PARALLEL AND DISTRIBUTED ALGORITHMS

17/7/18

Seq

- Flow chart
- Turing Machines
- RAM
  - Assembly
  - PseudoCodes
  - Code in high level

Parallel

- Flowchart/graphs
- PRAM
  - Generalize RAM to Parallel RAM
  - (abstract "a bit")
- Network Model
  - (Individual processes connected by interconnected network)

General Purpose GPUs are designed on PRAMs

(strong dependence on interconnection network)

Specify, design, analysing algos

- Some common techniques
  - Balanced Binary Trees
  - Pointer Jumping
  - Divide & conquer
  - Pipelining
  - partitioning
  - ...

[3-(n-1)]

Tree Algos
Graph Algos
Searching
Sorting
String

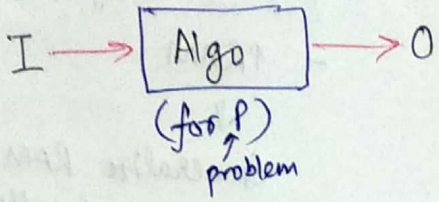
n- Limitations of PRAMs

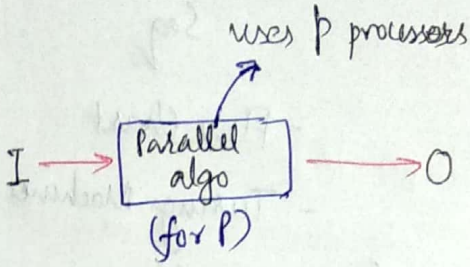
☆ Is a problem parallelizable p-complete.

23/7/18

§1 Parallel Programming Models

Performance measure





$|I| = n$

$T(n) \rightarrow$ running time

$S(n) \rightarrow$ space reqmt

$T_p(n) \rightarrow$ parallel running time

$S_p(n) \rightarrow$ parallel space reqmt.

Ideal speedup $= p$

$$E_p(n) \triangleq \frac{T_p(n)}{p\,T_1(n)} \rightarrow \frac{T_1(n)}{p\,T_p(n)}$$

effectiveness of resource usage

Efficiency

$T_1(n) \rightarrow$ not necessarily the best running time of a sequential algo to solve P.

$T^*(n) \rightarrow$ "best" sequential running time.

- optimal
- best known

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

Speedup (not space)

"effectiveness" of parallelization

ideally, we want $S_p(n) = \theta(p)$

$T_\infty(n) \rightarrow$ the best possible parallel running time

↑ no matter how many processors you have

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

$$T_1(n) \geq T^*(n)$$

* $E_p(n) \leq \frac{T_1(n)}{p\,T_\infty(n)}$

*Efficiency drops if $p \geq \frac{T_1(n)}{T_\infty(n)}$

Optimal no. of processors $= \frac{T_1(n)}{T_\infty(n)}$

Seq: RAM model is widely accepted
- Simple
- practical

Par: No accepted general model for parallel architecture.
but we need simple models, independent of but portable to different parallel architectures.
Conflicting ⇒ no "unanimous" model

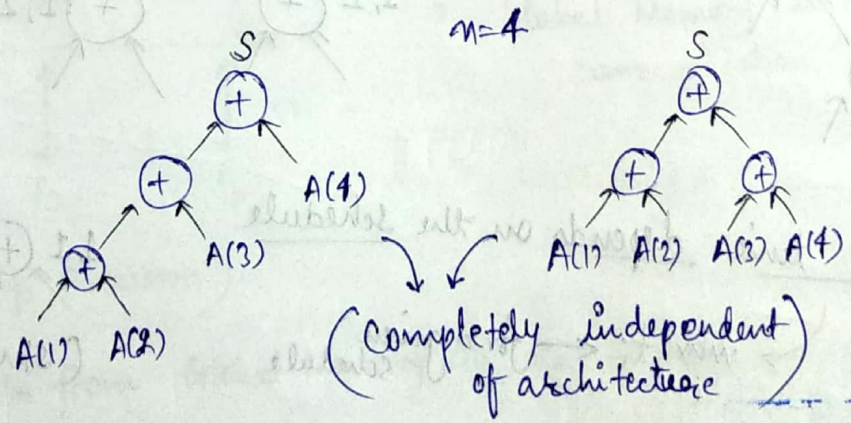
Existing models:-
- DAG
- Shared-memory model
- network model

Directed Acyclic Graphs

- Nodes with in-degree 0 → input
- Nodes with out-degree 0 → output
- Each node is a unit-time computation
- No loops are permitted
  (Soln: Loop unrolling)

$A = (A(1), A(2), \ldots, A(n))$ → n-array elements
(array index starts with 1 here)

To compute $S = A(1) + A(2) + \ldots + A(n)$



ANUBHAV JAIN NOTES

Scheduling a DAG to p processors
- Each op → 1 unit time
- when op is done, the result is available immediately.

v_1

v_2

:

v_K

(K nodes in the DAG)

$v_i \rightarrow (t_i, j)$

t_i: schedule this op at time t_i

j: on processor j

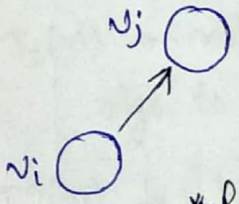
Constraints of a schedule

(1) $v_i \rightarrow (t_i, j)$

$v_{i'} \rightarrow (t_{i'}, k)$

$t_i = t_{i'} \Rightarrow j \neq k$ ⟶ One op. at a time for each processor.

(2) precedence

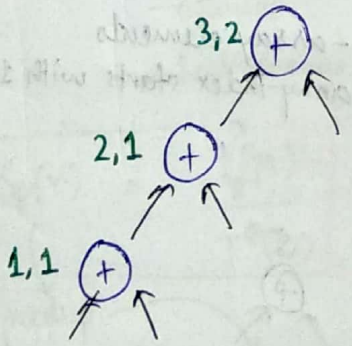


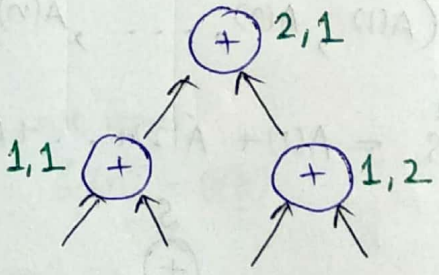
$t_i < t_j$

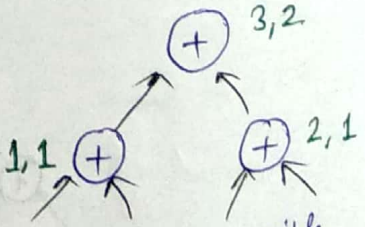
Assumption

All inputs are available at time zero(0)

* Possible Schedules:-







(stupid, but possible schedule)

Running time depends on the schedule

$\max_i t_i$ ← for a given schedule

$$T_p(n) = \min \left[\max_i t_i \right]$$

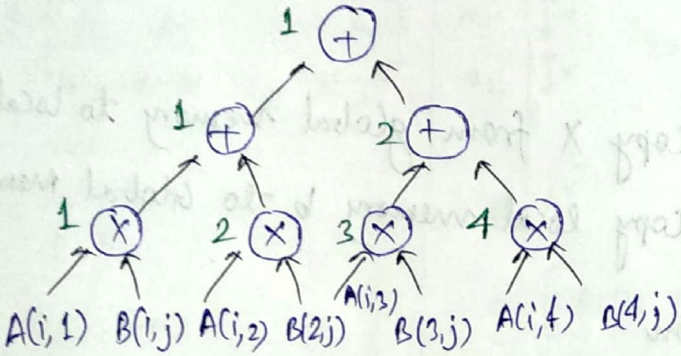
over all possible schedules

$A, B \rightarrow n \times n$ matrices

$C = AB$

$$C(i,j) = \sum_{k=1}^{n} A(i,k)\, B(k,j)$$

DAG for computing $C(i,j)$ $(n=4)$



1 cell → n^4 processors, $\log n$ time

n^2 values to compute

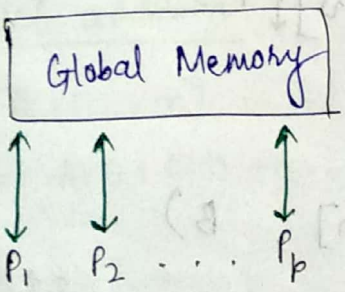
⇓

n^3 processors parallely can compute in $\log n$ time.

If $p = n^3$, then

$T_p(n) = \theta(\log n)$

Shared-Memory model (SMM)



(p Processors)

* No communication network among the processors.
* Global Memory is only way of communication.

- Fetch data from Global Memory to local Memory
- Do computations
- Write back the results to Global Memory

---

* No communication network (Pi cannot broadcast to other processors)
* Only way processors can communicate is via the Global Memory.

SMM

- Synchronous – same clock for all the processors
- Asynchronous – Different clocks for different processors.
  - (Synchronization overhead)

- **Synchronous shared-memory model**
  - PRAM
    - parallel
    - random-access

global read (X, a) — copy X from global memory to local memory a
global write (b, Y) — copy local memory b to Global Memory Y.

Matrix-vector multiplication

$w = A v$

- w: To compute
- A: $n \times n$
- v: n-dim col vector

p processors

$n = rp$

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix} \begin{matrix} \updownarrow r \text{ rows} \\ \updownarrow r \text{ rows} \\ \vdots \\ \updownarrow r \text{ rows} \end{matrix}$$

$$Av = \begin{bmatrix} A_1 v \\ A_2 v \\ \vdots \\ A_p v \end{bmatrix} \begin{matrix} \updownarrow r \\ \updownarrow r \\ \vdots \\ \updownarrow r \end{matrix}$$

Code for Processor $i \in \{1, 2, \ldots p\}$

global read (A[(i-1)r + 1, ir][1 ... n] , B)
global read (v, x)
Compute $y = Bx$
global write (y , W[(i-1)r + 1 , ir])

may run asynchronously.

24/7/18

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix} \bar{x} = \begin{bmatrix} A_1\bar{x} \\ A_2\bar{x} \\ \vdots \\ A_p\bar{x} \end{bmatrix}$$

Exclusive write — concurrent read

$$Ax = \begin{bmatrix} A_1 & A_2 & \cdots & A_p \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}$$

$A_1, A_2, \ldots, A_p$: r cols each; $x_1, x_2, \ldots, x_p$: r rows each

$r = \frac{n}{p}$

$A_i \rightarrow n \times r$

$$= [A_1x_1 + A_2x_2 + \ldots + A_px_p]$$

Each $z_i = A_i x_i$ is an n-dim vector.

Until all processors compute z_i, addition cannot start

⇒ Synchronous m/c is preferred.

(until programmer implements its own synchronization primitives)

↓

on asynchronous m/c.

Parallel addition

$A[1 \ldots n]$ $\quad n = 2^t$.

$S = A(1) + A(2) + \ldots + A(n)$

```
for i = 1,2,...,n                    → done by processor i
    global read (A(i), x)
    global write (x, B(i))

for k = 1,2,...,t                    → T_n(n) = θ(log_2 n)
    for i = 1,2,...,n/2^k
        global read (B(2i-1), x)
        global read (B(2i), y)
        set z = y+x
        global write (z, B(i))
if (i==1) output B(1)
```

for $i = 1, \ldots, n$
    Statement 1 } all n busy

for $i = 1, \ldots, \frac{n}{2}$
    Statement 2 } first $n/2$ busy, last $n/2$ waiting

for $i = \frac{n}{2}+1, \ldots, n$
    Statement 3. } last $n/2$ busy, first $n/2$ waiting.

Synchronization is not just existing a common clock, rather processors which are not busy need to wait for other processors to finish

Cleaned-up code

```
for i = 1, . . , n
        set   B(i) = A(i)
for K = 1, . . , t
        for i = 1, . . . , n/2^K
                set   B(i) = B(2i-1) + B(2i)
```

Read — exclusive / concurrent

Write — exclusive / concurrent

Types of PRAM

Increase in power ↓

- EREW
- CREW
- CRCW — common (write succeeds when all processors attempt to write the same value) → [on failing, value at location remains same]
  - priority (the highest priority processor succeeds)
  - arbitrary
- ~~ERCW~~

Sum CREW PRAM

↳ sum of concurrent write values is stored.

↳ can solve the array sum problem in O(1) time
(all processors write at same location concurrently)

Network Model

30/7/18

$P_1, P_2, \ldots, P_p$ → processors.

- No global memory
- The processors are connected by an interconnected network.
- Each link in the ntk. is bidirectional.

P_K : send (X, i) [non-blocking]

P_K : receive (Y, j) [blocking]

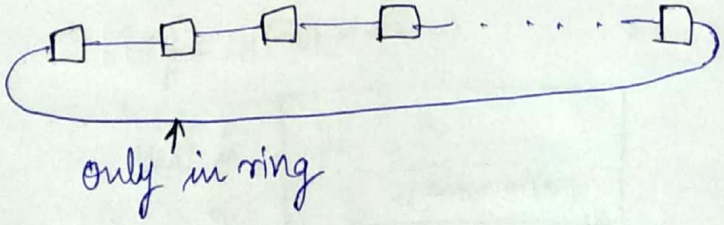
(i, k) or (j, k) need not be adjacent in the ntk.

If not, routing is needed

Network Parameters

- maximum degree (d)
- diameter (maximum-interprocessor distance) (D)

- Line / Ring

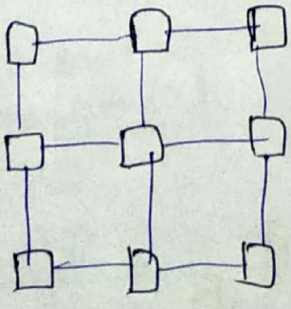


d = 2

D = p/2 (ring)

p-1 (line)

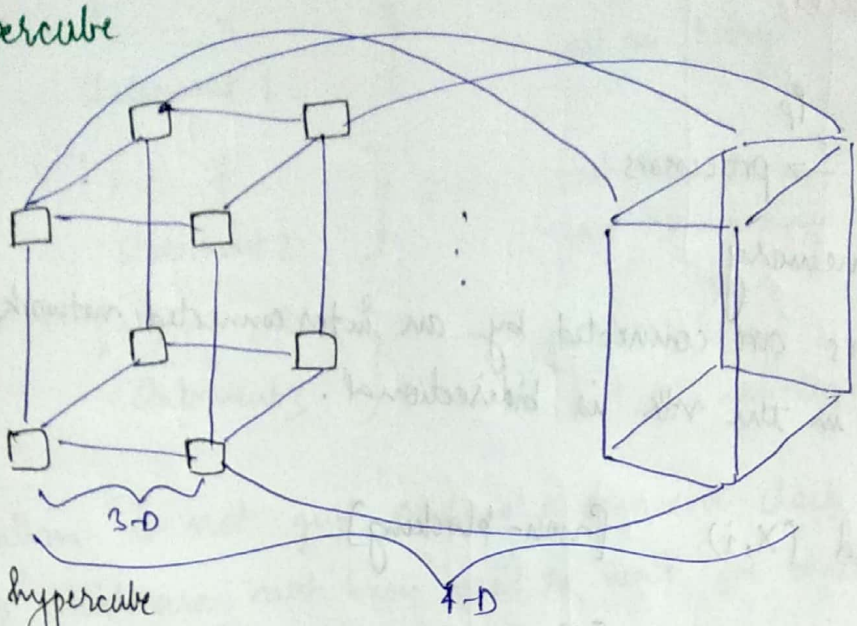
- Mesh



d = 4

$D = 2(\sqrt{p} - 1)$

- Hypercube



k - dim hypercube

$p = 2^k$

$d = k = \log_2 p$

$D = k = \log_2 p$

processors are numbered

$0, 1, 2, \ldots, p-1 = 2^k - 1$

that is by k-bit strings

$i \rightarrow (i_{k-1}\, i_{k-2} \ldots i_1 i_0)_2$

$i(l) \rightarrow (i_{k-1}\, i_{k-2} \ldots i_{l+1}\, i_l\, i_{l-1} \ldots i_1 i_0)_2$

Processor i is connected to Processors $i(l)$ for $l = 0, 1, 2, \ldots, k-1$

For each hop, only one bit is changed ⇒ Diameter = k.

- Matrix vector Multiplication on a ring

$A\,\overline{x}$

$(n \times n)$ $(n \times 1)$

$n = np$

$$[A_1\, A_2 \ldots A_p] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}$$

$= A_1 x_1 + A_2 x_2 + \ldots + A_p x_p$

Code for processor i

– Given A_i and x_i

1. Compute $z = A_i x_i$ ($n \times r$ times $r \times 1$)
2. If $i = 1$, set $y = 0$
   else receive$(y, i-1)$
3. Compute $y := y + z$
4. Send $(y, right)$
5. If $(i = 1)$ receive $(w, left)$ ← Answer

Parallel running time $= n * r * 1 = nr = \frac{n^2}{p}$

Sequential

Total running time $= np$

Communication overhead $= p(\sigma + \tau n)$

σ: setup time; τ: rate; τn: transmission time for an n-dim vector

$$T_p(n) = \frac{n^2}{p} + np + p(\sigma + \tau n)$$

$$= \frac{n^2}{p} + p(n + \sigma + \tau n)$$

minimized if

$$\frac{n^2}{p} = p(n + \sigma + \tau n)$$

that is, $$p = \frac{n}{\sqrt{n + \sigma + \tau n}}$$

Systolic matrix multiplication on a mesh.

$C = A B$ ($n \times n$ $n \times n$)

$A = (a_{ij})$

$B = (b_{ij})$

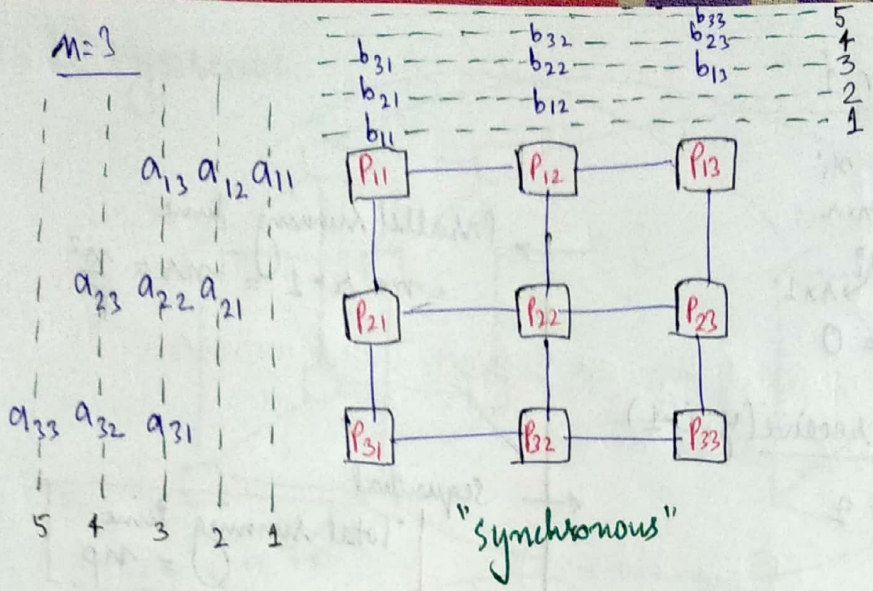
$$c_{ij} = \sum_{j=1}^{n} a_{ik} b_{kj}$$

P_{ij} computes c_{ij}

requires i^{th} row of A and j^{th} col of B.

We have $n \times n$ mesh

$\Rightarrow p = n^2$



"synchronous"

- P_{ij} gets a_{ik} from left and b_{kj} from top at time $(i+j+k-2)$
- Computes $a_{ik} b_{kj}$
- Adds to local sum.

P_{nn} finishes last at time $n+n+n-2 = 3n-2$

Communication propotional to $n-1$

Total running time = $O(n)$

ⓜ Parallel sum on a HC

$A = (a_0, a_1, \ldots, a_{n-1})$

$S = a_0 + a_1 + \ldots + a_{n-1}$

$n = 2^k$

k-dim hypercube

- Give a_i to processor i
- for $d = k-1, k-2, \ldots, 2, 1, 0$ (Sequential)
  for $i = 0, 1, \ldots, 2^d - 1$ (parallel, $O(1)$ time)
  - P_i receives (blocking) $a_{i(d)}$ from $P_{i(d)}$ ← will execute second
    and computes $a_i + a_{i(d)}$
  - $P_{i(d)}$ sends $a_{i(d)}$ to P_i ← will execute first

parallel in diff processors ⇒ can be written in any order

$n = 8$

$k = 3$

$l = k - 1 = 2$ → along dim 2

P_0 computes $a_0 + a_4$

P_1 computes $a_1 + a_5$

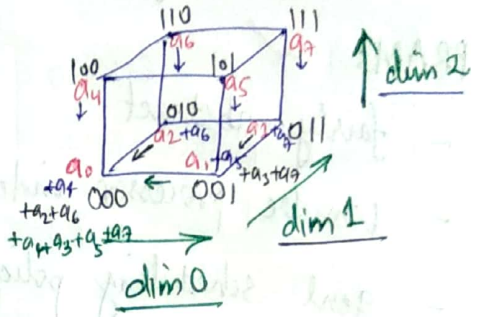
P_2 computes $a_2 + a_6$

P_3 computes $a_3 + a_7$

$l = 1$

P_0 computes $a_0 + a_2 + a_4 + a_6$

P_1 computes $a_1 + a_3 + a_5 + a_7$



$l = 0$

P_0 computes $(a_0 + a_2 + a_4 + a_6) + (a_1 + a_3 + a_5 + a_7) = S$

Running Time $= O(k) = O(\log n)$

• Parallel matrix multiplication in a HC

$C = A \; B$
$n \times n \quad n \times n$

$P_{i,j,k}$

$i, j, k \in \{0, 1, 2, \ldots, n-1\}$

$n = 2^q$

$p = n^3 = 2^{3q}$

⇒ **3q dimensional hypercube**

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \, b_{kj}$$

Give a_{ik} and b_{kj} to $P_{i,j,k}$

$P_{i,j,k}$ computes $a_{ik} \, b_{kj}$

$P_{i,j,0} \; P_{i,j,1} \ldots P_{i,j,n-1}$ compute $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \, b_{kj}$ in $O(\log n)$ time

different c_{ij} values are added in disjoint sub-hypercubes

Running Time $= O(\log n)$

31/7/18

DAGs

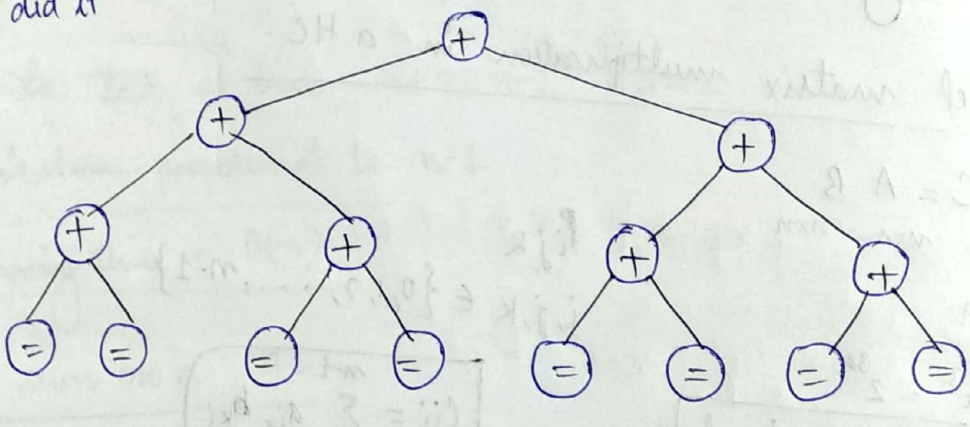
- complicated (no loops supported)
- scheduling not specified

Network

- algo is very much topology ~~specific~~ specific.

PRAMs

- fairly abstract.
- can be processor-independent
- genl (general) scheduling policy.
- widely accepted
- JáJá did it



Q → problem of size n

$P(n)$ = # of processors used $\quad (n)$

$T(n)$ = parallel running time $\quad O(\log n)$

$C(n)$ = cost of this algorithm

$= P(n) \cdot T(n) \quad O(n \log n)$

$W(n)$ = work = # of operations

$C(n) \geq W(n)$

Cost = $8 \times 4 = 32$

Work = $8+4+2+1 = 15$

$$W(n) = n + \sum_{i=1}^{\log_2 n} \frac{n}{2^i} \leq n + n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n = O(n)$$

Work-Time (WT) presentation

	Work
Parallel Step 1	$W_1(n)$
Parallel Step 2	$W_2(n)$
⋮	
Parallel step k	$W_k(n)$

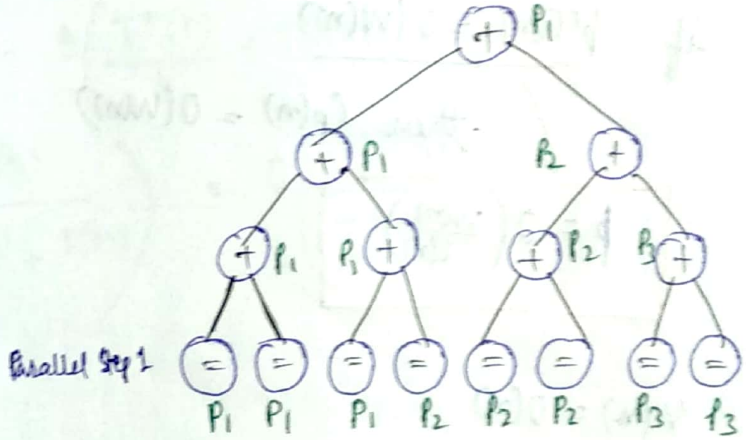
```
for 1 ≤ i ≤ u  pardo {
        block
}
```

(for: lower bound; u: upper bound; pardo: parallel do)

```
for 1 ≤ i ≤ n  pardo
    set B(i) := A(i)
for l = 1, 2, ..., log2 n  do {
    for 1 ≤ i ≤ n/2^l  pardo
        set B(i) := B(2i-1) + B(2i)
}
set S := B(1)
```

On 8 processors,
running time = 4

On 2 processors,
running time = 3+2+1+1
= 7.



Brent's scheduling principle
(WT scheduling principle)

Let A be an algorithm specified in the WT presentation level.

W(n) = the work done by A

T(n) = the parallel running time of A.

p is the available no. of processors.

A can be scheduled among these processors to obtain

$$T_p(n) = O\left(\frac{W(n)}{p} + T(n)\right)$$

$W(n) = W_1(n) + W_2(n) + \cdots + W_k(n)$

$k = T(n)$

$\forall i,\ 1 \le i \le k$,
we can schedule the work $W_i(n)$ on p processors to run in $\left\lceil \frac{W_i(n)}{p} \right\rceil$ time

ANUBHAV JAIN NOTES

$$T_p(n) = \sum_{i=1}^{k} \left\lceil \frac{W_i(n)}{p} \right\rceil \leq \sum_{i=1}^{k} \left(\frac{W_i(n)}{p} + 1 \right)$$

$$T_p(n) = \left[\frac{1}{p} \sum_{i=1}^{k} W_i(n) \right] + k$$

$$\boxed{T_p(n) = \frac{W(n)}{p} + T(n)}$$

Relation between cost and work

$$C_p(n) = p\, T_p(n)$$

$$= O(W(n) + p\, T(n))$$

if $p^* T(n) = O(W(n))$

↓ then $C_p(n) = O(W(n))$

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

$W(n) = O(n)$

$T(n) = O(\log n)$

$C_p(n) = O(W(n))$

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

6/8/18

Notion of Optimality

Sequential : merge / heap sort optimal

Q → a computational problem

$T^*(n)$ = the optimal sequential complexity to solve Q

A → parallel algo for Q

W(n) → work done by A

T(n) → parallel running time of A

A is called optimal (or work-optimal)

if $W(n) = \theta(T^*(n))$ → optimal algo may be parallel or sequential. not a good notion.

(in a "weak" sense)

In particular, a sequential algo with running time $T^*(n)$ is also optimal in this notion.

$A \rightarrow$ optimal

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

$T(n)$

p processors

$$T_p(n) = O\left(\frac{W(n)}{p} + T(n)\right) = O\left(\frac{T^*(n)}{p} + T(n)\right)$$

Speedup $S_p(n) = \frac{T^*(n)}{T_p(n)} = O\left(\frac{T^*(n)}{\frac{T^*(n)}{p} + T(n)}\right) = O\left(\frac{p\,T^*(n)}{T^*(n) + p\,T(n)}\right)$

If $p = \theta\left(\frac{T^*(n)}{T(n)}\right)$,

then $S_p(n) = \theta(p)$

A is called WT-optimal (optimal in the "strong" sense)

if

(i) A is work-optimal

(ii) For any work-optimal algo B to solve Q, we have

$T(n) = O(T'(n))$ ← parallel running time of B

if a lower bound on the parallel running time can be proved, and A achieves that running time, A is WT-optimal.

Example: Parallel sum of $A = (a_1, a_2, \ldots, a_n)$

See last class.

$W(n) = \theta(n)$

$T(n) = \theta(\log n)$

- Cannot be solved in less than $O(\log n)$ time in the worst case on a CREW PRAM.
- Can be solved in $O(1)$ time in a sum CRCW PRAM (given hardware support).
- notion of optimality depends on the PRAM model

6/8/18

§2 Parallelization Techniques

Technique 1

Balanced Binary trees (BBT)

Examples: Parallel Sum

New example:

$A = (a_1, a_2, a_3, \ldots, a_n)$

for $1 \le i \le n$, define

$S_i = a_1 + a_2 + \ldots + a_i$

To compute <u>all</u> the prefix sums $S_1, S_2, \ldots, S_n$

An <u>optimal</u> prefix-sum algorithm

$T^*(n) = \Theta(n)$: $\{ S_i = a_i + S_{i-1} \}$

T^*: Sequential [obvious]

<u>A recursive algo.</u>

if $n=1$, set $S_1 = a_1$ and return

for $1 \le i \le n/2$ <u>pardo</u>

set $b_i = a_{2i-1} + a_{2i}$

recursively compute the prefix sums of $b_1, b_2, \ldots, b_{n/2}$ in $c_1, c_2, \ldots, c_{n/2}$

for $1 \le i \le n$ <u>pardo</u> {

(*)

}

(*) if (i is even), set $S_i = c_{i/2}$

else if ($i = 1$), set $S_1 = a_1$

else set $S_i = c_{(i-1)/2} + a_i$

<u>PRAM Model</u> : CREW

EREW (with minor modification → make copy of array)

<u>W and T</u>

$T(n) = T(n/2) + \alpha$ (const)

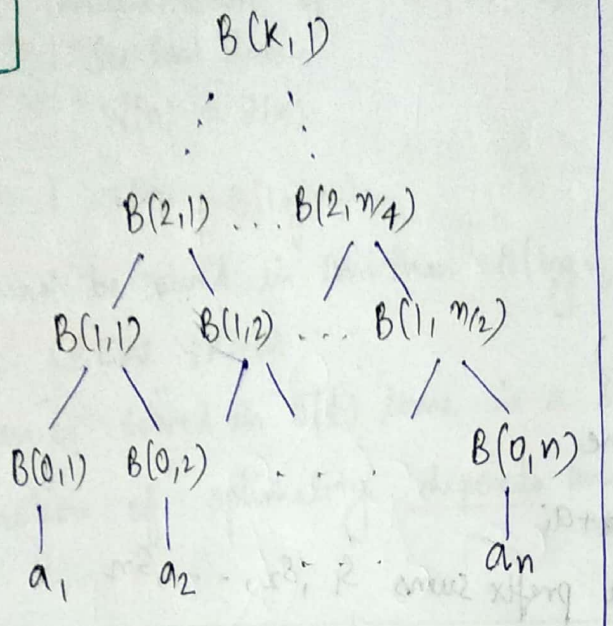
$= O(\log n)$

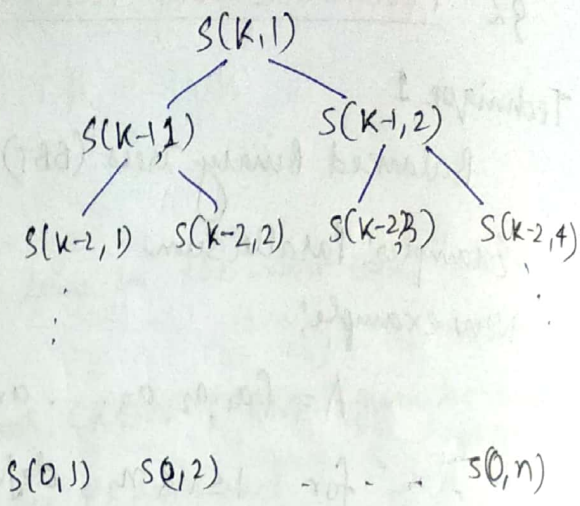
(also strongly optimal)

$W(n) = W(n/2) + \beta n$ (const)

$= O(n)$ optimal

$n = 2^k$





for $1 \le i \le n$ pardo

copy $B(0,i) := a_i$

for $h = 1,2,3,\ldots,k$ do

for $i = 1,2,\ldots,\frac{n}{2^h}$ pardo

Set $B(h,i) = B(h-1, 2i-1) + B(h-1, 2i)$

for $h = k, k-1, \ldots, 1, 0$ do

for $i = 1,2,\ldots \frac{n}{2^h}$ pardo

if (i is even) set $C(h,i) = C(h+1, i/2)$

else if ($i = 1$) set $C(h,1) = B(h, 1)$

else set $C(h,i) = C(h+1, \frac{i-1}{2}) + B(h,i)$

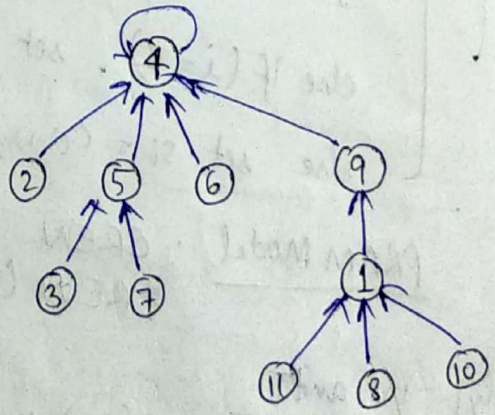
Technique 2

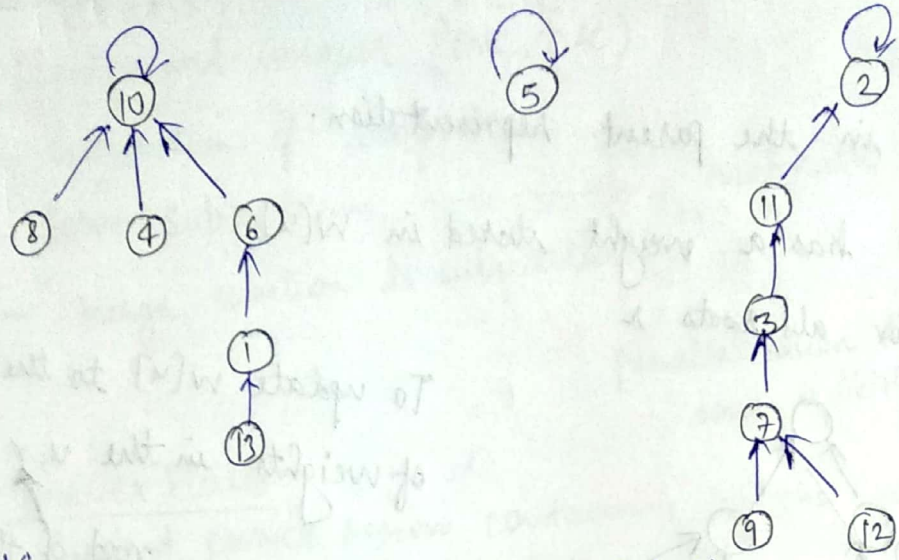
Pointer Jumping (PJ)

Iterative Doubling

Parent Representation

9	4	5	4	4	4	5	1	4	1	1
1	2	3	4	5	6	7	8	9	10	11





Input:-

P	6	2	11	. . .	1
	1	2	3		13

Output:-

S	10	2	2		10
	1	2	3		13

Task: for each $i \in [1, n]$, find the root $S(i)$ of the tree to which i belongs.

```
for 1 ≤ i ≤ n pardo {
    Initialize S(i) = P(i)
    while ( S(i) ≠ S(S(i)) ) {        → equality when S(i) = Root.
        set S(i) = S(S(i))
    }
}
```

PRAM Model : CREW

Running Time :

$O(\log h)$, where h is the height of the tallest tree in the input.

Work Done :

$O(n \log h)$

Not optimal → Worst case $O(n \log n)$ whereas sequential → $O(n)$.

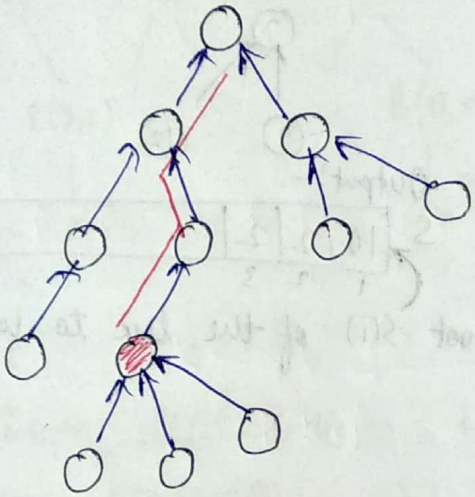
7/8/18

Tree Prefix.

F → a forest in the parent representation.

Each edge (u, p) has a weight stored in w[u]

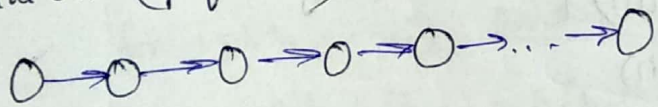
w[r] = 0 for all roots r.



To update w[u] to the sum of weights in the u, r path

r → root of the tree containing u

Special Case (prefix sum)



CREW PRAM.

```
for 1 ≤ i ≤ n pardo {
    set S(i) = P(i)
    while (S(i) ≠ S(S(i))) {
        set w[i] = w[i] + w[S[i]];
        set S(i) = S(S(i));
    }
}
```

Running Time = $O(\log h)$

h = height of the tallest tree in F

Work = $O(n \log h)$

Spl Case:

$h = n$

$T(n) = O(\log n)$

$W(n) = O(n \log n)$

A non-optimal way of computing prefix sums.

(optimal $W(n) = O(n)$)

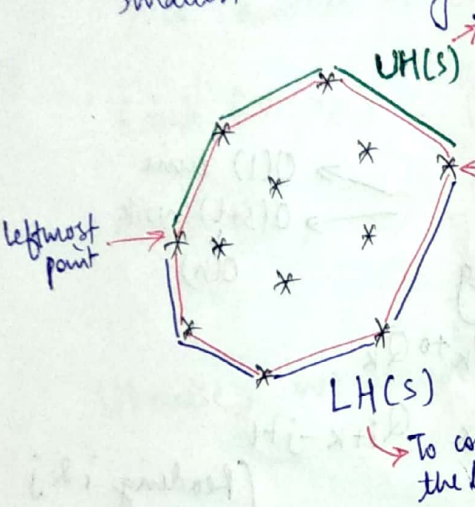
Technique 3.

Divide-and-conquer (DnC / D&C)

- Generation of subproblems. → naturally done in parallel
- Solve subproblems
- Merge solution to subproblems. → parallelization to be done explicitly.

Convex Hulls

Smallest convex region containing all the points.



S → set of points

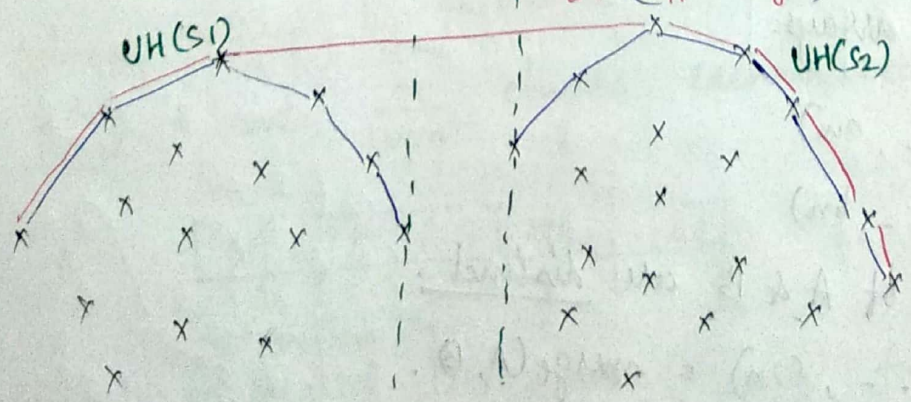
To compute CH(S)

Preparata-Hoey's algorithm

- Sort S $(P_1, P_2, \ldots, P_n)$ — Chap 4 Pipelined Merge Sort: $O(\log n)$ time, $O(n \log n)$ work
- Let $S_1 = (P_1, P_2, \ldots P_{n/2})$, $S_2 = (P_{n/2+1}, \ldots, P_n)$ — sequential $O(1)$ time
- Compute UH(S_1) and UH(S_2) in parallel — $T(n/2)$, $2W(n/2)$
- Merge UH(S_1) and UH(S_2) to UH(S)

$(U_1, U_2, \ldots, U_s)$ $(U_{s+1}, \ldots, U_{s+t})$

UT (Upper Tangent)



* Let us assume all x coordinates are unequal & all y coordinates are unequal.
* $n = 2^k$

ANUBHAV JAIN NOTES

- To compute the UT → The seq. algo returns i & j.

Can be done in $O(\log n)$ sequential time using a binary-search algorithm.

pt. of tangency

incorrect notation → $P_1, P_2, \ldots, P_i, P_{i+1}, \ldots, P_{n/2}$

correct notation → $U_1, U_2, \ldots, U_i, U_{i+1}, \ldots, U_s$

$P_{\frac{n}{2}+1}, \ldots, P_j, P_{j+1}, \ldots, P_n$

$U_{s+1}, \ldots, U_j, U_{j+1}, \ldots, U_{s+t}$

Output is

$P_1, P_2, \ldots, P_i, P_j, P_{j+1}, \ldots, P_n$

$U_1, U_2, \ldots, U_i, U_j, U_{j+1}, \ldots, U_{s+t}$

```
for 1 ≤ k ≤ s+t pardo {
    if i < k ≤ j , do nothing
    else if k ≤ i   copy U_k to Q_k
    else if k ≥ j   copy U_k  Q_{i+k-j+1}
}
```

→ $O(1)$ time
→ $O(s+t)$ work ↓ $O(n)$

(Reading i & j requires concurrent read)

CREW PRAM

$T(n) \leq T(n/2) + \alpha \log n$;

$W(n) \leq 2W(n/2) + \beta n$

$T(n) = O(\log^2 n)$

$W(n) = O(n \log n)$ → optimal (Computing Convex Hull has lower bound $n \log n$)

13/8/18

Technique 4

Merging two sorted arrays.

$A = (a_1, a_2, \ldots, a_n)$

$B = (b_1, b_2, \ldots, b_n)$

The $2n$ elements of A & B are distinct.

$C = (c_1, c_2, \ldots, c_{2n}) = \text{merge}(A, B)$.

Sequential Algo : $O(n)$ time — optimal

Parallel Algo

Goal : $O(n)$ work

$O(\log n)$ time

$X \rightarrow$ a finite set (of numbers)

Rank $(y; X)$ = # of elements of X that are $\leq y$

$Y \rightarrow$ another set $= \{y_1, y_2, \ldots, y_n\}$

Rank $(Y; X) = (r_1, r_2, \ldots, r_n)$ where $r_i = \text{rank}(y_i; X)$.

rank $(b_i; A \cup B)$ = position of b_i in C

$= \text{rank}(b_i; A) + \underbrace{\text{rank}(b_i; B)}_{i}$

Likewise, we require rank $(a_i; A \cup B)$

$= i + \text{rank}(a_i; B)$

It suffices to compute rank (A, B) and rank (B, A)

To compute rank (A, B)

Sol 1: for $1 \leq i \leq n$, pardo

make a binary search of a_i in B

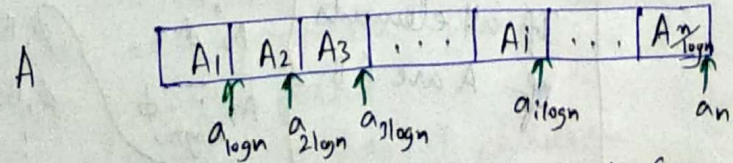
CREW PRAM

$O(\log n)$ time

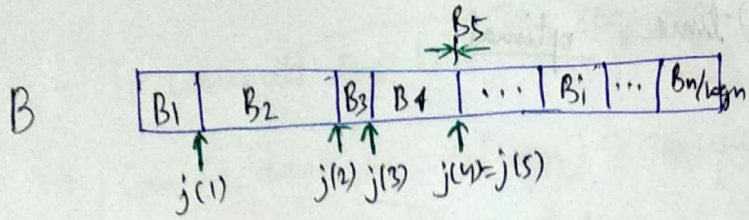
To improve → $O(n \log n)$ work

Optimal merging by partitioning.

Break A into $\frac{n}{\log n}$ chunks each of size $\log n$



Compute in parallel $j(i) = \text{rank}(a_{i\log n}, B)$ for $i = 1, 2, 3, \ldots, \frac{n}{\log n}$



$O(\log n)$ time

$O(n)$ work.

merge (A_i, B_i) in parallel for all i

→ $O(n)$ work

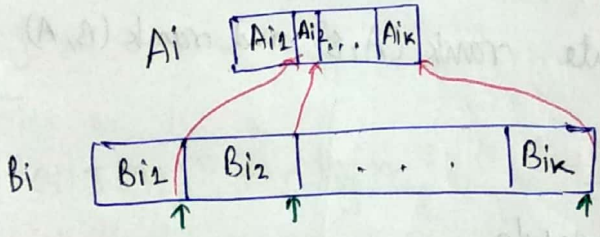
→ may be $\Theta(n)$ time

If $|B_i| \leq \log n$,

go ahead with merge (A_i, B_i)

If $|B_i| > \log n$,

break B_i into $\log n$-sized chunks.



merge (A_i, B_i) reduced to merge (A_{ij}, B_{ij}), $j = 1, 2, \cdots, k$

$$|A_{ij}| \leq |A_i| \leq \log n$$

$$B_{ij} \leq \log n$$

# of subproblems after 2-way partitioning

$$= \sum_{i=1}^{n/\log n} \left\lceil \frac{|B_i|}{\log n} \right\rceil \leq \sum_{i}^{n/\log n} \left(\frac{|B_i|}{\log n} + 1 \right) = \frac{2n}{\log n}$$

Each of these $\frac{2n}{\log n}$ subproblems are on chunks of A & B of sizes $\leq \log n$.

If all elements of A are $\leq$ B

A_1, ϕ
A_2, ϕ
$\vdots$
$A_{n/\log n}, \phi$
ϕ, B

ϕ, B_1
ϕ, B_2
$\vdots$
$\phi, B_{n/\log n}$

merge each pair of chunks
→ parallel across chunks
→ sequential for each chunk

Running Time = $O(\log n + \log n) = O(\log n)$

Work = $O\left(\frac{2n}{\log n} \times \log n\right) = O(n)$

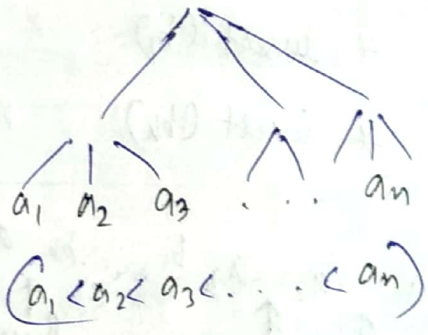
Technique 5

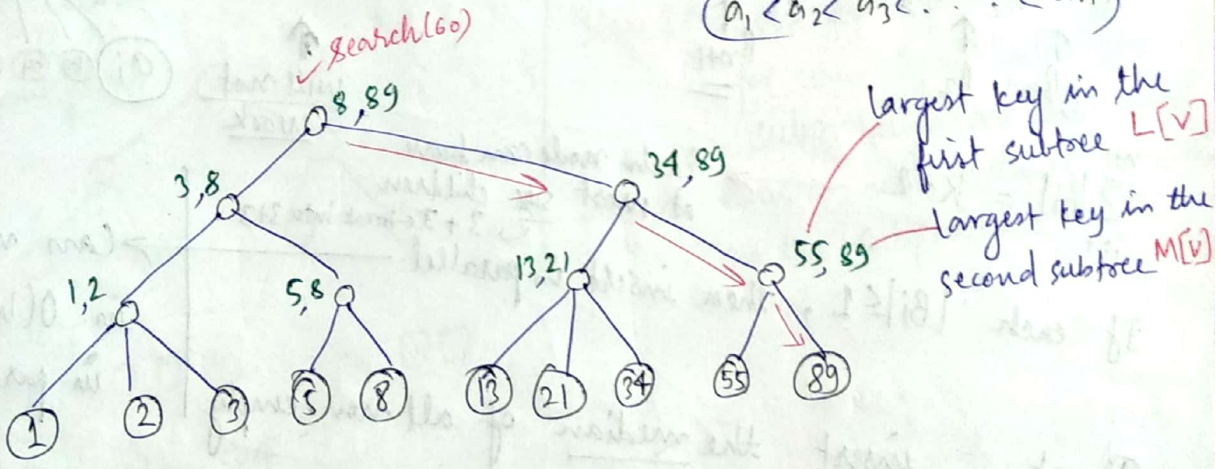
Pipelining

2-3 trees (a kind of B-tree)

A tree storing n keys
↳ they reside in leaves

Each intermediate node has either two or three children.





if $k \le L[v]$, go to the first subtree
else if $k \le M[v]$, go to second subtree
else go to third subtree.

$2^h \le n \le 3^h \Rightarrow h = \Theta(\log n)$

---

insertion in 2-3 trees
- search
- insert at appropriate position
- let p be the parent
- so long as p has 4 children
  - split p in 2 nodes and
  - set $p = p \to$ parent

$O(\log n)$ time

$L[v]$, $M[v]$ can be changed in each affected node.

$T \rightarrow a_1 < a_2 < \ldots < a_n$

To insert $b_1 < b_2 < \ldots < b_k$ in T

$k << n$

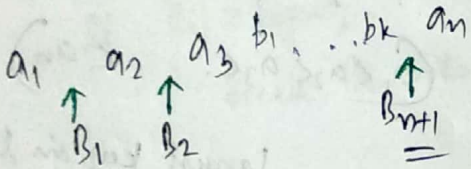
Sequential Complexity = $O(k \log n)$

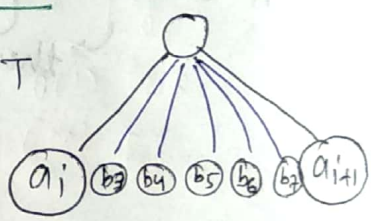
Goal: $O(\log n)$ time
$O(k \log n)$ work.

* insert (b_1)
* insert (b_k)

---

for $2 \le i \le k-1$ pardo
insert b_i in T





will not work

$\sum_{i=1}^{n+1} |B_i| = k-2$ → Each node can have at most ~~six~~ children
3 + 3 ← break into 3+3.

If each $|B_i| \le 1$, then insert in parallel. → Can manage in $O(\log n)$ time in parallel.

If not, insert the median of all non-empty B_i in parallel

splits B_i to subchunks of size $\approx \frac{|B_i|}{2}$

After $O(\log k)$ simple parallel insertions,
all b_j are inserted.

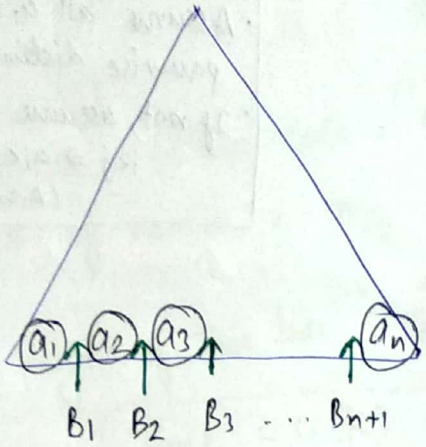
Running time = $O(\log k \log n)$ ← to remove
work done = $O(k \log n)$

14/8/18

$a_1 < a_2 < \ldots < a_n$

$b_1 < b_2 < \ldots < b_k$

$k << n$



$O(\log k)$ parallel insertion (wave) needed to insert all new keys.

These waves can be pipelined.

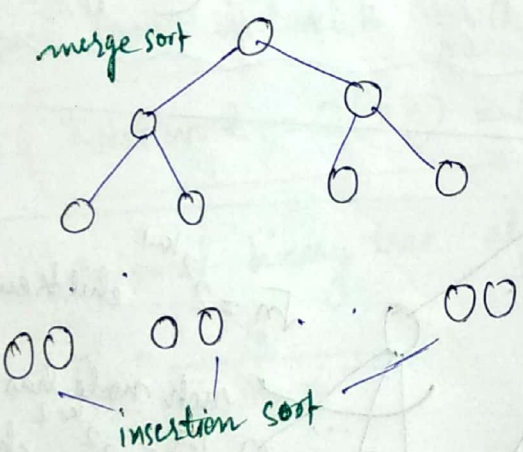
$O(k \log n)$ – work

$O(\log k \log n)$ – time $\Rightarrow O(\log k + \log n)$

$= O(\log n)$ [since $k << n$]

Technique 6

Accelerated cascading



Stage 1: Use an optimal algorithm for some levels until the subproblems are "small" enough

Stage 2: Run a non-optimal algorithm on the subproblems.

$A = (a_1, a_2, \ldots, a_n)$

To compute max(A)

Sequential:

$O(n)$ optimal running time

BBT (balanced binary tree)

↳ CREW PRAM

$O(n)$ work

$O(\log n)$ time

WT optimal

ANUBHAV JAIN NOTES

Common CRCW PRAM

- If all write same thing, then update with that value. else retain previously stored value.

- Assume all a_i's are pairwise distinct
- If not, assume $i<j \Rightarrow a_i < a_j$ (Actual $a_i = a_j$)

for $1 \le i \le n$, pardo $M(i) = 0$

for $1 \le i \le n$ and $1 \le j \le n$ pardo

set $B(i,j) = \begin{cases} 1 & \text{if } a_i \ge a_j \\ 0 & \text{if } a_i < a_j \end{cases}$

for $1 \le i \le n$ pardo

set $M(i) = B(i,1) \wedge B(i,2) \wedge \ldots \wedge B(i,n)$

concurrent write

$M(i) = 1$ only if $a(i) = \max$ elem

$O(1)$ time

(n^2 processors.)

$O(1)$ time

$O(n^2)$ work

$n = 2^{2^k}$

doubly-logarithmic tree

Level 1

Level 2

$\sqrt{n} = 2^{2^{k-1}}$ children

each node has $\sqrt{n} = 2^{2^{k-2}}$ children.

Level k

Level k+1

2 children.

} By construction

$k=2$ $n = 2^{2^2} = 16$

$n = 65536 = 2^{2^4}$

0 1 2 3

256 16 4 2 2

$0 \le i \le k$

# of children per node = $2^{2^{k-i-1}}$

# of nodes in level $i = 2^{2^k} - 2^{k-i}$

$i = k$

# of child per node = 2

# of nodes = $n/2$

Height of this tree = $k+1 = (\log\log n) + 1$

Time = $O(k) = O(\log\log n)$

Total work done in level i

$= O\left(\left(2^{2^{k-i-1}}\right)^2 \cdot \left(2^{2^k} - 2^{k-i}\right)\right) = O\left(2^{2^k}\right) = O(n)$ (n processors)

Work done in level k $= O\left(2^2 \times \frac{n}{2}\right) = O(n)$

Total work = $O(nk) = O(n \log\log n)$

---

1. Run balanced binary tree algo for $\log\log\log n$ levels.

$$n' = \frac{n}{\log\log n}$$

time = $O(\log\log\log n)$

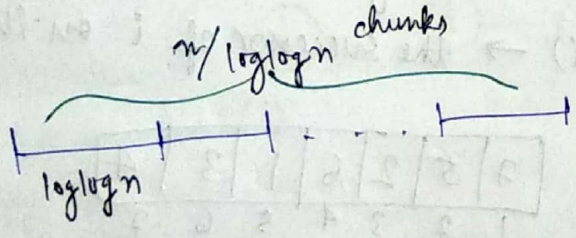
work = $O(n)$

2. Run the doubly logarithmic tree algo on these n' elements

Time = $O(\log\log n') = O(\log\log n)$

Work = $O(n' \log\log n')$

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



20/3/18

Stage 1: Use BBT for $\log\log\log n$ iterations to reduce the problem size to $n' = \frac{n}{\log\log n}$

- $O(\log\log\log n)$ time
- $O(n)$ work.

Stage 2: Apply the non optimal -

-algo (base on doubly log-depth) tree on the reduced sample.

- $O(\log\log n)$ time
- $O(n)$ work

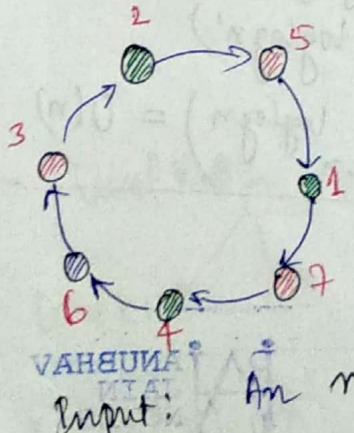
Instead of Stage 1, do

Stage 1': Break A into $n/\log\log n$ chunks each of size $\log\log n$. Compute in parallel max of each chunk by the obvious sequential algo.

- $O(\log\log n)$ time
- $O(n)$ work

Technique 7

Symmetry Breaking



Input: An n-cycle.

The successor array.

How to represent the cycle?

Successor array

$S(i) \rightarrow$ the successor of i on the cycle

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

Proper Coloring:-

If there is edge from i to j, then i & j will have different colors.

3-COLOR is NP-complete

Seq: Start with an arbitrary vertex
Alternately give colors 0, 1 to the vertices.
The last vertex may require the third color 2.

$O(n)$ time → optimal

$T^*(n) = O(n)$

Colors: $0, 1, \ldots, n-1$

Vertices: $1, 2, \ldots, n$.

Start with $C(i) = i-1$ [can be done in $O(1)$ time using $O(n)$ work]

Basic color contraction

Given a proper coloring C, to prepare another coloring C' s.t. no. of colors in C' < no. of colors in C

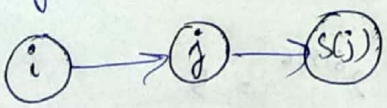
$O(1)$ time
$O(n)$ work

for $i = 1, 2, \ldots, n$ pardo

- find the least significant bit position k where $C(i)$ and $C(S(i))$ differ
  ($C(i) \neq C(S(i)) \Rightarrow k$ exists)
  (special hardware to do this in $O(1)$ time)
- set $C'(i) = 2^*k + C(i)_k$
  (k-th bit of $C(i)$) → $O(1)$ time

Claim: C' is again a valid coloring of the cycle.

Proof:



$j = S(i)$

$C'(i) = C'(j)$

$C'(i) = 2k + C(i)_k$ → 0 or 1

$C'(j) = 2l + C(j)_l$

$\Downarrow$

$\left\lfloor \frac{C'(i)}{2} \right\rfloor = \left\lfloor \frac{C'(j)}{2} \right\rfloor$

$k = l$

$C(i)_k = C(j)_l$

$C(i)_k = C(j)_k$

This contradicts the choice of k. $\Rightarrow C'(i) \neq C'(j)$

C uses q colors

(q-1) is a t-bit number for $t > 3$

(colors: 0, 1, ... q-1)

$2^{t-1} < q \leq 2^t$ ⟶ $(t < \log q + 1)$

$k \leq t-1$

$t-1 \; \ldots \; 1 \; 0$

$c'(i) \leq 2k + c(i)_k$

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

$= 2t-1 = 2\log q + 1.$

$c'(i) \leq 2\lfloor \log q \rfloor + 3$

[from q colors to $O(\log q)$ colors

$t=3$

$c \rightarrow 0, 1, 2, 3, 4, 5, 6, 7$

$c' \rightarrow 0, 1, 2, 3, 4, 5$

max → ($k_{max}=2$ $c'(i)=2k+c(i)_k$ $=4+1=5$)

If we iteratively use the color-contraction algorithm, we end up with a valid 6-coloring of cycle.

$\log^* n = u$ s.t. u is the smallest integer for which

$\underbrace{\log(\log(\log(\ldots(\log x))\ldots)}_{u \text{ times}} \leq 1$

$\log^* 1 = 0$

$\log^* 2 = 1$

$\log^* 4 = 2$

$\log^* 16 = 3$

$\log^* 65536 = 4$

$\log^* 2^{65536} = 5$

$O(\log^* n)$ iterations to reduce to 6-color case.

$O(\log^* n)$ time

$O(n \log^* n)$ work. ⟶ Overall stats. { practically OK; Theoretically not

From 6-coloring to 3-coloring

Given S, to produce P

S ← successor array

P ← predecessor array

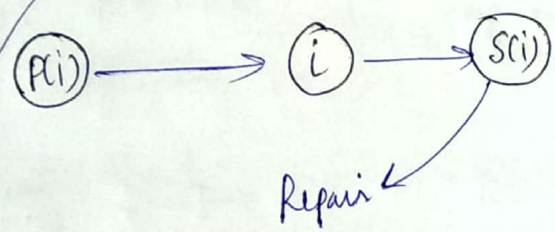
for $i = 1, 2, \ldots, n$ pardo
    set $P(S(i)) = i$

$O(1)$ time
$O(n)$ work

for $i = 1, 2, \ldots, n$ pardo
    [if $C(i) \neq 0, 1, 2$]
    assign to i a color in $\{0, 1, 2\}$
    different from the colors of $S(i)$ & $P(i)$
    [otherwise copy the original color of i]

$O(1)$ time
$O(n)$ work

- I/p & o/p in diff arrays.



- Remove 5
- Remove 4
- Remove 3.

- $C(i) = i - 1$ in parallel for all i
- Run the contraction algo once.
- Sort the vertices with respect to $C'(i)$

(n_x vertices of color x)

for $x = 3, 4, \ldots, 2\lfloor \log n \rfloor + 2$ do
    for $1 \leq i \leq n$ pardo
        change all vertices of color x to an admissible one in $\{0, 1, 2\}$.

Total work $= O\left(\sum_{x=3}^{2\lfloor \log n \rfloor + 2} n_x\right)$
$= O(n)$

time $= O(\log n)$
Work $= O(n)$

} provided sort can be done in this time & work bounds
↓ $O(\log n)$ ↓ $O(n)$

21/8/18

Optimal 3-coloring

n nodes $\leq 2\lfloor \log n \rfloor + 3$ colors } Seq Counting Sort can be used (Book: Radix) $O(n)$ time

To sort the nodes w.r.t their colors

General-purpose sorting $\Rightarrow$ $O(\log n)$ time
$O(n \log n)$ work

Exer: Parallelize this optimally.

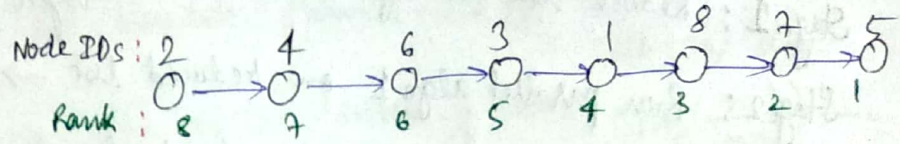
21/8/18

§ Lists and Trees

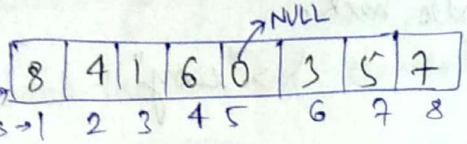
- Linked list is a special case of a Tree.

Lists L

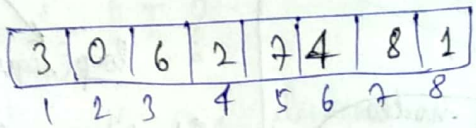
$1, 2, 3, \ldots, n$



Successor S

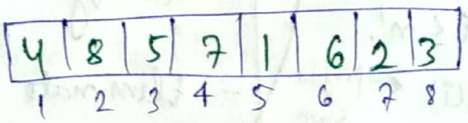


Predecessor P



List Ranking

Rank R



Given S (and P), compute R.

ID Rank

Seq: O(n) time [Find node with successor = 0 (NULL) → 5 : 1
keep ranking using predecessor array P(5) = 7 : 2]

Optimal

Parallel Algo 1

Pointer Jumping

```
for 1 ≤ i ≤ n pardo {
    copy S(i) to T(i)
    if S(i) = 0, set R(i) = 0, else set R(i) = 1
}
for 1 ≤ i ≤ n pardo {
    while (T(i) ≠ 0) and (T(T(i)) ≠ 0) {
        update R(i) := R(i) + R(T(i))
        set T(i) = T(T(i))
    }
}
```

O(1) time
O(n) work

O(log n) time

O(n log n) time

only T
↓
CREW, EREW PRAM

using T_1 & T_2, R_1 & R_2
$T_2(T_1(i))$

Parallel Algo 2

Stage 1: Reduce the size of the problem to $\frac{n}{\log n}$ by eliminating a set of nodes. ↓ O(n) work, O(log n · log log n) time

Stage 2: Run parallel algo 1 on reduced list → O(n) work, O(log n) time

Stage 3: Insert the eliminated nodes back. ↳ "easy"

Eliminate (n, S, P, R, I)

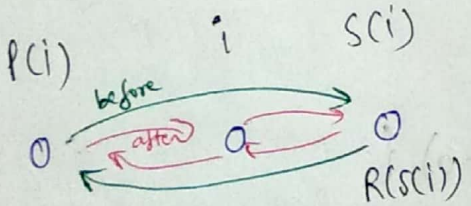
- Let $n' = |I|$. Number the nodes of I in the range $1 \le i \le n'$. — O(log n) time, O(n) work — Store these numbers in N(i) ← prefix sum
- for $1 \le i \le n$, pardo
  if ($i \in I$) {
  INFO (N(i)) = (i, S(i), P(i), R(i))
  R(P(i)) += R(i)
  P(S(i)) = P(i)
  S(P(i)) = S(i)
  }

O(1) time, O(n) work

loop (sequential)

- Identify a set I of nodes to eliminate
- Eliminate I.

I does not contain consecutive nodes.



27/8/18

List Ranking

Input: S[1...n], P[1...n]

Output: R[1...n]

- Take $n_0 := n$, and $k = 0$
- while ($n_k > n/\log n$), repeat : /* sequential */
  Find an independent set I in the remaining list → O(n_k) work, O(log n_k) time ← O(log n)

→ # of iterations $\le \log_{4/3} \log n = O(\log \log n)$

Record and remove I from the list → $O(n_k)$ work, $O(\log n_k)$ time

Set $k = k+1$

Remember the list elements from $1 \ldots n_k$ → $O(n_k)$ work, $O(\log n_k)$ time

- Run pointer-jumping algo on the remaining list → $O\left(\log\left(\frac{n}{\log n}\right)\right)$ time
- Insert back the deleted elements → $O(\Sigma n_k) = O(n)$ work, $O(\log\log n)$ time

$O(\log n)$ $O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right)$ work

$O(n)$

Q How to find I?

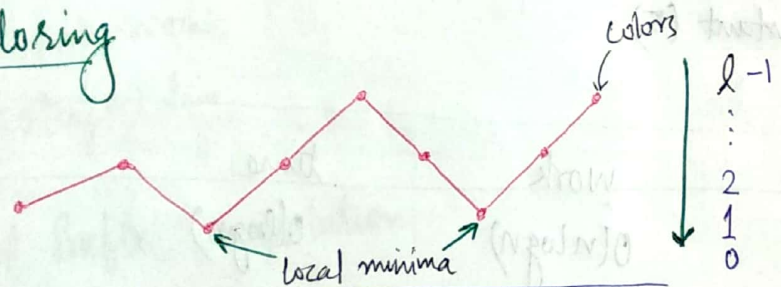
L_k → list with n_k elements

3-color L_k $(0, 1, 2)$

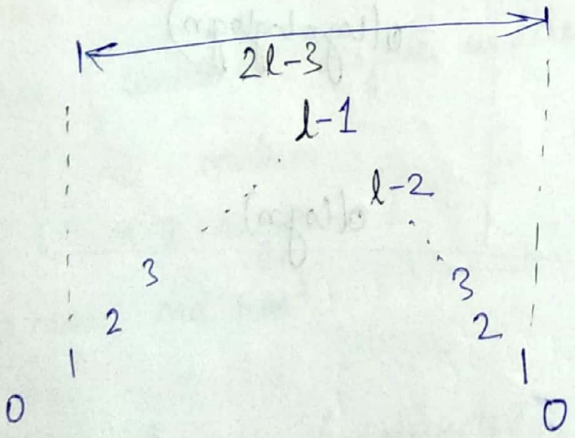
$O(n_k)$ work

$O(\log n_k)$ time

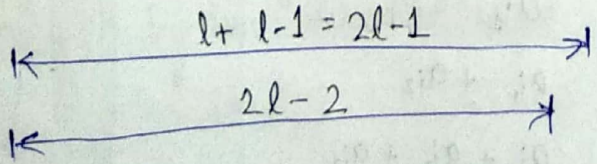
l-coloring



The set of all local minima is independent



$$|I| = \Omega\left(\frac{n_k}{2l-3}\right) = \Omega\left(\frac{n_k}{l}\right)$$



$l = 3$

$$|I| \leq \left\lceil \frac{n_k}{4} \right\rceil + \frac{1}{2}$$

$$|I| \geq \frac{n_k}{5}$$

ANUBHAV JAIN NOTES

$$n_{k+1} \leq \frac{4}{5} n_k$$

$$\forall k, \quad n_k \leq \left(\frac{4}{5}\right)^k \cdot n_0$$

$$= \left(\frac{4}{5}\right)^k n$$

Running time

$$= O(\log n \log\log n)$$

Work $= O\left(\sum_k n_k\right)$

$$= O\left(\sum_k \left(\frac{4}{5}\right)^k n\right)$$

$$= O\left(n \underbrace{\sum_k \left(\frac{4}{5}\right)^k}_{\leq \text{Constant } (5)}\right)$$

$$= O(n)$$

	work	time
Pointer Jumping	$O(n \log n)$	$O(\log n)$
Size reduction, then pointer jumping	$O(n)$	$O(\log n \log\log n)$
	$O(n)$	$O(\log n)$

Desired → can be achieved

[* section in book]

Generalized Prefix Sums

$b, s(b), s(s(b)), \ldots, e$ in $i_1, i_2, i_3, \ldots$

a_{i_1}

$a_{i_1} + a_{i_2}$

$a_{i_1} + a_{i_2} + a_{i_3}$

...

$a_{i_1} + a_{i_2} + a_{i_3} + \ldots + a_{i_n}$

Approach 1

Compute ranks from beginning

$i_1 - 1$ $\quad B[j] = A[i]$

$i_2 - 2$ $\quad$ Do normal prefix sum on B

$\vdots$

$i_j - j$ $\quad O(n)$ work

$\vdots$ $\quad O(\log n)$ time

$i_n - n$

Approach 2

Init: $R(i) = 1$

Instead: $R(i) = a_i$

Run the list-ranking algo

$O(n)$ work

$O(\log n)$ time

Parallel Prefix computation

Euler tour techniques

Tree (connected acyclic undirected graph)

n vertices

$n-1$ edges.

For now, no root.

Replace each edge (undirected) (u,v) by two directed edges (u,v) & (v,u)

For each vertex,

in-degree = out-degree.

[necessary] obvious

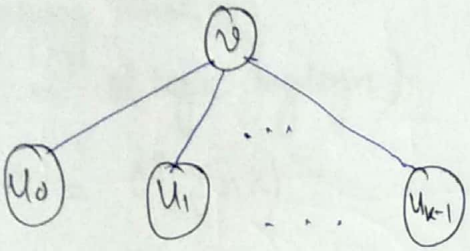
[sufficient] to prove

To specify

$s(u,v)$ $\forall (u,v)$

↑ Successor of directed edge (u,v)

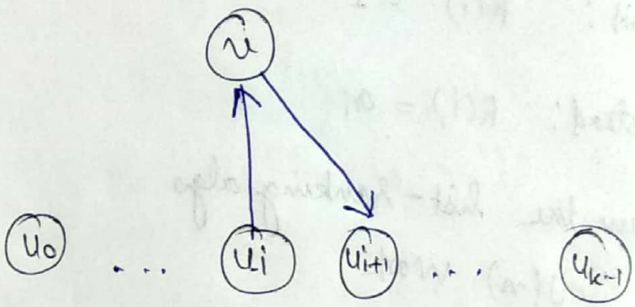
to compute an Eulerian tour in the tree.

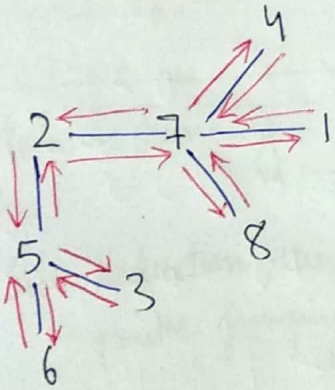


v has k neighbors

Define $s(u_i, v) = (v, u_{(i+1) \bmod k})$

Impose an ordering of the neighbours.



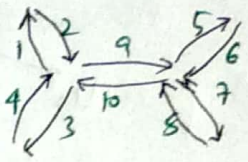


Vertex	ordered list of neighbors
1	7
2	5, 7
3	5
4	7
5	3, 6, 2
6	5
7	2, 8, 1, 4
8	7

Directed edge (u,v)	Successor $s(u,v)$
(4,7)	(7,2)
(7,4)	(4,7)
(1,7)	(7,4)
(7,1)	(1,7)
(7,8)	(8,7)
(8,7)	(7,1)
(2,7)	(7,8)
(7,2)	(2,5)
(2,5)	(5,3)
(5,2)	(2,7)
(3,5)	(5,6)
(5,3)	(3,5)
(5,6)	(6,5)
(6,5)	(5,2)

Claim: This defines an Eulerian tour in T

To show this is a "continuous" tour (no jumps)



[Proof by induction on n]

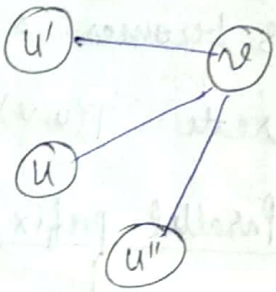
[base] $n = 2$

[Induction]

Let u be any leaf in T with $n \geq 3$

↓

(atleast 1 leaf exists bcoz T is acyclic)

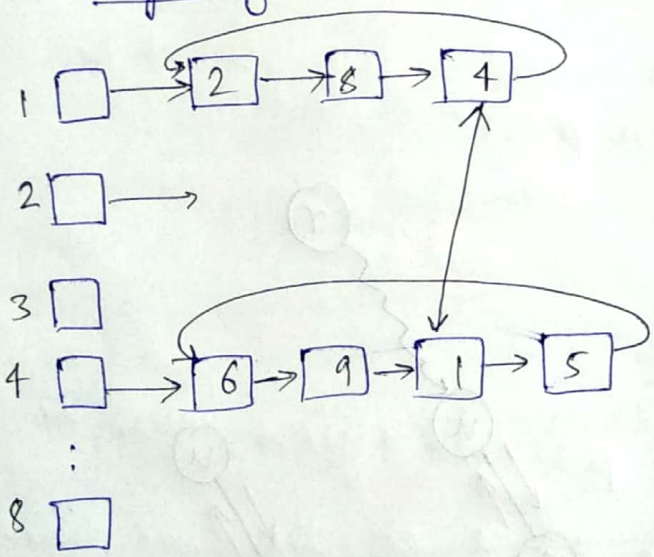


Splice u', u'' as u', u, u''

$\ldots u', u'' \ldots$ → The sequence of neighbors of v

Input

Adjacency list



$S(1,4) = (4,5)$

Two sets of pointers

- Linked list pointers
- $(u,v) \in E$

pointer from u's adjacency list node, storing v to v's adjacency list node storing u

The Eulerian tour can be computed in $O(1)$ time using $O(n)$ work.

28/8/18

Some Problems on Trees

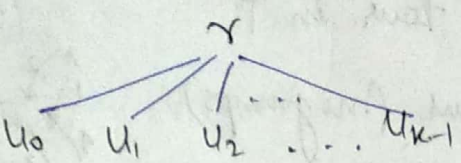
- Rooting a tree

$G = (V, E)$

Pick some $r \in V$ to be the root.

This imposes a child-parent relationship on the nodes of G

$\forall\, v \in V$, compute $p[v]$ (parent)



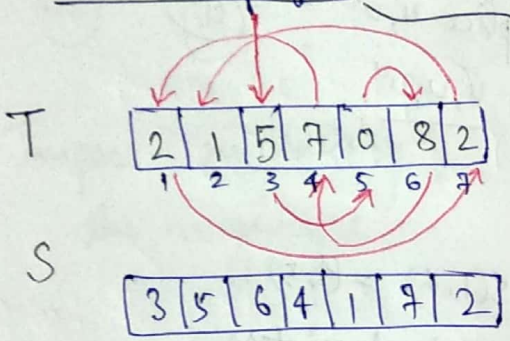
S → successor function defining the Eulerian tour.

$S(u_{k-1}, r) = 0$

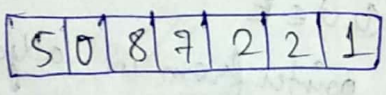
S becomes a linked list.

Create $T(u,v) = 1$ ∀ directed arc (u,v) of G

Parallel prefix on T w.r.t the ordering given by S. → store in U



T in the view of S



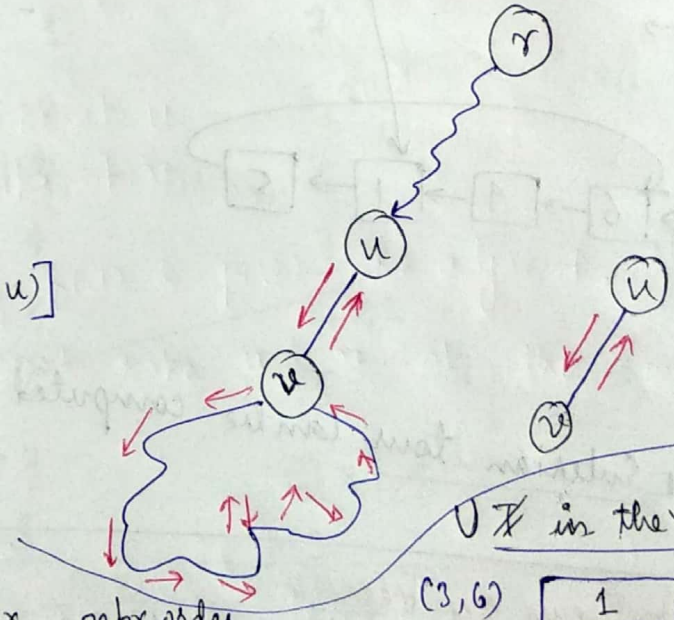
Prefix Sums

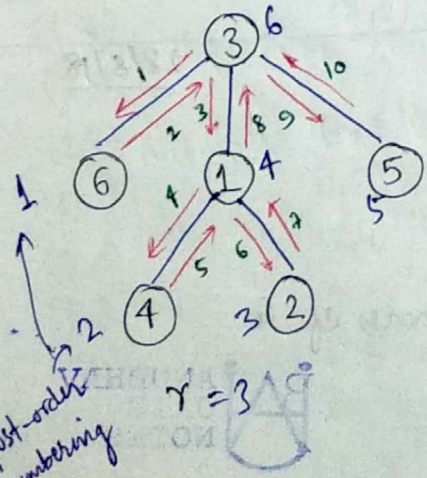
U 5 5 13 20 22 24 25

$(u,v) \in E$

$U[(u,v)] < U[(v,u)]$

set $P[v] = u$





Vertex	nbr order
1	4, 2, 3
2	1
3	6, 1, 5
4	1
5	3
6	3

O(log n) time
O(n) work

U in the view of S is

(3,6)	1
(6,3)	2
(3,1)	3 ←
(1,4)	4
(4,1)	5
(1,2)	6
(2,1)	7
(1,3)	8 ←
(3,5)	9
(5,3)	10

P[r] = 0

- Post order numbering

post[v] - ~~time~~ order, at which the tour is returning from v.

T in the view of S

(3,6)	0	0 ←
(6,3)	1	1
(3,1)	0	1
(1,4)	0	1
(4,1)	1	2 ←
(1,2)	0	2
(2,1)	1	3 ←
(1,2)	1	4 ←
(3,5)	0	4
(5,3)	1	5 ←

Post-order no. for root=6.

$\forall v$ set $(v \neq r)$

$T[v, p[v]] = 1$

$T[p[v], v] = 0$ ←

$$\text{set post}[v] = \begin{cases} \cup[v, p(v)] & \text{if } v \neq r \\ r & \text{if } v = r \end{cases}$$

Level of all nodes

$\forall v \neq r$ set

$T[(p(v), v)] = 1$

$T[(v, p(v))] = -1$

$$\text{level}[v] = \begin{cases} \cup[(p(v), v)] & \text{if } v \neq r \\ 0 & \text{if } v = r \end{cases}$$

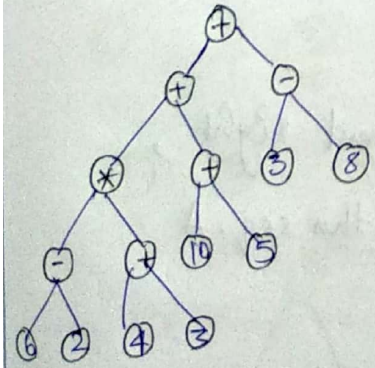
$v \neq r$

$\cup[(v, p(v))] - \cup[(p(v), v)]$ = # of nodes in the subtree rooted at v

---

Arithmetic Expression Evaluation

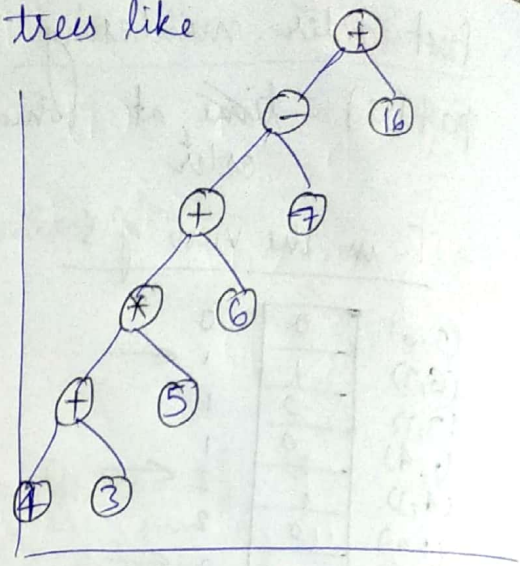
3/9/18



- → operators - binary
- leaves - values
- internal nodes - operators.

* Proceed from leaves to root
* Basic aim: To compute the value at the root.
* Additional Demand: Compute values at every node

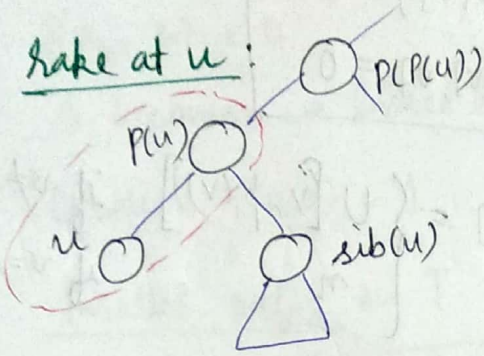
The process is inherently sequential on trees like



A primitive operation :-

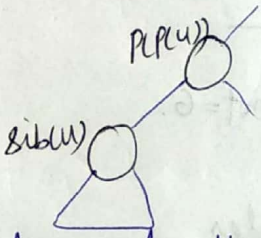
Rake at a leafe.

Rake at u :

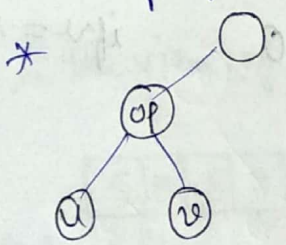


Assume $p(u) \neq root(1)$

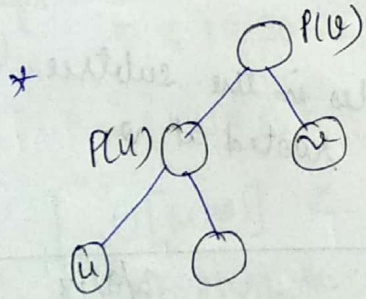
- Delete u and p(u)
- Attach sib(u) to p(p(u))



Aim: To perform raking at multiple leaves together



Don't rake at consecutive leaf nodes at the same time ($p(u) = p(v)$)



rake at u
rake at v

If p(u) & p(v) are adjacent, then we can't rake u & v at the same time.

Parallel tree contraction

- Number the leaves consecutively from left and right, excluding leftmost & rightmost nodes.
- call this seq. A

Use Euler Tour Technique

- leaf node have sub-tree size 1
- post order numbering from left to right
- prefix sum

O(log n) time
O(n) work.

$A = (a_1, a_2, a_3, \ldots)$

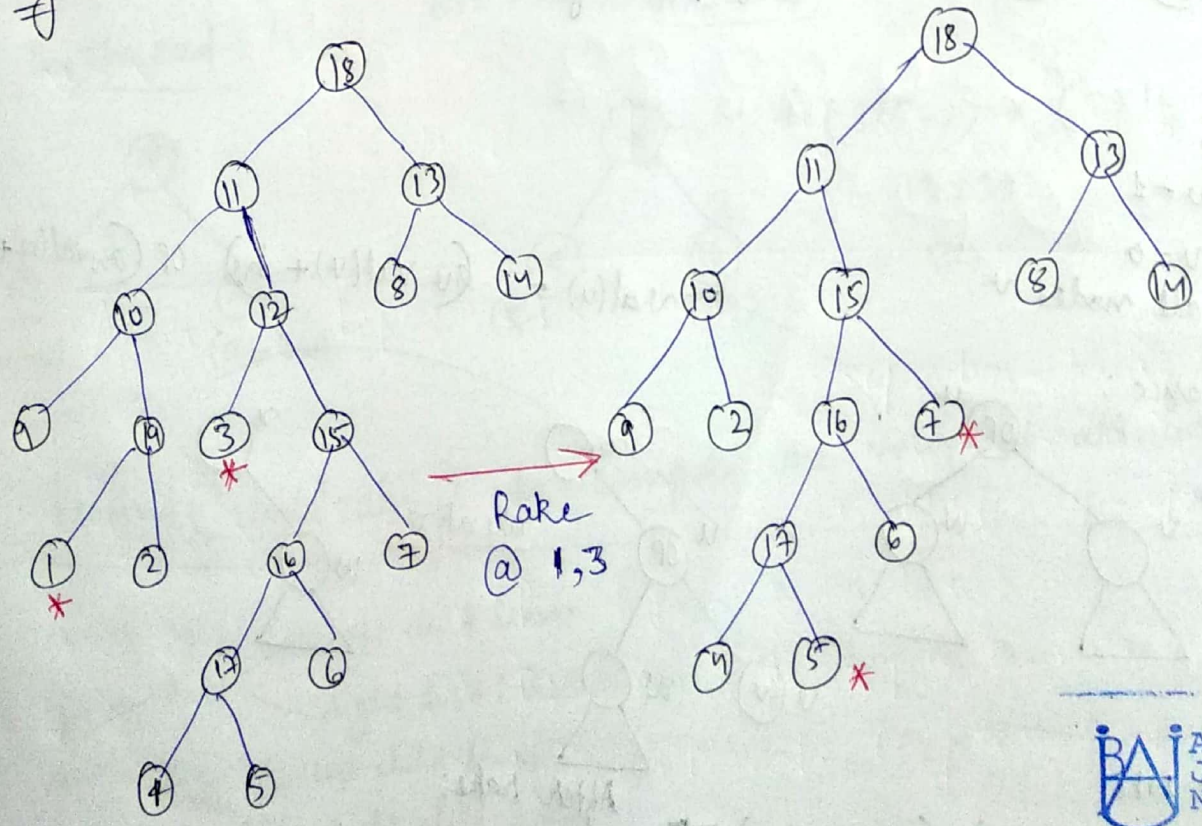
$A_{odd} = (a_1, a_3, a_5 \ldots)$

$A_{even} = (a_2, a_4, a_6, \ldots)$

while $|A| > 0$, do : → $\lceil \log n \rceil$ iterations, $n = |A|$ initially.

- parallel rake at all elements of A_{odd} that are left children
- parallel rake at the remaining leaves in A_{odd}
- set $A := A_{even}$

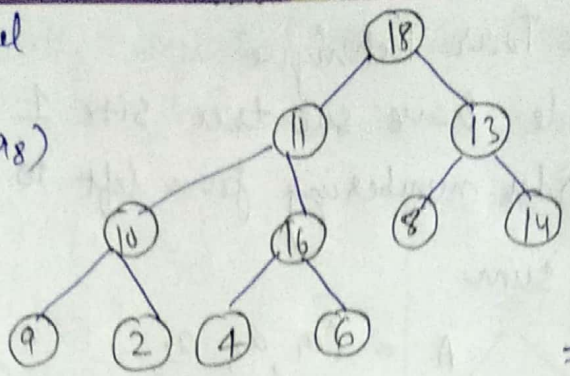
Eg.



Rake @ 5, 7 in parallel

$A = A_{even} = (a_2, a_4, a_6, a_8)$

$|A_{even}| = \left\lfloor \frac{|A|}{2} \right\rfloor$

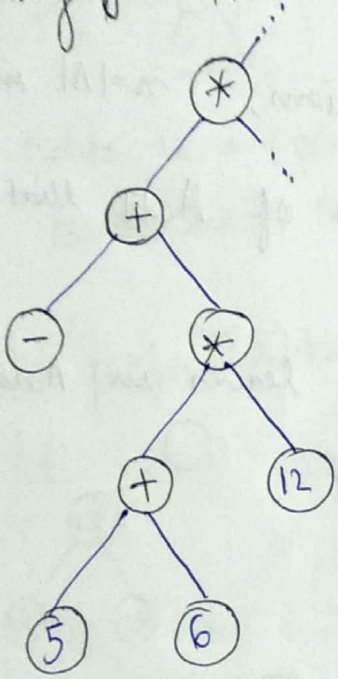


Running time $= O(\log n)$

Work done $= O(\sum |A_{odd}|)$

$= O\left(\sum \frac{n}{2^k}\right) = O(n)$

Raking for Arithmetic Expression Evaluation



Operations : $+, -, *$ (binary)

↓

unary minus can be $(-x)$ replaced by $(0-x)$

After raking at 8, sibling of 8 will record the information of the deleted nodes.

Store at each node v a pair (a_v, b_v).

If X is the value at v, then currently v stands for $a_v X + b_v$

Init:

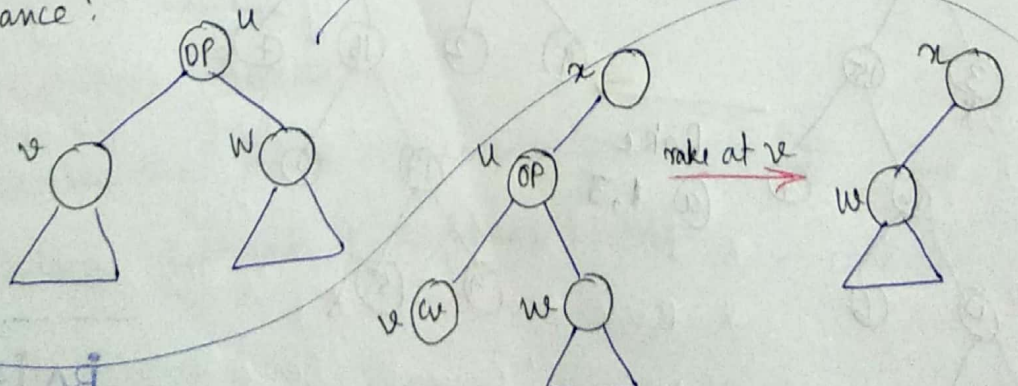
$a_v = 1$

$b_v = 0$

at all nodes v.

$\Rightarrow val(u) = (a_v\, val(v) + b_v)\ op\ (a_w val(w) + b_w)$

Invariance:



Before rake:

$val(u) = (a_v c_v + b_v)\ op\ (a_w x_w + b_w)$ → After rake: $val(w) = a_{w'} x_w + b_{w'}$

- OP = +

$a_w' x + b_w' = a_w x_w + (a_v c_v + b_v + b_w)$

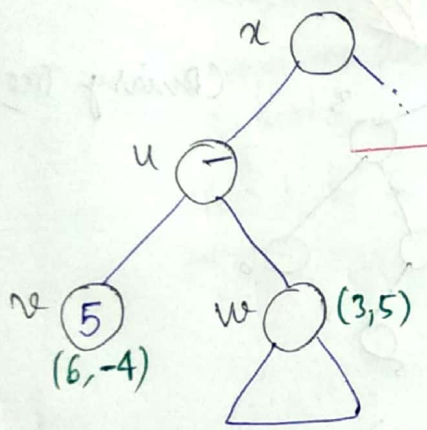
$\begin{cases} a_w' = a_w \\ b_w' = a_v c_v + b_v + b_w \end{cases}$

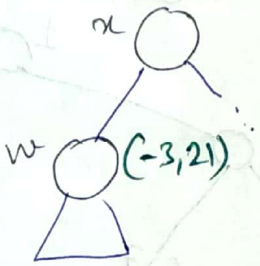
- OP = *

$a_w' x + b_w' = (a_v c_v + b_v) * (a_w x_w + b_w)$

$= (a_v c_v + b_v) a_w x_w + (a_v c_v + b_v) b_w$

$\begin{cases} a_w' = (a_v c_v + b_v) a_w \\ b_w' = (a_v c_v + b_v) b_w \end{cases}$



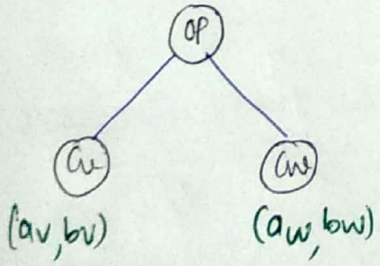


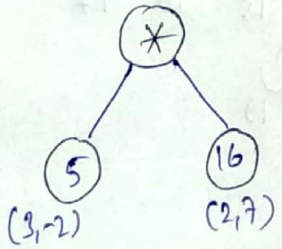
val(w) = (6×5−4) − (3 x_w + 5)

= −3 x_w + 21

63 = −3x_w + 21 ⇒ (x_w = −14)

In the end





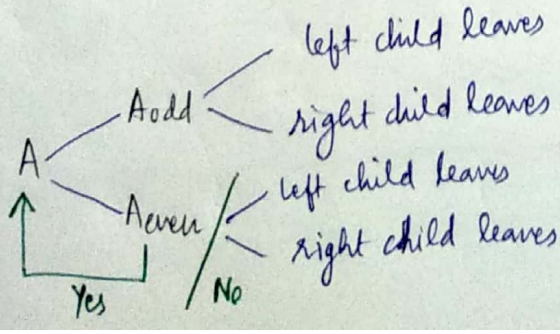
(3×5−2) * (2×16+7)

= 13 × 39

= 507

Rewind ("undo" taking) to compute the values at all internal nodes.

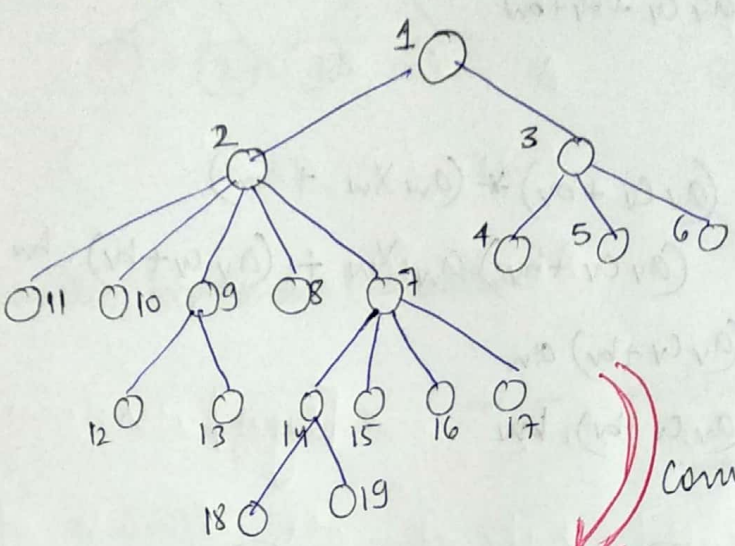
4/9/18



Computing minimum at every node:

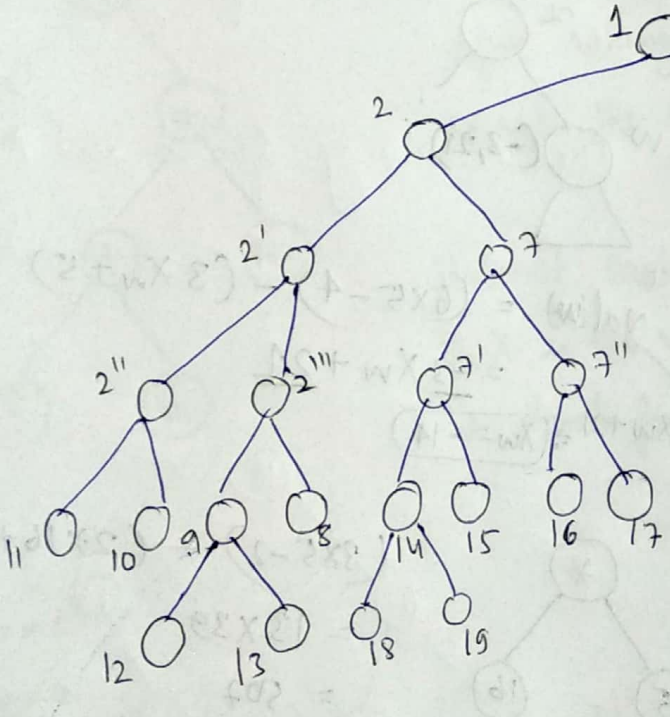
Initially; Every node contains a number

Finally: Every node computes the minimum in the subtree rooted at it.

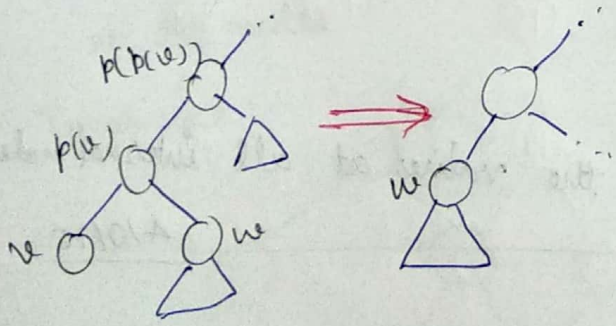


converts to

(Binary Tree)



Apply tree contraction



$$p(p(v)) = \min(p(p(v)), p(v), v)$$

Lowest common Ancestor (LCA)

LCA(12,19) = 2

LCA(17,19) = 7.

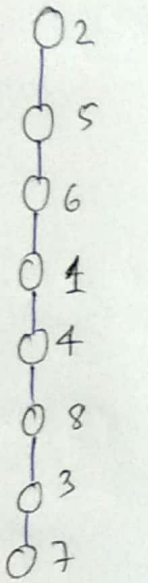
← Do some preprocessing

– Answer LCA queries as fast as possible.

  – O(1) time (ideal)

  – O(log n) ~~times~~ (ok)

Simple cases



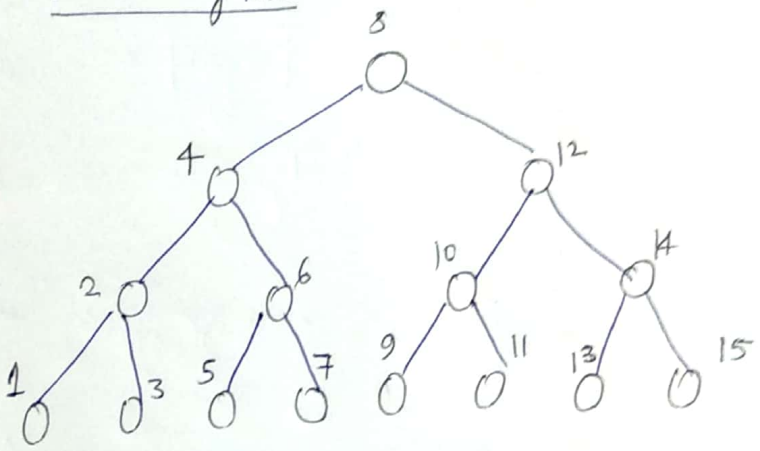
LCA(4,5) = 5

(Report no. closer to root)

↓

store depth in pre-processing

Full Binary Tree



in-order numbering (makes sense only for Binary Tree)

LCA(u,v)

$i = inorder(u) = r_{d-1} r_{d-2} \cdots r_k s_{k-1} \cdots s_0$

$j = inorder(v) = r_{d-1} r_{d-2} \cdots r_k t_{k-1} \cdots t_0$ (↕ ≠)

LCA(u,v) = node with inorder number $r_{d-1} r_{d-2} \cdots r_k 1 0 \cdots 0$

6 = 0110

15 = 1110

LCA(6,15) = 1000 = 8

15 = 1111

1 = 0001

LCA(15,1) = 1000 = 8

11 = 1011

13 = 1101

LCA(11,13) = 1100 = 12

ANUBHAV JAIN NOTES