

Parallel Computing (CS 309)

Introduction

Why Parallel Computing?

- **Processors have become faster** at a tremendous rate
- For the last 30 years the clock rate has been doubling about every 18 months
- However, this trend cannot continue for another 30 years because:
 - *processors must have a certain minimal size and*
 - *information cannot travel faster than the speed of light*

Why Parallel Computing?

- Nevertheless, the demand for computing power will continue to increase
- There are **several problems** which are very urgent and at the same time are very time consuming
- **Example: weather forecasting**
 - *Forecasting based on certain parameters*
 - *Use of several parameters for better prediction*
 - *More the parameters, more computational cost*
- **Then what is the solution:**
 - **Use of more than one processor**
 - To meet both kinds of demands, future and current, it is natural to consider use of more than one processor for solving a single task

Why Parallel Computing?

- Advantages:
 - Processors are not only becoming faster but also cheaper, hence parallel computing becomes more and more realistic
 - Production of multi-processor chips provides even a cheaper solution
 - **Reduced cost if *old Processors* are used:**
 - Currently the newest processors are expensive, but processors of the previous generation are cheap
 - If one strives for the maximum number of clock cycles per money unit, then clearly one should not buy the top model, but rather by four with half the speed

Issues in Parallel Computing

- By using P processors instead of 1, we may hope to become P times faster, but this is rarely achieved
- Therefore, it is not sure that four 2 GHz processors will solve a problem faster than a single 4 GHz processor

Parallel Computer

- **In simple terms:**
 - A computer with several processors is called a parallel computer
- **Parallel Computing:**
 - It is the study of the usage of parallel computers in order to solve tasks faster

Task Parallelism

- **In task parallelism, several processors to solve several processes**
 - The simplest way to use parallel computers
- **This can both be done for**
 - multi-user system, and
 - single-user system
- **For example:**
 - **Large data servers often have many processors**
 - The incoming data requests are scheduled over the processors and each request is handled by a single processor
 - Such systems are commercially employed and may have hundreds of processors

Task Parallelism

- **How to do task parallelism on a single-user system?**
 - various processes of the user may be allocated to several processors
 - At a lower level, the various kinds of instructions may be executed by several processors
- **Issues in a single-user system:**
 - The degree of parallelism is rather bounded here
 - **Since a single user typically runs only one time-demanding process**, and hence instructions cannot be subdivided arbitrarily

Task Parallelism

- Even though technologically different, this kind of parallelism can be characterized as being **task parallelism**
- **However, load balancing is a measure issue**
 - The most important theoretical question in this context is how to schedule the tasks
 - That means, how to allocate the tasks to the processors so that an even **load balancing** is achieved

Task Parallelism

- Load balancing problem has several variants
- For a fixed set of tasks:
 - goal is to minimize the make span, the time to complete all tasks, respecting interdependencies, if any
- In an online context:
 - more reasonable goal is to try to minimize the completion time, the time before any job has been completed, or to minimize the time before any job starts being done

Task Parallelism

- Task parallelism is important, however, how much we can do to improve it?
 - Not much
- In practical applications:
 - task parallelism is the most important branch of parallel computing
 - however, **from an algorithmic point of view, it is not very interesting** because there is not so much one can do
- We will not study Task Parallelism in this course

True Parallelism

- **This course is about**
 - how a single computational task can be efficiently solved using more than one processor
- ***P* processor does not do *P* times faster**
 - As said earlier, ideally with P processors the problem should be solved P times faster, however, this is rarely achieved

True Parallelism

- P processors will not work P times faster than a single processor
- There are many reason for it
 - Communication overhead
 - Memory sharing
 - Sub-steps for which we cannot possibly use P processors
 - *etc.*

True Parallelism

- **The central topic of this course is**
 - the design of algorithms which allow to use as many processors as possible,
 - while loosing at most a constant factor in efficiency
- That means, a **parallel algorithm is expected to be more work-intensive by a constant factor, but normally not more than that**

True Parallelism

- **For few problems we will also see $O(1)$ algorithms**
 - Example: we will see how to determine the maximum of n numbers in $O(1)$ time using $O(n^2)$ processors
- **Assumptions:**
 - Often we will first only consider the time: given arbitrarily many processors, how fast can the problem be solved.
 - Subsequently, we will try to reduce the number of processors employed so that the algorithm becomes optimal in terms of processors

Few Definitions

- Some important definitions describing the performance and quality of parallel algorithms
- **Time:**
 - **time** taken by a parallel algorithm in solving a problem of size n on a computer with P processors is denoted as
$$T(P, n)$$
- **Cost:**
 - The **cost** $C(P, n)$ of a parallel algorithm solving a problem of size n on a computer with P processors is defined as
$$C(P, n) = P * T(P, n)$$

Few Definitions

- **Work:**
 - The cost is to be distinguished from the **work**
 - The difference is that the work does not take idle time into account
 - Work $W(n, P)$ equals the sum over all processors of the number of performed operations
- **Speed-up:**
 - The **speed-up** $S(P, n)$ gives the number of times the parallel program is faster than the sequential program.
 - For some definition of the sequential time $T(1, n)$
$$S(P, n) = T(1, n) / T(P, n)$$

Few Definitions

- **For $T(1, n)$ there are several possibilities**
- Use of lower bound on the sequential time
 - This notion of $T(1, n)$ is not so nice, because it gives unreasonably small values of the speed-up
- Use of best known sequential algorithm for solving the problem to define $T(1,n)$
 - This is the best definition, because it expresses how much we can really gain by parallelization

Few Definitions

- Some authors defines $T(1, n)$ as the time of their parallel algorithm when used with a single processor.
- **With this notion of $T(1, n)$, the speed-up expresses how the algorithm scales:**
 - *if the speed-up is close to P (or $O(P)$ in theoretical contexts), one says that the algorithm scales well*
 - *If the speed-up is much lower than P (or $O(P)$ in theoretical contexts), one says that the algorithm scales poorly*

Few Definitions

- **Efficiency:**
 - The **efficiency** $E(P, n)$ of an algorithm is related to the speed-up:
 - It is given by
$$\begin{aligned} E(P, n) &= S(P, n) / P \\ &= T(1, n) / (P * T(P, n)) \\ &= C(1, n) / C(P, n). \end{aligned}$$
 - **So, it is the ratio of the costs, whereas the speed-up is the ratio of the times.**

Few Definitions

- Optimal Parallel Algorithm:
 - A parallel algorithm is called **optimal** if $E(P, n) = O(1)$,
 - that is, the cost of the parallel algorithm is at most a constant factor larger than that of the best sequential algorithm
- A study of the efficiency is especially important in practical contexts:
 - Often the efficiency starts to decrease quite strongly from a certain P value on

Few Definitions

- The typical reason for this is the increase of communication cost
 - when P becomes too large for the value of n , the communication costs (which often increase with P^2) become dominating
 - and any further increase of P will hardly lead to a decrease of the overall time.

CS 309: Evaluation Policy

	<i>Weightage (%)</i>
Quiz	20
Mid-semester Exam	30
End-semester Exam	40
Attendance	10
TOTAL	100

CS 359: Evaluation Policy

	<i>Weightage (%)</i>
Lab Assignment	50
Project (40)	
Mid-semester	15
End-semester	25
Attendance	10
TOTAL	100

Schedule for CS 309/359

Day	9:00 - 9:55	10:00 - 10:55	11:00 - 11:55	12:00-12:55	13:00 - 13:55	14:00 - 14:55	15:00 - 15:55	16:00 - 16:55
Monday			CS 309 (L)					
Tuesday				CS 309 (T)		CS 359 (1C 301)		
Wednesday								
Thursday		CS 309 (L)	CS 309 (L)					
Friday			CS 309 (L)					

Lab Structure

- **Project:** an important component of lab
- **Project Problems**
 - Large Matrix Factorization
 - Sparse Matrices
 - Dense matrices
 - Solution for linear equations
 - Solving well known problems like Travelling Salesman Problem (TSP) etc by parallel algorithm
 -

Reference Books

- M. J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, 2002.
- A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, second edition, Addison-Wesley, 2003.
- M. J. Quinn, *Parallel Computing in C with MPI and OpenMP*, McGraw -Hill, 2004.
- Henri Casanova, Arnaud Legrand and Yves Robert, *Parallel Algorithms*, CRC Press, 2010.
- M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw –Hill, 1988.

End

Introduction Part-2

Parallel Computing

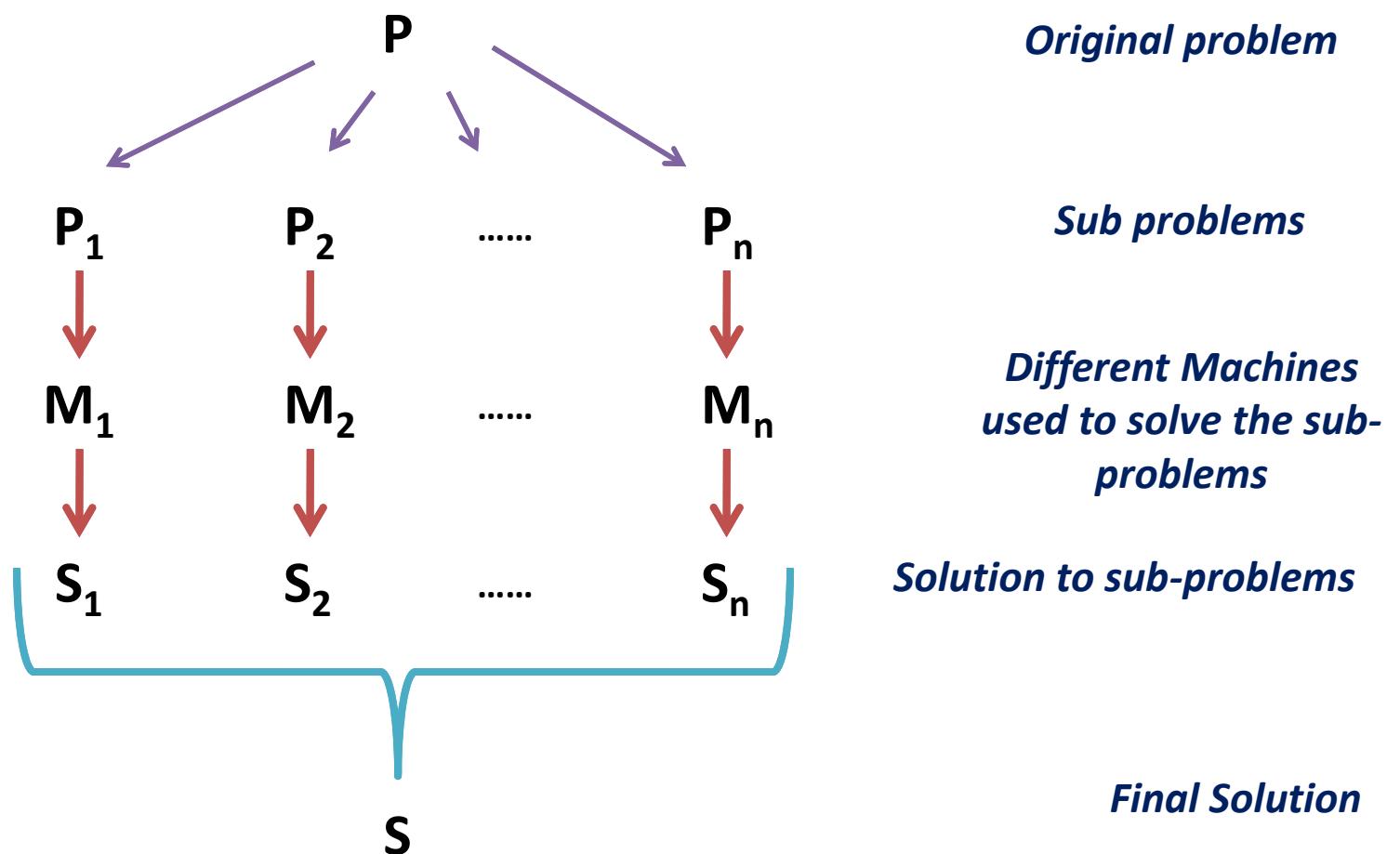
Parallel Computation

- Though processor speed is increasing, a single processor has **some speed limitations**
- On the other hand, computational demand is increasing day-by-day
- **Solution to meet computational demand**
 - One good thing is that the hardware cost is decreasing
 - So use several machines (processors) to solve a problem in place of using a single machine

Parallel Computation

- A problem P is to be solved
- Say we have n machines:
 - M_1, M_2, \dots, M_n
- Divide the problem P into n sub-problems:
 - P_1, P_2, \dots, P_n
- Solve each sub-problem P_i using machine M_i
- Combined the solutions of different machines to get solution to original problem

Parallel Computation



Parallel Computation

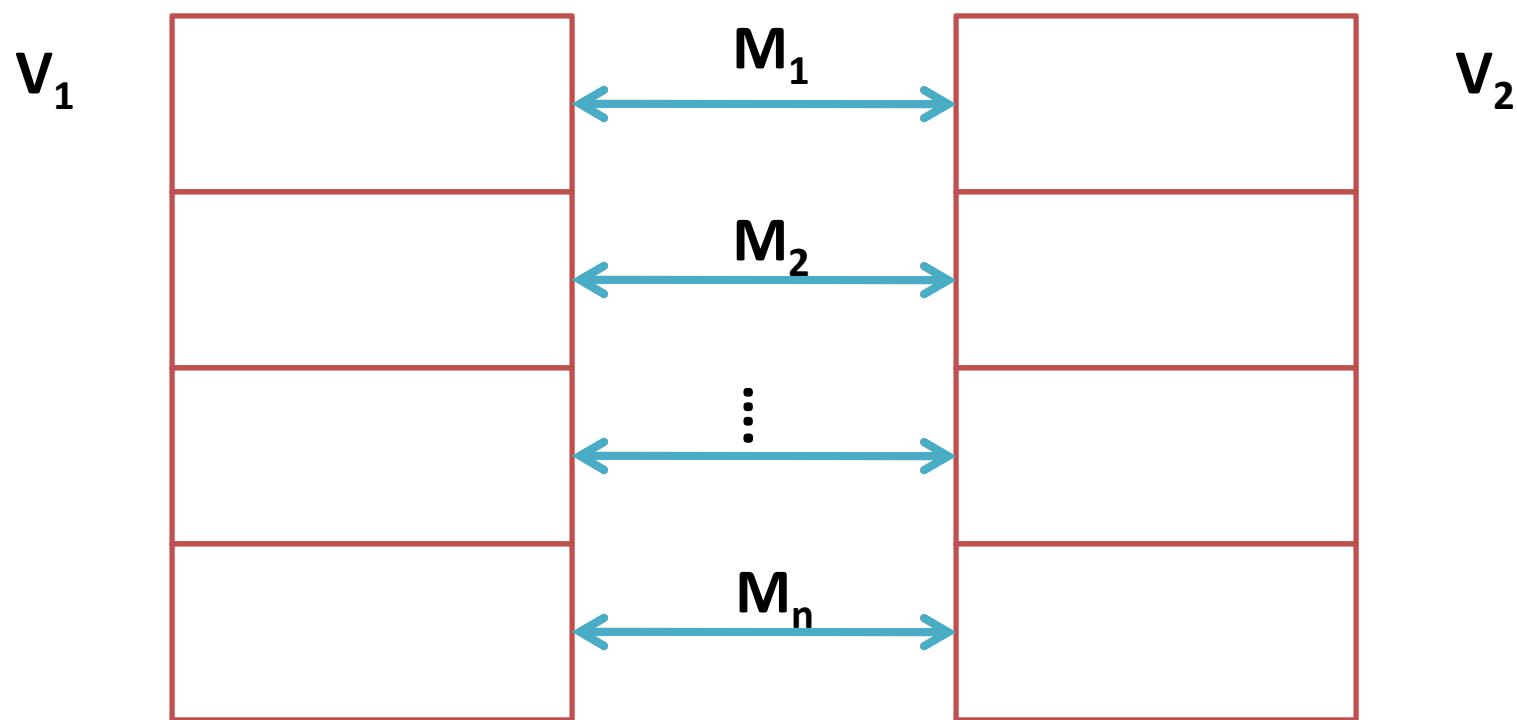
- Parallel Machines
 - Use of several machines to solve a problem
 - Different machines take a different sub-problem and solve simultaneously
 - All the results are combined to get the final solution
 - This idea of solving problem gives the idea of parallel machines
- Parallel Algorithms
 - Algorithms written for parallel machines

Parallel Computation

- **Sequential Algorithm → Parallel Algorithm**
 - Some of them are inherently parallel
 - Writing parallel algorithm for them is easy
 - We can easily divide those problems into sub-problems
 - Sub-problems can be solved by different machines
 - Results all machines can be combined
- **Example**
 - Two vector addition

Parallel Computation

- Addition of two vectors, V_1 and V_2
- Divide vectors into n parts and assign to n different machines (processors)



Parallel Computation

- In reality, we cannot always use available sequential algorithms to solve problem on parallel machine
- Most of the time, we may have to redesign the algorithm for parallel machine
- Goal of this course
 - How to design parallel algorithms for different problems

Model of Sequential Machine

- **In a sequential context:**
 - There is a single widely accepted cost model
 - It is **Von-Neumann** unit-cost model
- **Basic assumption of this model:**
 - All operations are equally expensive
 - Though, It is far from being exact
 - For example: accessing the **main memory** is costly as compared to accessing **registers or cache**

Model of Sequential Machine

- We go with this assumption in theoretical studies
- As long as the complexity of algorithms is given in O()-notation, this is even formally correct

Flynn's Classification

- Whole class of machines can be divided into four classes according to Flynn
- This division is based on
 - **Instruction stream**
 - Set of instructions to be performed by different machines
 - **Data stream**
 - Set of data items to be used by different machines

Flynn's Classification

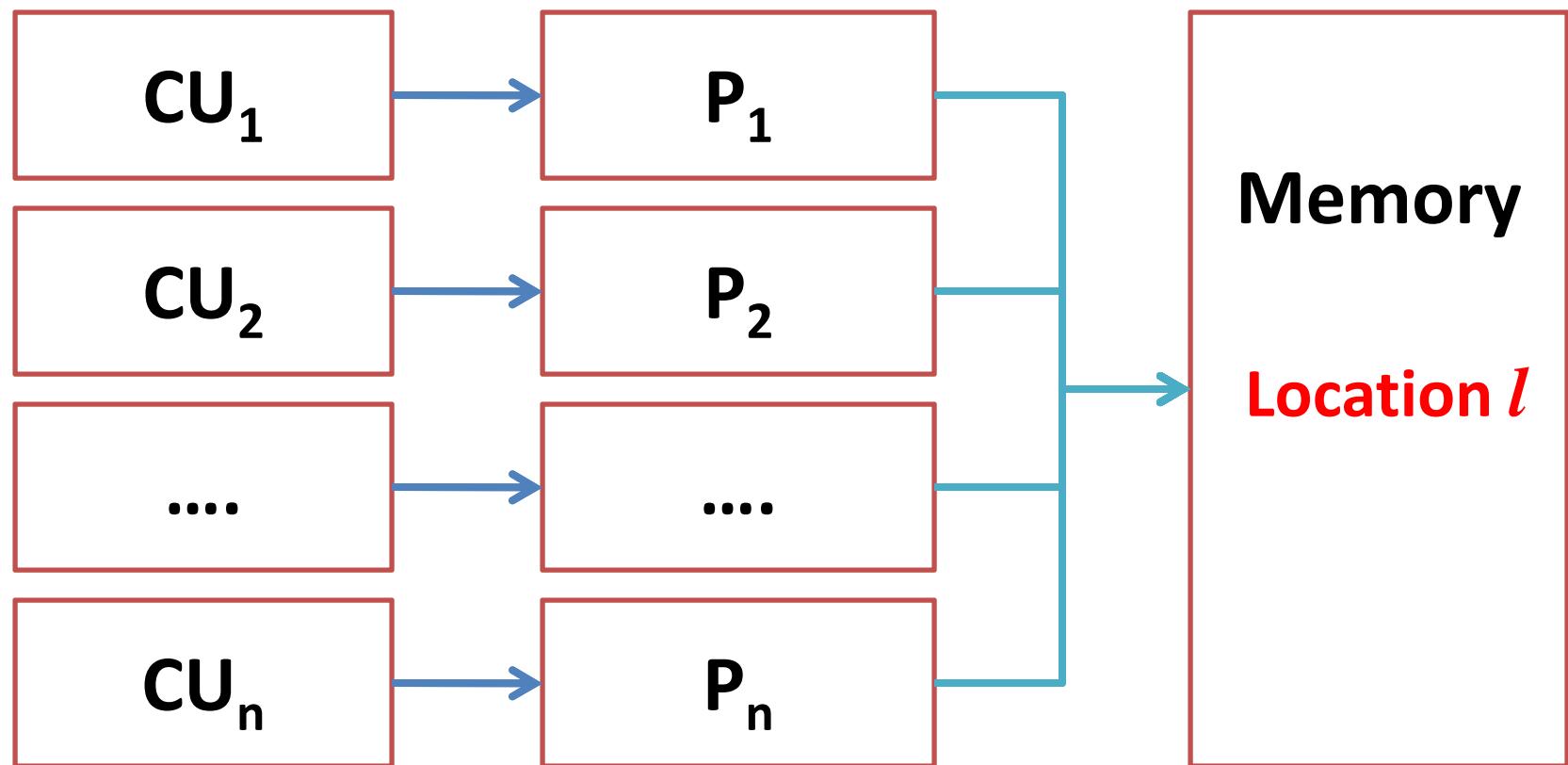
- **Four classes**
 - **SISD**: Single instruction stream, single data stream
 - **MISD**: Multiple instruction stream, single data stream
 - **SIMD**: Single instruction stream, multiple data stream
 - **MIMD**: Multiple instruction stream, Multiple data stream

SISD

- It relates to **normal sequential machine**
- Here we have
 - one CU, one processor and memory
- CU broadcasts the processor to perform stream of instruction, processor gets the data from memory and perform operations and stores the result in the memory



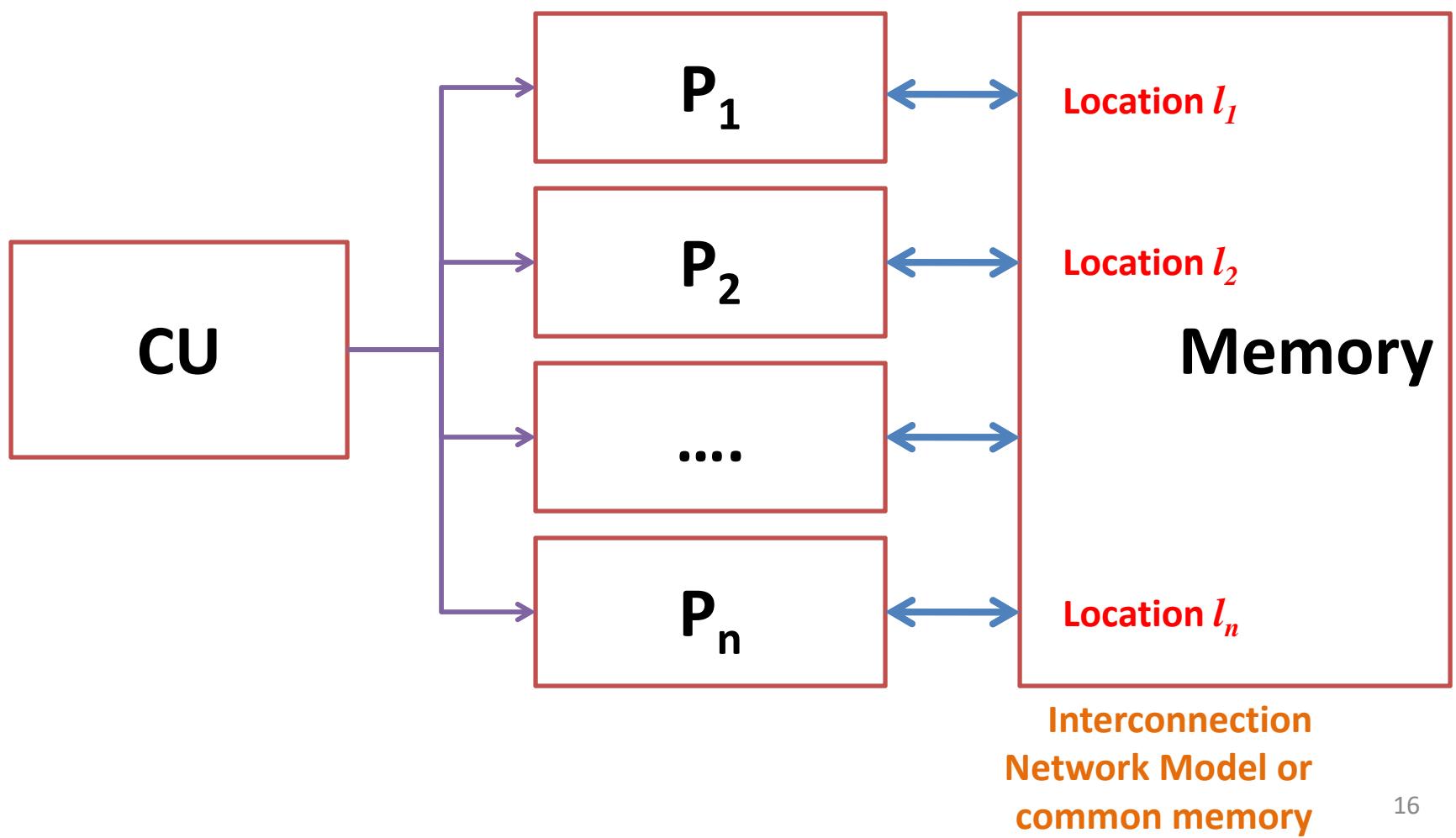
MISD



MISD

- Different control unit generates different instruction
- **Different instructions performed on same data**
 - All these different instructions are performed by processors on same data
- Example: Say data is x and y
 - CU₁ generates addition instruction
 - CU₂ generates subtraction instruction
 - and so on
- **In reality, we rarely need such machine, hence this type of model does not have much use**

SIMD



SIMD (2)

- One CU and Multiple processors
- CU broadcast a single stream of instructions to all the processors
- Processors take the data
 - either from the local memory or from a common memory based on the model of the machine
 - Also, a processor can take data from another processor (in case of network model when processor are connected to each other) and performs the operation

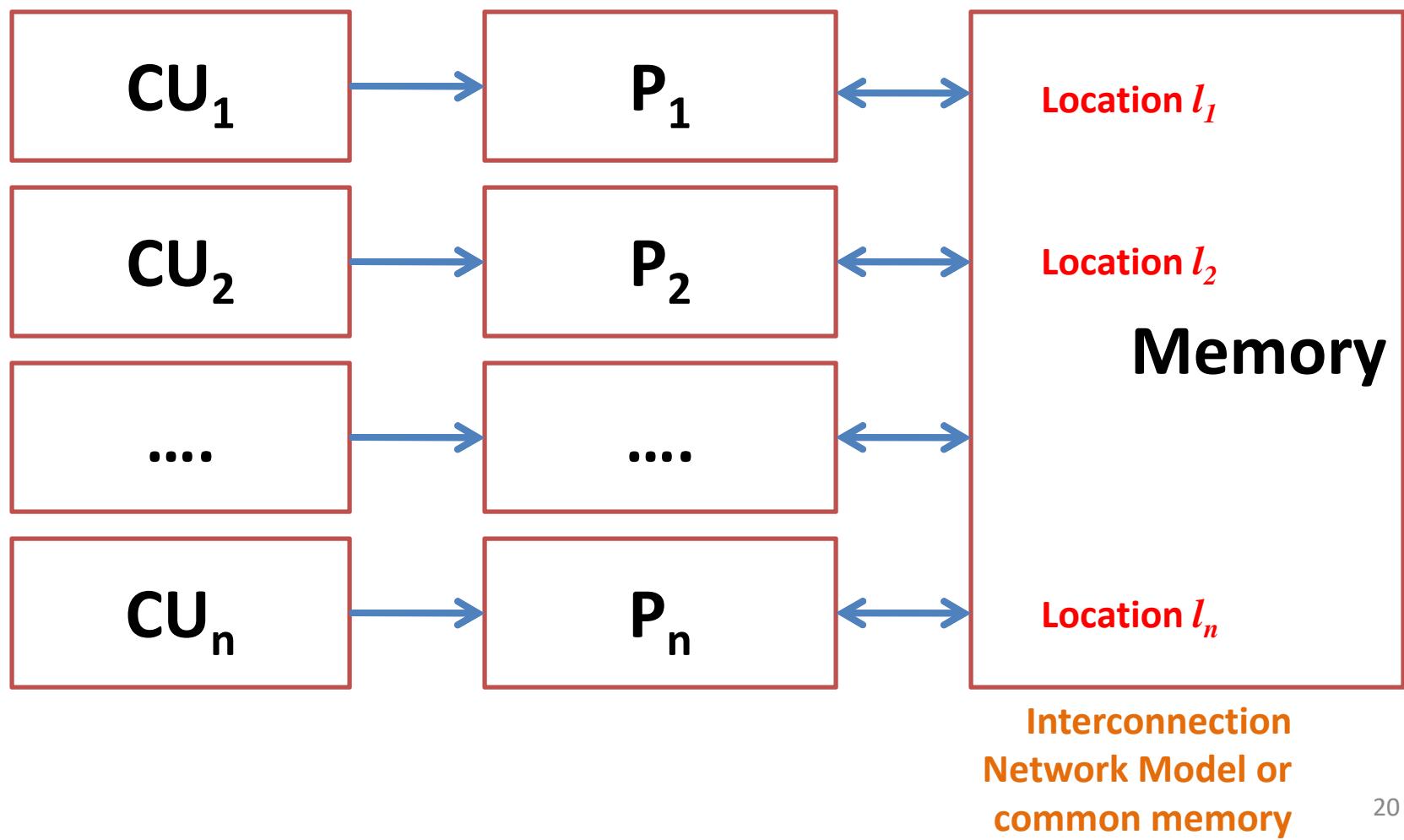
SIMD (3)

- **SIMD machine:**
 - It is synchronous
 - All processors execute the same instruction at all times
 - Since they execute these instructions on different data, it is still possible to obtain parallelism

SIMD (4)

- Idea behind SIMD machines is that if all processors execute the same instruction, there is **no need for the processors to generate these instructions themselves**:
 - Central host computer connected to all processors tells them what instruction to execute next
 - This implies that the PUs can be much simpler

MIMD



MIMD

- Multiple CUs and Multiple processors
- Different CUs broadcast different stream of instructions to their respective processors
- Processors take the data either from the local memory or from a common memory based on the model of the machine
- In case of Interconnection Network Model
 - In case of network model when processors are connected to each other, a processor can take data from another processor and can perform the operation

MIMD

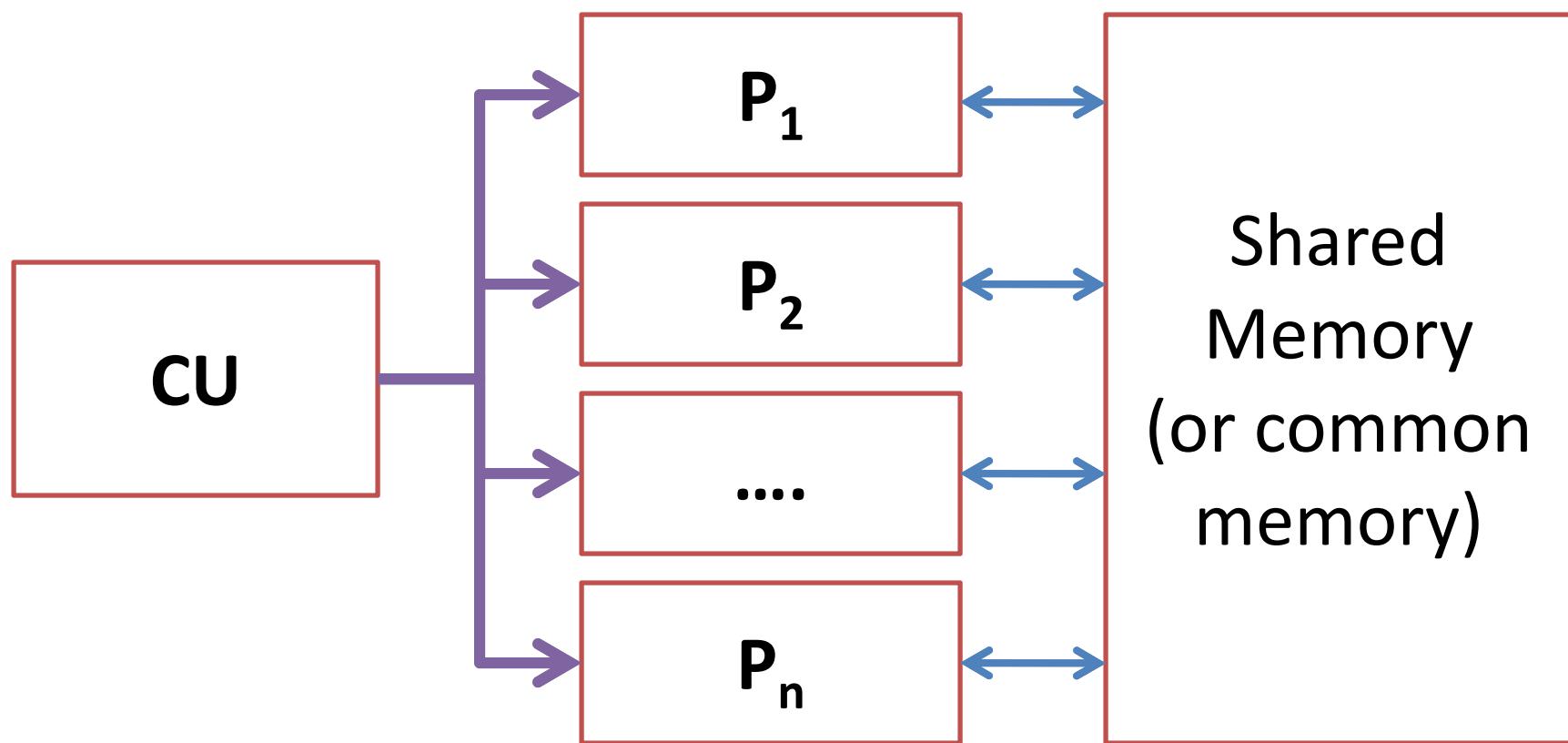
- **MIMD machine:**
 - theoretically, all processors may have their own program
 - in practice, all will execute the same program, but due to conditional instructions, they will not always execute the same instructions
 - all modern parallel computers, in which the CPUs are mostly conventional PC CPUs, are of this type

In this course

- We shall study different algorithms which are
 - mostly based on SIMD and
 - few based on MIMD

Shared Memory Model

- SIMD models which uses shared memory are called Shared Memory Model



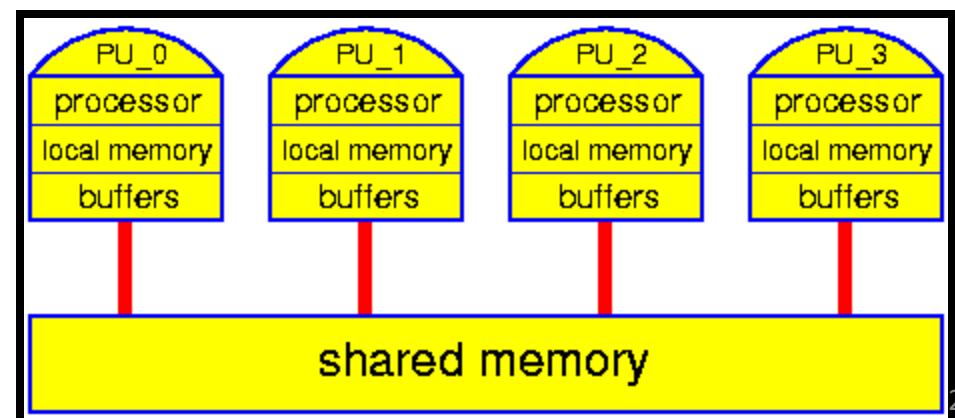
Shared Memory Model

- In Shared Memory model
 - we have n processors
 - Say, P_1, P_2, \dots, P_n
 - Each processor can be made active or inactive by setting the mask
 - All the active processors are allowed to perform operations by taking the data from the common memory and storing the results back in the memory

Shared-Memory Model

- All processors have direct access to a common memory
 - Hence constant time access to the memory
- Local memory: processors have a local memory in which they can store local data

**Shared memory
computers with
4 processors**



Shared Memory Model

- Different shared memory models
 - Division is based on different ways the memory is accessed
- Four types
 - CRCW: Concurrent read concurrent write
 - CREW: Concurrent read Exclusive write
 - ERCW: Exclusive read concurrent write
 - EREW: Exclusive read Exclusive write

Shared Memory Model

- Concurrent Read
 - Two or more processors allowed to read a particular memory location simultaneously
- Exclusive Read
 - No two or more processors allowed to read a particular memory location simultaneously

Shared Memory Model

- Concurrent Write
 - Two or more processors allowed to write at a particular memory location simultaneously
- Exclusive Write
 - No two or more processors allowed to write at a particular memory location simultaneously

Shared Memory Model

- CRCW: Concurrent read concurrent write
 - most flexible, most powerful model
 - it is assumed that in a step, **a position can be read and written by arbitrarily many processors**
 - but difficult to implement
- CREW: Concurrent read Exclusive write
 - more convenient model
 - arbitrarily many processors can read a position while at most one processors can write a position

Shared Memory Model

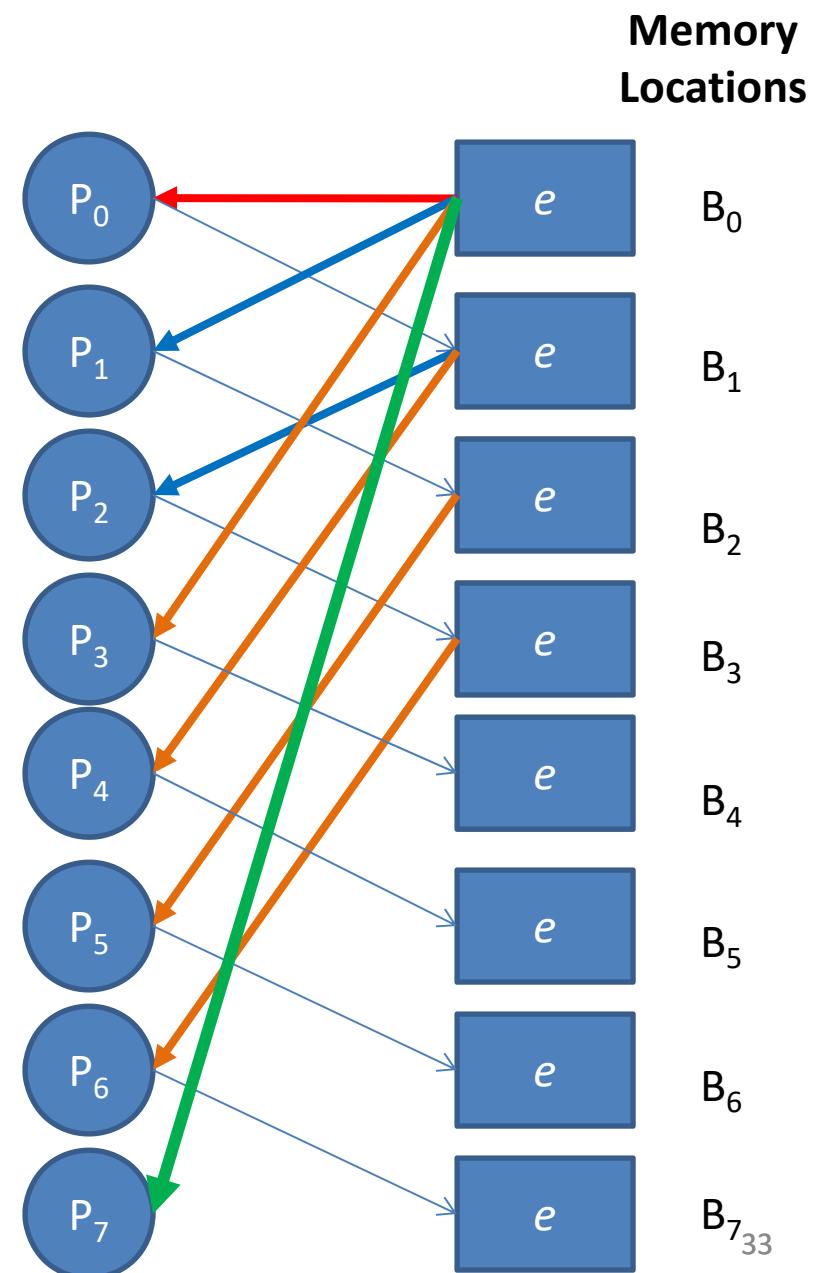
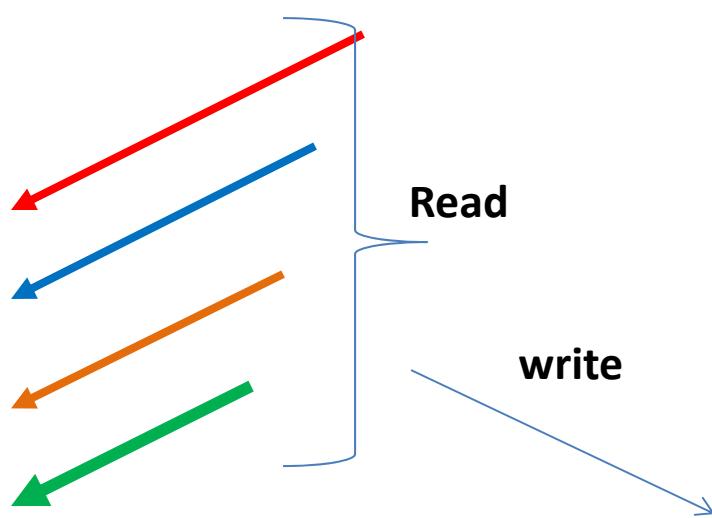
- ERCW: Exclusive read concurrent write (X)
 - not of much use
 - it does not even allow simultaneous read which is quite simple to implement
- EREW: Exclusive read Exclusive write
 - most restrictive model
 - a position in the memory can be accessed by at the most one processor

Concurrent Read

- It can be implemented through broadcasting
 - First one processor reads data from the specified location and makes a copy of it by writing it to a new location
 - Now data is available at two locations and can be read by two new processors simultaneously
 - The two new processors read the data and write at two new locations
 - Now data is available at four locations and can be read by four processors simultaneously
 - so on
- This way n processors can read data in $\log(n)$ time

Concurrent Read

Data to be read is x ,
initially, it is available
only at B_0



Concurrent Read

- In first step one processor reads , i.e. 2^0
- In second step 2 processors read , i.e. 2^1
- In third step 4 processors read, i.e. 2^2
- ...
- In k^{th} step # processors which read, i.e. 2^{k-1}
- If there are n processors to read the data then

$$2^0 + 2^1 + 2^2 + .. + 2^{k-1} = n$$

$$1(2^{k-1})/(2-1) = n$$

$$2^{k-1} = n$$

$$k = \log(n+1) = O(\log n)$$

Shared-Memory Model

- **How to handle write conflicts in CRCW**
- **Solution 1: [values written if they are same]**
 - Common CRCW machine,
 - Here, concurrent writes are only allowed if all written values are equal
 - **Example: Max of n numbers:** We will see an example, how on such a machine the maximum of n numbers can be computed with $O(n)$ work in $O(\log \log n)$ time.
- **Solution 2: Arbitrary value written**
 - Arbitrary CRCW machine
 - Here it is allowed to write different values and one of them will be written
 - The model does not specify which element is written, so the algorithms designer must be prepared for the worst possible choice

Shared-Memory Model

- **Solution 3: Randomly selected one of them is written**
 - **Random CRCW machine**
 - It is allowed to write different values and a randomly selected one of them will be written
 - Algorithms designer may exploit that some kind of average value
- **Solution 4:**
 - **A Maximum CRCW machine**
 - It allows to write different values and the largest of them will succeed

Shared-Memory Model

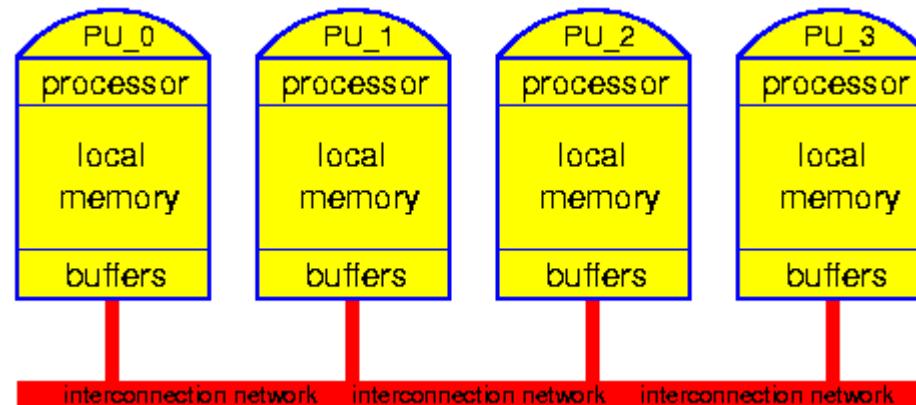
- Depending on the underlying hardware, any of these models may be made realistic
- These models are not very far apart:
 - For example, a step of a Maximum CRCW machine with P processors can be simulated on an EREW machine in $O(\log P)$ steps

Interconnection Network Models

(Distributed-Memory Model)

- It assumes that the processors are interconnected
 - They use some network topology
- **Minor role of topology here**
 - It is assumed that the topology plays a minor role and can thereby be neglected.
- Data exchange happens by communication through this network

**Distributed
Memory
System with 4
Processors**



Interconnection Network Models

(Distributed-Memory Model)

- **Total Cost:**
 - Cost of communication is linear in the amount of data sent or received plus some constant cost
 - So, time for a computation is estimated as follows
$$T_{\text{total}} = c_i * t_i + c_v * t_v + c_p * t_p$$
- Here
 - c_i , c_v and c_p are constants
 - Value of these constants depend on the state of technological development
 - t_i is the **number of internal operations**
 - t_v is the total **amount of data sent by a processor (routing volume)**
 - t_p gives the **total number of packets** sent by a processor

Data Transfer in DMM

- To send data from i^{th} processor to j^{th} processor,
 - A path from first to second is cleared, and then the data are transferred along this path *flit by flit*
 - **Flit** is a small data unit
- The idea is that these flits run through the network like the segments of a worm:
 - First, all of them stands at i^{th} processor, then the first *flit* enters the network

Distributed-Memory Model

- By the time the first flit reaches j^{th} processor, several other flits, depending on the number of **hops** from i^{th} processor to j^{th} processor, have entered in the network.
- In j^{th} processor, flits pile up and are recomposed.
- In this way the transfer time is roughly, something like

$$\textbf{\textit{Transfer time}} = f(d, l)$$

- where, d gives the number of hops from i^{th} to j^{th} processor
- l gives the number of flits.

DMM Cost Revisited

$$T_{\text{total}} = c_i * t_i + [c_v * t_v + c_p * t_p]$$

- the contribution $[c_v * t_v + c_p * t_p]$ to T_{total} is perceived as the loss
- For most applications
 - $t_i(P, n)$ decreases linearly with P
 - $t_p(P, n)$ tends to increase at least linearly with P
- To achieve acceptable efficiency
 - P must not be chosen too large in comparison to n

Distributed-Memory Model

- How to choose P ?
- Depends on
 1. the time complexity of the problem and
 2. the development of $t_p(P, n)$,
- How large P should be chosen need to be decided based on above two factors

Parallel Computing Models - Summary

- Fundamental distinction in parallel computation is on **how the memory is accessed**:
 - **Shared memory**: whether there is a shared memory to which all processors have access in constant time
 - **Distributed memory**: whether there is a distributed memory so that each PU has direct access only to its own local memory

Parallel Computing Models - Summary

- **Shared Memory:**
 - Data is accessed from a common area
 - It is exchanged by "parking" some information at some fixed position making it accessible to the other processors
- **Distributed Memory:**
 - Communication between processors to share/exchange data

More on SIMD Model

- Data Sharing in SIMD model can be done in two ways
 - Using shared memory
 - Using a interconnection network
- In both types of systems, all processors work synchronously
- Parallel algorithm written for these systems is a *synchronous algorithm*

More on SIMD Model

- Use of Shared Memory:
 - All processors have direct access to a common shared memory
 - All memory locations can be accessed by any of the processor
 - Parallel machines based on this Shared Memory SIMD model are commonly called **Shared Memory based Parallel (SMP) machines**

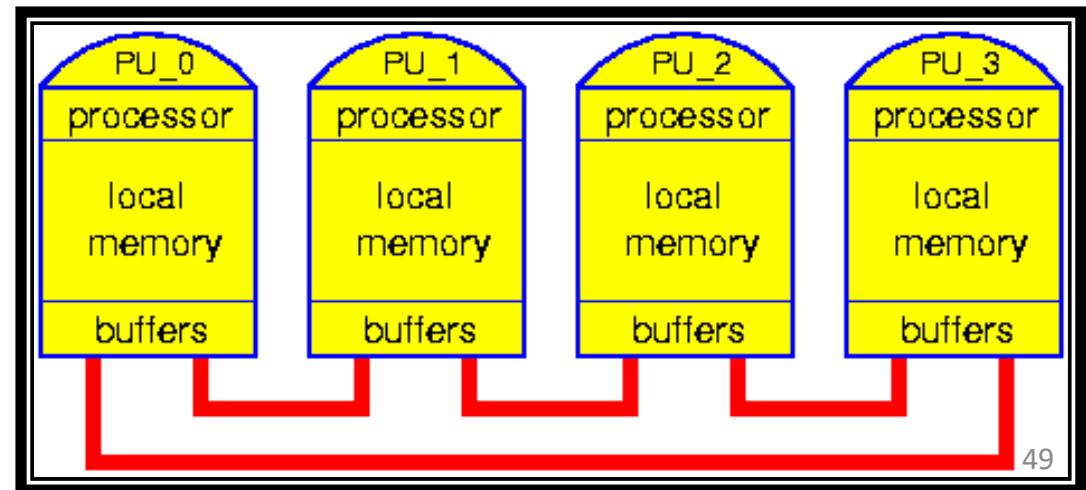
More on SIMD Model

- Use of Interconnection network:
 - Processors are connected to share data
 - Interconnection follows a particular topology
- In next slide we see more details about Interconnection Network based SIMD model

Interconnection-Network based SIMD Model

- Considers a **particular topology**
- Assumes that the structure of the network plays a **significant role**, and
- Algorithms will be designed with the network topology in mind

**Interconnection
Network Model
(connected by ring)**



Interconnection-Network based SIMD Model

- Time Complexity of algorithms
 - It is expressed in terms of steps for interconnection networks
- **A step = Certain computation + Communication**
- No limit on the communication
 - Often there is no limit on the amount of communication that can be performed in a step

Interconnection-Network based

SIMD Model

- **Assumptions about data transfer**
 - One packet of unit size over each link
 - It is assumed that one packet of unit size can be transferred over each connection in each step
 - No division of packets
 - These packets can neither be divided nor composed
 - processors can only communicate with neighbors
 - **longer distance communications**
 - Store and forwarding approach
 - achieved by forwarding packets along some path connecting the source to destination

Interconnection-Network based

SIMD Model

- Various types of Interconnection Network Model:
- **All port Vs. single port model**
 - whether processors can **communicate with all neighbors at the same time (all-port model)** or only with one of them (**single-port model**).
- **Half-duplex:**
 - Sometimes it is specified that the connections are **uni-directional**
 - Processors cannot send to a processors from which it is also receiving
- **Full duplex**
 - Communication to both sides
- **The interconnection-network model neglects the aspects of computation**
 - it allows to concentrate on the fundamental aspects of data exchange

Interconnection-Network based

SIMD Model

- Examples
 - Mesh Connected computers
 - Pyramid Models
 - Shuffle Exchange network..
 - Tree
 - Cube connected Cycle
-

More on MIMD Model

- Data Sharing in MIMD model can be done in two ways
 - Using shared memory
 - Using a interconnection network

More on MIMD Model

- Use of Shared Memory:
 - All processors share data through a shared memory
 - A processor writes at a memory location (at a common area) to share the data and another processor reads it from there
- Shared memory MIMD systems are commonly referred as *multiprocessors*
- Algorithm written for such system is called '*asynchronous parallel algorithm*'

More on MIMD Model

- Two types of MIMD model based on shared memory
 - **Tightly coupled system:**
 - Use of shared memory
 - Distance of a memory location from any of the processor is same
 - **Loosely coupled system:**
 - Use of shared memory
 - Each memory location is close to one of the processor

More on MIMD Model

- Using an interconnection network in MIMD to share data (or for communication)
 - MIMD systems which share data using interconnection network are commonly called Multicomputers
 - Interaction between processors happen through interconnection network using message passing
 - No shared memory
 - **Distributed memory:**
 - Each processor has its own memory
 - Due to use of distributed memory, these systems are also called Distributed Memory Model (or cluster)

More on MIMD Model

- Using an interconnection network in MIMD to share data (or for communication)*contd*
 - Algorithm written for these systems is called *distributed algorithm*
 - Time complexity of a distributed algorithm is analyzed based on instruction execution time + communication overhead

Synchronous Vs. Asynchronous Models

- **Another important distinction**
 - whether the processors are working synchronously or asynchronously
- **Synchronous model:**
 - All processors have access to a central clock
- **Asynchronous model:**
 - Each processing unit has its own speed
 - In practice systems are mostly asynchronous
- **Generally, in a parallel computer not necessarily all processors are of the same type**

Synchronous Vs. Asynchronous Models

- **Speed difference of processors:**
 - **Theoretically**, speed differences can be arbitrarily large
 - **In practice**, it is normally does not make sense to combine processors which are too different

In this course..

- In this course, we shall mostly focus on parallel algorithms for **SIMD model**
 - Shared memory based SIMD (SMP)
 - Interconnection based SIMD
- And few algorithms on
 - Interconnection based MIMD

End

PRAM Model

Parallel Random Access Machine

Part – 1

Computation Models

- **Computation model**
 - Goal is to provide a **realistic representation of the costs of programming**
- Model provides programmers a measure of algorithm complexity
 - It helps them to decide which algorithm is efficient to use

PRAM: Abstract computational model for parallel algorithms

Why do we need a Model?

- We want to develop computational models which, for a program, accurately represent
 - *cost and*
 - *performance*
- If the model is poor, optimum in model may not coincide with optimum observed in practice

The Random Access Machine Model

RAM model of sequential computers:

- Memory is a sequence of words, each capable of containing an integer
- Each memory access takes one unit of time
- Basic operations (add, multiply, compare) take one unit time.
- Instructions are not modifiable
- Read-only input tape, write-only output tape

What about parallel computers

- RAM model for sequential computers
 - It is generally considered a very successful bridging model between programmer and hardware
- Since RAM is so successful, it is generalized for parallel computers ...

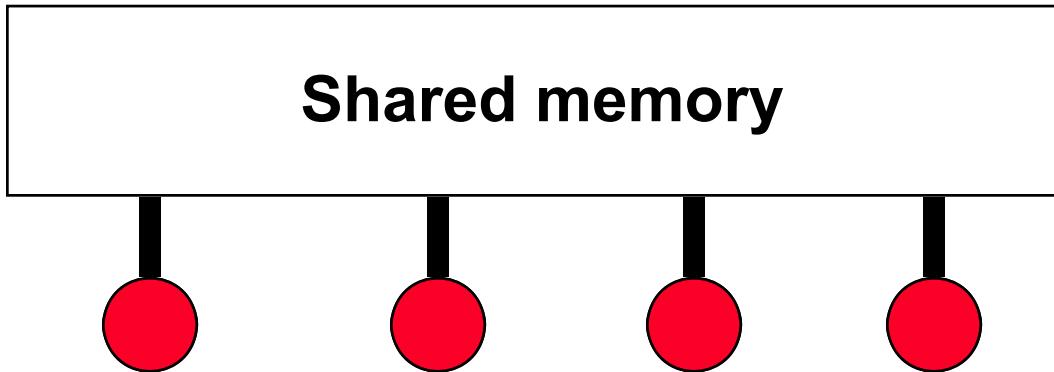
PRAM [Parallel Random Access Machine]

Abstract computational model for parallel algorithms

PRAM composed of:

- p processors, each with its own unmodifiable program
- A single shared memory composed of a sequence of words, each capable of containing an arbitrary integer
- Processors work synchronously
- Shared memory reads and writes
- Each processor has unique id in range 1 to p
- At each unit of time, a processor is either active or idle (depending on id)
- a read-only input tape, a write-only output tape

PRAM model of computation



- p processors, each with local memory
- Shared memory reads and writes
- All processors execute same program
- **At each time step, all processors execute same instruction on different data (“data-parallel”)**
- Focuses on concurrency only
- **PRAM model is a synchronous, SIMD, shared address space parallel computer**
- **It is an idealized model of a shared memory SIMD machine**

Variants of PRAM model

	Exclusive Write	Concurrent Write
Exclusive Read	EREW	ERCW
Concurrent Read	CREW	CRCW

More PRAM taxonomy

- **Different protocols can be used for reading and writing shared memory.**
 - EREW - exclusive read, exclusive write
A program is not allowed to have two processors access the same memory location at the same time.
 - CREW - concurrent read, exclusive write
 - ERCW: Exclusive read concurrent write
 - CRCW - concurrent read, concurrent write
Needs protocol for arbitrating write conflicts
 - CROW – concurrent read, owner write
Each memory location has an official “owner”
- **PRAM can emulate a message-passing machine by partitioning memory into private memories.**

Sub-variants of CRCW

- Common CRCW
 - CW iff all processors writing same value
- Arbitrary/Random CRCW
 - Arbitrary/random value of write set stored
- Priority CRCW
 - Value of min-index processor stored
- Combining CRCW
- Min/Max CRCW
 - Min/Max value out of all requests of processors written

More on PRAM model

- When specifying the performance of a PRAM algorithm
 - we will mostly specify the memory access model,
 - For example, we may say “On a CREW PRAM”
- However, the existence of a local memory is mostly ignored while using PRAM model:
 - We assume as if all the processors operate directly on the shared memory itself
 - Similar to what we do while writing sequential algorithms without mentioning the memory management
- This allows us to concentrate on the fundamental aspects of parallelization

Work-Time Paradigm

- Higher-level abstraction for PRAM algorithms
- WT algorithm = (finite) sequence of time steps with arbitrary number of operations at each step
- Two complexity measures
 - Step complexity $T(n)$
 - Work complexity $W(n)$

WT algorithm *work-efficient* if $W(n) = \Theta(T_S(n))$

optimal sequential
Algorithm

Work and Cost Efficiency

Work-efficient if $W(n) = \Theta(T_S(n))$

Cost-efficient if $C(n) = \Theta(T_S(n))$

Designing PRAM Algorithms

- Common strategies/fundamental techniques used in designing parallel algorithms
 - Balanced trees
 - Pointer jumping
 - Algorithm cascading
 - Euler tours
 - Divide and conquer
 - Symmetry breaking
 - ...

Balanced Trees

- **Key idea:**
 - Build balanced binary tree on input data, sweep tree up and down
- **Tree**
 - It is **not considered as a data structure**, but often a **control structure**
- **Examples of Use**
 - Sum of n numbers
 - Maximum and minimum of n numbers

Pointer Jumping

- Used for various algorithms on lists and trees
- Key idea:
 - In a linked structure, replace a pointer with the pointer it points to
 - Also known as recursive doubling, shortcutting
 - *The name recursive doubling comes from the fact that if all the pointers are jumped at every step, the pointers cover double the distance with each recursion*

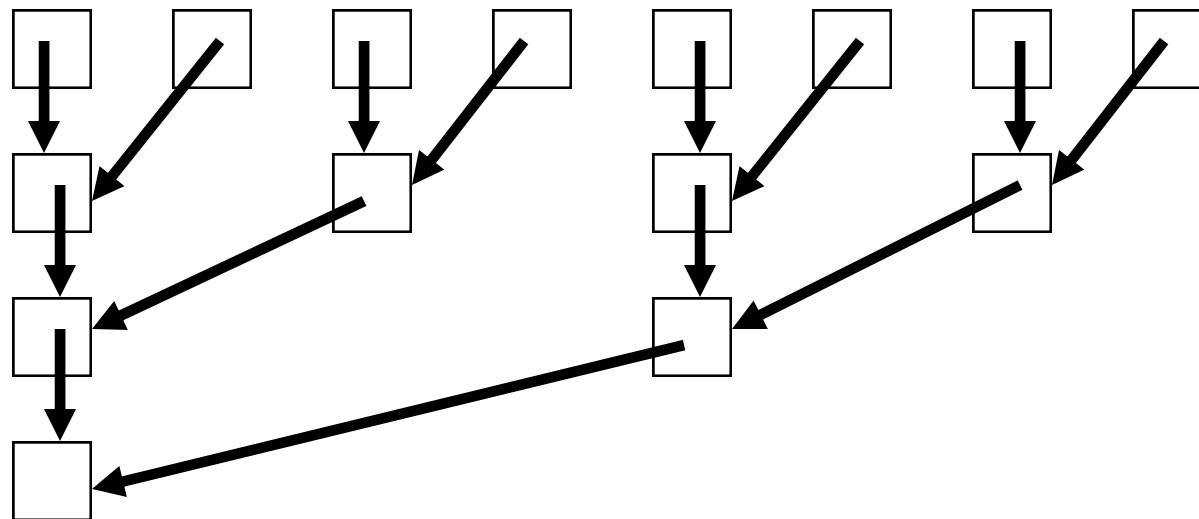
Pointer Jumping

- **Examples where Pointer Jumping is used:**
 - Prefix sum
 - List ranking
 - List packing
 - Finding the roots of forest represented as parent array P

Algorithm Examples on PRAM

Algorithm : Sum

- Given: Sequence a of $n = 2^k$ elements
- Given: Binary associative operator $+$
- Compute: $S = a_1 + \dots + a_n$



WT Description of Sum

```
integer B[1..n]
parfor i = 1 : n do //Concurrent loop
    B[i] = ai
enddo
for h = 1 to k do //Serial loop
    parfor i = 1 : n/2h do //Concurrent loop
        B[i] = B[2i-1] + B[2i] //EREW, synchronization needed
    enddo
enddo
S = B[1]
```

Note that there is no requirement of concurrent read here as i^{th} processor reads data from two locations from where no other processor takes data

// i^{th} processor first reads data from $(2i-1)^{th}$ and $(2i)^{th}$ location (note that these reads are done one after another as a processor can read data from only one location at a time), Once RHS reads are done, addition is performed and result is written at i^{th} location

Note that this modifies the original array, so a copy of the data need to be made if original data to be retained

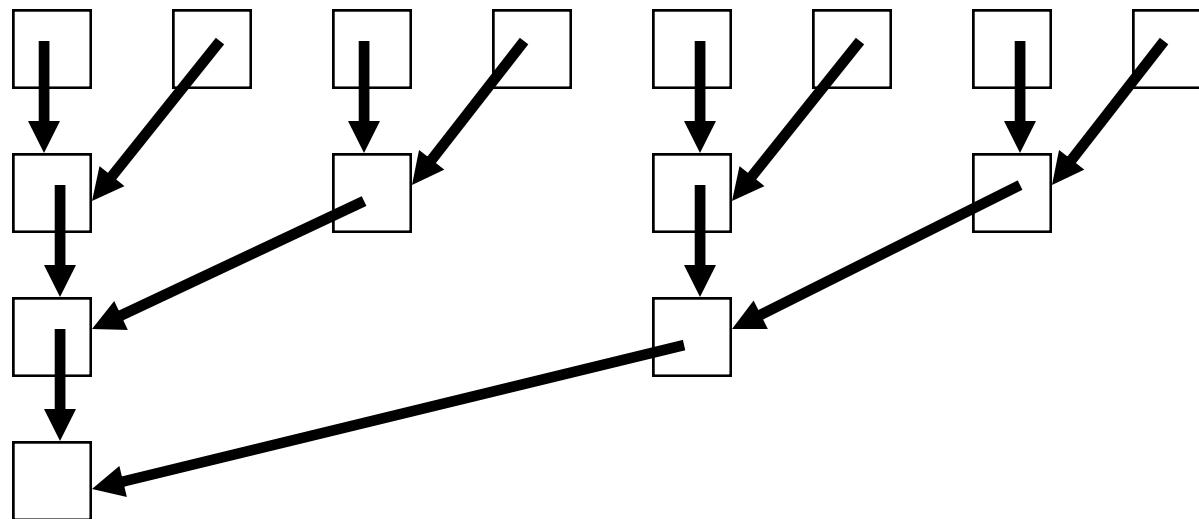
End

Algorithm Examples on PRAM

Part-2

Algorithm : Sum

- Given: Sequence a of $n = 2^k$ elements
- Given: Binary associative operator $+$
- Compute: $S = a_1 + \dots + a_n$



WT Description of Sum

```
integer B[1..n]
parfor i = 1 : n do //Concurrent loop
    B[i] = ai
enddo
for h = 1 to k do //Serial loop
    parfor i = 1 : n/2h do //Concurrent loop
        B[i] = B[2i-1] + B[2i] //EREW, synchronization needed
    enddo
enddo
S = B[1]
```

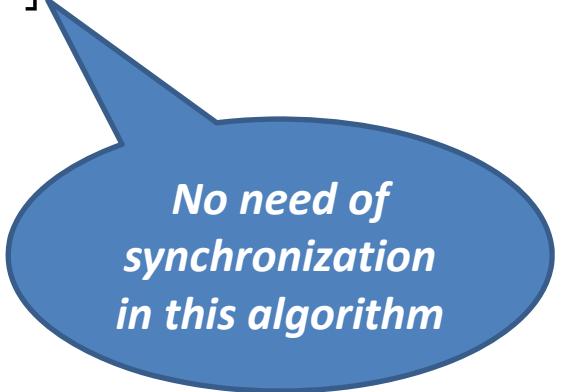
Note that there is no requirement of concurrent read here as i^{th} processor reads data from two locations from where no other processor takes data

// i^{th} processor first reads data from $(2i-1)^{th}$ and $(2i)^{th}$ location (note that these reads are done one after another as a processor can read data from only one location at a time), Once RHS reads are done, addition is performed and result is written at i^{th} location

Note that this modifies the original array, so a copy of the data need to be made if original data to be retained

Alternate Algorithm

```
integer B[1..n]
parfor i = 1 : n do      //Concurrent loop
    B[i] = ai
enddo
for h = 1 to k do        //Serial loop
    parfor i = 1 : n/2h do      //Concurrent loop
        B[i] = B[i] + B[i+ n/2h]
    enddo
enddo
S = B[1]
```



No need of synchronization in this algorithm

Points to note about WT program

- Contains both serial and concurrent operations
- Semantics of *parfor*
- Order of additions
 - It is different from sequential order: associativity critical

Analysis of operations of Sum

- $\Theta(\log n)$ steps, $\Theta(n)$ work
- EREW model
- Two variants
 - Inclusive: as discussed
 - Exclusive: $s_1 = I$, $s_k = x_1 + \dots + x_{k-1}$
- If n not power of 2, pad to next power

Complexity Measures of Sum

```
integer B[1..n]
```

```
parfor i = 1 : n do } //Concurrent loop O(1)  
    B[i] = ai  
enddo
```

```
for h = 1 to k do
```

```
    parfor i = 1 : n/2h do } //Concurrent  
        B[i] = B[2i-1] + B[2i]  
    enddo
```

```
Enddo
```

```
S = B[1]
```

Complexity Measures of Sum

Time complexity for Parallel Algorithm

$$T(n) = 1 + k + 1 = \Theta(\lg n)$$

Work complexity for Parallel Algorithm

$$W(n) = n + \sum_{h=1}^k \frac{n}{2^h} + 1 = \Theta(n)$$

Total cost for Parallel Algorithm

$$\text{cost} = nO(\log n) = O(n \log n)$$

- Recall definitions of step complexity $T(n)$ and work complexity $W(n)$
- Concurrent execution reduces number of steps

Is the algorithm work and cost efficient?

- Sequential time complexity for this problem
 - $T_s(n) = O(n)$
- Work Complexity: $W(n)=O(n)$
 - Work efficient:
 - Yes
 - Justification: $W(n)= O(T_s(n))$
- Cost: $C(n) = O(n \log n)$
 - Cost efficient:
 - No
 - Justification: $C(n) \neq O(T_s(n))$

List Ranking

- **List Ranking Problem**
 - Given a singly linked list L with n objects, for each node, compute the distance to the end of the list
- If d denotes the distance
 - $d[i] = \begin{cases} 0 & \text{if } next[i] = nil \\ d[next[i]] + 1 & \text{if } next[i] \neq nil \end{cases}$
- Complexity of Sequential algorithm:
 - $O(n)$
 - How?

List Ranking

Parallel Algorithm

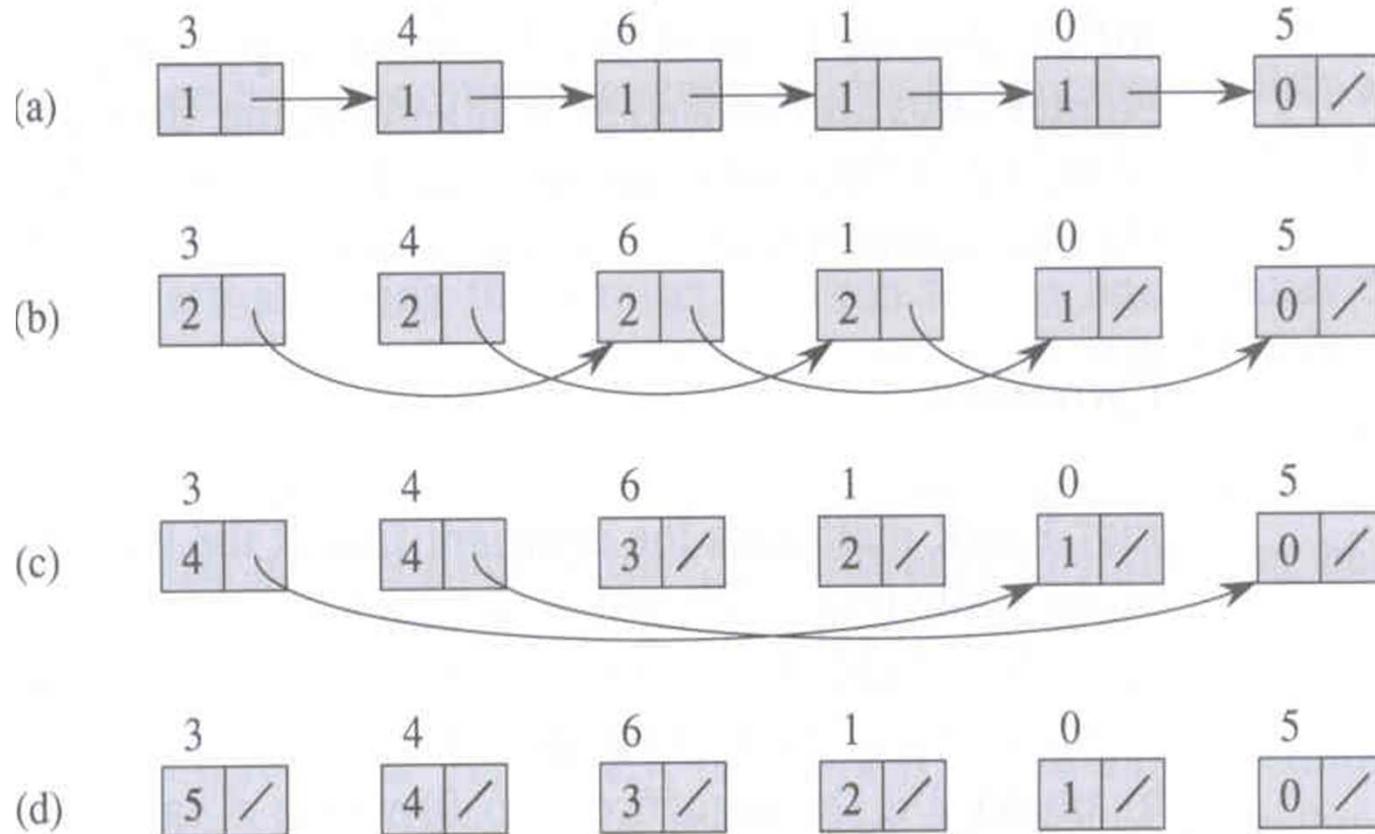
- *List stored in an array*
- Assign one processor to each node
- Assume there are as many processors as list objects
- For each node i , perform
 1. $d[i] = d[i] + d[next[i]]$
 2. $next[i] = next[next[i]]$ // pointer jumping

List Ranking - Pointer Jumping

LIST-RANK(L) (in $O(\log n)$ time) [EREW model]

1. **for** $i=1$ to n , **in parallel**
 2. **if** $next[i]=nil$
 3. $d[i] \leftarrow 0$
 4. **else**
 5. $d[i] \leftarrow 1$
 6. **while** there exists an object i such that $next[i] \neq nil$
 7. **for** $i=1$ to n , **in parallel**
 8. **if** $next[i] \neq nil$
 9. { $d[i] \leftarrow d[i] + d[next[i]]$
 10. $next[i] \leftarrow next[next[i]]$
 11. }

List Ranking - Example



List Ranking - Discussion

- Synchronization is important
 - Consider step 10: $(next[i] \leftarrow next[next[i]])$
 - Here, all processors must read right hand side before any processor write left hand side
- The list ranking algorithm is EREW
 - Consider step 9 ($d[i] \leftarrow d[i] + d[next[i]]$)
 - Here value of $d[i]$ is read by two processors, one by i^{th} processor and another by j^{th} processor (where $next[j] = i$)
 - Since model is EREW
 - First all processors read their own $d[i]$ and
 - then read $d[next[i]]$
 - That means, if $next[j] = i$, then i and j do not read $d[i]$ concurrently

List Ranking - Pointer Jumping

- **Analysis**

- After a pointer jumping, a list is transformed into two (interleaved) lists
- After that, four (interleaved) lists
- Each pointer jumping doubles the number of lists and halves their length
- After $\lceil \log n \rceil$ iteration of *while* loop, all lists contain only one node
- Total time or time steps: $O(\log n)$

Time, Cost, work
Work efficient?, Cost efficient??

List Ranking - Discussion

- Time Complexity
 - $O(\log n)$
- Cost
 - $O(n \log n)$ as processors used are n and time required by the algorithm is $O(\log n)$
- Work performance
 - performs $O(n \log n)$ work since n processors works for $O(\log n)$ time

List Ranking - Discussion

- **Work efficient**
 - A PRAM algorithm is work efficient *w.r.t* another algorithm if two algorithms are within a constant factor
 - Is the list ranking algorithm work-efficient *w.r.t* the serial algorithm?
 - No, because $O(n \log n)$ versus $O(n)$
- Speedup
 - $S = n / \log n$

Applications of List Ranking

- Expression Tree Evaluation
- Parentheses Matching
- Tree Traversals
- Euler tour of trees
- Ear–Decomposition of Graphs
- - - - many others

More on - List Ranking

- **The list ranking problem** was posed by Wyllie (1979), who solved it with a parallel algorithm using
 - logarithmic time ($\log n$) and
 - $O(n \log n)$ total steps (that is, $O(n)$ processors)
- **Subsequent researches**
 - Vishkin 1984; Cole & Vishkin 1989; Anderson & Miller 1990
 - In subsequent papers stated above, this was improved to linearly many steps (using $O(n/\log n)$ processors), on EREW-PRAM
 - In this number of steps matches the sequential algorithm

More on - List Ranking

- List ranking can equivalently be viewed as performing a prefix sum operation on a given list, where values to be summed are all equal to 1
- The list ranking problem can be used to solve many problems on trees via an Euler tour technique (Tarjan & Vishkin, 1985)
 - For this, one forms a linked list that includes two copies of each edge of the tree, one in each direction, places the nodes of this list into an ordered array using list ranking, and then performs prefix sum computations on the ordered array
- **Example: computation of height of each node**
 - For instance, the height of each node in the tree may be computed by an algorithm of this type in which the prefix sum adds 1 for each downward edge and subtracts 1 for each upward edge.

PRAM Model (Part 3)

Prefix Sum of a List

- **Input:**
 - Sequence x of $n = 2^k$ elements, binary associative operator $+$
- **Output:**
 - Sequence s of $n = 2^k$ elements, with $s_k = x_1 + \dots + x_k$
- **Example:**

$x = [1, 4, 3, 5, 6, 7, 0, 1]$

$s = [1, 5, 8, 13, 19, 26, 26, 27]$

Parallel Prefix Sum on a List

- **Prefix computation**
 - Input $\langle x_1, x_2, \dots, x_n \rangle$, a binary, associative operator \otimes ,
[assume $n = 2^k$]
 - Output $\langle y_1, y_2, \dots, y_n \rangle$, where $y_k = x_1 \otimes x_2 \dots \otimes x_k$
- **Example-1**
 - if $x_k = 1$ for $k=1..n$ and $\otimes = +$
 - Then $y_k = k$, for $k = 1..n$
- **Example-2**
 - $x = [1, 4, 3, 5, 6, 7, 0, 1]$
 - $y = [1, 5, 8, 13, 19, 26, 26, 27]$
- **Complexity of Sequential algorithm:**
 - $O(n)$

Parallel Prefix Sum on a List

- **Notation**
 - $[i, j] = x_i \otimes x_{i+1} \otimes \dots \otimes x_j$
 - $[k, k] = x_k$
 - $[i, k] \otimes [k+1, j] = [i, j]$
- Idea: perform prefix computation on a linked list so that
 - each node k contains $[k, k] = x_k$ initially
 - finally each node k contains $[1, k] = y_k$

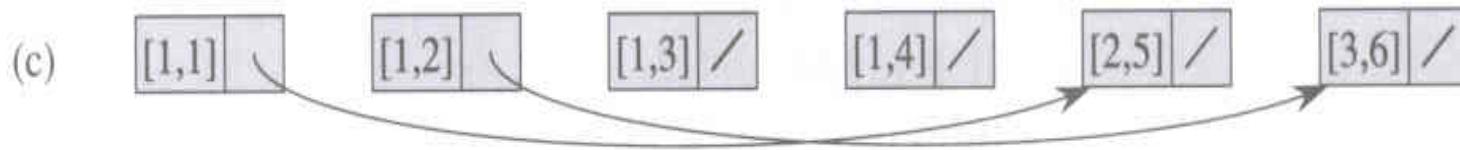
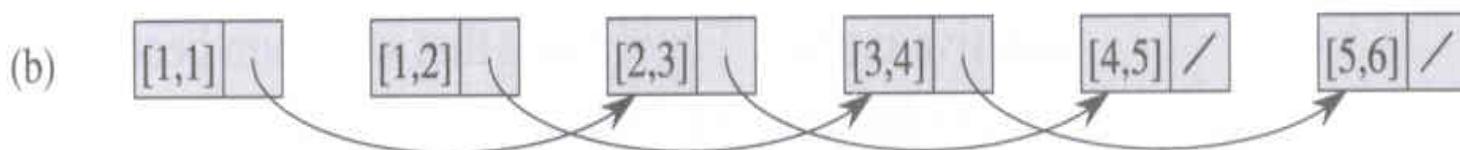
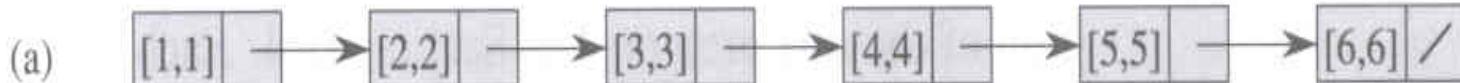
Parallel Prefix on a List (2)

- *List_prefix(L, X)*

// L : list, $X: \langle x_1, x_2, \dots, x_n \rangle$

1. *for each node i , in parallel*
 2. { $y[i] = x_i$ }
 3. *While exists a node i such that $\text{next}[i] \neq \text{nil}$ do*
 4. *for each node i , in parallel*
 5. { *if $\text{next}[i] \neq \text{nil}$ then*
 6. { $y[\text{next}[i]] = y[i] \otimes y[\text{next}[i]]$ // i updates its successor
 7. $\text{next}[i] = \text{next}[\text{next}[i]]$
 8. }
 9. }

Parallel Prefix on a List (3)



Parallel Prefix on a List (4)

- **Analysis**
 - Initially k -th node has $[k,k]$ as y-value, points to $(k+1)$ -th node
 - At the first iteration,
 - k -th node fetches $[k+1,k+1]$ from its successor and
 - perform $[k,k] \otimes [k+1,k+1]$ resulting in $[k,k+1]$ and
 - *update its successor*
 - At the second iteration
 - k -th node fetches $[k+1,k+2]$ from its successor and
 - perform $[k-1,k] \otimes [k+1,k+2]$ resulting in $[k-1,k+2]$ and
 - update its successor

Parallel Prefix on a List (5)

- Running time: $O(\log n)$
 - After $\lceil \log n \rceil$, all lists contain only one node
- Work performed: $O(n \log n)$
- Is the prefix-sum algorithm work-efficient w.r.t the serial algorithm?
 - No, because $O(n \log n)$ versus $O(n)$
- Speedup
 - $S = n / \log n$

List Packing

- **Problem:**
 - Consider an array of upper and lower case letters
 - Delete the lower case letters and compact the upper case to the low-order end of the array

1	2	3	4	5	6	7	8
a	G	H	i	n	w	b	N
G	H	w	N				

It is an Application of Prefix Sums

List Packing

- Implementation via Prefix Sum
 - Assign 1 to items to be packed and 0 to items to be deleted.
 - Perform the prefix sums on the 1/0
 - If upper case, store in location = sum, otherwise do nothing
 - Last sum provides number of packed items

List Packing

1	2	3	4	5	6	7	8
a	G	H	i	n	w	b	N
0	1	1	0	0	1	0	1
0	1	2	2	2	3	3	4
G	H	W	N				

Suppose: don't know how 0/1 assigned. Can I still pack?

Finding Max – in EREW PRAM

- What is the time complexity for the Exclusive-write?
 - Initially elements “think” that they might be the maximum
 - First iteration: For $n/2$ pairs, compare.
 - $n/2$ elements might be the maximum.
 - Second iteration: $n/4$ elements might be the maximum.
 - $\log n$ th iteration: one element is the maximum.
 - So Fast_max with Exclusive-write takes $O(\log n)$.
- **Parallel Time complexity**
 - $T_p = O(\log n)$ **(use of EREW-PRAM)**

Finding Max – in CRCW PRAM

- **Finding max problem**

- Given an array of n elements,
find the maximum(s)
- sequential algorithm is $O(n)$

- **Data structure for parallel
algorithm**

- Array $A[1..n]$
- Array $m[1..n]$. $m[i]$ is true if $A[i]$
is the maximum
- Use of n^2 processors

		$A[j]$					
		5	6	9	2	9	m
$A[i]$	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
	9	F	F	F	F	F	T
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T

$\max 9$

Finding Max – in CRCW PRAM

- **Fast_max(A, n)**
 1. *for i = 1 to n do, in parallel*
 2. *m[i] = true // A[i] is potentially maximum*
 3. *for i = 1 to n, j = 1 to n do, in parallel*
 4. *if A[i] < A[j] then*
 5. *m[i] = false*
 6. *for i = 1 to n do, in parallel*
 7. *if m[i] = true then max = A[i]*
 8. *return max*
- **Time complexity: O(1) [use of CRCW-PRAM]**

Finding Max – in CRCW PRAM

- Concurrent-write
 - In step 4 and 5, processors with $A[i] < A[j]$ write the same value ‘false’ into the same location $m[i]$
 - This actually implements $m[i] = (A[i] \geq A[1]) \wedge \dots \wedge (A[i] \geq A[n])$
- Is this work efficient?
 - No, n^2 processors in $O(1)$
 - $O(n^2)$ work vs. sequential algorithm is $O(n)$

Finding Max – in CRCW PRAM

- Work-time complexity
 - First parallel for loop: n comparisons
 - Second parallel for loop: n^2 parallel comparisons
 - Third parallel for loop: n comparisons
 - Total work = $n + n^2 + n = O(n^2)$
- Time complexity
 - First parallel for loop: $O(1)$
 - Second parallel for loop: $O(1)$
 - Third parallel for loop: $O(1)$
 - Overall complexity= $O(1)$

Brent Theorem

- **Theorem:**
 - Let A be an algorithm with
 - m operations that
 - runs in time $O(t)$
 - on some PRAM (with some number of processors).
 - It is possible to simulate A
 - in time $O(t + m/p)$ on
 - a PRAM of same type with p processors

Brent Theorem-Example

- **Example: maximum of n elements on an EREW PRAM**
 - It can be done in $O(\log n)$ with $O(n)$ processors
 - What happens if we have fewer processors? Say p
 - By the theorem, with p processors, one can simulate the same algorithm in time $O(\log n + n / p)$
[number of operations here would be $m=(n/2)+(n/4)+..1=n-1=O(n)$]
 - If $p = n / \log n$, then we can simulate the same algorithm in $O(\log n + \log n) = O(\log n)$ time, which has the same complexity!
 - This theorem is useful to obtain lower-bounds on number of required processors that can still achieve a given complexity.

Designing of Work Optimal Algorithms

Part-1

Introduction

- Sometimes, a parallel algorithm may be optimal in time but may not be optimal in work
- Similarly, there may be an algorithm which is optimal in work however, not optimal in time
- We will see designing of few work optimal algorithms here
- Common techniques used are
 - Tricks to reduce the number of inputs (*Ex. Finding sum of n numbers*)
 - Accelerated Cascading: (*Ex. Finding max of n numbers*)
 - Independent sets (*Ex. List ranking algorithms*)

Technique - 1

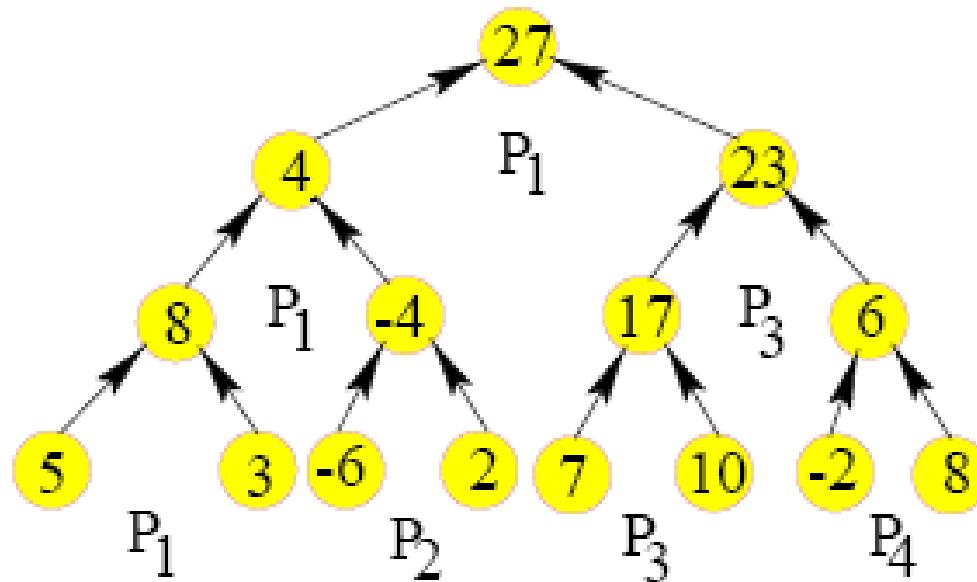
- Designing of work/cost optimal algorithm by using trick to reduce the number of inputs
- Example:
 - Sum of n numbers

Sum of n Numbers

- We select number of processors intelligently to get work-optimal solution
- Sequential algorithm to sum n numbers
 - Time complexity: $O(n)$
- Simple parallel algorithm using n processors
 - Time $T_p(n) = O(\log n)$
 - Work: $W(n) = O(n \log n)$

Simple Parallel Algorithm

- Adding n numbers in parallel
 - Use of n processors
 - Time $T_p(n) = O(\log n)$
 - Work: $W(n) = O(n \log n)$



A work-optimal algorithm for adding n numbers

- **Reduce the input size from n to $(n/\log n)$**
 - To do this divide the n numbers among the $(n/\log n)$ processors, each processor given $\log n$ numbers
- **Step 1**
 - Use only $(n/\log n)$ processors and assign $(\log n)$ numbers to each processor.
 - Each processor adds $(\log n)$ numbers sequentially in $O(\log n)$ time.

A work-optimal algorithm for adding n numbers

- **Step 2**
 - We have only *($n/\log n$) numbers left*
 - *We now execute our original algorithm on these ($n/\log n$) numbers*
- Time computation
 - Time to sum $(\log n)$ numbers: $O(\log n)$
 - Time to find parallel sum:
$$\log(n/\log n) = \log n - \log \log n = O(\log n)$$
- **Hence, Now:**
 - $T(n) = O(\log n)$
 - $W(n) = O(n/\log n \times \log n) = O(n)$

Important Note

- This kind of trick does not work for all kinds of problems
- We will see that we have to really work hard to get optimum work-time algorithm for a problem
- For that we have to analyze the problem carefully, have to extract new properties of it in designing work-optimal algorithm

Technique - 2

- Designing of work/cost optimal algorithm by using accelerated cascading
- Example:
 - Max of n numbers

Maximum of n Numbers

- Use of accelerated cascading to design work optimal algorithm
- Accelerated Cascading:
 - Mixing of different algorithms to get over all good performance
 - Idea is to take the best from two or more algorithms and match them in a way so that we get a better algorithm both in terms of work and in time.

Accelerated Cascading

- Study of Accelerated cascading:
 - We shall study the Accelerated Cascading using an example as the way we merge two algorithm depends very much on what problem we are solving.
- We can get general idea through this example

CRCW Algorithm to get Max of n numbers - *revisited*

- Let us first see CRCW Algorithm to get Maximum of n numbers
- This is a **very fast algorithm** to get maximum of n numbers
- We have seen this algorithm takes $O(1)$ time when $O(n^2)$ processors are used

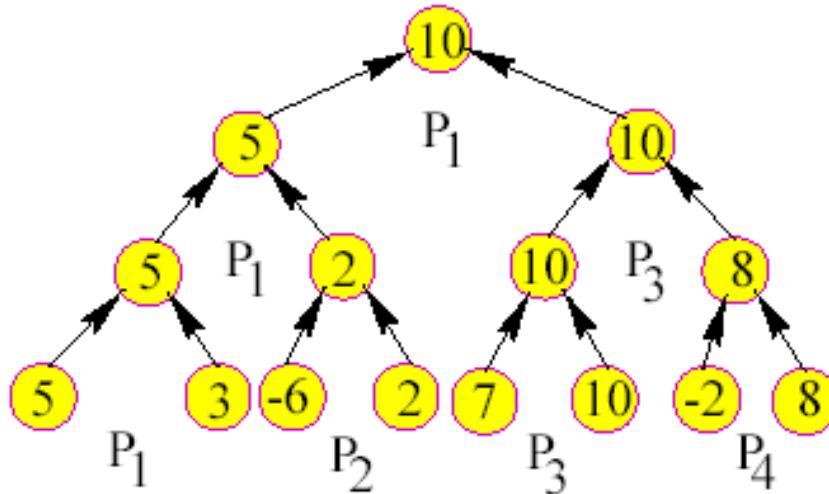
CRCW Algorithm to get Max of n numbers - *revisited*

- We see that this algorithm has
 - Time complexity: $O(1)$
 - Work complexity: $O(n^2)$ – *not work optimal*
- Though this algorithm is good in terms of time, it is not good in terms of work
- This is because sequentially we can find out the maximum in
 - $O(n)$ work and
 - $O(n)$ time

CRCW Algorithm to get Max of n numbers - *revisited*

- Now if this algorithm is not good
 - Then why should we study it?
- Actually we can merge this algorithm with another simple algorithm to get a better algorithm
- **We study that algorithm next**
 - It is based on binary tree type of computation
 - Elements are compared and maximum is assigned to parent node, maximum element reaches the root

Optimal Computation of Maximum



- This is the same algorithm which we used for adding *n* numbers

Optimal Computation of Maximum

- This algorithm takes $O(n)$ processors and $O(\log n)$ time
- We can reduce the processor complexity to $O(n / \log n)$
 - We have seen how to do that
- Hence the algorithm does optimal $O(n)$ work

Optimal Computation of Maximum

- Now we have two algorithms
 - Algorithm 1: Time: $O(1)$, Work: $O(n^2)$
 - Algorithm 2: Time: $O(\log n)$, Work: $O(n)$
- Now we want to merge these two algorithms so that we get
 - Work: $O(n)$ *a work optimal algorithm*
 - Time better than $O(\log n)$ if possible
- **Actually, we shall be able to improve only on time, but will not get work-optimal algorithm (we will see this)**

Optimal Computation of Maximum

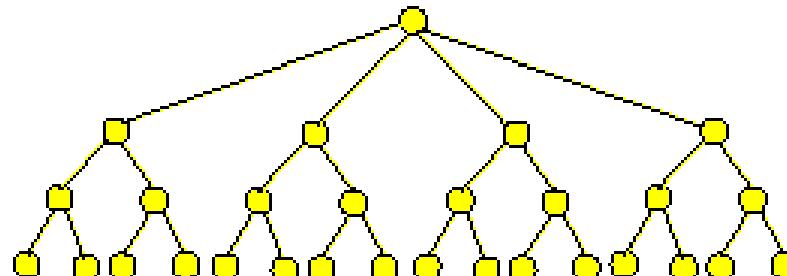
- Idea to get better algorithm
 - In binary tree algorithm, we have done pairwise comparisons using a processor for each pair to the maximum
 - Now idea is to **compare more than two elements at a time** and get their maximum as we have a good algorithm to get maximum in $O(1)$ time

Optimal Computation of Maximum

- We make use of accelerated cascading tree
- cascading tree
 - Looks similar to binary tree, but actually not
 - Here, in place of considering only two numbers for comparison, we consider a lot of numbers
 - Number of elements compared depends on the level of the node at which comparison is performed
 - Actually, the number of elements compared at a node is same as the number of children it has
 - We use a number of processors to compare the elements at a particular level

An $O(\log \log n)$ time Algorithm

- Instead of a binary tree, we use a more complex tree. Assume that $n = 2^{2^k}$.
- The root of the tree has $2^{2^{k-1}} = \sqrt{n}$ children.
- Each node at the i -th level has $2^{2^{k-i-1}}$ children for $0 \leq i \leq k-1$
- Each node at level k has two children.



An $O(\log \log n)$ time Algorithm

Some Properties

- The depth of the tree is k . Since

$$n = 2^{2^k}, k = O(\log \log n)$$

- The number of nodes at the i -th level is

$$2^{2^k - 2^{k-i}}, \text{ for } 0 \leq i < k.$$

Prove this by induction.

An $O(\log \log n)$ time Algorithm

The Algorithm

- The algorithm proceeds level by level, starting from the leaves.
- At every level, we compute the maximum of all the children of an internal node by the $O(1)$ time algorithm.
- The time complexity is $O(\log \log n)$ since the depth of the tree is $O(\log \log n)$.

An $O(\log \log n)$ time Algorithm

- Now the important thing to see is if we could reduce the total work to $O(n)$
- Let us compute the total work that we do in this algorithm
- **Total Work:**
 - Recall that the $O(1)$ time algorithm needs $O(p^2)$ work for p elements.
 - Each node at the i -th level has $2^{2^{k-i-1}}$ children.
 - So the total work for each node at the i -th level is $O(2^{2^{k-i-1}})^2$.

An $O(\log \log n)$ time Algorithm

Total Work:

- There are $2^{2^k - 2^{k-i}}$ nodes at the i -th level.
Hence the total work for the i -th level is:

$$O(2^{2^{k-i-1}})^2 * 2^{2^k - 2^{k-i}} = O(2^{2^k}) = O(n)$$

- For $O(\log \log n)$ levels, the total work is $O(n \log \log n)$. This is suboptimal.

An $O(\log \log n)$ time Algorithm

- Recall, we wanted an algorithm which has
 - Work: $O(n)$ -*a work optimal algorithm*
 - Time better than $O(\log n)$
- Say this proposed algorithm is Algorithm 3
- We could only improve over time but still we have to do $O(n \log \log n)$ work, hence the algorithm is not work-optimal.
- What is the way ahead?
 - We shall merge Algorithm 2 and Algorithm 3 to get better results

Accelerated Cascading

- The first algorithm which is based on a binary tree, is optimal but slow
 - Work: $O(n)$ [*optimal*], Time: $O(\log n)$ [*slow*]
- The second algorithm based on cascading tree is suboptimal, but very fast
 - Work: $O(n \log \log n)$ [*suboptimal*], Time: $O(\log \log n)$ [*fast*]
- We combine these two algorithms through the accelerated cascading strategy

Accelerated Cascading

- If we want an algorithm should match some specific time, then what we do as follows
- A standard technique
 - Reduce the size of the input
 - When size of the input is reduced to certain level, then we will apply a very fast algorithm
 - This is done in Accelerated Cascading
- Accelerated Cascading is used to mix two algorithms

Accelerated cascading

- How do we proceed? What is the trick?
- Step 1: We start with the slow but optimal algorithm until the size of the problem is reduced to a certain value.
 - We do this as we know that as long as we are using binary tree, we are doing $O(n)$ work
 - However, we should not execute the optimal algorithm too long because we know that the depth of the binary tree is $O(\log n)$, so if we execute completely till $\log n$ level then of course we need $O(\log n)$ time

Accelerated cascading

- Step 1: *(...contd)*
 - So we have to execute the optimal algorithm to an extent where our time requirement is still within the limit of $O(\log \log n)$
 - Then we switch over to the fast algorithm
- Step 2:
 - Then we use the suboptimal but very fast algorithm.

Accelerated cascading

Phase 1. [use of optimal but slow algorithm]

- We apply the binary tree algorithm
- Up to what level we should execute this
 - It is enough to start from the leaves and go up to $\log \log \log n$ levels.
 - Application up to this much level is enough as due to this the number of candidates reduces to
$$\frac{n}{2^{\log \log \log n}} = \frac{n}{\log \log n}.$$
 - When we have reduced the number of inputs to as mention above, we can stop use of this algorithm

Accelerated cascading

- The total work done so far is $O(n)$ and the total time is $O(\log \log \log n)$
- Work done is $O(n)$
 - as we were executing optimal work algorithm so far
- And total spend time: $O(\log \log \log n)$
 - as we have only executed to $(\log \log \log n)$ levels
- In Phase 2,
 - Now we have to show that from this point onwards, if we execute the fast algorithm then our processor requirement will be low enough so that we get $O(n)$ work and also the time requirement should be $O(\log \log n)$ time

Accelerated cascading

Phase 2. [we use fast algorithm over the remaining elements]

- In this phase, we use the fast algorithm on the $n' = O\left(\frac{n}{\log \log n}\right)$ remaining candidates.
- The total work is $O(n' \log \log n') = O(n)$.
- The total time is $O(\log \log n') = O(\log \log n)$.
- Note that this trick mostly work in CRCW model only.

Theorem: Maximum of n elements can be computed in $O(\log \log n)$ time and $O(n)$ work on the Common CRCW PRAM.

Applications

- Parallel prefix
 - *very important problem*
 - *pops up every where while designing parallel algorithms*
- List ranking:
 - *Used in algorithms which use trees for computing different functions on trees*
 - *Used in evaluation of arithmetic expressions*
 - *Used in many graph algorithms*

End

PRAM Model

Few more examples and Simulation Theorem

Position of First 1 in a sequence of 0's & 1's

Computing the position of the first one in the sequence of 0's and 1's in a constant time

00101000

00000000
00000001
01101000
00010100

Position of First 1 in a Sequence of 0's & 1's

- Use of CRCW PRAM
- Assume that we have number of processors available same as the number of elements

Algorithm A

(2 parallel steps and n^2 processors)

```
for i = 1 to n, do in parallel
  for j = 1 to n, do in parallel
    if (i < j)
      { if C[i] = 1 and C[j] = 1
        then C[j] := 0
      }
```

Part 1

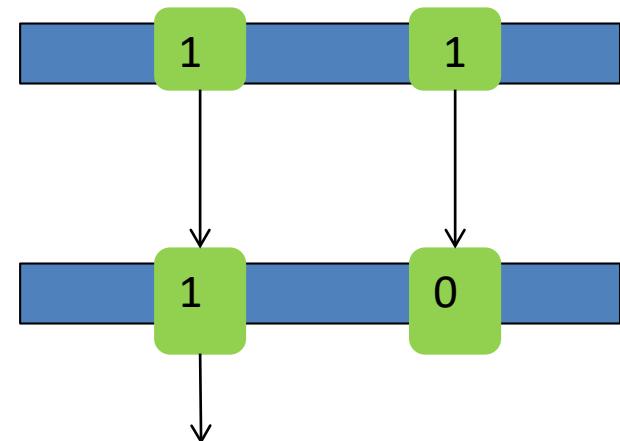
```
for each 1 ≤ i ≤ n do in parallel
  if C[i] = 1 then FIRST-ONE-POSITION := i
```

Part 2

Example:

C=[0,0,0,1,0,0,0,1,1,1,0,0,0,1]

FIRST-ONE-POSITION(C)= 4



After the first parallel step
 C will contain a single 1
and all other elements will
be zero

Position of First 1 in a Sequence of 0's & 1's

- In Algorithm A
 - A particular location is read by at most n processors
 - See the table below for 4 elements array C and 16 processors, P_{ij} , $i=1..n, j=1..n$
 - **A processors P_{ij} reads data from locations i and j (or we can say it works) only if $i < j$**

Data Array			
C			
P_{11}	P_{12}	P_{13}	P_{14}
P_{21}	P_{22}	P_{23}	P_{24}
P_{31}	P_{32}	P_{33}	P_{34}
P_{41}	P_{42}	P_{43}	P_{44}

1	This location is accessed by P_{12} , P_{13} and P_{14}
2	This location is accessed by P_{23} and P_{24}
3	This location is accessed by P_{34}
4	This location is not accessed by any processor

Position of First 1 in a Sequence of 0's & 1's

Analysis of Algorithm A

- Concurrent Read (CR)

– Part 1:

- All processors are working in parallel
- So there exist many processors (in fact n) which will read data from a *particular location* of array C

– Part 2:

- there exist no concurrent read in Part 2

Position of First 1 in a Sequence of 0's & 1's

Analysis of Algorithm A (...*contd*)

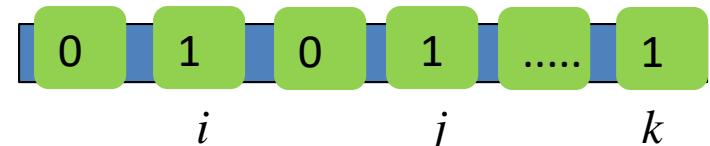
- Concurrent Write (CW)

- Part 1:

- There may be more than one processor (at most $n-1$) writing at the same location of array C
 - Processor (i, k) and processor (j, k) both will write 0 at location k
 - **Note that** processor (i, k) is the processor comparing the bits at location i and k

- Part 2:

- In array C, there will only be a single location which may contain bit '1', so no concurrent write at variable **FIRST-ONE-POSITION**



Reducing number of processors

Algorithm B

- it reports if there is any **1** in the sequence

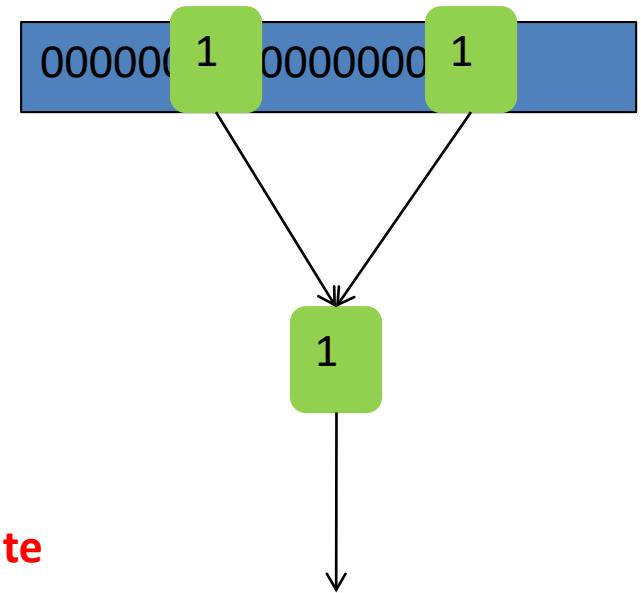
There-is-one:=0

for each $1 \leq i \leq n$ do in parallel

if $C[i] = 1$

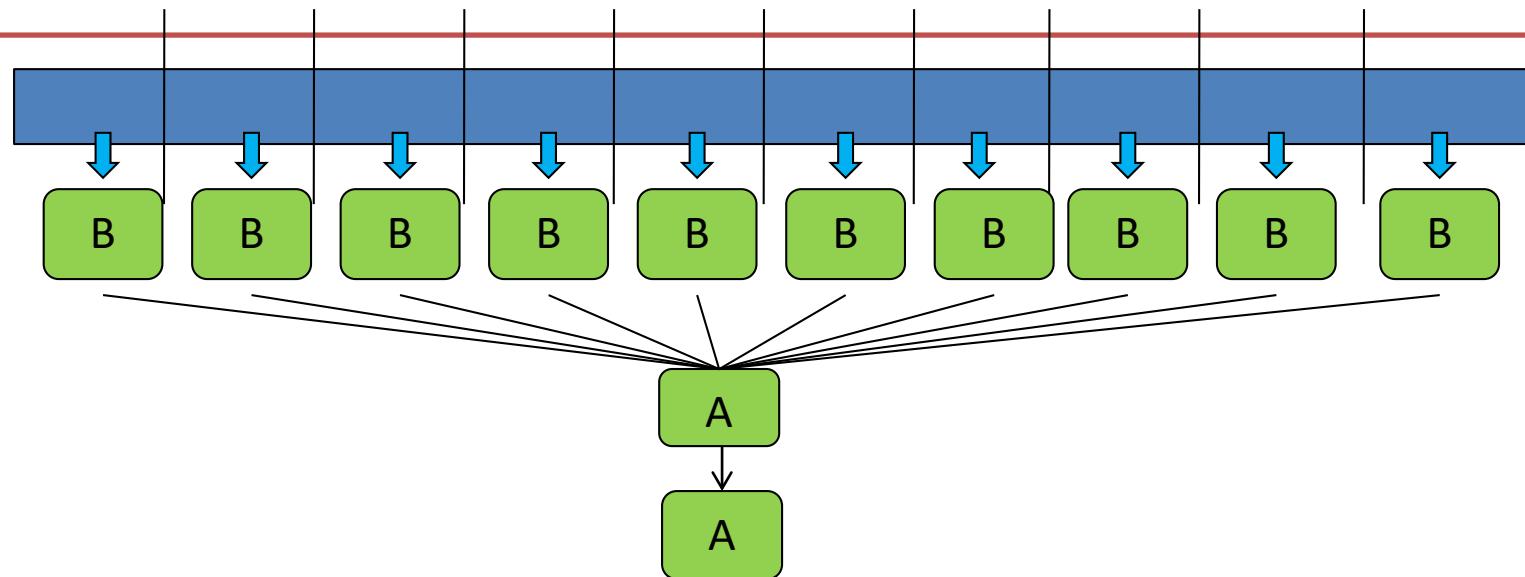
then $There-is-one := 1$

Concurrent write



Now we can merge two algorithms A and B

1. Partition table C into segments of size \sqrt{n}
2. In each segment apply the algorithm B
 - ✓ Each block has \sqrt{n} elements, hence needs \sqrt{n} processors
 - ✓ Hence total processors needed would be $= \sqrt{n} \times \sqrt{n}$
 - ✓ Hence, we can operate on all the blocks in parallel
3. Find position of the first one in these sequence by applying algorithm A
4. Apply algorithm A to this single segment and compute the final value



Complexity

- Processors required = $(\sqrt{n})^2 = n$
- Algorithm A is applied twice and each time to the array of length \sqrt{n}
- Time complexity
 - If input sequence length is n , then (\sqrt{n}) segments
 - The time is $O(1)$ for algorithm A and B, both
 - Overall $T_p = O(1)$
- **What if elements are more than n**
 - Total time to find out the position in sequence depends on the number of segments created, if elements are more than n
 - Process elements sequentially in chuck of n elements using algorithm A and B and stop when first chuck with a 1 detected

Are all PRAMs equivalent?

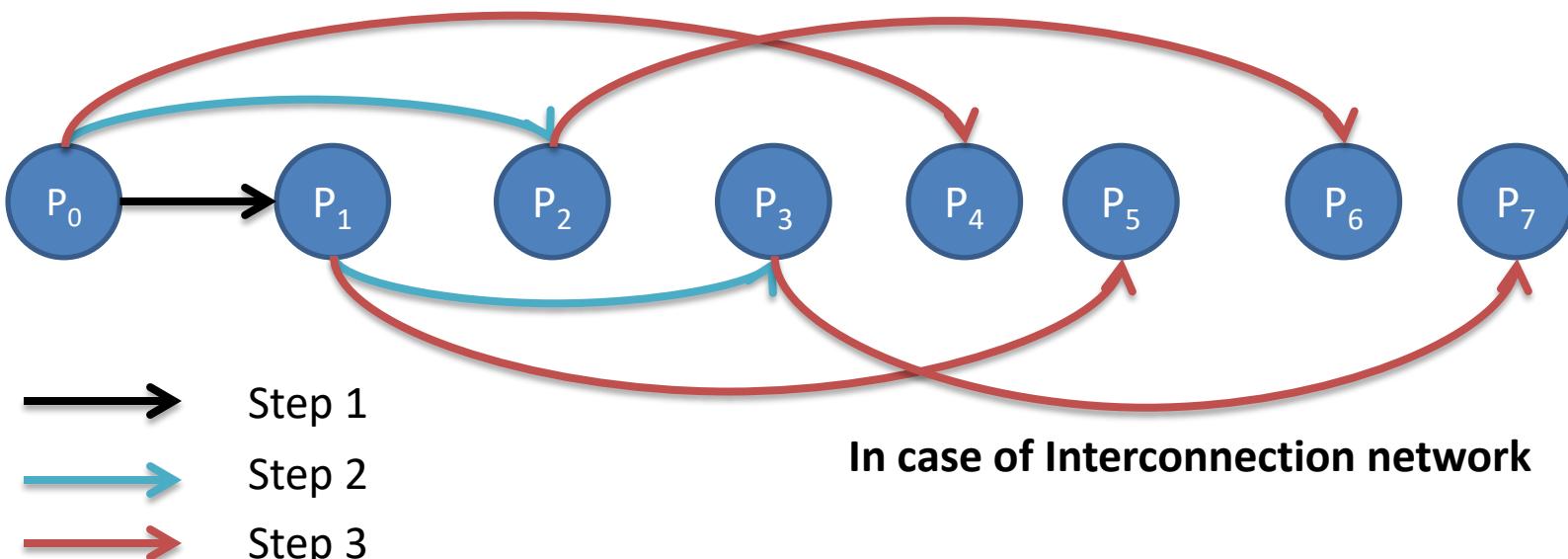
- Consider the following problem
 - given an array of n elements, $\langle e_1, e_2, \dots, e_n \rangle$ all distinct, find whether some element e is present in the array
- On a **CREW PRAM**, we can have following algorithm that works in time $O(1)$ on n processors:
 - initialize a boolean variable to FALSE
 - Each processor i reads e_i and e and compare them
 - if equal, then write TRUE into the boolean variable (since only one processor will write, so we are ok for CREW)

Are all PRAMs equivalent?

- On an **EREW PRAM**, one cannot do better than $O(\log n)$
 - Each processor must read e separately
 - at the worst a complexity of $O(n)$, with sequential reads
 - at the best a complexity of $O(\log n)$, with series of doubling of the value at each step so that eventually everybody has a copy (just like a broadcast in a binary tree, or in fact a k -ary tree for some constant k)
 - Generally, diffusion of information to n processors on an **EREW PRAM** takes $O(\log n)$
- **Conclusion: CREW PRAMs are more powerful than EREW PRAMs**

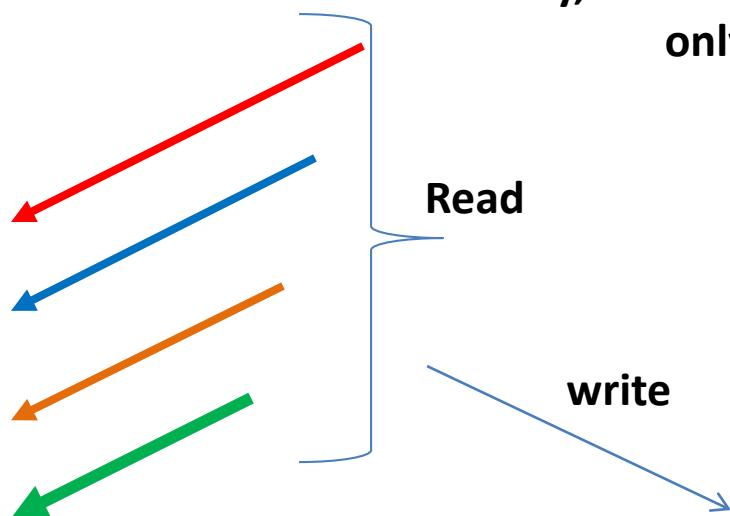
Are all PRAMs equivalent?

- Broadcasting the value of e to all processors (**in interconnection network**)
- Initially only one processor has e
 - At the worst a complexity of $O(n)$, with sequential distribution to all
 - At the best, it needs $O(\log n)$ time for distribution using doubling of the value in binary tree fashion

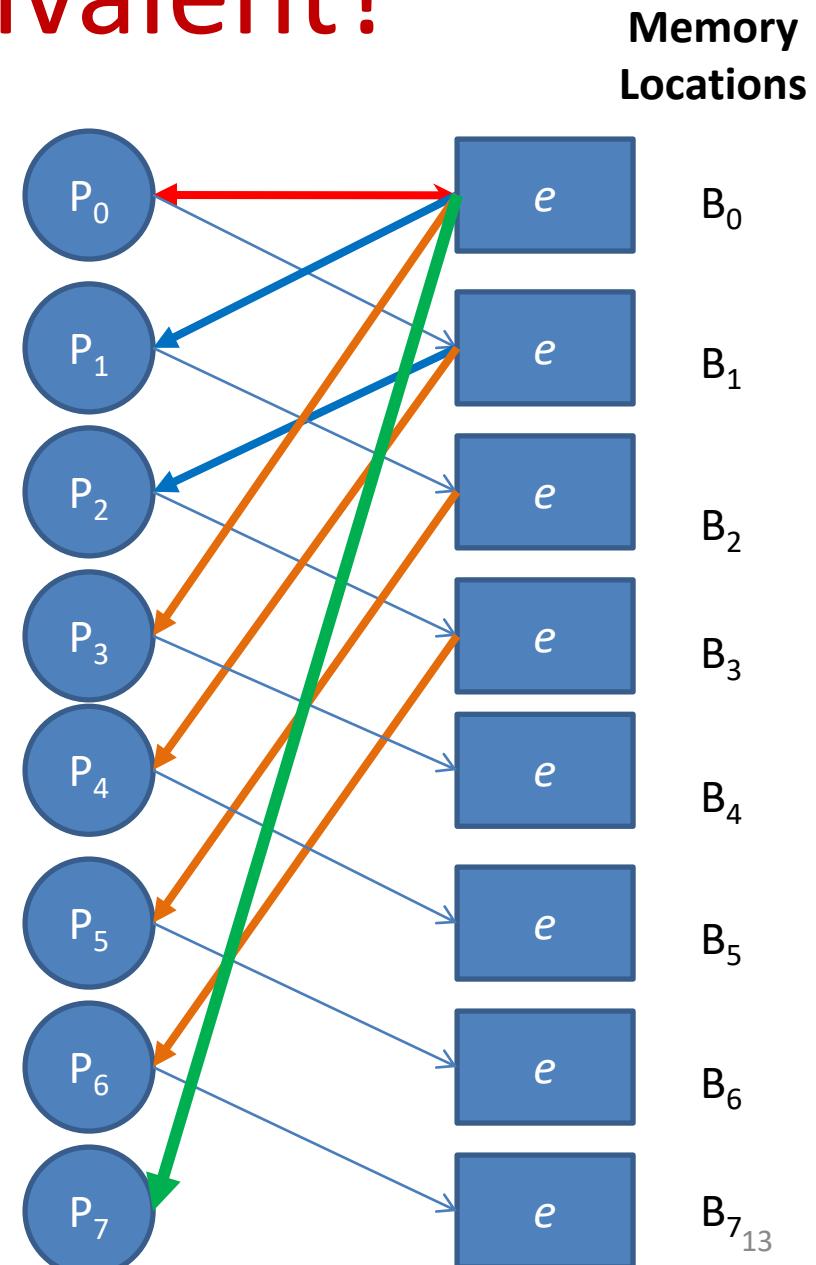


Are all PRAMs equivalent?

- Broadcasting the value of e to all processors (in case of shared memory)
- It takes $O(\log n)$ time



Date to be read is e , initially, it is available only at B_0



CRCW vs. EREW

- **CRCW**
 - Easier to program, runs faster, more powerful
 - However, hardware implementations are expensive
 - Used infrequently
 - Implemented hardware of CRCW is slower than that of EREW
 - In reality one cannot find maximum in $O(1)$ time using CRCW

CRCW vs. EREW - Discussion

- **EREW**
 - Programming model is too restrictive
 - Cannot implement powerful algorithms
- Some would say: use of either EREW or CRCW is wrong while discussing a model
- Processors must be connected by a network, and only be able to communicate with other *via* the network, so network should be part of the model

Simulating CRCW with EREW

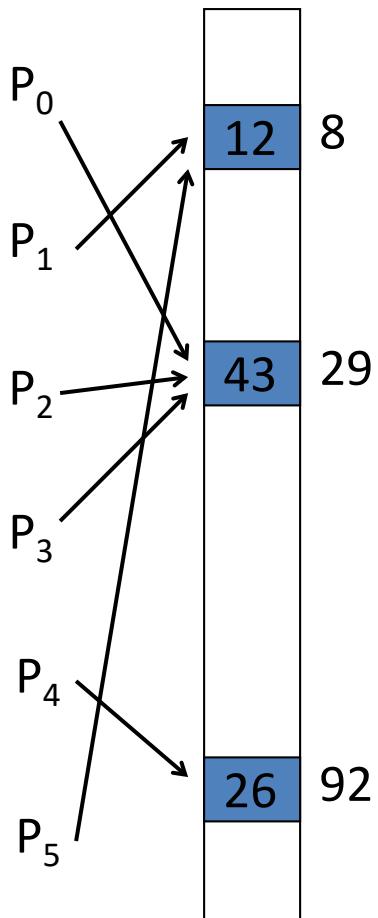
- CRCW algorithms are faster than EREW algorithms
 - How much fast?
- Theorem
 - A p -processor CRCW algorithm can be no more than $O(\log p)$ times faster than the best p -processor EREW algorithm

Simulation Theorem

- **Simulation theorem:** Any algorithm running on a **CRCW PRAM** with p processors cannot be more than $O(\log p)$ times faster than the best algorithm on a EREW PRAM with p processors for the same problem
- **Proof:**
 - “Simulate” concurrent writes
 - When P_i writes value x_i to address I_i , one replaces the write by an (exclusive) write of (I_i, x_i) to $A[i]$, where $A[i]$ is some auxiliary array with one slot per processor
 - Then one sorts array A by the first component of its content
 - Processor i of the EREW PRAM looks at $A[i]$ and $A[i-1]$
 - if the first two components are different **or** if $i = 0$, write value x_i to address I_i
 - Since A is sorted according to the first component, writing is exclusive

Proof (continued)

Picking one processor for each competing write



$P_0 \rightarrow (29, 43)$	$= A[0]$
$P_1 \rightarrow (8, 12)$	$= A[1]$
$P_2 \rightarrow (29, 43)$	$= A[2]$
$P_3 \rightarrow (29, 43)$	$= A[3]$
$P_4 \rightarrow (92, 26)$	$= A[4]$
$P_5 \rightarrow (8, 12)$	$= A[5]$

sort

$A[0]=(8,12)$	P_0 writes
$A[1]=(8,12)$	P_1 nothing
$A[2]=(29,43)$	P_2 writes
$A[3]=(29,43)$	P_3 nothing
$A[4]=(29,43)$	P_4 nothing
$A[5]=(92,26)$	P_5 writes

Proof (continued)

- Note that we said that we sort array A
- If we have an algorithm that sorts p elements with $O(p)$ processors in $O(\log p)$ time, we are done
- Turns out that there is such an algorithm: Cole's Algorithm, which does sorting in $O(\log p)$ time
 - basically a merge-sort in which lists are merged in constant time!
 - It is a beautiful algorithm, but we do not really have time for it, and it is rather complicated
- Therefore, the proof is complete

End

More Problems on PRAM Model

Roots of a Forest of Directed Trees

Example: Finding the roots of forest represented as parent array P

$P[i] = j$ if and only if (i, j) is a forest edge

$P[i] = i$ if and only if i is a root

Desired output: is an array S , such that $S[j]$ is the root of the tree containing vertex j , for $1 \leq j \leq n$

Termination detection?

Use self-loops to recognize roots, i.e., a vertex i is a root if and only if $P[i] = i$

Algorithm - Roots of Forest

```
1. for i=1:n, in parallel
2.     S[i] := P[i]
3. for i=1:n, in parallel
4. {
5.     while S[i] != S[S[i]]
6.         S[i] := S[S[i]]
7. }
```

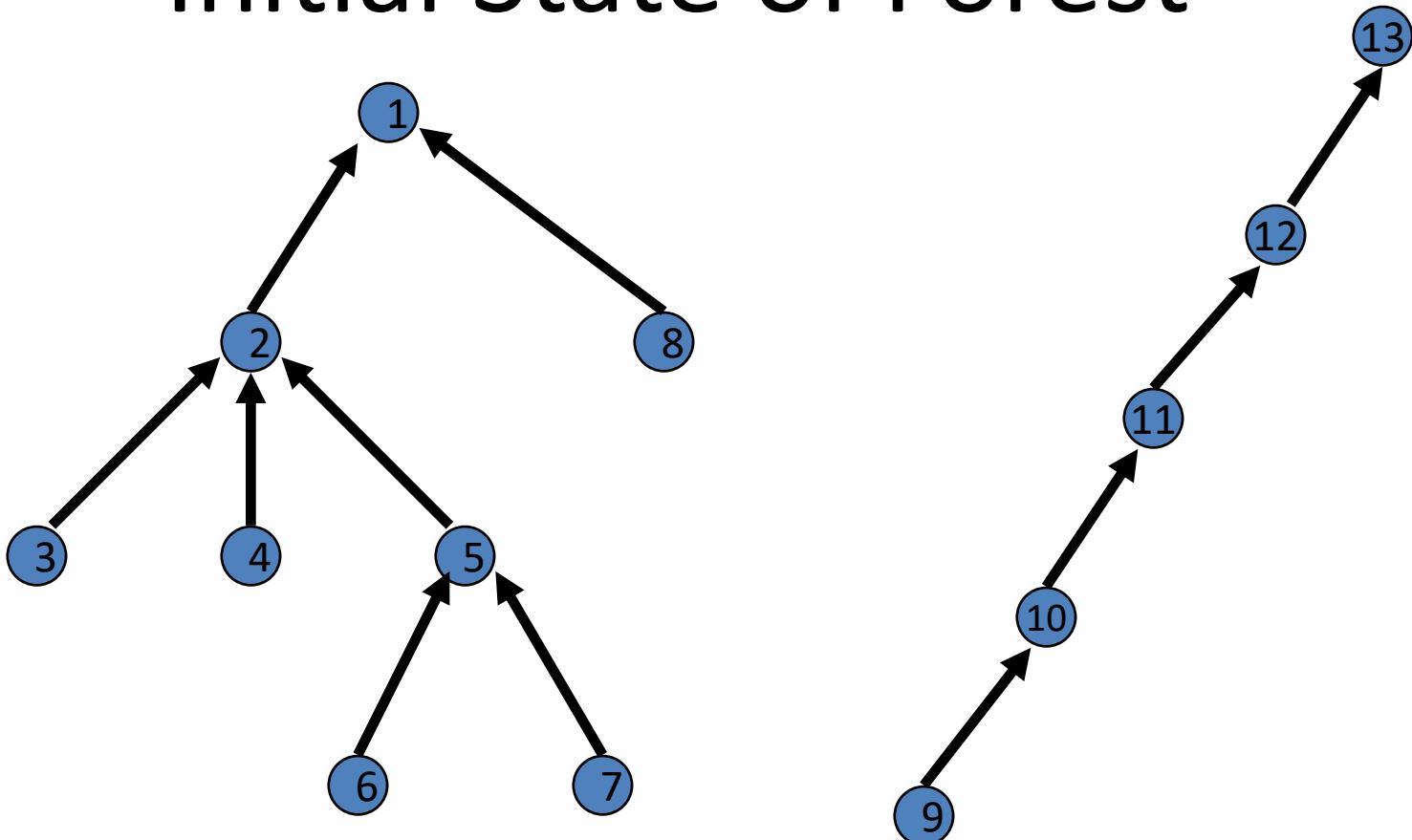
}

To create a copy of the parent array

- **Note**
 - that in line 6 all instances of $S[S[i]]$ are evaluated before any of the assignments to $S[i]$ are performed
 - follows CREW PRAM model
 - All writes are exclusive as i^{th} processor writes at i^{th} location of the array S
 - However, more than a constant number of vertices may read values from a common vertex as they all may have common parent

Convention used in drawing trees: Draw trees with edges directed from child to parent

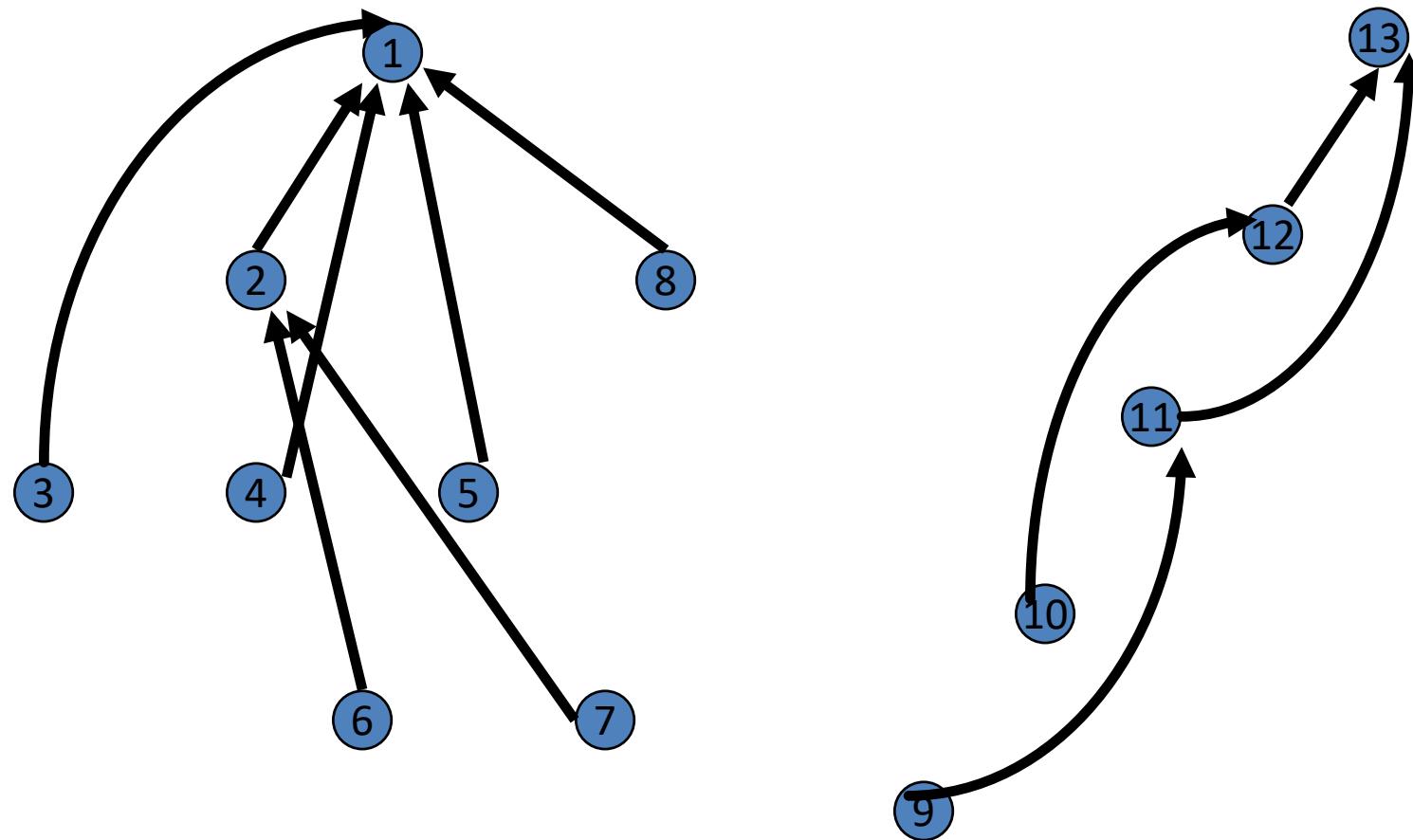
Initial State of Forest



Representation of tree in the form of an array (initial)

<i>Index</i>	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>S</i>	<i>Value</i>	1	1	2	2	2	5	5	1	10	11	12	13

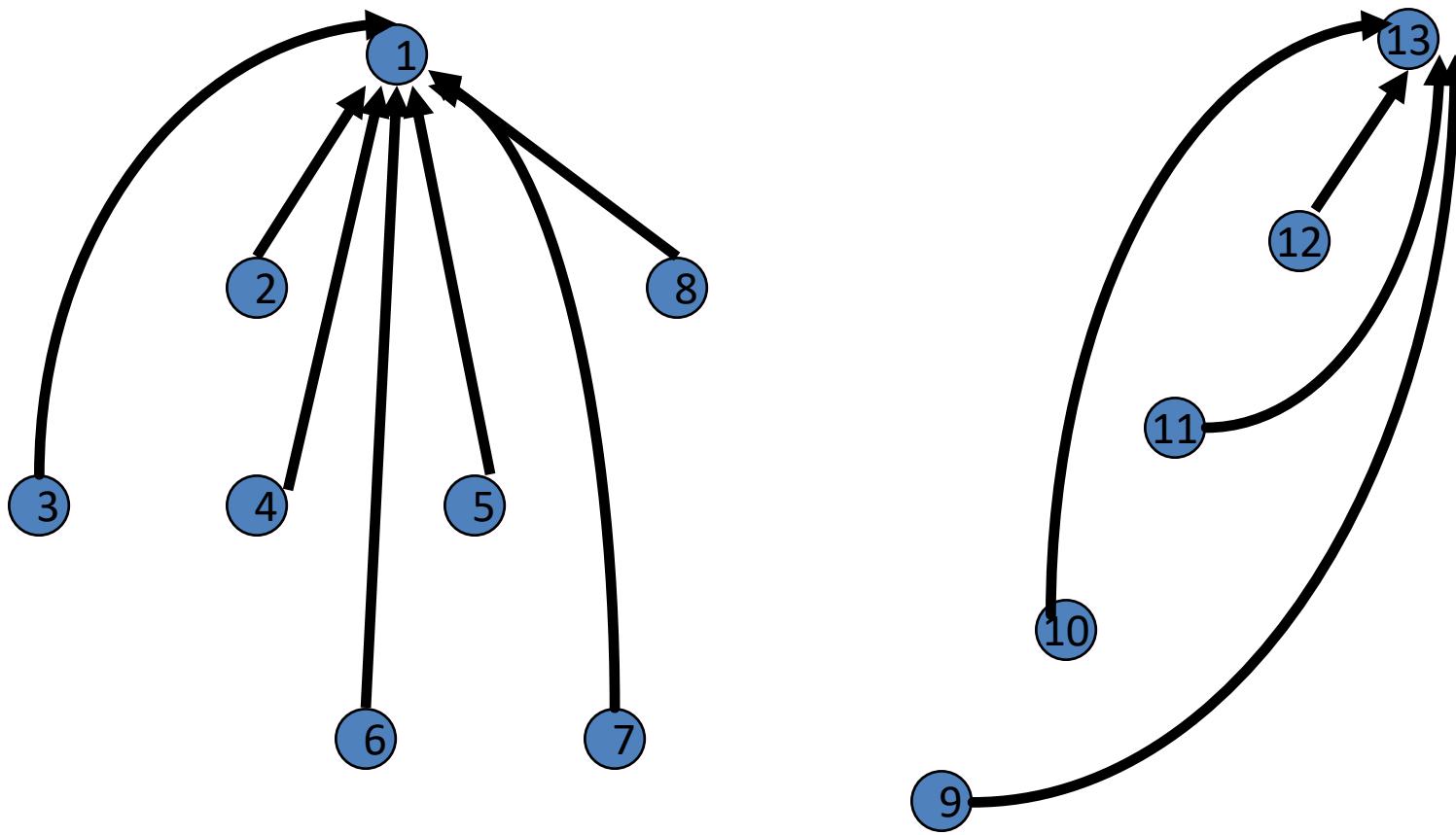
After One Iteration



Representation of tree in the form of an array (after 1 iteration)

<i>Index</i>	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>S</i>	<i>Value</i>	1	1	1	1	1	2	2	1	11	12	13	13

After another iteration

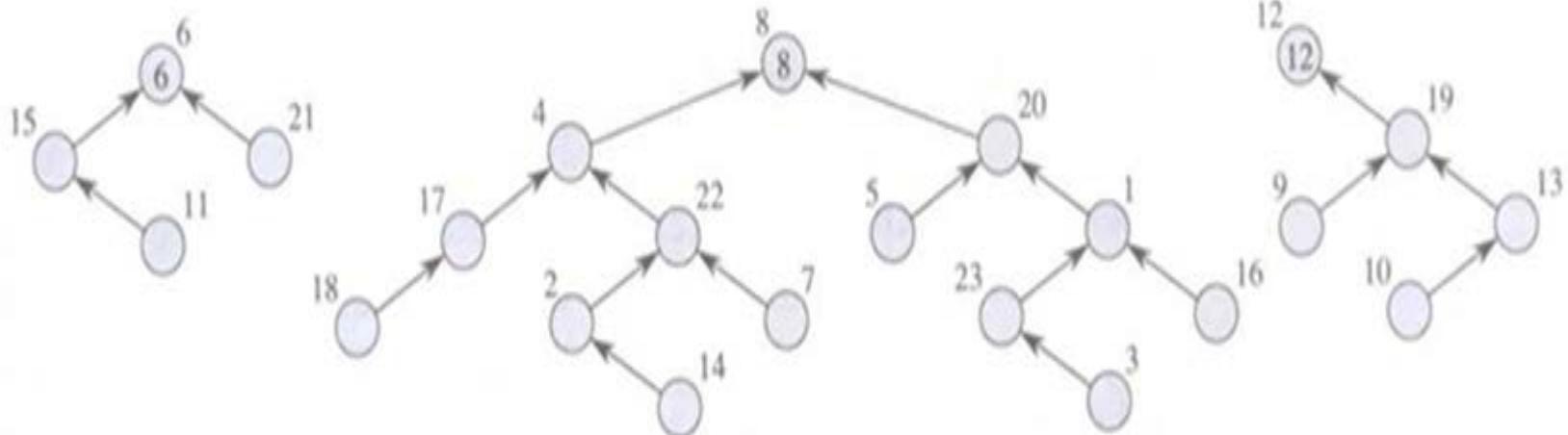


Representation of tree in the form of an array (after 2 iterations)

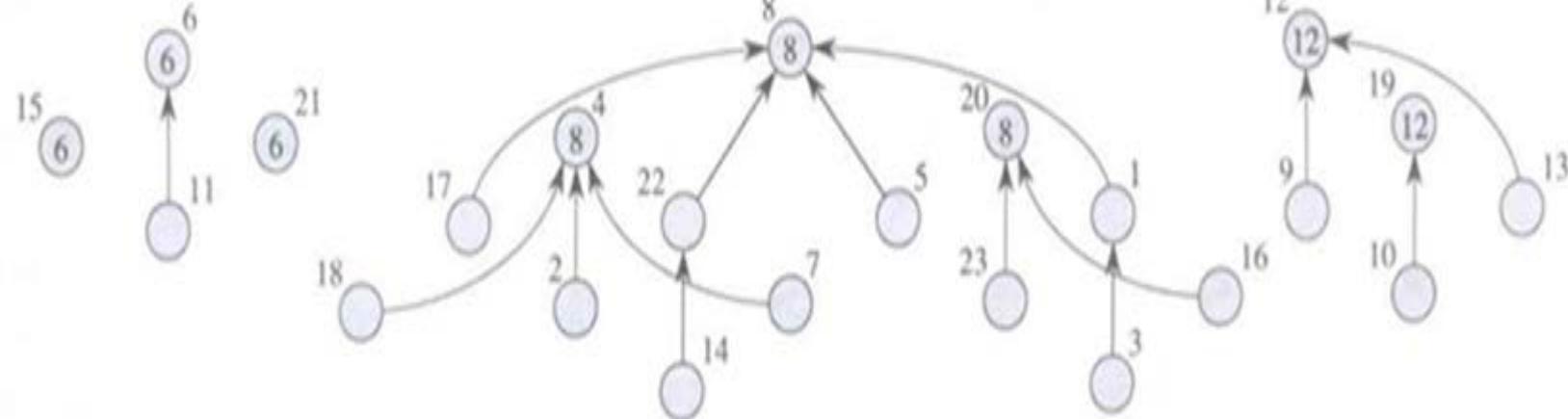
<i>Index</i>	1	2	3	4	5	6	7	8	9	10	11	12	13
<i>S</i>	<i>Value</i>	1	1	1	1	1	1	1	13	13	13	13	13

Concurrent Read – Finding Roots

(a)

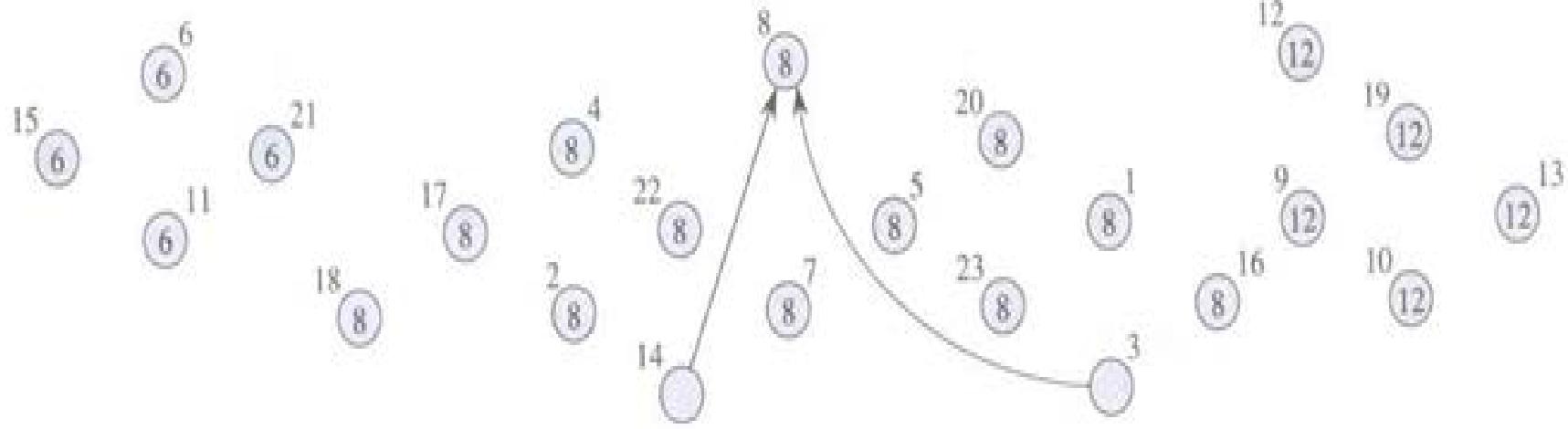


(b)

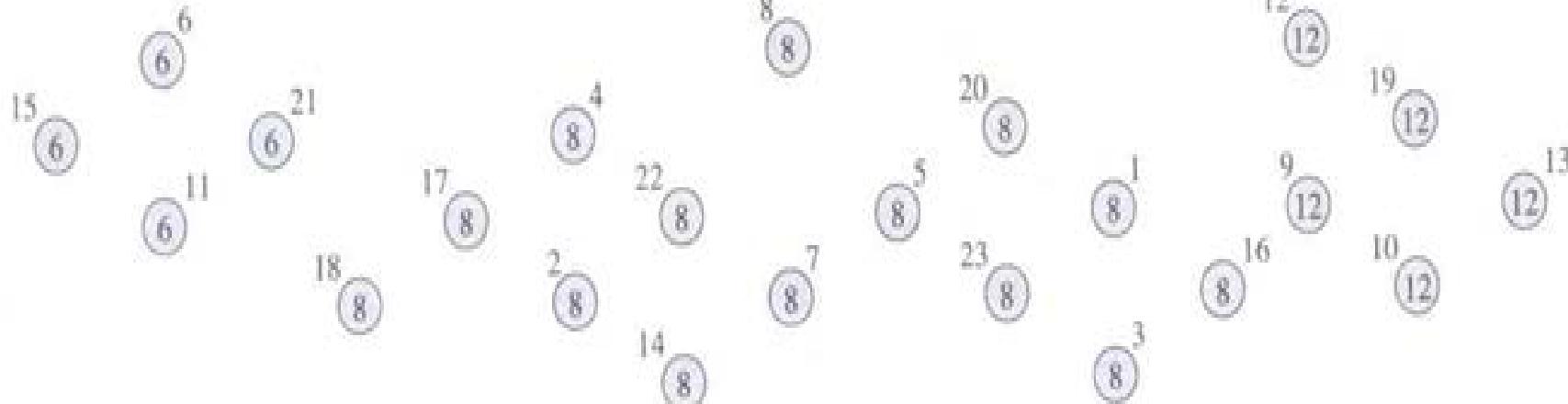


Concurrent Read – Finding Roots

(c)



(d)





Analysis of Pointer Jumping

- At each step, tree distance between i and $S[i]$ doubles unless $S[i]$ is a root
- **CREW model**
 - Correctness can be obtained by using induction on h
 - If h is the height of maximum-depth tree in the forest then
 - $O(\log h)$ steps, $O(n \log h)$ work
 - $T_p = O(\log h)$
 - $T_S(n) = O(n)$ [sequential time complexity]
- **Work efficient??**
 - $O(n)$ Vs. $O(n \log h)$
 - Not work-efficient unless h is a constant
- In particular, for a linked list, this algorithm would take $\Theta(\log n)$ steps and $\Theta(n \log n)$ work

Analysis of Pointer Jumping

- These bounds are weak, but we cannot assert anything stronger without assuming more about the input data
- **Exercise:**
 - An interesting exercise is to associate with each vertex i the distance d_i to its successor measured along the path in the tree, and to modify the algorithm to correctly maintain this quantity
 - On termination, the d_i will be the distance of vertex i from the root of its tree

Algorithm - revisited

```
1. Algorithm 1  
2. for i=1:n, in parallel  
3.     S[i] := P[i]  
  
4. for i=1:n, in parallel  
5. {  
6.     while S[i] != S[S[i]] ←  
7.         S[i] := S[S[i]]  
8. }
```



To create a copy of the parent array

Each processor has its own version of while loop.

There is no way to synchronize while-loop of one processor with the one with another for data consistency.

- The algorithm **glosses over one important detail: how do we know when to stop iterating the **while-loop**?**
- The first idea is to use a fixed iteration count, as follows.
 - Since the height of the **tallest tree has a trivial upper bound of n** ,
 - we do not need to repeat the pointer jumping loop more than **$\log n$** times
- Based on this we can have algorithm shown in next slide

Algorithm - revisited

1. Algorithm 2

```
2. for i=1:n, in parallel
3.     S[i] := P[i]

1. for k=1:log n
2. {
3.     for i=1:n, in parallel
4.         S[i] := S[S[i]]
5. }
```

- Though correct, bound on the while loop as ‘ $\log n$ ’ is inefficient since our forest might consist of many shallow and bushy trees.
- Its work complexity is $\Theta(n \log n)$ instead of $O(n \log h)$, and its step complexity is $\Theta(\log n)$ instead of $O(\log h)$.
- The second idea of an “honest” termination detection algorithm, presented in next slide



Algorithm - revisited

Algorithm 3

```
1. for i=1:n, in parallel
2.     S[i] := P[i]

1. repeat           ←————— // O(log h) run time
2. {
3.     for i=1:n, in parallel
4.     {
5.         S[i] := S[S[i]]
6.         if S[i] ≠ S[S[i]]
7.             M[i] = 1
8.         else
9.             M[i] = 0
10.    }
11.    t = REDUCE(M, +)    // O(log n) run time, O(n) work
12. }
13. until t = 0          //... work O(n)
```

This approach has the desired work complexity of $O(n \log h)$, but its step complexity is $O(\log n \log h)$, since we perform an $O(\log n)$ step reduction in each of the $O(\log h)$ iterations.

Algorithm - revisited

- In the design of parallel algorithms, minimizing work complexity is most important, hence we would probably favor the use of the honest termination detection as in *Algorithm 3*
- However, the basic algorithm, even with this modification, is not fully work efficient
- The algorithm can be made work efficient using paradigm like **algorithm cascading**

Algorithm - revisited

- Look at Step 11:
 - When Exclusive write (EW) is used, sum of elements of $M[i]$ in Step 11 is done in $O(\log n)$ time using balanced tree structure
 - If we have concurrent write available we have better algorithm as discussed below
- Step 11 can be made to run in $O(1)$ time using CRCW PRAM
 - Use a flag which is set as ‘true’ initially (before step 11)
 - For all i , i th processor checks $M[i]$, and if $M[i]$ is 1, it writes ‘false’ to flag
 - If flag remains ‘true’ that means $M[i]$, for all i are zero
 - Algorithm can be stopped if flag remains ‘true’ after Step 11

Comparisons

	Algorithm 1	Algorithm 2	Algorithm 3
Step Complexity (T_p)	$O(\log h)$	$O(\log n)$	$O(\log h \log n)$ [$O(\log h) [O(1) + O(\log n)]$]
Work	$O(n \log h)$	$O(n \log n)$	$O(n \log h)$ [$O(\log h) [O(n) + O(n)]$]

Designing of Parallel Algorithms
using
Euler Tour Technique
(Part – 1)

An overview

- Use of Euler tour technique
 - Computation of different tree functions
 - Tree contraction
 - Evaluation of arithmetic expressions

Introduction

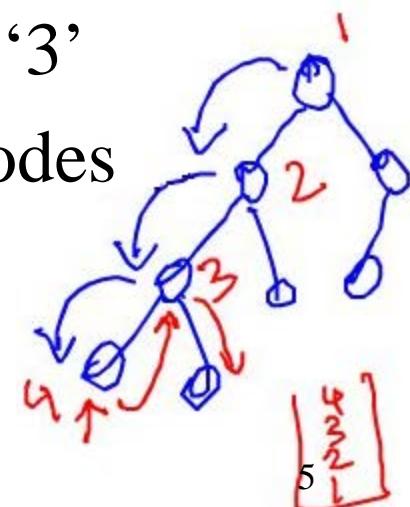
- We will discuss Euler Tour technique
- It is a very important technique in Tree and graph computations in parallel computing
- Euler tour technique bypasses the depth first search in sequential computing
- Sequential Computing in evaluation of tree functions mostly uses DFS
 - many tree functions which we learnt in sequential computing are based on depth first search (DFS)

Introduction-DFS

- **Depth First Search (DFS)**
- In DFS: we progressively go down in the tree until we cannot go any further
- So when we hit a leaf and cannot go any further, then what we do is that we go back and try to explore other parts of the tree
- This is done by keeping all the nodes that are encountered during the search in a stack
- Stack is a structure which follows *last in first out (LIFO)* strategy

Introduction-DFS

- For example below, stack content looks like this when we perform DFS on the shown tree
- We start from node ‘1’ and keep going down till node ‘4’ and insert all the nodes that we encounter in the stack
- When we hit leaf node ‘4’ then we pop out ‘4’ and look what is beneath it, it is node ‘3’
- So we go back to ‘3’ and explore other nodes adjacent to ‘3’.. And so on



Introduction-DFS

- DFS follows a very simple strategy and runs in $O(n)$ time
- However unfortunately we cannot use this strategy in designing parallel algorithm
- This is because as we can see that there is a very strict sequential order in which we visit the nodes
- Hence we have to do **something else** to design parallel algorithm
- We use **Euler Tour technique** for this

Introduction-DFS

- We will see that we can compute many tree function using Euler tour technique which is parallel version of DFS
- We can compute a host of tree functions like
 - *preorder* numbering
 - *inorder* numbering
 - *postorder* numbering
 - Finding number of descendants of a node
 - And so on
- We can do many such computations on tree using Euler Tour technique

Introduction-DFS

- Now we see Euler tour technique
- We see how we conduct Euler tour given a tree
- We shall also see the computation of two tree functions
 - Rooting a tree
 - *preorder* numbering

Introduction-DFS

- **Rooting a tree:**
 - A given tree may not have a specified root
 - Rooting a tree involves finding root of a tree in such cases
- **Why is this important for us?**
 - Without a root we cannot establish parent-child relationship
 - In absence of parent-child relationship, there is no meaning of computation of various tree functions
 - Hence rooting a tree is very important

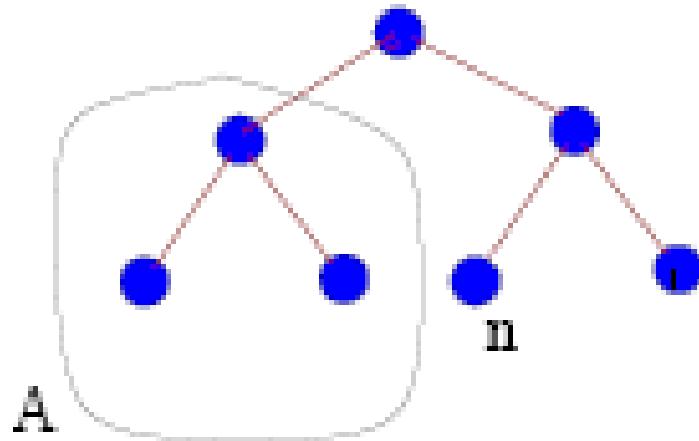
Introduction-DFS

- **Exercise:** few other cases where we can use Euler tour technique
 - **Tree contraction:** a very versatile technique
 - **Evaluation of arithmetic expression:**
 - A tree where leaves of the tree are numbers, may be integers or real numbers and internal nodes of a tree are operator like ‘+’ and ‘*’
 - Objective is to evaluate value of whole expression tree
 - It is a very important problem in compiler designing
 - To evaluate an expression, a compiler constructs a tree
 - We will see a parallel algorithm for this

Problems in Parallel Computations of Tree Functions

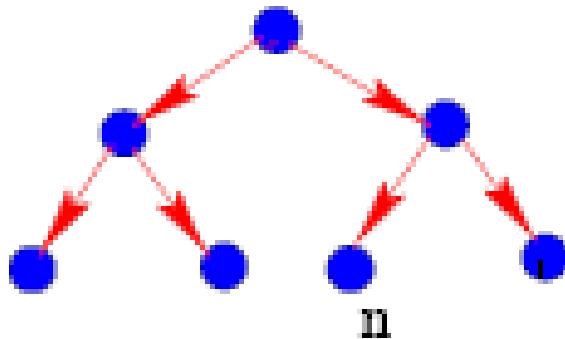
- Most sequential algorithms for these problems use **depth-first** search for solving the problems
- We can write recursive code of few lines along with stack data structure to perform DFS in a tree
- However, depth-first search seems to be inherently sequential in some sense.
 - That means unless we have seen one part of a tree we can not say much about some other part of tree

Parallel Depth-first Search



- For example: if we want to depth numbering of the nodes, it is difficult to do depth-first search in parallel
- We cannot assign depth-first numbering to the node n unless we have assigned depth-first numbering to all the nodes in the subtree A

Parallel depth-first search

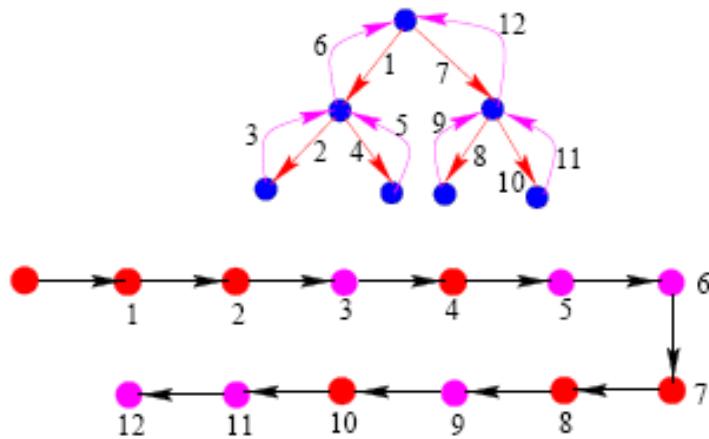


- **Basic Idea to make DFS parallel**
 - We know that there is a definite order of visiting the nodes in depth-first search
 - **We can introduce additional edges in the tree to get this order**
 - The Euler tour technique converts a tree into **a list by adding additional edges**
 - By this way we can convert tree to a linked list

Parallel depth-first search

- So Euler tour technique converts a tree into a linked-list by introducing additional edges
- Now why is it called Euler tour?
 - Euler Tour: Swiss mathematician Euler found that if you have a graph and you want to traverse all the nodes of the graph without traversing any edge more than once, it should be an Euler graph
 - After converting tree to an Euler graph, it is possible to traverse the graph without traversing any edge more than once
 - What are the conditions for a graph to be like that?
- Next slide shows how do we do it

Parallel Depth-first Search



- The **red** (or, **magenta**) arrows are followed when we visit a node for the first (or, second) time
- If the tree has n nodes, we can construct a list with $2n - 2$ nodes, where each arrow (directed edge) is a node of the list
- Assign numbers to edges
- There are 12 edges and 7 nodes to start with

Parallel Depth-first Search

- How do we make the list
 - Start from the root
 - Every time we visit a new node it is indicated by a **red arrow**
 - When we are doing backtracking, then those arrows are denoted by **magenta color**
 - This is the way we denote edges and we can see that we have introduced extra edges (which are actually **magenta edges**)

Parallel Depth-first Search

- How do we make the list*contd*
 - So in the list, a **red node** represent **a red edge** in the tree and a **magenta node** represent a **magenta edge**
 - There is no relation of the black edges present in the list with respect to the tree
 - These links are introduced additionally to make it a linked list
- When we start making the list, we assume that there is a dummy node from where we come to visit root and since root is a new node at the start we represent a move from dummy node to root as a **red edge**

Parallel Depth-first Search

- Now we can use this list for various computations, and that is the beauty of it
 - For example, if we want to assign the depth-first numbering to a node, we can just check how many red nodes are there before you arrive to this node
- Now using the previously developed techniques we can do many computations on this list

Parallel Depth-first Search

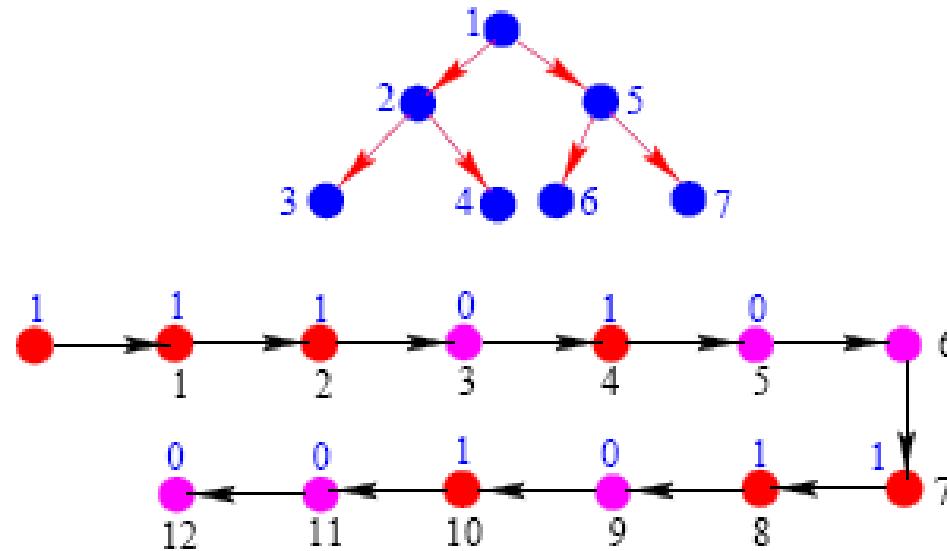
- For example, assume
 - work optimal list ranking algorithm
 - i.e. work: $O(n)$ work and $O(\log n)$ time.
 - [we did not discuss this in this course]
- Now if we have this algorithm, we can do list ranking in this list, only thing is that the size of the list has grown, that is we started with n nodes in the tree, but now we have $2n-2$ nodes in the list
- This is due to the fact that a tree with n nodes have $n-1$ edges, and since here for every edge we have an extra edge, hence the number of edges have doubled

Parallel Depth-first Search

- However, size of the list is still $O(n)$ if we started with $O(n)$ tree
- Hence we can still claim all the work optimal performance on this list too
- That means we can still do list ranking on this list in $O(n)$ work and $O(\log n)$ time using optimal algorithm

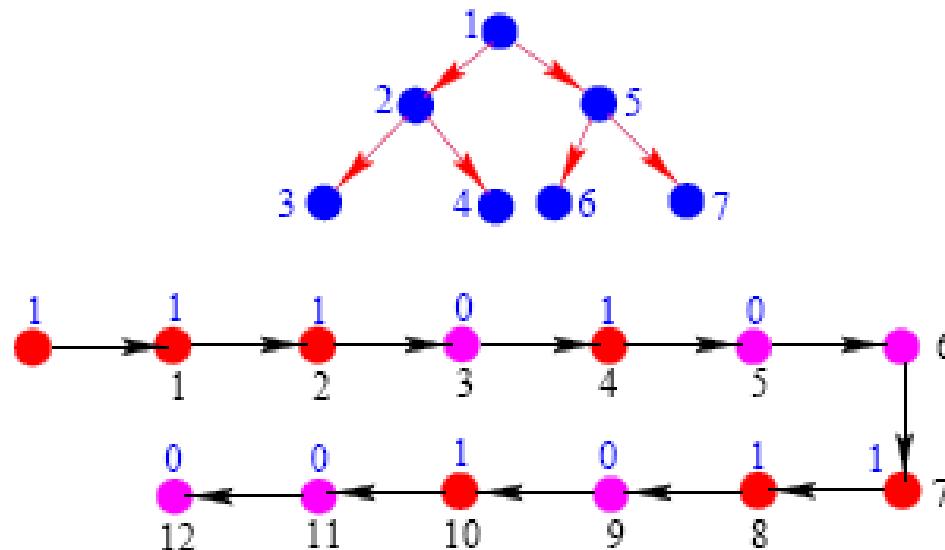
Parallel Depth-first Search

- Another observation
 - If we have a node v in the original tree and assume for time being that this tree is rooted, i.e. node ‘1’ is the root of this tree
 - [Rooting itself is a problem and we will see an algorithm for it using Euler tour technique]



Parallel Depth-first Search

- Another observation*contd*
 - If we assume, for a node $v \in T$, $p(v)$ is the parent of v .
 - Then Each **red** node in the list represents an edge of the nature $\langle p(v) , v \rangle$
 - We can determine the **preorder numbering** of a node of the tree by counting the **red** nodes in the list

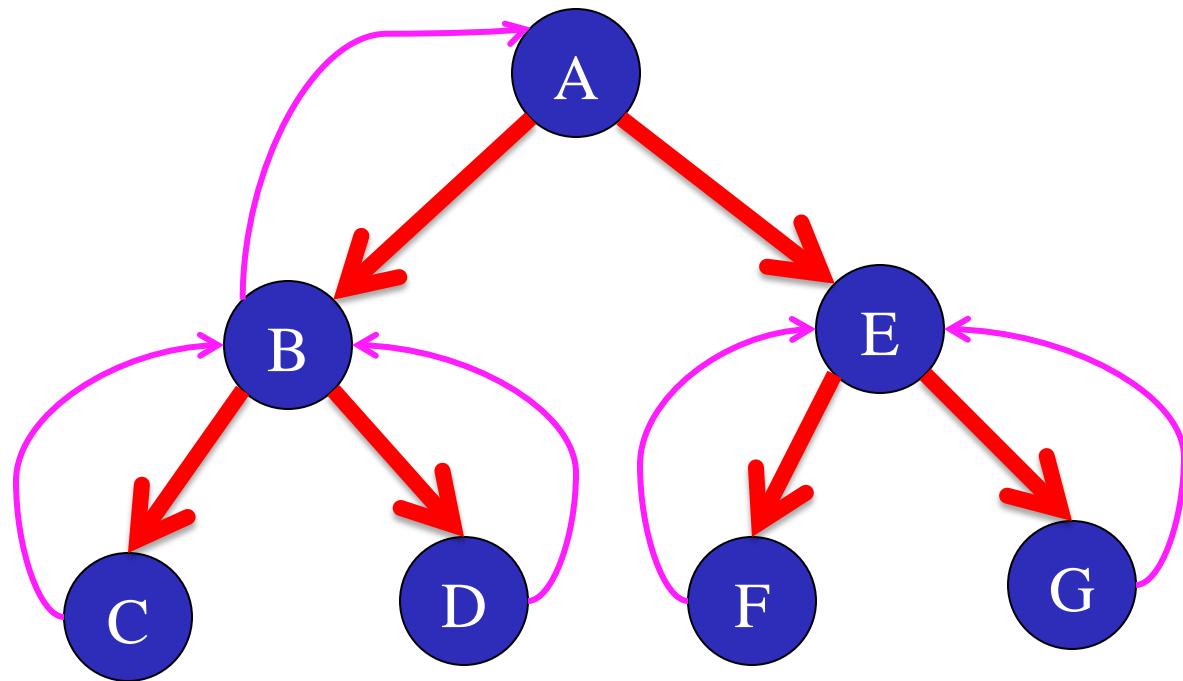


Parallel Depth-first Search

- **Preorder numbering**
 - Given the list based on DFS search, we can do preorder numbering using list ranking (or rather prefix sum) algorithm
 - Just assign value ‘1’ to red nodes and ‘0’ to magenta nodes and perform prefix sum to get the preorder numbering of the nodes
- **Exercise**
 - Similarly we can do inorder numbering, postorder numbering and can find the descends
 - Hint: You need to carefully assign weights to the nodes to perform a particular operation

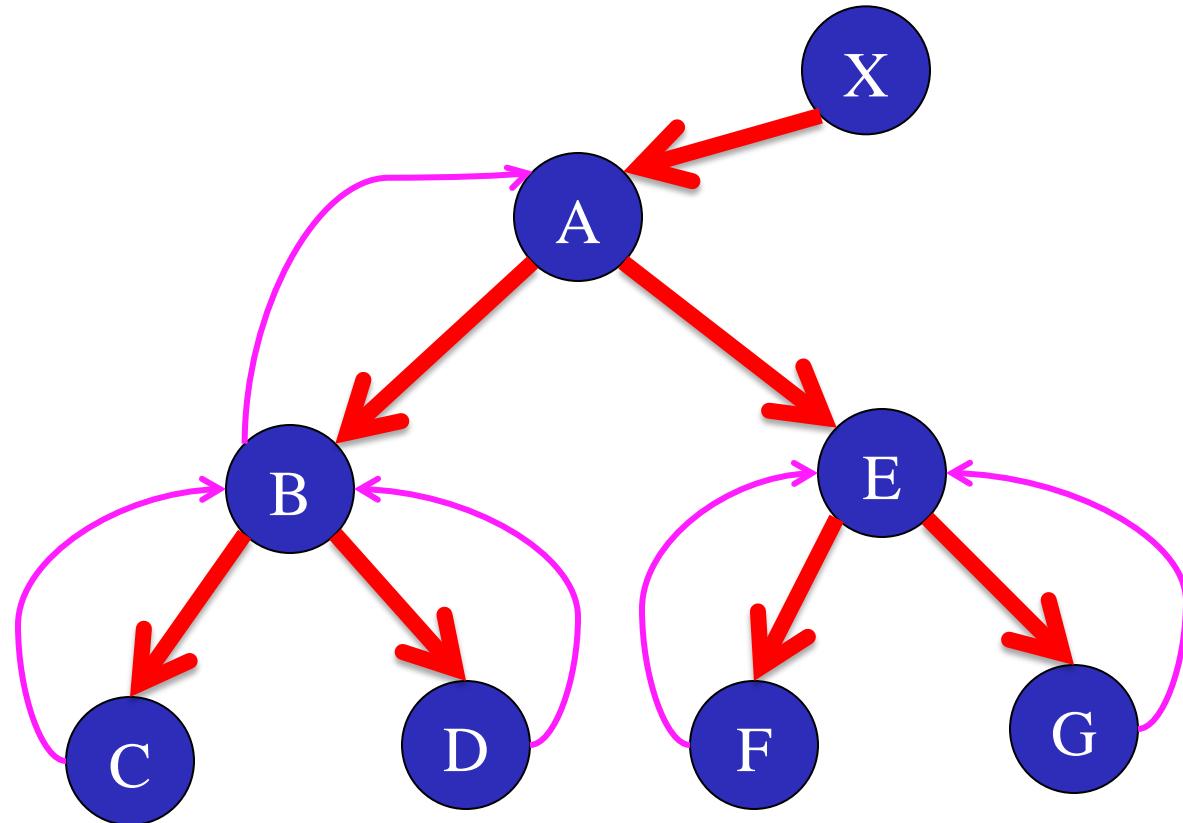
Parallel Depth-first Search –Example

Given Tree structure



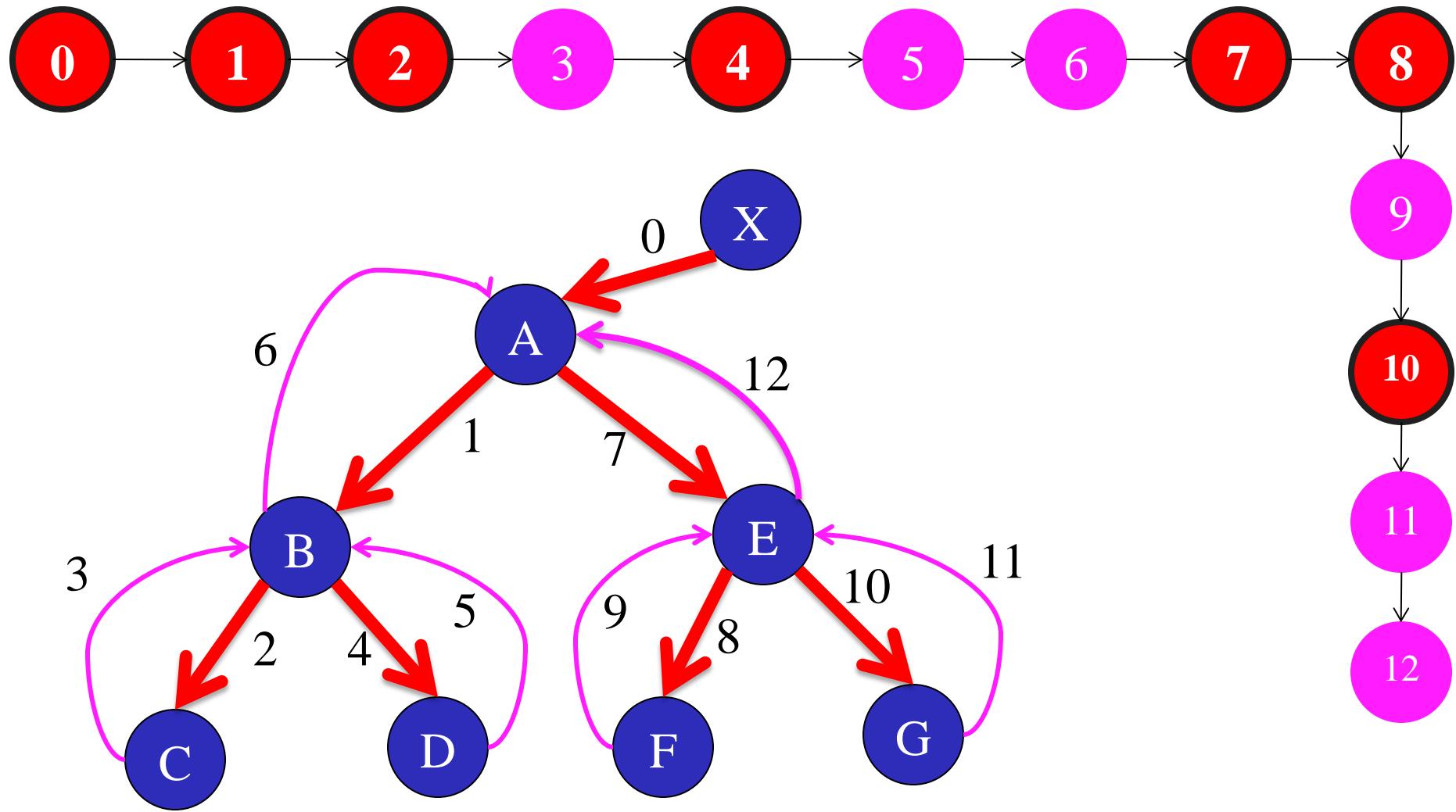
Parallel Depth-first Search –Example

Introduce a dummy node **X**



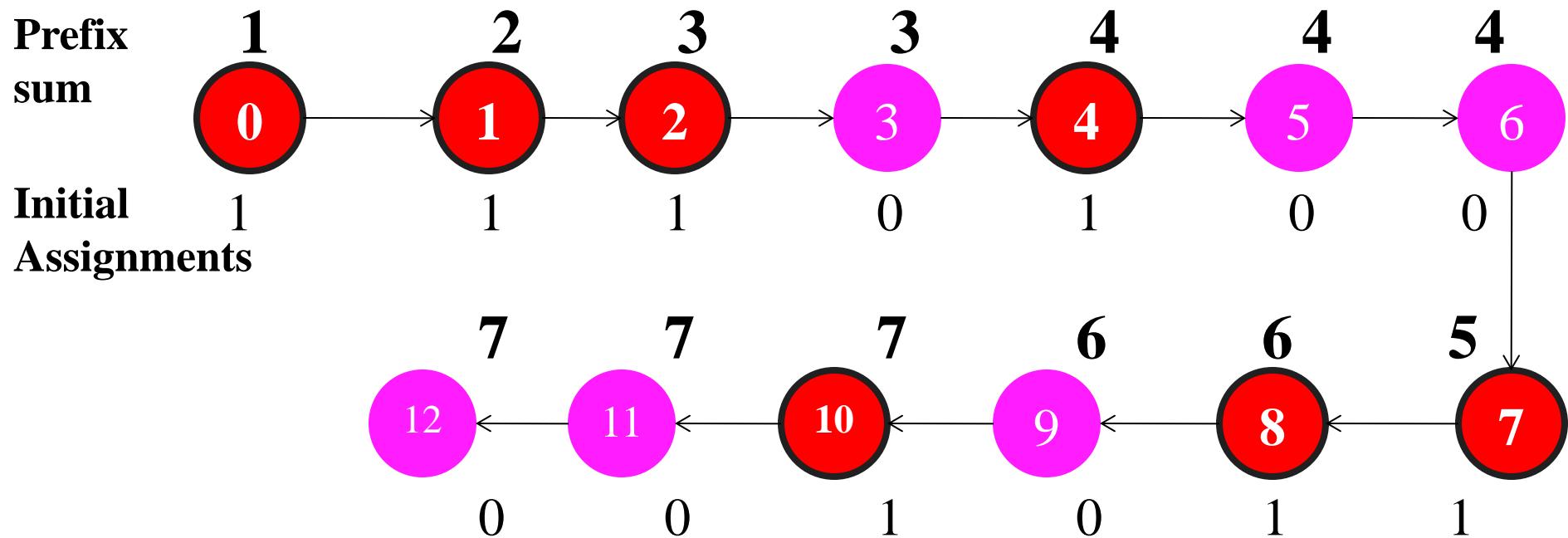
Parallel Depth-first Search –Example

List for this tree



Parallel Depth-first Search –Example

List for this tree



Parallel Depth-first Search –Example

Node	DFS			DFS order of the Vertex
	Prefix	Related Vertex	order of the Vertex	
	Sum	Edge	in Tree	
0	1	<X,A>	A	1
1	2	<A,B>	B	2
2	3	<B,C>	C	3
4	4	<B,D>	D	4
7	5	<A,E>	E	5
8	6	<E,F>	F	6
10	7	<E,G>	G	7

End

Part-1

Designing of Parallel Algorithms
using
Euler Tour Technique
(Part-2)

Euler Tour Technique

- **Some background**

- Let $T = (V, E)$ be a given tree and let $T' = (V, E')$ be a directed graph obtained from T
 - T : original tree
 - T' : Graph obtained by introducing magenta edges
- Each edge $(u, v) \in E$ is replaced by two edges $\langle u, v \rangle$ and $\langle v, u \rangle$
- So note that the number of vertices are same in T and T' , only more edges are introduced in T'

Euler Tour Technique

- **Some background ...*contd***
 - Now whole idea to do all this is to construct an Eulerian graph
 - Eulerian graph: recall it is a graph where we can traverse all the nodes of the graph by going through each edge exactly once
 - Eulerian graph is only possible if a graph has indegree and outdegree same
- **In newly constructed graph T'**
 - Both the **indegree** and **outdegree** of an internal node of the tree are now same
 - The indegree and outdegree of a leaf is **1** each
 - Hence T' is an Eulerian graph

Euler Tour Technique

- Now if we go back and see our graph with red and magenta edges, we see that by introducing the extra edges we have satisfied the condition for Eulerian graph
- Now look at any vertex, number of edges coming in and going out are same
- For leaves also, we have $\text{indegree}=\text{outdegree}$ and it is **1** here

Euler Tour Technique

More definitions

- Euler circuit:
 - An Euler circuit of a graph is an edge-disjoint circuit which traverses all the nodes
 - To note, we want to traverse all the nodes because if we want to compute tree functions then we have to have this property
- A graph permits an Euler circuit if and only if each vertex has equal indegree and outdegree
 - Which is satisfied in the new graph that we construct

Euler Tour Technique

- How to construct Euler circuit?
 - If we have to construct Euler circuit, then we have to specify the successor edge for each edge
 - And we have to do this (i.e. specifying the successor edge) in parallel
 - we can't do this sequentially because then it will kill whole idea, of making algorithm fast

Euler Tour Technique

- How to construct Euler circuit? ...contd
 - Once list of successor edge is there, we have to simply do list ranking by assigning weights to different nodes of the list to implement various tree functions

Constructing an Euler Tour

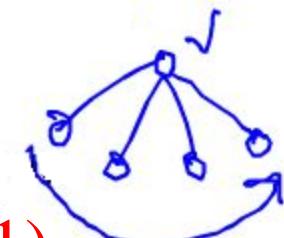
- **Procedure**

- Each edge on an Euler circuit has a unique successor edge, (as we know already)
- For each vertex $v \in V$, we fix an ordering of the vertices adjacent to v . *[using their vertex numbers assuming that each vertex has been assigned a number]*
- If d is the degree of vertex v , the vertices adjacent to v are:

$$adj(v) = \langle u_0, u_1, \dots, u_{d-1} \rangle$$

- The successor of edge $\langle u_i, v \rangle$ is:

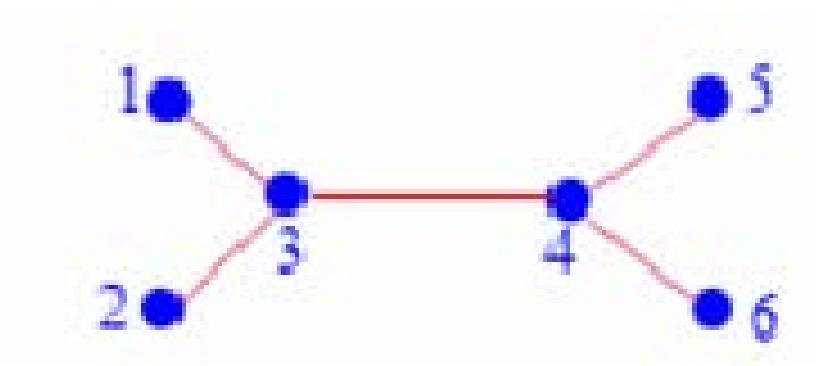
$$s(\langle u_i, v \rangle) = \langle v, u_{(i+1) \bmod d} \rangle, 0 \leq i \leq (d-1)$$



Constructing an Euler Tour

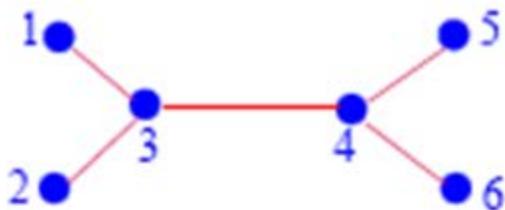
- The successor of edge $\langle u_i, v \rangle$ is:
 $s(\langle u_i, v \rangle) = \langle v, u_{(i+1) \bmod d} \rangle, 0 \leq i \leq (d - 1)$
 - Here the value of $u_{(i+1) \bmod d}$ is basically u_{i+1}
 - This little complicated notation used so that the last node points to the first node
 - This is since for last node, say $i=3$ (**when we have 4 adjacent nodes**), so $(3+1) \bmod 4 = 0$,
 $s(\langle u_3, v \rangle) = \langle v, u_0 \rangle$

Constructing an Euler Tour

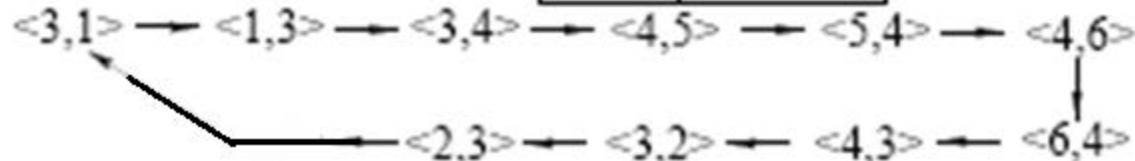


- **An Example:**
 - Consider above tree, remember we have not rooted it yet
 - Now you can see why rooting is so important, because if we get this particular tree and are asked to tell who is parent of ‘3’, we can’t answer that
 - This is because if I chose ‘4’ as root, then ‘4’ becomes its parent, however, if I choose ‘1’ as the root then ‘1’ becomes the parent
- As of now, we will work with an un-rooted tree, and then we will see how to root a tree

Constructing an Euler Tour



v	adj(v)	edge	successor
1	3	<3,1>	<1,3>
2	3	<3,2>	<2,3>
3	2, 1, 4	<2,3>	<3,1>
4	3, 5, 6	<1,3>	<3,4>
5	4	<4,3>	<3,2>
6	4	<3,4>	<4,5>
		<5,4>	<4,6>
		<6,4>	<4,3>
		<4,5>	<5,4>
		<4,6>	<6,4>

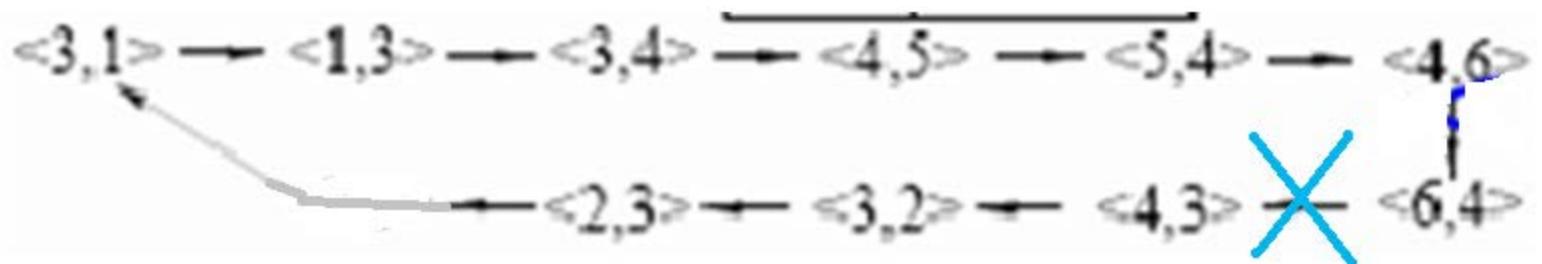


Constructing an Euler Tour

- As we have seen in the previous example that after computing the successor for every edge, **we get a circularly linked list**
 - Other words this list goes around and around
- Now we cannot do anything useful unless we break this list, because we cannot do list ranking in a circularly linked list
- Actually there is no meaning of list ranking because there is no start of the list, there is no end of the list
- **So we have to break this list somewhere so that we get a linear linked list**

Constructing an Euler Tour

- Once we have linear linked list, we can perform list ranking
- And actually breaking a circularly linked list to linear linked list is nothing but introducing root for the tree
- So now for example, if we want node ‘4’ to be the root of the tree, we break it at certain point as below so that every thing make sense with ‘4’ as a root
- We will see this in detail later



Constructing an Euler Tour

- Before going further, let us prove a very basic thing
 - This is very crucial if the Euler tour technique has to be a correct technique
- The idea here is that when we are introducing an edge and a reverse edge (or replacing each edge by two edges), are we really creating a single circular linked list or we creating multiple node disjoint list
- If we are not creating single circular list, then we are in trouble because now the tree has gone, and from a single tree we have created multiple lists, but this is not that really we want

Constructing an Euler Tour

- So the lemma in the next slide proves that the successor function, as we have defined creates only one cycle, and not a set of edge disjoint cycles
- We can prove the lemma through induction that we have only a single linked list

Correctness of Euler Tour

- Consider the graph $T' = (V, E')$, where E' is obtained by replacing each $e \in E$ by two directed edges of opposite directions

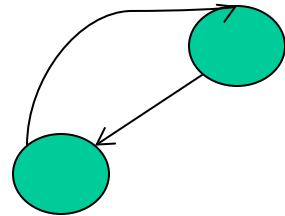
Lemma: The successor function s defines only one cycle and not a set of edge-disjoint cycles in T'

Proof: We have already shown that the graph is Eulerian

- We prove the lemma through induction

Correctness of Euler Tour

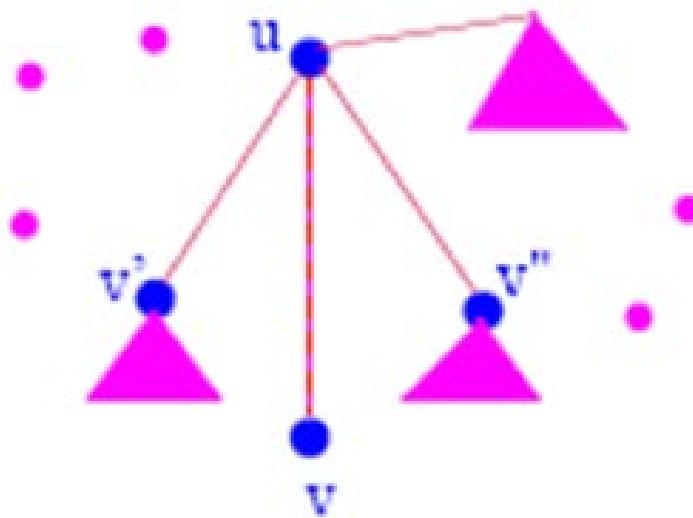
Basis of Induction: When the tree has 2 nodes, there is only one edge and one cycle with two edges



To show that if, suppose, the claim is true for n nodes

Then, we should show that it is true when there are $n + 1$ nodes

Correctness of Euler Tour



- Now let us assume that the whole tree, except node v , was present
- We can introduce an extra node by introducing a **leaf** to an existing tree, like the leaf v
- Initially, $\text{adj}(u) = \langle \dots, v', v'', \dots \rangle$
Hence, $s(\langle v', u \rangle) = \langle u, v'' \rangle$

Correctness of Euler Tour

- After the introduction of v ,

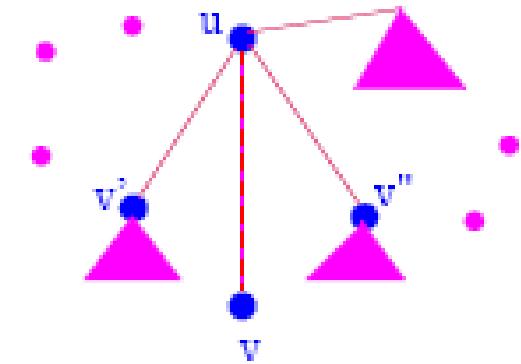
$$adj(u) = \langle \dots, v', v, v'', \dots \rangle$$

- So now $s(\langle v', u \rangle) = \langle u, v \rangle$

- and

$$s(\langle v, u \rangle) = \langle u, v'' \rangle$$

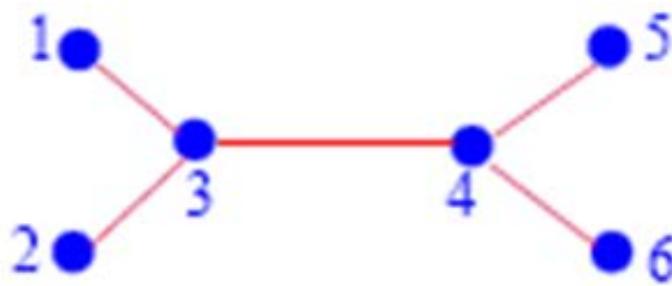
- So in other words what we are doing by inserting this new vertex is just that we are extending the already existing cycle, and we are not introducing a new cycle
- Hence, there is only one cycle after v is introduced



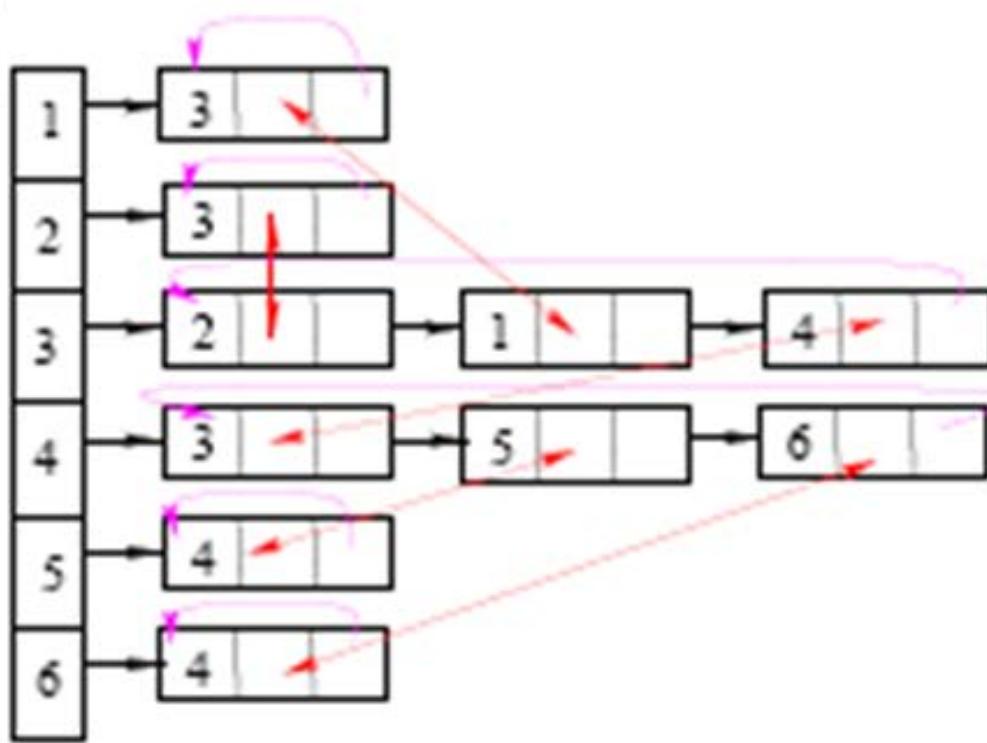
Construction of Euler Tour in Parallel

- Now the question is that, how do we construct Euler tour in parallel?
 - Basically, we have the adjacency information, we can write down the adjacency information as shown on next slide
 - For every node, we can write down the nodes which are adjacent to it and we introduce **some arrows just to keep the list in a circular list** because, we know that when we are trying to find the successor of the last edge, we have to use that modulo operation, that means we have to come back to the first edge

Construction of Euler Tour in Parallel



v	adj(v)
1	3
2	3
3	2, 1, 4
4	3, 5, 6
5	4
6	4



Construction of Euler Tour in Parallel

- There are two types of arrows in the previous example
- Magenta arrow:
 - to show the adjacency list of a node
- Red arrow
 - to tell the adjacency list of a particular node which may be the part of a adjacency list of another node
 - For a particular node, red arrow tells exact location of the adjacency list of that node, so that we can go there in constant time and pickup an successor edge

Construction of Euler Tour in Parallel

- We assume in all PRAM algorithm that
 - Input is given in an array
 - We also know how an input for a list can be given in an array
 - Every array location is indexed by the number of the node and that location will contain the index of the successor node, that is how we give linked list in an array

Construction of Euler Tour in Parallel

- We assume that the tree is given as a set of adjacency lists for the nodes. The adjacency list $L[v]$ for v is given in an array
- Consider a node v and a node u_i adjacent to v
- We need:
 - The successor for $\langle u_i, v \rangle$ which is $\langle v, u_{(i+1) \bmod d} \rangle$
 - This is done by making the list circular
 - $\langle u_i, v \rangle$. This is done by keeping a direct pointer from u_i in $L[v]$ to v in $L[u_i]$

Construction of Euler Tour in Parallel

- **Example:**
- If for v , $\text{adj}(v) = \{u_0, u_1, u_2\}$

$$\text{Then } s(u_0, v) = (v, u_1)$$

$$s(u_1, v) = (v, u_2)$$

$$s(u_2, v) = (v, u_3)$$

$$s(u_3, v) = (v, u_0)$$

Construction of Euler Tour in Parallel

- It is now possible to allocate one processor for **every element of this list**
- There are many adjacency lists
- For every element in the adjacency list we can allocate one processor which can pickup the successor in constant time for that particular edge
- This is what meant by constructing a Euler tour in $O(1)$ time

Construction of Euler Tour in Parallel

- We can construct an Euler tour in $O(1)$ time using $O(n)$ processors
- One processor is assigned to each node of the adjacency list
- There is no need of concurrent reading, hence the EREW PRAM model is sufficient

Points to Note

- Here we are just constructing Euler tour given a tree
- We are not including the cost of list ranking here
 - This is because the construction of Euler tour is independent of what you do with it
- Once we have Euler tour for a tree, we can perform variety of computations and tree functions using that

Points to Note

- So since every processor is assigned to an element of the list, we can do it without any read conflict
- That is a processor can decide on the successor without conflicting with another processor so it can be done using EREW model

End

Rooting a tree

- We will discuss parallel algorithm for rooting a tree
- Further we will discuss how to contract a tree
 - Tree contraction will serve basis for arithmetic expression evaluation

Why is rooting important?

- For doing any tree computation, we need to know the parent $p(v)$ for each node v
- Hence, we need to **root** the tree, say at a vertex ' r '

Rooting a Tree

- How do we choose vertex ‘*r*’
 - As such, there is no particular choice which is important
- However, in the algorithm that we will discuss
 - It allows us to specify a vertex which we like to make as a root, if we have any choice
 - And it will root the tree at that vertex

Rooting a tree

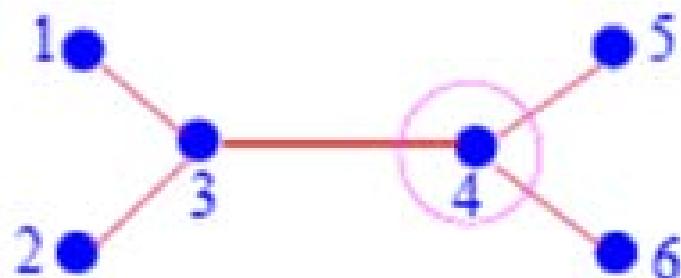
- When we say that the tree will be rooted at ‘r’, it **will not be physically rooted** at ‘r’, but it will construct a linked list considering ‘r’ as a root
- As we have seen earlier we had a circular linked list
- To get ‘r’ as the root, we need to break the circular list at a suitable point and make it a single linked list
- Then we can do list ranking and derive all the tree functions

Rooting a tree

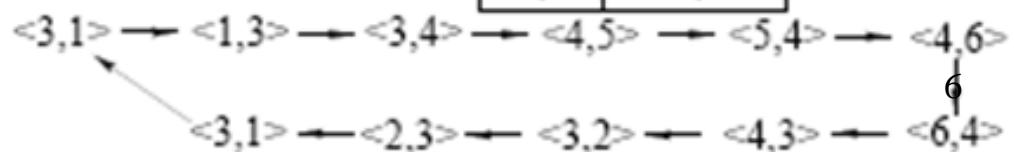
- To root a tree
 - We first construct an Euler tour
 - Euler tour gives a circular linked list
 - Then, for the vertex r , set $s(< u_{d-1}, r >) = 0$.
 - i.e. make $s(< u_{d-1}, r >)$ as null
 - where u_{d-1} is the last vertex adjacent to r
- In other words, we break the Euler tour at ' r '

Rooting a tree - Example

- Consider following graph and corresponding Euler tour, say we want to make '4' as the root



v	adj(v)	edge	successor
1	3	<3,1>	<1,3>
2	3	<3,2>	<2,3>
3	2, 1, 4	<2,3>	<3,1>
4	3, 5, 6	<1,3>	<3,4>
5	4	<4,3>	<3,2>
6	4	<3,4>	<4,5>
		<5,4>	<4,6>
		<6,4>	<4,3>
		<4,5>	<5,4>
		<4,6>	<6,4>

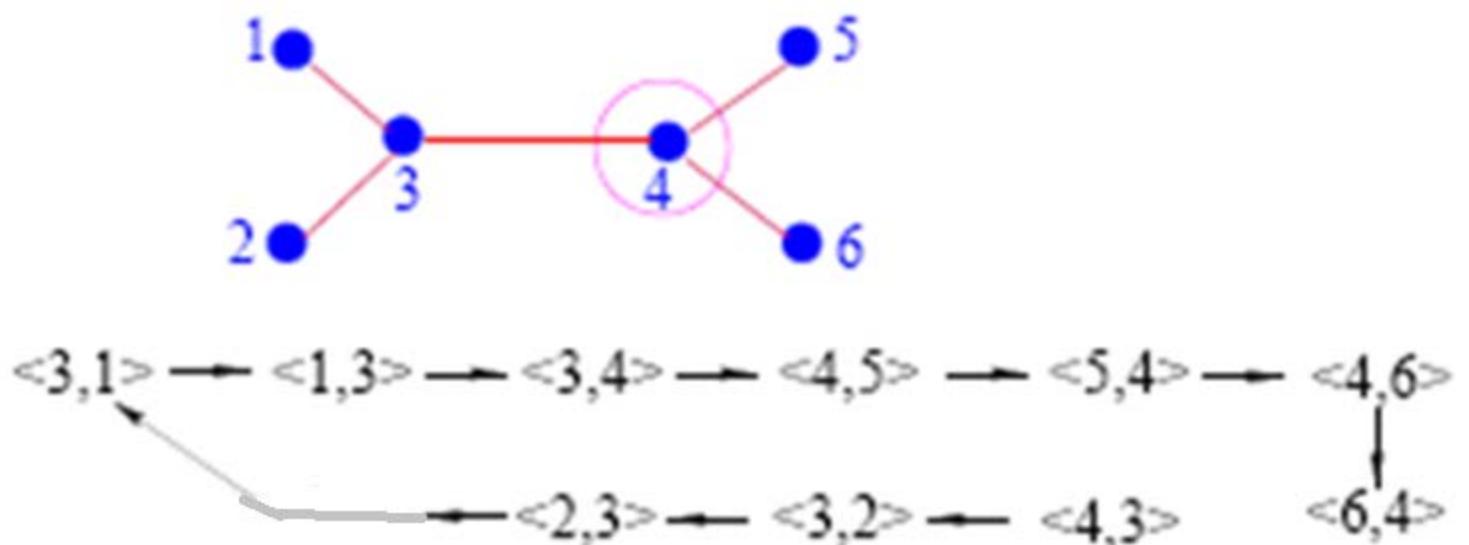


Rooting a tree - Example

- If we want to make, ‘4’ as the root, we analyze all its adjacent nodes
- Say, ordering among the vertices adjacent to ‘4’ is {3,5,6}, then the last node adjacent to ‘4’ is ‘6’ and we can make successor of $\langle 6,4 \rangle$ as null
- i.e. we can make $s(\langle 6,4 \rangle) = \text{NULL}$
 - [Note that originally $s(\langle 6,4 \rangle)$ would have been $\langle 4,3 \rangle$

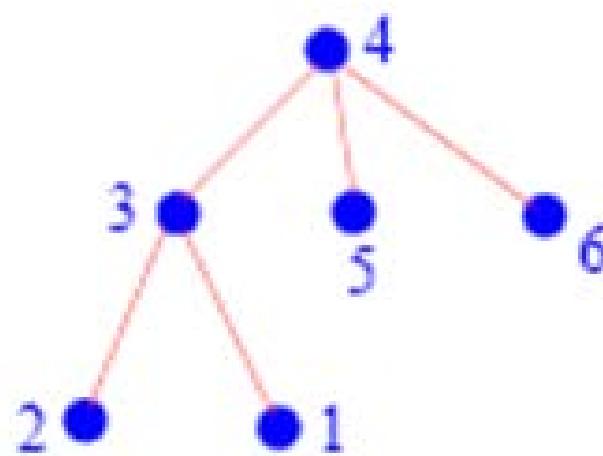
Rooting a tree - Example

- So in the table we delete the successor of $\langle 6,4 \rangle$ and make it zero
 - [see the table shown in previous slide]
- Circular list after breaking it at this point is shown below



Rooting a tree - Example

- After recognizing the root, we can redraw the tree as follows



$\begin{matrix} <4,3> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <3,2> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <2,3> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <3,1> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <1,3> \\ | \\ 1 \end{matrix}$

\downarrow

$\begin{matrix} <6,4> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <4,6> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <5,4> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <4,5> \\ | \\ 1 \end{matrix} \longrightarrow \begin{matrix} <3,4> \\ | \\ 1 \end{matrix}$

Computing $\text{Parent}(v)$

- Essentially, rooting a tree means establishing parent-child relationship for each node
- Now once we have obtained the root, we can perform various tree computations using parent-child relationship among the nodes
- Algorithm:
 - **Input:** The Euler tour of a tree and a special vertex r
 - **Output:** For each vertex $v \neq r$, we want the parent $p(v)$ of v in the tree with root ' r '

Computing $\text{Parent}(v)$

Algorithm

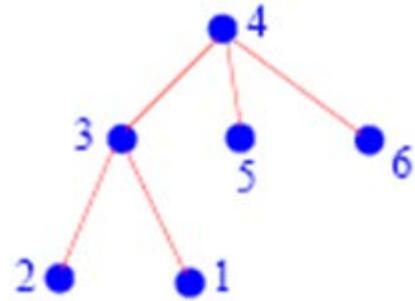
begin

1. **Breaking the cycle:** Set $s(< u, r >) = 0$, where u is the last vertex in the adjacency list of r .
2. **Assign a weight 1** to each edge of the list and compute parallel prefix.
3. **Setting the Parent:** For each edge $< x, y >$ in the list, set $x = p(y)$ whenever the prefix sum of $< x, y >$ is smaller than the prefix sum of $< y, x >$.

[This is due to the reason that parent will be visited before child]

end

Computing $\text{Parent}(v)$ -Example

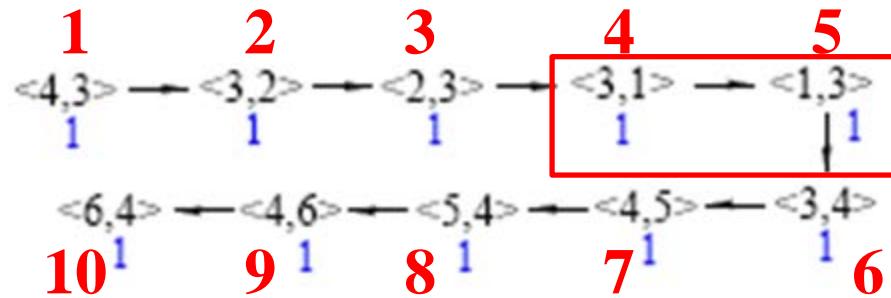
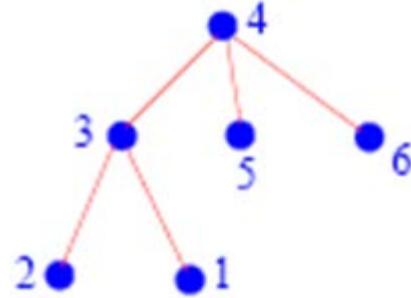


$\begin{matrix} <4,3> \\ 1 \end{matrix} \longrightarrow \begin{matrix} <3,2> \\ 1 \end{matrix} \longrightarrow \begin{matrix} <2,3> \\ 1 \end{matrix} \longrightarrow \begin{matrix} <3,1> \\ 1 \end{matrix} \longrightarrow \begin{matrix} <1,3> \\ 1 \end{matrix}$
 \downarrow
 $\begin{matrix} <6,4> \\ 1 \end{matrix} \longleftarrow \begin{matrix} <4,6> \\ 1 \end{matrix} \longleftarrow \begin{matrix} <5,4> \\ 1 \end{matrix} \longleftarrow \begin{matrix} <4,5> \\ 1 \end{matrix} \longleftarrow \begin{matrix} <3,4> \\ 1 \end{matrix}$

$x = p(y) \quad \text{if}$
prefix sum of $\langle x,y \rangle$ is smaller than
prefix sum of $\langle y,x \rangle$

- Consider root of the tree as ‘4’
- Assign number 1 to each edge of the list obtained after breaking the circular list
- Perform prefix sum
- Now let us say we want to know the relationship of ‘3’ with ‘1’

Computing $\text{Parent}(v)$ -Example



$x = p(y)$ if

prefix sum of $\langle x, y \rangle$ is smaller than
prefix sum of $\langle y, x \rangle$

- Prefix sum of $\langle 3,1 \rangle$ is 4 and prefix sum of $\langle 1,3 \rangle$ is 5 hence 3 is the parent of 1
- Complexity: this is within the complexity of list ranking
- So assuming list ranking best complexity, we can do it in $O(n)$ work and $O(\log n)$ time

Computation of tree functions

- Given a tree T , to compute many tree functions:
 - We first construct the Euler tour of T
 - Then we root the tree at a vertex
- Afterwards, we can compute:
 - The **postorder** number of each vertex
 - The **preorder** number of each vertex
 - The **inorder** number of each vertex
 - The **level** of each vertex
 - The **number of descendants** of each vertex

Computation of leaf nodes in the tree

- Various functions can be computed very easily by just assigning appropriate weights to the nodes and by performing prefix sum
- How can we get leaf in a rooted tree using Euler tour technique?
 - *Hint: In an Euler path for a rooted tree, the leaves appear from left to right*

Computation of **Number of descendants of a node**

- **Number of descendants of a node:** for this just see the difference of the value of the prefix sum when you first came to the node until the time you leave that node

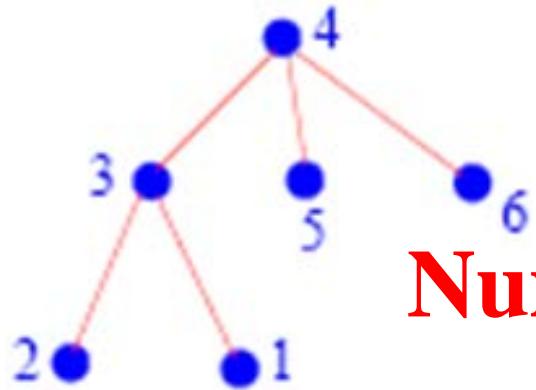
Set $w(u, \text{parent}(u)) = 1$, $w(\text{parent}(u), u) = 0$.

- ▶ Each node u is counted once by counting its “back edge” $u \rightarrow \text{parent}[u]$.
- ▶ $\text{value}(u, \text{parent}[u]) - \text{value}(\text{parent}[u], u)$ counts all $v \rightarrow \text{parent}[v]$ edges traversed after visiting u for the first time and before leaving u for the last time.

Hence $\text{size}(u) =$

$$\begin{cases} n-1 & u \text{ is the root,} \\ \text{value}(u, \text{parent}(u)) - \text{value}(\text{parent}(u), u) - 1 & \text{otherwise} \end{cases}$$

is the number of descendants of u .



Example –

Number of descendants of a node

Edge	Edge-weight	Prefix sum
<4,3>	0	0
<3,2>	0	0
<2,3>	1	1
<3,1>	0	1
<1,3>	1	2
<3,4>	1	3
<4,5>	0	3
<5,4>	1	4
<4,6>	0	4
<6,4>	1	5

Node	Descendants Count Formula	Descendants Count Value
1	$[\text{value}(<1,3>) - \text{value}(<3,1>)] - 1$	$1 - 0 - 1 = 0$
2	$[\text{value}(<2,3>) - \text{value}(<3,2>)] - 1$	$1 - 0 - 1 = 0$
3	$[\text{value}(<3,4>) - \text{value}(<4,3>)] - 1$	$3 - 0 - 1 = 2$
4	$[n - 1]$	$6 - 1 = 5$
5	$[\text{value}(<5,4>) - \text{value}(<4,5>)] - 1$	$4 - 3 - 1 = 0$
6	$[\text{value}(<6,4>) - \text{value}(<4,6>)] - 1$	$5 - 4 - 1 = 0$

Example –

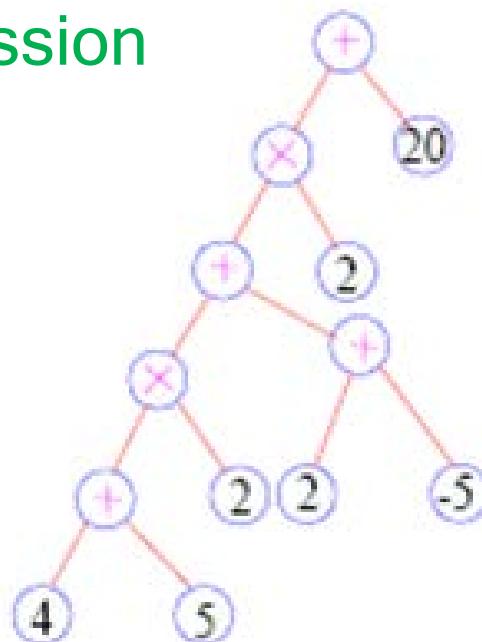
Number of descendants of a node

- In previous approach we have seen
 - $w(<p(u), u>) = 0$ and $w(<u, p(u)>) = 1$
 - In this case
 $\text{size}(u) = \text{prefix-sum}(<u, p(u)>) - \text{prefix-sum}(<p(u), u>) - 1$
- Alternative Approach:
 - $w(<p(u), u>) = 1$ and $w(<u, p(u)>) = 0$
 - In that case
 $\text{size}(u) = \text{prefix-sum}(<u, p(u)>) - \text{prefix-sum}(<p(u), u>)$
- Third Approach
 - Assign weight 1 to all edges and then
 $\text{size}(u) = (\text{prefix-sum}(<u, p(u)>) - \text{prefix-sum}(<p(u), u>) - 1)/2$

Tree Contraction

- Some tree computations cannot be solved efficiently with the Euler tour technique alone
- For example, An important problem is evaluation of an arithmetic expression given as a binary tree
- Tree contraction serves the basis of computing the solution of an arithmetic expression

$$((4 + 5) * 2 + (-5 + 2)) * 2 + 20$$



Tree Contraction

- It is important to know that we don't have any control over the size and nature of the expression tree
 - It is not necessarily be a balanced tree
- An expression tree can be **quite thin and long**
- How do we evaluate the tree?
 - To evaluate the tree, we start from the bottom and try to shrink leaf with parent by performing the operation mentioned in the parent node on values at leaves

Tree Contraction

- In sequential computing
 - there are algorithms which use prefix or postfix notation of the expression and solve them **using stack**
- For parallel computation
 - if tree would have been balanced, we could have shrinked and evaluated the tree in the same way as we computed parallel sum of an array elements or max of n numbers

Tree Contraction

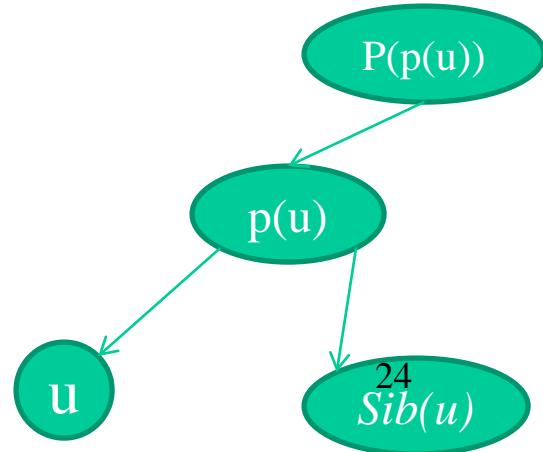
- We do not have balanced tree always
 - However, we still want to evaluate the expression in $O(\log n)$ time
- To achieve we will first see the **algorithm which allows us to shrink the tree** very fast
- Then we will use it in evaluation of arithmetic expressions

Tree Contraction

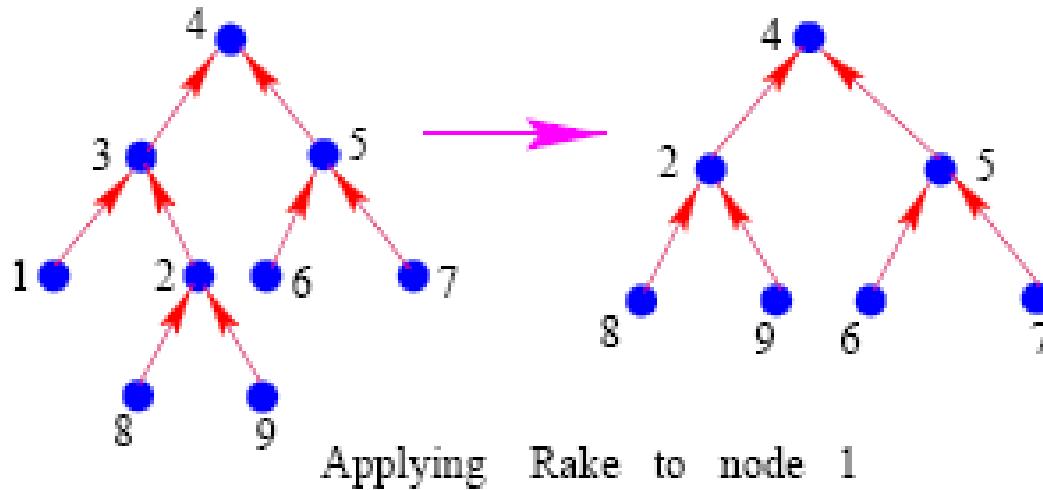
- In an arithmetic expression tree:
 - each leaf holds a constant and each internal node holds an arithmetic operator like $+, *$
- The goal is to compute the value of the expression at the root
- The tree contraction technique is a systematic way of shrinking a tree into a single vertex
- We successively apply the operation of merging a leaf with its parent or merging a degree-2 vertex with its parent [RAKE Operation]

The RAKE operation

- Let $T = (V, E)$ be a rooted binary tree and for each vertex v , $p(v)$ is its parent
- $sib(u)$ is the child of $p(u)$. We consider only binary trees
- RAKE operation is applied at a leaf
- In the RAKE operation for a leaf u such that $p(u) \neq r$
 - Remove u and $p(u)$ from T , and
 - Connect $sib(u)$ to $p(p(u))$



The RAKE operation



- In our tree contraction algorithm, we apply the rake operation repeatedly to reduce the size of the binary tree
- We need to apply rake to many leaves in parallel in order to achieve a fast running time

The RAKE operation

- While applying RAKE operation parallel in many leaves, **we need to be little careful**
- We cannot apply RAKE operation to nodes whose **parents are consecutive** on the tree
- For example:
 - We can't apply RAKE operation on leaf '1' and '8' at the same time as there is a conflict
 - If we remove both, it would not be clear that who will be the child of node '4'
- **We need to apply the rake operation to non-consecutive leaves as they appear from left to right**

The RAKE operation

- To perform RAKE operation neatly, we do following
 - We first label the leaves consecutively from left to right
 - In an Euler path for a rooted tree, the leaves appear from left to right
 - Leaf can be identified easily, as for example, if '*u*' is a leaf then in Euler path, two consecutive edges will occur in the form $\langle p(u), u \rangle, \langle u, p(u) \rangle$
 - If '3' is a leaf the two consecutive edges will appear like $\langle 1, 3 \rangle, \langle 3, 1 \rangle$ in Euler tour

The RAKE operation

- We can assign a weight **1** to each edge of the kind $(u, p(u))$ where u is a leaf
- We exclude the leftmost and the rightmost leaves
 - These **two leaves** will be the **two children of the root** when the tree is contracted to a three-node tree
- We do a prefix sum on the resulting list and the leaves are numbered from left to right

The RAKE operation

- We now store all the n leaves in an array A
- A_{odd} is the subarray consisting of the odd-indexed elements of A
- A_{even} is the subarray consisting of the even-indexed elements of A
- We can create the arrays A_{odd} and A_{even} in $O(1)$ time and $O(n)$ work
 - [This is just done by looking at the values of prefix sum, so actually A_{odd} and A_{even} are not created in reality, we just check the even and odd indexed elements by their values]

Tree Contraction Algorithm

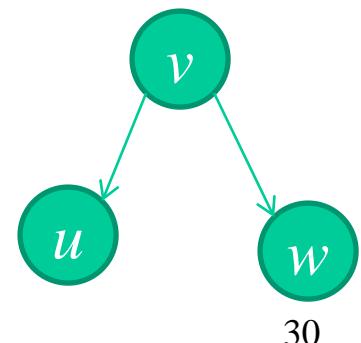
begin

for $\lceil \log(n+1) \rceil$ iterations do

1. Apply the rake operation in parallel to all the elements of A_{odd} that are **left children**
2. Apply the rake operation in parallel to the rest of the elements in A_{odd} . [After RAKing A_{odd} , it becomes empty]
3. Set $A := A_{even}$

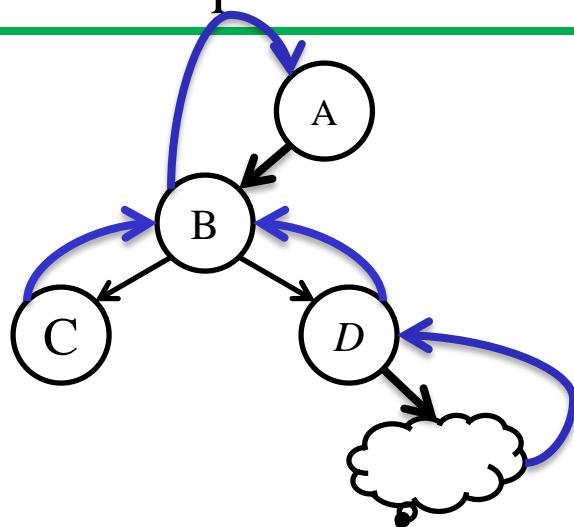
end

RAKE operation is applied on A_{odd} two times (first for all left children and then on the right children) to avoid conflicts

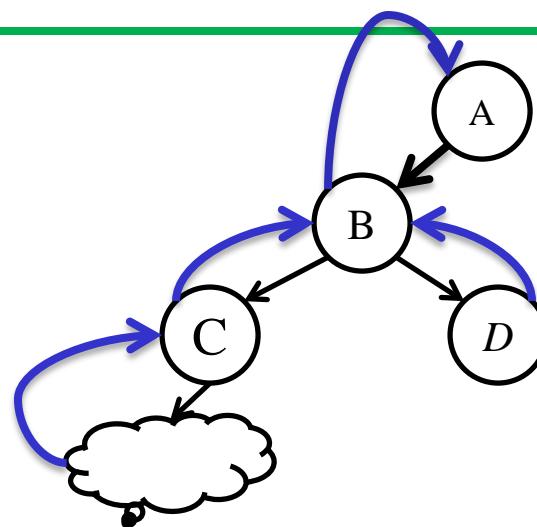


Left or Right Child

- How do you find out whether a node is left child or right?
- Can be done by analyzing the Euler tour. See following example



$\langle A, B \rangle$ → $\langle B, C \rangle$ → $\langle C, B \rangle$ → $\langle B, D \rangle$ → $\langle D, \dots \rangle$
→ $\langle \dots, D \rangle$ → $\langle D, B \rangle$ → $\langle B, A \rangle$



$\langle A, B \rangle$ → $\langle B, C \rangle$ → $\langle C, \dots \rangle$ → → $\langle \dots, C \rangle$
→ $\langle C, B \rangle$ → $\langle B, D \rangle$ → $\langle D, B \rangle$ → $\langle B, A \rangle$

Note the location of $\langle A, B \rangle$ and $\langle B, A \rangle$ in the Euler tour

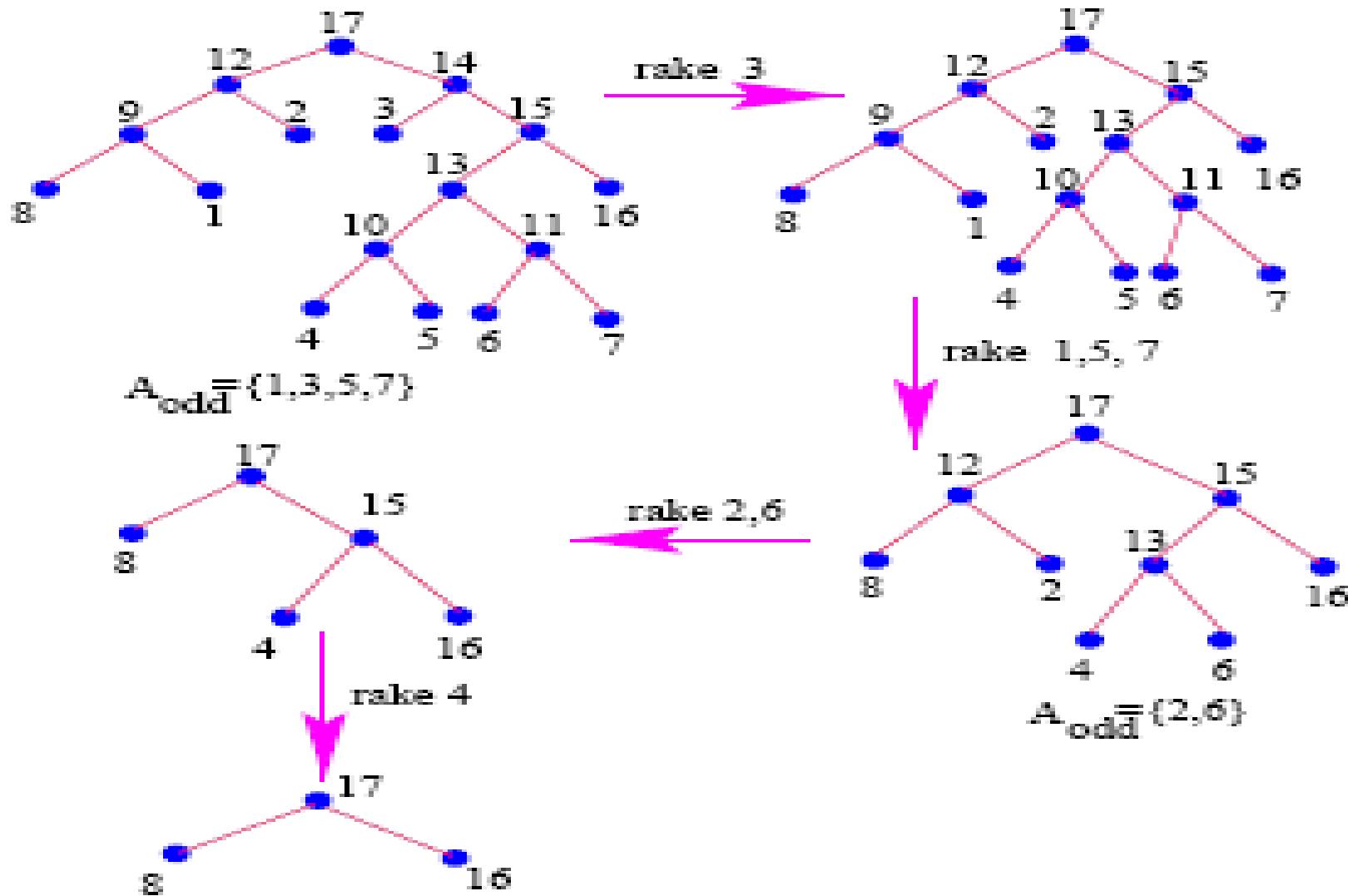
Left child: $\langle A, B \rangle$ is followed by leaf node edges

Right Child: $\langle B, A \rangle$ is preceded by the leaf node edges

Example

- In the example given in the next slide, we will not perform rake operation on the node ‘8’ and ‘16’ as they are the leftmost and rightmost children respectively
- Every RAKE operation is constant time, as we know that to perform it we have to just do some pointer adjustments

Tree contraction algorithm



Correctness of tree contraction

- Whenever the rake operation is applied in parallel to several leaves, the parents of any two such leaves are not adjacent
- The number of leaves reduces by half after each iteration of the loop
- Hence the tree is contracted in $O(\log n)$ time
- Euler tour takes $O(n)$ work
- The total number of operations for all the iterations of the loop is:

$$O\left(\sum_i \left(\frac{n}{2^i}\right)\right) = O(n)$$

End

Parallel Evaluation of Arithmetic Expression

(Use of Tree Contraction Technique)

Introduction

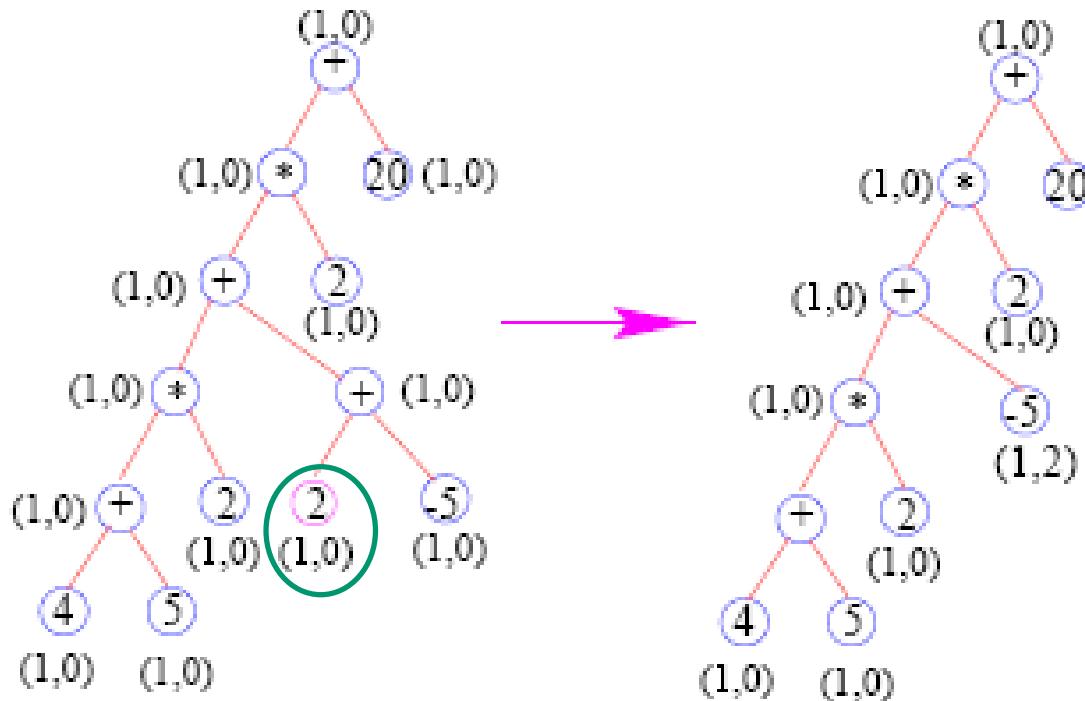
- Just to get motivated, let us first start with an example given in the next slide
- After that we will see the algorithm
- We will again re-visit the same example at the end
- **Basic Idea:**
 - We need to perform tree contraction at a **leaf** (also referred as **RAKE operation**)
 - The idea is that whenever we are performing RAKE operation, we have to keep track whatever is happening at the internal node because internal node should correctly capture the value of the sub-tree hanging at that node

Introduction

- This is done by assigning some levels to internal nodes and manipulating these levels
 - We will first see how these levels are manipulated and then we will see the algorithm
 - As we know, in the tree, leaves are numbers and internal nodes are operators
- We will start with leaf nodes assigning level (1,0) for every leaf node
 - This means value of that node is $(1*x+0)=x$, where x is the number at that leaf node
 - So by assigning level (1,0) to all leaf, we still get the value same as the original
 - We will see how we assign levels to other internal nodes

Introduction

- We are doing RAKing at node ‘2’



$$(1 * 2 + 0) + (-5 * 1 + 0) = (1 * -5 + 2)$$

$(1,2)$

Introduction

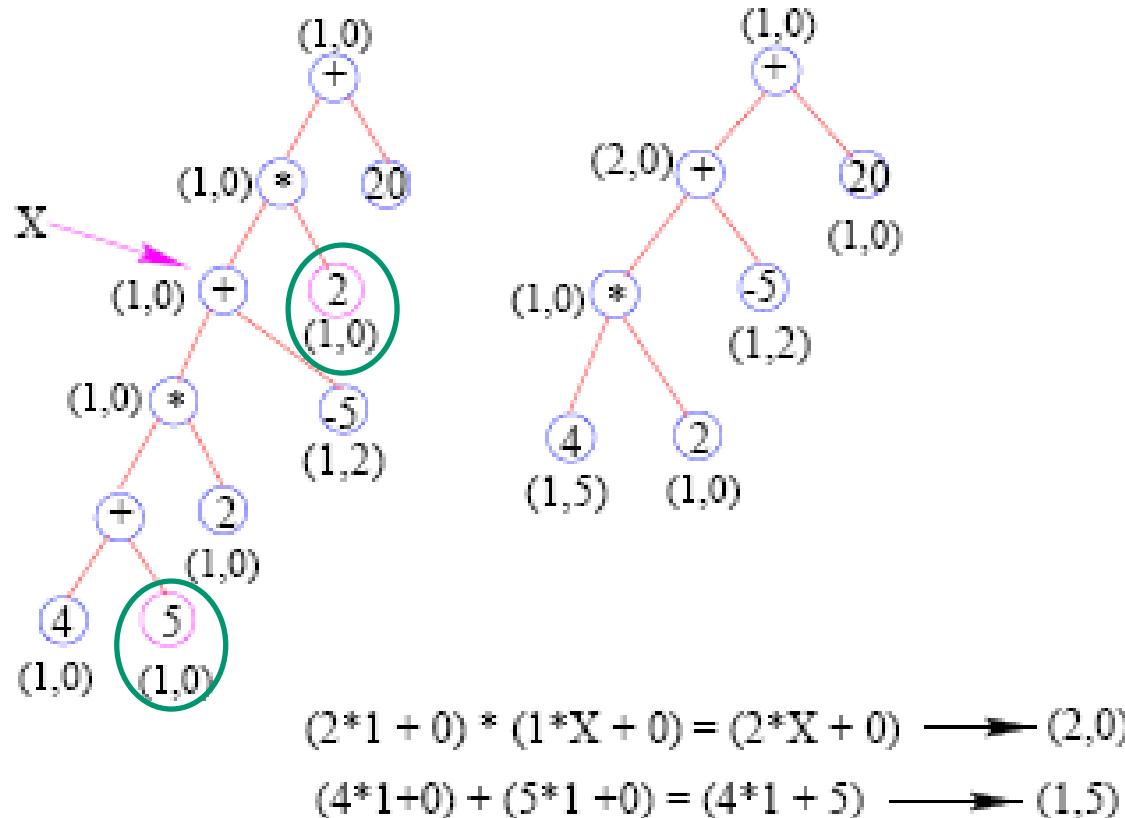
- Now main question is that when we do RAKing, we should not destroy the operation that we are doing in that part of the tree
- When we perform RAKE operation at node ‘2’, node ‘-5’ goes up and gets connected at ‘+’, we change the level of the node as (1,2)
- This is done since the value of node ‘2’ is $2*1 + 0$ and value of node ‘5’ is $5*1+0$, so combining both of them $[(2*1 + 0) + (-5*1 + 0)] = (-5*1 + 2)$, so node ‘-5’ can get new level as (1,2)
- By this trick, we maintain the correct value of the sub-tree by changing the level of the node
- **This level manipulation can be done in O(1) time**

Introduction

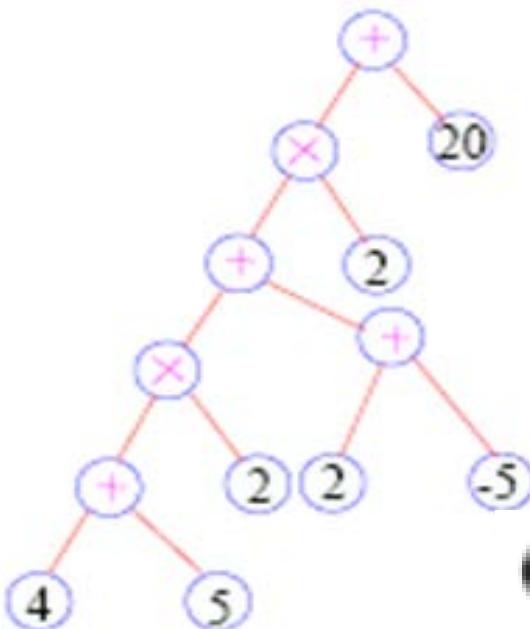
- In previous example, we have to do simple sub-tree movement, however, how do we do level manipulation when we have to move a very big sub-tree
- Example:
 - Let us see the RAKing of node ‘2’ with level (1,0) [see next slide]
 - So we have to remove ‘*’-‘2’ link and move the sub-tree with ‘+’ as root up
 - Now since we have not evaluated the sub-tree with root ‘+’, we don’t know its value, can we still do raking? and answer is yes, we can
 - Assume that the value of the node with ‘+’ as the root is, say ‘X’.
 - Then we can compute the level of new node as follows
 - $(2*1+0)*(1*X+0) = (2*X+0) \rightarrow (2,0)$, so (2,0) is the new level
 - Its interesting to see that without knowing the actual value of the node, we can still fix the new level

Introduction

- Now to perform RAKE operation at node '5', can change the level as (1,5) as explained below



Evaluation of arithmetic expressions



$$((4 + 5) * 2 + (-5 + 2)) * 2 + 20$$

- If we evaluate an expression tree **bottom-up**, it will take **$O(n)$** time for a long and skinny tree
- Hence we apply **tree contraction**

Evaluation of arithmetic expressions

- summary

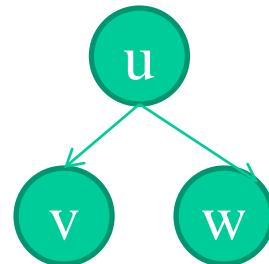
- We do not completely evaluate each internal node, we evaluate the internal nodes partially
- For each internal node v , we associate a label (a_v, b_v) where a_v and b_v are constants
- The value of the expression at node is:
 $(a_v X + b_v)$, where X is an unknown value for the expression of the sub-tree rooted at v
- So we capture the evaluation in the level

Keeping an invariant in Algorithm

Invariant:

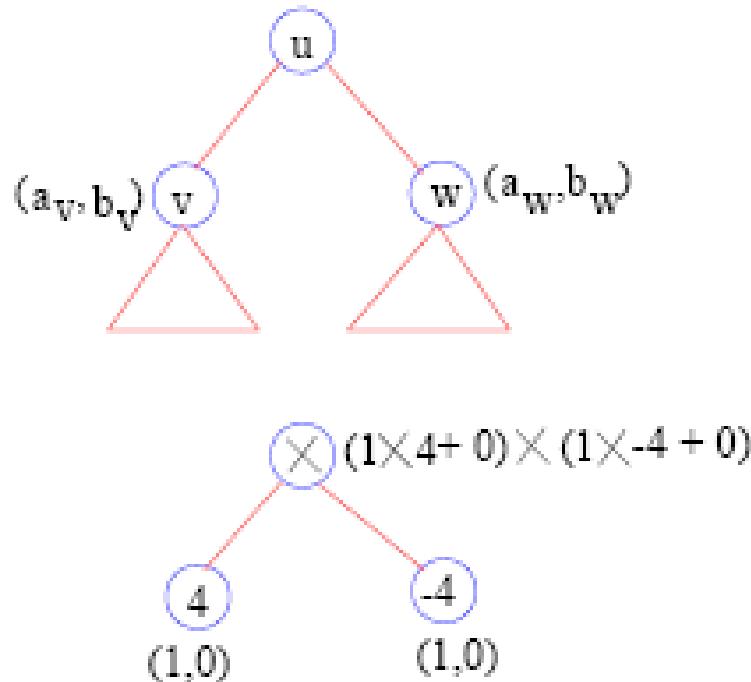
- Let u be an internal node which holds the operation $\oplus \in \{+, \times\}$.
- Let v and w are the children of u with labels (a_v, b_v) and (a_w, b_w) .
- Then the value at u is:

$$val(u) = (a_v val(v) + b_v) \oplus (a_w val(w) + b_w)$$

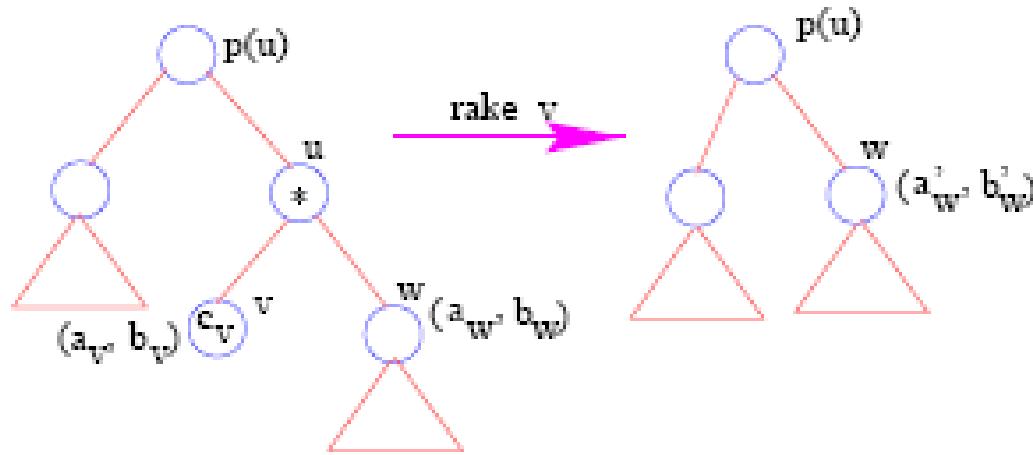


Keeping an invariant in Algorithm

- An example for a general case



Applying the rake operation



- The value at node u is [if we rake node ' v ']:
$$val(u) = (a_v c_v + b_v) \times (a_w X + b_w)$$
- X is the unknown value at node w

Applying the rake operation

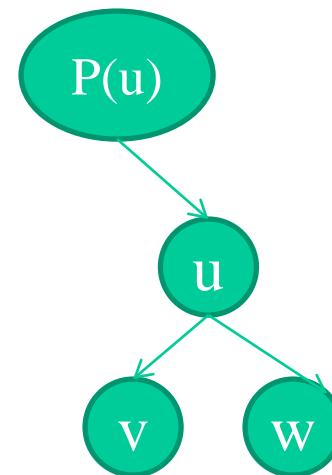
- The contribution of $val(u)$ to the value of node $p(u)$ is:

$$\begin{aligned} a_u \times val(u) + b_u &= a_u[(a_v c_v + b_v) \times (a_w X + b_w)] + b_u \\ &= [a_u(a_v c_v + b_v)a_w]X + a_u(a_v c_v + b_v)b_w + b_u \end{aligned}$$

- We can adjust the labels of node w to (a'_w, b'_w)

$$a'_w = a_u(a_v c_v + b_v) a_w$$

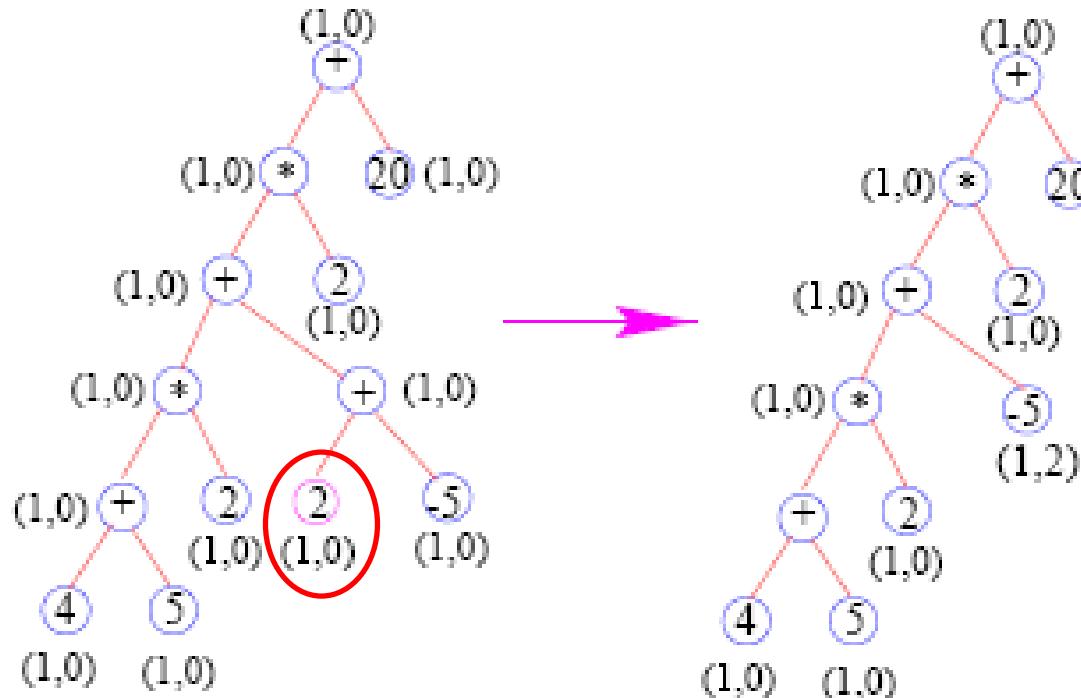
$$b'_w = a_u(a_v c_v + b_v) b_w + b_u$$



Complexity of expression evaluation

- The correctness of the expression evaluation depends on correctly maintaining the invariants
- We start with a label $(1, 0)$ for each leaf and correctly maintain the invariant at each rake operation.
- We have already proved the correctness of the rake operation
- Hence, evaluation of an expression given as a binary tree takes $O(n)$ work and $O(\log n)$ time
- We get the same complexity whether we start from the tree or from the Eulerian tour

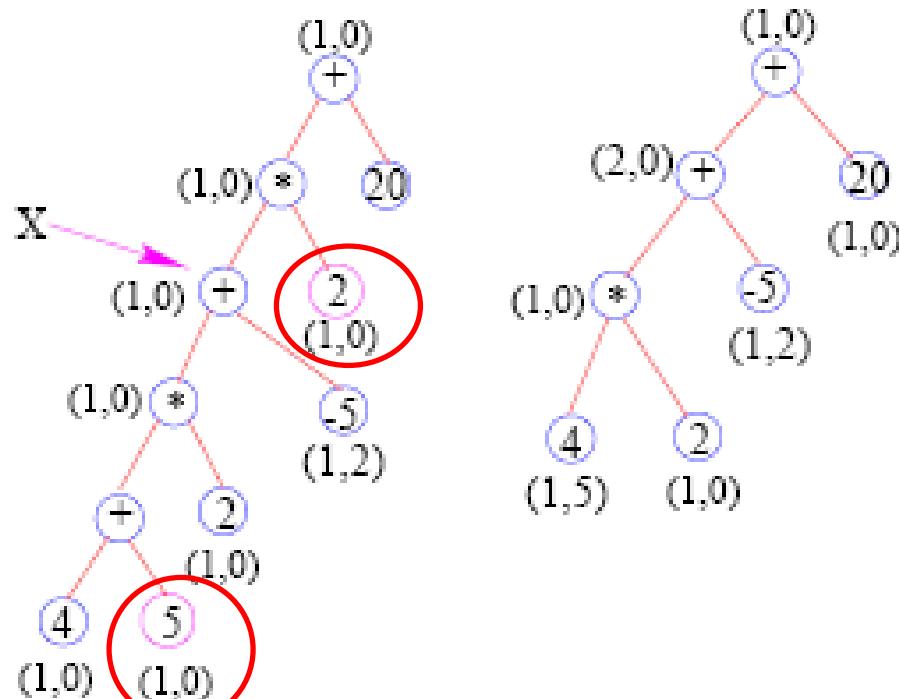
Example - *revisited*



$$(1 * 2 + 0) + (-5 * 1 + 0) = (1 * -5 + 2)$$
$$(1,2)$$

We do the RAKE operation at node '2' [picking the odd numbered leaves in parallel]

Example - *revisited*

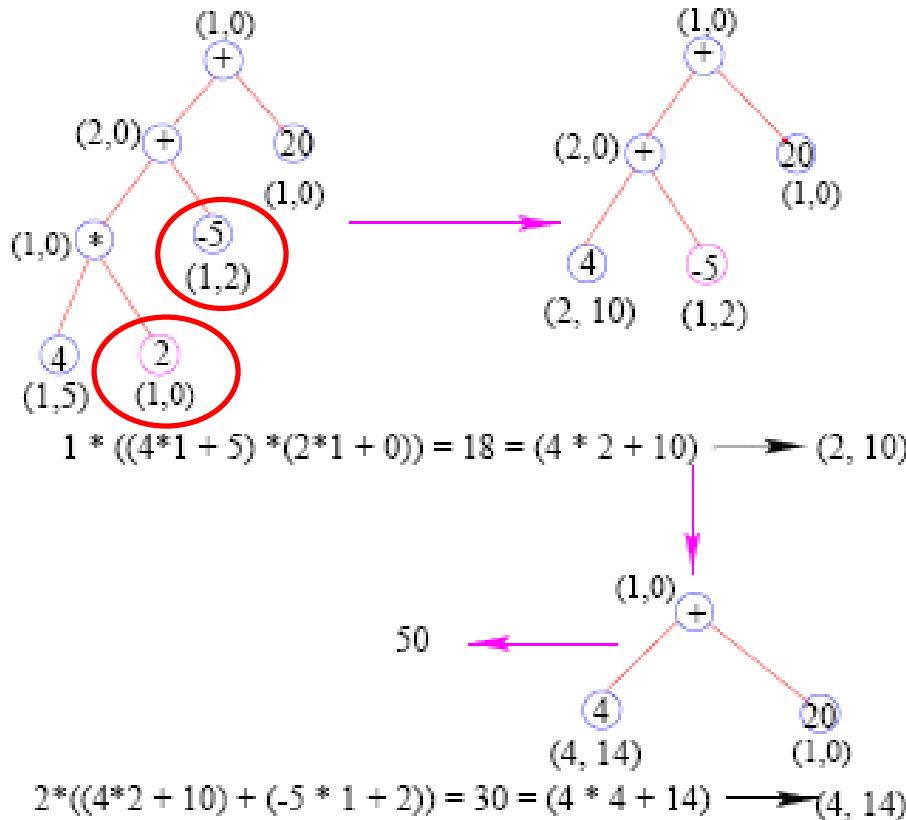


$$(2 * 1 + 0) * (1 * X + 0) = (2 * X + 0) \longrightarrow (2, 0)$$

$$(4*1+0) + (5*1+0) = (4*1 + 5) \longrightarrow (1,5)$$

We do the RAKE operation in parallel at ‘5’ and ‘2’ [picking the odd numbered leaves in parallel] 16

Example - *revisited*

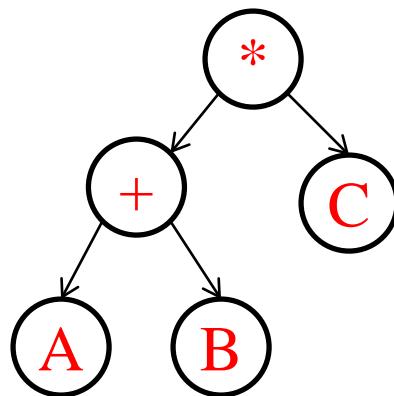


We do the RAKE operation at '2' and later '-5' [not in parallel as their parent are adjacent]

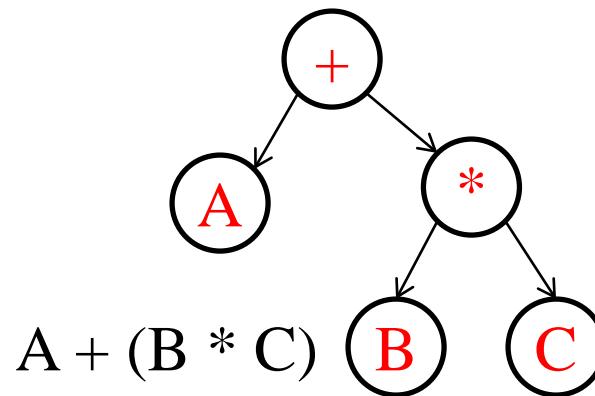
Once the tree has only three nodes, we directly evaluate the complete tree. 17

Notes

- What is the effect of operator priorities in evaluation?
 - We operate on the expression tree for parallel evaluation of arithmetic expressions
 - So it is assumed that when tree has been constructed, priority of the operators have been taken care
 - Parallel evaluation just operate on the expression tree and has nothing to do with the operators priorities
 - See the below example:
 - Evaluation will be different depending on the input expression tree



$$(A + B) * C$$



$$A + (B * C)$$

End

Parallel Algorithms for Searching

Overview

- A parallel search algorithm
- A parallel merging algorithm
 - Will require good searching algorithm
 - If we have a good parallel searching algorithm, we can do merging better
- A parallel sorting algorithm
 - Requires a good parallel merging algorithm

Overview

- Approach in developing all these algorithms
 - We will first start with an algorithm which is **not work optimal**
 - Then we will reduce the input in some way to **make it work optimal**

A parallel search algorithm

- We know that binary search performs quite good
- If we have only one processor, and n elements, we can perform searching in $O(\log n)$ time
- Now idea here is that can we do better than this if we have multiple processors
 - Development of parallel algorithm
- **The parallel algorithm discussed here is similar to binary search**

A parallel search algorithm

- We discuss an algorithm for parallel searching in a sorted array
- Say n is the number of elements in a sorted array S and we use p , $p < n$ processors for searching
- The complexity of the parallel searching algorithm is:
$$O\left(\frac{\log n}{\log(p+1)}\right)$$
- **We notice that this time is much better than sequential binary search**

A parallel search algorithm

Input:

- (i) A sorted array $S = x_1, x_2, \dots, x_n$ with n elements.
- (ii) A query element y .

Output: Two elements x_i, x_{i+1} in S such that
 $x_i \leq y \leq x_{i+1}$.

A parallel search algorithm

- Basic Idea → Reduce Search space
 - Reduce the search space as done in binary search
- This is similar to reducing the size of the array
 - as we have already done in many parallel algorithms

A parallel search algorithm

- The algorithm consists of **a series of iterations**
 - This is to reduce the size of the array where the element y is located.
- **In each iteration:**
 - we divide the current array in $p + 1$ equal parts and locate y in one of the parts.
 - This is continued until the size of the array where y is located is reduced to p .

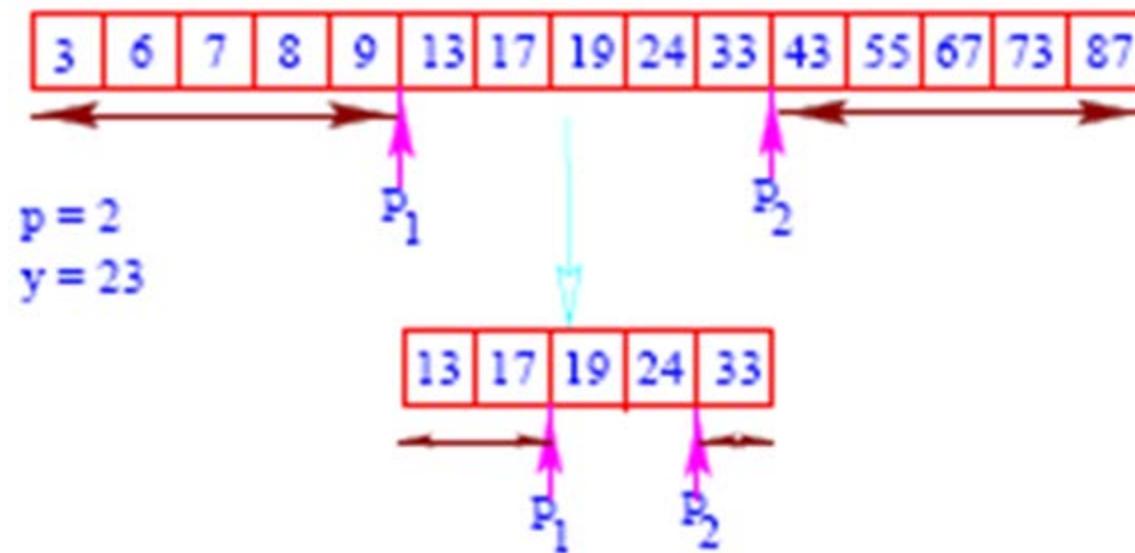
A parallel search algorithm



- Then we allocate one processor to each element and perform comparison to check condition $x_i \leq y \leq x_{i+1}$ for every element in parallel in $O(1)$ time
- Direct comparisons are done to find two elements x_i, x_{i+1} such that $x_i \leq y \leq x_{i+1}$
- So this is the trick to reduce elements from n to p

A parallel search algorithm

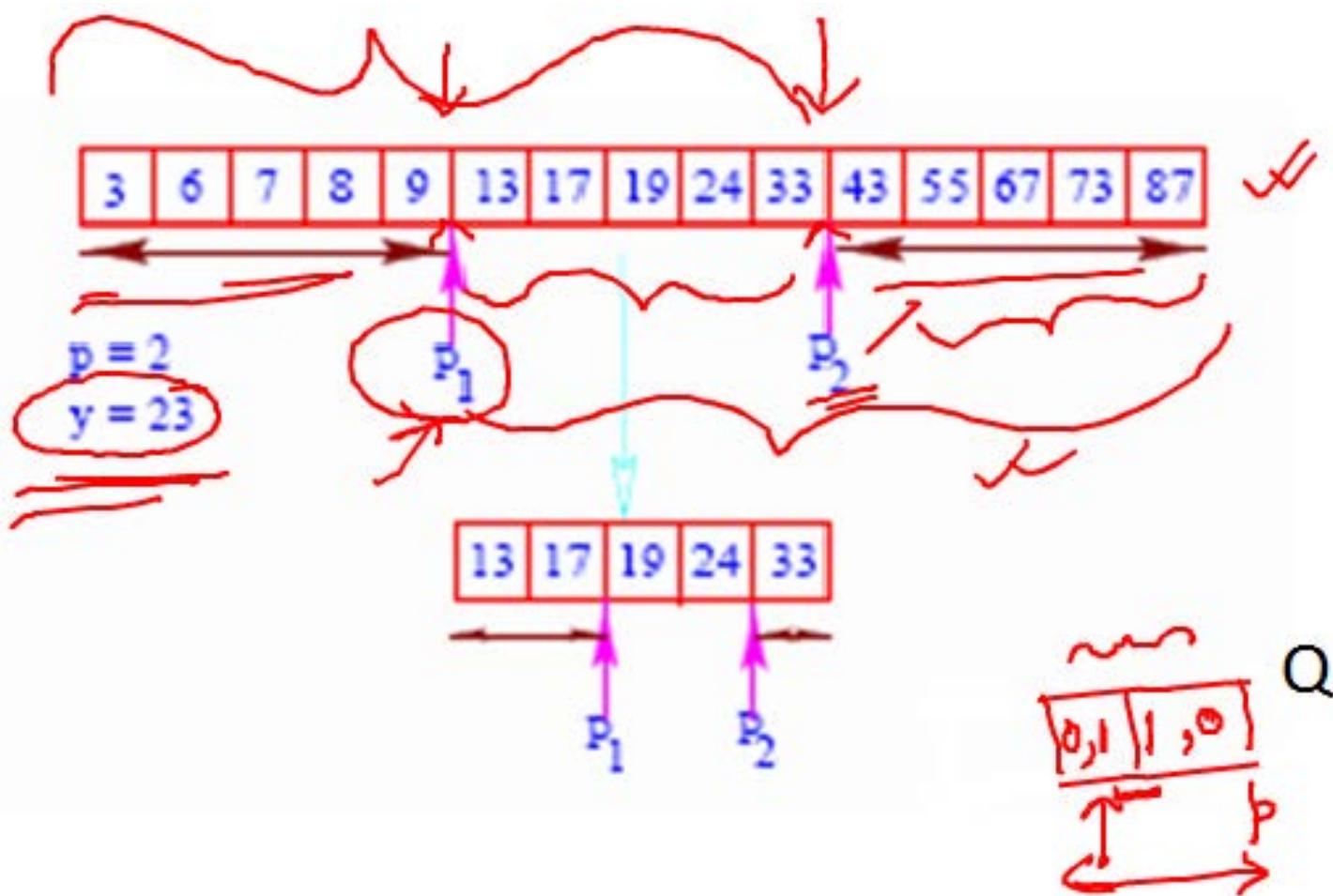
- **Example:** with two processors
 - Knowing the number of processor, we can allocate parts of the array to the processors
 - In example below, first part (actually the last element of the part) is given to P_1 and second part (last element) to P_2



A parallel search algorithm

- For each of the $p + 1$ parts of the array, a processor checks whether $y < x_l$, where x_l is the last element of the part
- If $y < x_l$, the sub-array to the right of x_l can be rejected. If $y > x_l$, the sub-array to the left of x_l can be rejected
- In each iteration we identify only one sub-array for further search

A parallel search algorithm



A parallel search algorithm

- This gives an array Q , of size p , with each cell representing subpart of the array
- Each cell of the array contains number $(0,1)$ or $(1,0)$
- $(0,1)$ tells that element which is searched is present in the right of that sub-array whereas $(1,0)$ says that element which is searched is present in the left of that sub-array (refer previous slide)
- Once array Q is known, we have to just find out two subsequent cells i and $i+1$ with entries $(0,1),(1,0)$
- And further searched is required to be done in the sub-array allocated to processor $(i+1)$

A parallel search algorithm

- Iteration 1 reduces the size of the searched array from n to $n/(p+1)$
- This is repeated again and again
- When the size of the remaining array is the same as the number of processors, we do the following:
 - We allocate one processor for each element.
 - The processor assigned to element x_i checks whether $x_i \leq y \leq x_{i+1}$.
 - Hence it takes $O(1)$ time to locate y once the size of the array has been reduced to p .

Complexity

- We need to analyze what is the complexity of reducing the size of the array to p .
- At the first iteration, we are reducing the size of the array from n to $n/(p+1)$.
- Suppose, the size reduces to p after k iterations.

$$\frac{n}{(p+1)^k} = p$$

$$\log n = \log p + k \log(p+1)$$

$$k = \frac{\log n - \log p}{\log(p+1)}$$

$$k < \frac{\log n}{\log(p+1)}$$

$$k = O\left(\frac{\log n}{\log(p+1)}\right)$$

Complexity

- So complexity

$$O\left(\frac{\log n}{\log(p+1)}\right)$$

- If only one processor *i.e.* $p=1$, then the case is same as binary search and complexity will be

$$O\left(\frac{\log n}{\log(1+1)}\right) = O\left(\frac{\log n}{\log(2)}\right) = O(\log n)$$

Memory Model

- We need the CREW PRAM model
 - One processor needs to compare two neighboring values
 - We can do some manipulation to make it EREW

End

Parallel Algorithm for Merging

Merging

- In merging, we will make use of parallel search algorithm (already discussed)
- We have seen, for searching we need concurrent read facility (CREW)
 - We will see that even by using CREW model, we will be able to design a $O(\log \log n)$ algorithm
- This algorithm will be special in a sense because usually to get fast algorithm (for example $O(\log \log n)$ time algorithm), usually we need to use CRCW model, or even more powerful model

Merging

- We use the parallel search algorithm to design an optimal $O(\log\log n)$ time merging algorithm
- *Notations:*
 - $\text{rank}(x : X)$ is the number of elements of X that are $\leq x$.
 - **Ranking a sequence:** $Y = (y_1, y_2, \dots, y_m)$ in X is the same as computing the integer array : (r_1, r_2, \dots, r_m) where $r_i = \text{rank}(y_i, X)$.

Merging – an example

A	1	5	9	13	17	19	20	23	26
---	---	---	---	----	----	----	----	----	----

B	3	4	11	16	22	24	27	28	30
---	---	---	----	----	----	----	----	----	----

$$\text{rank}(20:A) = 7 \quad \text{rank}(20:B) = 4$$

$$\text{rank}(20: A+B) = 7 + 4 = 11$$

- If $\text{rank}(a_i, A) = r_1$ and $\text{rank}(a_i, B) = r_2$
- $\text{rank}(a_i, A + B) = r_1 + r_2$.
- Hence, a_i should go to the entry number $r_1 + r_2$ in the merged array.

Merging

- Basic idea of merging:
 - rank each element of B in array A and
 - Similarly, rank each element of array A in array B
 - Use ranks to get position of an element in merged array

Merging

- No ranking of all the elements together:
 - Idea here is not to do the ranking at once for all the elements, because if we do the ranking at once, we cannot achieve fast algorithm *i.e.* $O(\log \log n)$ time complexity
- First compute rank for a short sequence:
 - So first we would do the ranking for a short sequence into a big sequence,
- Then try to do exact merging
 - Try to build exact merge algorithm
- Next slide shows something which can be directly seen from the searching algorithm

Merging

- Merging is essentially a kind of searching
- In searching, we will keep the length of the short sequence as $m = O(n^s)$, where n is the length of long sequence in which searching is to be performed and m is the length of the short sequence.
- Usually we would take $s=0.5$, i.e. $m = \sqrt{n}$

Ranking a short sequence in a sorted sequence

- Say, X is a sorted sequence with n elements
- Y is an arbitrary sorted sequence of size m such that $m = O(n^s)$, where s is a constant and $0 < s < 1$.
- If we use $p = \left\lfloor \frac{n}{m} \right\rfloor = O(n^{1-s})$ processors, then we can rank each element of Y in X in $O(\frac{\log n}{\log p}) = O(1)$ time in $O(n)$ operations.

$$p = \alpha(m^{1-s})$$

$$\begin{aligned}\log p &= (1-s)\log m \\ \frac{\log n}{(1-s)\log m} &= O(1)\end{aligned}$$

A fast merging algorithm

- We now discuss a fast algorithm for merging two sorted arrays A and B with n and m elements each
 - Fast merging is an essential component in any sorting algorithm based on divide-and-conquer
- We will first design an $O(\log\log m)$ time and $O((m + n) \log\log m)$ work algorithm for fast merging

A fast merging algorithm

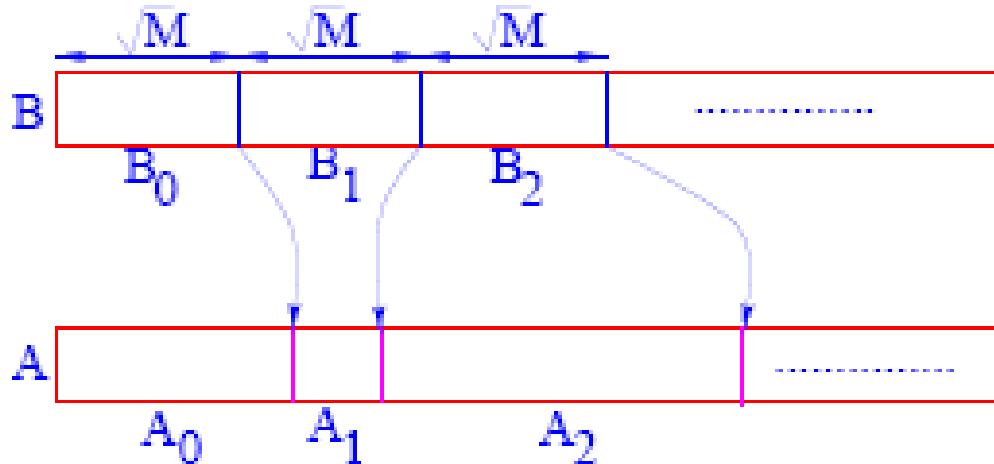
- Since we know that merging can be done sequentially in $O(m+n)$ time, hence above is not a work optimal
- So later, we will improve the work to $O(m+n)$ using some more tricks and refinement to make the algorithm work optimal

A fast merging algorithm

Problem:

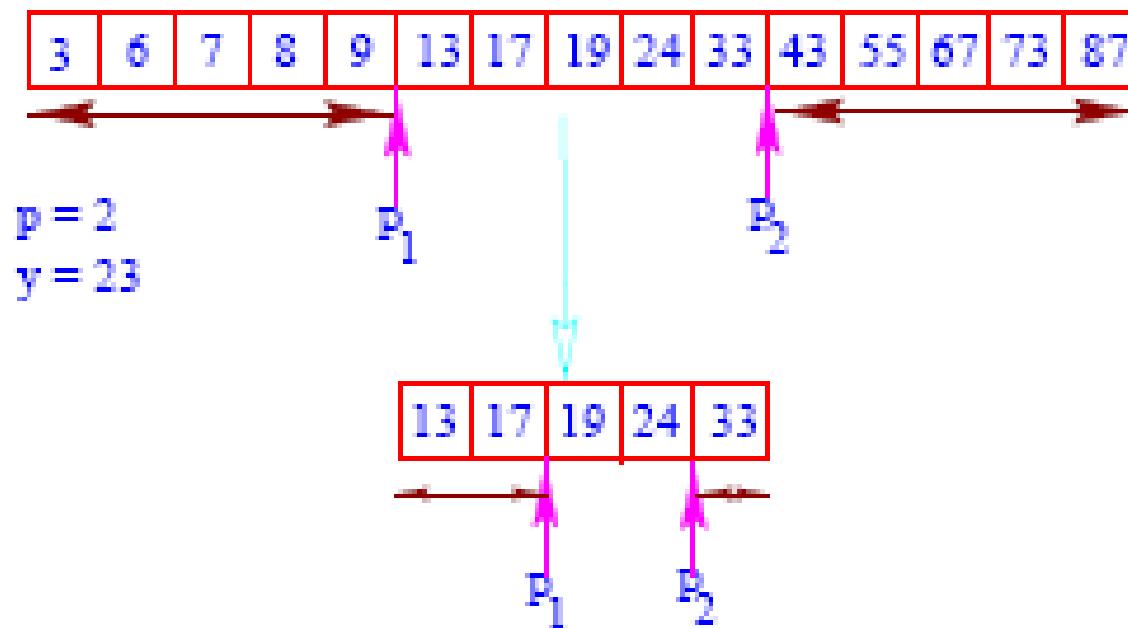
- ✓ Input: Two sorted sequences A and B of lengths n and m respectively.
- ✓ Output: $\text{rank}(B : A)$ and $\text{rank}(A : B)$
[as we know that if we have the ranks, we can merge two arrays using the rank information]
- ❖ We use a strategy similar to the searching algorithm we discussed earlier
- ❖ We divide the array B into \sqrt{m} parts, each part with \sqrt{m} elements
- ❖ We start with ranking the last element from each part of B into A

Ranking a sample of elements

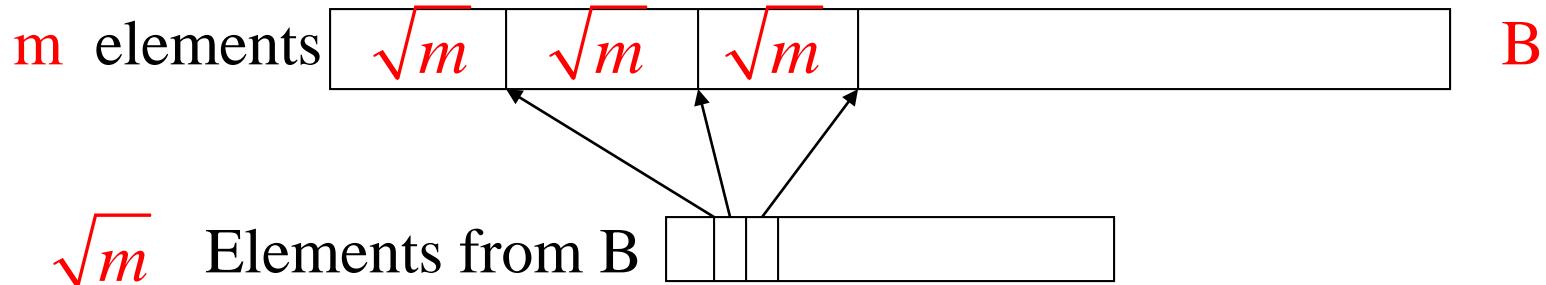


- We start with a sample of \sqrt{m} elements from B
- We choose every \sqrt{m} -th element from B
- These \sqrt{m} elements can be ranked in A in $O(1)$ time through binary search in parallel using m processors

A parallel search algorithm



Ranking of \sqrt{m} elements in $O(1)$ time



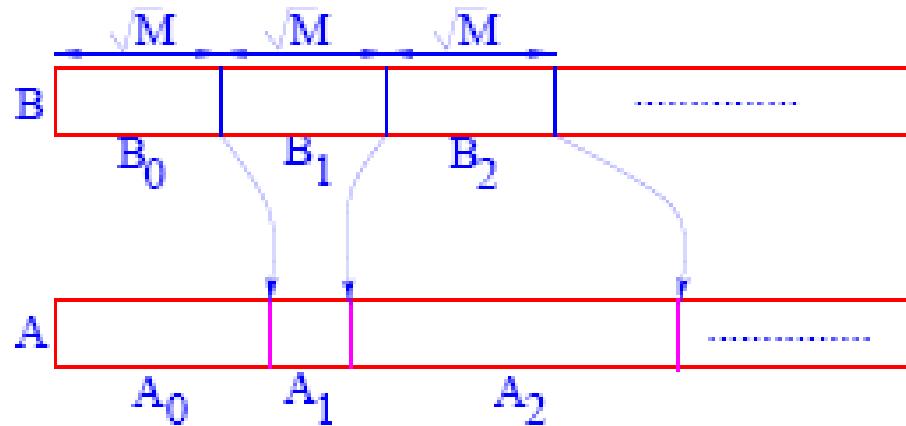
For simplicity (without loss of generality) consider that $m > n$

For every element in B , we allocate \sqrt{m} processors

In one step, we identify the block of \sqrt{m} elements in A where an element of B will be ranked

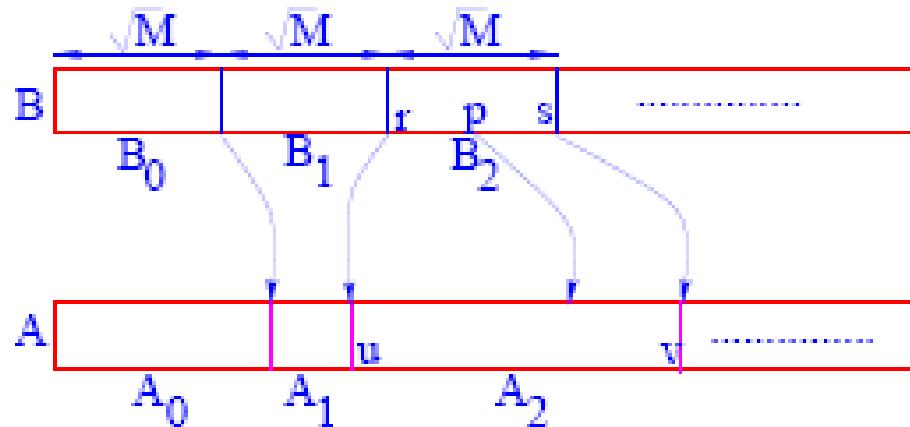
We find the rank in another step

Ranking a sample of elements



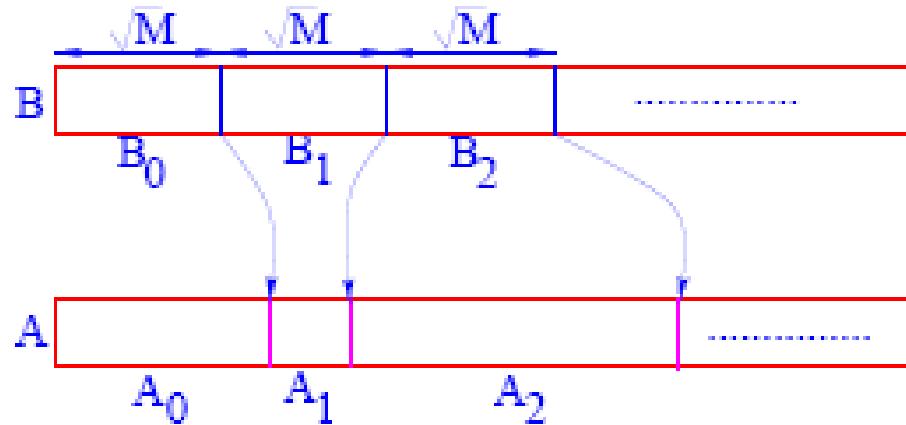
- B is partitioned into \sqrt{m} blocks, each of size \sqrt{m}
- After the ranking of the \sqrt{m} elements from B in A , A is also partitioned into \sqrt{m} blocks
- We can now merge the blocks in A and B pair-wise recursively

Independent subproblems



- Consider the first element of B_2 and the first element of B_3 , the elements r and s
- Now, r is ranked at u and s is ranked at v
- Consider an element p such that $r < p < s$, p must be ranked in between u and v
- Hence, all the elements in B_2 must be ranked in A_2 and vice versa

Ranking a sample of elements



- Suppose at the current level of recursion, the size of the two sub-problems B' and A' are m' and n'
- If $m' > n'$, then we divide B' into $\sqrt{m'}$ parts and apply the algorithm recursively.
- If $n' > m'$, we divide A' into $\sqrt{n'}$ parts and apply the algorithm recursively.

An example

1	5	9	13	17	19	20	23	26
---	---	---	----	----	----	----	----	----

3	4	11	16	22	24	27	28	30
---	---	----	----	----	----	----	----	----

1	5	9
---	---	---

13	17	19
----	----	----

20	23	26
----	----	----

3	4
---	---

11	16
----	----

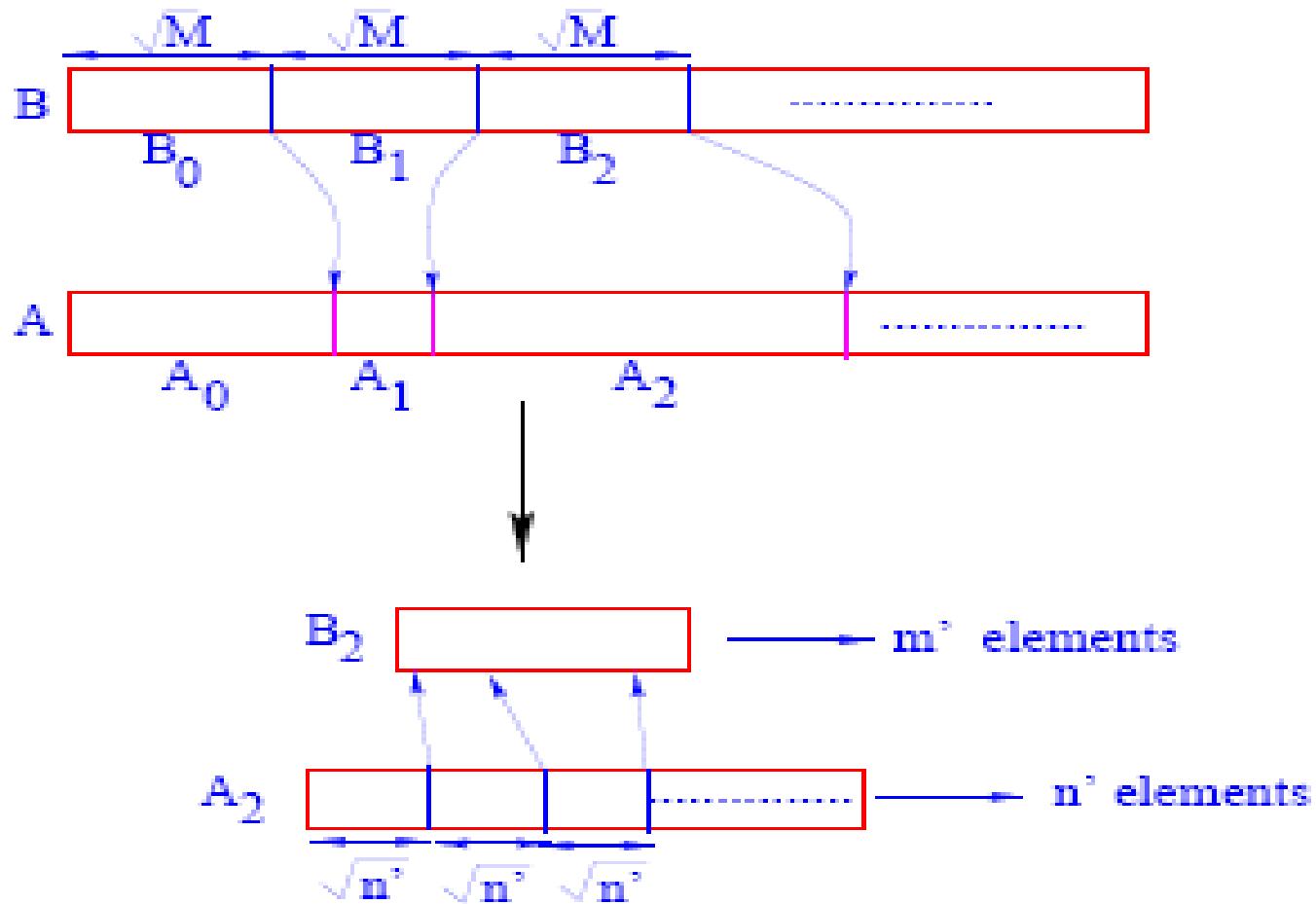
22	24	27	28	30
----	----	----	----	----

(i)

(ii)

(iii)

Recursion



Recursion

- The recursion between all pairs of blocks can be solved in parallel.
- The recursion stops when the size of the sub-problems is small and we can merge the very small blocks through a sequential algorithm in $O(1)$ time.
- At the end of the algorithm, we know $\text{rank}(B : A)$ and $\text{rank}(A : B)$. Hence, we can move the elements to another array in sorted order.

Complexity

- In this problem, every time we are dividing one of the array in \sqrt{n} size of blocks
- So every step we are reducing the problem size by a factor of \sqrt{n}
- Actually there are two tasks to be carried out here:
 - one is to divide one of the array in \sqrt{n} blocks,
 - and second is to **rank the boundary** elements of these blocks in the another array
- We have already seen that ranking of boundary elements of \sqrt{n} blocks takes O(1) time

Complexity

- Hence, the recursion satisfies the recurrences either:

$$T(n) = T(\sqrt{n}) + O(1)$$

or

$$T(m) = T(\sqrt{m}) + O(1)$$

- The processor requirement is $O(m + n)$.
- The total work done is $O(m + n) \log\log m$.

Solving the recurrence relation

$$\begin{aligned} T(n) &= T(\sqrt{n}) + O(1) \\ \underbrace{T(2^m)}_{m = \log n} &= T(2^{\frac{m}{2}}) + O(1) \\ &= T(2^{\frac{m}{2^k}}) + O(1) \\ \frac{m}{2^k} &= 1 \\ m &= 2^k \\ k &= \log m \\ &= \underline{\underline{\log \log n}} \end{aligned}$$

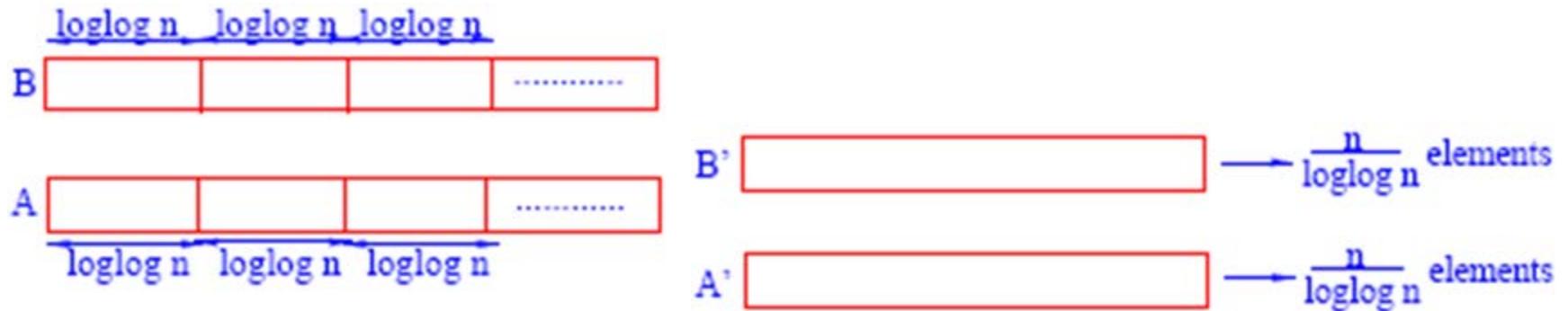
Merging of two sorted sequences

- The algorithm that we have just seen is having following complexity
 - $O(\log \log n)$ time and
 - $O(m+n)*O(\log \log m)$ work
- Next we attempt to design a work optimal algorithm for merging

An optimal Merging Algorithm

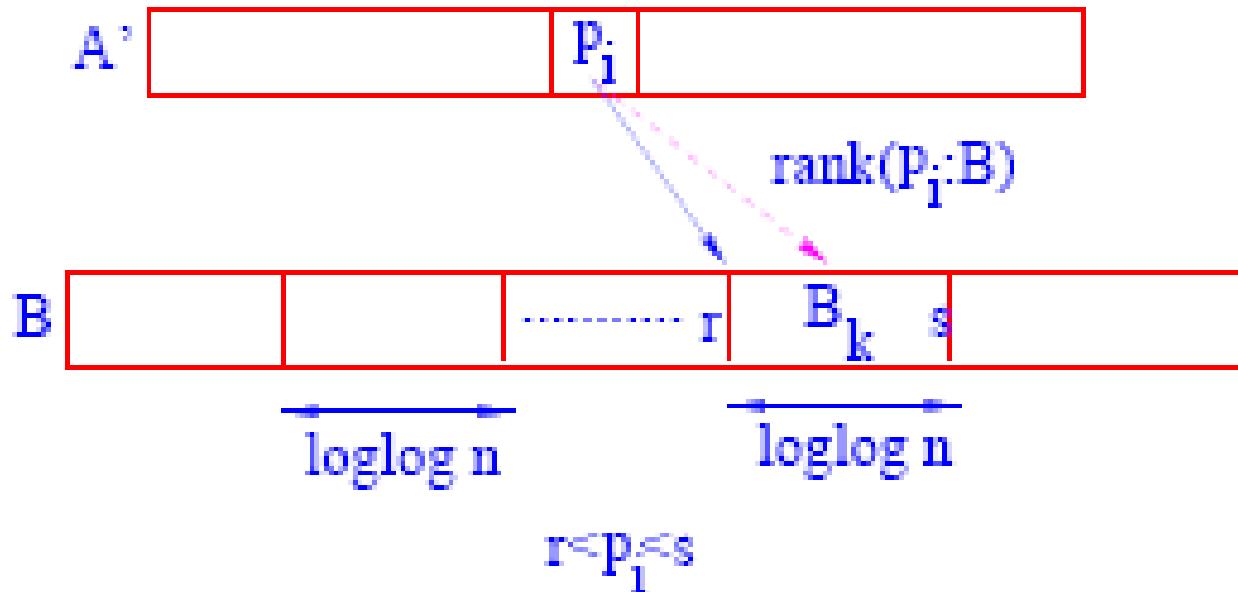
- To make the algorithm optimal, we need to reduce the work to $O(m + n)$.
- We use a different sampling strategy and use the fast algorithm that we have designed.
 - For simplicity, we assume that each array has n elements.
- We divide arrays A and B into blocks of size $\log \log n$
- We choose the last element from each block as our sample and form two arrays A' and B'
- Hence each of A' and B' has $\frac{n}{\log \log n}$ elements

Taking the Samples



- Now we compute $\text{rank}(A' : B')$ and $\text{rank}(B' : A')$ using the algorithm we have designed
- $\text{rank}(A' : B')$
 - Here elements in A' and B' : $m' = O(n/\log \log n)$
 - Processors used $p' = O(n/\log \log n)$
- So time complexity: $O(\log \log m') = O(\log \log n)$
- Work : $O[p' \times (O(\log \log n))]$
 - $= O[(n/\log \log n) \times (\log \log n)] = O(n)$

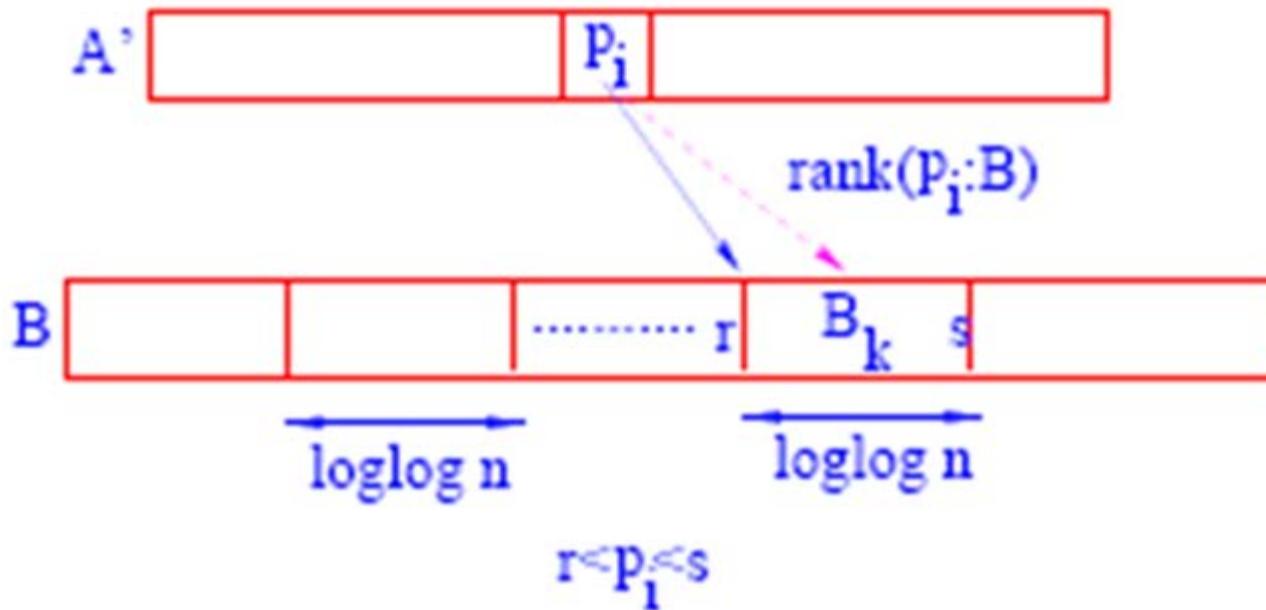
Ranking the Elements



- We now compute $\text{rank}(A' : B)$ as follows
- Suppose the elements in A' are:

$$p_1, p_2, \dots, p_n / \log\log n$$

Ranking the Elements

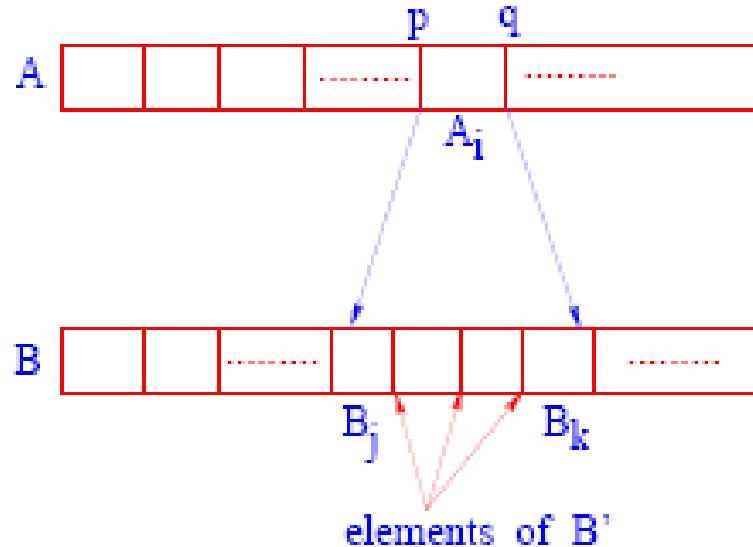


- Consider $p_i \in A'$
- If $\text{rank}(p_i : B')$ is the first element in block B_k
 - Then $\text{rank}(p_i, B)$ must be some element in block B_k
- We can do binary search using one processor to locate $\text{rank}(p_i, B)$

Ranking the Elements

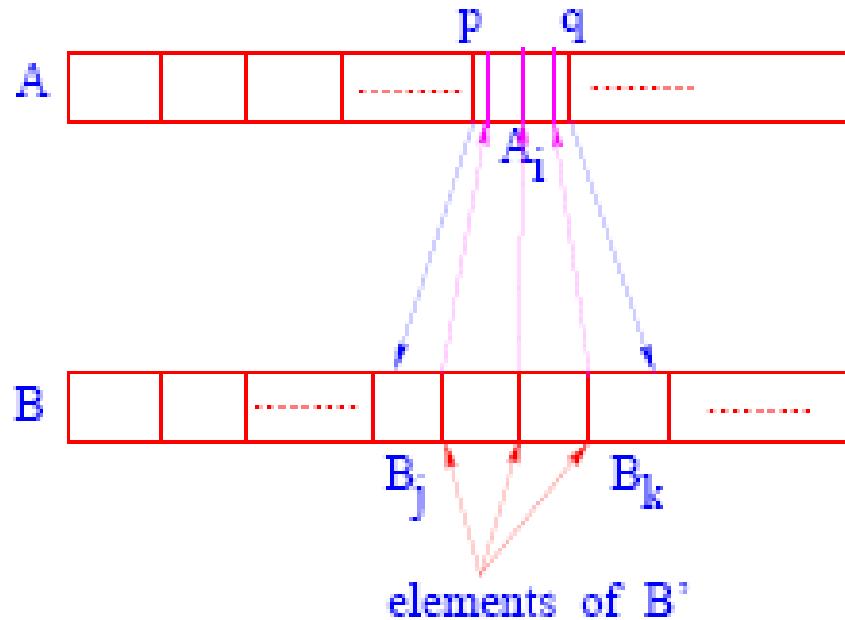
- We allocate one processor for p_i
 - The processor does a binary search in B_k
- Since there are $O(\log \log n)$ elements in B_k , this search takes $O(\log \log \log n)$ time
- The search for all the elements in A' can be done in parallel and requires $O\left(\frac{n}{\log \log n}\right)$ processors
- So work : $O[\log \log \log n \times (n/\log \log n)] = O(n)$
- We can compute $\text{rank}(B' : A)$ in a similar way

Recursion Again



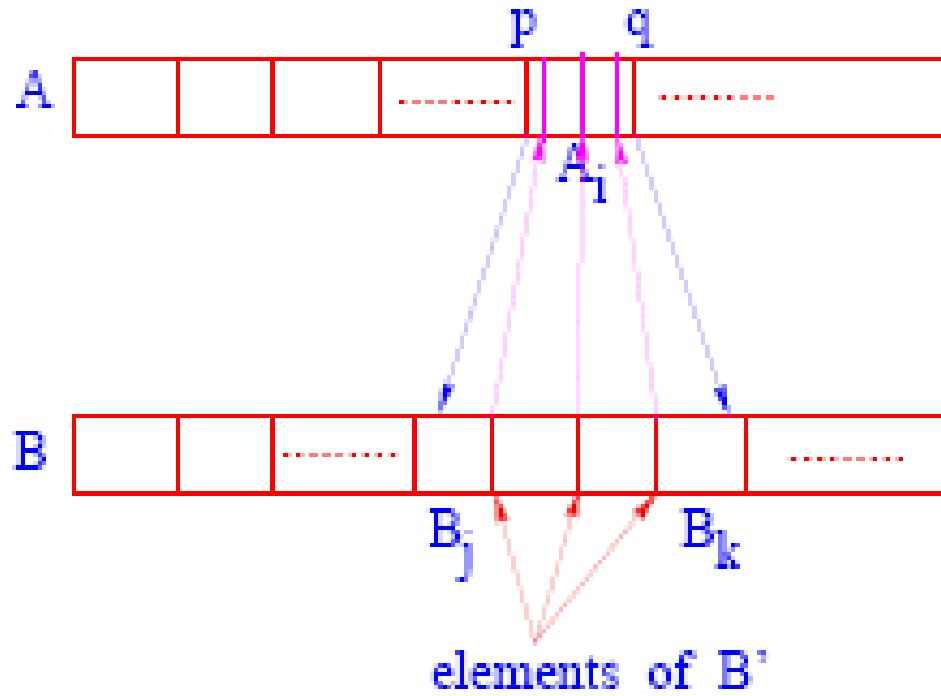
- Consider A_i , a $\log \log n$ block in A
- We know $\text{rank}(p : B)$ and $\text{rank}(q : B)$ for the two boundary elements p and q of A_i
- Now we can call our algorithm recursively with A_i and all the elements in B in between $\text{rank}(p : B)$ and $\text{rank}(q : B)$

Recursion Again



- Now, the problem is, there may be too many elements in between $\text{rank}(p : B)$ and $\text{rank}(q : B)$.
 - Actually there may be many $\log \log n$ size blocks in between $\text{rank}(p : B)$ and $\text{rank}(q : B)$

Recursion Again



- The boundaries of all these blocks must be ranked in A_i
- Hence we get pairs of blocks, one $\log \log n$ block from B and a smaller block from A_i

Solving the Sub-problems

- Now each of the two blocks participating in a sub-problem has size at most $\log\log n$
- And there are $O(\frac{n}{\log\log n})$ such pairs
- We assign one processor to each pair.
- Each processor merges the elements in the pair sequentially in $O(\log\log n)$ time
- All the merging can be done in parallel since we have $\frac{n}{\log\log n}$ processors

Complexity

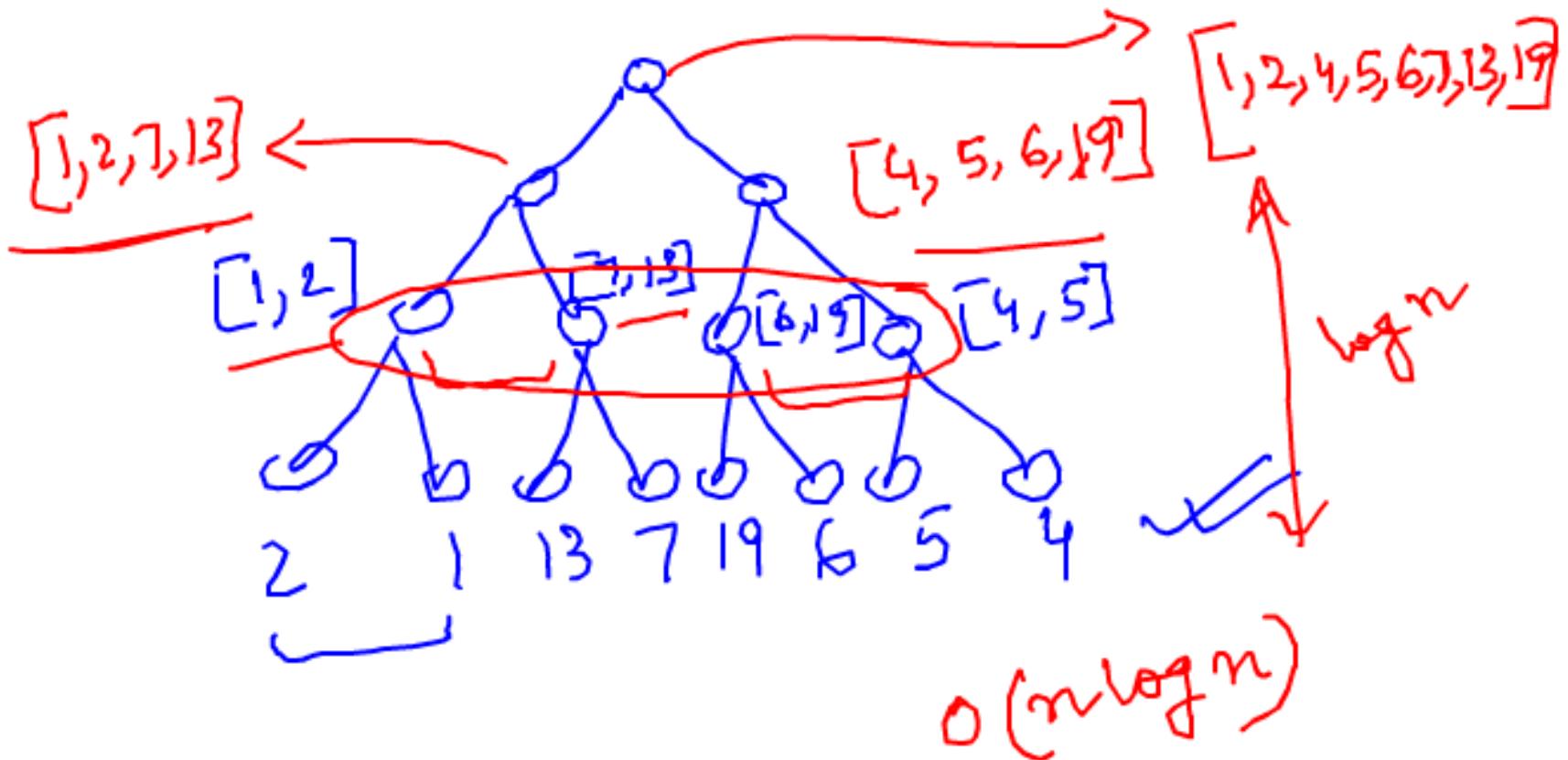
- Computing $\text{rank}(A' : B')$ and $\text{rank}(B' : A')$ take $O(\log\log n)$ time and $O(n)$ work
- Computing $\text{rank}(A' : B)$ and $\text{rank}(B' : A)$ take $O(\log\log\log n)$ time and $O(n)$ work
- The final merging also takes the same time and work
- Hence, we can merge two sorted arrays of length n each in $O(n)$ work and $O(\log\log n)$ time on the CREW PRAM

End

An efficient sorting algorithm

- We can use this merging algorithm to design an efficient sorting algorithm.
- Recall the sequential merge sort algorithm.
- Given an unsorted array, we go on dividing the array into two parts recursively until there is one element in each leaf.
- We then merge the sorted arrays pair-wise up the tree.
- At the end we get the sorted array at the root.

Sequential Merge Sort



- Sequential merging of two arrays takes $O(n)$ time

An efficient sorting algorithm

- We can use the idea of sequential merge sort to sort the arrays in parallel
- Here also we can keep on dividing the array to be sorted in equal halves till we are left with two elements
- Once we have left with arrays of 2 elements, we can start merging
- In case of parallel algorithm, we can use parallel merging technique which we have already discussed

An efficient sorting algorithm

- We can use the optimal merging algorithm to merge the sorted arrays at each level [for parallel merging]
- There are n elements in each level of this binary tree distributed among several arrays depending upon the level
- Hence we need $O(\log \log n)$ time and $O(n)$ work for all the pairwise mergings at each node at a particular level

An efficient sorting algorithm

- The binary tree has a depth of $O(\log n)$
- Hence we can sort n elements in
 - $O(\log n \log \log n)$ time and
 - $O(n \log n)$ work

Better sorting algorithms?

- This sorting algorithm is work-optimal since the sequential lower bound for sorting is $\Omega(n \log n)$.
- However, it is not time optimal.
- Cole's pipelined merge sort algorithm is an optimal $O(\log n)$ time and $O(n \log n)$ work sorting algorithm on the EREW PRAM.
- Cole achieved this by performing merging at multiple level

End