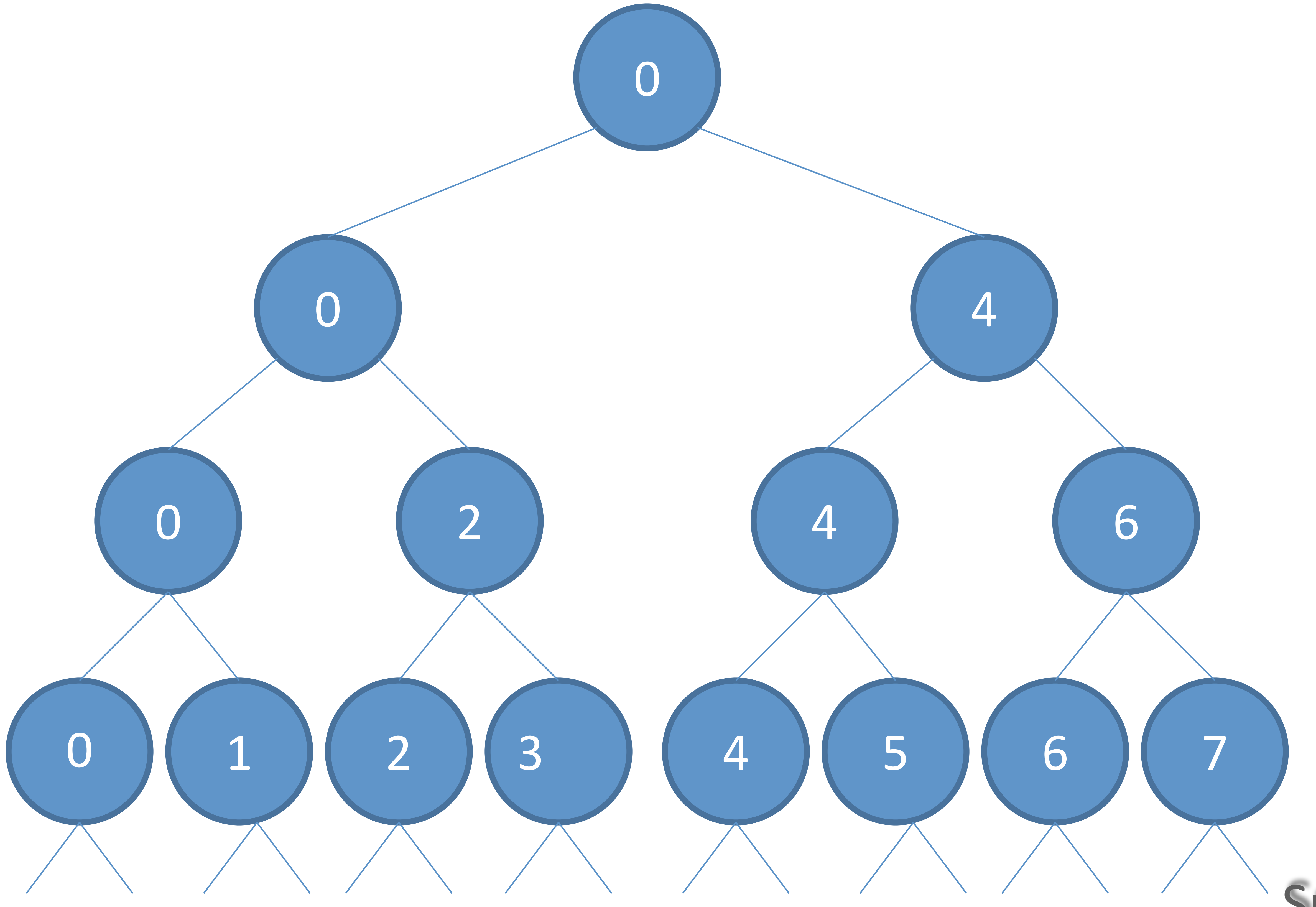
COL380

Introduction to
Parallel & Distributed Programming

Parallel algorithm technique: Balanced binary tree

Reduction

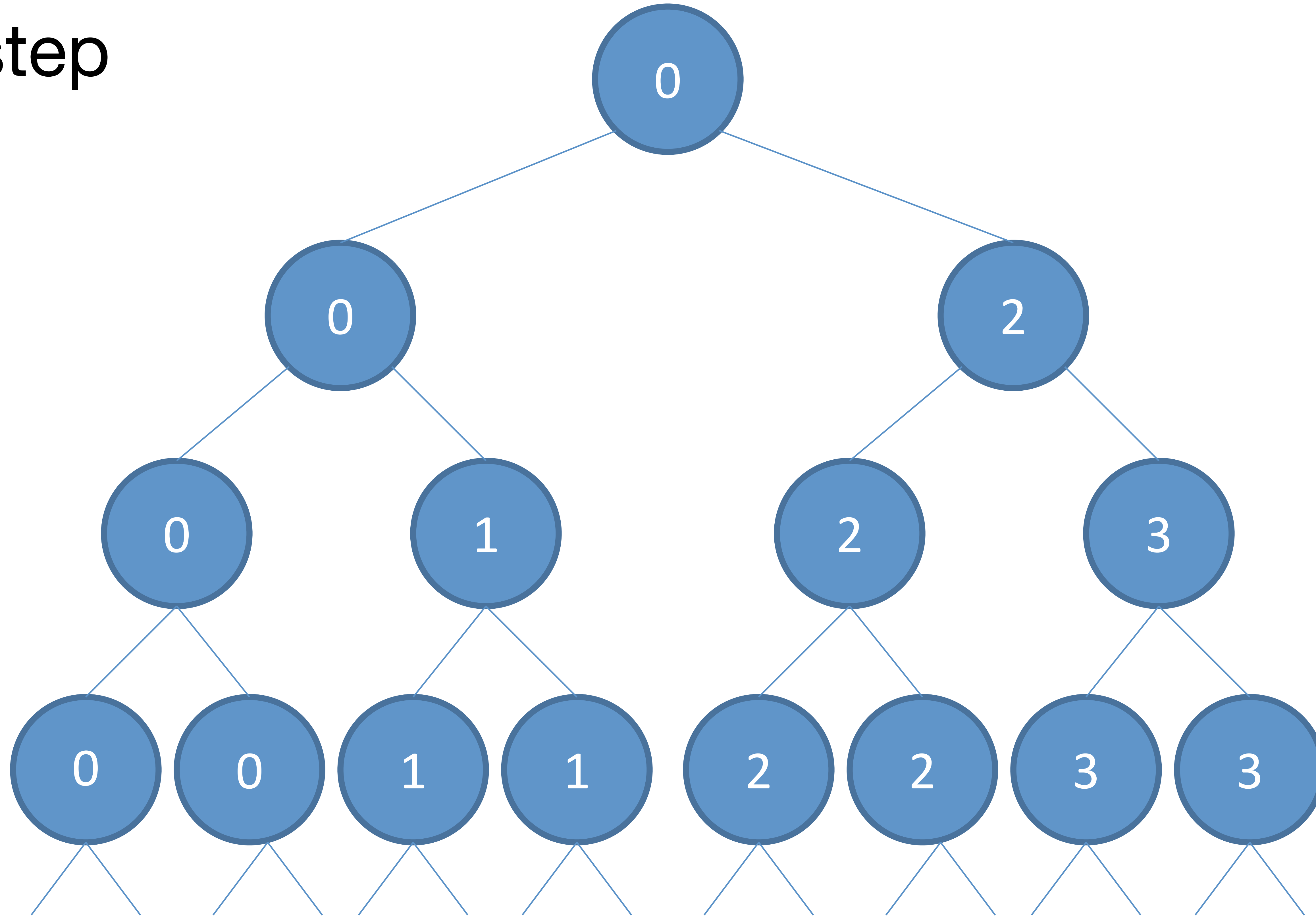
- n operands => log n steps
- How do you map?
  - $n/2^i$ processors per step
  - step i: if $!(id\%2^i)$



Subodh Kumar

Reduction

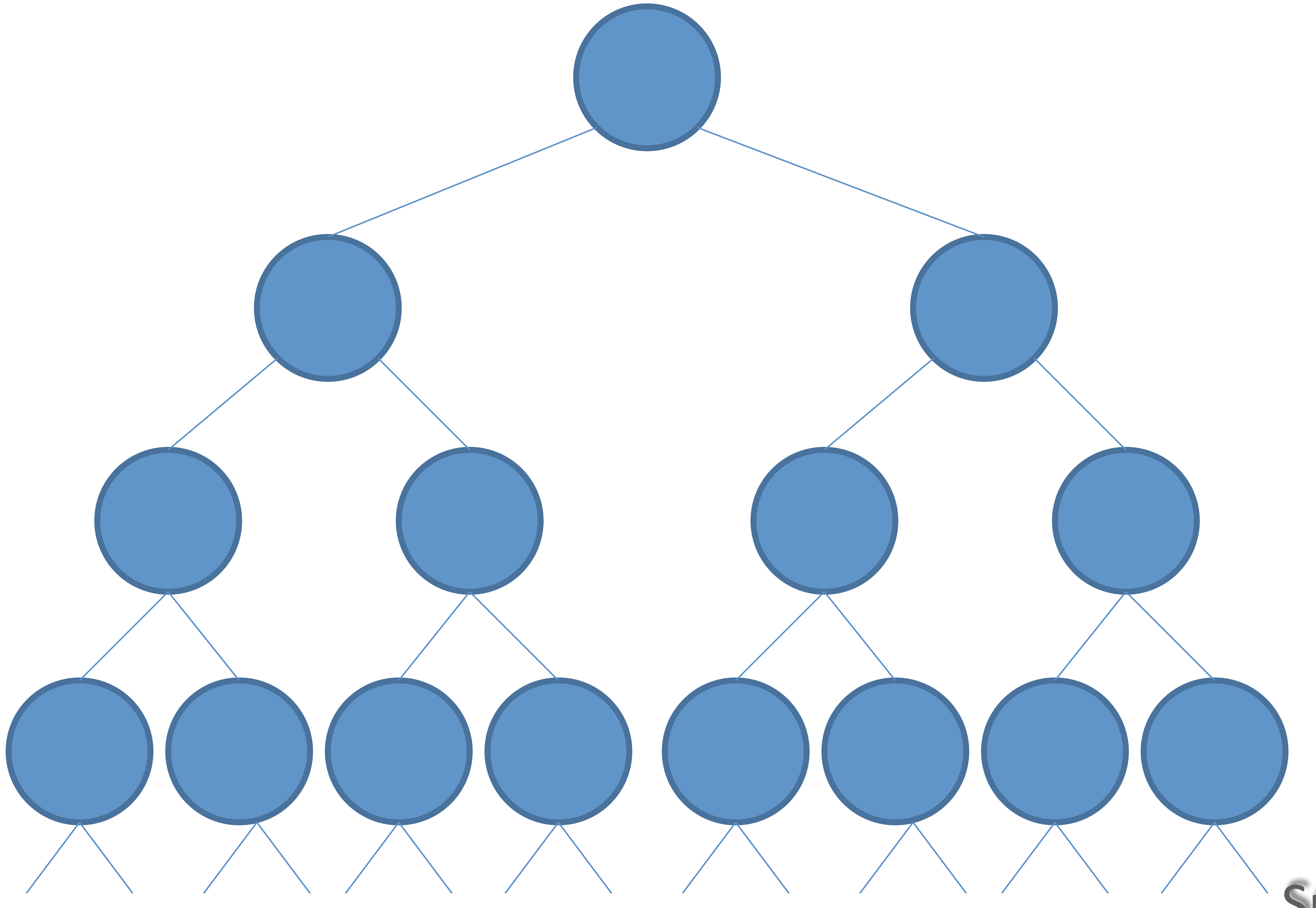
- n operands => log n steps
- Only have p processors per step
- Agglomerate and Map



Subodh Kumar

Prefix Sums

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

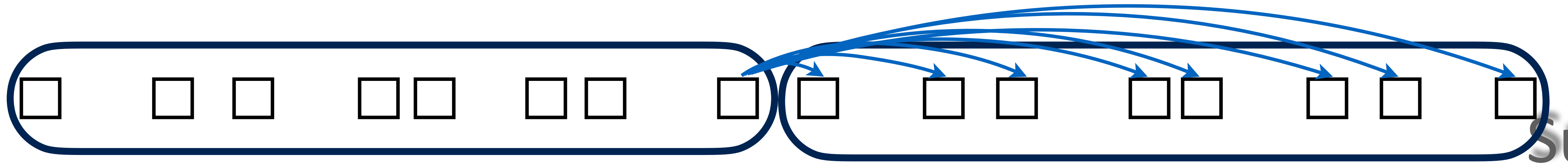


Subodh Kumar

Recursive Prefix Sums

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

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$



Subodh Kumar

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

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

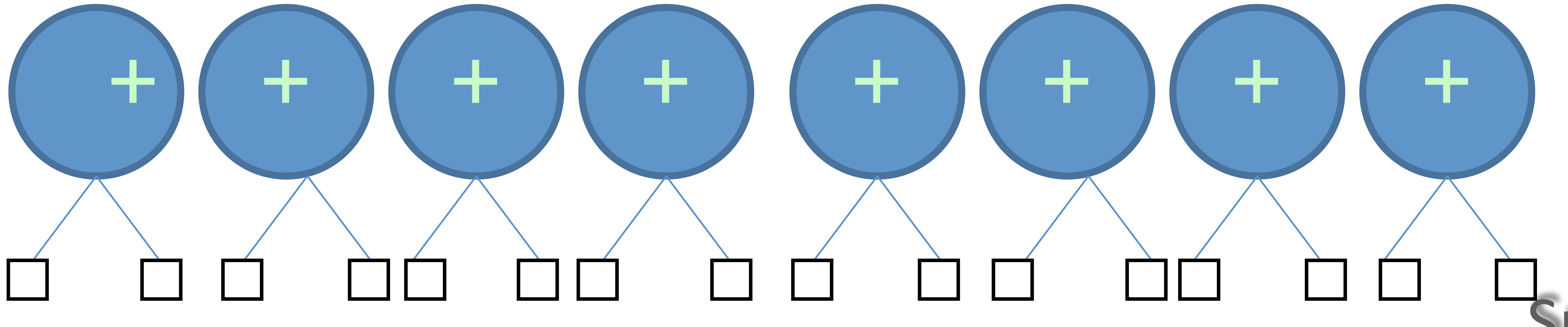
- P[0] = x[0]
- For i = 1 to n-1
  - P[i] = P[i-1] + x[i]

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

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

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



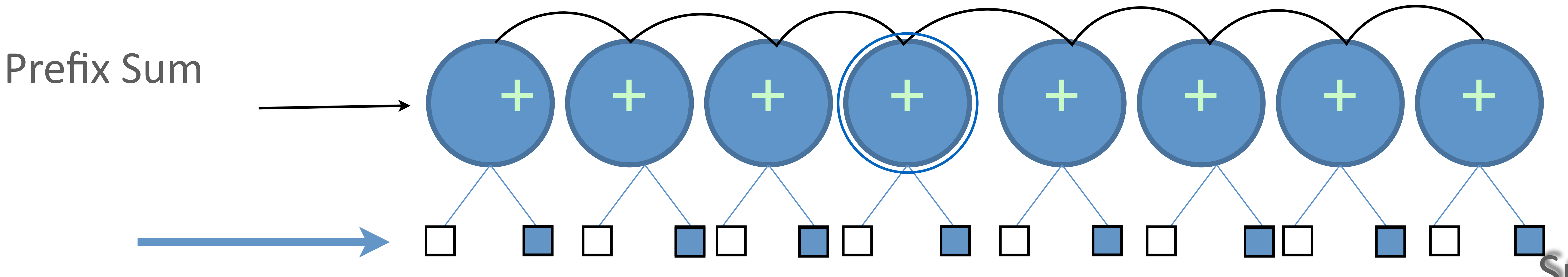
Subodh Kumar

Recursive Prefix Sums

$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

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

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



Subodh Kumar

Recursive Prefix Sums

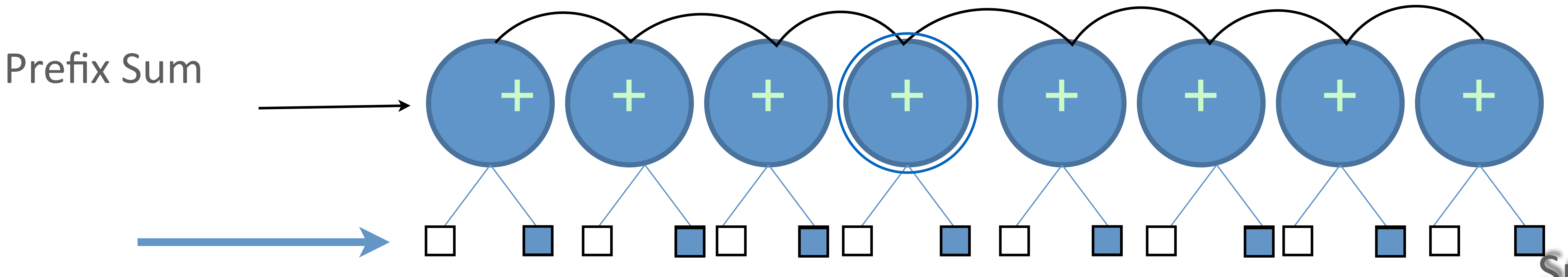
$T(n) = T(n/2) + O(1)$
$W(n) = 2W(n/2)+Kn/2$

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

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

$T(n) = T(n/2) + O(1)$
$W(n) = W(n/2)+Kn/2$

$W(n) = O(n)$



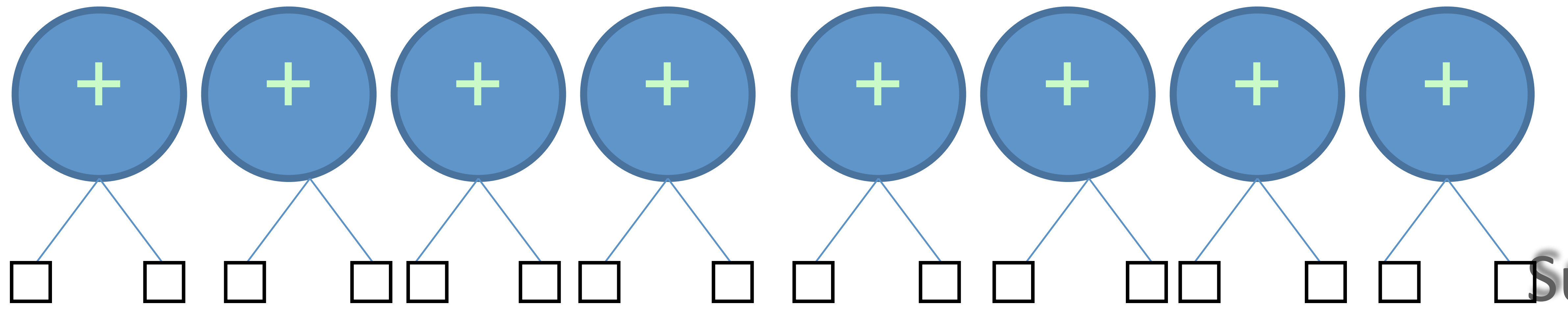
Subodh Kumar

Recursive Prefix Sums

```
prefixSums(P, x, [0:n))
{
    forall i in [0:n/2)
        y[i] =  OP(x[2*i], x[2*i+1])
    prefixSum(z, y, [0:n/2))
    P[0] = x[0]
    forall i in [1:n)
        if(i&1) P[i] = z[i/2]
        else    P[i] = OP(z[i/2-1 ], x[i])
}
```

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

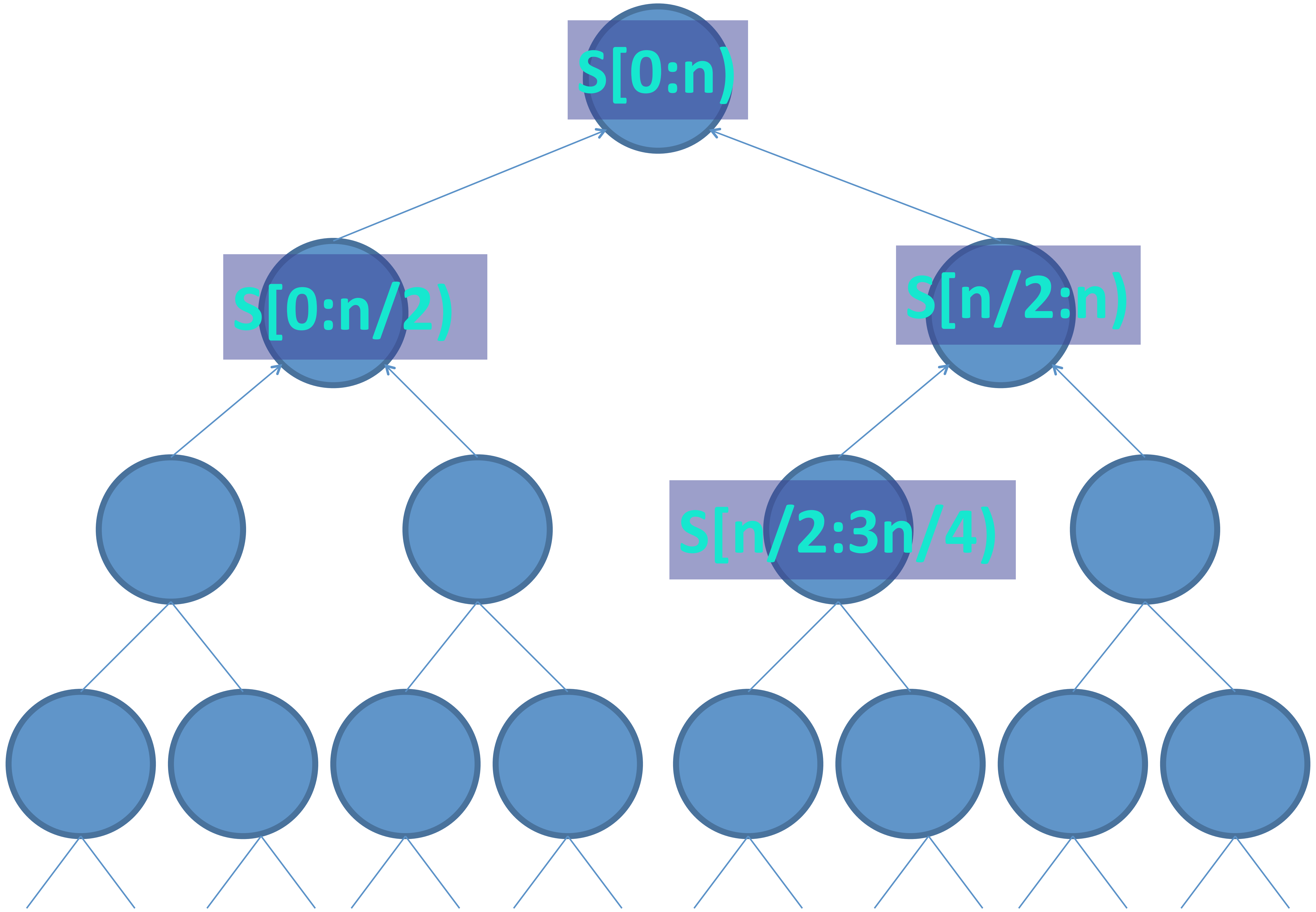
Prefix Sum



Subodh Kumar

Prefix Sums (Binary Tree)

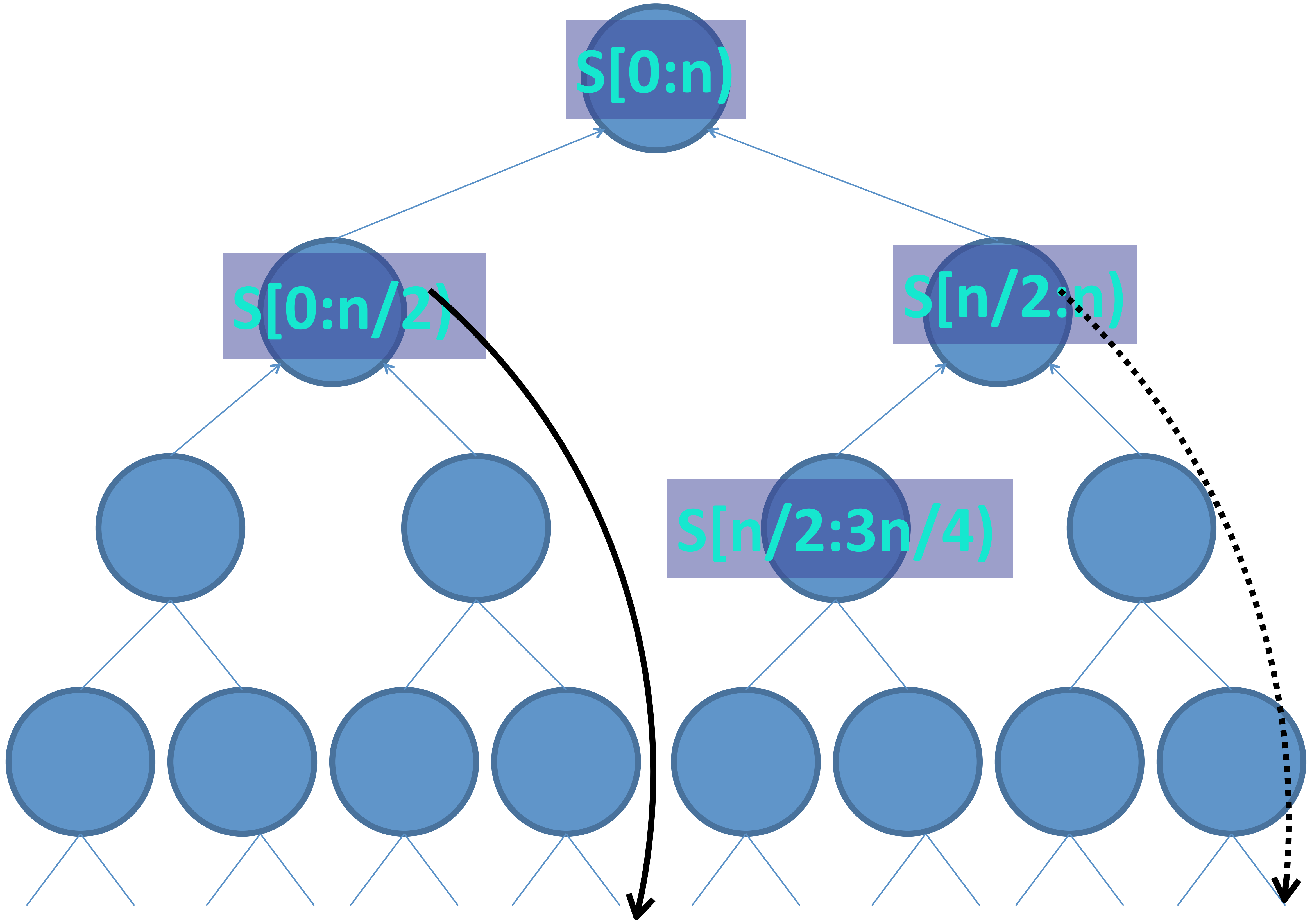
```
P[0] = x[0]
For i = 1 to n-1
  P[i] = P[i-1] + x[i]
```



Subodh Kumar

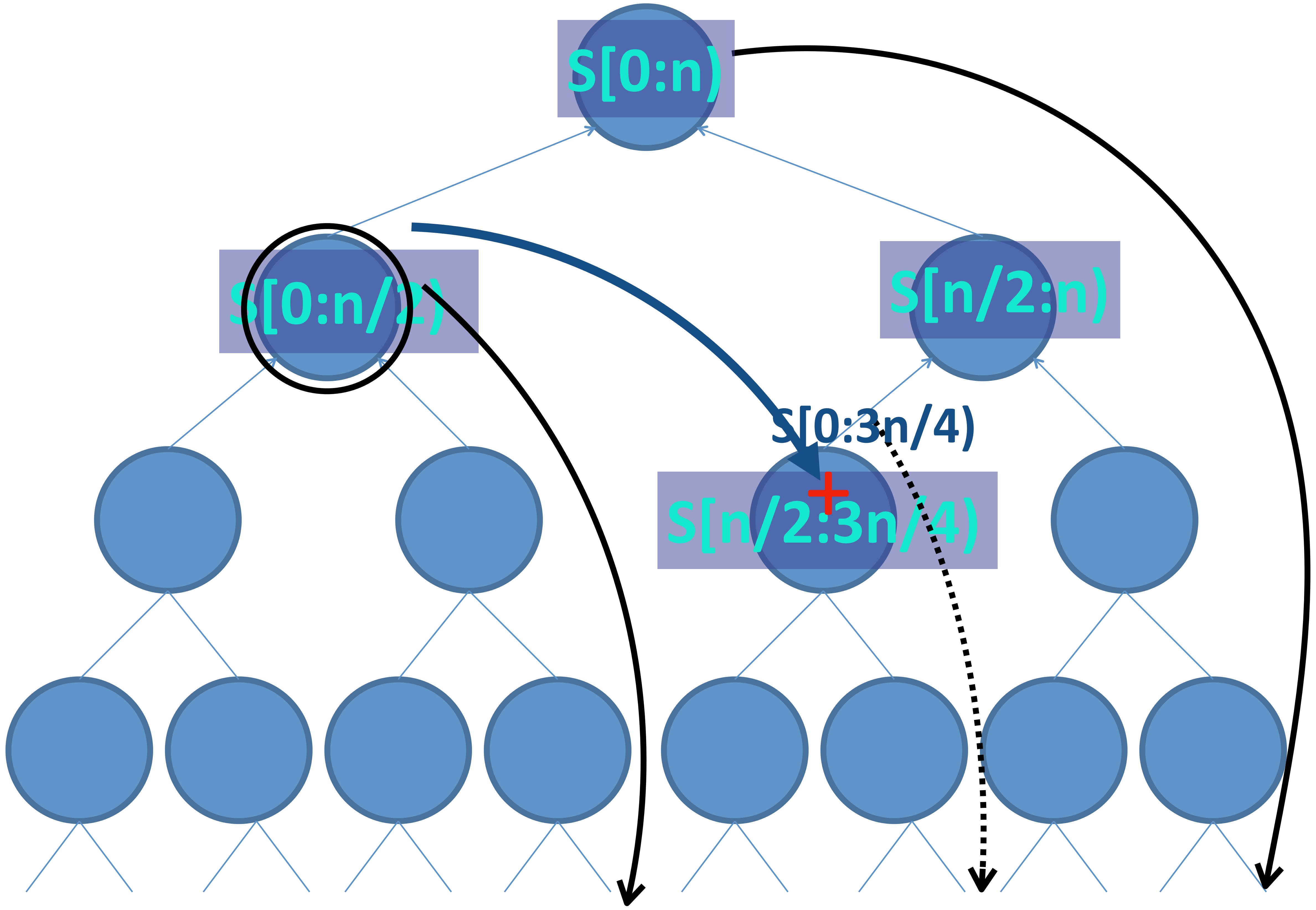
Prefix Sums (Binary Tree)

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



Prefix Sums (Binary Tree)

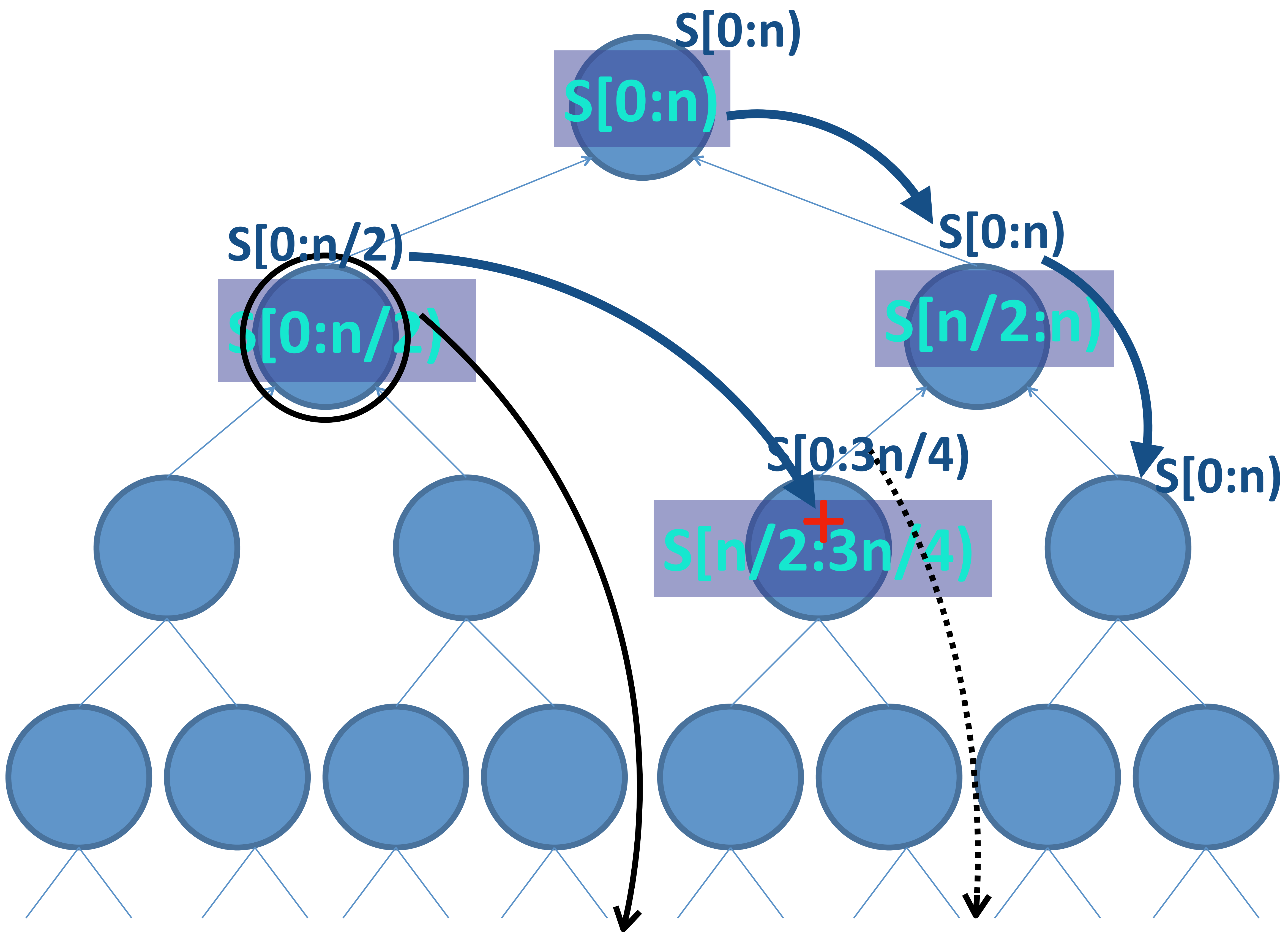
P[0] = x[0]
For i = 1 to n-1
  P[i] = P[i-1] + x[i]



Prefix Sums (Binary Tree)

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

forall i = 0 to n
  B[0][i] = A[i]
for h = 1 to log n
  forall i in 0:n/2^h
    B[h][i] = B[h-1][2i] **OP** B[h-1][2i+1]
for h = log n to 0
  C[h][0] = B[h][0]
  forall i in 1:n/2^h
    Odd i: C[h][i] = C[h+1][i/2]
    Even i: C[h][i] = C[h+1][i/2-1] **OP** B[h][i]



Subodh Kumar

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

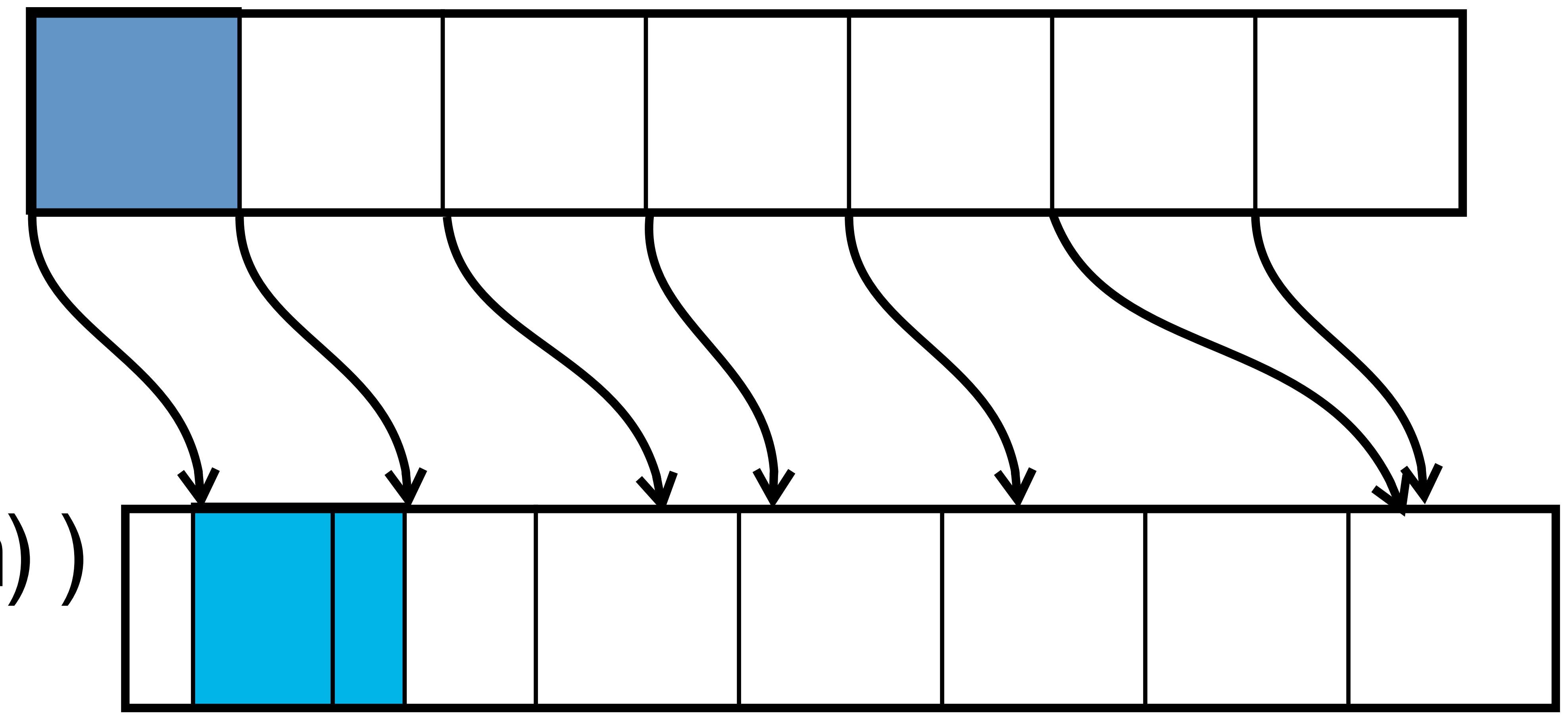
Parallel algorithm techniques: PARTITIONING

Merge Sorted Sequences (A,B)

- Determine Rank of each element in A U B
- Rank(x, A U B) = Rank(x, A) + Rank(x, B)
  - Only need to compute the rank in the other list, if A and B are each sorted already
- Find Rank(A, B), and similarly Rank(B, A)
- Find Rank by binary search
  - O(log n) time
- O(n log n) work

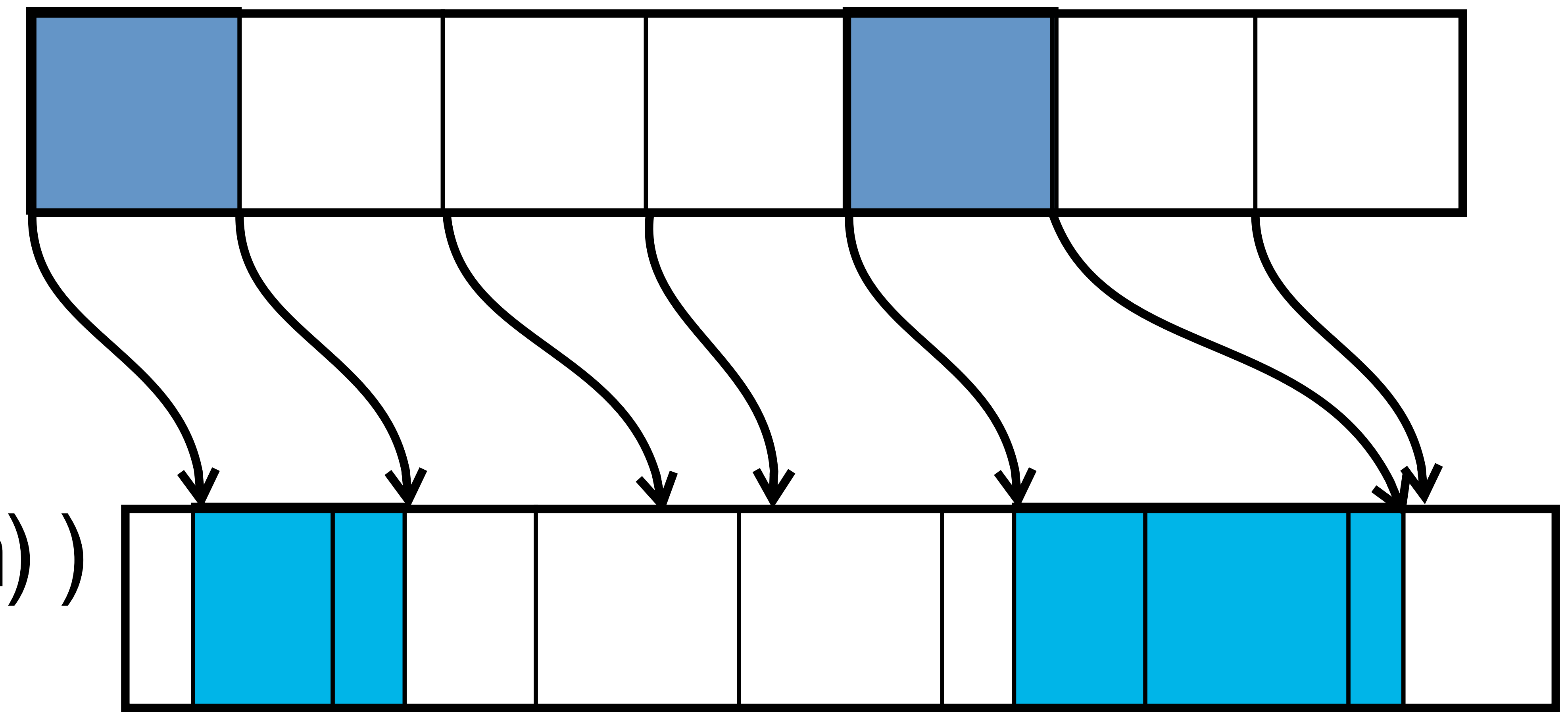
Towards Optimal Merge (A,B)

- Partition A and B into ***log n*** sized blocks
- Select from B, elements i * log n, i ∈ 0:n/log n
- Rank each selected 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



Towards Optimal Merge (A,B)

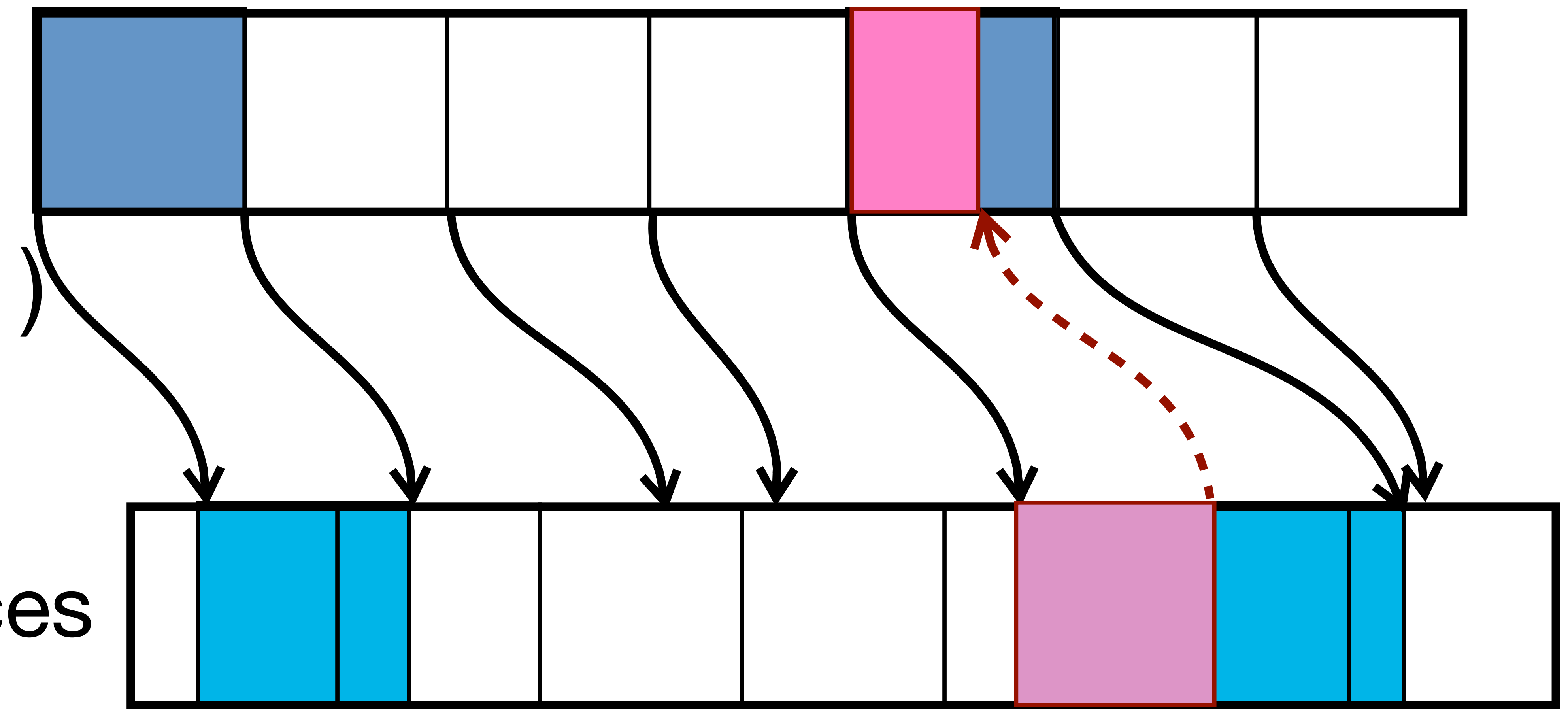
- Partition A and B into ***log n*** sized blocks
- Select from B, elements i * log n, $i \in 0{:}n/\log n$
- Rank each selected element of B in A
  - Binary search
- Merge pairs of sub-sequences
  - If $|A_i| = \log(n)$, Sequential merge in time $O(\log(n))$
  - Otherwise, partition A_i into log n blocks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



Towards Optimal Merge (A,B)

- Partition A and B into ***log n*** sized blocks
- Select from B, elements i * log n, $i \in 0{:}n/\log n$
- Rank each selected 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)



Can we do better?

Subodh Kumar

Fast Merge (A,B)

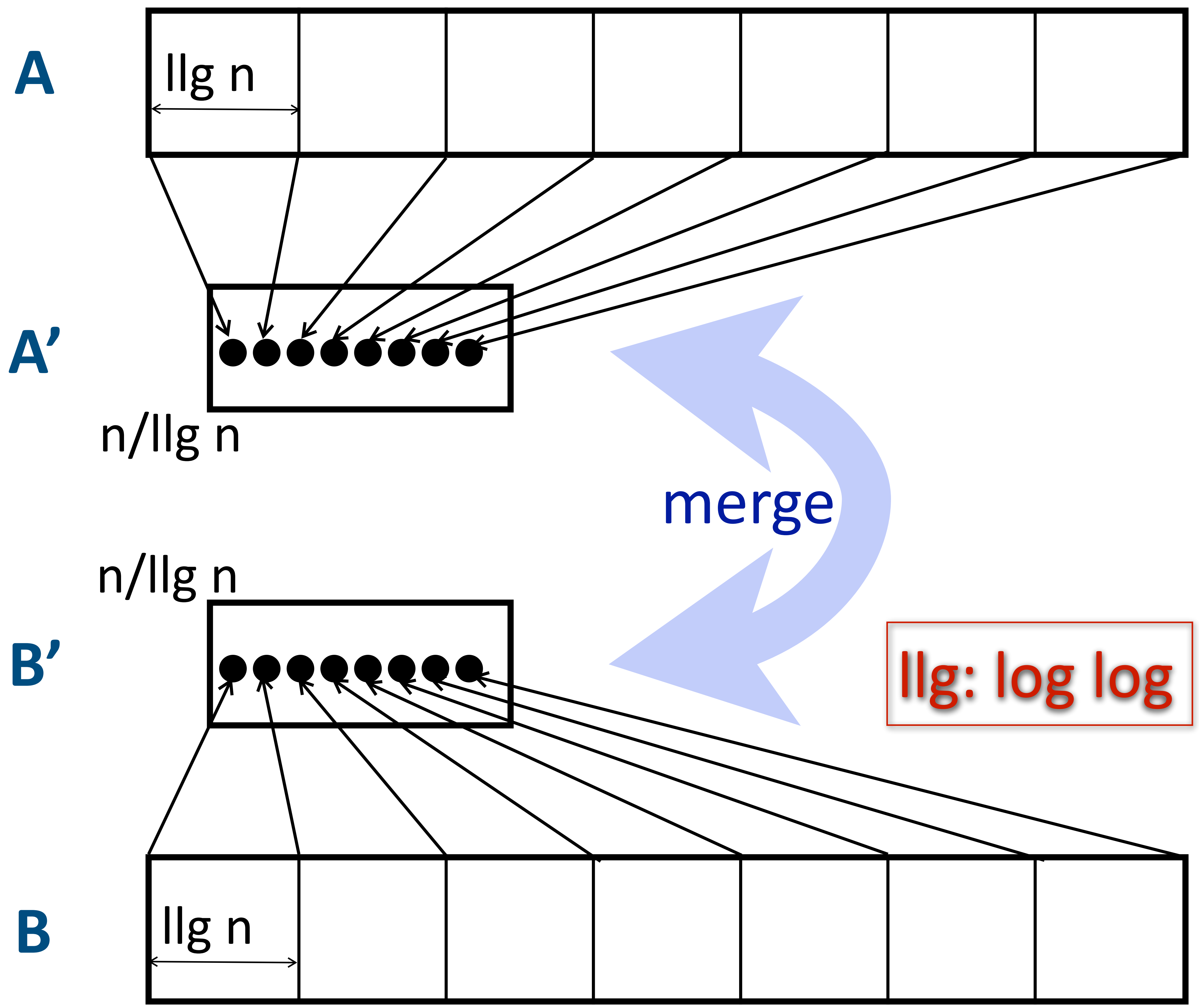
- Partition A and B into $\sqrt{n}$ blocks
- Select from B, elements $i\sqrt{n}$, $i \in (0{:}\ \sqrt{n}]$
- Rank each selected element of B in A
  - Parallel search using $\sqrt{n}$ processors each search
- Similarly rank $\sqrt{n}$ selected elements from A in B
- Recursively merge pairs of sub-sequences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

Not work optimal

Subodh Kumar

Optimal Merge (A,B)

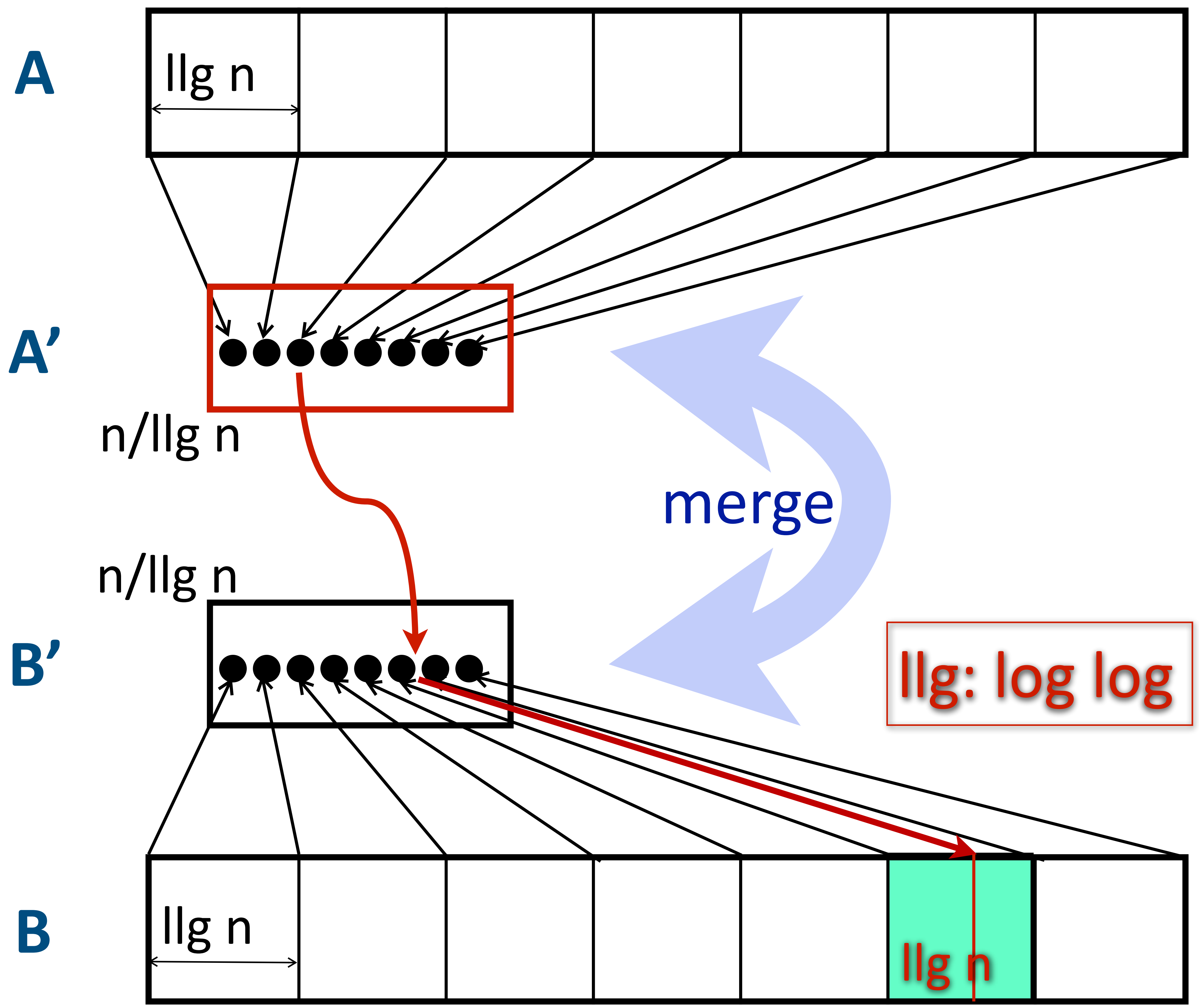
- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size ***log log n***
  - A_1, A_2, ..
  - B_1, B_2, ..
- Select first element of each block
  - A' = p_1, p_2 ..
  - B' = q_1, q_2 ..
- Now merge loglogn sized blocks n/loglogn times



Subodh Kumar

Optimal Merge (A,B)

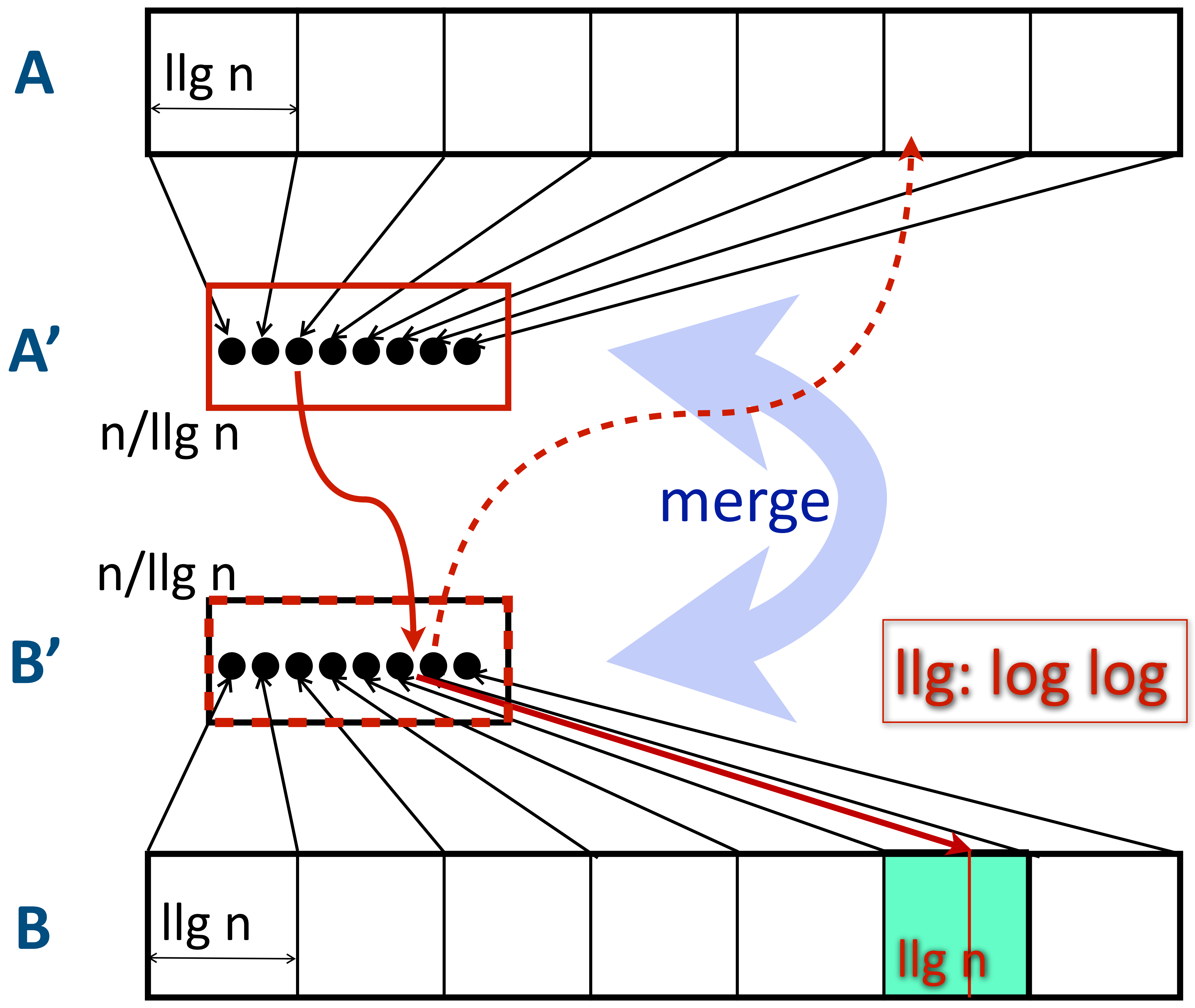
- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size ***log log n***
  - $A_1, A_2, ..$
  - $B_1, B_2, ..$
- Select first element of each block
  - $A' = p_1, p_2 ..$
  - $B' = q_1, q_2 ..$
- Now merge loglogn sized blocks n/loglogn times



Subodh Kumar

Optimal Merge (A,B)

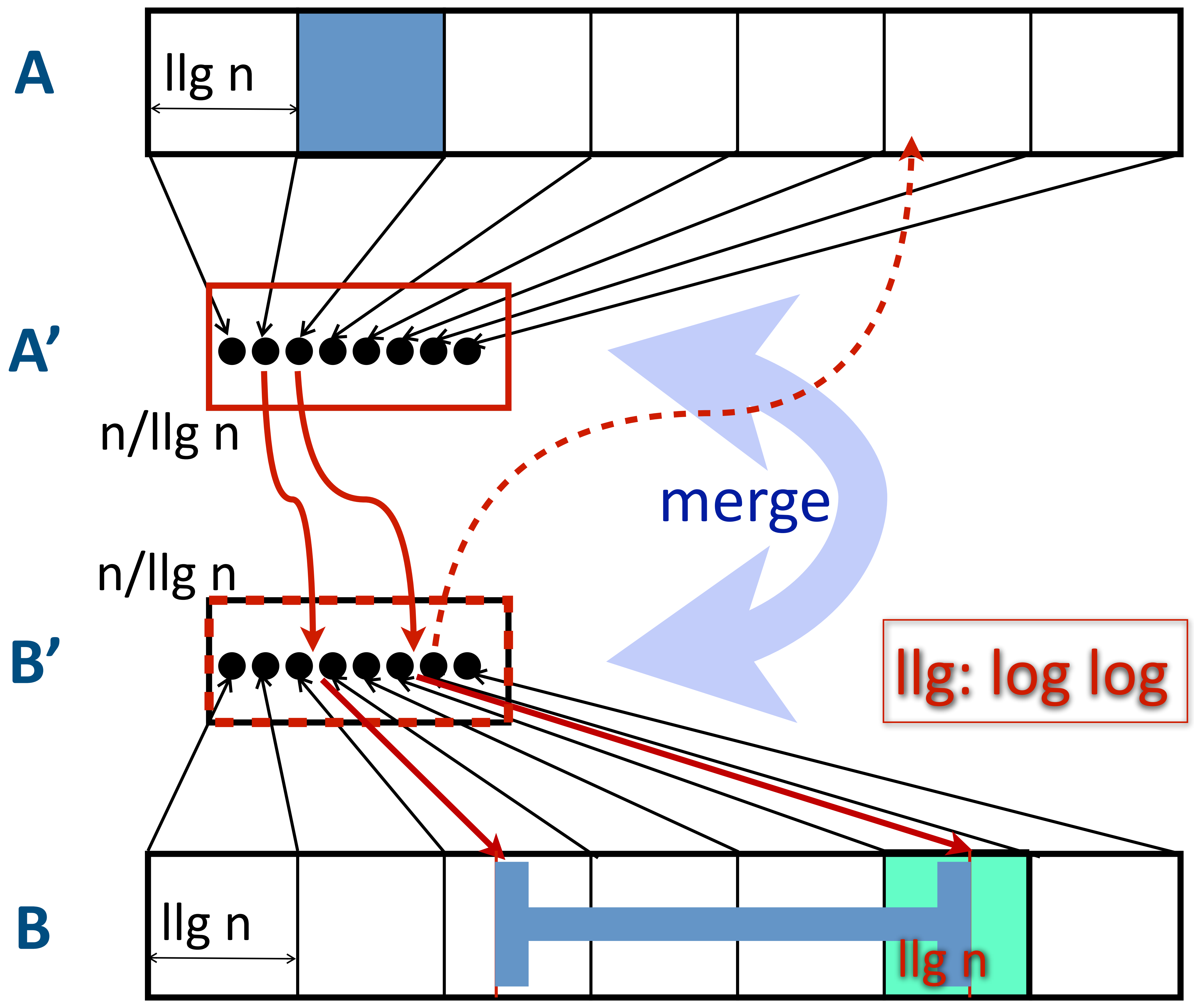
- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size ***log log n***
  - A_1, A_2, ..
  - B_1, B_2, ..
- Select first element of each block
  - A' = p_1, p_2 ..
  - B' = q_1, q_2 ..
- Now merge loglogn sized blocks n/loglogn times



Subodh Kumar

Optimal Merge (A,B)

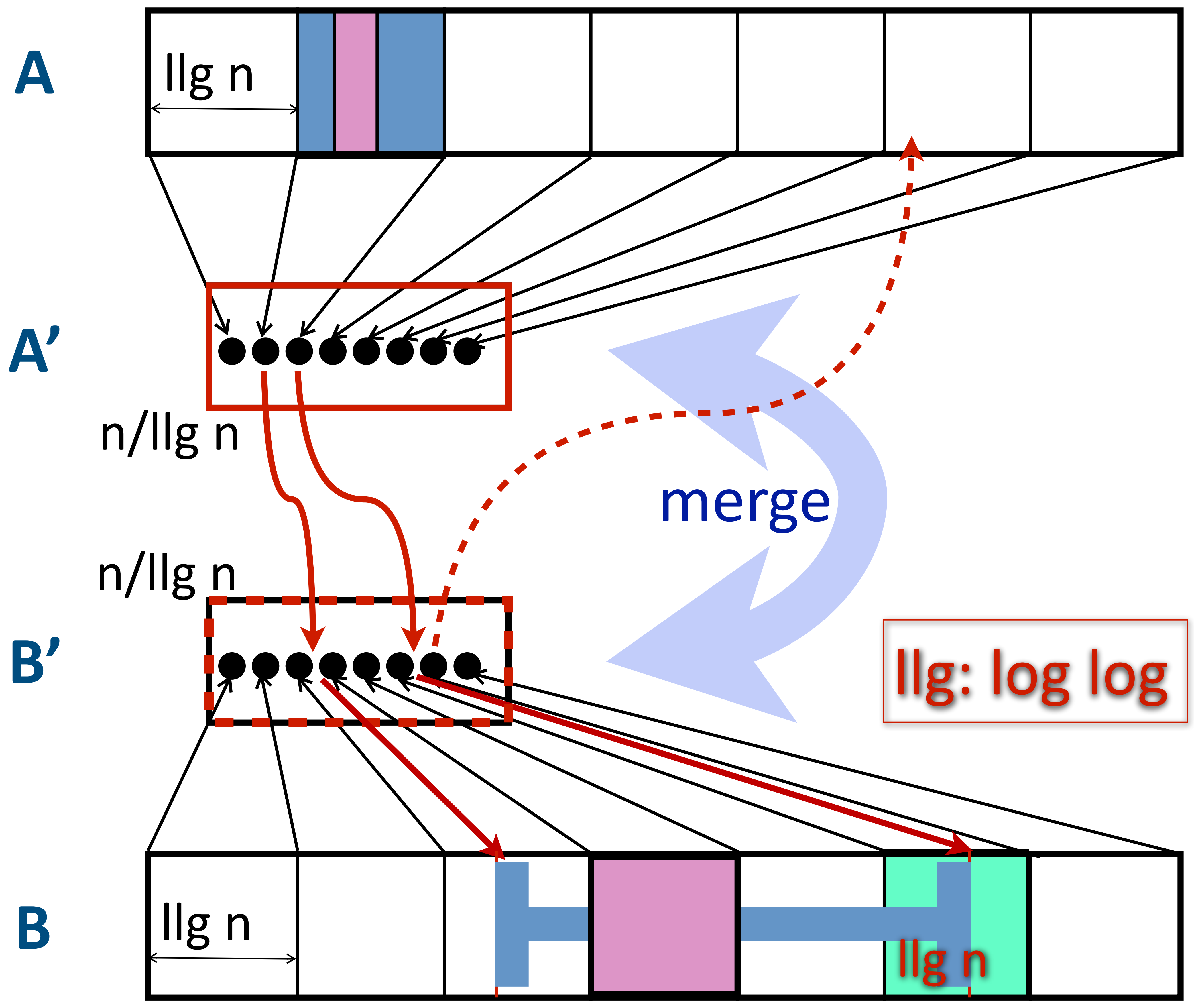
- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size ***log log n***
  - A_1, A_2, ..
  - B_1, B_2, ..
- Select first element of each block
  - A' = p_1, p_2 ..
  - B' = q_1, q_2 ..
- Now merge loglogn sized blocks n/loglogn times



Subodh Kumar

Optimal Merge (A,B)

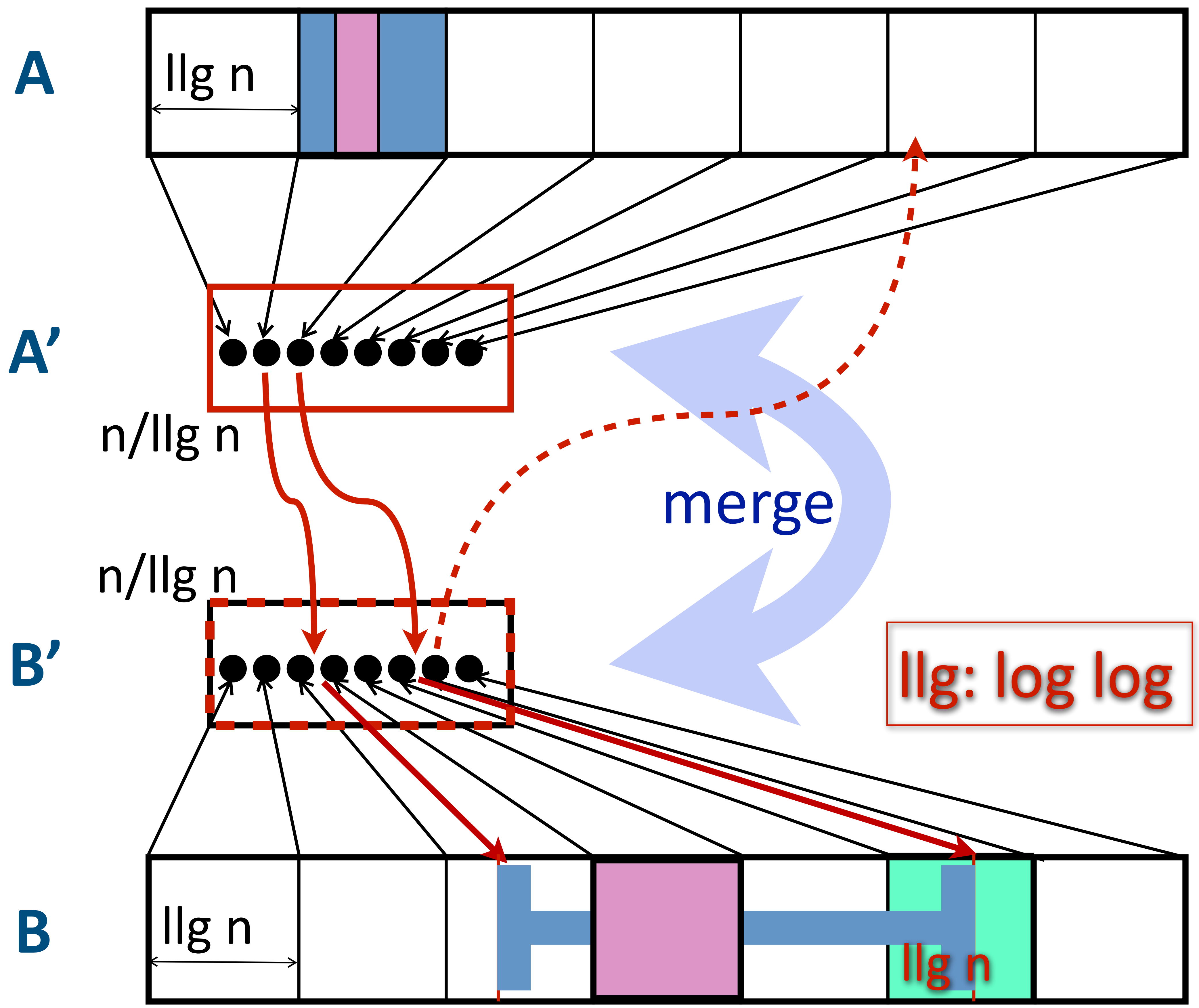
- Use the fast, but non-optimal, algorithm on small enough subsets
- Subdivide A and B into blocks of size ***log log n***
  - $A_1, A_2, ..$
  - $B_1, B_2, ..$
- Select first element of each block
  - $A' = p_1, p_2 ..$
  - $B' = q_1, q_2 ..$
- Now merge loglogn sized blocks n/loglogn times



Subodh Kumar

Optimal Merge (A,B)

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



Subodh Kumar

Quick Sort

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

```
QuickSort(int A[], int first, int last)
{
    Select random m in [first:last]    // A[m] is pivot
    forall i in [first:last]
        flag[i] = A[i] < A[m];
    Split(A); // Separate flag values 0 and 1
              // Prefix sum, A[P[m]] = A[m]
    Quicksort A[first:k-1] and A[k+1:last]
}
```

$T(n) \sim T(n/2) + O(\log n)$
$W(n) \sim 2W(n/2) + O(n)$

Subodh Kumar

Partitioning Approach

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

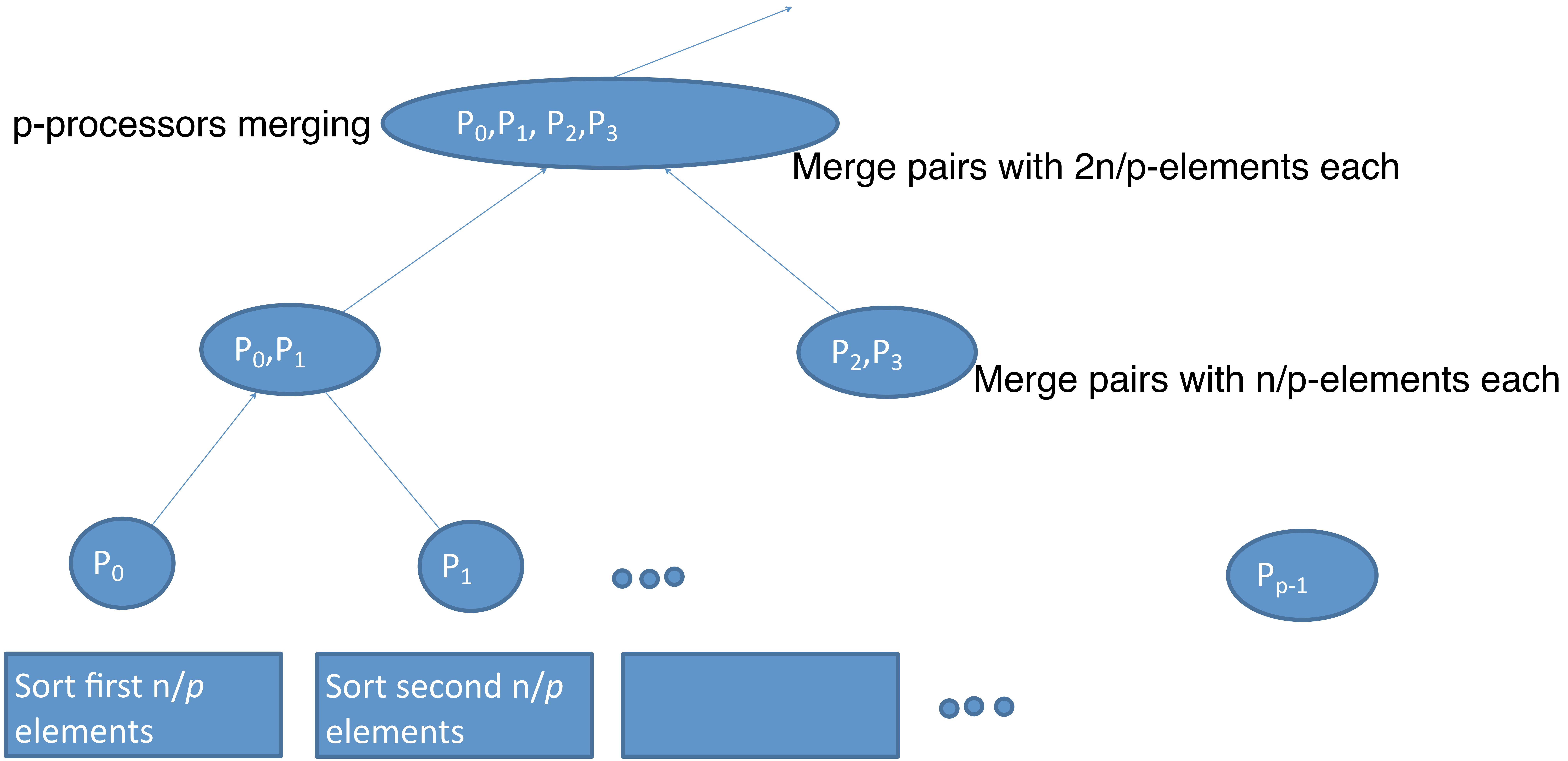
Subodh Kumar

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
- $T(n) = T(n/2) + O(\log \log n)$
- $W(n) = 2\, W(n/2) + O(n)$

Sort n/p elements, then Merge

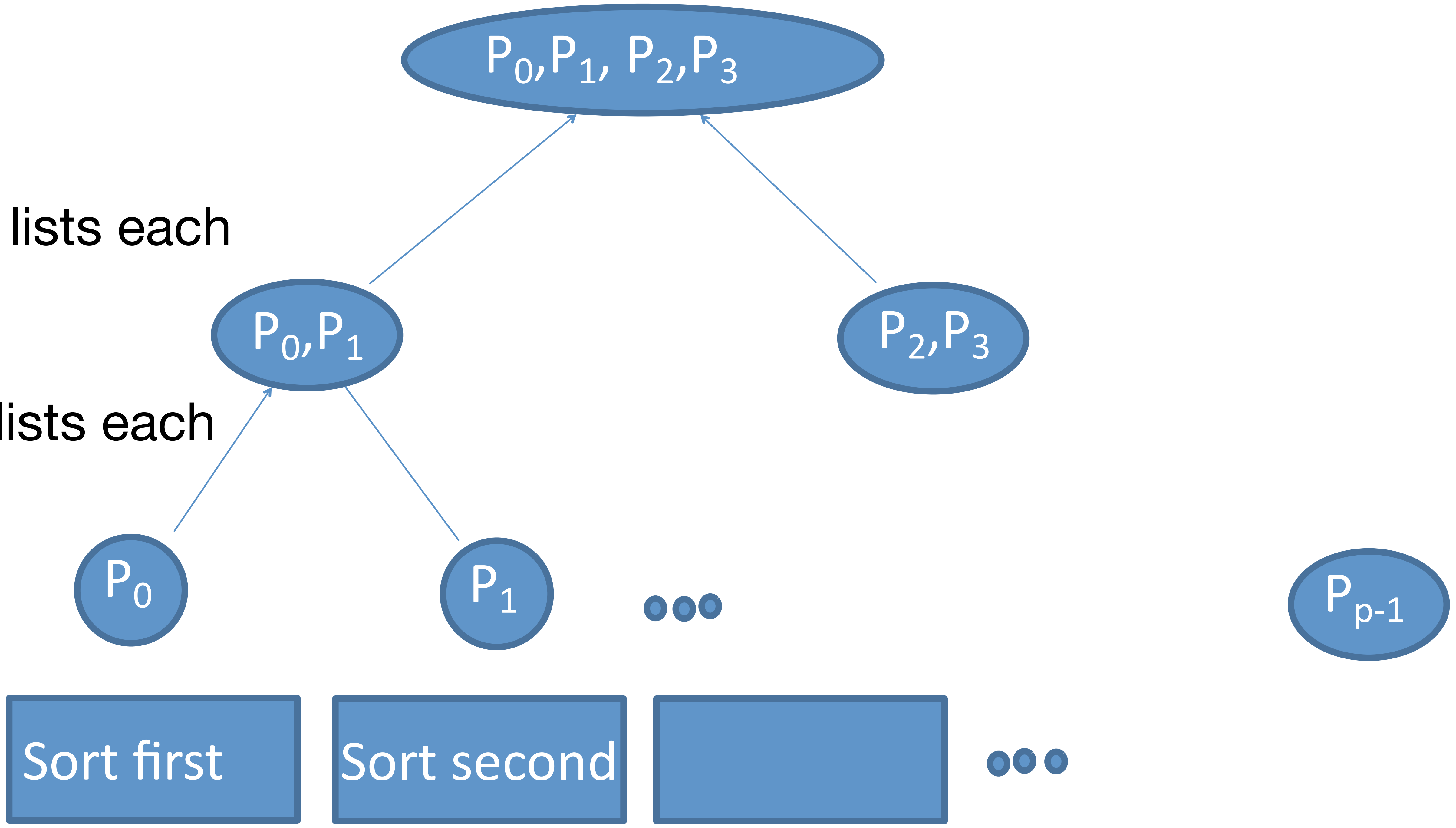


HOW EFFICIENTLY CAN YOU MERGE?

Subodh Kumar

Merge Sort

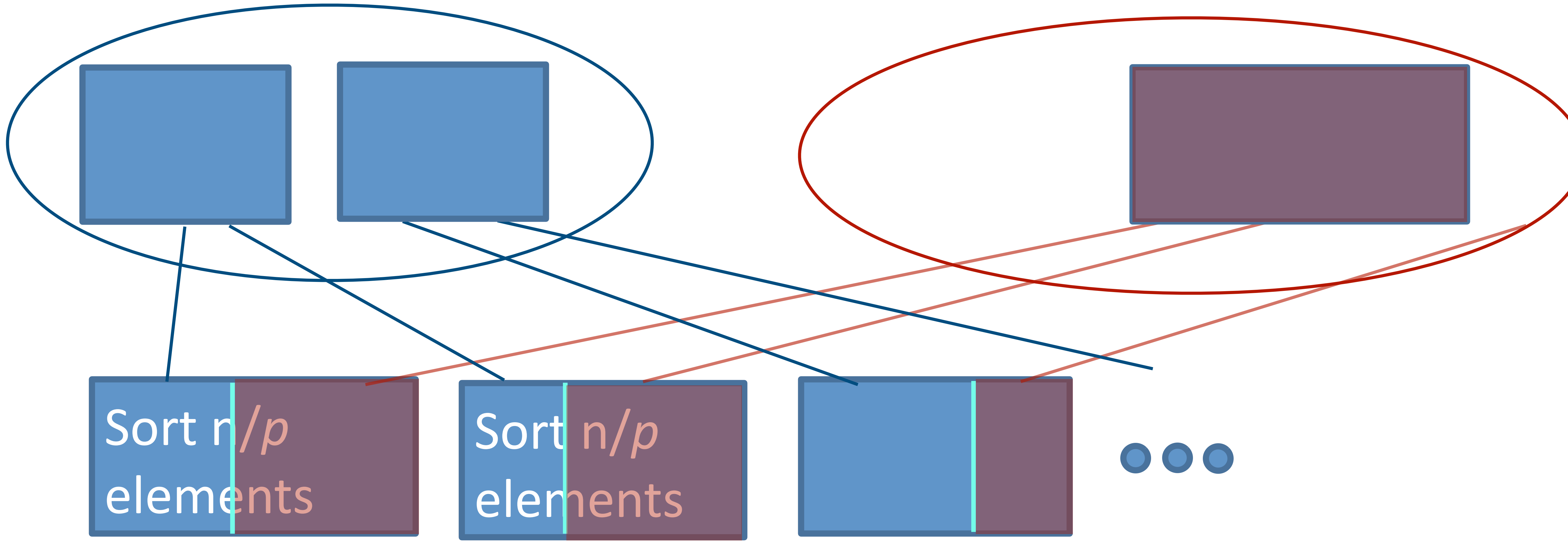
- Divide into p groups
  - Locally sort each group
  - n/p log (n/p) = O((n log n)/p)
- Parallel merge p groups
  - Binary tree: log(p) stages
    - @leaf (level 1), 2 processors merge two n/p sized lists each
      - time = O(n/p)
    - @Level i: 2^{i+1} processors merge two 2^i n/p sized lists each
      - time = O(n/p)
    - @Root: p processor merge two n/2 sized list
      - time = O(n/p)
    - O(n/p) log p



Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - Sort each group independently
  - O(n/p log n)
- Choose median of one of the groups
- Partition each group into “less” and “more” set
  - Binary search of the median: (log n)
- Separate into low and high
  - Merge p/2 “less” and p/2 “more” pairs
  - Each sequentially: O(n/p)
- Now we have p/2 “less” lists and p/2 “more” lists
  - Partition and recurse
- Total time = O(n/p log n) with high probability



Subodh Kumar

Parallel Bucket Sort

- Divide the range [a,b] of numbers into p equal sub-ranges
  - Or, buckets
- Divide input into p blocks
  - arbitrarily
- Each p_j sorts the elements in its block into p buckets
  - "Sends" i^{th} bucket to p_i
  - p_i collects bucket i from each other processor
  - For uniformly distributed input, expected bucket size is uniform
- Locally sort each bucket

O(n/p log n/p + p log p)?

But, real risk of load imbalance

Sample sort:

Choose a sample of size s

Sort the samples

Choose **B-1** evenly spaced element from the sorted list

These splitters provide ranges for **B** buckets

Parallel Splitter Selection

- Divide n elements equally into B blocks
- (Quick)Sort each block
- For each sorted block:

$(n/B \log n/B)$

  - Choose B-1 evenly spaced samples
- Use the B*(B-1) elements as samples
- Sort the samples
  - Choose B-1 **Splitters**

$(B^2 \log B)$

- Arrange elements by bucket in output array

$(n/B + B \log B)$

  - Count the number of elements in each bucket
    - Perform prefix Sum of counts; Reserve space per bucket
  - In-place
  - No bucket contains more than **2*n/B** elements

Subodh Kumar

Parallel algorithm techniques: ACCELERATED CASCADING

Min-find

Input: array with n numbers

<u>Algorithm A1 using $O(n^2)$ processors:</u>

```
 parallel for i in (0:n]
     M[i]:=0
parallel for i,j  in (0:n]
     if i≠j && C[i] < C[j]
          M[j]=1
parallel for i in (0:n]
     if M[i]=0
          min = A[i]
```

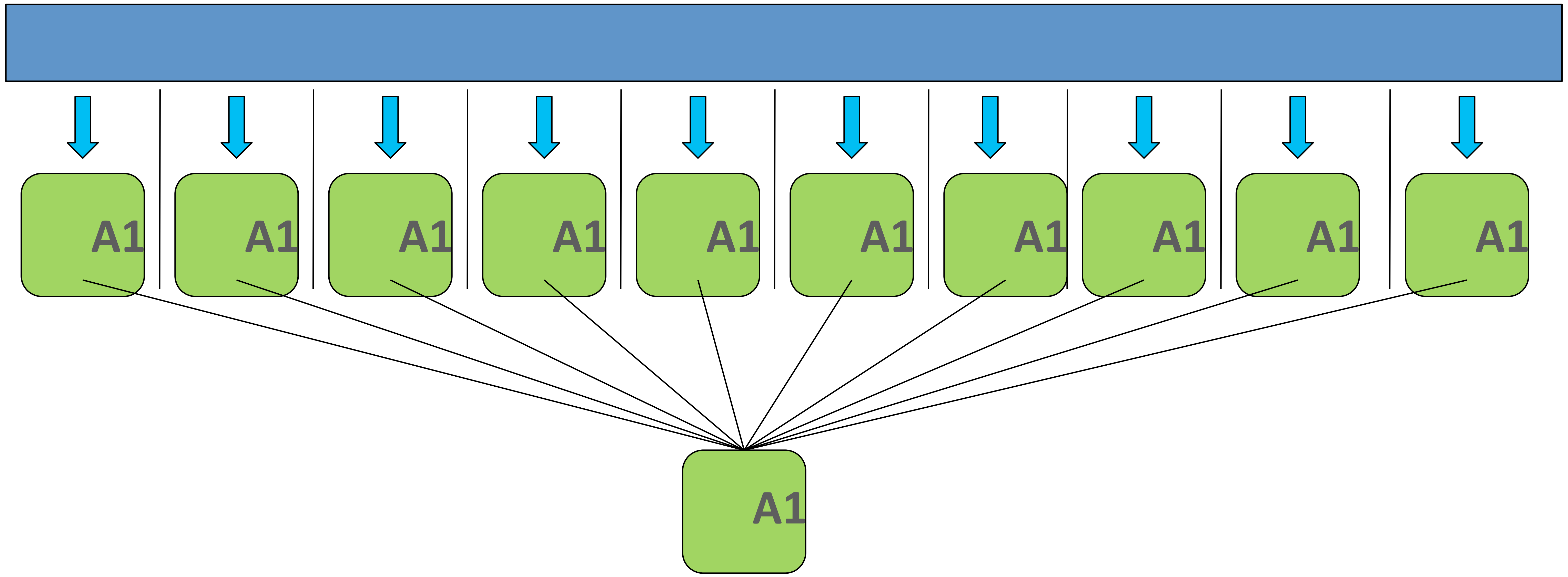
Not optimal: $O(n^2)$ work

Subodh Kumar

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 $n\sqrt{n}$



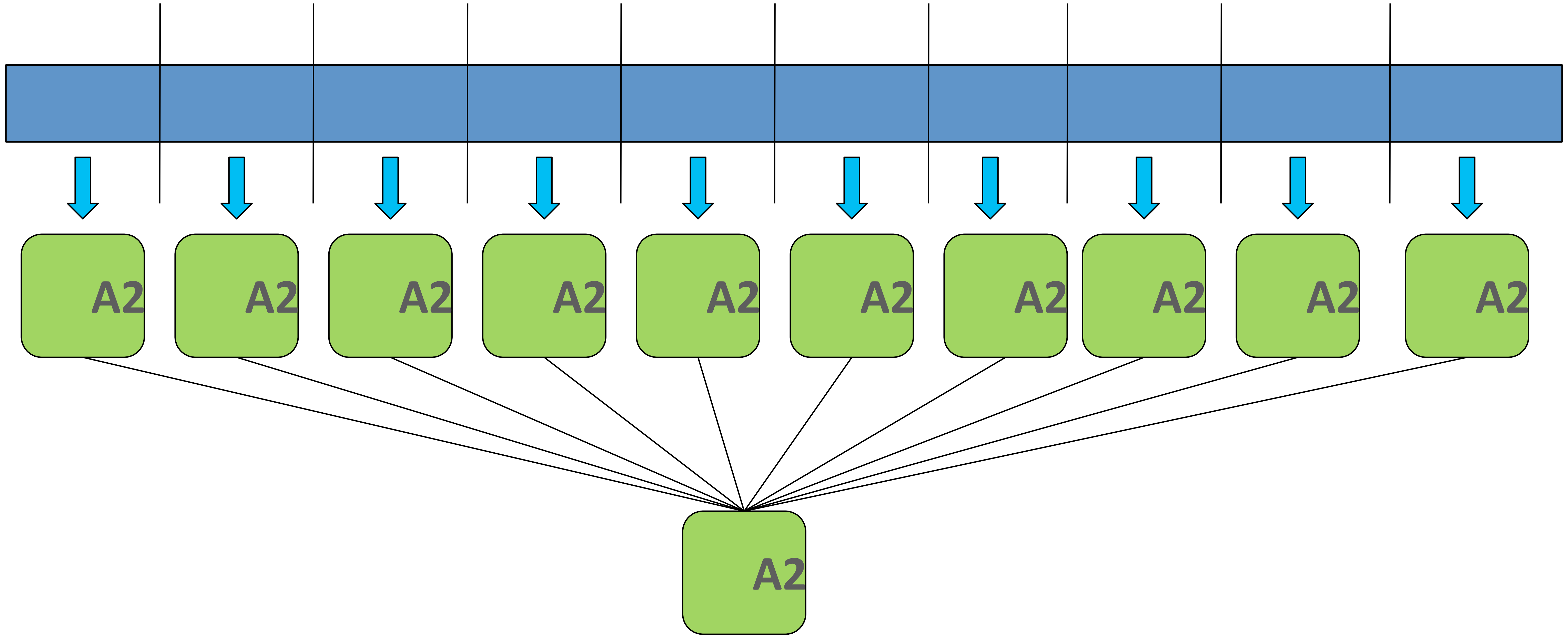
Step 1: Partition into disjoint blocks of size $\sqrt{n}$

Step 2: Apply A1 to each block $n\sqrt{n}$

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

Subodh Kumar

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



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

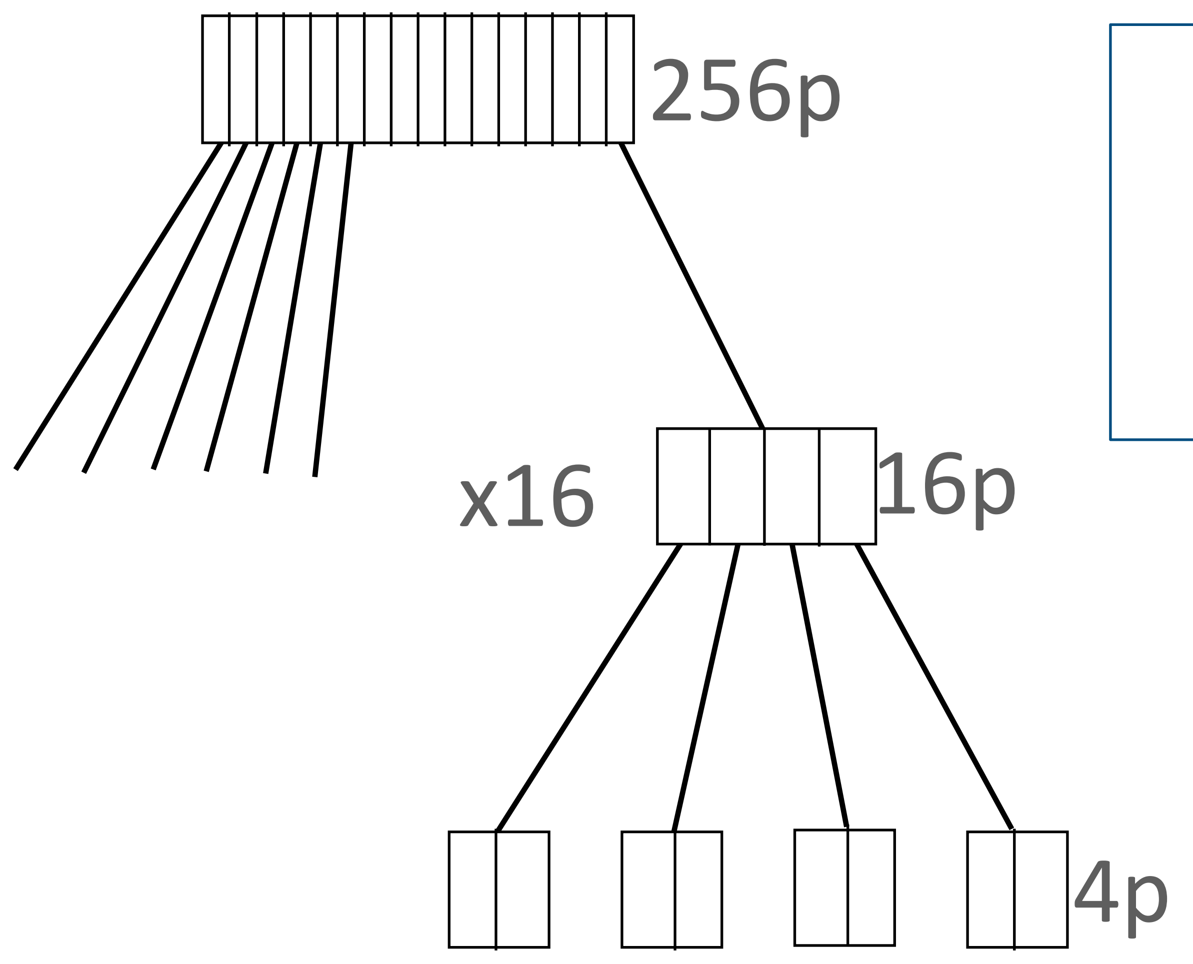
Step 2: Apply A2 to each block $\quad n^{1/2} \cdot n^{3/4}$

Step 3: Apply A2 to the results from the step 2 $\quad n^{3/4}$

Subodh Kumar

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 2: $n/n^{1/2}$ minima



A_1 A_2 A_3 ..

n^2 -> $n^{1+1/2}$ -> $n^{1+1/4}$ -> $n^{1+1/8}$ -> $n^{1+1/2^k}$ .. $\sim n^{1+\varepsilon}$

Algorithm A_∞ takes ?? with $n^{1+\varepsilon}$ processors

$\boxed{n^{\frac{1}{2^i}} = O(1) \text{ at leaf}}$

Doubly logarithmic-depth tree

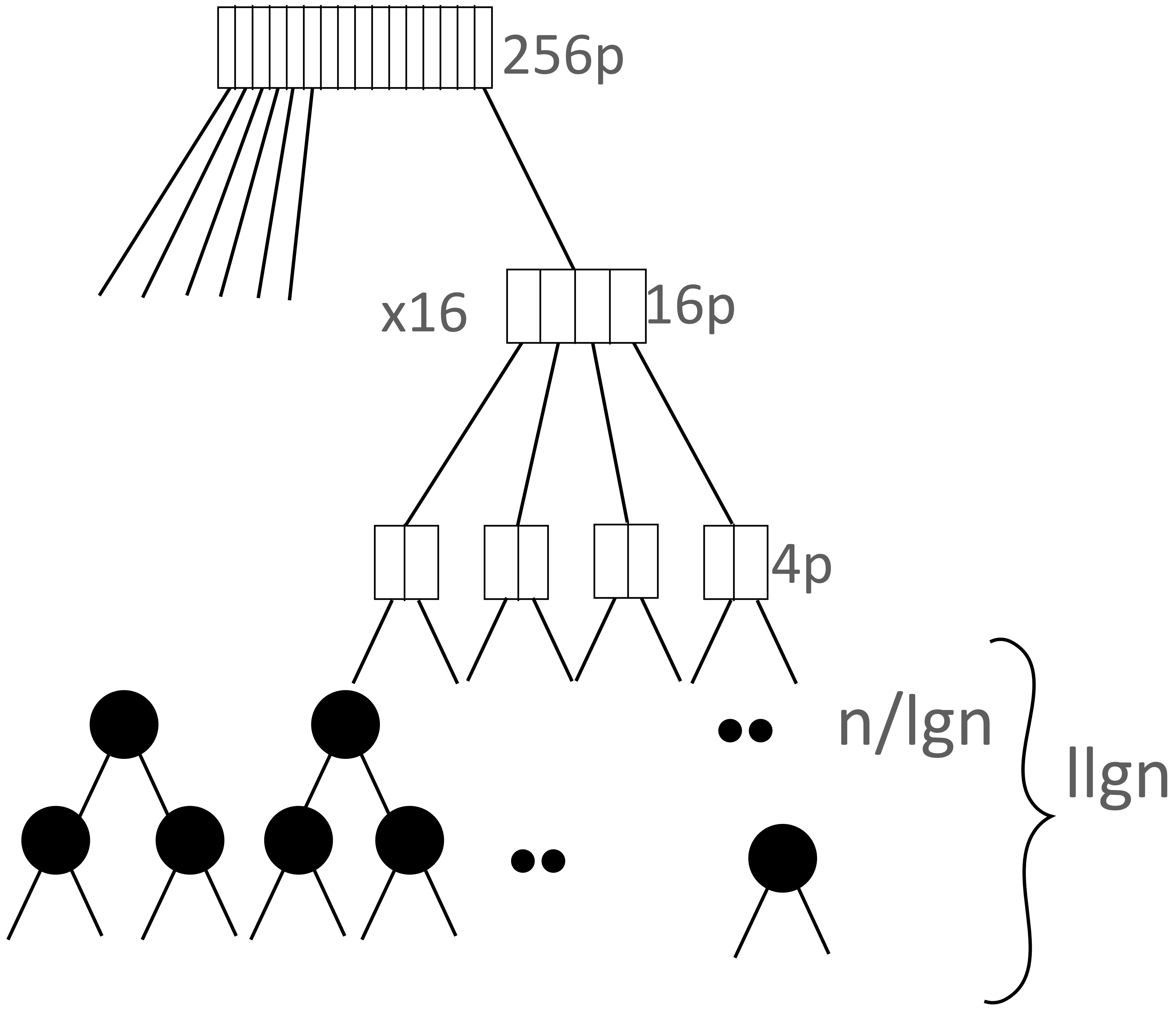
n log log n work, log log n time

Min-Find Review

- Constant-time algorithm
  - $O(n^2)$ work
- O(log n) Balanced Tree Approach
  - O(n) work (Work-Optimal)
- O(loglog n) Doubly-log depth tree Approach
  - O(n loglog 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 loglog n levels
  - Then switch to O(n loglog n)-work algorithm



n work, log log n time

Subodh Kumar