

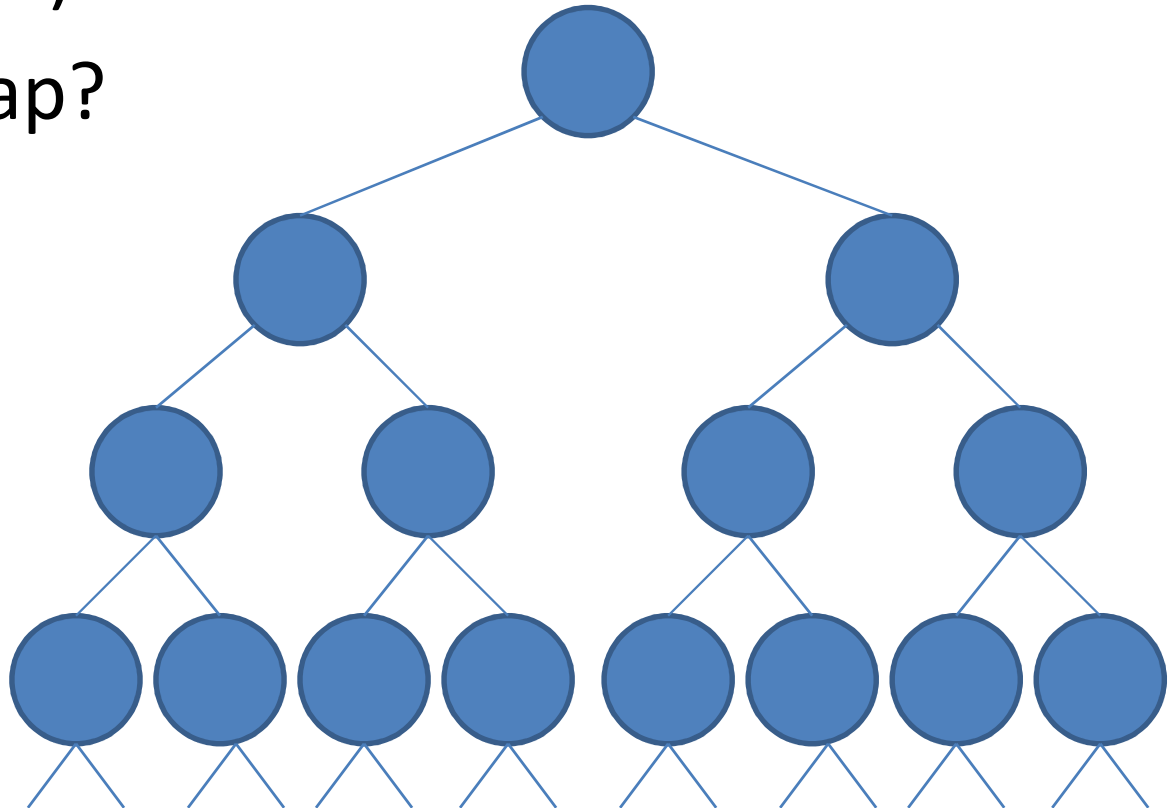
CSL 860: Modern Parallel Computation

PARALLEL ALGORITHM TECHNIQUES: BALANCED BINARY TREE

Reduction

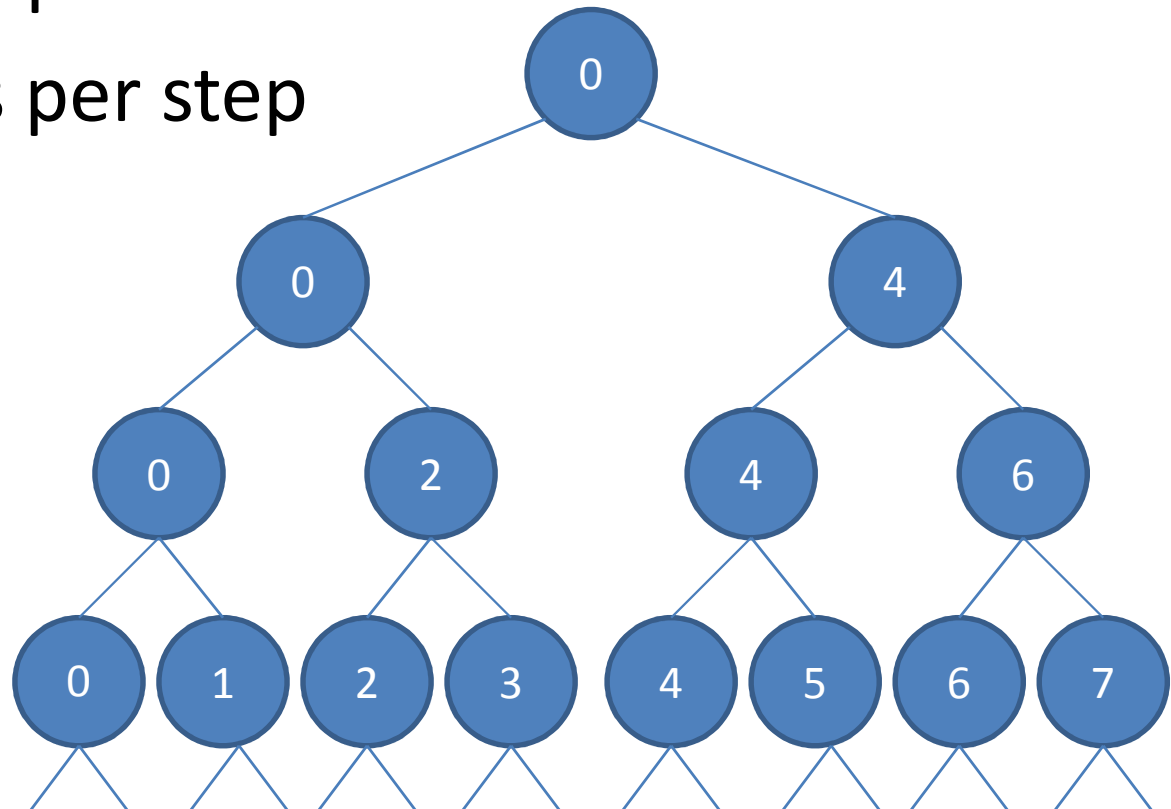
- n operands $\Rightarrow \log n$ steps
- Total work = $O(n)$
- How do you map?

**Balance Binary tree
technique**



Reduction

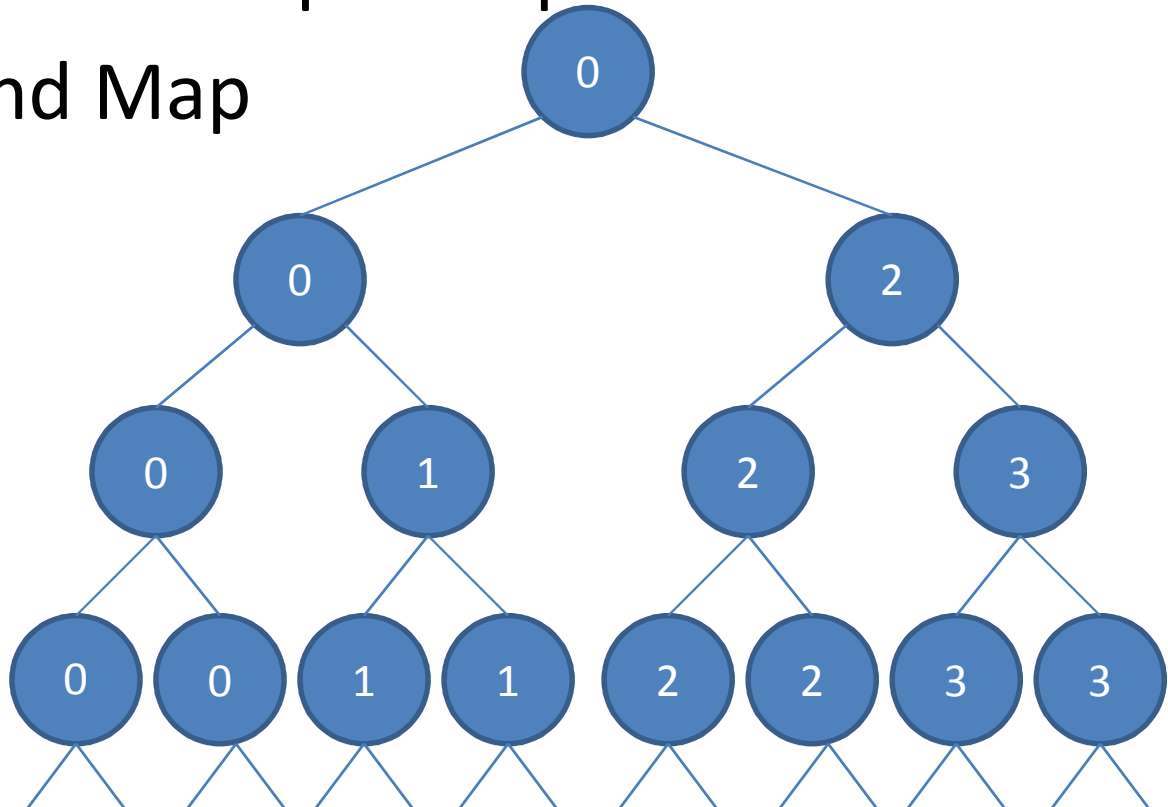
- n operands $\Rightarrow \log n$ steps
- How do you map?
- $n/2^i$ processors per step



Reduction

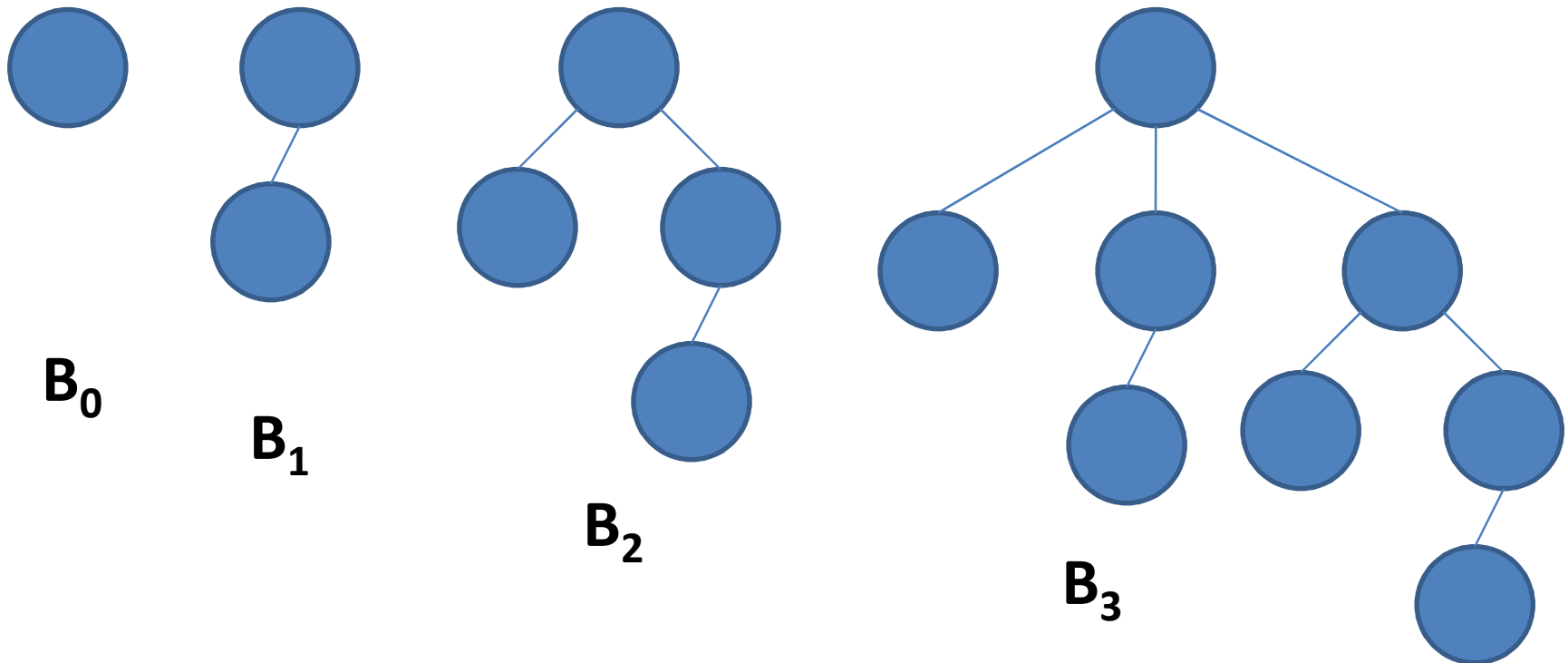
- n operands $\Rightarrow \log n$ steps
- Only have p processors per step
- Agglomerate and Map

**Processor dependence:
Binomial tree**



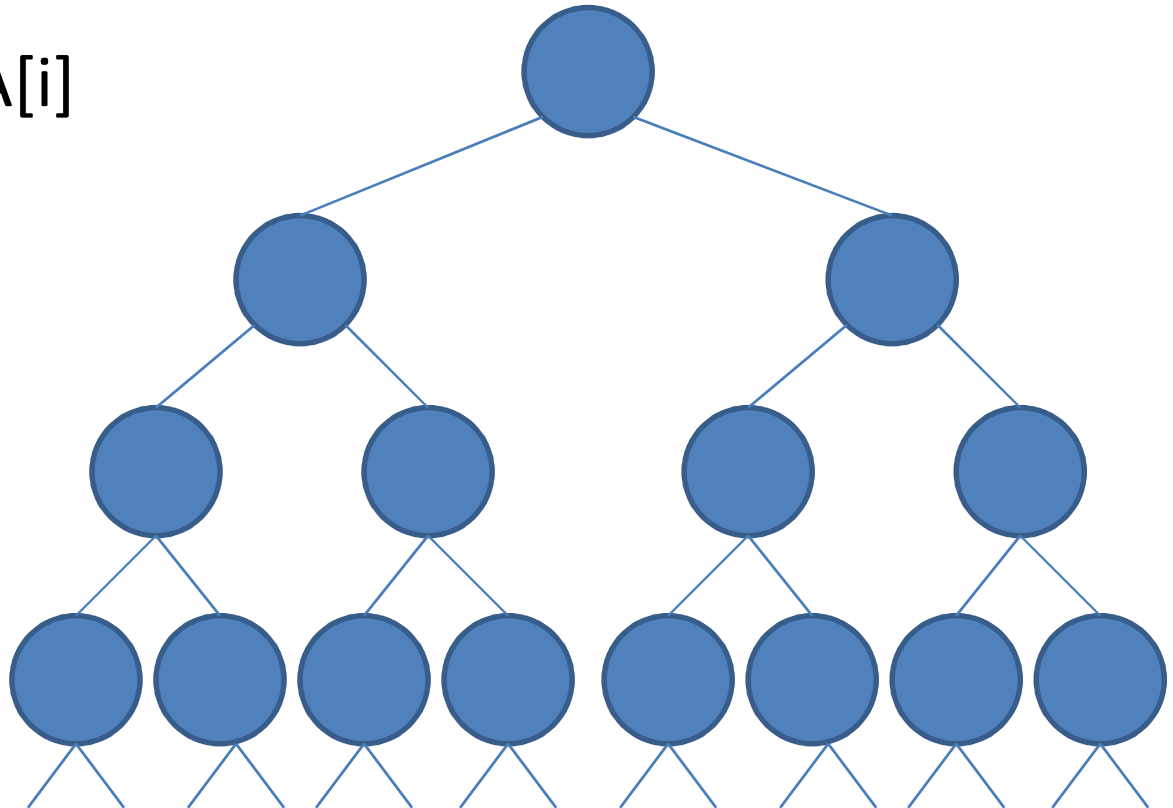
Binomial Tree

- B_0 : single node: *Root*
- B_k : *Root* with k binomial subtrees, $B_0 \dots B_{k-1}$



Prefix Sums

- $P[0] = A[0]$
- For $i = 1$ to $n-1$
 - $P[i] = P[i-1] + A[i]$



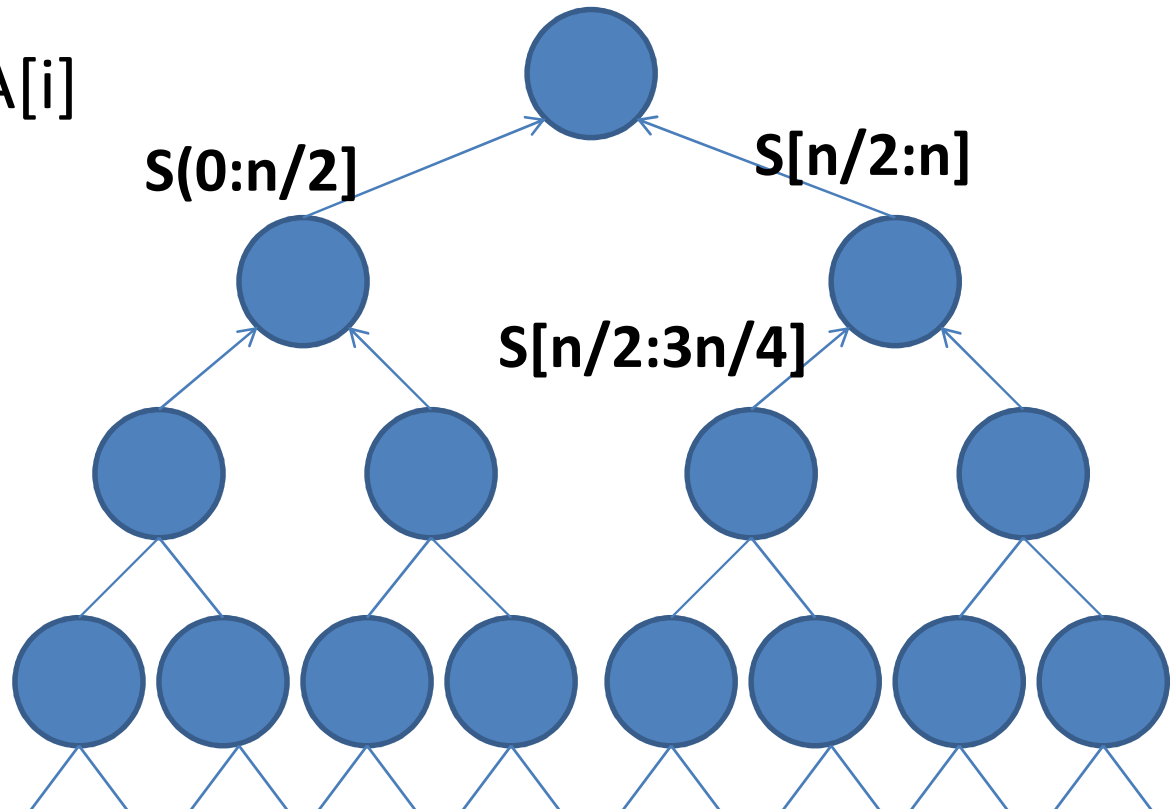
Recursive Prefix Sums

```
prefixSums(s, x, 0:n)
{
  parallel for i in 0:n/2
    y[i] = Op(x[2*i], x[2*i+1])
  prefixSum(z, y, 0:n/2)
  s[0] = x[0]
  parallel for i in 1:n
    if(i&1) s[i] = z[i/2]
    else   s[i] = op(z[i/2-1 ], x[i])
}
```

Or $op^{-1}(z[i/2], x[i])$ if op invertible

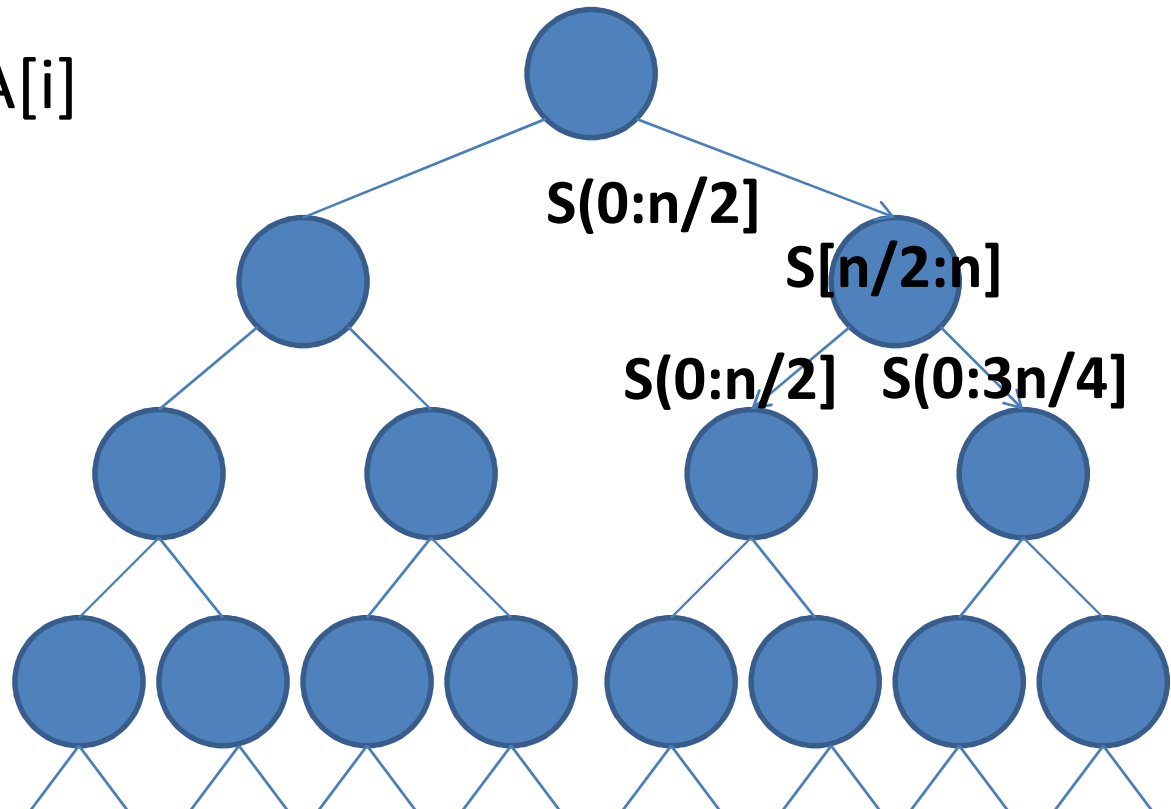
Prefix Sums

- $P[0] = A[0]$
- For $i = 1$ to $n-1$
 - $P[i] = P[i-1] + A[i]$



Prefix Sums

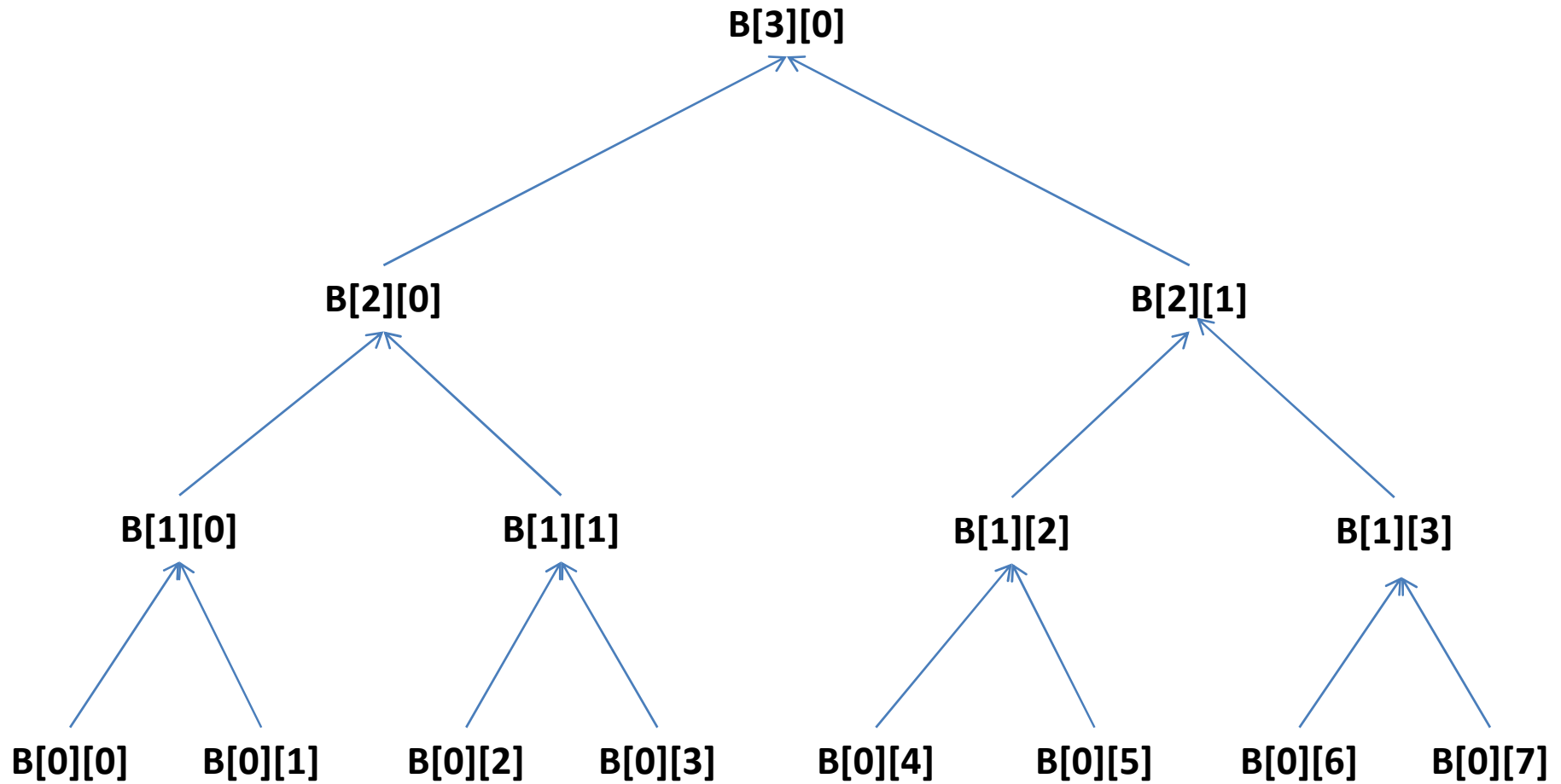
- $P[0] = A[0]$
- For $i = 1$ to $n-1$
 - $P[i] = P[i-1] + A[i]$



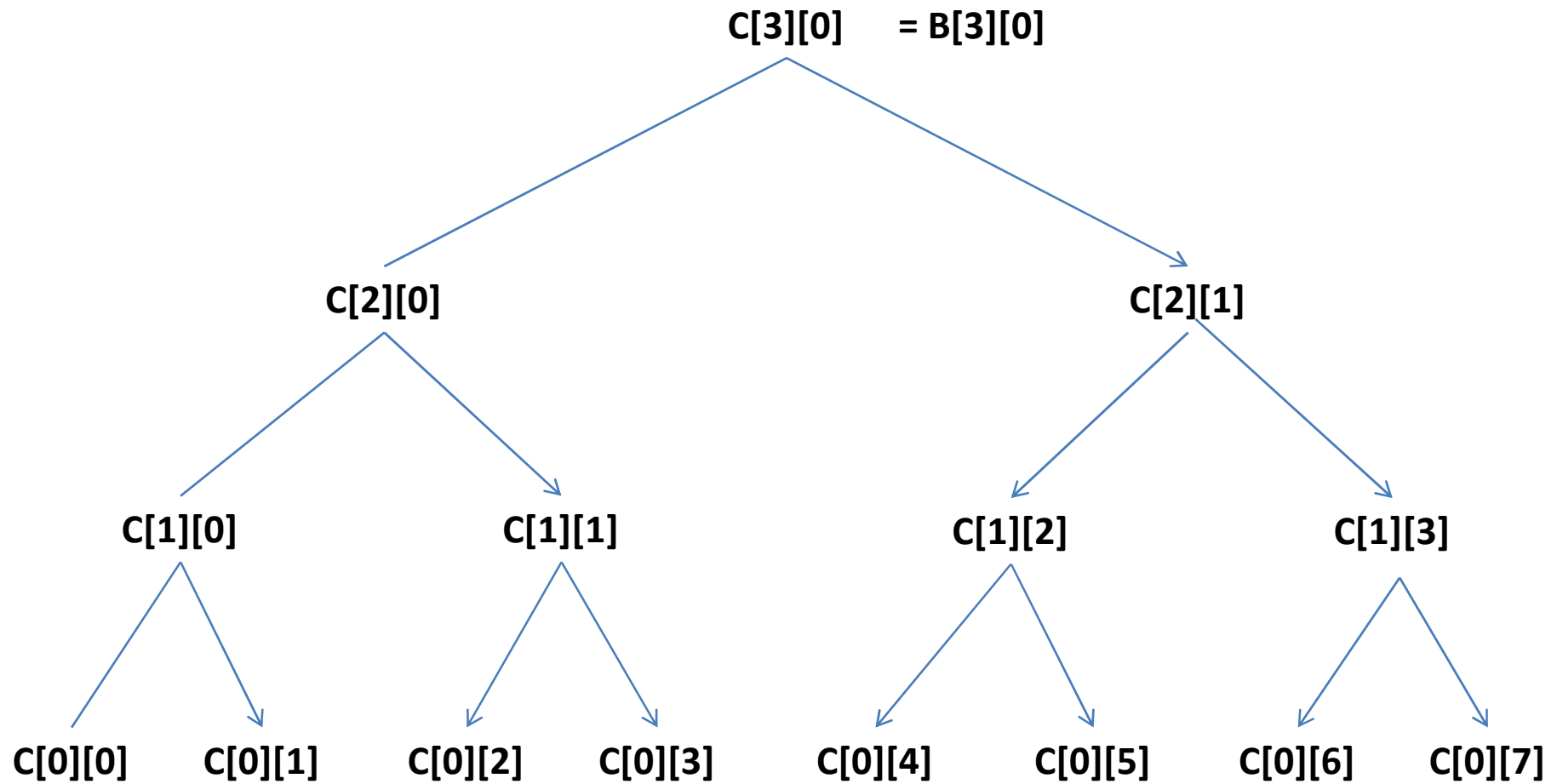
Non-recursive Prefix Sums

- parallel for i in $0:n$
 - $B[0][i] = A[i]$
- for h in $0:\log n$
 - parallel for i in $0:n/2^h$
 - $B[h][i] = B[h-1][2i] \text{ op } B[h-1][2i+1]$
- for h in $\log n:0$
 - $C[h][0] = B[h][0]$
 - parallel for i in $1:n/2^h$
 - Odd: $C[h][i] = C[h+1][i/2]$
 - Even: $C[h][i] = C[h+1][(i/2-1)] \text{ op } B[h][i]$

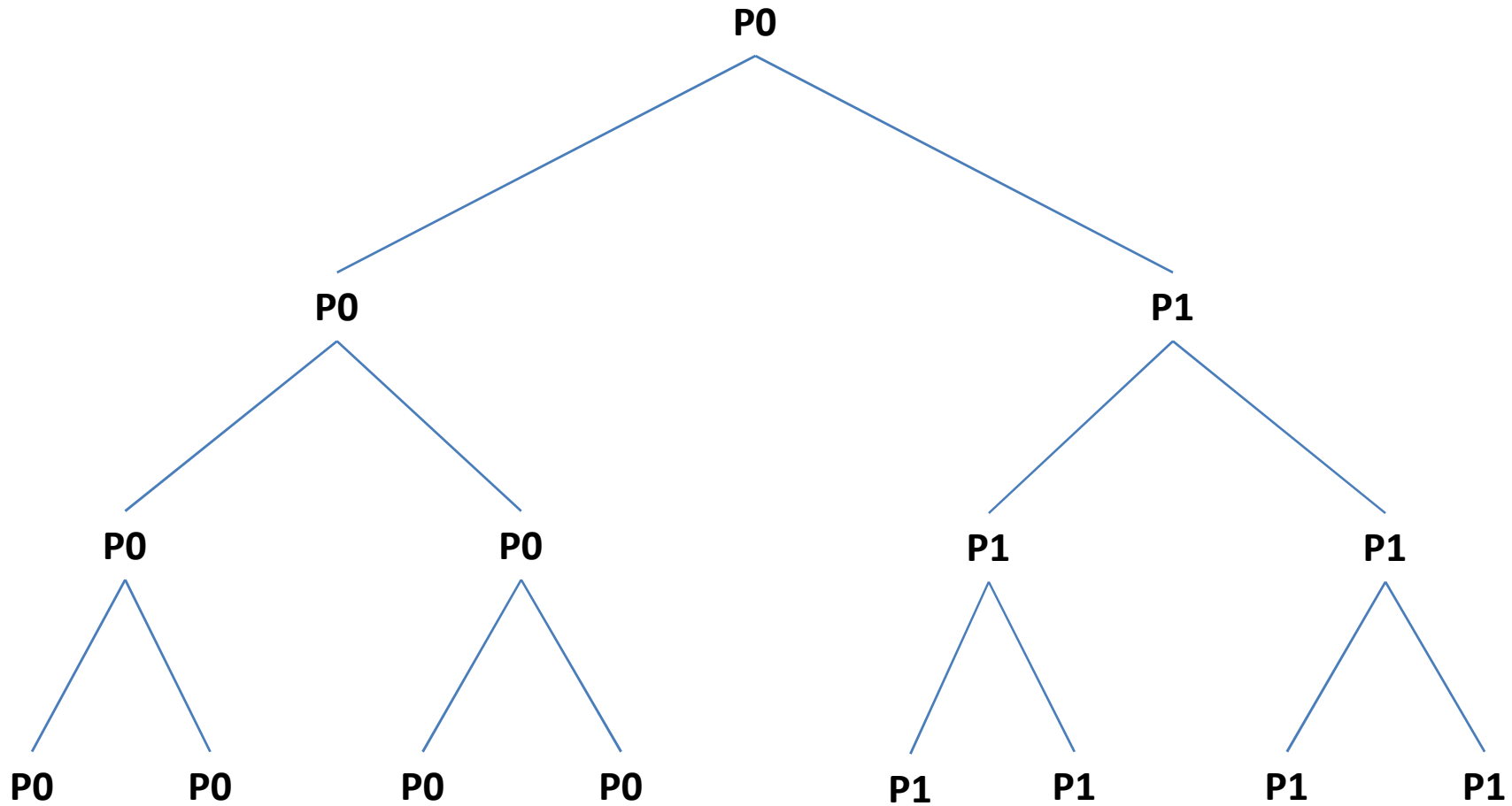
Prefix Sums: Data flow up



Prefix Sums: Data flow down



Processor Mapping



Balanced Tree Approach

- Build binary tree on the input
 - Hierarchically divide into groups
 - and groups of groups..
- Traverse tree upwards/downwards
- Useful to think of “tree” network topology
 - Only for algorithm design
 - Later map sub-trees to processors

PARALLEL ALGORITHM TECHNIQUES: PARTITIONING

Merge Sorted Sequences (A,B)

- Determine Rank of each element in $A \cup B$
- $\text{Rank}(x, A \cup B) = \text{Rank}(x, A) + \text{Rank}(x, B)$
 - Only need one of them, if A and B are each sorted
- Find $\text{Rank}(A, B)$, and similarly $\text{Rank}(B, A)$
- Find Rank by binary search
- $O(\log n)$ time
- $O(n \log n)$ work

Optimal Merge (A,B)

- Partition A and B into 'log n' sized blocks
- Choose from B, elements $i * \log n, i = 0:n/\log n$
- Rank each chosen element of B in A
 - Binary search
- Merge pairs of sub-sequences
 - If $|A_i| = \log(n)$, Sequential merge in time $O(\log(n))$
 - Otherwise, partition A_i into log n blocks
 - And Recursively subdivide B_i into sub-sub-sequences
- Total time is $O(\log(n))$
- Total work is $O(n)$

Optimal Merge (A,B)

- Partition A and B into \sqrt{n} blocks
- Choose from B, elements i (\sqrt{n}), $i=(0: \sqrt{n}]$
- Rank each chosen element of B in A
 - Parallel search using \sqrt{n} processors each search
- Recursively merge pairs of sub-sequences
 - Total time: $T(n) = O(1)+T(n/2) = O(\log \log n)$
 - Total work: $W(n) = O(n)+T(n/2) = O(n \log \log n)$
- “Fast” but still need to reduce work

Optimal Merge (A,B)

- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size $\log \log n$
 - A_1, A_2, \dots
 - B_1, B_2, \dots
- Select first element of each block
 - $A' = p_1, p_2, \dots$
 - $B' = q_1, q_2, \dots$
- Now merge $\log \log n$ sized blocks $n / \log \log n$ times

Optimal Merge (A,B)

- Merge A' and B' – find $\text{Rank}(A':B')$, $\text{Rank}(B':A')$
 - using fast non-optimal algorithm
 - Time = $O(\log \log n)$
 - Work = $O(n)$
- Compute $\text{Rank}(A':B)$ and $\text{Rank}(B':A)$
 - If $\text{Rank}(p_i, B)$ is r_i , p_i lies in block B_{r_i}
 - Search sequentially
 - Time = $O(\log \log n)$
 - Work = $O(n)$
- Compute ranks of remaining elements
 - Sequentially
 - Time = $O(\log \log n)$
 - Work = $O(n)$

Quick Sort

- Choose the pivot
 - Select median?
- Subdivide into two groups
 - Group sizes linearly related with high probability
- Sort each group independently

QuickSort Algorithm

```
QuickSort(int A[], int first, int last)
{
    Select random  $m$  in [first:last] //  $A[m]$  is pivot
    parallel for  $i$  in [first:last]
        flag[i] =  $A[i] < A[m]$ ;
    Split(A); // Separate flag values 0 and 1,  $A[m]$  goes to  $k$ 
        // Use Prefix Sum
    Quicksort A[first:k-1] and A[k+1:last]
}
```

Quick Sort

- Choose the pivot
 - Select median?
- Subdivide into two groups
 - Group sizes linearly related with high probability
- Sort each group independently
- Expected $O(\log n)$ rounds
- Time per round = $O(\log n)$
- Total work = $O(n \log n)$ with high probability

Partitioning Approach

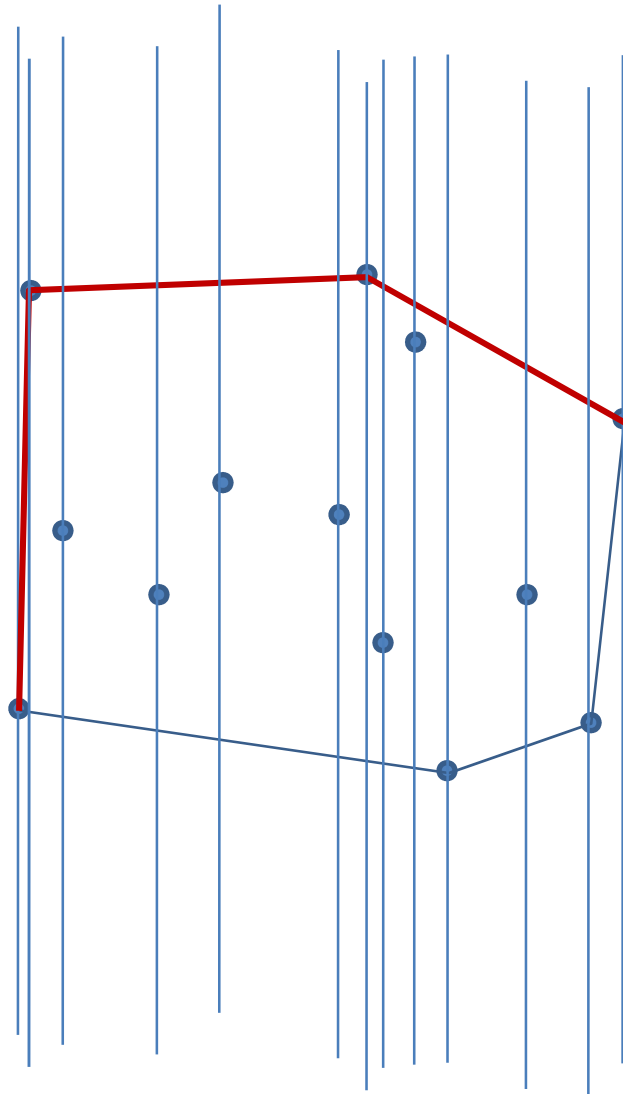
- Break into p roughly equal sized problems
- Solve each sub-problem
 - Preferably, independently of each other
- Focus on subdividing into independent parts

PARALLEL ALGORITHM TECHNIQUES: DIVIDE AND CONQUER

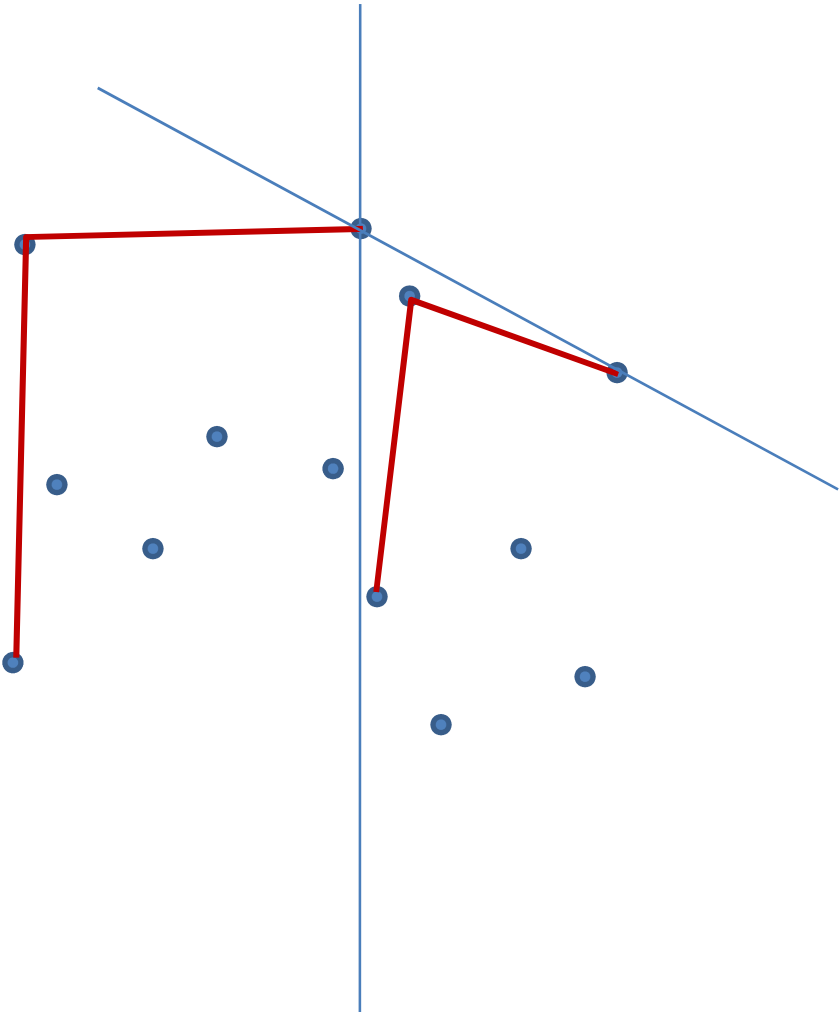
Merge Sort

- Partition data into two halves
 - Assign half the processors to each half
 - If only one processor remains, sequentially sort
- Sort each half
- Merge results
- More on this later

Convex Hull



Convex Hull



**PARALLEL ALGORITHM TECHNIQUES:
ACCELERATED CASCADING**

Min-find

Input: array with n numbers

Algorithm A1 using $O(n^2)$ processors:

parallel for i in $(0:n]$

$M[i] := 0$

parallel for i, j in $0:n$

 if $i \neq j \ \&\& \ C[i] < C[j]$

$M[j] = 1$

parallel for i in $0:n$

 if $M[i] = 0$

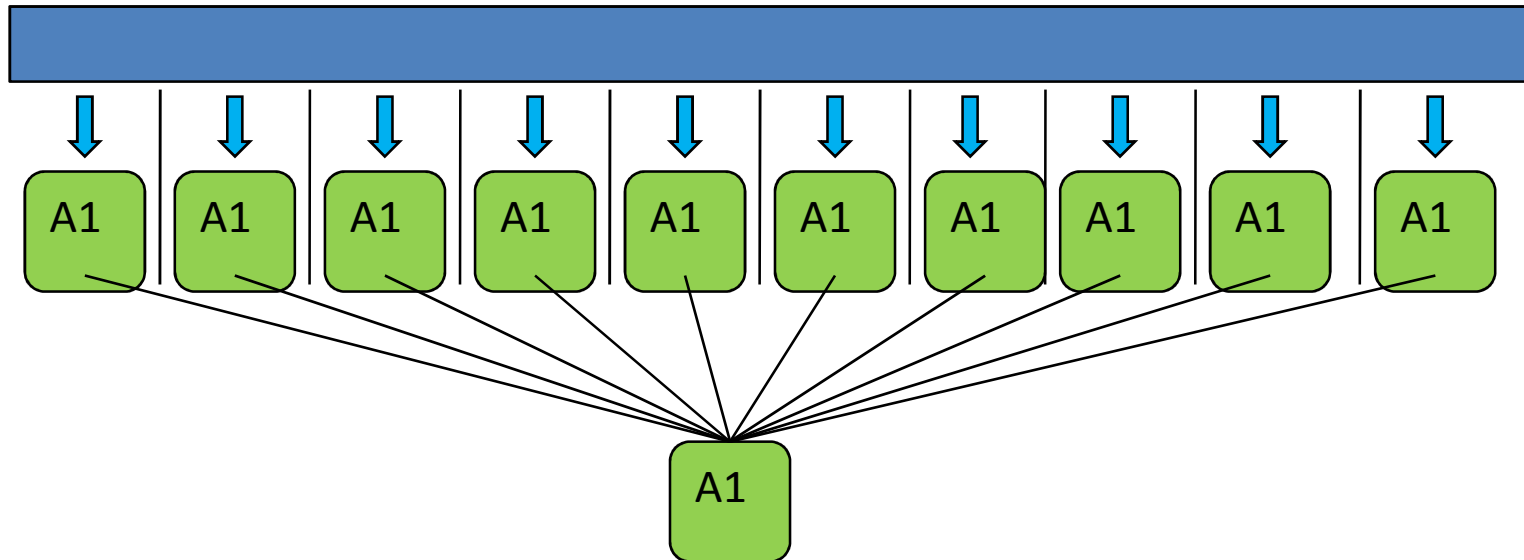
$\text{min} = A[i]$

Not optimal: $O(n^2)$ work)

Optimal Min-find

- Balanced Binary tree
 - $O(\log n)$ time
 - $O(n)$ work => **Optimal**
- Use Accelerated cascading
- Make the tree branch much faster
 - Number of children of node $u = \sqrt{n_u}$
 - Where n_u is the number of leaves in u 's subtree
 - Works if the operation at each node can be performed in $O(1)$

From n^2 processors to $n\sqrt{n}$



Step 1: Partition into disjoint blocks of size \sqrt{n}

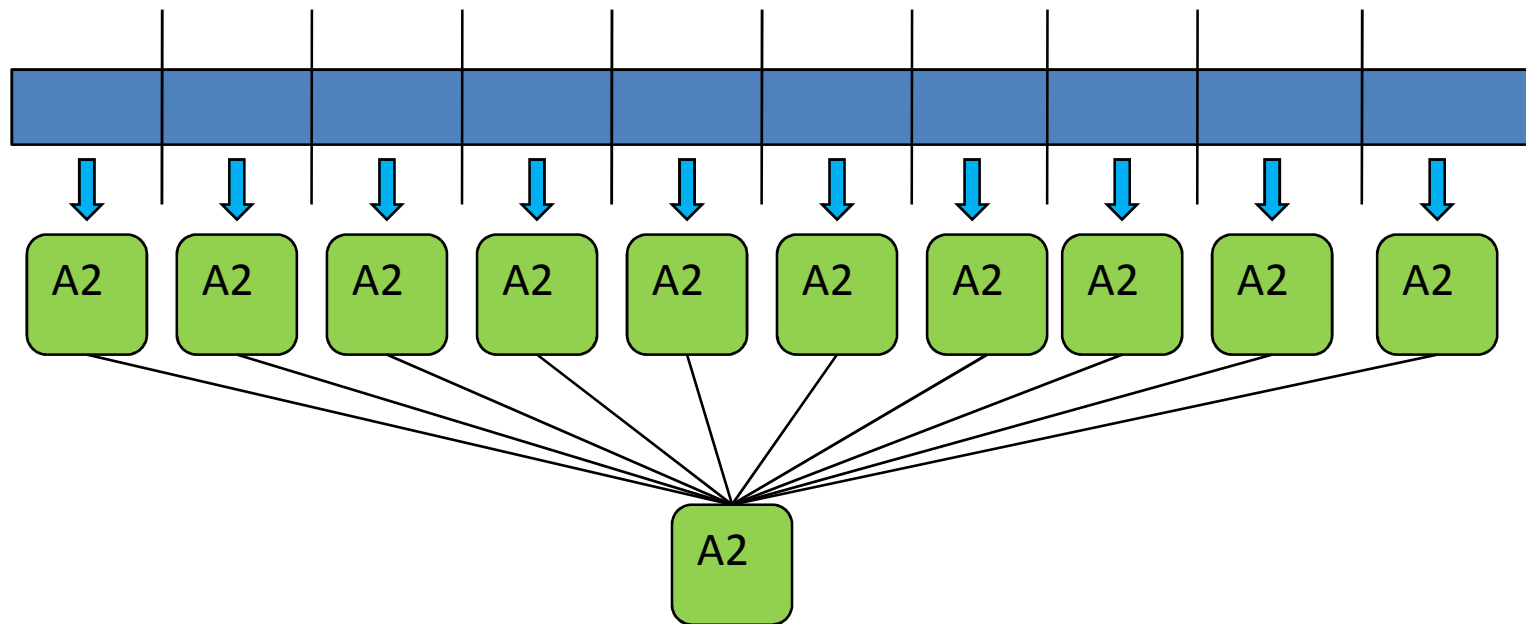
Step 2: Apply A1 to each block

$$n\sqrt{n}$$

Step 3: Apply A1 to the results from the step 2

$$n$$

From $n\sqrt{n}$ processors to $n^{1+1/4}$



Step 1: Partition into disjoint blocks of size \sqrt{n}

Step 2: Apply A2 to each block

Step 3: Apply A2 to the results from the step 2

$$n^{1/2} \quad n^{3/4}$$

$$n^{3/4}$$

$n^2 \rightarrow n^{1+1/2} \rightarrow n^{1+1/4} \rightarrow n^{1+1/8} \rightarrow n^{1+1/16} \rightarrow \dots \rightarrow n^{1+1/k} \sim n^1 ?$

- Algorithm A_k takes “O(1) time” with $n^{1+\epsilon_k}$ processors

Algorithm A_{k+1}

1. Partition input array C (size n) into disjoint blocks of size $n^{1/2}$ each
2. Solve for each block in parallel using algorithm A_k
3. Re-apply A_k to the results of step 3: $n / n^{1/2}$ minima

**Doubly logarithmic-depth tree
n log log n work, log log n time**

Min-Find Review

- Constant-time algorithm
 - $O(n^2)$ work
- $O(\log n)$ Balanced Tree Approach
 - $O(n)$ work **Optimal**
- $O(\log \log n)$ Doubly-log depth tree Approach
 - $O(n \log \log n)$ work
 - Degree is high at the root, reduces going down
 - #Children of node $u = \sqrt{\text{\#nodes in tree rooted at } u}$
 - Depth = $O(\log \log n)$

Accelerated Cascading

- Solve recursively
- Start bottom-up with the optimal algorithm
 - until the problem sizes is smaller
- Switch to fast (non-optimal algorithm)
 - A few small problems solved fast but non-work-optimally
- Min Find:
 - Optimal algorithm for lower $\log \log n$ levels
 - Then switch to $O(n \log \log n)$ -work algorithm

n work, $\log \log n$ time