\ell = \lceil 2\sqrt{p/k} \rceil$ gives

$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{4\sqrt{p/k}} = \frac{s}{4}\sqrt{\frac{k}{p}}$$

  where we assume $s \geq 6\sqrt{p/k}$.

How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix M^i) hits at least

$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$

zeroes.
- Choosing $\ell = \lceil 2\sqrt{p/k} \rceil$ gives

$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{4\sqrt{p/k}} = \frac{s}{4}\sqrt{\frac{k}{p}}$$

where we assume $s \geq 6\sqrt{p/k}$.

How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix M^i) hits at least
$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$
zeroes.
- Choosing $\ell = \lceil 2\sqrt{p/k} \rceil$ gives
$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{4\sqrt{p/k}} = \frac{s}{4}\sqrt{\frac{k}{p}}$$
where we assume $s \geq 6\sqrt{p/k}$.

How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix M^i) hits at least

$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$

  zeroes.
- Choosing $\ell = \lceil 2\sqrt{p/k} \rceil$ gives

$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{4\sqrt{p/k}} = \frac{s}{4}\sqrt{\frac{k}{p}}$$

  where we assume $s \geq 6\sqrt{p/k}$.

How many pairs do we have?

- there are $k\ell$ blocks in total
- there are $k \cdot \ell^2$ matrix entries in total
- there are at least $k \cdot \ell^2 - p$ zeros.
- choosing a random diagonal (same for every matrix M^i) hits at least
$$\frac{k\ell^2 - p}{2\ell - 1} \geq \frac{k\ell}{2} - \frac{p}{2\ell}$$
zeroes.
- Choosing $\ell = \lceil 2\sqrt{p/k} \rceil$ gives
$$k' \geq \frac{3}{4}\sqrt{pk} \text{ and } s' = \lfloor \frac{s}{\ell} \rfloor \geq \frac{s}{4\sqrt{p/k}} = \frac{s}{4}\sqrt{\frac{k}{p}}$$
where we assume $s \geq 6\sqrt{p/k}$.

Lemma 41

Let $T(k,s,p)$ *be the number of parallel steps required on a comparison tree to solve the* (k,s) *merging problem. Then*

$$T(k,p,s) \geq \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}}$$

provided that $p \geq 2ks$ *and* $p \leq ks^2/36$

Induction Step:

Assume that

$$T(k', s', p) \geq \frac{1}{4} \log \frac{\log \frac{p}{k'}}{\log \frac{p}{k's'}}$$

Induction Step:

Assume that

$$T(k',s',p) \geq \frac{1}{4}\log\frac{\log\frac{p}{k'}}{\log\frac{p}{k's'}}$$
$$\geq \frac{1}{4}\log\frac{\log\frac{4}{3}\sqrt{\frac{p}{k}}}{\log\frac{16}{3}\frac{p}{ks}}$$

Induction Step:

Assume that

$$
\begin{aligned}
T(k',s',p) &\geq \frac{1}{4}\log\frac{\log\frac{p}{k'}}{\log\frac{p}{k's'}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{4}{3}\sqrt{\frac{p}{k}}}{\log\frac{16}{3}\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\frac{1}{2}\log\frac{p}{k}}{7\log\frac{p}{ks}}
\end{aligned}
$$

Induction Step:

Assume that

$$
\begin{aligned}
T(k',s',p) &\geq \frac{1}{4}\log\frac{\log\frac{p}{k'}}{\log\frac{p}{k's'}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{4}{3}\sqrt{\frac{p}{k}}}{\log\frac{16}{3}\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\frac{1}{2}\log\frac{p}{k}}{7\log\frac{p}{ks}} \\
&\geq \frac{1}{4}\log\frac{\log\frac{p}{k}}{\log\frac{p}{ks}} - 1
\end{aligned}
$$

Induction Step:

Assume that

$$\begin{aligned} T(k', s', p) &\geq \frac{1}{4} \log \frac{\log \frac{p}{k'}}{\log \frac{p}{k's'}} \\ &\geq \frac{1}{4} \log \frac{\log \frac{4}{3}\sqrt{\frac{p}{k}}}{\log \frac{16}{3}\frac{p}{ks}} \\ &\geq \frac{1}{4} \log \frac{\frac{1}{2}\log \frac{p}{k}}{7\log \frac{p}{ks}} \\ &\geq \frac{1}{4} \log \frac{\log \frac{p}{k}}{\log \frac{p}{ks}} - 1 \end{aligned}$$

This gives the induction step.

Theorem 42

Merging requires at least $\Omega(\log \log n)$ time on a CRCW PRAM with n processors.

Simulations between PRAMs

Theorem 43

We can simulate a p-processor priority CRCW PRAM on a p-processor EREW PRAM with slowdown $\mathcal{O}(\log p)$.

Simulations between PRAMs

Theorem 44
We can simulate a p-processor priority CRCW PRAM on a $p \log p$-processor common CRCW PRAM with slowdown $\mathcal{O}(1)$.

Simulations between PRAMs

Theorem 45
We can simulate a p-processor priority CRCW PRAM on a p-processor common CRCW PRAM with slowdown $\mathcal{O}(\frac{\log p}{\log\log p})$.

Simulations between PRAMs

Theorem 46
We can simulate a p-processor priority CRCW PRAM on a p-processor arbitrary CRCW PRAM with slowdown $\mathcal{O}(\log\log p)$.

Lower Bounds for the CREW PRAM

Ideal PRAM:

- every processor has unbounded local memory
- in each step a processor reads a global variable
- then it does some (unbounded) computation on its local memory
- then it writes a global variable

Lower Bounds for the CREW PRAM

Ideal PRAM:

- every processor has unbounded local memory
- in each step a processor reads a global variable
- then it does some (unbounded) computation on its local memory
- then it writes a global variable

Lower Bounds for the CREW PRAM

Ideal PRAM:

- every processor has unbounded local memory
- in each step a processor reads a global variable
- then it does some (unbounded) computation on its local memory
- then it writes a global variable

Lower Bounds for the CREW PRAM

Ideal PRAM:

- every processor has unbounded local memory
- in each step a processor reads a global variable
- then it does some (unbounded) computation on its local memory
- then it writes a global variable

Lower Bounds for the CREW PRAM

Definition 47
An input index i affects a memory location M at time t on some input I if the content of M at time t differs between inputs I and $I(i)$ (i-th bit flipped).

$$L(M,t,I) = \{i \mid i \text{ affects } M \text{ at time } t \text{ on input } I\}$$

Lower Bounds for the CREW PRAM

Definition 47
An input index i affects a memory location M at time t on some input I if the content of M at time t differs between inputs I and $I(i)$ (i-th bit flipped).

$L(M,t,I) = \{i \mid i \text{ affects } M \text{ at time } t \text{ on input } I\}$

Lower Bounds for the CREW PRAM

Definition 48
An input index i affects a processor P at time t on some input I if the state of P at time t differs between inputs I and $I(i)$ (i-th bit flipped).

$K(P, t, I) = \{i \mid i \text{ affects } P \text{ at time } t \text{ on input } I\}$

Lower Bounds for the CREW PRAM

Definition 48
An input index i affects a processor P at time t on some input I if the state of P at time t differs between inputs I and $I(i)$ (i-th bit flipped).

$K(P, t, I) = \{i \mid i \text{ affects } P \text{ at time } t \text{ on input } I\}$

Lower Bounds for the CREW PRAM

Lemma 49

If $i \in K(P, t, I)$ *with* $t > 1$ *then either*

- $i \in K(P, t-1, I)$, *or*
- P *reads a global memory location* M *on input* I *at time* t, *and* $i \in L(M, t-1, I)$.

Lower Bounds for the CREW PRAM

Lemma 50

If $i \in L(M, t, I)$ with $t > 1$ then either

- *A processor writes into M at time t on input I and $i \in K(P, t, I)$, or*
- *No processor writes into M at time t on input I and*
  - *either $i \in L(M, t-1, I)$*
  - *or a processor P writes into M at time t on input $I(i)$.*

Let $k_0 = 0, \ell_0 = 1$ and define

$$k_{t+1} = k_t + \ell_t \text{ and } \ell_{t+1} = 3k_t + 4\ell_t$$

Lemma 51

$|K(P,t,I)| \leq k_t$ and $|L(M,t,I)| \leq \ell_t$ for any $t \geq 0$

Let $k_0 = 0, \ell_0 = 1$ and define

$$k_{t+1} = k_t + \ell_t \text{ and } \ell_{t+1} = 3k_t + 4\ell_t$$

Lemma 51

$|K(P,t,I)| \leq k_t$ and $|L(M,t,I)| \leq \ell_t$ for any $t \geq 0$

base case ($t = 0$):

- No index can influence the local memory/state of a processor before the first step (hence $|K(P,0,I)| = k_0 = 0$).
- Initially every index in the input affects exactly one memory location. Hence $|L(M,0,I)| = 1 = \ell_0$.

base case ($t = 0$):

- No index can influence the local memory/state of a processor before the first step (hence $|K(P, 0, I)| = k_0 = 0$).
- Initially every index in the input affects exactly one memory location. Hence $|L(M, 0, I)| = 1 = \ell_0$.

induction step ($t \to t+1$):

$K(P, t+1, I) \subseteq K(P, t, I) \cup L(M, t, I)$, where M is the location read by P in step $t+1$.

induction step ($t \to t+1$):

$K(P,t+1,I) \subseteq K(P,t,I) \cup L(M,t,I)$, where M is the location read by P in step $t+1$.

Hence,

$$|K(P,t+1,I)|$$

induction step ($t \to t+1$):

$K(P, t+1, I) \subseteq K(P, t, I) \cup L(M, t, I)$, where M is the location read by P in step $t+1$.

Hence,

$$|K(P, t+1, I)| \leq |K(P, t, I)| + |L(M, t, I)|$$

induction step ($t \to t+1$):

$K(P, t+1, I) \subseteq K(P, t, I) \cup L(M, t, I)$, where M is the location read by P in step $t+1$.

Hence,

$$\begin{aligned} |K(P, t+1, I)| &\le |K(P, t, I)| + |L(M, t, I)| \\ &\le k_t + \ell_t \end{aligned}$$

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

Case 1:

A processor P writes into location M at time $t+1$ on input I.

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

Case 1:
A processor P writes into location M at time $t+1$ on input I.

Then,

$$|L(M, t+1, I)|$$

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

Case 1:
A processor P writes into location M at time $t+1$ on input I.

Then,

$$|L(M, t+1, I)| \leq |K(P, t+1, I)|$$

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

Case 1:
A processor P writes into location M at time $t+1$ on input I.

Then,

$$\begin{aligned}|L(M,t+1,I)| &\le |K(P,t+1,I)| \\ &\le k_t + \ell_t\end{aligned}$$

induction step ($t \to t+1$):

For the bound on $|L(M, t+1, I)|$ we have two cases.

Case 1:
A processor P writes into location M at time $t+1$ on input I.

Then,

$$
\begin{aligned}
|L(M,t+1,I)| &\le |K(P,t+1,I)| \\
&\le k_t + \ell_t \\
&\le 3k_t + 4\ell_t = \ell_{t+1}
\end{aligned}
$$

Case 2:

No processor P writes into location M at time $t+1$ on input I.

Case 2:
No processor P writes into location M at time $t+1$ on input I.

An index i affects M at time $t+1$ iff i affects M at time t or some processor P writes into M at $t+1$ on $I(i)$.

Case 2:
No processor P writes into location M at time $t+1$ on input I.

An index i affects M at time $t+1$ iff i affects M at time t or some processor P writes into M at $t+1$ on $I(i)$.

$$L(M,t+1,I) \subseteq L(M,t,I) \cup Y(M,t+1,I)$$

Case 2:
No processor P writes into location M at time $t+1$ on input I.

An index i affects M at time $t+1$ iff i affects M at time t or some processor P writes into M at $t+1$ on $I(i)$.

$L(M, t+1, I) \subseteq L(M, t, I) \cup Y(M, t+1, I)$

$Y(M, t+1, I)$ is the set of indices u_j that cause some processor P_{w_j} to write into M at time $t+1$ on input I.

$Y(M, t+1, I)$ is the set of indices u_j that cause some processor P_{w_j} to write into M at time $t+1$ on input I.

$Y(M, t+1, I)$ is the set of indices u_j that cause some processor P_{w_j} to write into M at time $t+1$ on input I.

Fact:

For all pairs u_s, u_t with $P_{w_s} \neq P_{w_t}$ either
$u_s \in K(P_{w_t}, t+1, I(u_t))$ or $u_t \in K(P_{w_s}, t+1, I(u_s))$.

$Y(M, t+1, I)$ is the set of indices u_j that cause some processor P_{w_j} to write into M at time $t+1$ on input I.

Fact:
For all pairs u_s, u_t with $P_{w_s} \neq P_{w_t}$ either
$u_s \in K(P_{w_t}, t+1, I(u_t))$ or $u_t \in K(P_{w_s}, t+1, I(u_s))$.

Otherwise, P_{w_t} and P_{w_s} would both write into M at the same time on input $I(u_s)(u_t)$.

Let $U = \{u_1, \ldots, u_r\}$ denote all indices that cause some processor to write into M.

Let $U = \{u_1, \dots, u_r\}$ denote all indices that cause some processor to write into M.

Let $V = \{(I(u_1), P_{w_1}), \dots\}$.

Let $U = \{u_1, \ldots, u_r\}$ denote all indices that cause some processor to write into M.

Let $V = \{(I(u_1), P_{w_1}), \ldots\}$.

We set up a bipartite graph between U and V, such that $(u_i, (I(u_j), P_{w_j})) \in E$ if u_i affects P_{w_j} at time $t+1$ on input $I(u_j)$.

Let $U = \{u_1, \dots, u_r\}$ denote all indices that cause some processor to write into M.

Let $V = \{(I(u_1), P_{w_1}), \dots\}$.

We set up a bipartite graph between U and V, such that $(u_i, (I(u_j), P_{w_j})) \in E$ if u_i affects P_{w_j} at time $t + 1$ on input $I(u_j)$.

Each vertex $(I(u_j), P_{w_j})$ has degree at most k_{t+1} as this is an upper bound on indices that can influence a processor P_{w_j}.

Let $U = \{u_1, \dots, u_r\}$ denote all indices that cause some processor to write into M.

Let $V = \{(I(u_1), P_{w_1}), \dots\}$.

We set up a bipartite graph between U and V, such that $(u_i, (I(u_j), P_{w_j})) \in E$ if u_i affects P_{w_j} at time $t+1$ on input $I(u_j)$.

Each vertex $(I(u_j), P_{w_j})$ has degree at most k_{t+1} as this is an upper bound on indices that can influence a processor P_{w_j}.

Hence, $|E| \le r \cdot k_{t+1}$.

For an index u_j there can be at most k_{t+1} indices u_i with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r - k_{t+1})$ pairs u_i, u_j with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r - k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

For an index u_j there can be at most k_{t+1} indices u_i with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r - k_{t+1})$ pairs u_i, u_j with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r - k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

For an index u_j there can be at most k_{t+1} indices u_i with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r - k_{t+1})$ pairs u_i, u_j with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r - k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

For an index u_j there can be at most k_{t+1} indices u_i with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r - k_{t+1})$ pairs u_i, u_j with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r - k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

For an index u_j there can be at most k_{t+1} indices u_i with $P_{w_i} = P_{w_j}$.

Hence, there must be at least $\frac{1}{2}r(r - k_{t+1})$ pairs u_i, u_j with $P_{w_i} \neq P_{w_j}$.

Each pair introduces at least one edge.

Hence,

$$|E| \geq \frac{1}{2}r(r - k_{t+1})$$

This gives $r \leq 3k_{t+1} \leq 3k_t + 3\ell_t$

Recall that $L(M, t+1, i) \subseteq L(M, t, i) \cup Y(M, t+1, I)$

$|L(M, t+1, i)| \leq 3k_t + 4\ell_t$

Recall that $L(M, t+1, i) \subseteq L(M, t, i) \cup Y(M, t+1, I)$

$|L(M, t+1, i)| \leq 3k_t + 4\ell_t$

Recall that $L(M, t+1, i) \subseteq L(M, t, i) \cup Y(M, t+1, I)$

$|L(M, t+1, i)| \leq 3k_t + 4\ell_t$

$$\begin{pmatrix} k_{t+1} \\ \ell_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} k_t \\ \ell_t \end{pmatrix} \qquad \begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Eigenvalues:

$$\lambda_1 = \frac{1}{2}(5 + \sqrt{21}) \text{ and } \lambda_2 = \frac{1}{2}(5 - \sqrt{21})$$

Eigenvectors:

$$v_1 = \begin{pmatrix} 1 \\ -(1 - \lambda_1) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ -(1 - \lambda_2) \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_{t+1} \\ \ell_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} k_t \\ \ell_t \end{pmatrix} \qquad \begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Eigenvalues:

$$\lambda_1 = \frac{1}{2}(5 + \sqrt{21}) \text{ and } \lambda_2 = \frac{1}{2}(5 - \sqrt{21})$$

Eigenvectors:

$$v_1 = \begin{pmatrix} 1 \\ -(1 - \lambda_1) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ -(1 - \lambda_2) \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_{t+1} \\ \ell_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} k_t \\ \ell_t \end{pmatrix} \qquad \begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Eigenvalues:

$$\lambda_1 = \frac{1}{2}(5 + \sqrt{21}) \text{ and } \lambda_2 = \frac{1}{2}(5 - \sqrt{21})$$

Eigenvectors:

$$v_1 = \begin{pmatrix} 1 \\ -(1 - \lambda_1) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ -(1 - \lambda_2) \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_{t+1} \\ \ell_{t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} k_t \\ \ell_t \end{pmatrix} \qquad \begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Eigenvalues:

$$\lambda_1 = \frac{1}{2}(5 + \sqrt{21}) \text{ and } \lambda_2 = \frac{1}{2}(5 - \sqrt{21})$$

Eigenvectors:

$$v_1 = \begin{pmatrix} 1 \\ -(1 - \lambda_1) \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ -(1 - \lambda_2) \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{21}}(v_1 - v_2)$$

$$\begin{pmatrix} k_t \\ \ell_t \end{pmatrix} = \frac{1}{\sqrt{21}}\left(\lambda_1^t v_1 - \lambda_2^t v_2\right)$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{21}}(v_1 - v_2)$$

$$\begin{pmatrix} k_t \\ \ell_t \end{pmatrix} = \frac{1}{\sqrt{21}}\left(\lambda_1^t v_1 - \lambda_2^t v_2\right)$$

$$v_1 = \begin{pmatrix} 1 \\ \frac{3}{2} + \frac{1}{2}\sqrt{21} \end{pmatrix} \text{ and } v_2 = \begin{pmatrix} 1 \\ \frac{3}{2} - \frac{1}{2}\sqrt{21} \end{pmatrix}$$

$$\begin{pmatrix} k_0 \\ \ell_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{21}}(v_1 - v_2)$$

$$\begin{pmatrix} k_t \\ \ell_t \end{pmatrix} = \frac{1}{\sqrt{21}}\left(\lambda_1^t v_1 - \lambda_2^t v_2\right)$$

Solving the recurrence gives

$$k_t = \frac{\lambda_1^t}{\sqrt{21}} - \frac{\lambda_2^t}{\sqrt{21}}$$

$$\ell_t = \frac{3+\sqrt{21}}{2\sqrt{21}}\lambda_1^t + \frac{-3+\sqrt{21}}{2\sqrt{21}}\lambda_2^t$$

with $\lambda_1 = \frac{1}{2}(5+\sqrt{21})$ and $\lambda_2 = \frac{1}{2}(5-\sqrt{21})$.

Theorem 52

The following problems require logarithmic time on a CREW PRAM.

- *Sorting a sequence of* $x_1, \ldots, x_n$ *with* $x_i \in \{0, 1\}$
- *Computing the maximum of* n *inputs*
- *Computing the sum* $x_1 + \cdots + x_n$ *with* $x_i \in \{0, 1\}$

A Lower Bound for the EREW PRAM

Definition 53 (Zero Counting Problem)

Given a monotone binary sequence $x_1, x_2, \ldots, x_n$ determine the index i such that $x_i = 0$ and $x_{i+1} = 1$.

We show that this problem requires $\Omega(\log n - \log p)$ steps on a p-processor EREW PRAM.

A Lower Bound for the EREW PRAM

Definition 53 (Zero Counting Problem)

Given a monotone binary sequence $x_1, x_2, \dots, x_n$ determine the index i such that $x_i = 0$ and $x_{i+1} = 1$.

We show that this problem requires $\Omega(\log n - \log p)$ steps on a p-processor EREW PRAM.

Let I_i be the input with i zeros folled by $n - i$ ones.

Index i affects processor P at time t if the state in step t is differs between I_{i-1} and I_i.

Index i affects location M at time t if the content of M after step t differs between inputs I_{i-1} and I_i.

Let I_i be the input with i zeros folled by $n - i$ ones.

Index i affects processor P at time t if the state in step t is differs between I_{i-1} and I_i.

Index i affects location M at time t if the content of M after step t differs between inputs I_{i-1} and I_i.

Let I_i be the input with i zeros folled by $n - i$ ones.

Index i affects processor P at time t if the state in step t is differs between I_{i-1} and I_i.

Index i affects location M at time t if the content of M after step t differs between inputs I_{i-1} and I_i.

Lemma 54

If $i \in K(P, t)$ *then either*

- $i \in K(P, t-1)$, *or*
- P *reads some location* M *on input* I_i *(and, hence, also on* I_{i-1}*) at step* t *and* $i \in L(M, t-1)$

Lemma 55

If $i \in L(M,t)$ *then either*

- $i \in L(M,t-1)$, *or*
- *Some processor* P *writes* M *at step* t *on input* I_i *and* $i \in K(P,t)$.
- *Some processor* P *writes* M *at step* t *on input* I_{i-1} *and* $i \in K(P,t)$.

Define

$$C(t) = \sum_P |K(P,t)| + \sum_M \max\{0, |L(M,t)| - 1\}$$

$C(T) \geq n$, $C(0) = 0$

Claim:
$C(t) \leq 6C(t-1) + 3|P|$

This gives $C(T) \leq \frac{6^T - 1}{5} 3|P|$ and hence $T = \Omega(\log n - \log |P|)$.

Define

$$C(t) = \sum_P |K(P,t)| + \sum_M \max\{0, |L(M,t)| - 1\}$$

$C(T) \geq n$, $C(0) = 0$

Claim:

$C(t) \leq 6C(t-1) + 3|P|$

This gives $C(T) \leq \frac{6^T - 1}{5} 3|P|$ and hence $T = \Omega(\log n - \log |P|)$.

Define

$$C(t) = \sum_P |K(P,t)| + \sum_M \max\{0, |L(M,t)| - 1\}$$

$C(T) \geq n$, $C(0) = 0$

Claim:
$C(t) \leq 6C(t-1) + 3|P|$

This gives $C(T) \leq \frac{6^T - 1}{5} 3|P|$ and hence $T = \Omega(\log n - \log |P|)$.

Define

$$C(t) = \sum_P |K(P,t)| + \sum_M \max\{0, |L(M,t)| - 1\}$$

$C(T) \geq n$, $C(0) = 0$

Claim:
$C(t) \leq 6C(t-1) + 3|P|$

This gives $C(T) \leq \frac{6^T - 1}{5} 3|P|$ and hence $T = \Omega(\log n - \log |P|)$.

For an index i to newly appear in $L(M,t)$ some processor must write into M on either input I_i or I_{i-1}.

Hence, any index in $K(P,t)$ can at most generate two new indices in $L(M,t)$.

This means that the number of new indices in any set $L(M,t)$ (over all M) is at most

$$2\sum_{P}|K(P,t)|$$

For an index i to newly appear in $L(M,t)$ some processor must write into M on either input I_i or I_{i-1}.

Hence, any index in $K(P,t)$ can at most generate two new indices in $L(M,t)$.

This means that the number of new indices in any set $L(M,t)$ (over all M) is at most

$$2\sum_P |K(P,t)|$$

For an index i to newly appear in $L(M,t)$ some processor must write into M on either input I_i or I_{i-1}.

Hence, any index in $K(P,t)$ can at most generate two new indices in $L(M,t)$.

This means that the number of new indices in any set $L(M,t)$ (over all M) is at most

$$2\sum_P |K(P,t)|$$

Hence,

$$\sum_M |L(M,t)| \le \sum_M |L(M,t-1)| + 2\sum_P |K(P,t)|$$

We can assume wlog. that $L(M,t-1) \subseteq L(M,t)$. Then

$$\sum_M \max\{0, |L(M,t)| - 1\} \le \sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)|$$

Hence,

$$\sum_M |L(M,t)| \le \sum_M |L(M,t-1)| + 2\sum_P |K(P,t)|$$

We can assume wlog. that $L(M,t-1) \subseteq L(M,t)$. Then

$$\sum_M \max\{0, |L(M,t)| - 1\} \le \sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)|$$

Hence,

$$\sum_M |L(M,t)| \le \sum_M |L(M,t-1)| + 2\sum_P |K(P,t)|$$

We can assume wlog. that $L(M,t-1) \subseteq L(M,t)$. Then

$$\sum_M \max\{0, |L(M,t)| - 1\} \le \sum_M \max\{0, |L(M,t-1)| - 1\} + 2\sum_P |K(P,t)|$$

For an index i to newly appear in $K(P,t)$, P must read a memory location M with $i \in L(M,t)$ on input I_i (and also on input I_{i-1}).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step t on input I_i, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs I_i a processor can read at most $|K(P,t-1)|+1$ different memory locations (why?).

For an index i to newly appear in $K(P,t)$, P must read a memory location M with $i \in L(M,t)$ on input I_i (and also on input I_{i-1}).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step t on input I_i, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs I_i a processor can read at most $|K(P,t-1)|+1$ different memory locations (why?).

For an index i to newly appear in $K(P,t)$, P must read a memory location M with $i \in L(M,t)$ on input I_i (and also on input I_{i-1}).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step t on input I_i, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs I_i a processor can read at most $|K(P,t-1)| + 1$ different memory locations (why?).

For an index i to newly appear in $K(P,t)$, P must read a memory location M with $i \in L(M,t)$ on input I_i (and also on input I_{i-1}).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step t on input I_i, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs I_i a processor can read at most $|K(P,t-1)| + 1$ different memory locations (why?).

For an index i to newly appear in $K(P,t)$, P must read a memory location M with $i \in L(M,t)$ on input I_i (and also on input I_{i-1}).

Since we are in the EREW model at most one processor can do so in every step.

Let $J(i,t)$ be memory locations read in step t on input I_i, and let $J_t = \bigcup_i J(i,t)$.

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Over all inputs I_i a processor can read at most $|K(P,t-1)|+1$ different memory locations (why?).

Hence,

$$\sum_P |K(P,t)|$$

Hence,

$$\sum_P |K(P,t)| \le \sum_P |K(P,t-1)| + \sum_{M \in J_t} |L(M,t-1)|$$

Hence,

$$\begin{aligned}\sum_P |K(P,t)| &\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\ &\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)| - 1) + J_t\end{aligned}$$

Hence,

$$
\begin{aligned}
\sum_P |K(P,t)| &\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\
&\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)| - 1) + J_t \\
&\le 2\sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)| - 1) + |P|
\end{aligned}
$$

Hence,

$$\begin{aligned}
\sum_P |K(P,t)| &\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\
&\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + J_t \\
&\leq 2\sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + |P| \\
&\leq 2\sum_P |K(P,t-1)| + \sum_M \max\{0, |L(M,t-1)|-1\} + |P|
\end{aligned}$$

Hence,

$$\begin{aligned}
\sum_P |K(P,t)| &\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\
&\le \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + J_t \\
&\le 2\sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + |P| \\
&\le 2\sum_P |K(P,t-1)| + \sum_M \max\{0, |L(M,t-1)|-1\} + |P|
\end{aligned}$$

Hence,

$$\begin{aligned}
\sum_P |K(P,t)| &\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} |L(M,t-1)| \\
&\leq \sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + J_t \\
&\leq 2\sum_P |K(P,t-1)| + \sum_{M\in J_t} (|L(M,t-1)|-1) + |P| \\
&\leq 2\sum_P |K(P,t-1)| + \sum_M \max\{0,|L(M,t-1)|-1\} + |P|
\end{aligned}$$

Recall

$$\sum_M \max\{0,|L(M,t)|-1\} \leq \sum_M \max\{0,|L(M,t-1)|-1\} + 2\sum_P |K(P,t)|$$

This gives

$$
\begin{aligned}
&\sum_P K(P,t) + \sum_M \max\{0, |L(M,t)| - 1\} \\
&\qquad \le 4\sum_M \max\{0, |L(M,t-1)| - 1\} + 6\sum_P |K(P,t-1)| + 3|P|
\end{aligned}
$$

Hence,

$$C(t) \le 6C(t-1) + 3|P|$$

This gives

$$\sum_P K(P,t) + \sum_M \max\{0, |L(M,t)| - 1\}$$
$$\leq 4\sum_M \max\{0, |L(M,t-1)| - 1\} + 6\sum_P |K(P,t-1)| + 3|P|$$

Hence,

$$C(t) \leq 6C(t-1) + 3|P|$$

Lower Bounds for CRCW PRAMS

Theorem 56
Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function. f can be computed in $\mathcal{O}(1)$ time on a common CRCW PRAM with $\leq n2^n$ processors.

Can we obtain non-constant lower bounds if we restrict the number of processors to be polynomial?

Boolean Circuits

- nodes are either AND, OR, or NOT gates or are special INPUT/OUTPUT nodes
- AND and OR gates have unbounded fan-in (indegree) and ounbounded fan-out (outdegree)
- NOT gates have unbounded fan-out
- INPUT nodes have indegree zero; OUTPUT nodes have outdegree zero
- size is the number of edges
- depth is the longest path from an input to an output

Theorem 57

Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function with n inputs and $m \le n$ outputs, and circuit C computes f with depth $D(n)$ and size $S(n)$. Then f can be computed by a common CRCW PRAM in $\mathcal{O}(D(n))$ time using $S(n)$ processors.

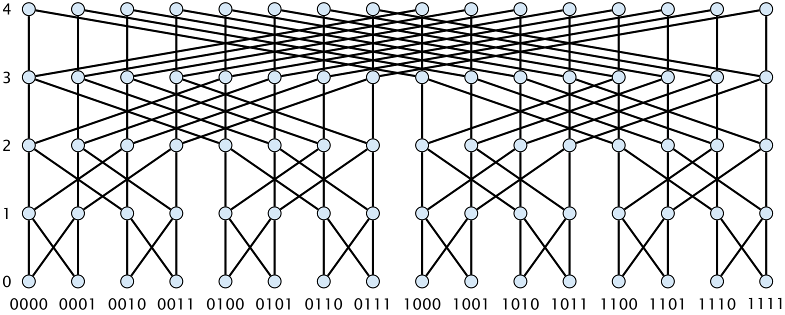
Given a family $\{C_n\}$ of circuits we may not be able to compute the corresponding family of functions on a CRCW PRAM.

Definition 58

A family $\{C_n\}$ of circuits is logspace uniform if there exists a deterministic Turing machine M s.t

- M runs in logarithmic space.
- For all n, M outputs C_n on input 1^n.

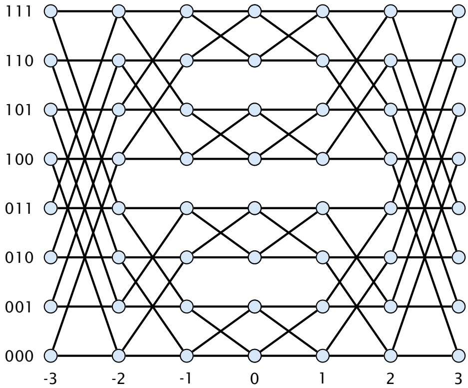
Bufferfly Network BF(d)



- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [2]^d, \ell \in [d+1]\}$, where $\bar{x} = x_0 x_1 \ldots x_{d-1}$ is a bit-string of length d
- edge set $E = \{\{(\ell, \bar{x}), (\ell+1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$

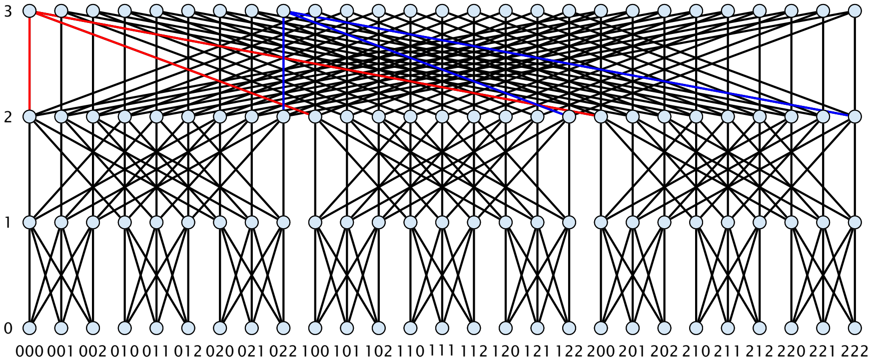
Sometimes the first and last level are identified.

Beneš Network



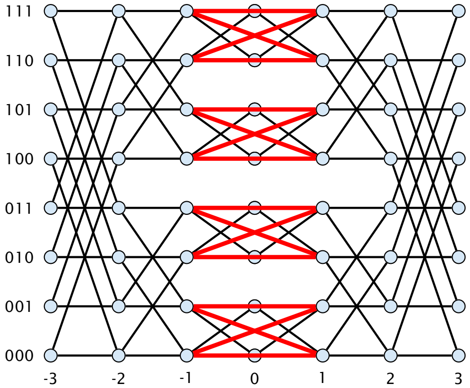
- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [2]^d, \ell \in \{-d, \ldots, d\}\}$
- edge set
  $E = \{\{(\ell, \bar{x}), (\ell + 1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$
  $\quad \cup \{\{(-\ell, \bar{x}), (\ell - 1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [2]^d, x'_i = x_i \text{ for } i \neq \ell\}$

n-ary Bufferfly Network BF(n, d)



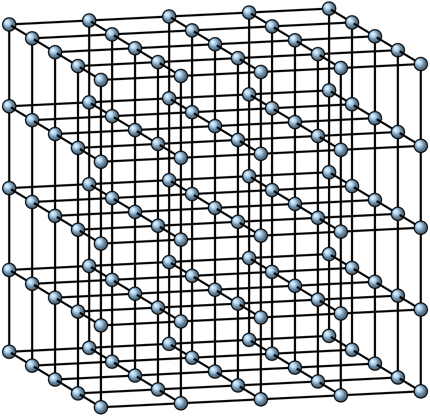
- node set $V = \{(\ell, \bar{x}) \mid \bar{x} \in [n]^d, \ell \in [d+1]\}$, where $\bar{x} = x_0 x_1 \dots x_{d-1}$ is a bit-string of length d
- edge set $E = \{\{(\ell, \bar{x}), (\ell+1, \bar{x}')\} \mid \ell \in [d], \bar{x} \in [n]^d, x_i' = x_i \text{ for } i \neq \ell\}$

Permutation Network $\mathrm{PN}(n, d)$



- There is an n-ary version of the Benes network (2 n-ary butterflies glued at level 0).
- identifying levels 0 and 1 (or 0 and -1) gives $\mathrm{PN}(n, d)$.

The d-dimensional mesh $M(n,d)$



- node set $V = [n]^d$
- edge set $E = \{\{(x_0,\dots,x_i,\dots,x_{d-1}),(x_0,\dots,x_i+1,\dots,x_{d-1})\} \mid x_s \in [n] \text{ for } s \in [d] \setminus \{i\}, x_i \in [n-1]\}$

Remarks

$M(2, d)$ is also called d-dimensional hypercube.

$M(n, 1)$ is also called linear array of length n.

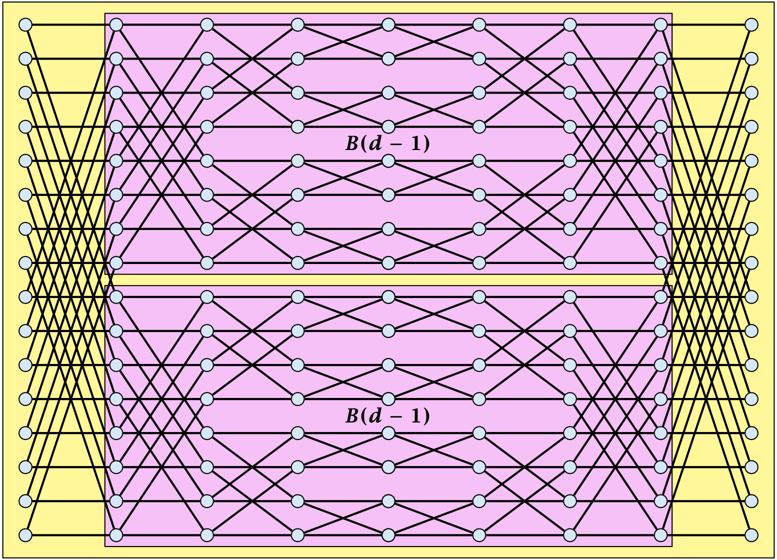
Permutation Routing

Lemma 59
On the linear array $M(n,1)$ any permutation can be routed online in $2n$ steps with buffersize 3.

Permutation Routing

Lemma 60
On the Beneš network any permutation can be routed offline in $2d$ steps between the sources level $(+d)$ and target level $(-d)$.

Recursive Beneš Network



Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at (α, d) and $(\alpha(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\alpha, -d)$ and $(\alpha(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing

base case $d = 0$

trivial

induction step $d \to d + 1$

- The packets that start at $(\bar{a}, d)$ and $(\bar{a}(d), d)$ have to be sent into different sub-networks.
- The packets that end at $(\bar{a}, -d)$ and $(\bar{a}(d), -d)$ have to come out of different sub-networks.

We can generate a graph on the set of packets.

- Every packet has an incident source edge (connecting it to the conflicting start packet)
- Every packet has an incident target edge (connecting it to the conflicting packet at its target)
- This clearly gives a bipartite graph; Coloring this graph tells us which packet to send into which sub-network.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Permutation Routing on the n-ary Beneš Network

Instead of two we have n sub-networks $B(n, d-1)$.

All packets starting at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to be send to different sub-networks.

All packets ending at positions $\{(x_0, \ldots, x_{d-2}, x_{d-1}, d) \mid x_{d-1} \in [n]\}$ have to come from different sub-networks.

The conflict graph is an n-uniform 2-regular hypergraph.

We can color such a graph with n colors such that no two nodes in a hyperedge share a color.

This gives the routing.

Lemma 61

On a d-dimensional mesh with sidelength n we can route any permutation (offline) in $4dn$ steps.

We can simulate the algorithm for the n-ary Beneš Network.

Each step can be simulated by routing on disjoint linear arrays. This takes at most $2n$ steps.

We can simulate the algorithm for the n-ary Beneš Network.

Each step can be simulated by routing on disjoint linear arrays. This takes at most $2n$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \ldots, -1, 0, 1, \ldots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \ldots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \ldots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \dots, -1, 0, 1, \dots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \dots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \dots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \dots, -1, 0, 1, \dots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \dots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \dots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \ldots, -1, 0, 1, \ldots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \ldots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \ldots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \dots, -1, 0, 1, \dots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \dots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \dots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \ldots, -1, 0, 1, \ldots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \ldots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \ldots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

We simulate the behaviour of the Beneš network on the n-dimensional mesh.

In round $r \in \{-d, \ldots, -1, 0, 1, \ldots, d-1\}$ we simulate the step of sending from level r of the Beneš network to level $r+1$.

Each node $\bar{x} \in [n]^d$ of the mesh simulates the node $(r, \bar{x})$.

Hence, if in the Beneš network we send from $(r, \bar{x})$ to $(r+1, \bar{x}')$ we have to send from $\bar{x}$ to $\bar{x}'$ in the mesh.

All communication is performed along linear arrays. In round $r < 0$ the linear arrays along dimension $-r-1$ (recall that dimensions are numbered from 0 to $d-1$) are used

$$\bar{x}_{d-1} \ldots \bar{x}_{-r} \alpha \bar{x}_{-r-2} \ldots \bar{x}_0$$

In rounds $r \geq 0$ linear arrays along dimension r are used.

Hence, we can perform a round in $\mathcal{O}(n)$ steps.

Lemma 62

We can route any permutation on the Beneš network in $\mathcal{O}(d)$ steps with constant buffer size.

The same is true for the butterfly network.

Lemma 62

We can route any permutation on the Beneš network in $\mathcal{O}(d)$ *steps with constant buffer size.*

The same is true for the butterfly network.

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \ldots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d-1$ and columns of length n^d.

We route in 3 phases:

1. Permute packets along the rows such that afterwards no column contains packets that have the same target row. $\mathcal{O}(d)$ steps.
2. We can use pipeling to permute every column, so that afterwards every packet is in its target row. $\mathcal{O}(2d + 2d)$ steps.
3. Every packet is in its target row. Permute packets to their right destinations. $\mathcal{O}(d)$ steps.

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \ldots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d-1$ and columns of length n^d.

We route in 3 phases:

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \ldots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d - 1$ and columns of length n^d.

We route in 3 phases:

1. Permute packets along the rows such that afterwards no column contains packets that have the same target row. $\mathcal{O}(d)$ steps.
2. We can use pipeling to permute **every** column, so that afterwards every packet is in its target row. $\mathcal{O}(2d + 2d)$ steps.
3. Every packet is in its target row. Permute packets to their right destinations. $\mathcal{O}(d)$ steps.

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \dots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d-1$ and columns of length n^d.

We route in 3 phases:

1. Permute packets along the rows such that afterwards no column contains packets that have the same target row. $\mathcal{O}(d)$ steps.
2. We can use pipeling to permute **every** column, so that afterwards every packet is in its target row. $\mathcal{O}(2d + 2d)$ steps.
3. Every packet is in its target row. Permute packets to their right destinations. $\mathcal{O}(d)$ steps.

The nodes are of the form $(\ell, \bar{x})$, $\bar{x} \in [n]^d, \ell \in -d, \dots, d$.

We can view nodes with same first coordinate forming columns and nodes with the same second coordinate as forming rows. This gives rows of length $2d-1$ and columns of length n^d.

We route in 3 phases:

1. Permute packets along the rows such that afterwards no column contains packets that have the same target row. $\mathcal{O}(d)$ steps.
2. We can use pipeling to permute **every** column, so that afterwards every packet is in its target row. $\mathcal{O}(2d + 2d)$ steps.
3. Every packet is in its target row. Permute packets to their right destinations. $\mathcal{O}(d)$ steps.

Lemma 63
We can do offline permutation routing of (partial) permutations in $2d$ steps on the hypercube.

Lemma 64
We can sort on the hypercube $M(2,d)$ in $\mathcal{O}(d^2)$ steps.

Lemma 65
We can do online permutation routing of permutations in $\mathcal{O}(d^2)$ steps on the hypercube.

Lemma 63
We can do offline permutation routing of (partial) permutations in $2d$ steps on the hypercube.

Lemma 64
We can sort on the hypercube $M(2,d)$ in $\mathcal{O}(d^2)$ steps.

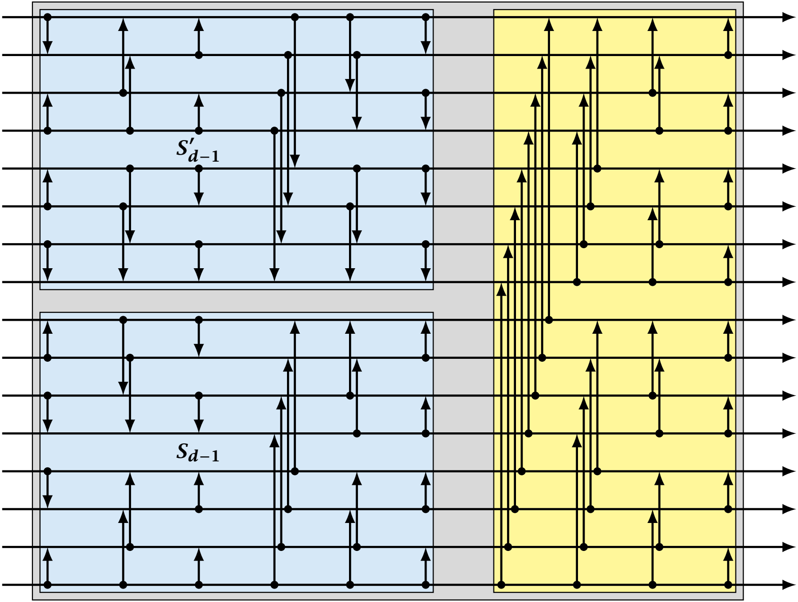
Lemma 65
We can do online permutation routing of permutations in $\mathcal{O}(d^2)$ steps on the hypercube.

Lemma 63
We can do offline permutation routing of (partial) permutations in $2d$ steps on the hypercube.

Lemma 64
We can sort on the hypercube $M(2,d)$ in $\mathcal{O}(d^2)$ steps.

Lemma 65
We can do online permutation routing of permutations in $\mathcal{O}(d^2)$ steps on the hypercube.

Bitonic Sorter S_d



ASCEND/DESCEND Programs

Algorithm 11 ASCEND(procedure *oper*)

1: **for** $dim = 0$ **to** $d - 1$
2:     **for all** $\bar{a} \in [2]^d$ **pardo**
3:         oper$(\bar{a}, \bar{a}(dim), dim)$

Algorithm 11 DESCEND(procedure *oper*)

1: **for** $dim = d - 1$ **to** 0
2:     **for all** $\bar{a} \in [2]^d$ **pardo**
3:         oper$(\bar{a}, \bar{a}(dim), dim)$

oper should only depend on the dimension and on values stored in the respective processor pair $(\bar{a}, \bar{a}(dim), V[\bar{a}], V[\bar{a}(dim)])$.

oper should take constant time.

Algorithm 11 oper$(a, a', dim, T_a, T_{a'})$

1: **if** $a_{dim}, \ldots, a_0 = 0^{dim+1}$ **then**
2: $\quad T_a = \min\{T_a, T_{a'}\}$

Performing an ASCEND run with this operation computes the minimum in processor 0.

We can sort on $M(2, d)$ by using d DESCEND runs.

We can do offline permutation routing by using a DESCEND run followed by an ASCEND run.

Algorithm 11 oper$(a, a', dim, T_a, T_{a'})$

1: **if** $a_{dim}, \ldots, a_0 = 0^{dim+1}$ **then**
2: $\quad T_a = \min\{T_a, T_{a'}\}$

Performing an ASCEND run with this operation computes the minimum in processor 0.

We can sort on $M(2, d)$ by using d DESCEND runs.

We can do offline permutation routing by using a DESCEND run followed by an ASCEND run.

Algorithm 11 oper$(a, a', dim, T_a, T_{a'})$

if $a_{dim}, \ldots, a_0 = 0^{dim+1}$ **then**
    $T_a = \min\{T_a, T_{a'}\}$

Performing an ASCEND run with this operation computes the minimum in processor 0.

We can sort on $M(2, d)$ by using d DESCEND runs.

We can do offline permutation routing by using a DESCEND run followed by an ASCEND run.

Algorithm 11 oper$(a, a', dim, T_a, T_{a'})$

1: **if** $a_{dim}, \ldots, a_0 = 0^{dim+1}$ **then**
2: $\quad T_a = \min\{T_a, T_{a'}\}$

Performing an ASCEND run with this operation computes the minimum in processor 0.

We can sort on $M(2,d)$ by using d DESCEND runs.

We can do offline permutation routing by using a DESCEND run followed by an ASCEND run.

We can perform an ASCEND/DESCEND run on a linear array $M(2^d, 1)$ in $\mathcal{O}(2^d)$ steps.

The CCC network is obtained from a hypercube by replacing every node by a cycle of degree d.

- nodes $\{(\ell,\bar{x}) \mid \bar{x} \in [2]^d, \ell \in [d]\}$
- edges $\{\{(\ell,\bar{x}),(\ell,\bar{x}(\ell)\} \mid x \in [2]^d, \ell \in [d]\}$

constand degree

Lemma 66
Let $d = 2^k$. An ASCEND run of a hypercube $M(2, d + k)$ can be simulated on CCC(d) in $\mathcal{O}(d)$ steps.

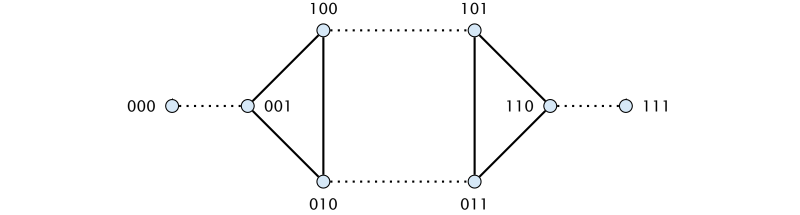
The shuffle exchange network SE(d) is defined as follows

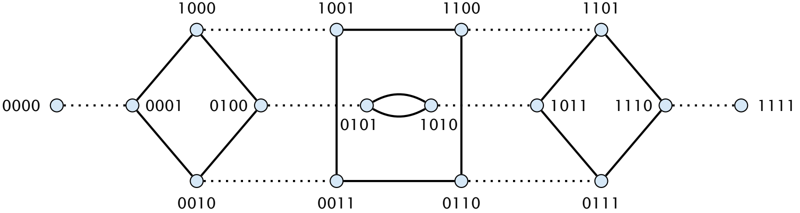
- nodes: $V = [2]^d$
- edges:
  $E = \Big\{\{x\bar{\alpha}, \bar{\alpha}x\} \mid x \in [2], \bar{\alpha} \in [2]^{d-1}\Big\} \cup \Big\{\{\bar{\alpha}0, \bar{\alpha}1\} \mid \bar{\alpha} \in [2]^{d-1}\Big\}$

constand degree

Edges of the first type are called shuffle edges. Edges of the second type are called exchange edges

Shuffle Exchange Networks





Lemma 67

We can perform an ASCEND run of $M(2, d)$ on $SE(d)$ in $\mathcal{O}(d)$ steps.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

For the following observations we need to make the definition of parallel computer networks more precise.

Each node of a given network corresponds to a processor/RAM.

In addition each processor has a read register and a write register.

In one (synchronous) step each neighbour of a processor P_i can write into P_i's write register or can read from P_i's read register.

Usually we assume that proper care has to be taken to avoid concurrent reads and concurrent writes from/to the same register.

Simulations between Networks

Definition 68
A configuration C_i of processor P_i is the complete description of the state of P_i including local memory, program counter, read-register, write-register, etc.

Suppose a machine M is in configuration $(C_0, \dots, C_{p-1})$, performs t synchronous steps, and is then in configuration $C = (C'_0, \dots, C'_{p-1})$.

C'_i is called the t-th successor configuration of C for processor i.

Simulations between Networks

Definition 68
A configuration C_i of processor P_i is the complete description of the state of P_i including local memory, program counter, read-register, write-register, etc.

Suppose a machine M is in configuration $(C_0, \dots, C_{p-1})$, performs t synchronous steps, and is then in configuration $C = (C'_0, \dots, C'_{p-1})$.

C'_i is called the t-th successor configuration of C for processor i.

Simulations between Networks

Definition 68

A configuration C_i of processor P_i is the complete description of the state of P_i including local memory, program counter, read-register, write-register, etc.

Suppose a machine M is in configuration $(C_0, \dots, C_{p-1})$, performs t synchronous steps, and is then in configuration $C = (C'_0, \dots, C'_{p-1})$.

C'_i is called the t-th successor configuration of C for processor i.

Simulations between Networks

Definition 69

Let $C = (C_0, \dots, C_{p-1})$ a configuration of M. A machine M' with $q \geq p$ processors weakly simulates t steps of M with slowdown k if

- in the beginning there are p non-empty processors sets $A_0, \dots, A_{p-1} \subseteq M'$ so that all processors in A_i know C_i;
- after at most $k \cdot t$ steps of M' there is a processor $Q^{(i)}$ that knows the t-th successors configuration of C for processor P_i.

Simulations between Networks

Definition 70
M' simulates M with slowdown k if

- M' weakly simulates machine M with slowdown k
- and **every** processor in A_i knows the t-th successor configuration of C for processor P_i.

We have seen how to simulate an ASCEND/DESCEND run of the hypercube $M(2, d+k)$ on CCC(d) with $d = 2^k$ in $\mathcal{O}(d)$ steps.

Hence, we can simulate $d + k$ steps (one ASCEND run) of the hypercube in $\mathcal{O}(d)$ steps. This means slowdown $\mathcal{O}(1)$.

Lemma 71

Suppose a network S with n processors can route any permutation in time $\mathcal{O}(t(n))$. Then S can simulate any ***constant degree*** *network M with at most n vertices with slowdown $\mathcal{O}(t(n))$.*

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Map the vertices of M to vertices of S in an arbitrary way.

Color the edges of M with $\Delta + 1$ colors, where $\Delta = \mathcal{O}(1)$ denotes the maximum degree.

Each color gives rise to a permutation.

We can route this permutation in S in $t(n)$ steps.

Hence, we can perform the required communication for one step of M by routing $\Delta + 1$ permutations in S. This takes time $t(n)$.

A processor of M is simulated by the same processor of S throughout the simulation.

Lemma 72

Suppose a network S with n processors can sort n numbers in time $\mathcal{O}(t(n))$. Then S can simulate any network M with at most n vertices with slowdown $\mathcal{O}(t(n))$.

Lemma 73

There is a constant degree network on $\mathcal{O}(n^{1+\epsilon})$ nodes that can simulate any constant degree network with slowdown $\mathcal{O}(1)$.

Suppose we allow concurrent reads, this means in every step all neighbours of a processor P_i can read P_i's read register.

Lemma 74

A constant degree network M that can simulate any n-node network has slowdown $\Omega(\log n)$ (independent of the size of M).

Suppose we allow concurrent reads, this means in every step all neighbours of a processor P_i can read P_i's read register.

Lemma 74

A constant degree network M that can simulate any n-node network has slowdown $\Omega(\log n)$ (independent of the size of M).

We show the lemma for the following type of simulation.

- There are representative sets A_i^t for every step t that specify which processors of M simulate processor P_i in step t (know the configuration of P_i after the t-th step).
- The representative sets for different processors are disjoint.
- for all $i \in \{1, \dots, n\}$ and steps t, $A_i^t \neq \emptyset$.

This is a step-by-step simulation.

We show the lemma for the following type of simulation.

- There are representative sets A_i^t for every step t that specify which processors of M simulate processor P_i in step t (know the configuration of P_i after the t-th step).
- The representative sets for different processors are disjoint.
- for all $i \in \{1, \ldots, n\}$ and steps t, $A_i^t \neq \emptyset$.

This is a step-by-step simulation.

Suppose processor P_i reads from processor P_{j_i} in step t.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let k_t be the largest distance (maximized over all i, j_i).

Then the simulation of step t takes time at least k_t.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

Suppose processor P_i reads from processor P_{j_i} in step t.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let k_t be the largest distance (maximized over all i, j_i).

Then the simulation of step t takes time at least k_t.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

Suppose processor P_i reads from processor P_{j_i} in step t.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let k_t be the largest distance (maximized over all i, j_i).

Then the simulation of step t takes time at least k_t.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

Suppose processor P_i reads from processor P_{j_i} in step t.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let k_t be the largest distance (maximized over all i, j_i).

Then the simulation of step t takes time at least k_t.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

Suppose processor P_i reads from processor P_{j_i} in step t.

Every processor $Q \in M$ with $Q \in A_i^{t+1}$ must have a path to a processor $Q' \in A_i^t$ and to $Q'' \in A_{j_i}^t$.

Let k_t be the largest distance (maximized over all i, j_i).

Then the simulation of step t takes time at least k_t.

The slowdown is at least

$$k = \frac{1}{\ell} \sum_{t=1}^{\ell} k_t$$

We show

- The simulation of a step takes at least time $\gamma \log n$, or
- the size of the representative sets shrinks by a lot

$$\sum_i |A_i^{t+1}| \leq \frac{1}{n^\epsilon} \sum_i |A_i^t|$$

We show

- The simulation of a step takes at least time $\gamma \log n$, or
- the size of the representative sets shrinks by a lot

$$\sum_i |A_i^{t+1}| \leq \frac{1}{n^\epsilon} \sum_i |A_i^t|$$

Suppose there is no pair (i, j) such that i reading from j requires time $\gamma \log n$.

Suppose there is no pair (i, j) such that i reading from j requires time $\gamma \log n$.

- For every i the set $\Gamma_{2k}(A_i)$ contains a node from A_j.
- Hence, there must exist a j_i such that $\Gamma_{2k}(A_i)$ contains at most

$$|C_{j_i}| := \frac{|A_i| \cdot c^{2k}}{n-1} \leq \frac{|A_i| \cdot c^{3k}}{n} \ .$$

processors from $|A_{j_i}|$

Suppose there is no pair (i, j) such that i reading from j requires time $\gamma \log n$.

- For every i the set $\Gamma_{2k}(A_i)$ contains a node from A_j.
- Hence, there must exist a j_i such that $\Gamma_{2k}(A_i)$ contains at most
$$|C_{j_i}| := \frac{|A_i| \cdot c^{2k}}{n-1} \leq \frac{|A_i| \cdot c^{3k}}{n} \ .$$
processors from $|A_{j_i}|$

If we choose that i reads from j_i we get

$$|A_i'|$$

If we choose that i reads from j_i we get

$$|A_i'| \leq |C_{j_i}| \cdot c^k$$

If we choose that i reads from j_i we get

$$\begin{aligned}|A_i'| &\le |C_{j_i}| \cdot c^k \\ &\le c^k \cdot \frac{|A_i| \cdot c^{3k}}{n}\end{aligned}$$

If we choose that i reads from j_i we get

$$\begin{aligned}|A_i'| &\le |C_{j_i}| \cdot c^k \\ &\le c^k \cdot \frac{|A_i| \cdot c^{3k}}{n} \\ &= \frac{1}{n}|A_i| \cdot c^{4k}\end{aligned}$$

If we choose that i reads from j_i we get

$$
\begin{aligned}
|A_i'| &\le |C_{j_i}| \cdot c^k \\
&\le c^k \cdot \frac{|A_i| \cdot c^{3k}}{n} \\
&= \frac{1}{n}|A_i| \cdot c^{4k}
\end{aligned}
$$

If we choose that i reads from j_i we get

$$\begin{aligned}|A_i'| &\leq |C_{j_i}| \cdot c^k \\ &\leq c^k \cdot \frac{|A_i| \cdot c^{3k}}{n} \\ &= \frac{1}{n}|A_i| \cdot c^{4k}\end{aligned}$$

Choosing $k = \Theta(\log n)$ gives that this is at most $|A_i|/n^{\epsilon}$.

Let ℓ be the total number of steps and s be the number of short steps when $k_t < \gamma \log n$.

In a step of time k_t a representative set can at most increase by c^{k_t+1}.

Let h_ℓ denote the number of representatives after step ℓ.

Let ℓ be the total number of steps and s be the number of short steps when $k_t < \gamma \log n$.

In a step of time k_t a representative set can at most increase by c^{k_t+1}.

Let h_ℓ denote the number of representatives after step ℓ.

Let ℓ be the total number of steps and s be the number of short steps when $k_t < \gamma \log n$.

In a step of time k_t a representative set can at most increase by c^{k_t+1}.

Let h_ℓ denote the number of representatives after step ℓ.

$$n \le h_\ell \le h_0\Big(\frac{1}{n^\epsilon}\Big)^s \prod_{t\in\text{long}} c^{k_t+1} \le \frac{n}{n^{\epsilon s}} \cdot c^{\ell+\sum_t k_t}$$

If $\sum_t k_t \ge \ell(\frac{\epsilon}{2}\log_c n - 1)$, we are done. Otw.

$$n \le n^{1-\epsilon s+\ell\frac{\epsilon}{2}}$$

This gives $s \le \ell/2$.

Hence, at most 50% of the steps are short.

$$n \le h_\ell \le h_0\left(\frac{1}{n^\epsilon}\right)^s \prod_{t\in\mathrm{long}} c^{k_t+1} \le \frac{n}{n^{\epsilon s}} \cdot c^{\ell+\sum_t k_t}$$

If $\sum_t k_t \ge \ell(\frac{\epsilon}{2}\log_c n - 1)$, we are done. Otw.

$$n \le n^{1-\epsilon s+\ell\frac{\epsilon}{2}}$$

This gives $s \le \ell/2$.

Hence, at most 50% of the steps are short.

$$n \leq h_\ell \leq h_0 \Big(\frac{1}{n^\epsilon}\Big)^s \prod_{t \in \text{long}} c^{k_t+1} \leq \frac{n}{n^{\epsilon s}} \cdot c^{\ell + \sum_t k_t}$$

If $\sum_t k_t \geq \ell(\frac{\epsilon}{2}\log_c n - 1)$, we are done. Otw.

$$n \leq n^{1-\epsilon s + \ell\frac{\epsilon}{2}}$$

This gives $s \leq \ell/2$.

Hence, at most 50% of the steps are short.

$$n \leq h_\ell \leq h_0 \Big(\frac{1}{n^\epsilon}\Big)^s \prod_{t \in \text{long}} c^{k_t+1} \leq \frac{n}{n^{\epsilon s}} \cdot c^{\ell + \sum_t k_t}$$

If $\sum_t k_t \geq \ell(\frac{\epsilon}{2} \log_c n - 1)$, we are done. Otw.

$$n \leq n^{1-\epsilon s + \ell \frac{\epsilon}{2}}$$

This gives $s \leq \ell/2$.

Hence, at most 50% of the steps are short.

Deterministic Online Routing

Lemma 75
A permutation on an $n \times n$-mesh can be routed online in $\mathcal{O}(n)$ steps.

Deterministic Online Routing

Definition 76 (Oblivious Routing)

Specify a path-system $\mathcal{W}$ with a path $P_{u,v}$ between u and v for every pair $\{u,v\} \in V \times V$.

A packet with source u and destination v moves along path $P_{u,v}$.

Deterministic Online Routing

Definition 77 (Oblivious Routing)

Specify a path-system $\mathcal{W}$ with a path $P_{u,v}$ between u and v for every pair $\{u, v\} \in V \times V$.

Definition 78 (node congestion)

For a given path-system the node congestion is the maximum number of path that go through any node $v \in V$.

Definition 79 (edge congestion)

For a given path-system the edge congestion is the maximum number of path that go through any edge $e \in E$.

Deterministic Online Routing

Definition 80 (dilation)

For a given path system the dilation is the maximum length of a path.

Lemma 81
Any oblivious routing protocol requires at least $\max\{C_f, D_f\}$ *steps, where* C_f *and* D_f*, are the congestion and dilation, respectively, of the path-system used. (node congestion or edge congestion depending on the communication model)*

Lemma 82
Any reasonable oblivious routing protocol requires at most $\mathcal{O}(D_f \cdot C_f)$ *steps (unbounded buffers).*

Theorem 83 (Borodin, Hopcroft)

For any path system $\mathcal{W}$ there exists a permutation $\pi : V \to V$ and an edge $e \in E$ such that at least $\Omega(\sqrt{n}/\Delta)$ of the paths go through e.

Let $\mathcal{W}_v = \{P_{v,u} \mid u \in V\}$.

We say that an edge e is z-popular for v if at least z paths from $\mathcal{W}_v$ contain e.

For any node v there are many edges that are are quite popular for v.

$|V| \times |E|$-matrix $A(z)$:

$$A_{v,e}(z) = \begin{cases} 1 & e \text{ is } z\text{-popular for } v \\ 0 & \text{otherwise} \end{cases}$$

Define

- $$A_v(z) = \sum_e A_{v,e}(z)$$
- $$A_e(z) = \sum_v A_{v,e}(z)$$

Lemma 84

Let $z \leq \frac{n-1}{\Delta}$.

For every node $v \in V$ *there exist at least* $\frac{n}{2\Delta z}$ *edges that are* z *popular for* v. *This means*

$$A_v(z) \geq \frac{n}{2\Delta z}$$

Lemma 85

There exists an edge e' that is z-popular for at least z nodes with $z = \Omega(\sqrt{n}\Delta)$.

$$\sum_e A_e(z) = \sum_v A_v(z) \geq \frac{n^2}{2\Delta z}$$

There must exist an edge e'

$$A_{e'}(z) \geq \left\lceil \frac{n^2}{|E| \cdot 2\Delta z} \right\rceil \geq \left\lceil \frac{n}{2\Delta^2 z} \right\rceil$$

where the last step follows from $|E| \leq \Delta n$.

We choose z such that $z = \frac{n}{2\Delta^2 z}$ (i.e., $z = \sqrt{n}/(\sqrt{2}\Delta)$).

This means e' is $\lceil z \rceil$-popular for $\lceil z \rceil$ nodes.

We can construct a permutation such that z paths go through e'.

Deterministic oblivious routing may perform very poorly.

What happens if we have a random routing problem in a butterfly?

Suppose every source on level 0 has p packets, that are routed to random destinations.

How many packets go over node v on level i?

From v we can reach $2^d/2^i$ different targets.

Hence,

$$\Pr[\text{packet goes over } v] \le \frac{2^{d-i}}{2^d} = \frac{1}{2^i}$$

Suppose every source on level 0 has p packets, that are routed to random destinations.

How many packets go over node v on level i?

From v we can reach $2^d/2^i$ different targets.

Hence,

$$\Pr[\text{packet goes over } v] \leq \frac{2^{d-i}}{2^d} = \frac{1}{2^i}$$

Suppose every source on level 0 has p packets, that are routed to random destinations.

How many packets go over node v on level i?

From v we can reach $2^d/2^i$ different targets.

Hence,

$$\Pr[\text{packet goes over } v] \leq \frac{2^{d-i}}{2^d} = \frac{1}{2^i}$$

Suppose every source on level 0 has p packets, that are routed to random destinations.

How many packets go over node v on level i?

From v we can reach $2^d/2^i$ different targets.

Hence,

$$\Pr[\text{packet goes over } v] \leq \frac{2^{d-i}}{2^d} = \frac{1}{2^i}$$

Expected number of packets:

$$E[\text{packets over } v] = p \cdot 2^i \cdot \frac{1}{2^i} = p$$

since only $p2^i$ packets can reach v.

But this is trivial.

Expected number of packets:

$$\mathrm{E}[\text{packets over } v] = p \cdot 2^i \cdot \frac{1}{2^i} = p$$

since only $p2^i$ packets can reach v.

But this is trivial.

What is the probability that at least r packets go through v.

$$
\begin{aligned}
\Pr[\text{at least } r \text{ path through } v] &\le \binom{p\cdot 2^i}{r}\cdot\left(\frac{1}{2^i}\right)^r \\
&\le \left(\frac{p2^i\cdot e}{r}\right)^r\cdot\left(\frac{1}{2^i}\right) \\
&= \left(\frac{pe}{r}\right)^r
\end{aligned}
$$

$$
\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \le d2^d\cdot\left(\frac{pe}{r}\right)^r
$$

What is the probability that at least r packets go through v.

$$\Pr[\text{at least } r \text{ path through } v] \le \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r$$

$$\le \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right)$$

$$= \left(\frac{pe}{r}\right)^r$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v]$$

$$\le d2^d \cdot \left(\frac{pe}{r}\right)^r$$

What is the probability that at least r packets go through v.

$$
\begin{aligned}
\Pr[\text{at least } r \text{ path through } v] &\leq \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \\
&\leq \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right) \\
&= \left(\frac{pe}{r}\right)^r
\end{aligned}
$$

$$
\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v]
$$

$$
\leq d2^d \cdot \left(\frac{pe}{r}\right)^r
$$

What is the probability that at least r packets go through v.

$$\Pr[\text{at least } r \text{ path through } v] \le \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \le \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right) = \left(\frac{pe}{r}\right)^r$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \le d2^d \cdot \left(\frac{pe}{r}\right)^r$$

What is the probability that at least r packets go through v.

$$\begin{aligned}\Pr[\text{at least } r \text{ path through } v] &\leq \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \\ &\leq \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right) \\ &= \left(\frac{pe}{r}\right)^r\end{aligned}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v]$$

$$\leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

What is the probability that at least r packets go through v.

$$\begin{aligned}
\Pr[\text{at least } r \text{ path through } v] &\leq \binom{p \cdot 2^i}{r} \cdot \left(\frac{1}{2^i}\right)^r \\
&\leq \left(\frac{p2^i \cdot e}{r}\right)^r \cdot \left(\frac{1}{2^i}\right) \\
&= \left(\frac{pe}{r}\right)^r
\end{aligned}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v]$$

$$\leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v]$

$$\leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose r as $2ep + (\ell + 1)d + \log d = \mathcal{O}(p + \log N)$, where N is number of sources in BF(d).

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v] \leq \frac{1}{N^\ell}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose r as $2ep + (\ell + 1)d + \log d = \mathcal{O}(p + \log N)$, where N is number of sources in BF(d).

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v] \leq \frac{1}{N^\ell}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose r as $2ep + (\ell + 1)d + \log d = \mathcal{O}(p + \log N)$, where N is number of sources in BF(d).

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v] \leq \frac{1}{N^\ell}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose r as $2ep + (\ell + 1)d + \log d = \mathcal{O}(p + \log N)$, where N is number of sources in $\mathrm{BF}(d)$.

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v] \leq \frac{1}{N^\ell}$$

$$\Pr[\text{there exists a node } v \text{ sucht that at least } r \text{ path through } v] \leq d2^d \cdot \left(\frac{pe}{r}\right)^r$$

Choose r as $2ep + (\ell + 1)d + \log d = \mathcal{O}(p + \log N)$, where N is number of sources in $\mathrm{BF}(d)$.

$$\Pr[\text{exists node } v \text{ with more than } r \text{ paths over } v] \leq \frac{1}{N^\ell}$$

Scheduling Packets

Assume that in every round a node may forward at most one packet but may receive up to two.

We select a random rank $R_p \in [k]$. Whenever, we forward a packet we choose the packet with smaller rank. Ties are broken according to packet id.

Random Rank Protocol

Scheduling Packets

Assume that in every round a node may forward at most one packet but may receive up to two.

We select a random rank $R_p \in [k]$. Whenever, we forward a packet we choose the packet with smaller rank. Ties are broken according to packet id.

Random Rank Protocol

Scheduling Packets

Assume that in every round a node may forward at most one packet but may receive up to two.

We select a random rank $R_p \in [k]$. Whenever, we forward a packet we choose the packet with smaller rank. Ties are broken according to packet id.

Random Rank Protocol

Scheduling Packets

Assume that in every round a node may forward at most one packet but may receive up to two.

We select a random rank $R_p \in [k]$. Whenever, we forward a packet we choose the packet with smaller rank. Ties are broken according to packet id.

Random Rank Protocol

Definition 86 (Delay Sequence of length s)

- delay path $\mathcal{W}$
- lengths $\ell_0, \ell_1, \dots, \ell_s$, with $\ell_0 \geq 1, \ell_1, \dots, \ell_s \geq 0$ lengths of delay-free sub-paths
- collision nodes $v_0, v_1, \dots, v_s, v_{s+1}$
- collision packets $P_0, \dots, P_s$

Definition 86 (Delay Sequence of length s)

- delay path $\mathcal{W}$
- lengths $\ell_0, \ell_1, \dots, \ell_s$, with $\ell_0 \geq 1$, $\ell_1, \dots, \ell_s \geq 0$ lengths of delay-free sub-paths
- collision nodes $v_0, v_1, \dots, v_s, v_{s+1}$
- collision packets $P_0, \dots, P_s$

Definition 86 (Delay Sequence of length s)

- delay path $\mathcal{W}$
- lengths $\ell_0, \ell_1, \ldots, \ell_s$, with $\ell_0 \geq 1$, $\ell_1, \ldots, \ell_s \geq 0$ lengths of delay-free sub-paths
- collision nodes $v_0, v_1, \ldots, v_s, v_{s+1}$
- collision packets $P_0, \ldots, P_s$

Definition 86 (Delay Sequence of length s)

- delay path $\mathcal{W}$
- lengths $\ell_0, \ell_1, \ldots, \ell_s$, with $\ell_0 \geq 1$, $\ell_1, \ldots, \ell_s \geq 0$ lengths of delay-free sub-paths
- collision nodes $v_0, v_1, \ldots, v_s, v_{s+1}$
- collision packets $P_0, \ldots, P_s$

Properties

- $\text{rank}(P_0) \geq \text{rank}(P_1) \geq \cdots \geq \text{rank}(P_s)$
- $\sum_{i=0}^{s} \ell_i = d$
- if the routing takes $d + s$ steps than the delay sequence has length s

Properties

- $\text{rank}(P_0) \geq \text{rank}(P_1) \geq \cdots \geq \text{rank}(P_s)$
- $\sum_{i=0}^{s} \ell_i = d$
- if the routing takes $d + s$ steps than the delay sequence has length s

Properties

- $\mathrm{rank}(P_0) \geq \mathrm{rank}(P_1) \geq \cdots \geq \mathrm{rank}(P_s)$
- $\sum_{i=0}^{s} \ell_i = d$
- if the routing takes $d + s$ steps than the delay sequence has length s

Definition 87 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length d from a source to a target
- s integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with v_i being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s + 1$ packets $P_0, \ldots, P_s$, where P_i is a packet with path through v_i and v_{i-1}
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

Definition 87 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length d from a source to a target
- s integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with v_i being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s + 1$ packets $P_0, \ldots, P_s$, where P_i is a packet with path through v_i and v_{i-1}
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

Definition 87 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length d from a source to a target
- s integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with v_i being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s+1$ packets $P_0, \ldots, P_s$, where P_i is a packet with path through v_i and v_{i-1}
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

Definition 87 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length d from a source to a target
- s integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with v_i being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s+1$ packets $P_0, \ldots, P_s$, where P_i is a packet with path through v_i and v_{i-1}
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

Definition 87 (Formal Delay Sequence)

- a path $\mathcal{W}$ of length d from a source to a target
- s integers $\ell_0 \geq 1, \ell_1, \ldots, \ell_s \geq 0$ and $\sum_{i=0}^{s} \ell_i = d$
- nodes $v_0, \ldots v_s, v_{s+1}$ on $\mathcal{W}$ with v_i being on level $d - \ell_0 - \cdots - \ell_{i-1}$
- $s+1$ packets $P_0, \ldots, P_s$, where P_i is a packet with path through v_i and v_{i-1}
- numbers $R_s \leq R_{s-1} \leq \cdots \leq R_0$

We say a formal delay sequence is active if $\operatorname{rank}(P_i) = k_i$ holds for all i.

Let N_s be the number of formal delay sequences of length at most s. Then

$$\Pr[\text{routing needs at least } d + s \text{ steps}] \le \frac{N_s}{k^{s+1}}$$

We say a formal delay sequence is active if $\text{rank}(P_i) = k_i$ holds for all i.

Let N_s be the number of formal delay sequences of length at most s. Then

$$\Pr[\text{routing needs at least } d + s \text{ steps}] \le \frac{N_s}{k^{s+1}}$$

Lemma 88

$$N_s \le \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

Lemma 88

$$N_s \leq \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

- there are N^2 ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose ℓ_i's with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most C^{s+1} ways to choose the collision packets where C is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \leq k_s \leq \cdots \leq k_0 < k$

Lemma 88

$$N_s \leq \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

- there are N^2 ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose ℓ_i's with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most C^{s+1} ways to choose the collision packets where C is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \leq k_s \leq \cdots \leq k_0 < k$

Lemma 88

$$N_s \leq \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

- there are N^2 ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose ℓ_i's with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most C^{s+1} ways to choose the collision packets where C is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \leq k_s \leq \cdots \leq k_0 < k$

Lemma 88

$$N_s \le \left(\frac{2eC(s+k)}{s+1} \right)^{s+1}$$

- there are N^2 ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose ℓ_i's with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most C^{s+1} ways to choose the collision packets where C is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \le k_s \le \cdots \le k_0 < k$

Lemma 88

$$N_s \le \left(\frac{2eC(s+k)}{s+1}\right)^{s+1}$$

- there are N^2 ways to choose $\mathcal{W}$
- there are $\binom{s+d-1}{s}$ ways to choose ℓ_i's with $\sum_{i=0}^{s} \ell_i = d$
- the collision nodes are fixed
- there are at most C^{s+1} ways to choose the collision packets where C is the node congestion
- there are at most $\binom{s+k}{s+1}$ ways to choose $0 \le k_s \le \cdots \le k_0 < k$

Hence the probability that the routing takes more than $d + s$ steps is at most

$$N^3 \cdot \left(\frac{2e \cdot C \cdot (s+k)}{(s+1)k}\right)^{s+1}$$

We choose $s = 8eC - 1 + (\ell + 3)d$ and $k = s + 1$. This gives that the probability is at most $\frac{1}{N^\ell}$.

Hence the probability that the routing takes more than $d + s$ steps is at most

$$N^3 \cdot \left(\frac{2e \cdot C \cdot (s+k)}{(s+1)k}\right)^{s+1}$$

We choose $s = 8eC - 1 + (\ell + 3)d$ and $k = s + 1$. This gives that the probability is at most $\frac{1}{N^\ell}$.

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with p packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with p packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with p packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with p packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

- With probability $1 - \frac{1}{N^{\ell_1}}$ the random routing problem has congestion at most $\mathcal{O}(p + \ell_1 d)$.
- With probability $1 - \frac{1}{N^{\ell_2}}$ the packet scheduling finishes in at most $\mathcal{O}(C + \ell_2 d)$ steps.

Hence, with high probability routing random problems with p packets per source in a butterfly requires only $\mathcal{O}(p + d)$ steps.

What do we do for arbitrary routing problems?

Valiants Trick

Where did the scheduling analysis use the butterfly?

We only used

- all routing paths are of the same length d
- there are a polynomial number of delay paths

Choose paths as follows:

- route from source to random destination on target level
- route to real target column (albeit on source level)
- route to target

All phases run in time $\mathcal{O}(p + d)$ with high probability.

Valiants Trick

Where did the scheduling analysis use the butterfly?

We only used

- all routing paths are of the same length d
- there are a polynomial number of delay paths

Choose paths as follows:

- route from source to random destination on target level
- route to real target column (albeit on source level)
- route to target

All phases run in time $\mathcal{O}(p + d)$ with high probability.

Valiants Trick

Where did the scheduling analysis use the butterfly?

We only used

- all routing paths are of the same length d
- there are a polynomial number of delay paths

Choose paths as follows:

- route from source to random destination on target level
- route to real target column (albeit on source level)
- route to target

All phases run in time $\mathcal{O}(p + d)$ with high probability.

Valiants Trick

Where did the scheduling analysis use the butterfly?

We only used

- all routing paths are of the same length d
- there are a polynomial number of delay paths

Choose paths as follows:

- route from source to random destination on target level
- route to real target column (albeit on source level)
- route to target

All phases run in time $\mathcal{O}(p + d)$ with high probability.

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \ldots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \ldots, k\} \rightarrow \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \leq c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i(u,v) = \sum_{w:(v,w)\in E} f_i(v,w)$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$:
    $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

Multicommodity Flow Problem

- undirected (weighted) graph $G = (V, E, c)$
- commodities (s_i, t_i), $i \in \{1, \dots, k\}$
- a multicommodity flow is a flow $f : E \times \{1, \dots, k\} \to \mathbb{R}^+$
  - for all edges $e \in E$: $\sum_i f_i(e) \le c(e)$
  - for all nodes $v \in V \setminus \{s_i, t_i\}$: $\sum_{u:(u,v)\in E} f_i((u,v)) = \sum_{w:(v,w)\in E} f_i((v,w))$

Goal A (Maximum Multicommodity Flow)
maximize $\sum_i \sum_{e=(s_i,x)\in E} f_i(e)$

Goal B (Maximum Concurrent Multicommodity Flow)
maximize $\min_i \sum_{e=(s_i,x)\in E} f_i(e)/d_i$ (throughput fraction), where d_i is demand for commodity i

Valiants Trick

A Balanced Multicommodity Flow Problem is a concurrent multicommodity flow problem in which incoming and outgoing flow is equal to

$$c(v) = \sum_{e=(v,x)\in E} c(e)$$

Valiants Trick

For a multicommodity flow S we assume that we have a decomposition of the flow(s) into flow-paths.

We use $C(S)$ to denote the congestion of the flow problem (inverse of througput fraction), and $D(S)$ the length of the longest routing path.

Valiants Trick

For a multicommodity flow S we assume that we have a decomposition of the flow(s) into flow-paths.

We use $C(S)$ to denote the congestion of the flow problem (inverse of througput fraction), and $D(S)$ the length of the longest routing path.

For a network $G = (V, E, c)$ we define the characteristic flow problem via

- demands $d_{u,v} = \frac{c(u)c(v)}{c(V)}$

Suppose the characteristic flow problem has a solution S with $C(S) \leq F$ and $D(S) \leq F$.

For a network $G = (V, E, c)$ we define the characteristic flow problem via

- demands $d_{u,v} = \frac{c(u)c(v)}{c(V)}$

Suppose the characteristic flow problem has a solution S with $C(S) \le F$ and $D(S) \le F$.

Definition 89

A (randomized) oblivious routing scheme is given by a path system $\mathcal{P}$ and a weight function w such that

$$\sum_{p \in \mathcal{P}_{s,t}} w(p) = 1$$

Construct an oblivious routing scheme from S as follows:

- let $f_{x,y}$ be the flow between x and y in S
- $$f_{x,y} \geq d_{x,y}/C(S) \geq d_{x,y}/F = \frac{1}{F}\frac{c(x)c(y)}{c(V)}$$
- for $p \in \mathcal{P}_{x,y}$ set $w(p) = f_p/f_{x,y}$

gives an oblivious routing scheme.

Valiants Trick

We apply this routing scheme twice:

- first choose a path from $\mathcal{P}_{s,v}$, where v is chosen uniformly according to $c(v)/c(V)$
- then choose path according to $\mathcal{P}_{v,t}$

If the input flow problem/packet routing problem is balanced doing this randomization results in flow solution S (twice).

Hence, we have an oblivious scheme with congestion and dilation at most $2F$ for (balanced inputs).

Valiants Trick

We apply this routing scheme twice:

- first choose a path from $\mathcal{P}_{s,v}$, where v is chosen uniformly according to $c(v)/c(V)$
- then choose path according to $\mathcal{P}_{v,t}$

If the input flow problem/packet routing problem is balanced doing this randomization results in flow solution S (twice).

Hence, we have an oblivious scheme with congestion and dilation at most $2F$ for (balanced inputs).

Example: hypercube.

Oblivious Routing for the Mesh

We can route any permutation on an $n \times n$ mesh in $\mathcal{O}(n)$ steps, by x-y routing. Actually $\mathcal{O}(d)$ steps where d is the largest distance between a source-target pair.

What happens if we do not have a permutation?

$x - y$ routing may generate large congestion if some pairs have a lot of packets.

Valiants trick may create a large dilation.

Oblivious Routing for the Mesh

We can route any permutation on an $n \times n$ mesh in $\mathcal{O}(n)$ steps, by x-y routing. Actually $\mathcal{O}(d)$ steps where d is the largest distance between a source-target pair.

What happens if we do not have a permutation?

$x - y$ routing may generate large congestion if some pairs have a lot of packets.

Valiants trick may create a large dilation.

Oblivious Routing for the Mesh

We can route any permutation on an $n \times n$ mesh in $\mathcal{O}(n)$ steps, by x-y routing. Actually $\mathcal{O}(d)$ steps where d is the largest distance between a source-target pair.

What happens if we do not have a permutation?

$x - y$ routing may generate large congestion if some pairs have a lot of packets.

Valiants trick may create a large dilation.

Oblivious Routing for the Mesh

We can route any permutation on an $n \times n$ mesh in $\mathcal{O}(n)$ steps, by x-y routing. Actually $\mathcal{O}(d)$ steps where d is the largest distance between a source-target pair.

What happens if we do not have a permutation?

$x - y$ routing may generate large congestion if some pairs have a lot of packets.

Valiants trick may create a large dilation.

Let for a multicommodity flow problem P $C_{\text{opt}}(P)$ be the optimum congestion, and $D_{\text{opt}}(P)$ be the optimum dilation (by perhaps different flow solutions).

Lemma 90

There is an oblivious routing scheme for the mesh that obtains a flow solution S with $C(S) = \mathcal{O}(C_{\text{opt}}(P) \log n)$ and $D(S) = \mathcal{O}(D_{\text{opt}}(P))$.

Let for a multicommodity flow problem P $C_{\text{opt}}(P)$ be the optimum congestion, and $D_{\text{opt}}(P)$ be the optimum dilation (by perhaps different flow solutions).

Lemma 90

There is an oblivious routing scheme for the mesh that obtains a flow solution S with $C(S) = \mathcal{O}(C_{\text{opt}}(P) \log n)$ and $D(S) = \mathcal{O}(D_{\text{opt}}(P))$.

Lemma 91

For any oblivious routing scheme on the mesh there is a demand P such that routing P will give congestion $\Omega(\log n \cdot C_{\text{opt}})$.