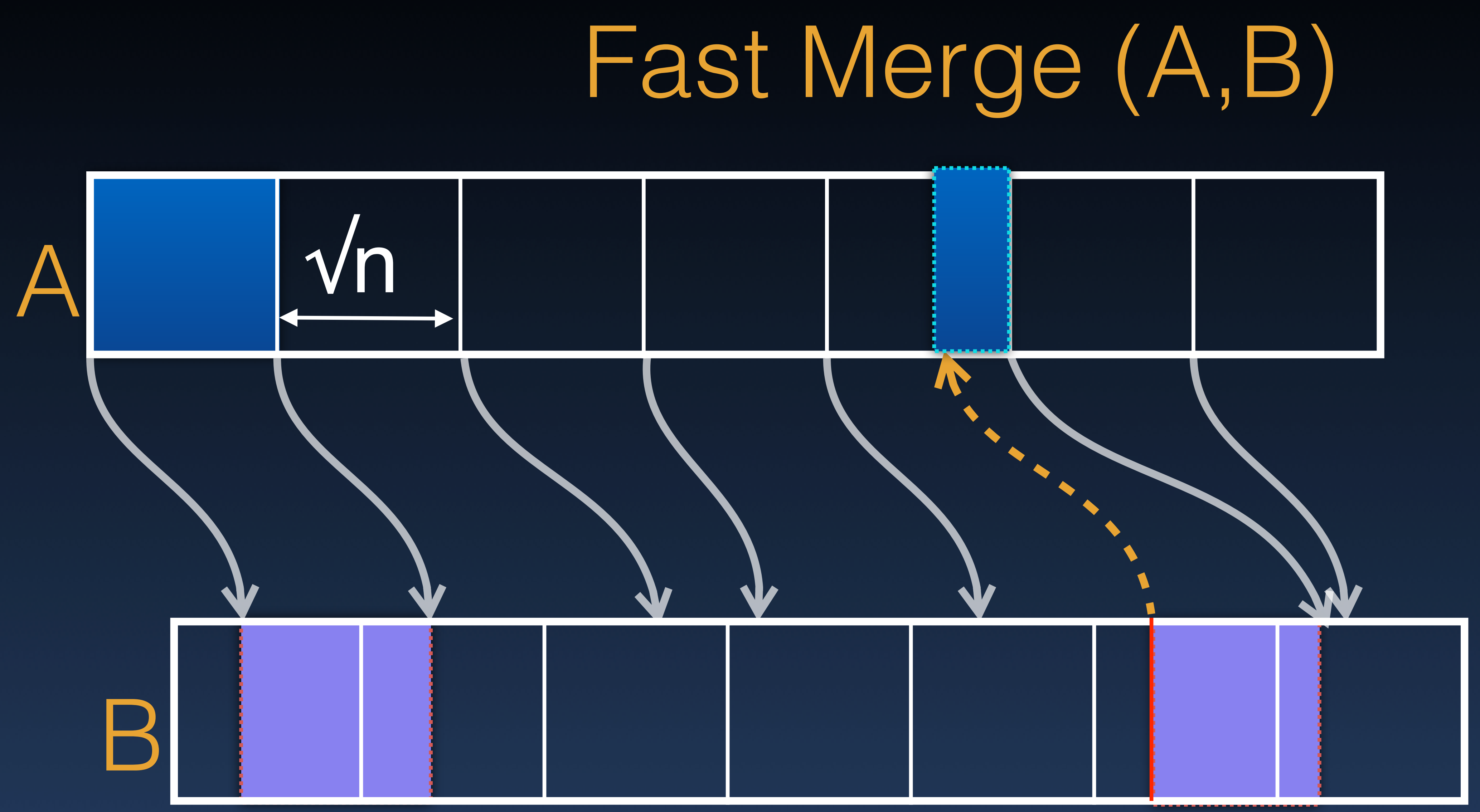
COL380

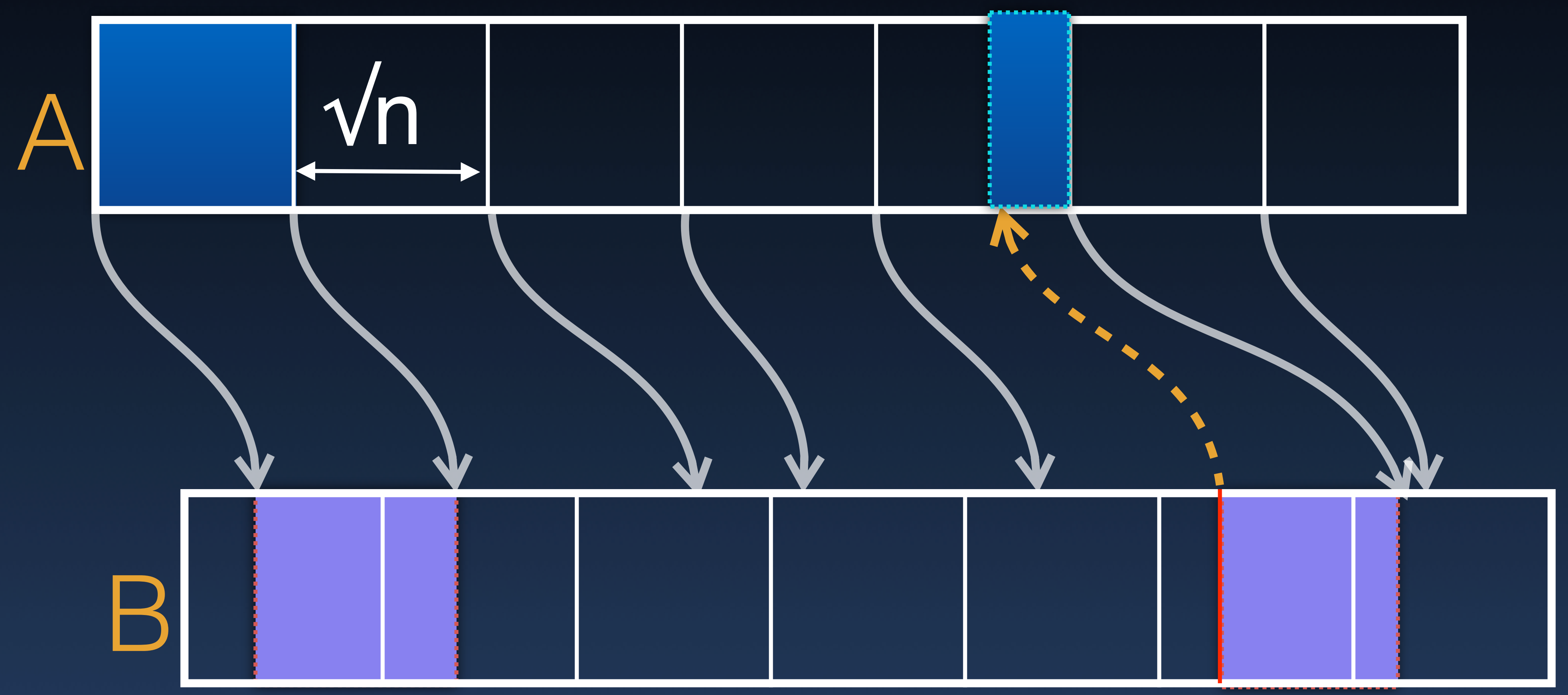
Introduction to
Parallel & Distributed Programming



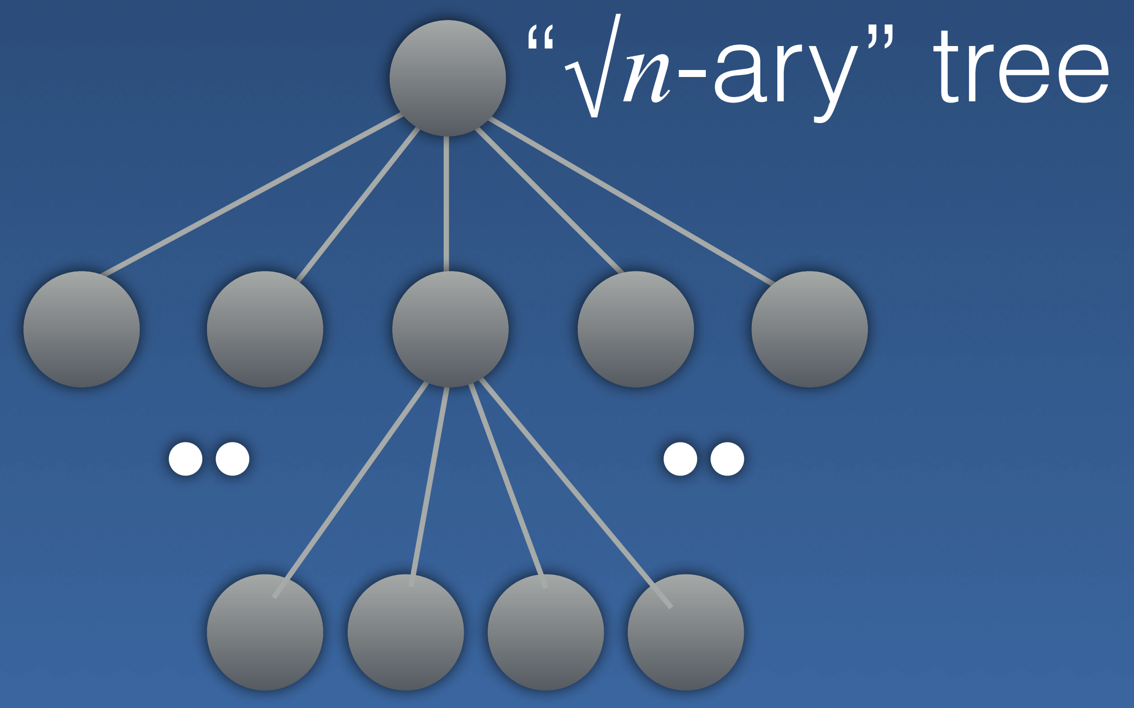
- Partition A and B into ***√n*** blocks each
- Select from A, elements i√n, i ∈ [0: √n)
- Rank each selected element of A in B
  - ➡ √n Parallel searches, use √n processors for each search
- Similarly rank √n selected elements from B in A
- <u>Recursively merge pairs of sub-sequences</u>
  - ➡ Total time: $T(n) = O(1)+T(\sqrt{n}) = O(\log \log n)$
  - ➡ Total work: $W(n) = O(n)+ \sqrt{n}\, W(\sqrt{n}) = O(n \log \log n)$

Subodh Kumar

Fast Merge (A,B)



- Partition A and B into ***√n*** blocks each
- Select from A, elements $i\sqrt{n}$, $i \in [0: \sqrt{n})$
- Rank each selected element of A in B
  - ➡ √n Parallel searches, use √n processors for each search
- Similarly rank √n selected elements from B in A
- <u>Recursively merge pairs of sub-sequences</u>
  - ➡ Total time: $T(n) = O(1)+T(\sqrt{n}) = O(\log \log n)$
  - ➡ Total work: $W(n) = O(n)+ \sqrt{n}\, W(\sqrt{n}) = O(n \log \log n)$



Subodh Kumar

Min-find

CRCW

Input: array A with n elements

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

A

Min-find

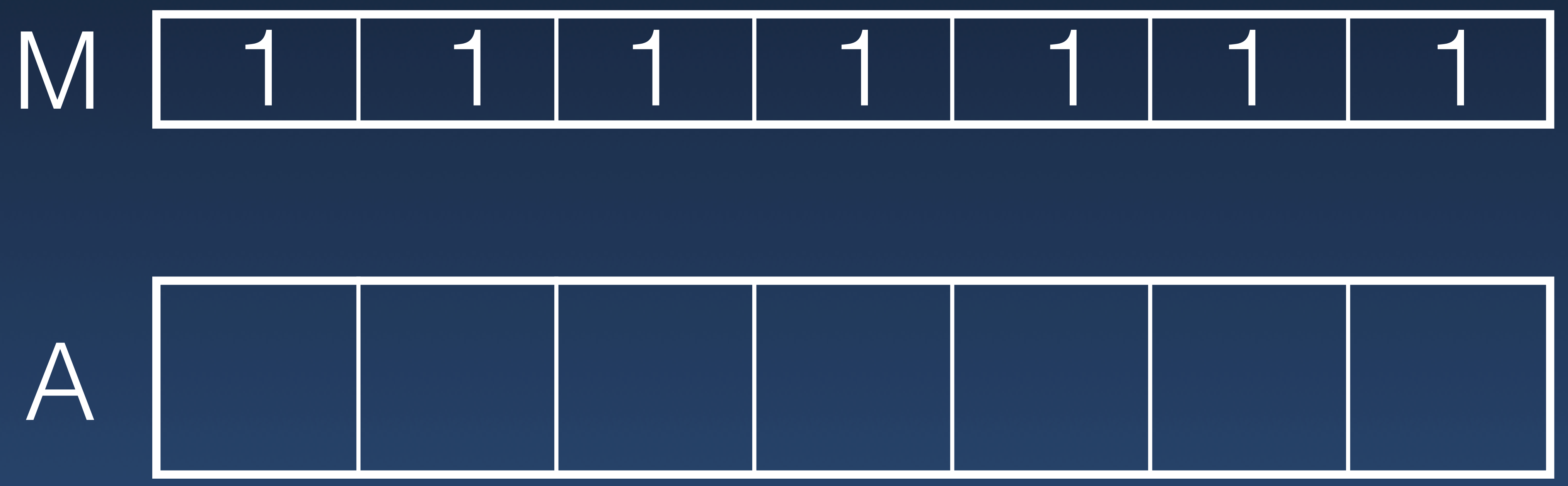
CRCW

Input: array A with n elements

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

forall i in [0:n)

M[i] = 1



Subodh Kumar

Min-find

Input: array A with n elements

CRCW

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

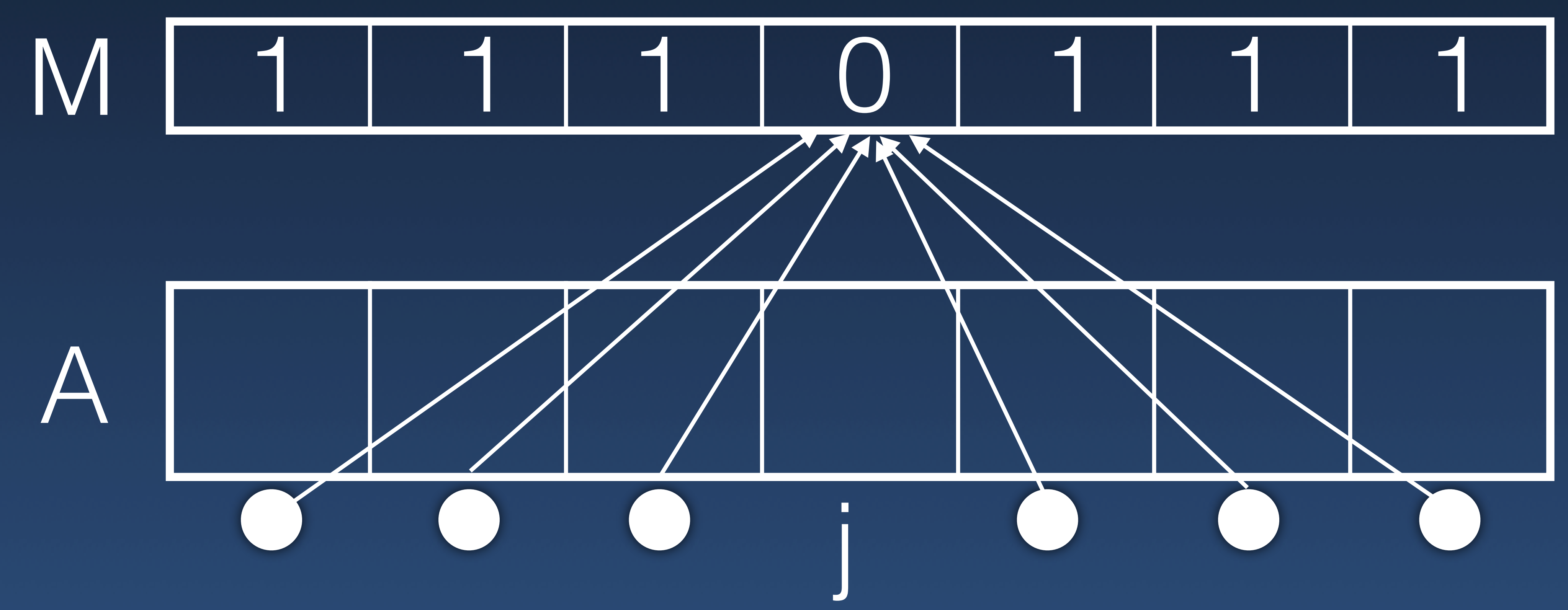
forall i in [0:n)

M[i] = 1

forall i,j in [0:n)

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

M[j] = 0



Min-find

Input: array A with n elements

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

forall i in [0:n)

M[i] = 1

forall i,j in [0:n)

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

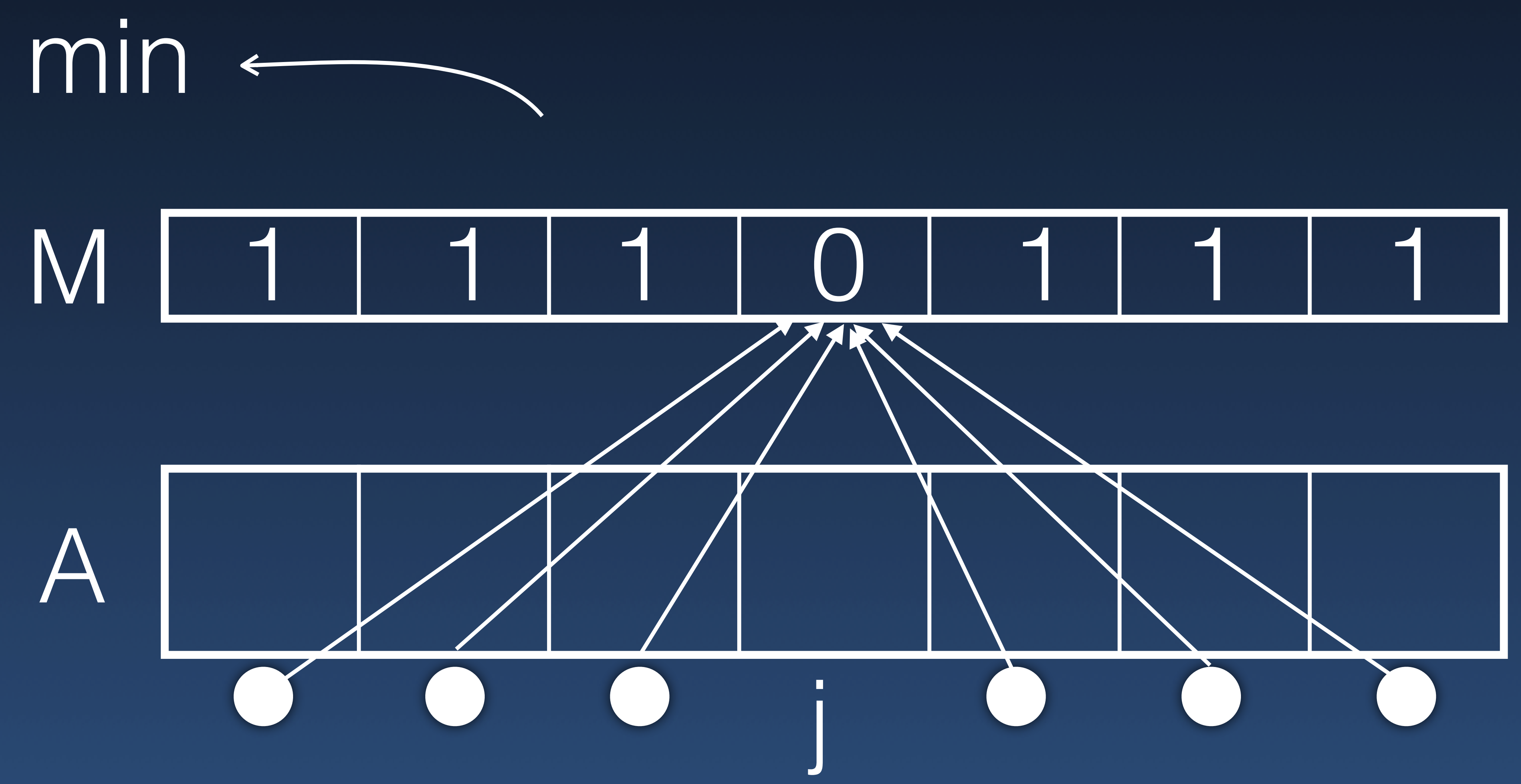
M[j] = 0

forall i in (0:n]

if M[i]==1

min = A[i]

CRCW



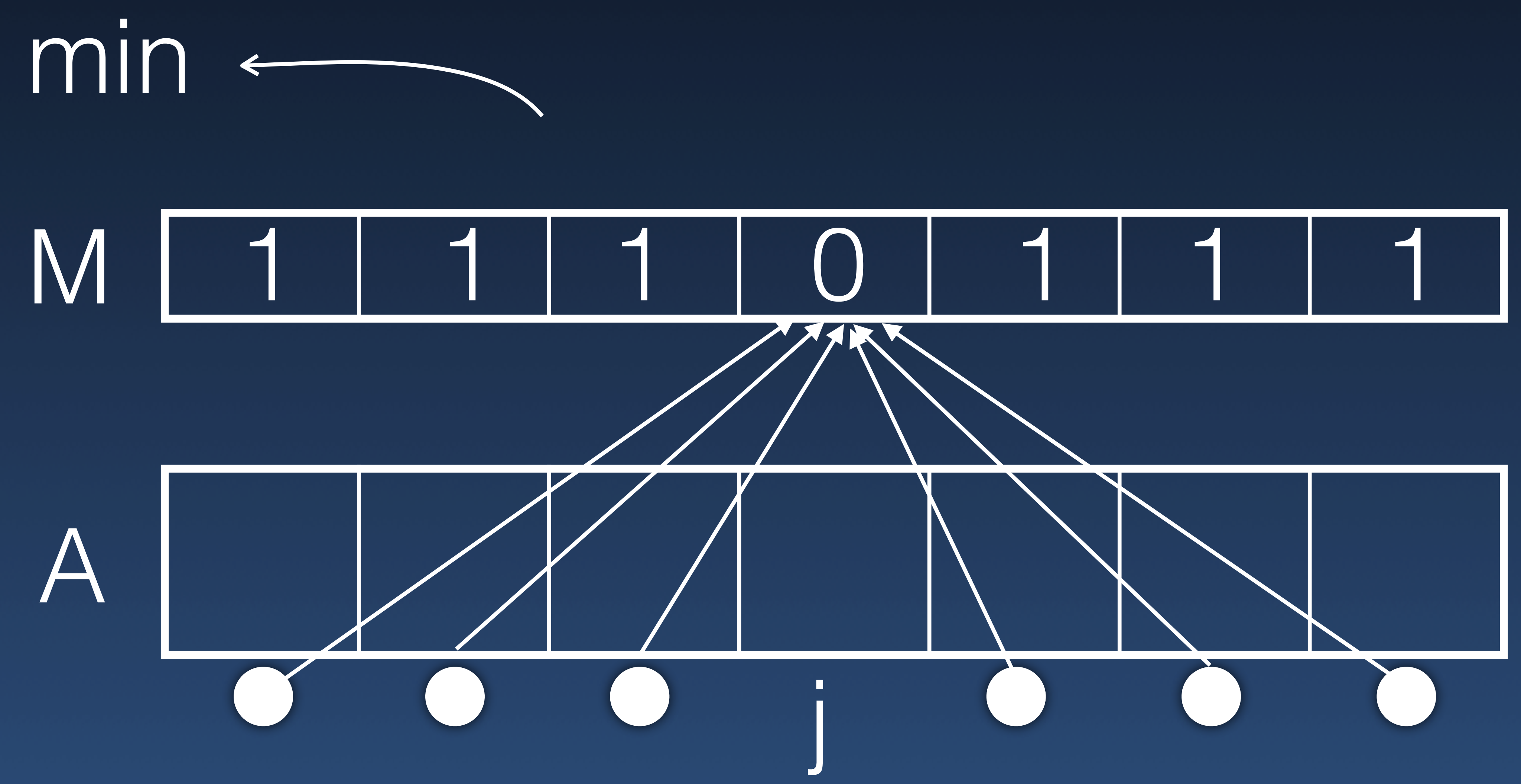
Subodh Kumar

Min-find

Input: array A with n elements

CRCW

<u>Algorithm A1 using O(n²) processors:</u>



forall i in [0:n)

    M[i] = 1

forall i,j in [0:n)

    if i≠j && A[i] < A[j]

        M[j] = 0

forall i in (0:n]

    if M[i]==1

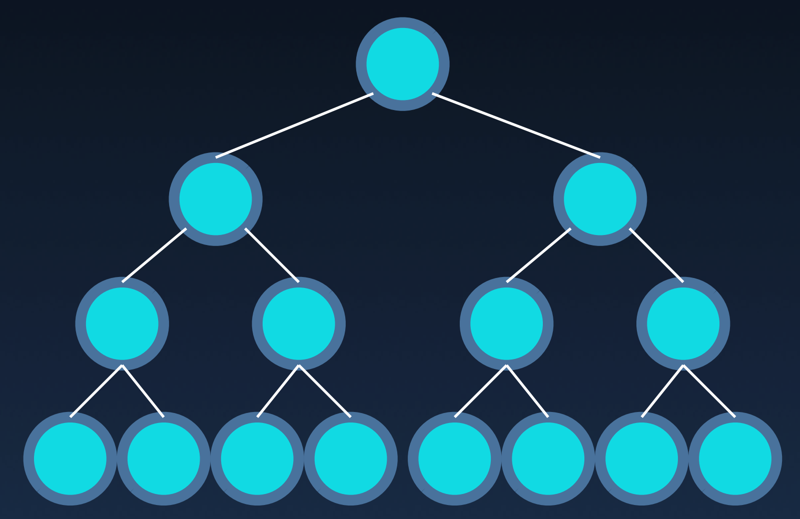
        min = A[i]

O(1) time, $O(n^2)$ work: Not optimal

Subodh Kumar

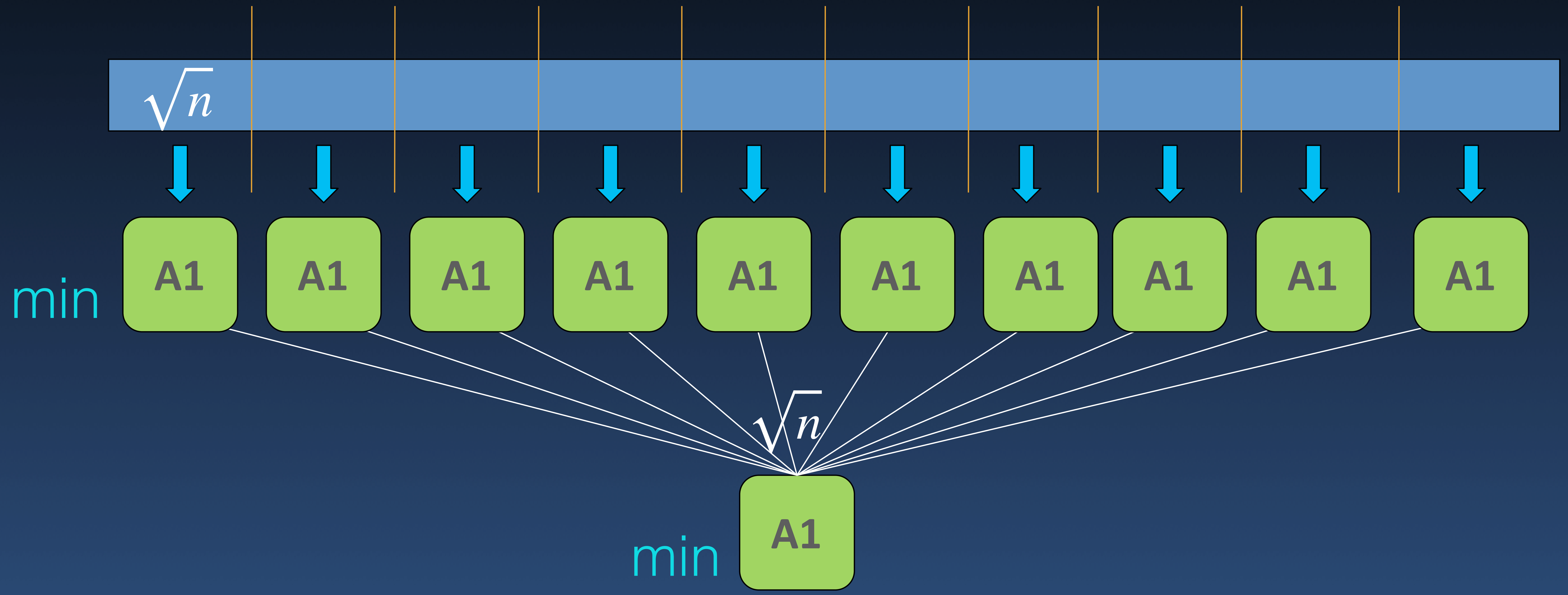
Optimal Min-find

- Balanced Binary tree
  - ➡ O(log n) time
  - ➡ O(n) work => Optimal



- Make the tree branch quicker
  - ➡ Number of children of node handling n items = $\sqrt{n}$
    - amounts to n leaves in this subtree
- Use Accelerated cascading

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

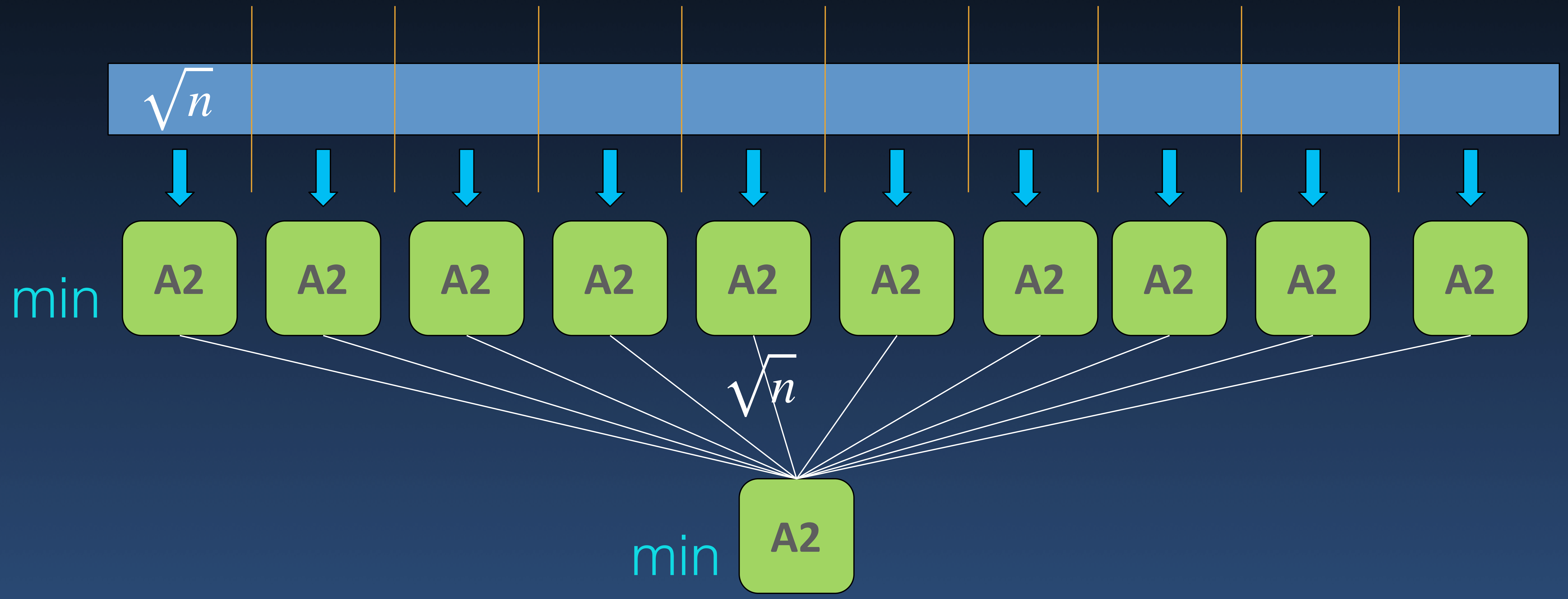


<u>Algorithm A2</u>

	A2 work
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 $n^{1+1/4}$



Algorithm A3

	A3 work	A2 work
Step 1: Partition into disjoint blocks of size		
Step 2: Apply A2 to each block	$n^{\frac{1}{2}} n^{\frac{3}{4}}$	$n\sqrt{n}$
Step 3: Apply A2 to the results from the step 2	$n^{\frac{3}{4}}$	n

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

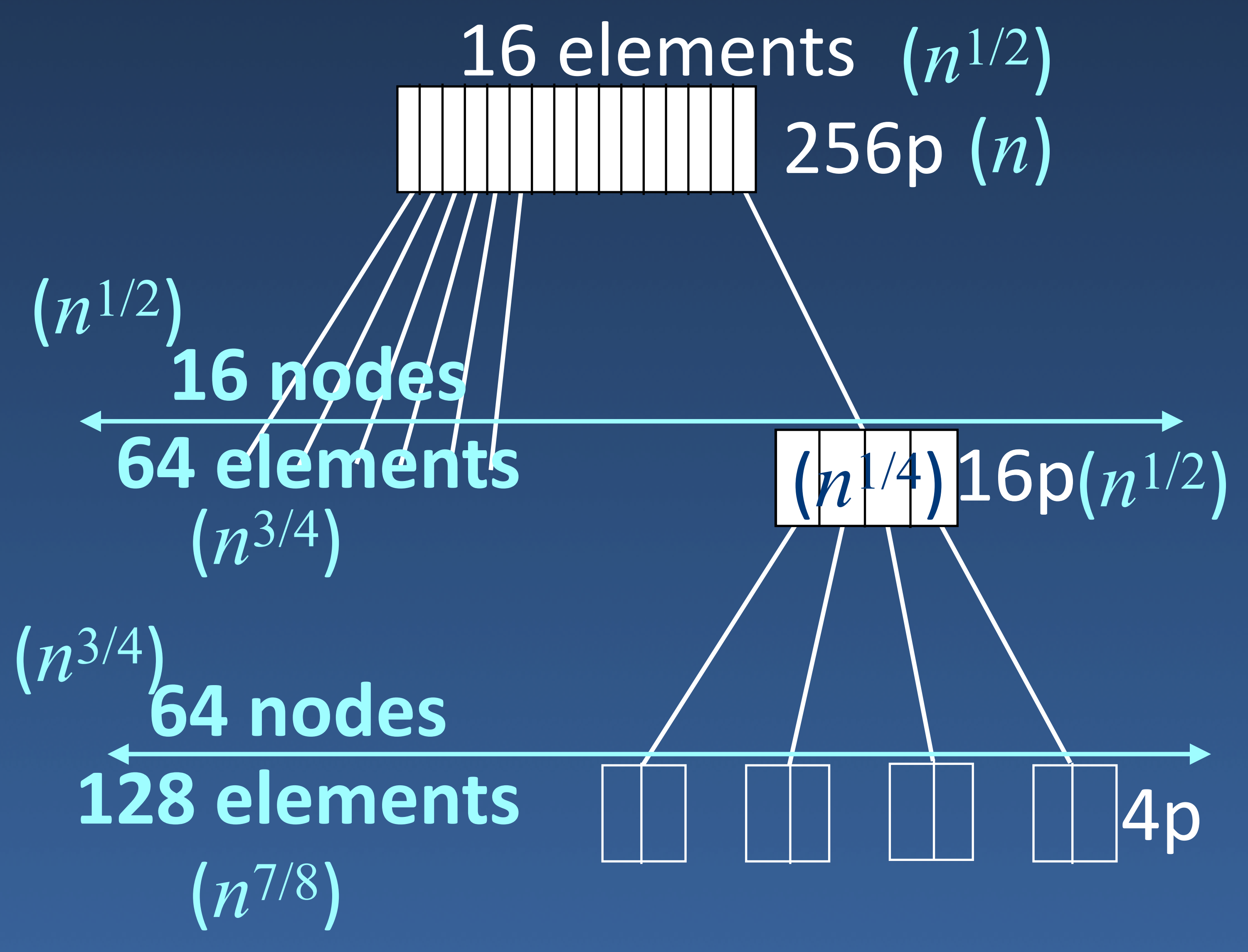
A_1		A_2		A_3 ..						
n^2	->	$n^{1+1/2}$	->	$n^{1+1/4}$	->	$n^{1+1/8}$	->	$n^{1+1/2^k}$	..	$\sim n^{1+\varepsilon}$

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

Doubly-log depth tree: $\boxed{n^{\frac{1}{2^i}} = O(1) \text{ at leaf}}$

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: minimum of $n^{1/2}$ 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 $n^{1+\epsilon}$ processors

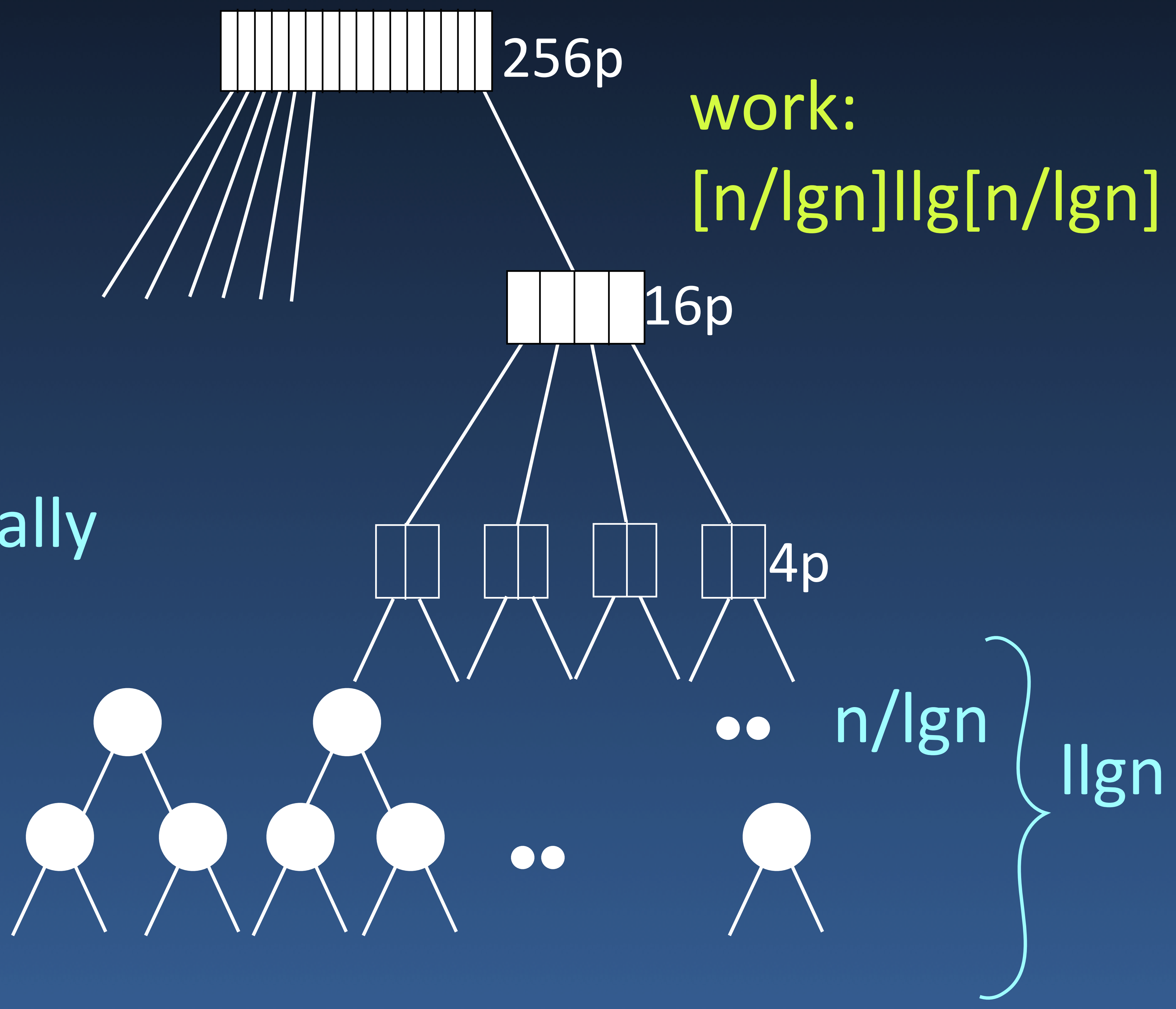
Doubly-log depth tree: $n^{\frac{1}{2^i}}$ = O(1) at leaf

n loglog n work, loglog n time

- O(log n) Balanced tree approach
  - ➡ O(n) work (Work-Optimal)
- Constant-time algorithm
  - ➡ $O(n^2)$ work
- O(loglog n) Doubly-log depth tree approach
  - ➡ O(n loglog n) work
  - ➡ O(loglog n) time

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



O(n) work, O(loglog n) time

Quick Sort

- Choose the pivot
    - ➡ Select median?
- Subdivide into two groups
    - ➡ Group sizes linearly related with high probability (expect log(n) rounds)
- Sort each independently in parallel

```
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]
}
```

Quick Sort

- Choose the pivot
  - ➡ Select median?
- Subdivide into two groups
  - ➡ Group sizes linearly related with high probability (expect log(n) rounds)
- Sort each independently in parallel

```
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]
}
```

flag	1	0	1	0	0	1	1

Quick Sort

- Choose the pivot
  - ➡ Select median?
- Subdivide into two groups
  - ➡ Group sizes linearly related with high probability (expect log(n) rounds)
- Sort each independently in parallel

```
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]
}
```

psum	1	1	2	2	2	3	4
flag	1	0	1	0	0	1	1

Quick Sort

- Choose the pivot
  - ➡ Select median?
- Subdivide into two groups
  - ➡ Group sizes linearly related with high probability (expect log(n) rounds)
- Sort each independently in parallel

```
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]
}
```

psum	1	1	2	2	2	3	4
flag	1	0	1	0	0	1	1
		5		6	7		

(i - psum[i]) + psum[n-1]

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 independently in parallel
- 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]
}
```

psum	1	1	2	2	2	3	4
flag	1	0	1	0	0	1	1
		5		6	7		

(i - psum[i]) + psum[n-1]

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

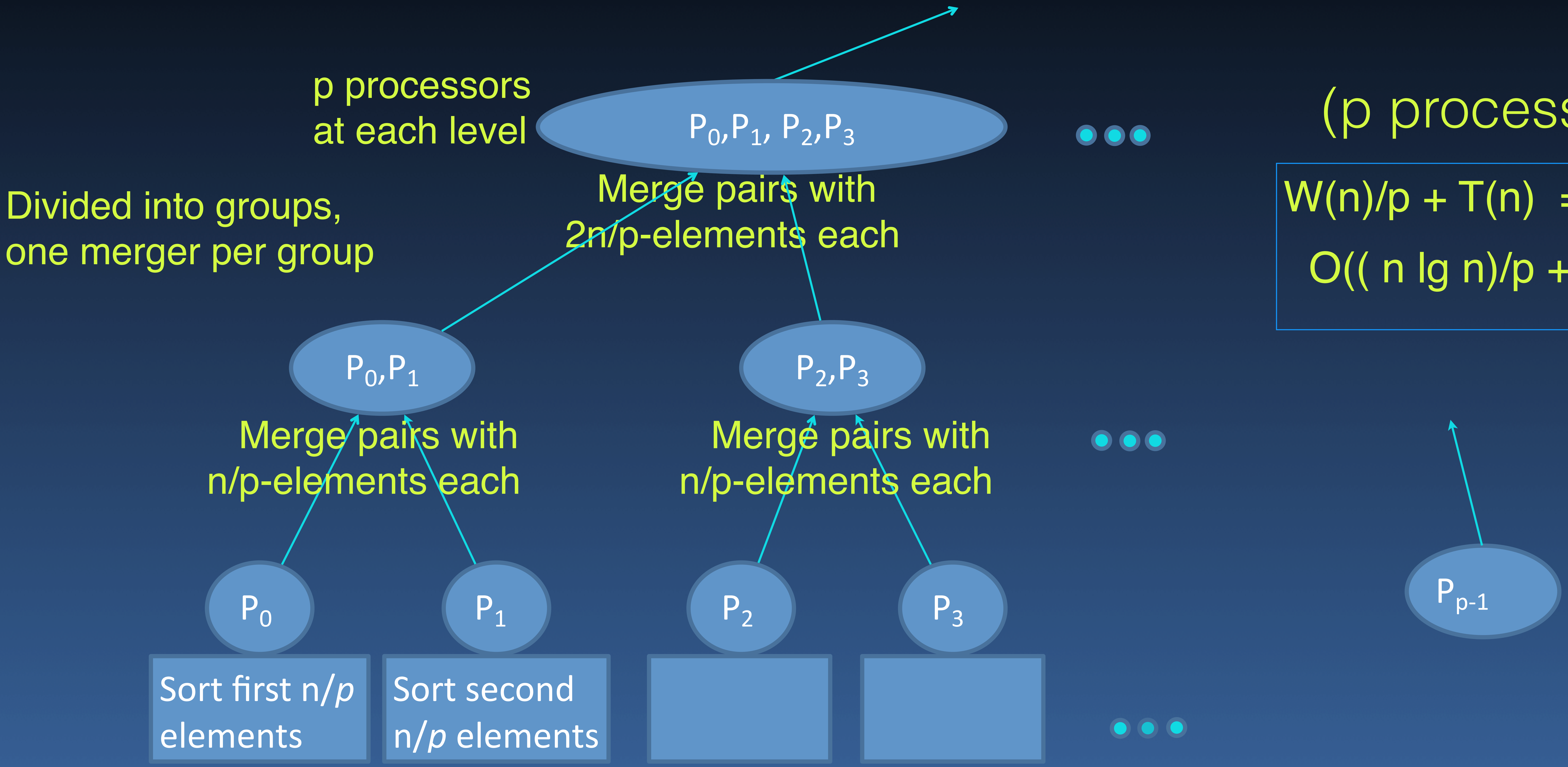
Subodh Kumar

Merge Sort

- Partition data into two halves
  - ➡ Assign half the processors to each half
- Sort each half in parallel
- Merge results in parallel
  - ➡ $T(n) = T(n/2) + O(\log \log n)$
  - ➡ $W(n) = 2\,W(n/2) + O(n)$

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

Sort each partition and Merge



(p processors)

$$W(n)/p + T(n) =$$
$$O((n \lg n)/p + O(\lg n \lg\lg n)$$

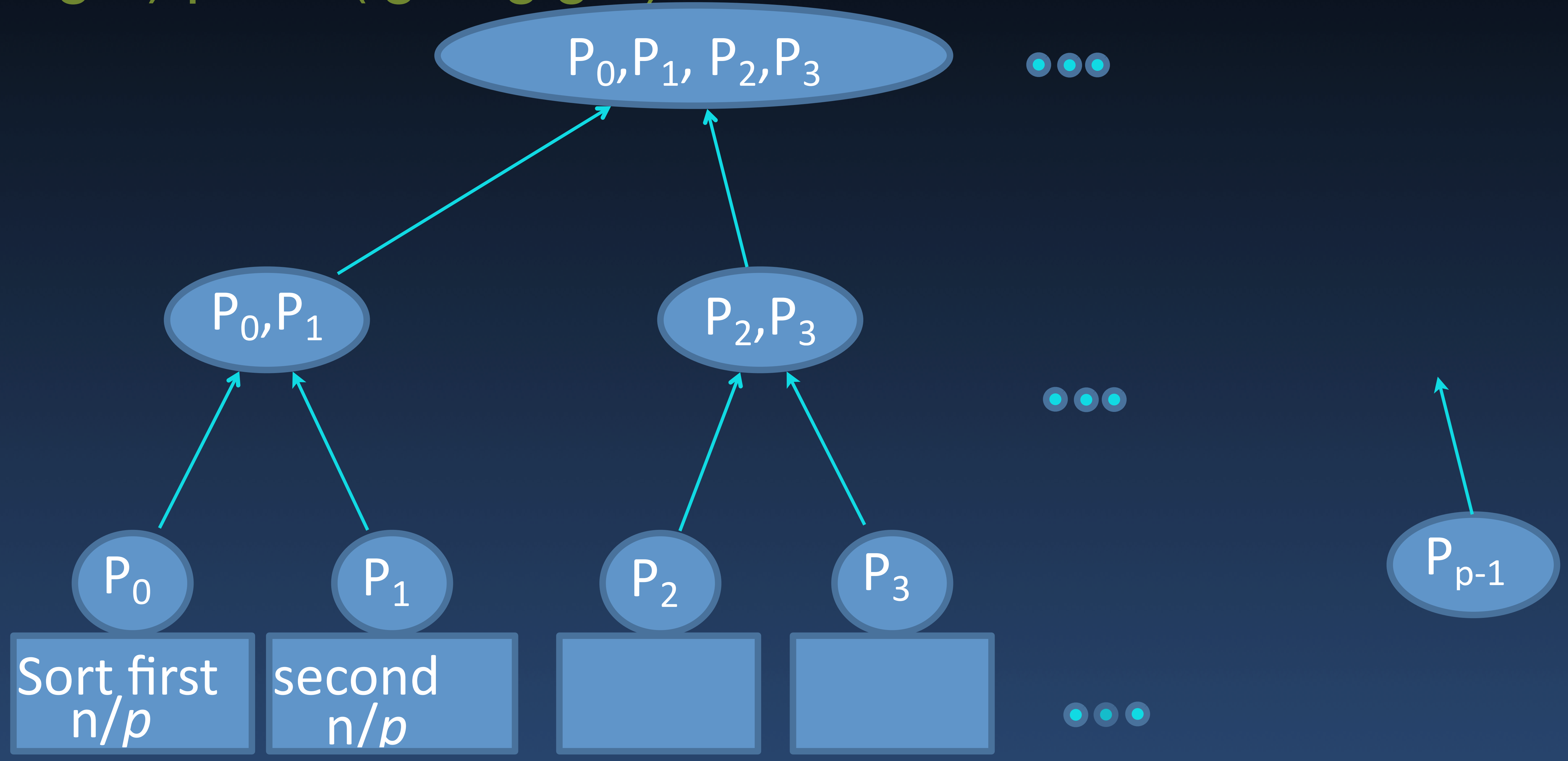
How efficiently can you merge?

Subodh Kumar

Merge Sort

- Divide into p groups
  - ➡ Locally sort each group
  - ➡ n/p log (n/p) = <u>O((n log n)/p)</u>
- Parallel merge p groups
  - ➡ Binary tree: log(p) stages
    - @level 1: 2 processors merge two n/p size lists, each in O(n/p) time
    - @level i: 2^{i+1} processors merge two 2^i n/p size lists, each in O(n/p + llg n/p) time
    - @root: p processors merge two n/2 size lists in O(n/p + llg n) time
  - ➡ <u>O(n/p lg p + lg p llg n)</u> total time for mergers

O((n lg n)/p + O(lg n lglg n)



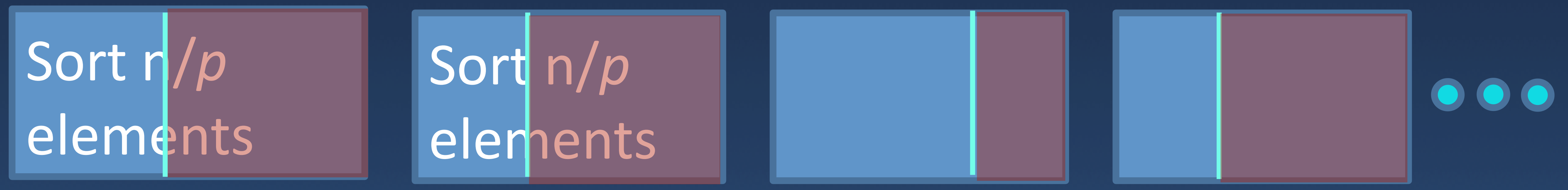
Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time

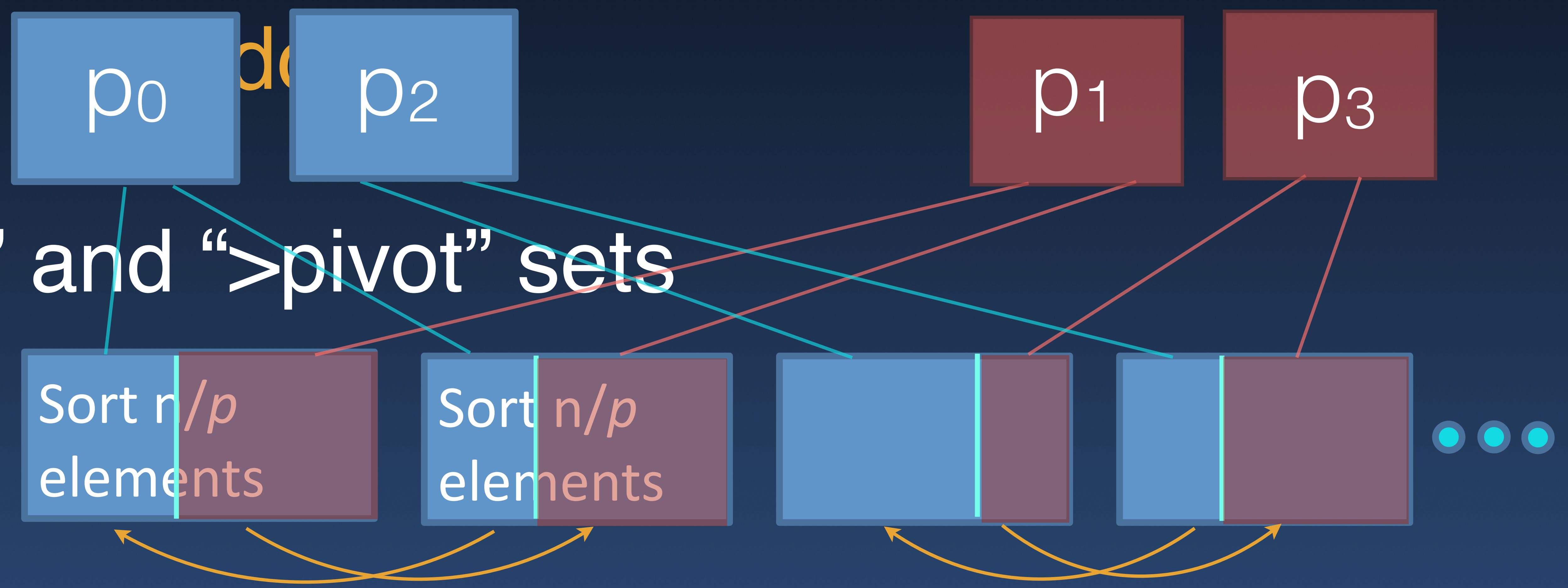
Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group (broadcast)
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)



Hyper Quick Sort

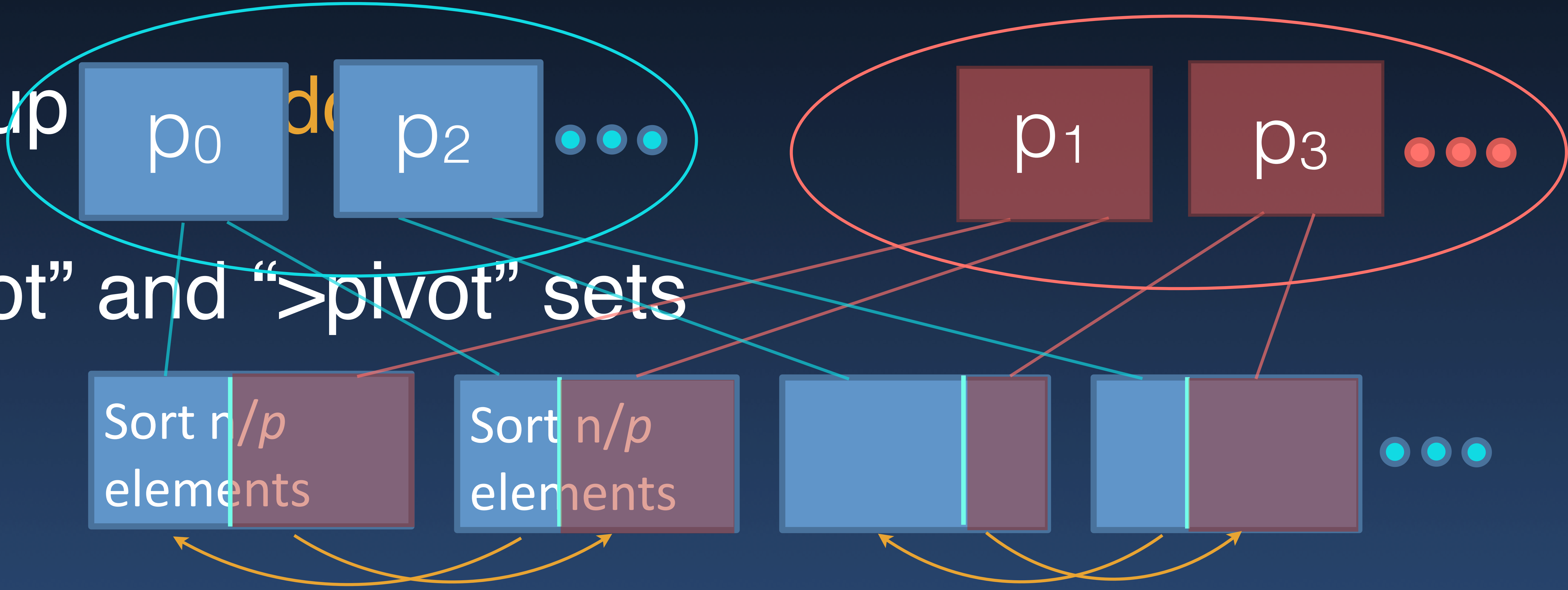
- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group



- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)

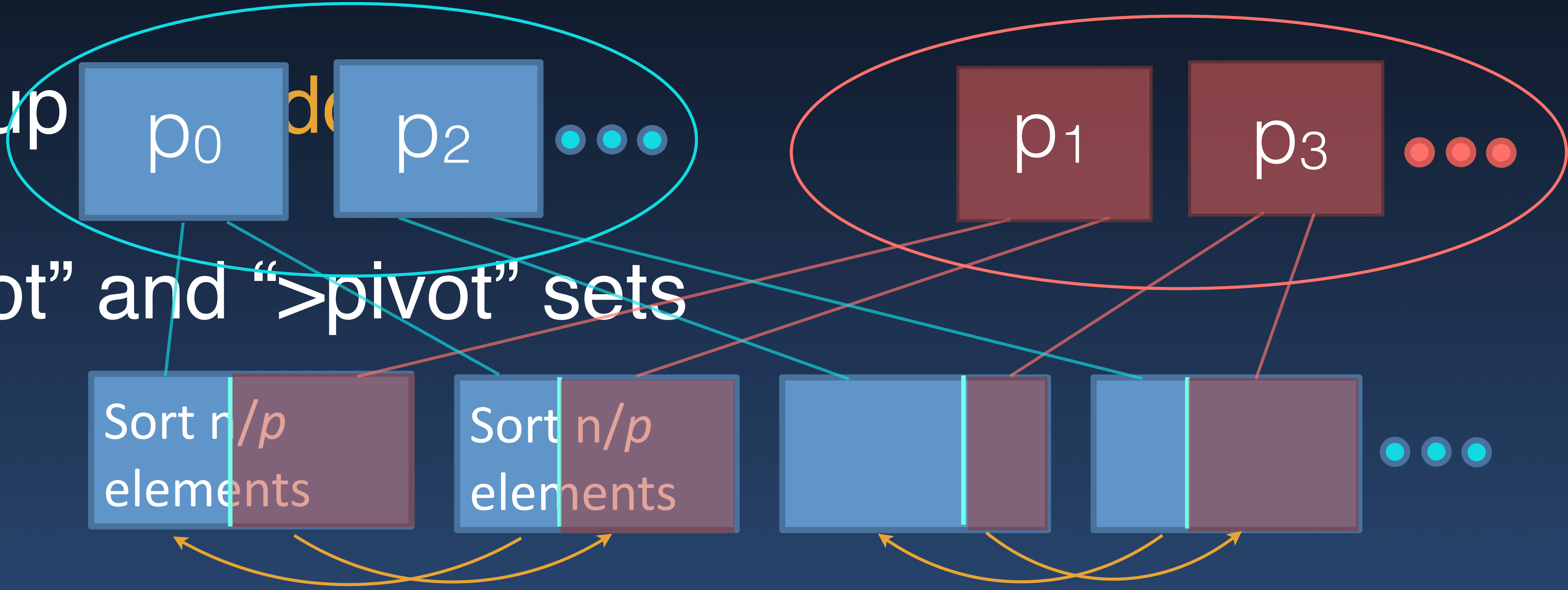


- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)
- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

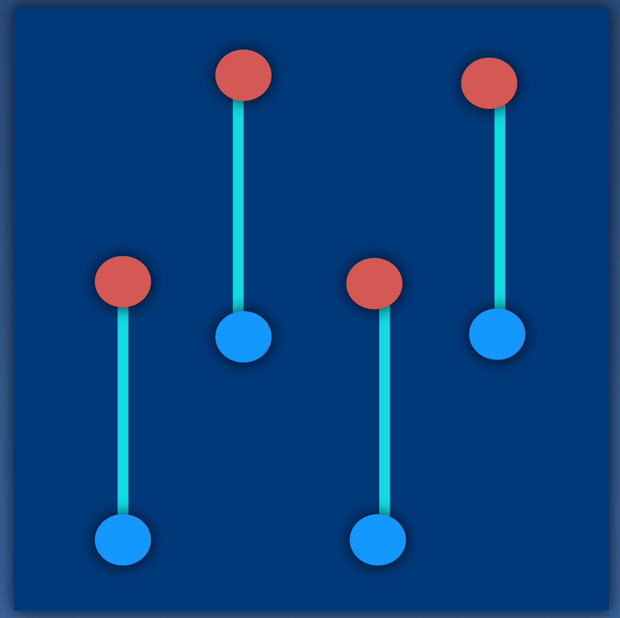
Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group



- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)

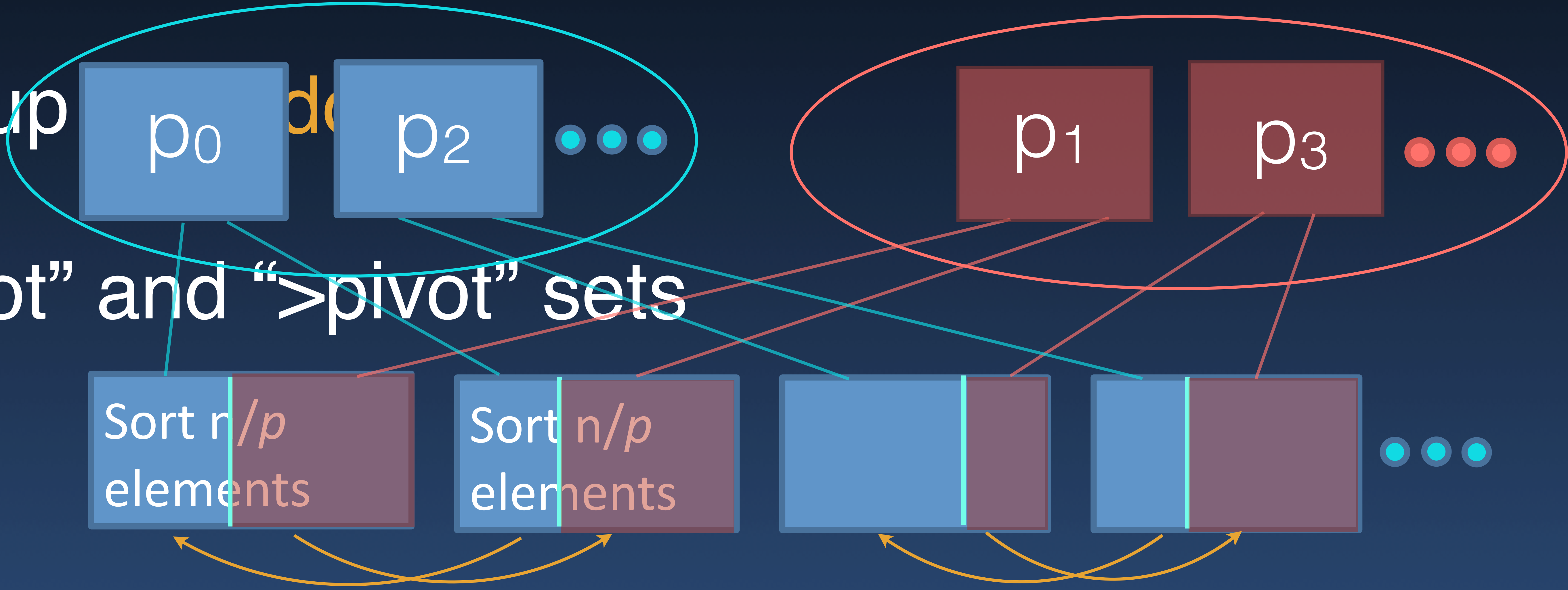


- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

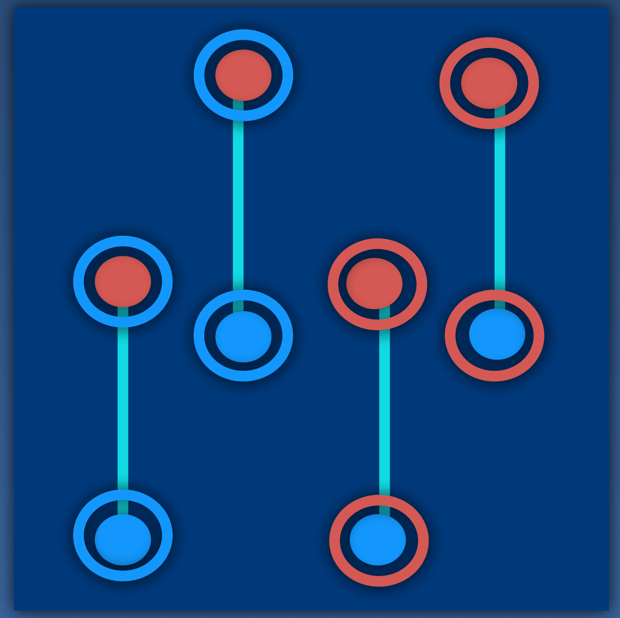
Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group



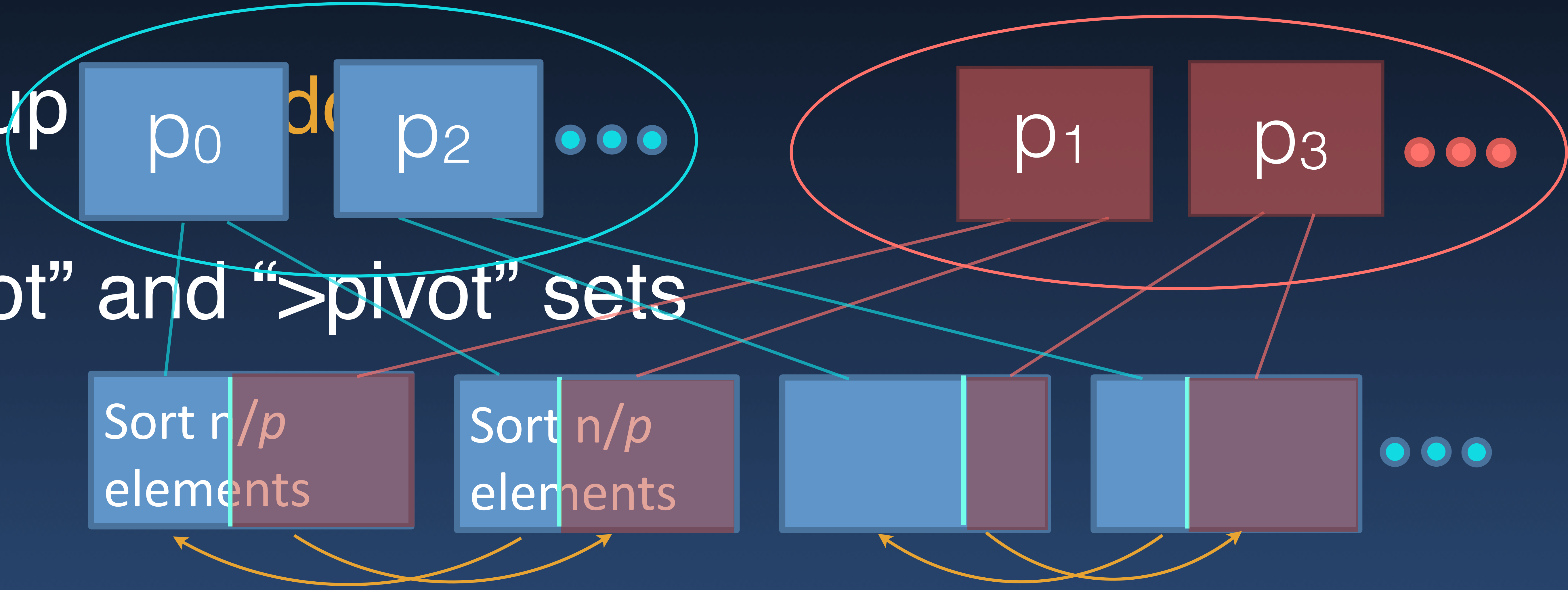
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)



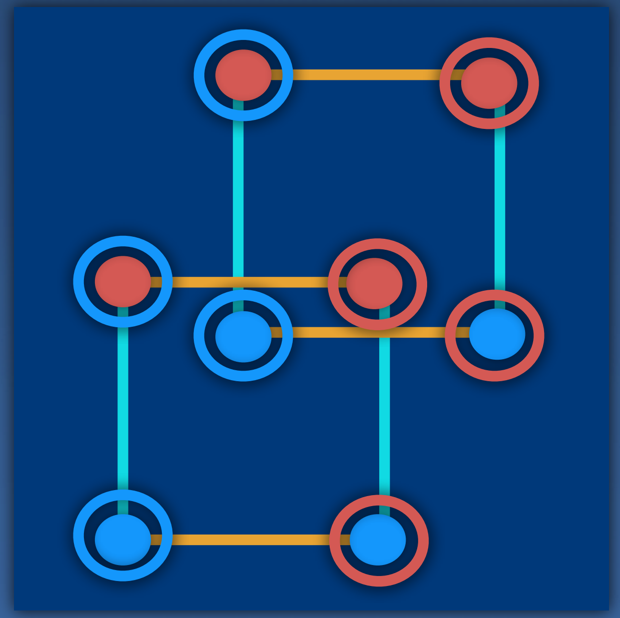
- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)



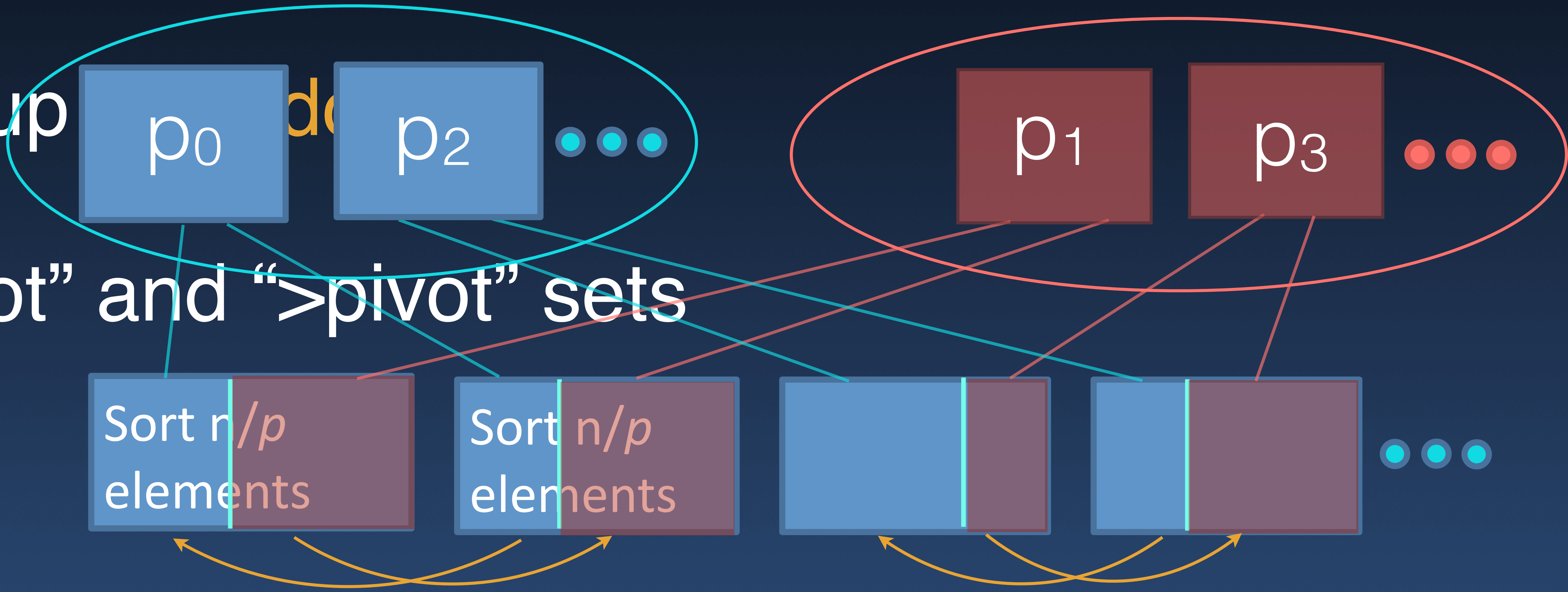
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)



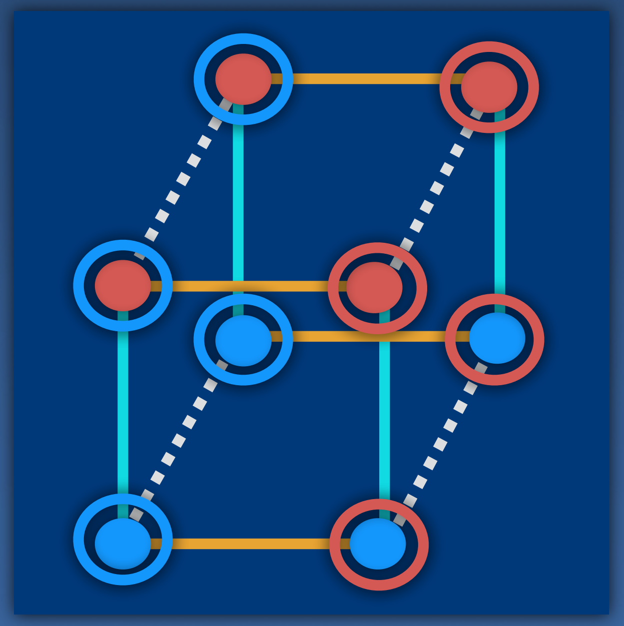
- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group



- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)

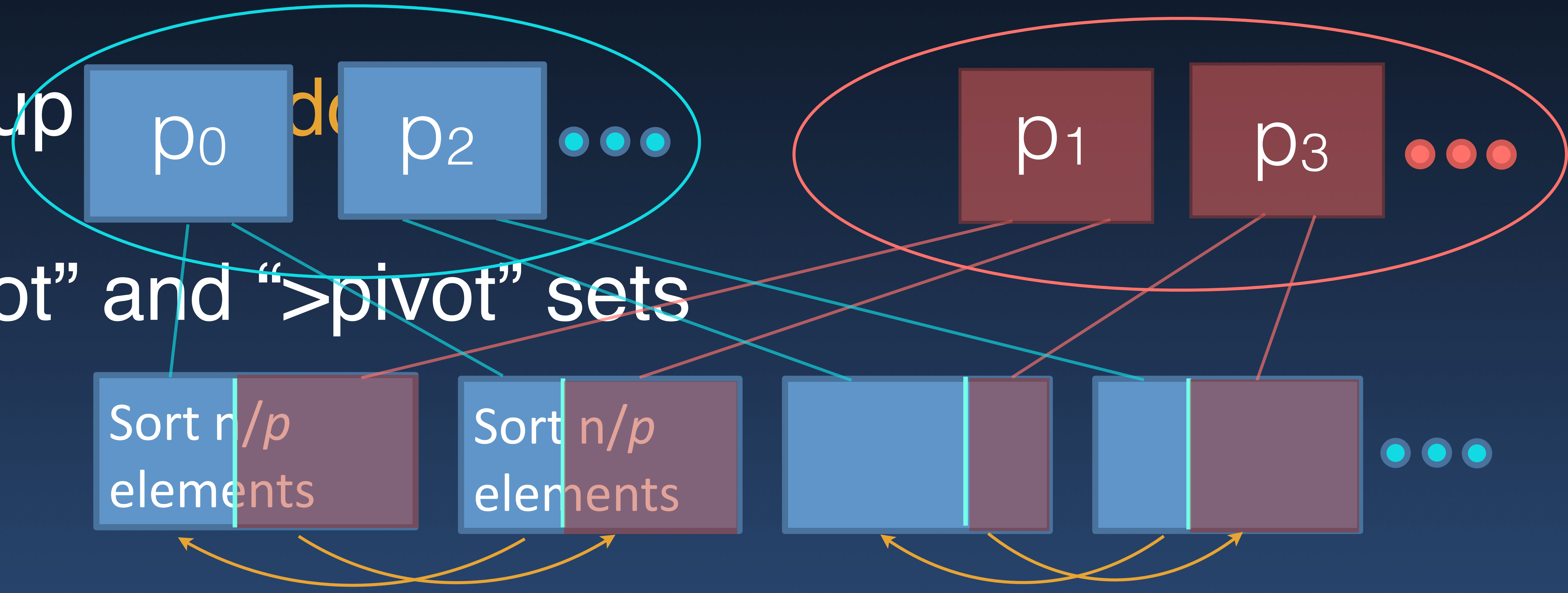


- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

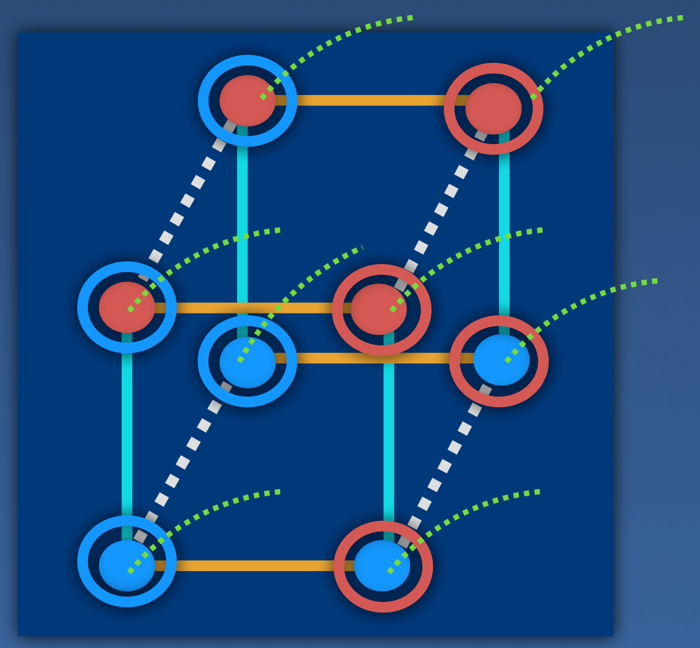
Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)



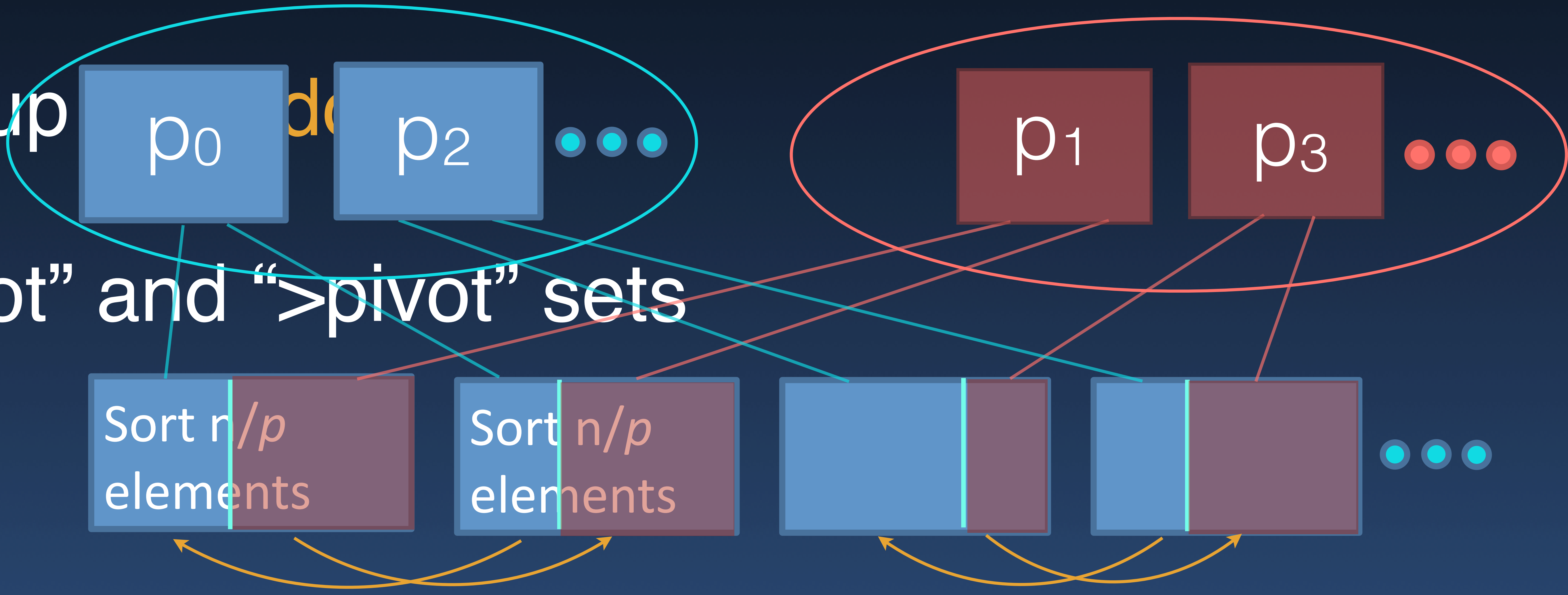
- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)
- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors



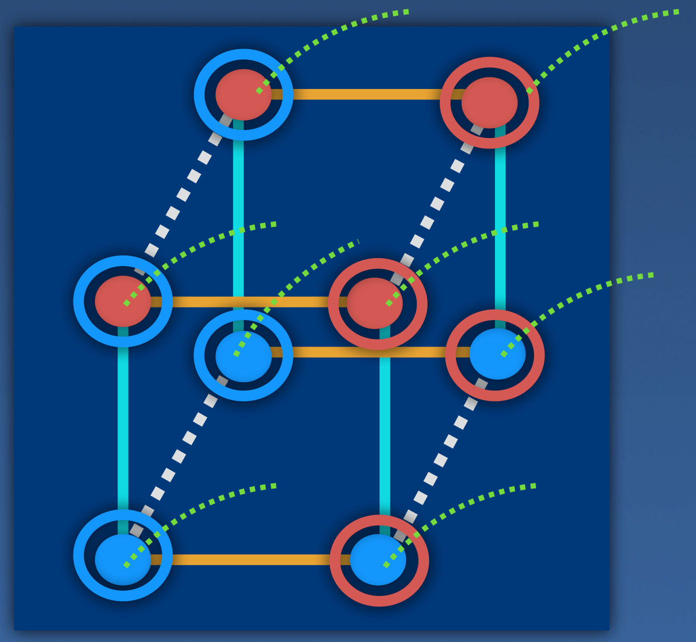
Subodh Kumar

Hyper Quick Sort

- Partition into p groups
  - ➡ Sort each group independently in O(n/p log n) time
- Pivot = median of any one group
- Partition each group into “<pivot” and “>pivot” sets
  - ➡ Binary search for pivot in (log n)



- Merge p/2 “<pivot” pairs, and p/2 “>pivot” pairs (pair-wise exchange)
  - ➡ Each sequentially: O(n/p)



O(n/p log n) expected

- Low = p/2 “<pivot” lists; High = p/2 “>pivot” lists
  - ➡ Recurse on Low with p/2 processors and High with p/2 processors

Parallel Bucket Sort

- Divide the range [a,b] of numbers into p sub-ranges
- Partition input into p equal-sized blocks
- Processor p_j sorts the elements in block j into p bins
  - ➡ bin k for sub-range k
  - ➡ “Send” $\boldsymbol{i}^{th}$ bucket to $\mathbf{p}_i$
- p_i collects bucket $\boldsymbol{i}$ from all processors
  - ➡ For uniformly distributed input, expected bucket size is uniform
- Processor p_i locally sorts bucket $\boldsymbol{i}$

Parallel Bucket Sort

- Divide the range [a,b] of numbers into p sub-ranges
- Partition input into p equal-sized blocks

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

- Processor p_j sorts the elements in block j into p bins
    - bin k for sub-range k
    - "Send" $\boldsymbol{i}^{th}$ bucket to p_i
- p_i collects bucket $\boldsymbol{i}$ from all processors
    - For uniformly distributed input, expected bucket size is uniform
- Processor p_i locally sorts bucket $\boldsymbol{i}$

Real risk of load imbalance

Subodh Kumar

Parallel Bucket Sort

- Divide the range [a,b] of numbers into p sub-ranges
- Partition input into p equal-sized blocks

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

- Processor p_j sorts the elements in block j into p bins
  - ➡ bin k for sub-range k
  - ➡ "Send" i^{th} bucket to p_i

Choose evenly separated splitters from data for bucketing
  Choose a sample of sufficient size s
  Sort the samples
  Choose **B-1** evenly spaced samples

- p_i collects bucket i from all processors
  - ➡ For uniformly distributed input, expected bucket size is uniform
- Processor p_i locally sorts bucket i

Real risk of load imbalance

Subodh Kumar

Parallel Splitter Selection

- Partition n elements equally into B blocks
- (Quick)Sort each block
- From each sorted block:
  - ➡ Choose B-1 evenly spaced **samples**
- Sort B*(B-1) samples
  - ➡ Choose B-1 evenly spaced **splitters**
- Arrange elements by bucket in output array
  - ➡ No bucket contains more than **2*n/B** elements

Parallel Splitter Selection

- Partition n elements equally into B blocks
- (Quick)Sort each block (n/B log n/B)
- From each sorted block:
  - ➡ Choose B-1 evenly spaced **samples**
- Sort B*(B-1) samples (B^2 log B)
  - ➡ Choose B-1 evenly spaced **splitters**
- Arrange elements by bucket in output array (n/B + B log B)
  - ➡ No bucket contains more than **2*n/B** elements

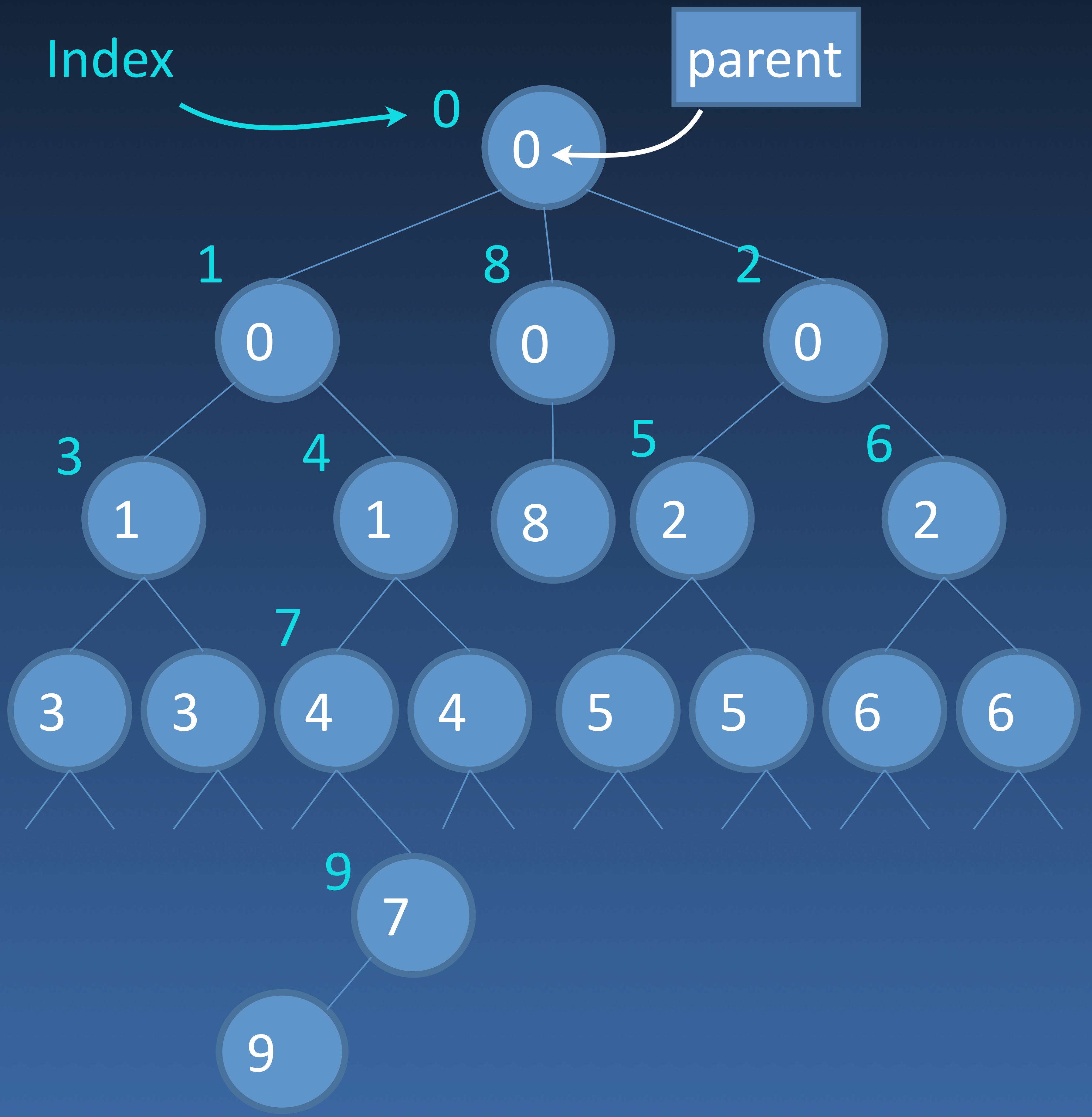
Subodh Kumar

Arrange Elements

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

- For each input element
  - ➡ Find its bucket: Binary search Splitters
- Count the number of elements in each bucket
  - ➡ Find destination using B-1 prefix sums
  - ➡ Destination[i] = exclusive psum[i]
- Reserve space per bucket
- For each input element
  - ➡ Write in element i output bucket at Destination[i]

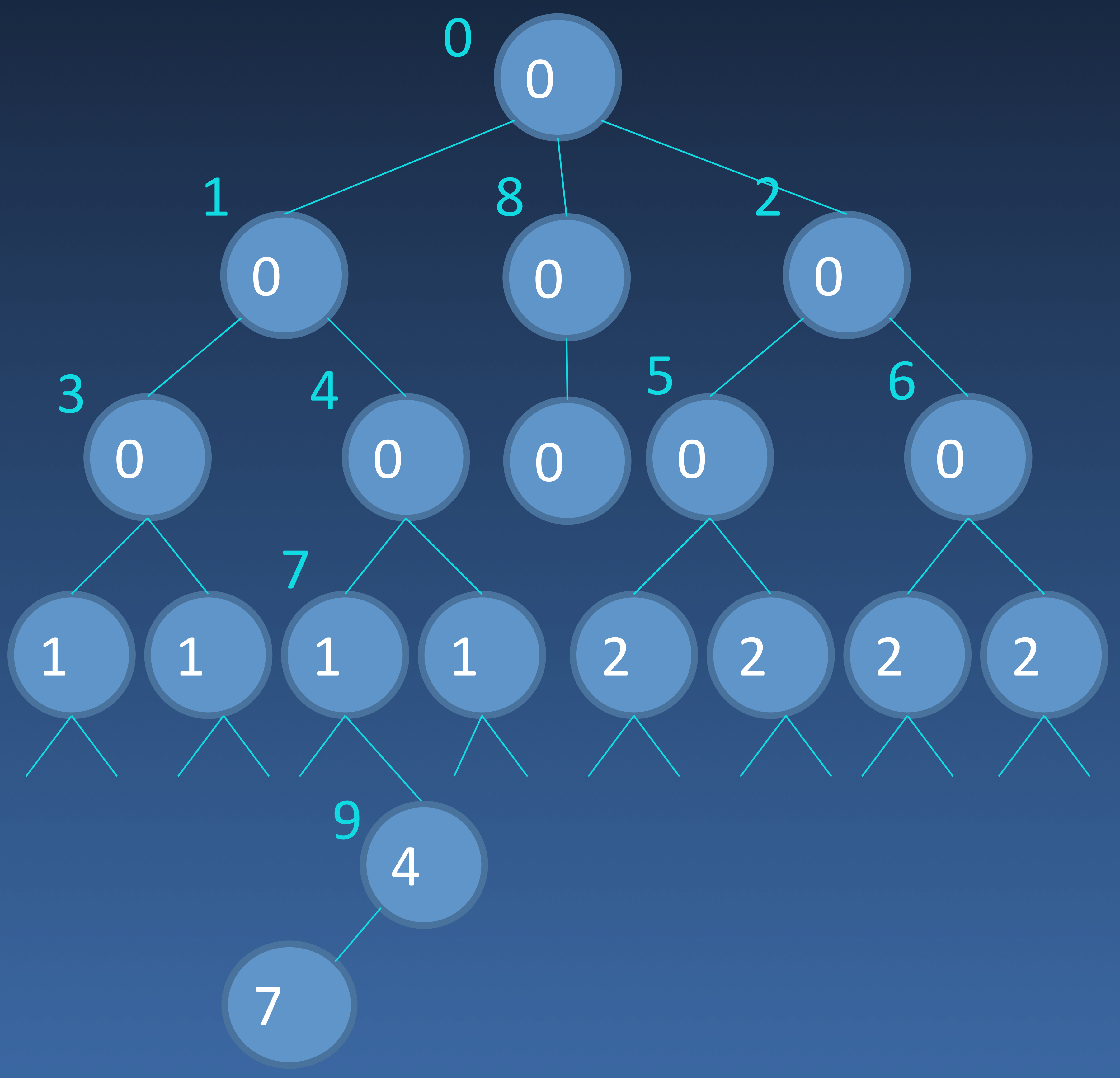
Find Roots in a Forest



- p(i): parent of node i
- Do in parallel
  - p(i) = p(p(i))
- Stop if p(i) = i
- Time: log(height)
- Work: n log(height)

Subodh Kumar

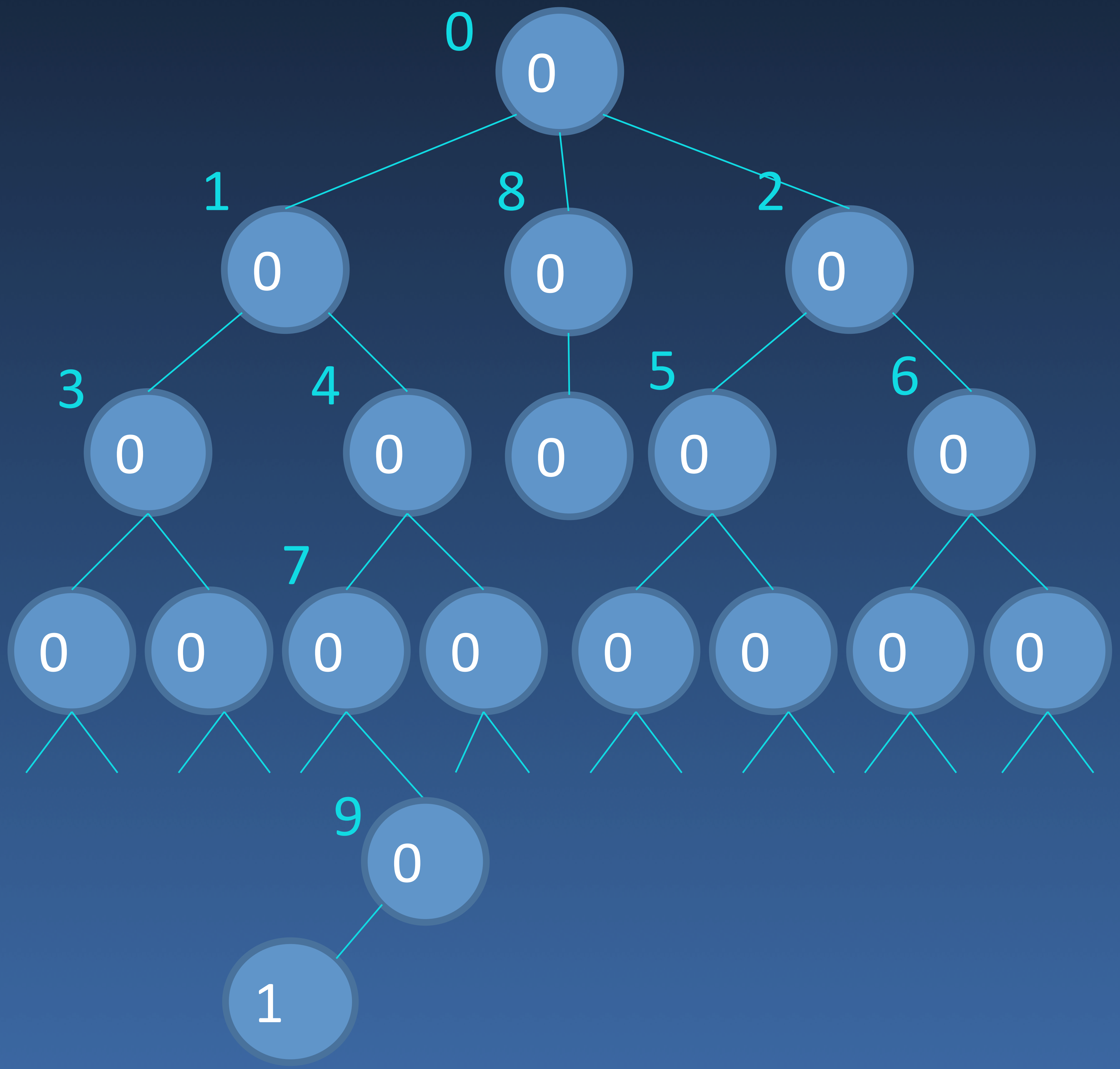
Find Roots in a Forest



- p(i): parent of node i
- Do in parallel
  - p(i) = p(p(i))
- Stop if p(i) = i
- Time: log(height)
- Work: n log(height)

Subodh Kumar

Find Roots in a Forest



- p(i): parent of node i
- Do in parallel
  - p(i) = p(p(i))
- Stop if p(i) = i
- Time: log(height)
- Work: n log(height)

Subodh Kumar

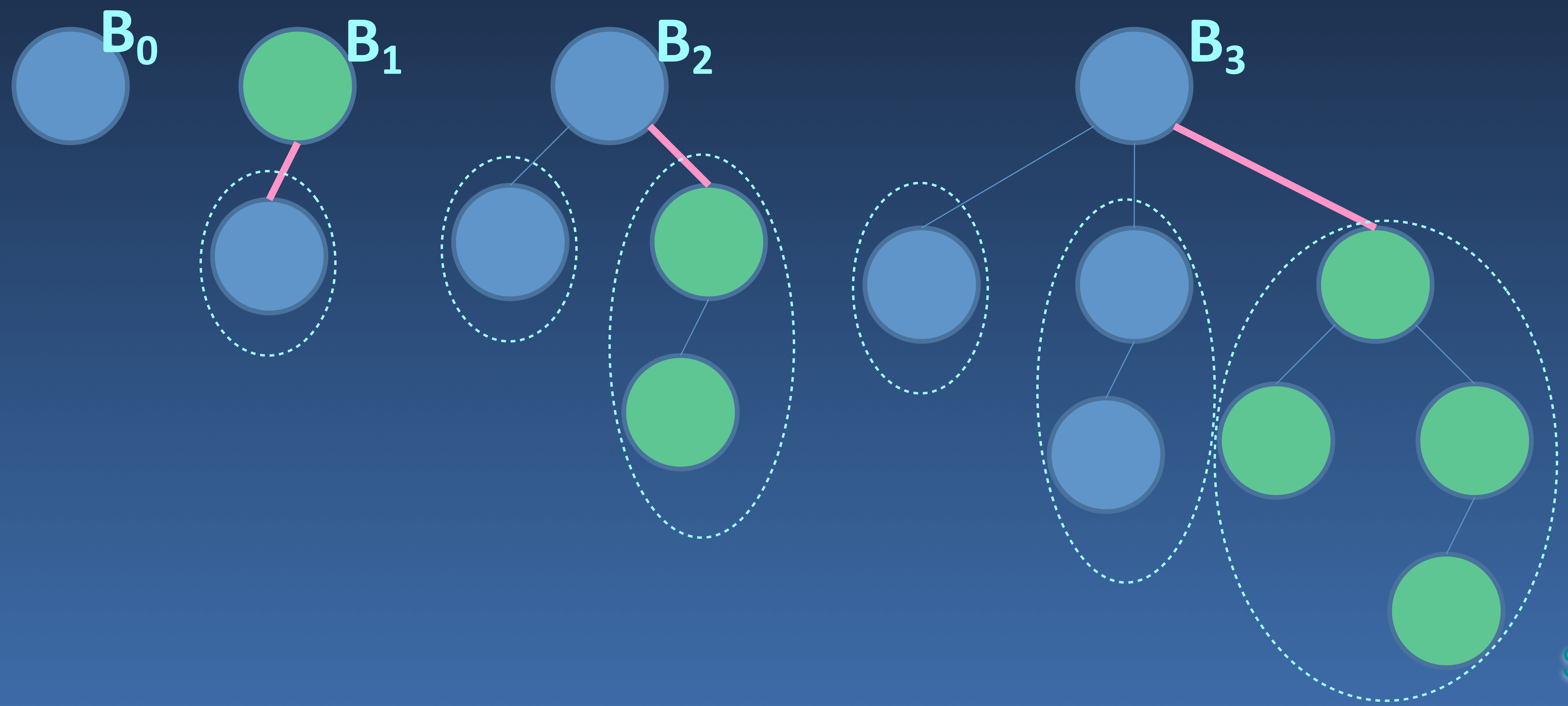
- Progressively push computation to all elements at a given distance
  - ➡ Doubling the distance at each step
  - ➡ After k steps the computation has been performed for elements within a distance of 2k
- Applies to array, list, tree

- Linked List, Skip lists, Trees, Heap
- Parallelize single operation
- Parallelize *k* operations
- Concurrent operations
  - ➡ Lock at low granularity
  - ➡ Lock-free operations
  - ➡ Lazy deletions
- Modify algorithm to avoid serializing data structures

Binomial Tree

- B_0: single node (Root)
- B_k: Root with k binomial subtrees, B_0 ... B_{k-1}

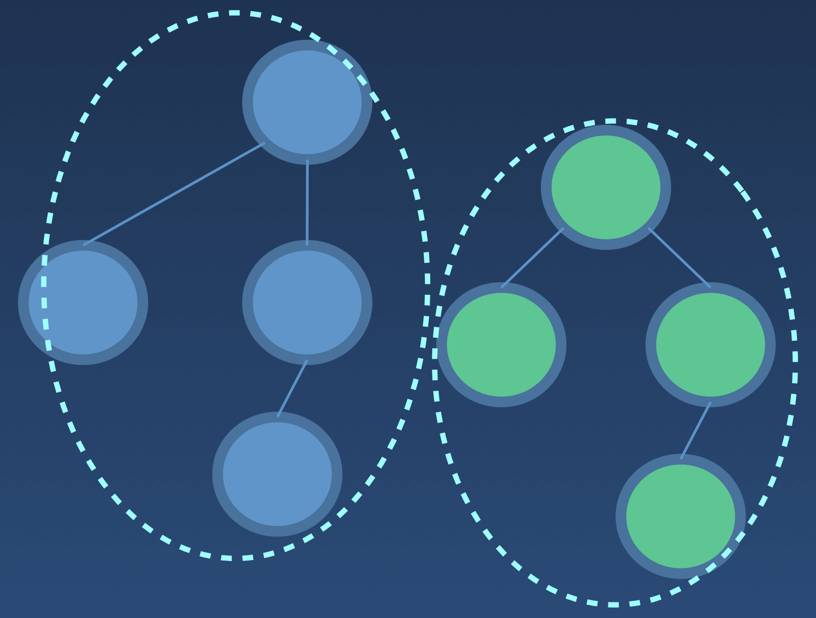
- B_0: single node (Root)
- B_k: Root with k binomial subtrees, B_0 ... B_{k-1}



- A binomial tree of rank r contains exactly 2^r nodes
- A node of rank r contains one son of each rank i, $0 <= i < r$
- Heap ordered: node-key ≥ parent-key
- A forest of binomial trees
- Exactly 1, 2, or 3 trees of each rank up to 1+log n
- The minimum root of rank i is smaller than the roots of higher rank
  - ➡ Minimum root of rank 0 is the minimum element

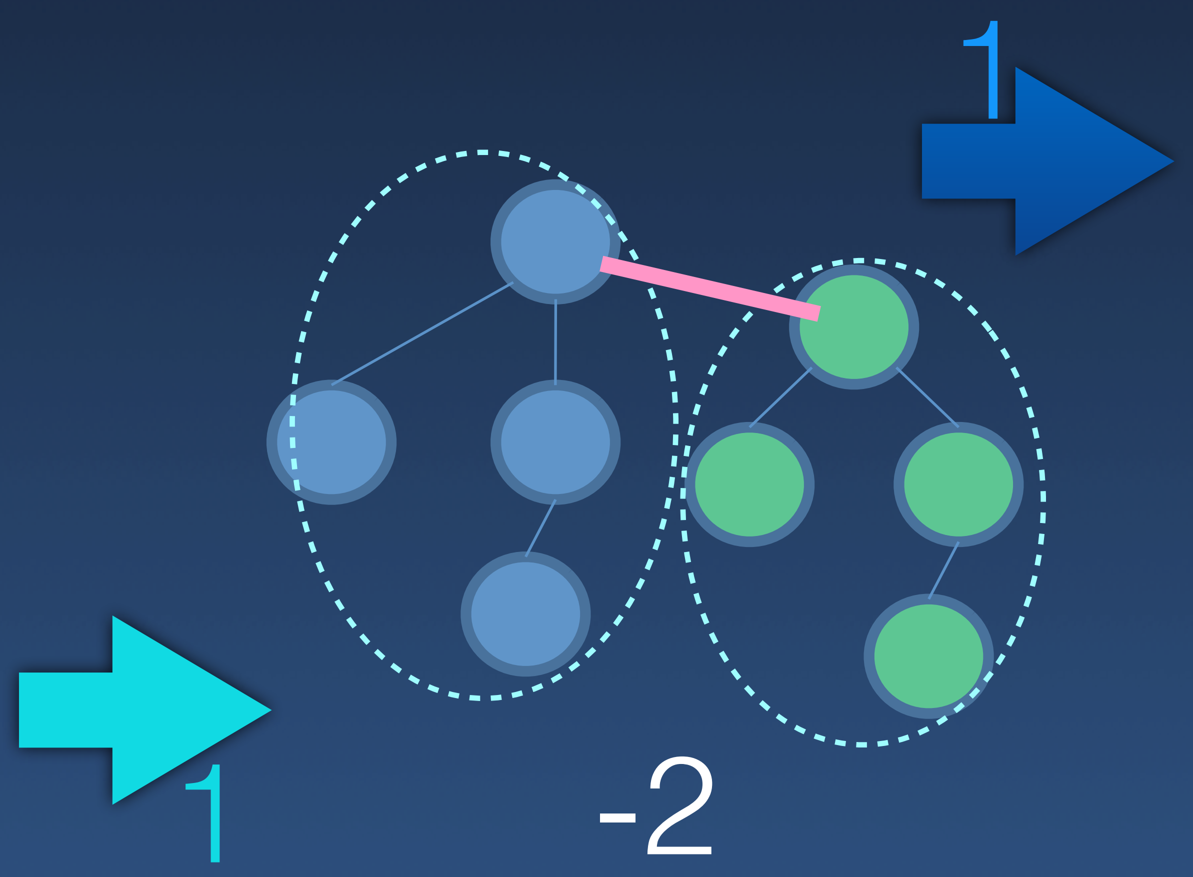
Forest Operation: PARLINK

- for each i
  - ➡ If there are ≥3 roots
    - Link two <u>non-minimum</u> trees to make a tree of rank i+1
  - ➡ If there were 3 (or more) of this rank
    - the count decreases by at least 1
  - ➡ Otherwise, the count increases by at most 1



Forest Operation: PARLINK

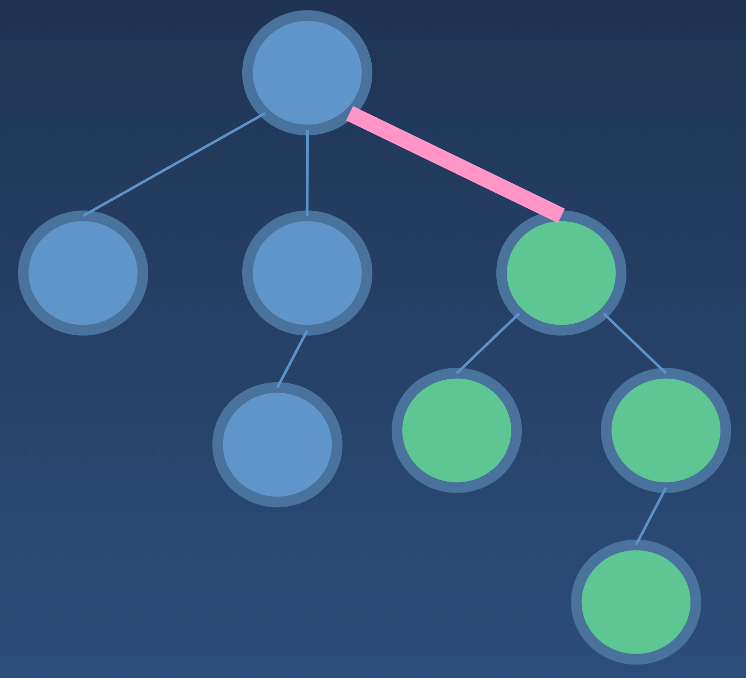
- for each i
  - ➡ If there are ≥3 roots
    - Link two <u>non-minimum</u> trees to make a tree of rank i+1
  - ➡ If there were 3 (or more) of this rank
    - the count decreases by at least 1
  - ➡ Otherwise, the count increases by at most 1



O(1), O(log n) processors

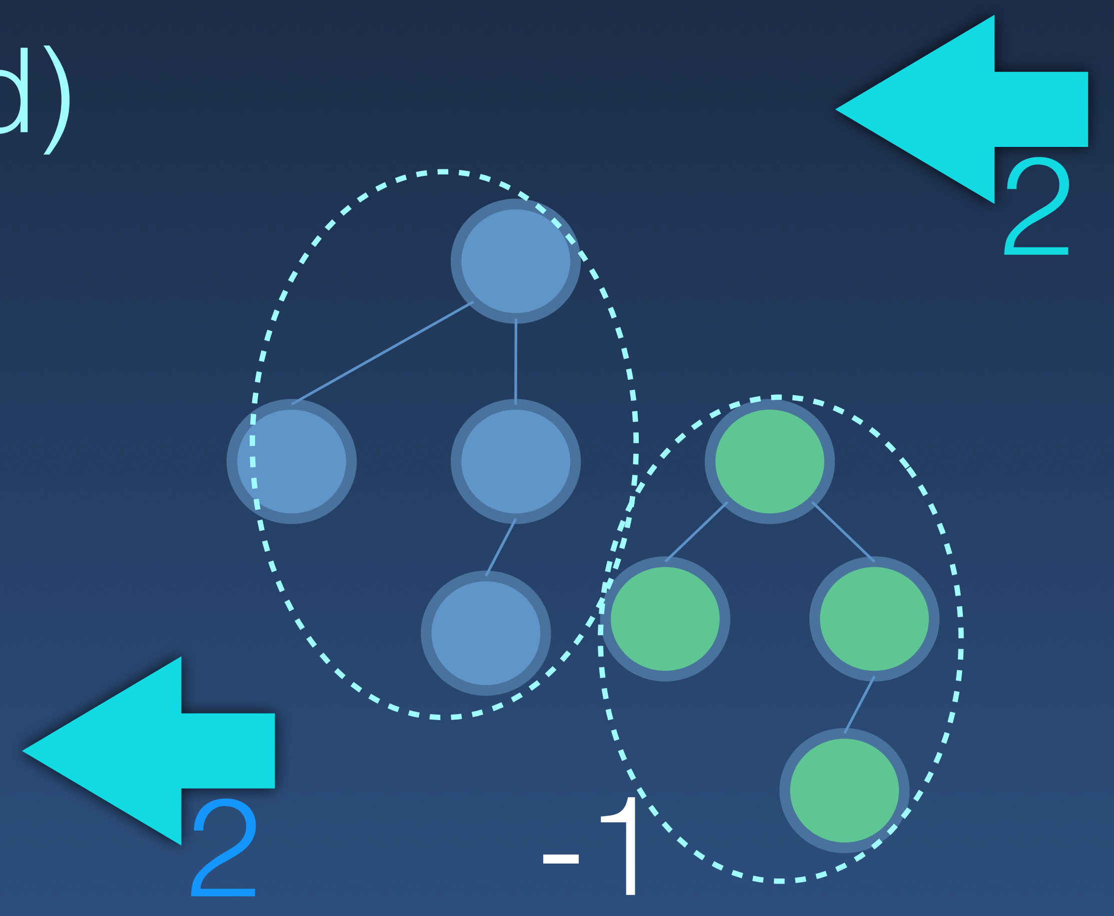
Forest Operation: PARUNLINK

- for each i
  - ➡ Unlink the <u>minimum</u> root for each rank *i* (from its rank i-1 child)
    - creating two trees of rank *i*-1
  - ➡ The number of trees of rank i increases by at most 1
    - Except rank 0, which can increase by 2
  - ➡ The new minimum for rank i ≤ old minimum at rank i+1



Forest Operation: PARUNLINK

- for each i
  - ➡ Unlink the minimum root for each rank *i* (from its rank i-1 child)
    - creating two trees of rank *i*-1
  - ➡ The number of trees of rank i increases by at most 1
    - Except rank 0, which can increase by 2
  - ➡ The new minimum for rank i ≤ old minimum at rank i+1



O(1), O(log n) processors

- INSERT
  - ➡ Create a new tree of rank 0 with the element
  - ➡ PARLINK
- EXTRACTMIN
  - ➡ Remove min root of rank 0
  - ➡ PARUNLINK
  - ➡ PARLINK
    - To ensure no more than 3 trees of any rank