A Further Improvement

- Parallel search first.
- Consider a sorted array A of n element and we want to search for an element x.
- Given p processors, we can always search at positions (indices) 1, n/p, 2n/p, ..., n.
- Record the result of each comparison as a 1 or 0 with 1 for position i indicating that $A[i] < x$ and 0 indicating that $A[i] >= x$.
- The sequence of p results will have :
  - Either all 1's : x is not in A
  - Either all 0's : x is not in A
  - A shift from 1's to 0's : x is likely in the n/p segment corresponding to the shift from 1 to 0.

Search in Parallel

- We can identify the next step depending on the three cases.
  - Either all 1's : x is not in A
  - Either all 0's : x is not in A
  - A shift from 1's to 0's : x is likely in the n/p segment corresponding to the shift from 1 to 0.
    - Therefore, search recursively in the corresponding segment of size n/p while still using p processors.
- The recurrence relation for the time taken is
  - $T(n) = T(n/p) + O(1)$, for a solution of $T(n) = O(\log_p n)$.

Search in Parallel

- Consider typical values of p.
- For p = O(1), no change in time taken asymptotically.
- For p = O(log n), the time taken is O(log n/ loglog n).
- For p = O($n^{1/2}$), the time taken is O(log n/log $n^{1/2}$) = O(1)!
  - Of course, looks like wasteful from a work point of view.
  - Let us see what it is good for!

$0 < \varepsilon < 1$ $p = n^{\varepsilon}$, $T(n) = O(1/\varepsilon)$, $W(n) = O(n^{\varepsilon})$

From Parallel Search to Merge

- Recall our idea to arrive at an optimal algorithm to merge two sorted arrays A and B.
- We rank a few elements of A in B to partition B into sub-arrays.
- Let us consider ranking $n^{1/2}$ elements of A in B.
- We have n processors, so each search can use $n^{1/2}$ processors!
- Each search now finishes in O(1) time.

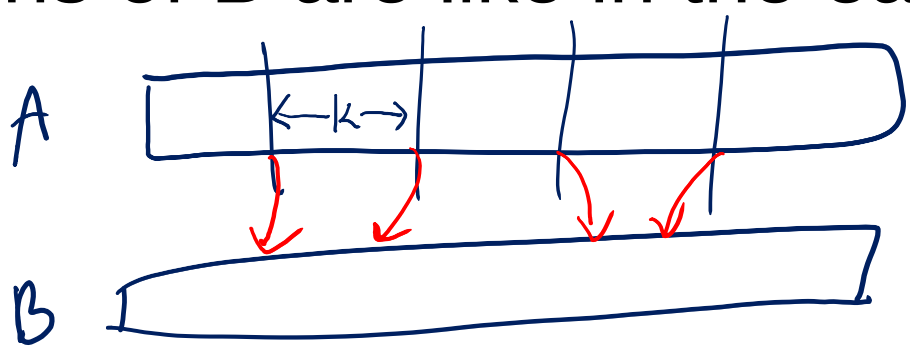
$T(n) = O(\log_p n)$ $W(n) = O(p \log_p n)$

$p = \sqrt{n}$: $T(n) = O(1)$ $W(n) = O(\sqrt{n})$

$p = O(1)$: $T(n) = O(\log n)$ $W(n) = O(\log n)$

From Parallel Search to Merge

- Let us consider ranking $n^{1/2}$ elements of A in B.
- We have n processors, so each search can use $n^{1/2}$ processors!
- Each search now finishes in O(1) time.
- There is a downside however.
- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.



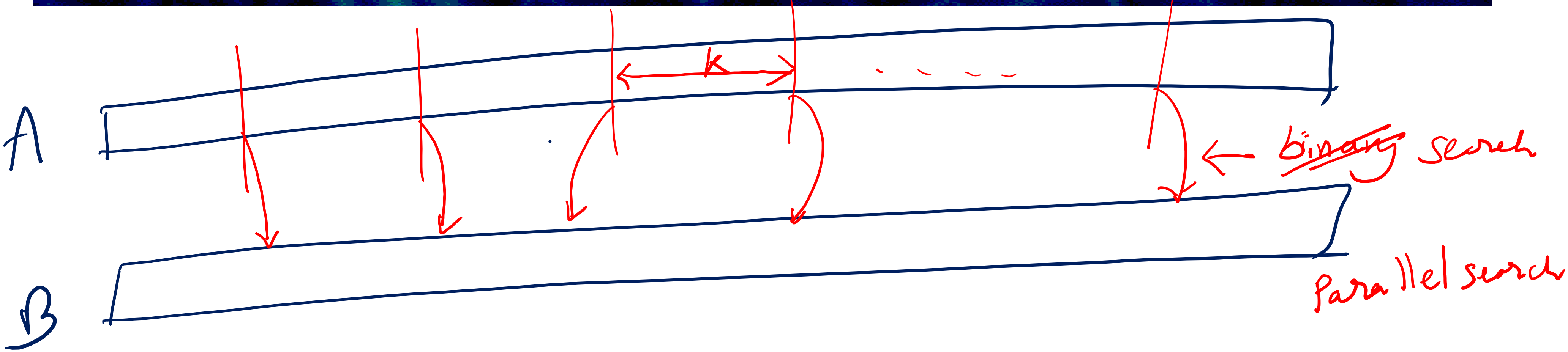
$k = \sqrt{n}$

#proc/search $= \sqrt{n}$

#ranks $= \sqrt{n}$

p processors / search : Time $= \log_p n$ //

Work $= p$ //



Merge

Rank $\frac{n}{k}$ elements of A into B

Total Work $= \frac{n}{k} \times p$, Total Time $= \log_p n$

find a k, p

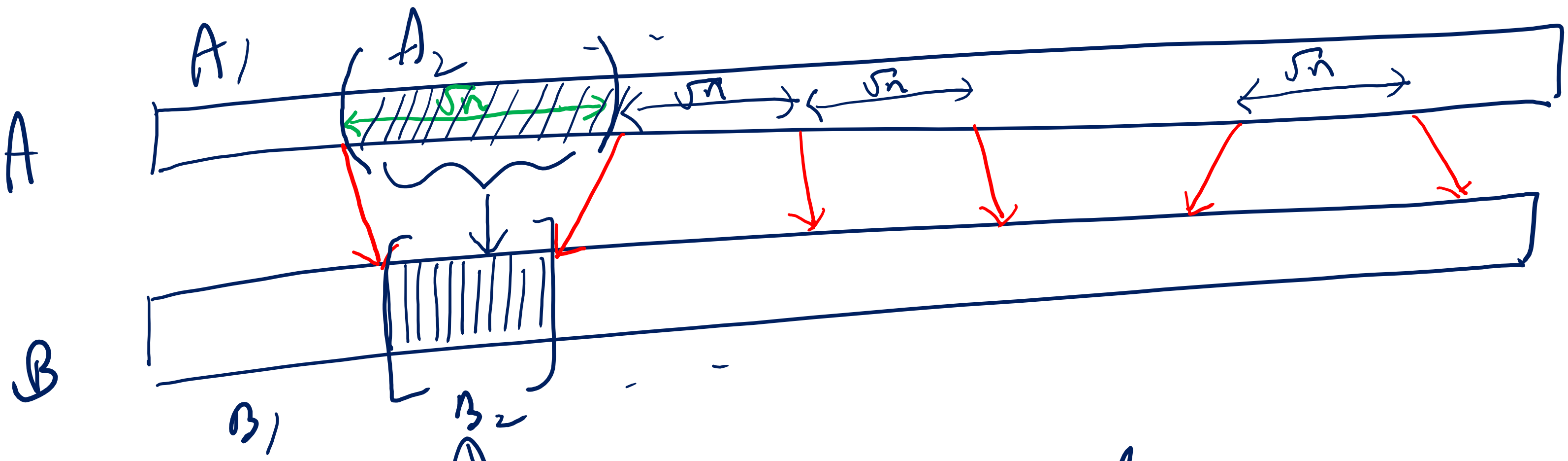
s.t. $\log_p n = O(1)$,

work $= O(n)$

Processors used across all ranks $= p \times \frac{n}{k}$

$p = \sqrt{n}$, $k = \sqrt{n}$

$p = n^{\epsilon}$, $k = n^{\epsilon}$



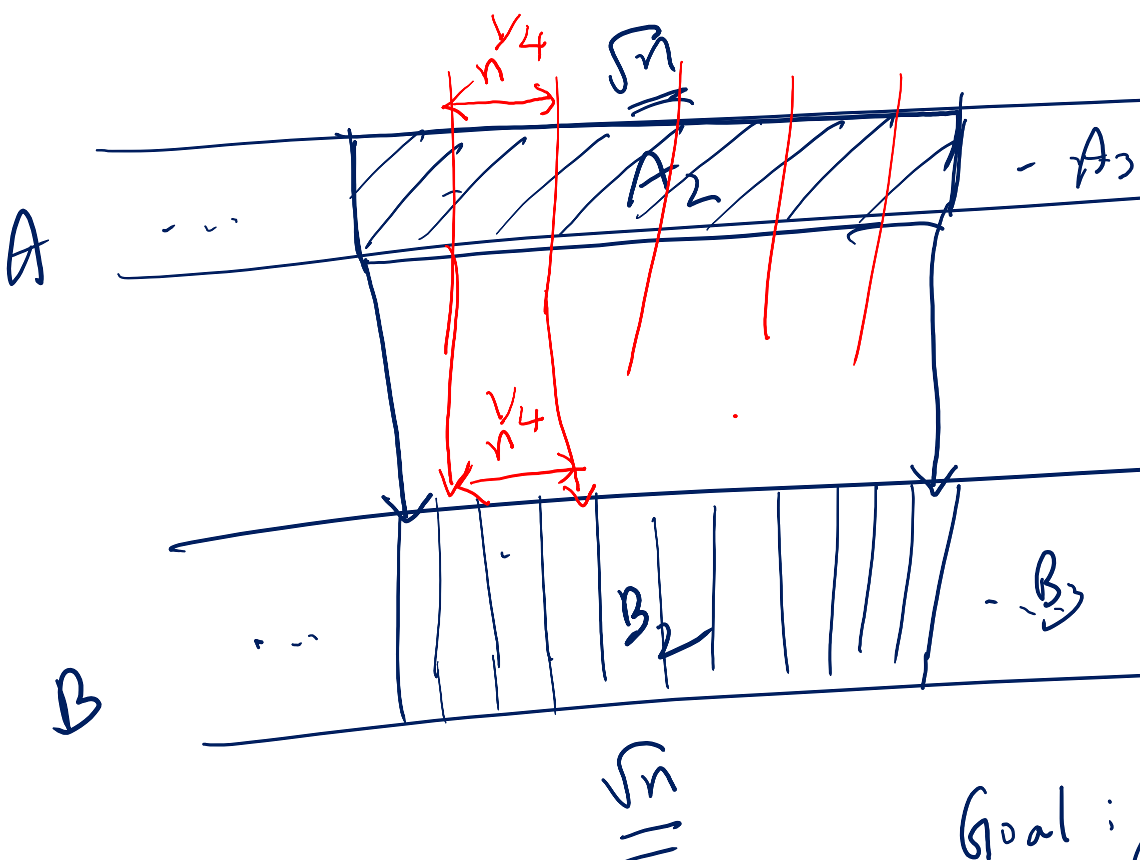
Reduce $|B_2|$ to $O(\sqrt{n})$

Merge in sequential manner

Time =
- $O(1)$ for searching / ranking
- $O(\underbrace{\text{length of } A_2}_{=\sqrt{n}} + \underbrace{\text{length of } B_2}_{\text{Dont know } = \sqrt{n}}) = O(\sqrt{n})$

Rank t elements of A_2 in B_2.

$\sqrt{n}$ Processors



$\text{Time} = \log_{p=\sqrt{n}/t} \sqrt{n}$ / search

$= \log \sqrt{n} / (\log\sqrt{n} - \log t)$

Total of $\sqrt{n}$ processors available.

$\#\text{proc}/\text{search} = \dfrac{\sqrt{n}}{t}$

$\text{work}/\text{search} = p = \sqrt{n}/t$

$\text{Total work} = \dfrac{\sqrt{n}}{t} \times t = \sqrt{n}$

Goal: Merge time $\sqrt{n}$ $n^{1/4}$

lengths of $|A_2|, |B_2|$

$$\boxed{\frac{\log\sqrt{n}}{\log\sqrt{n} - \log t} = O(1)}$$

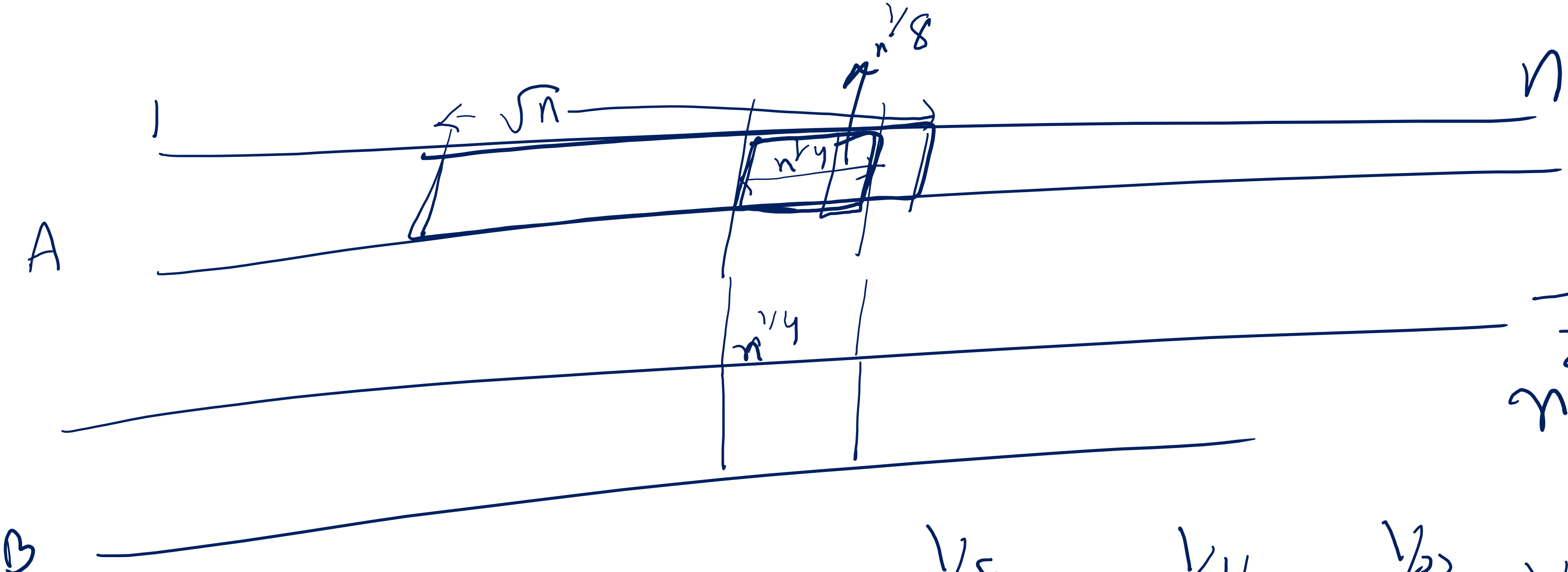
$t = n^{\frac{k-1}{2}}$

$t = n^{1/4}$

$\log\sqrt{n} = 2\log\sqrt{n} - 2\log t$

$2\log t = \log\sqrt{n}$

$\log t = \log\sqrt{\sqrt{n}}$



$n^{\frac{1}{2^i}} = O(1)$

$i = ?$

$$n \xrightarrow{1} \sqrt{n} \xrightarrow{2} n^{1/4} \xrightarrow{3} n^{1/8} \to n^{1/16} \to n^{1/32} \cdots n^{1/2^{i-1}} \xrightarrow{i} O(1)$$

$= n^{1/2^i}$

$\#\text{steps} = O(\log \log n)$

recurs.

From Parallel Search to Merge

- There is a downside however.
- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.
- But, can use recursion to make further progress.

From Parallel Search to Merge

- The partitions of A are now much longer at $n^{1/2}$ each.
- The partitions of B are like in the earlier case, unknown.
- But, can use recursion to make further progress.
- Recursively apply the same procedure on each partition of A into the corresponding partition of B.
- Notice that each part of A is only $n^{1/2}$ in size.
- We want to rank $n^{1/4}$ element of each part of A into the corresponding B.

From Parallel Search to Merge

- The recurrence relation guiding this process is captured by $T(n, m) = \max_i T(n^{1/2}, m_i) + O(1)$.
  - In the above, n and m refer to the length of A and B respectively.
  - And, m_i refers to the length of the i^{th} partition of B.
- Can show that $T(n,m) = O(\log \log n)$.
- Once recursion ends, each partition of A and partitions of B will be $O(\log\log n)$ long, and we merge them sequentially.

$O(1)$

$\text{Time} = O(\log\log n)$ $\rightarrow$ $O(\log\log n)$

$\text{Work} = O(n \log\log n)$

non optimal

$O(n)$

fun ↑↑

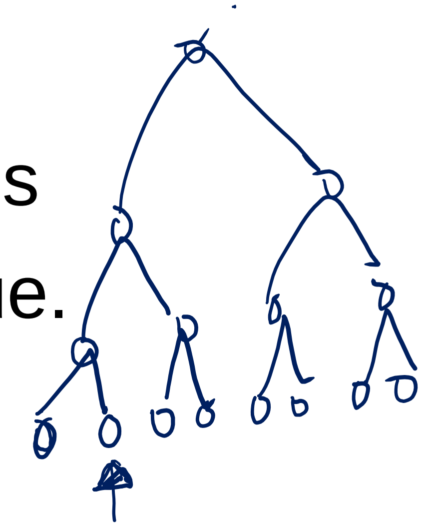
$n = 10^{100}$ $\quad \log\log n = 2$

$n = 10^{1000}$ $\quad \log\log n = 3$

→ solve a smaller problem

The Power of CRCW – Minima

- Two points of interest
  - Illustrate the power of CRCW models
  - Illustrate another optimality technique.
- Find the minima of n elements.
  - Input: An array A of n elements
  - Output: The minimum element in A.
- From what we already know:
  - Standard sequential algorithm not good enough
  - Can use an upward traversal, with min as the operator at each internal node. Time = O(log n), work = O(n).



O(n) time

The Power of CRCW – Minima

- Our solution steps:
  - Design a O(n²) work, O(1) time algorithm.
  - Gain optimality by sacrificing runtime to O(log log n).

Friday

An O(1) Time Algorithm

j

Processor

Processor

i

A	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	-- 1	1	1
18	0	0	0	--	1
26	0	0	0	0	--

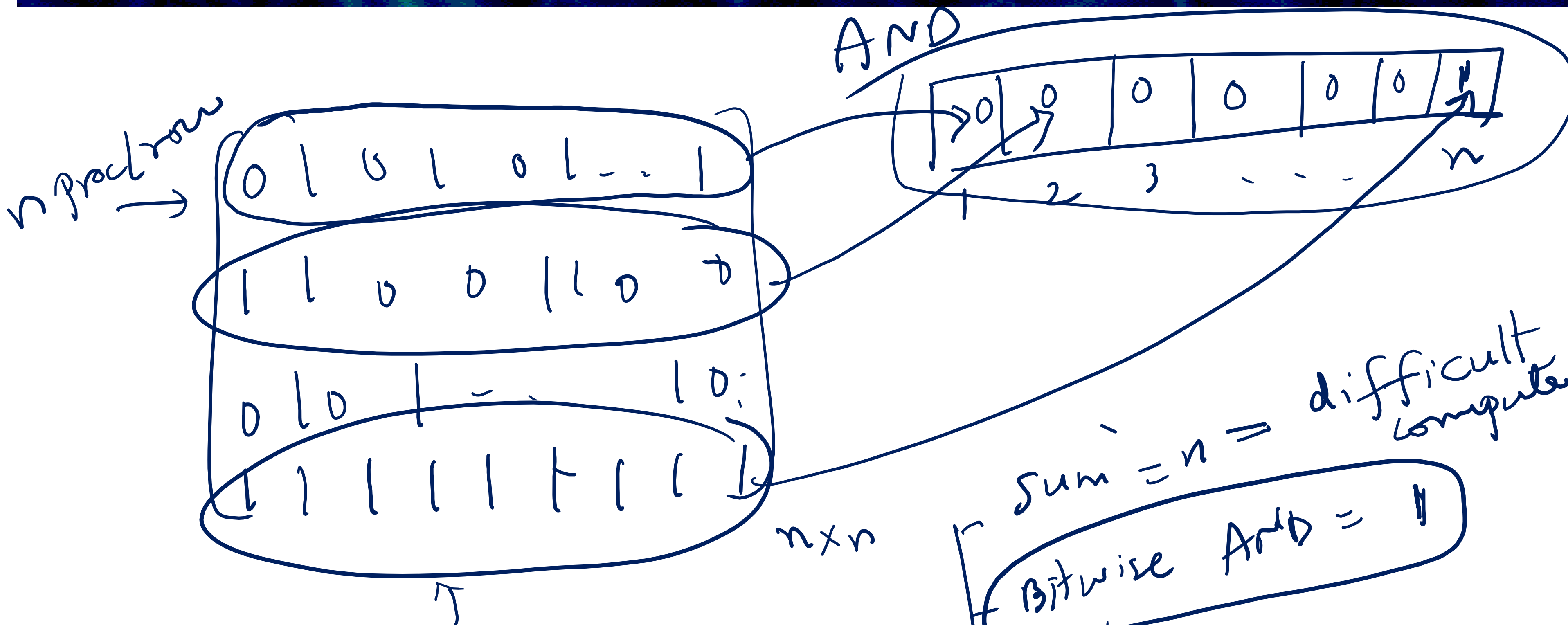
all 1's except the diagonal element

$n \times n$

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.

$A(i,j) = 0$ if $A(i) > A(j)$

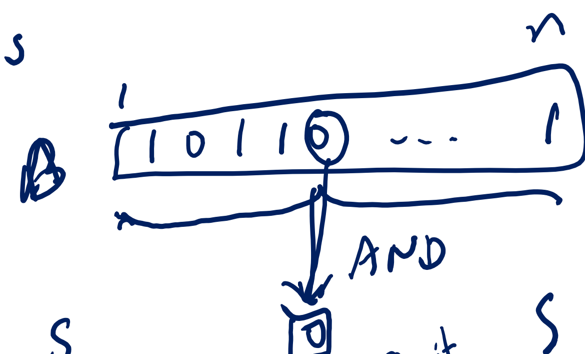
n^2 Processors



sum = n = difficult to compute

Bitwise AND = 1

CRCW helps



init: $S = \cancel{1}\ 0$

$$S = \begin{cases} 1 & \text{if all } B(i) = 1 \\ 0 & \text{if any } B(i) = 0 \end{cases}$$

if $B(i) = 0$ write 0 to S // for all i in parallel

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
  - How?

An O(1) Time Algorithm

	12	17	8	18	26
12	--	1	0	1	1
17	0	--	0	1	1
8	1	1	--	1	1
18	0	0	0	--	1
26	0	0	0	0	--

Work = n^2

Time = O(1)

- Use n^2 processors.
- Compare A[i] with A[j] for each i and j.
- Now can identify the minimum.
  - How?
- Where did we need the CRCW model?

Towards Optimality

- The earlier algorithm is heavy on work.
- To reduce the work, we proceed as follows.
- We derive an O(nlog log n) work algorithm running in O(log log n) time.
- For this, use a doubly logarithmic tree.
  - Defined in the following.

Work : $n^2 \longrightarrow n \log \log n$ not optimal

Time : $O(1) \longrightarrow O(\log \log n)$

optimal

start with a smaller problem size

$x = O\left(n/\log n\right)$

$x \log x = n$

$x = n$

$\log x + \log\log x = \log n$

LHS $\uparrow\uparrow$

$x < n$

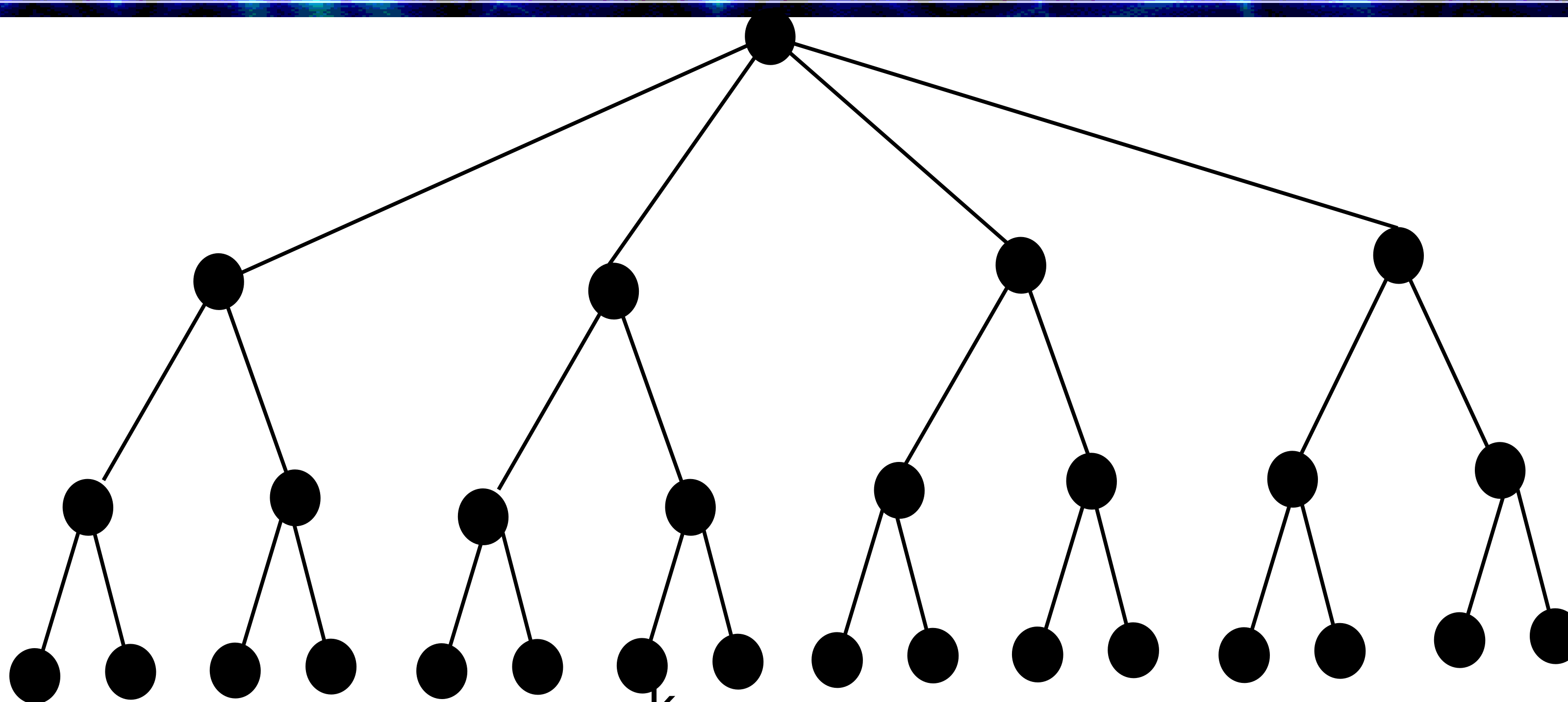
RHS $\uparrow\uparrow$

$x = \dfrac{n}{\log n}$

LHS: $\dfrac{n}{\log n} \times \log\left(\dfrac{n}{\log n}\right) = \dfrac{n}{\log n}(\log n - \log\log n)$

$= n - \dfrac{n \log\log n}{\log n}$

Doubly Logarithmic Tree



- Let there be $n = 2^{2^k}$ leaves, the root is level 0. The root has $\sqrt{n} = 2^{2^{k-1}}$ children.
- In general, a node at level i has $2^{2^{k-i-1}}$ children, for $0 \le i \le k-1$.
- Each node at level k has two leaf nodes as children.