COL380

Introduction to
Parallel & Distributed Programming

Agenda

- Parallel Algorithms
  - ➡ Merging
  - ➡ Minimum finding
  - ➡ Sorting

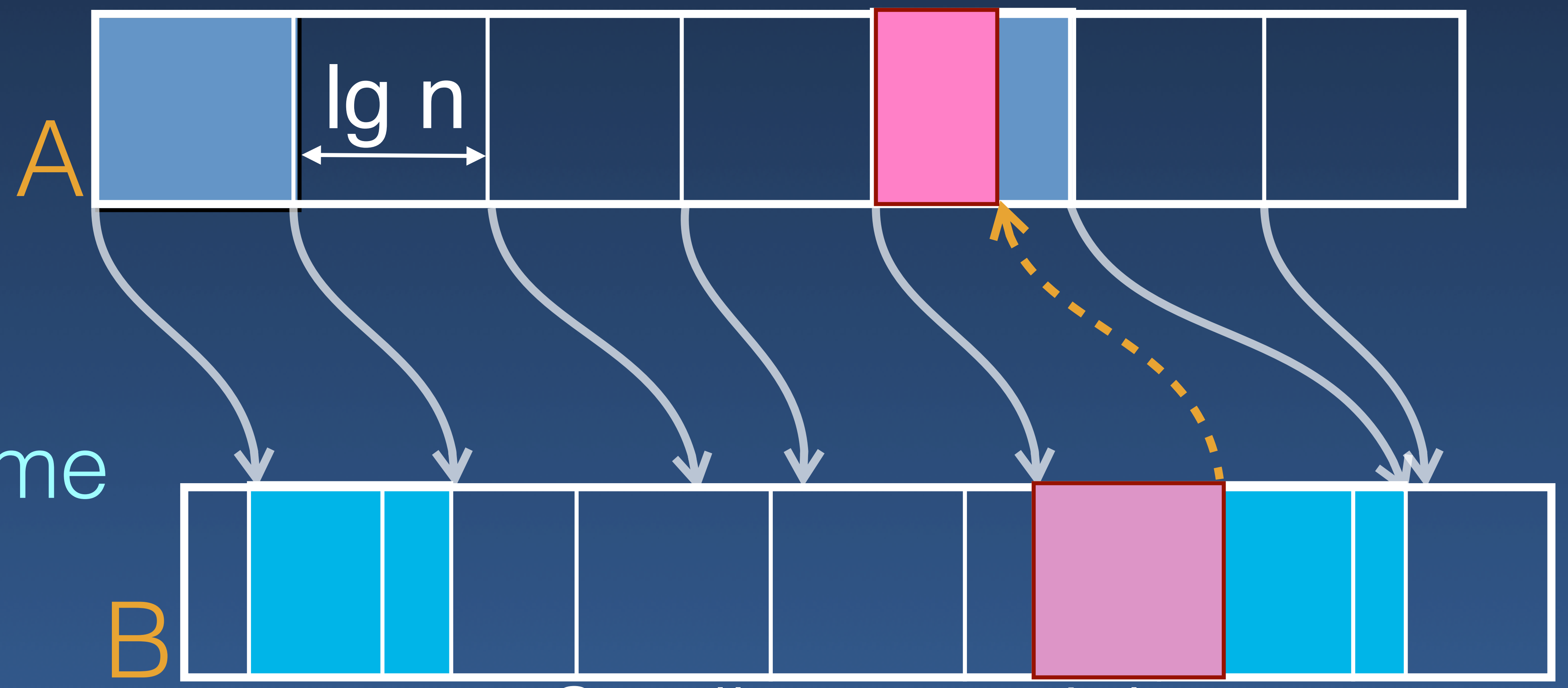
Towards Optimal Merge(A,B)

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

Total time is O(log n)
Total work is O(n)

limited by

Data Partitioning Technique

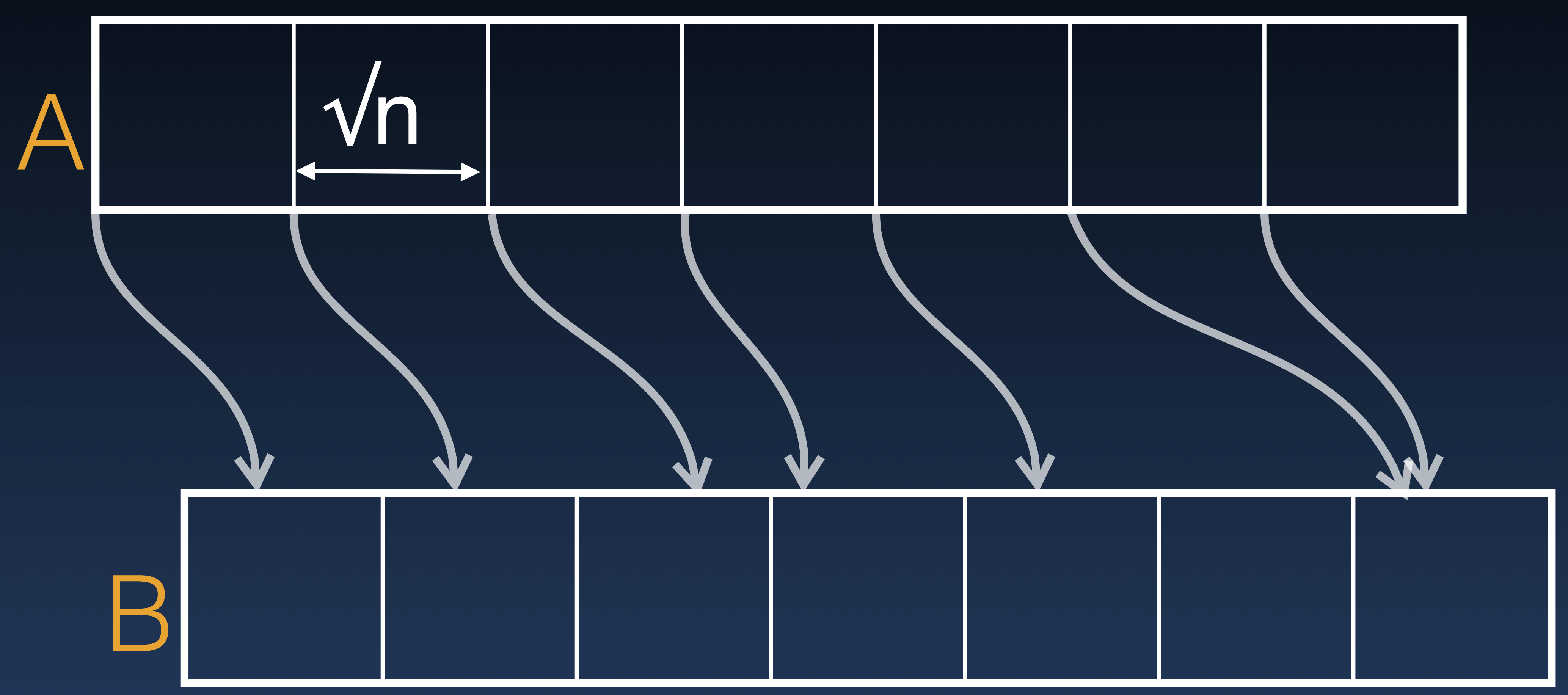


Fast Merge (A,B)

- Partition A and B into ***√n*** blocks each
- Select from A, elements $i\sqrt{n}$, $i \in [0: \sqrt{n})$

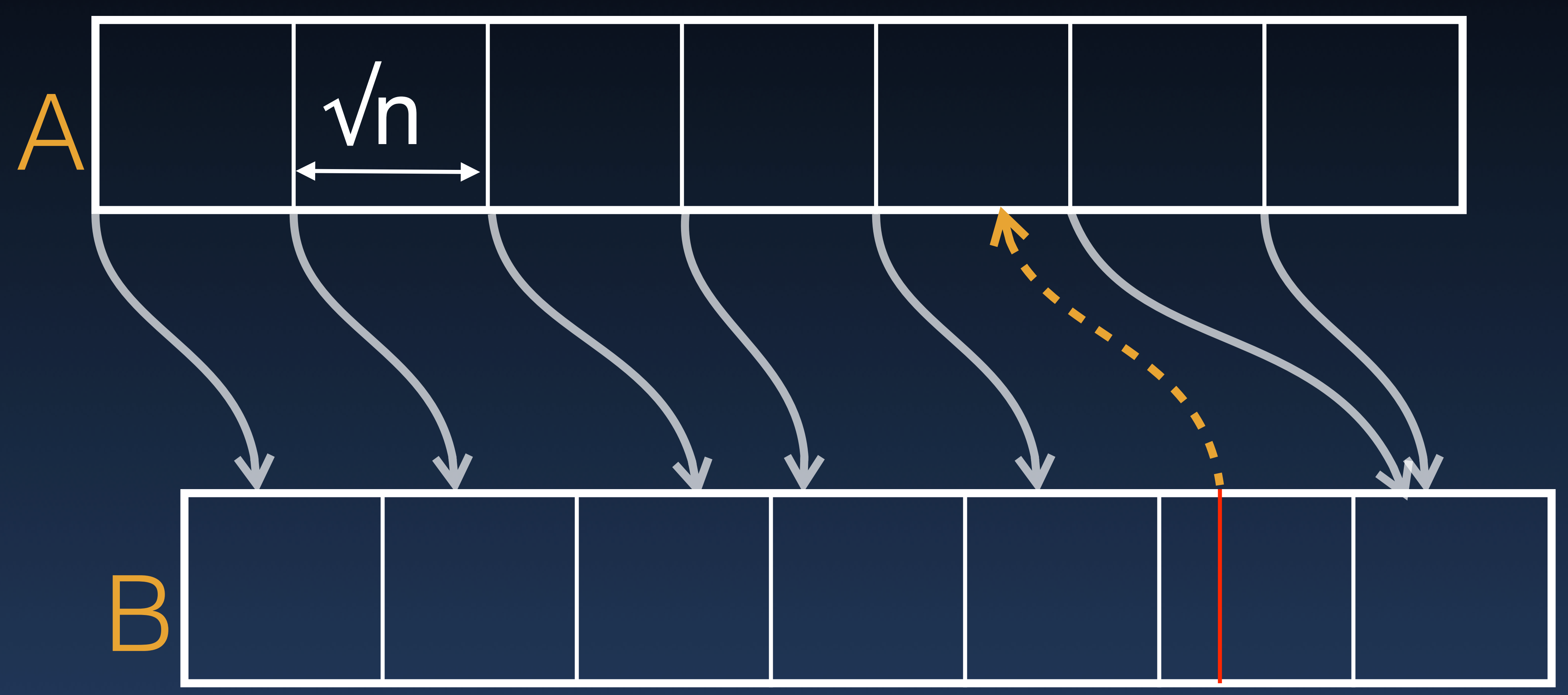
Fast Merge (A,B)

- 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



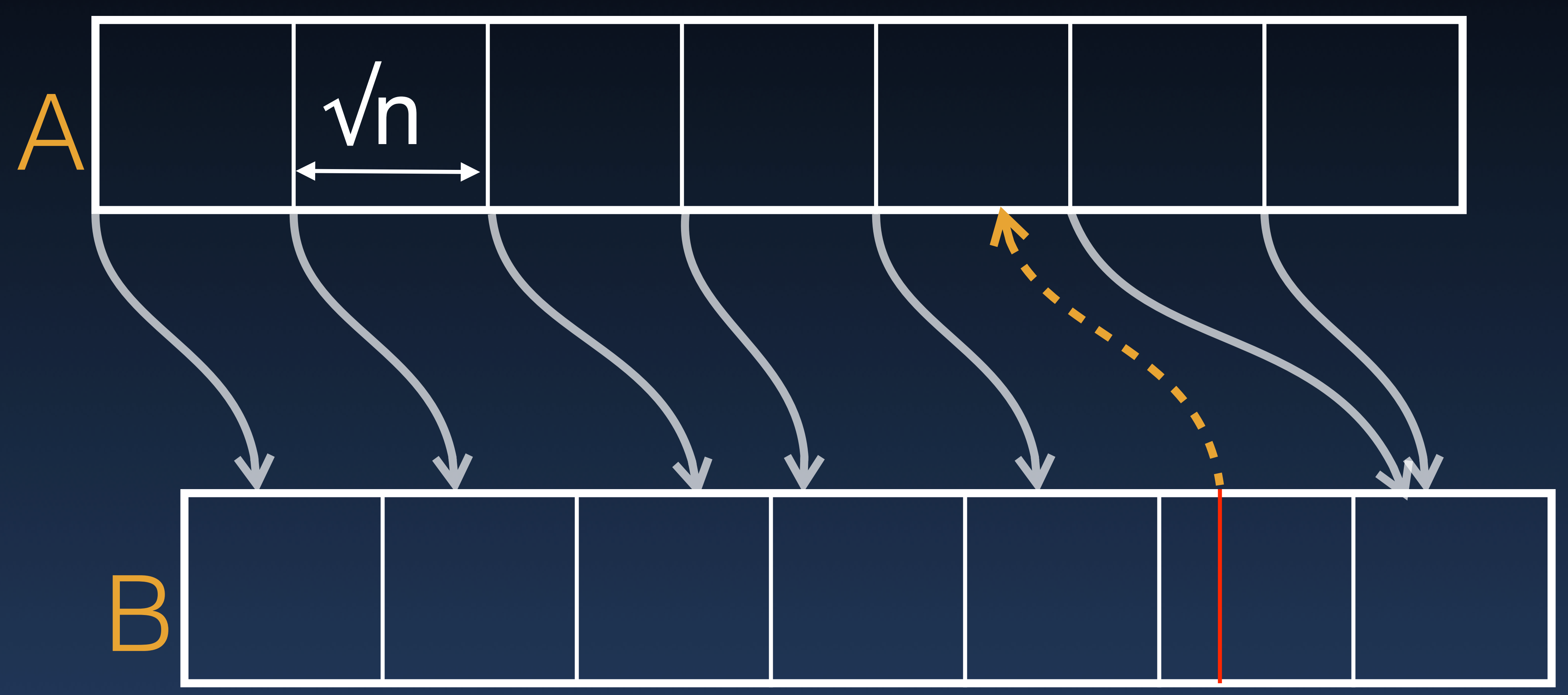
Fast Merge (A,B)

- 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



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
  - ➡ $\sqrt{n}$ Parallel searches, use $\sqrt{n}$ processors for each search
- Similarly rank $\sqrt{n}$ selected elements from B in A
- 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 too much work
Not work optimal

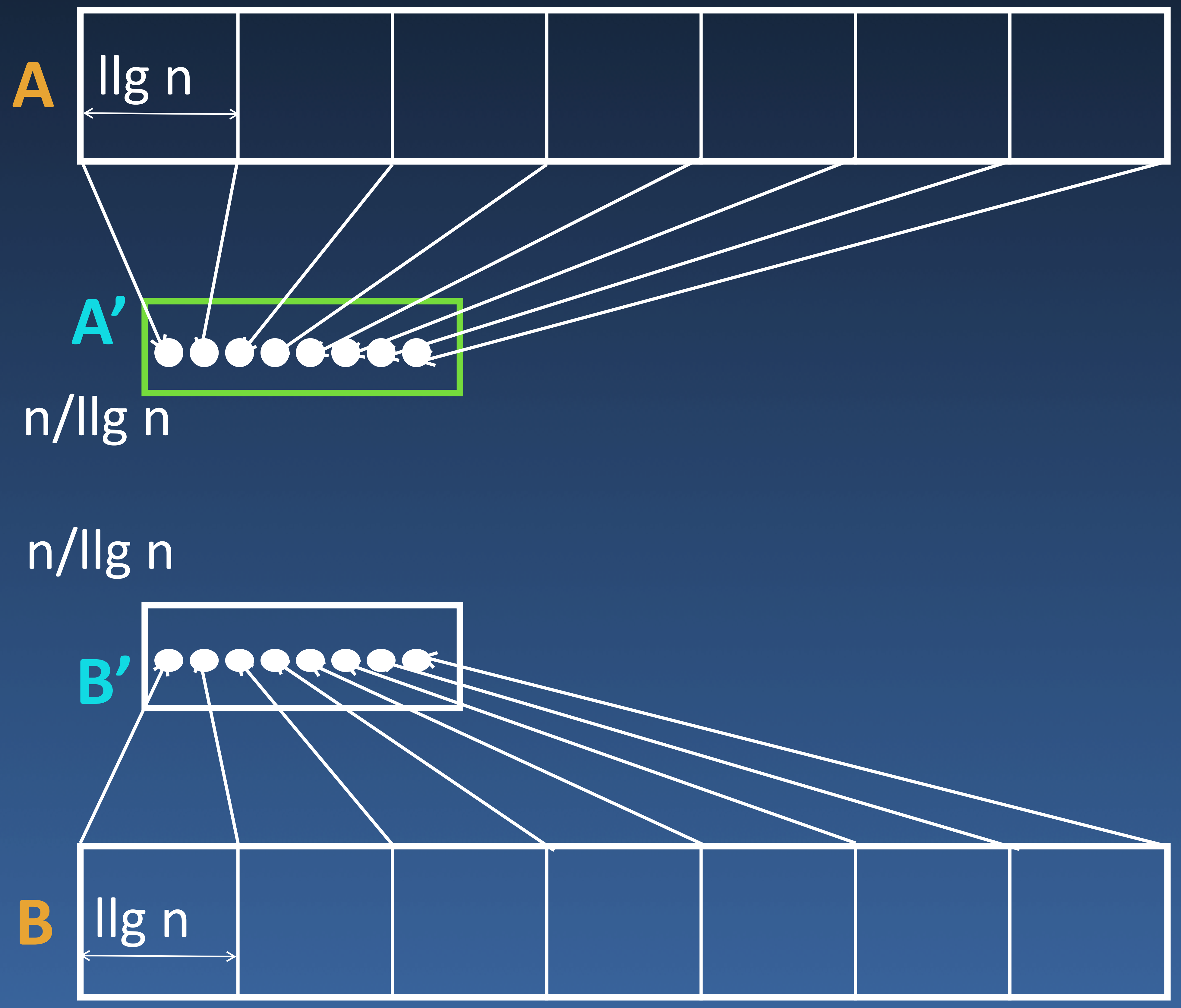
Subodh Kumar

Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..

Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$

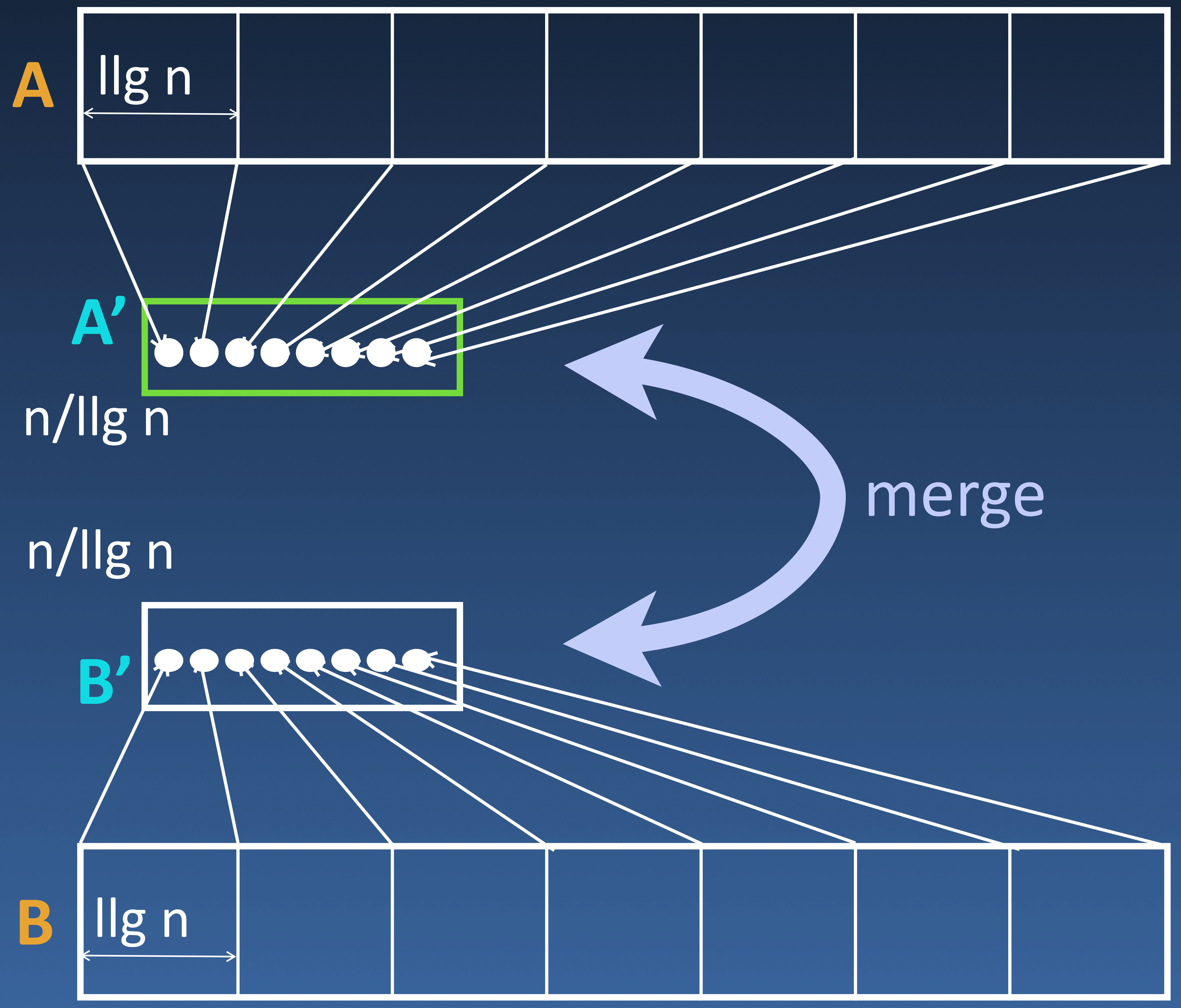


- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$

Subodh Kumar

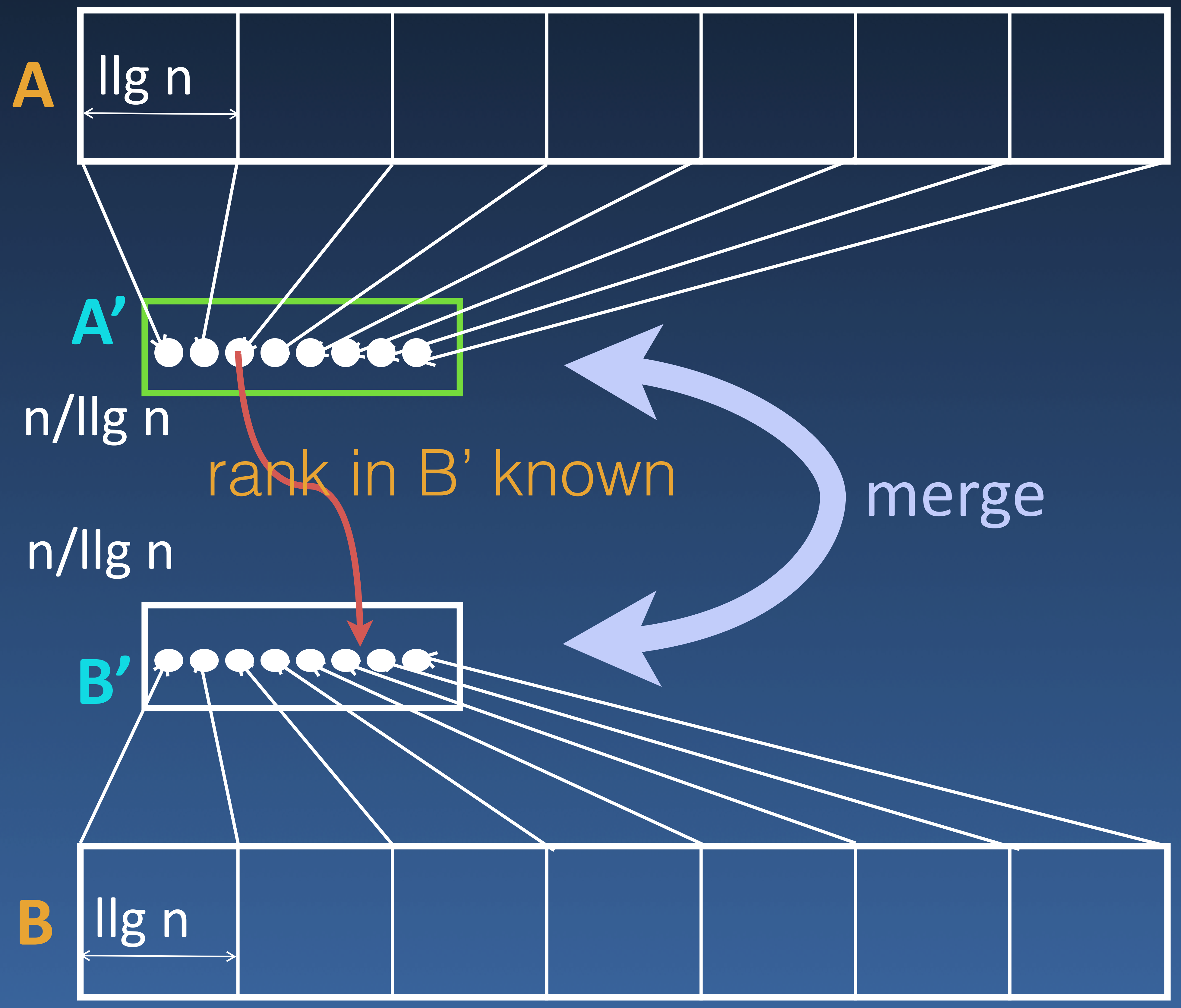
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$
- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$



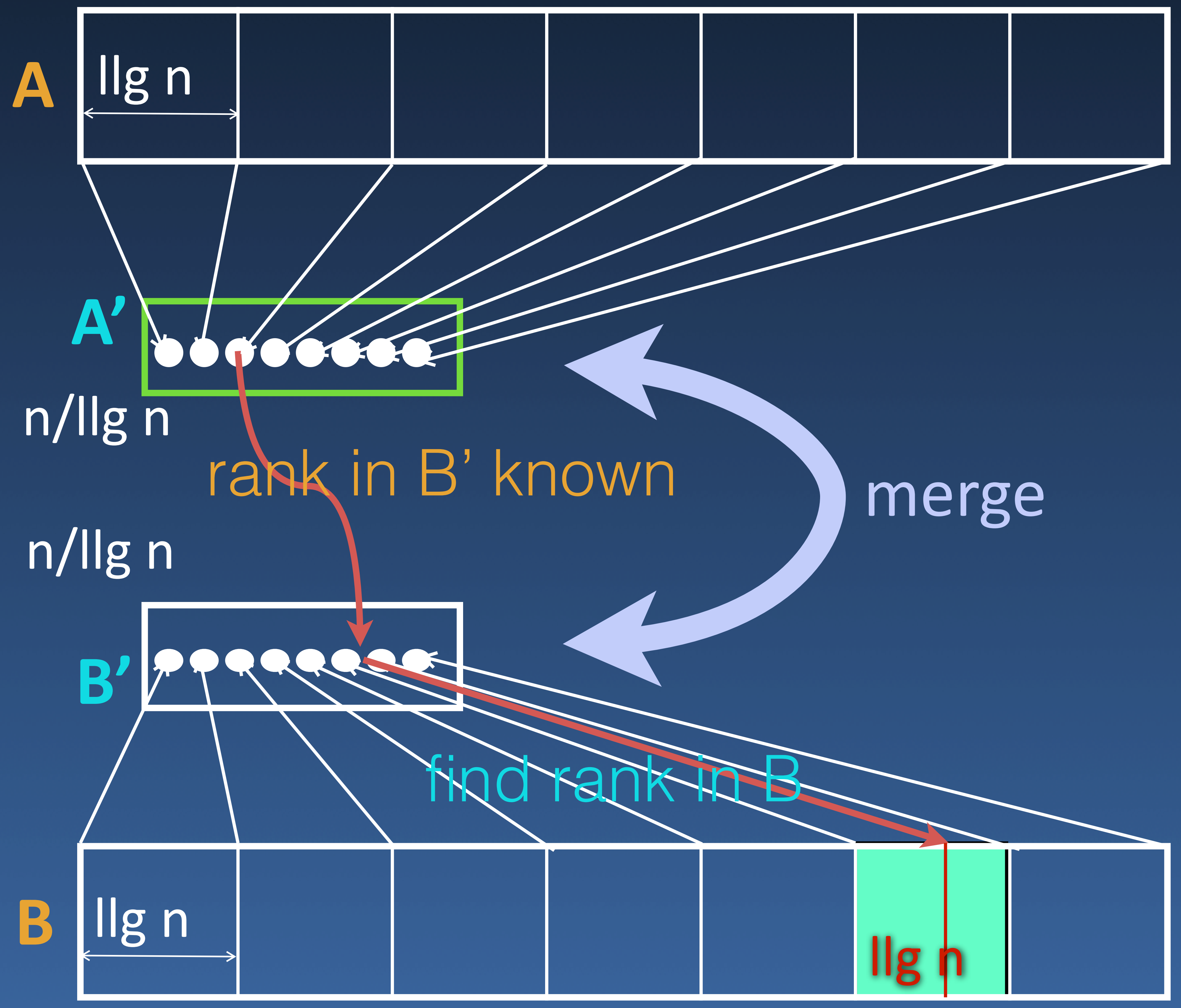
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$
- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$

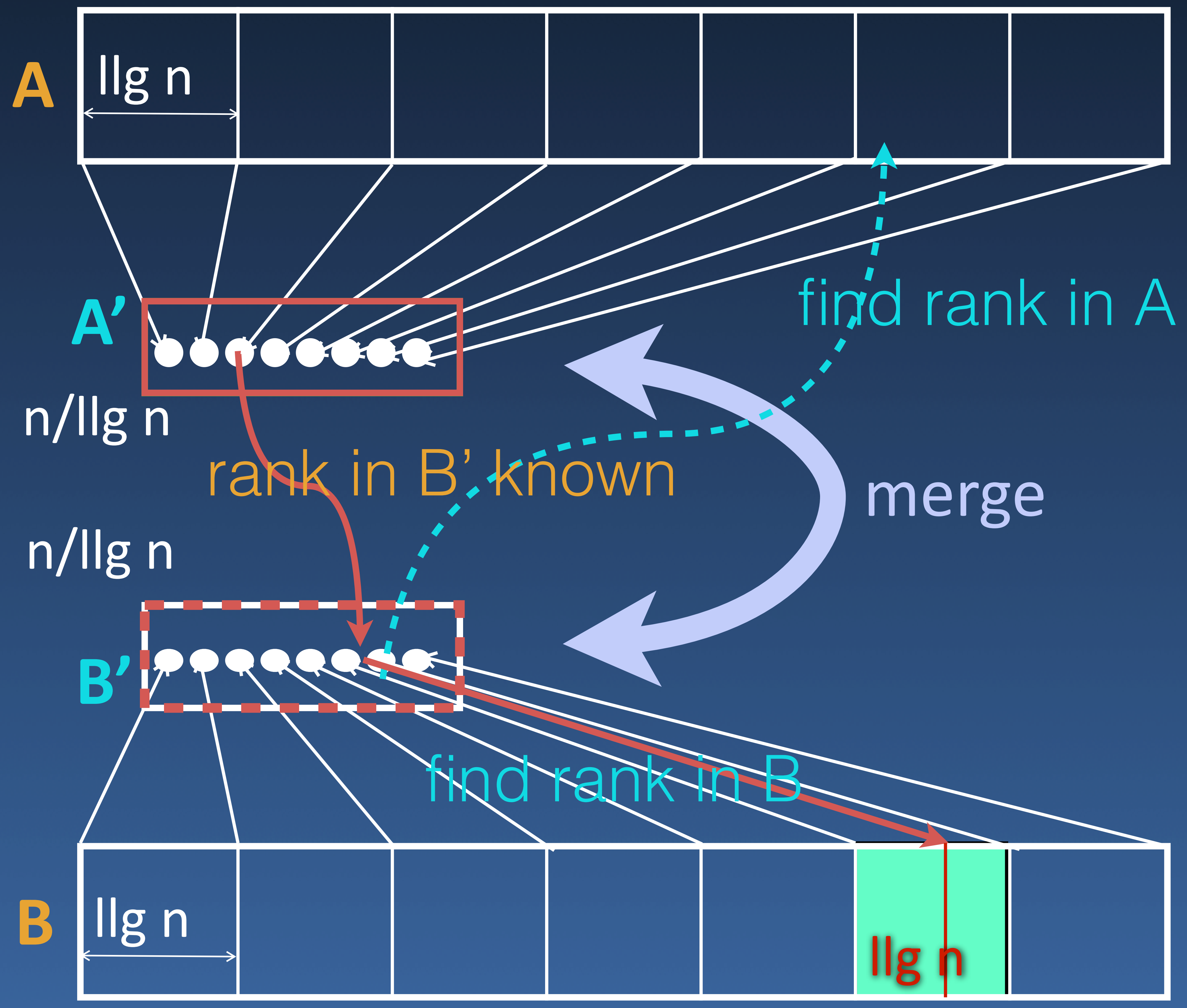


Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..

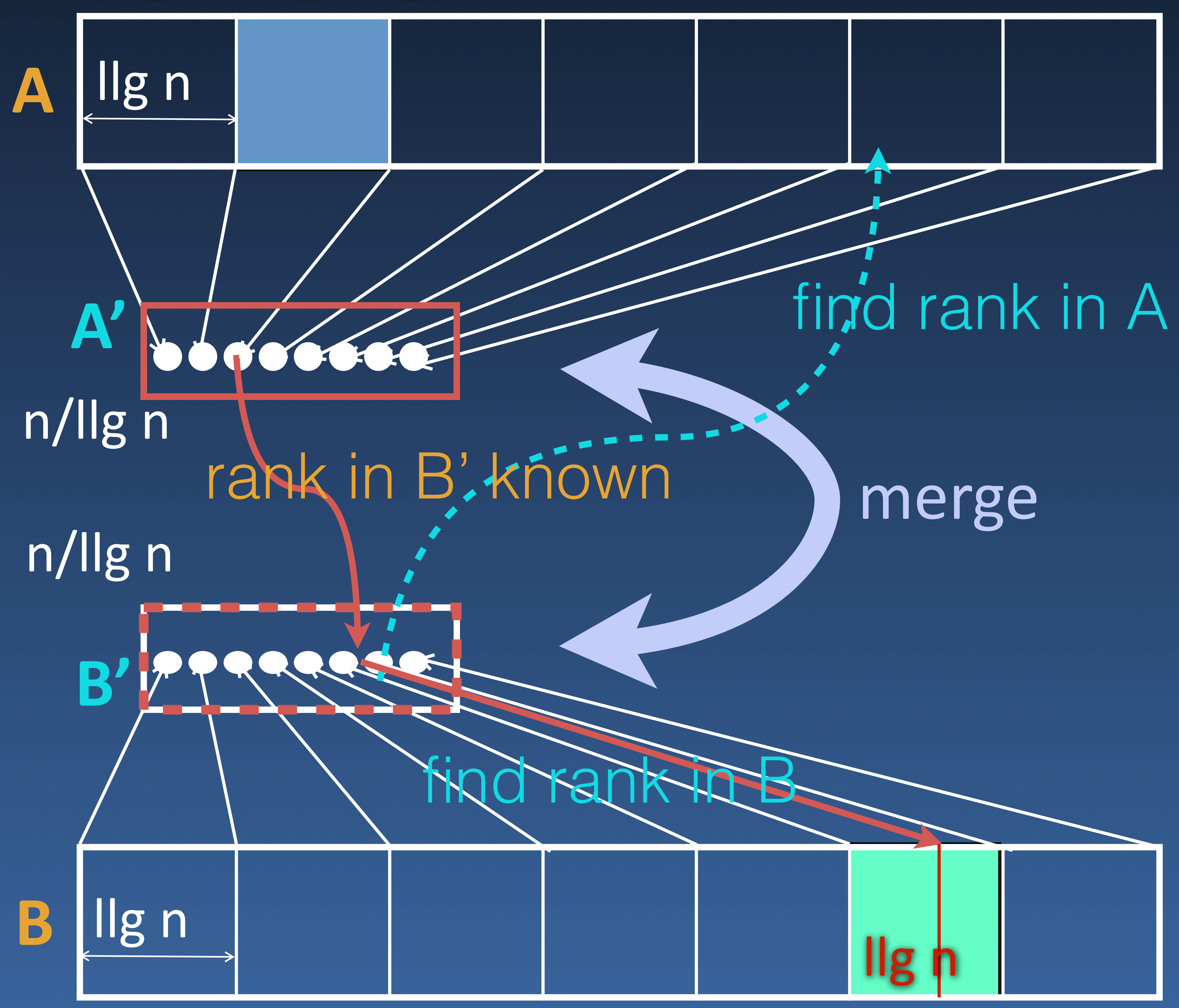


- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..



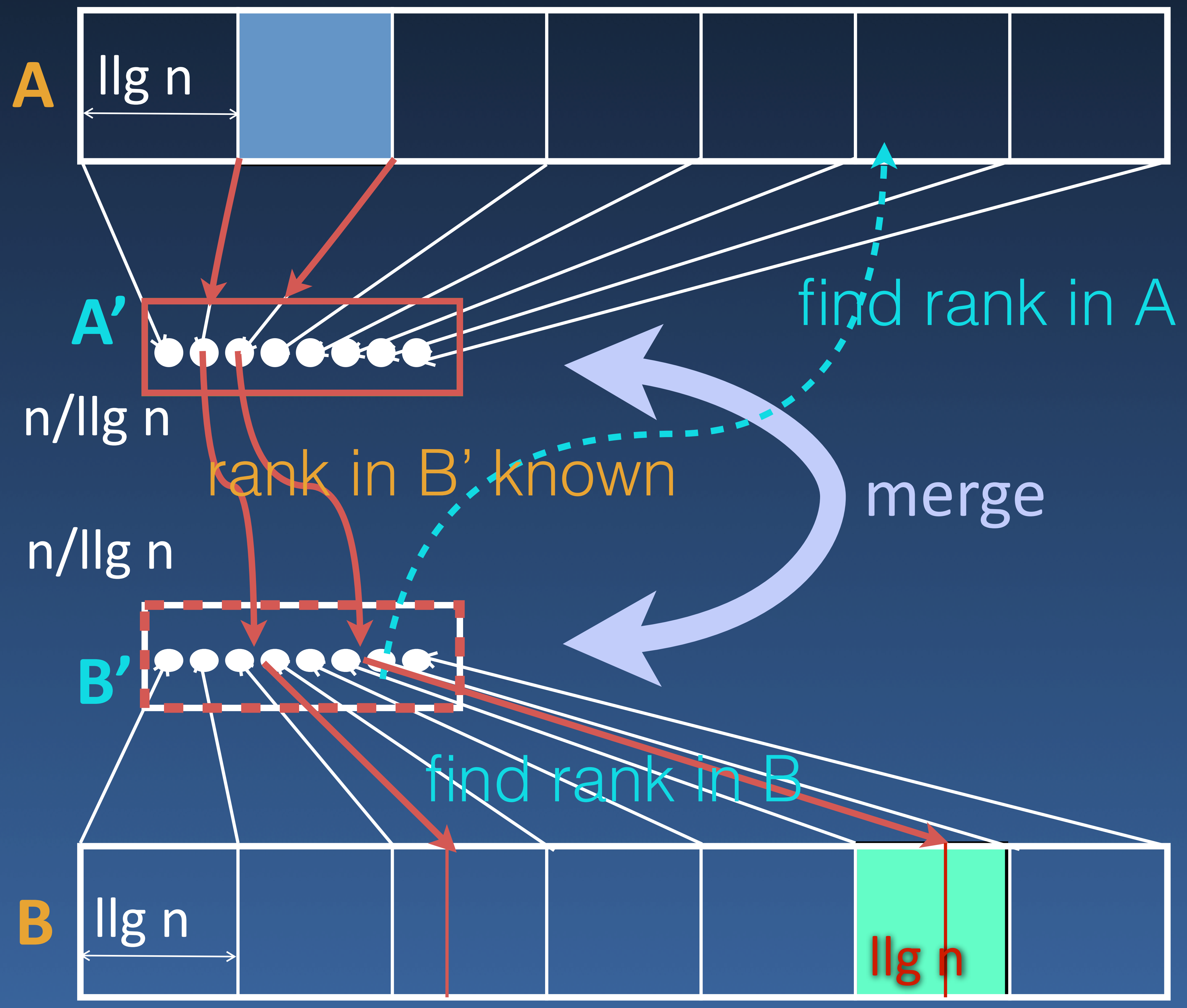
Optimal Merge (A,B)

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ 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 separated blocks of A and B



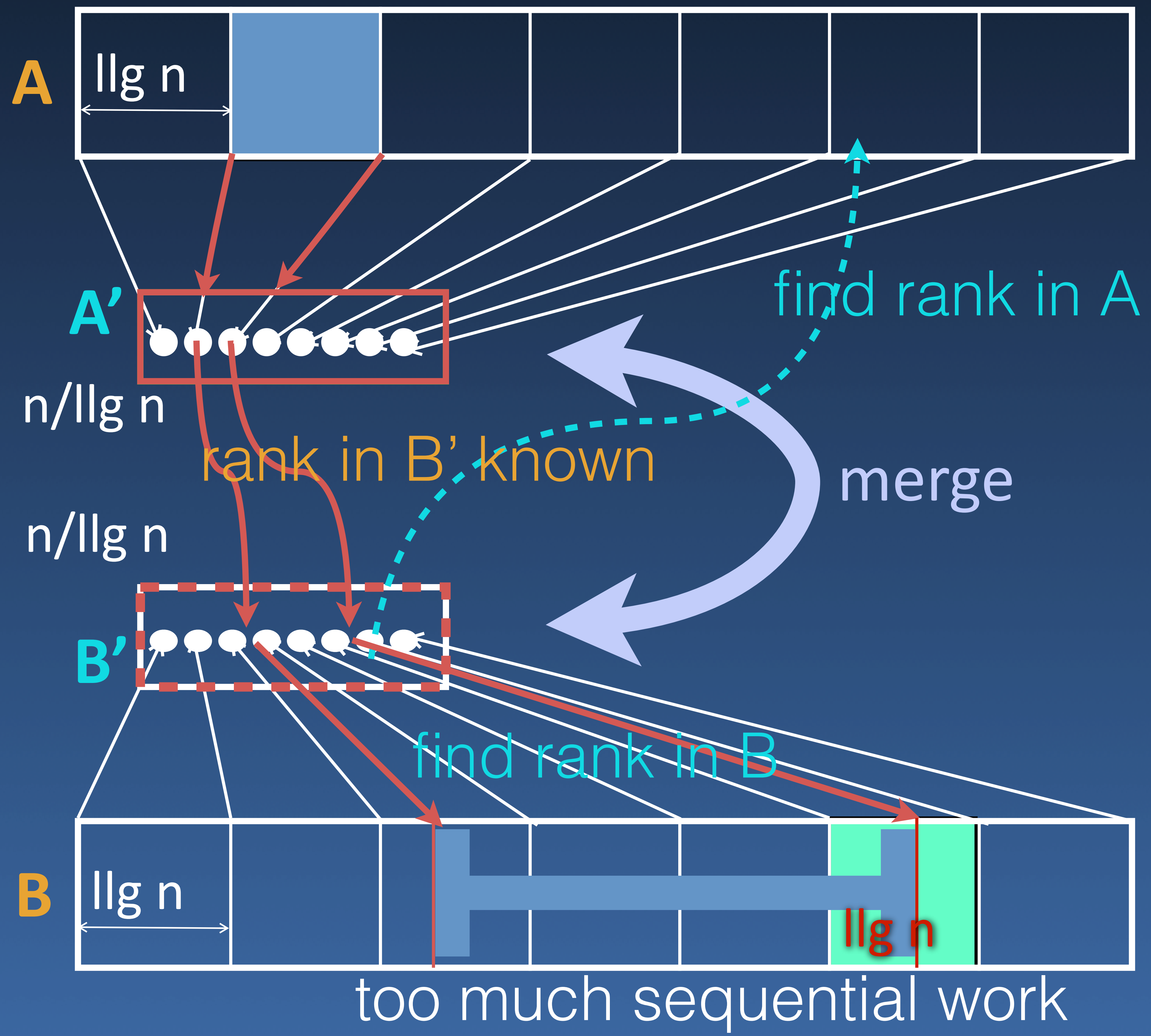
Subodh Kumar

- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ 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 separated blocks of A and B



Optimal Merge (A,B)

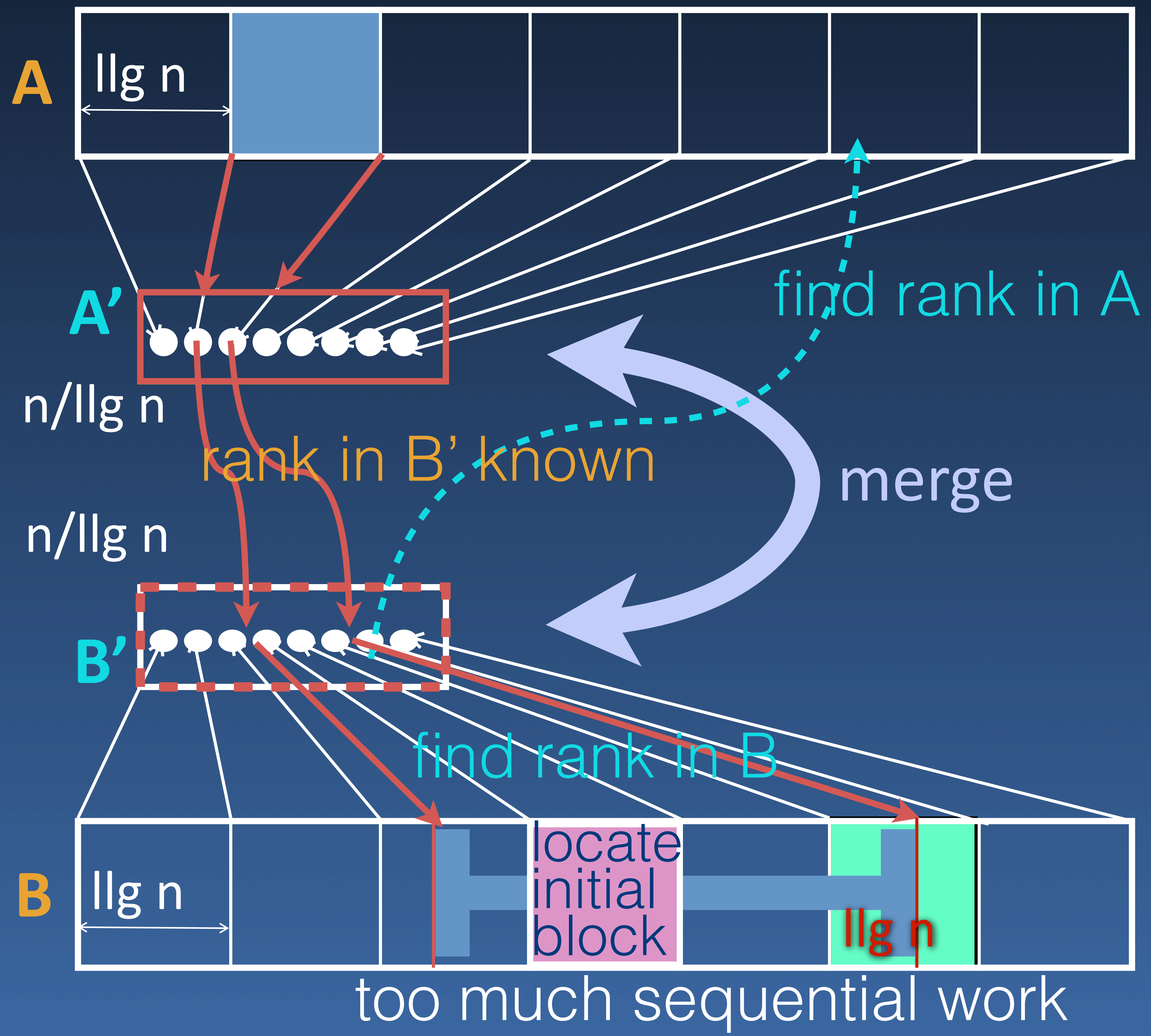
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



Subodh Kumar

Optimal Merge (A,B)

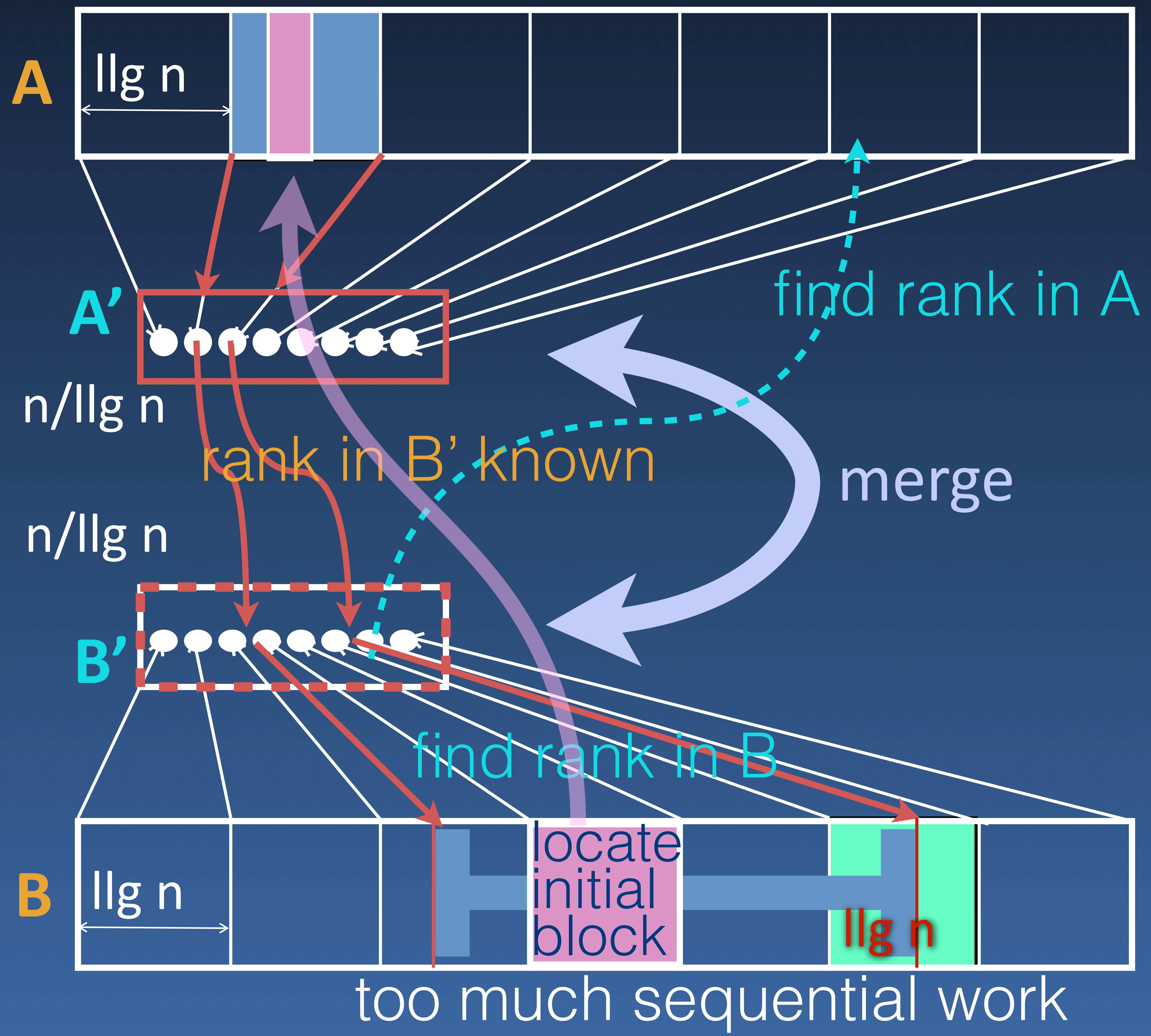
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A’ = p_1, p_2 ..
  - ➡ B’ = q_1, q_2 ..
- Now merge separated blocks of A and B



Subodh Kumar

Optimal Merge (A,B)

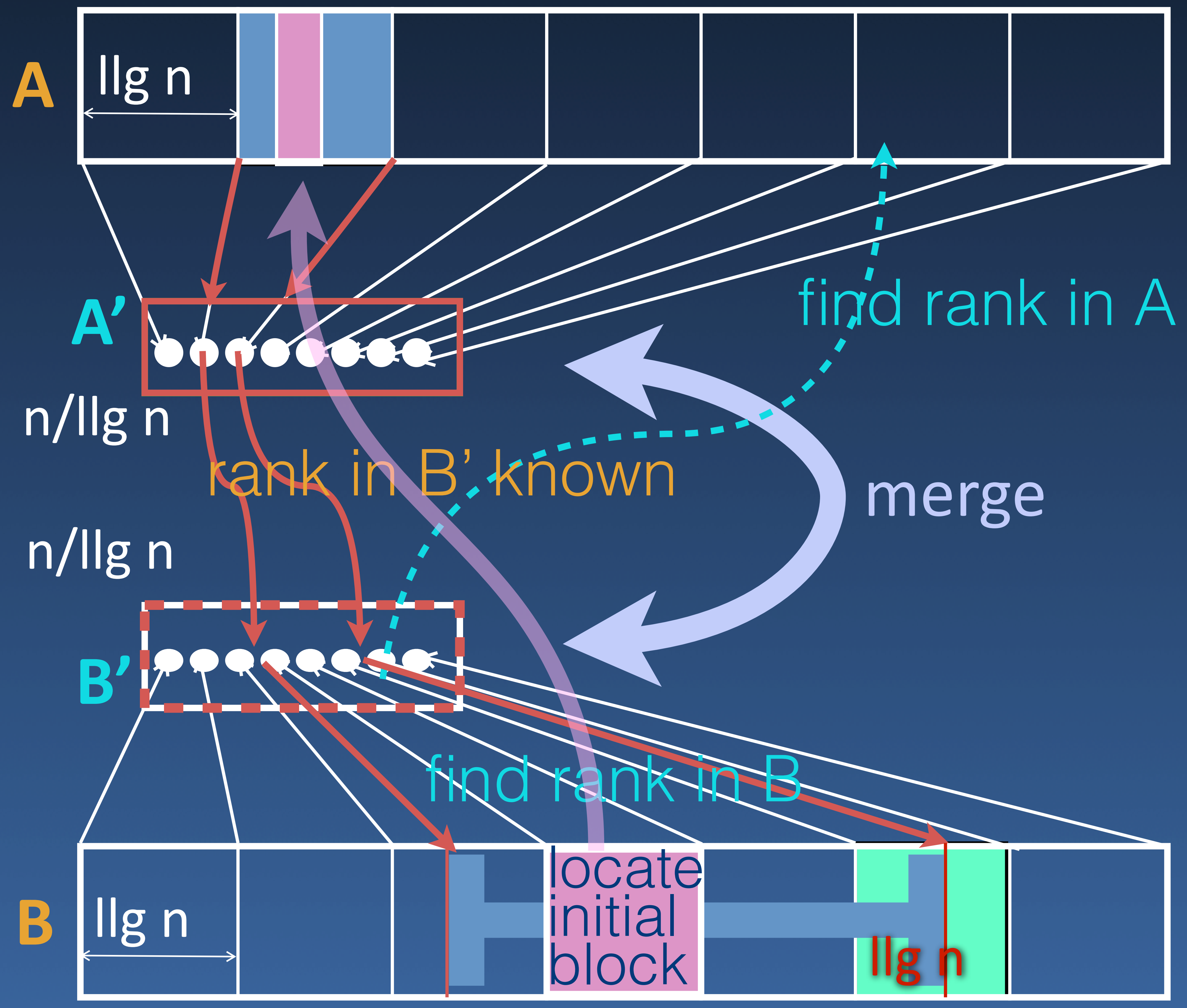
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ A_1, A_2, ..
  - ➡ B_1, B_2, ..
- Select first element of each block
  - ➡ A' = p_1, p_2 ..
  - ➡ B' = q_1, q_2 ..
- Now merge separated blocks of A and B



Subodh Kumar

Optimal Merge (A,B)

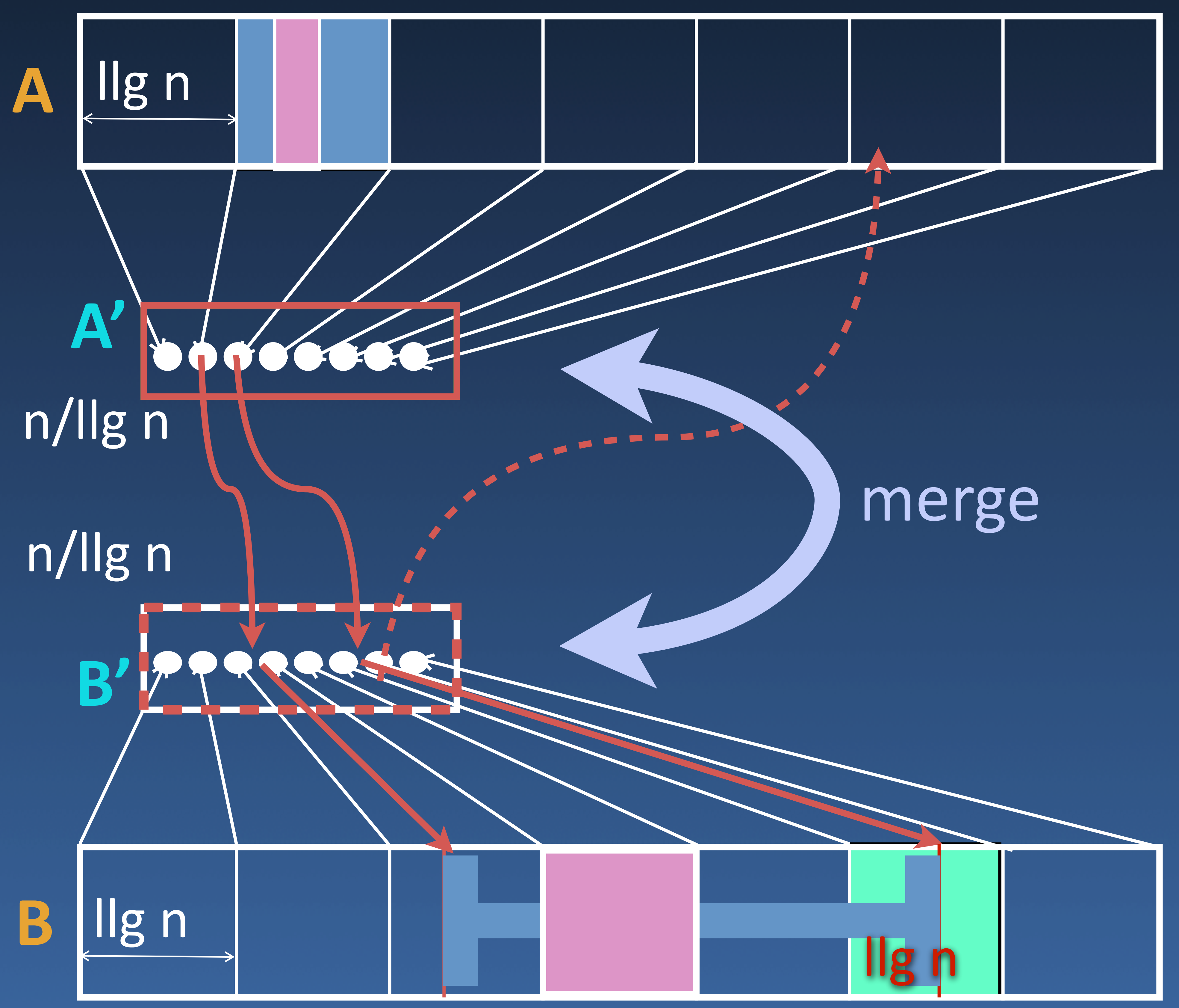
- Use the fast, non-optimal algorithm on small enough subsets
- Subdivide A and B into blocks of size ***llg n*** (llg = log log)
  - ➡ $A_1, A_2, ..$
  - ➡ $B_1, B_2, ..$
- Select first element of each block
  - ➡ $A' = p_1, p_2 ..$
  - ➡ $B' = q_1, q_2 ..$
- Merge O(llg n) sized blocks O(n/llg n) times



Subodh Kumar

Optimal Merge (A,B)
(Analysis)

1. Merge A' and B' – find Rank(A':B'), Rank(B':A')
   - Use 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)



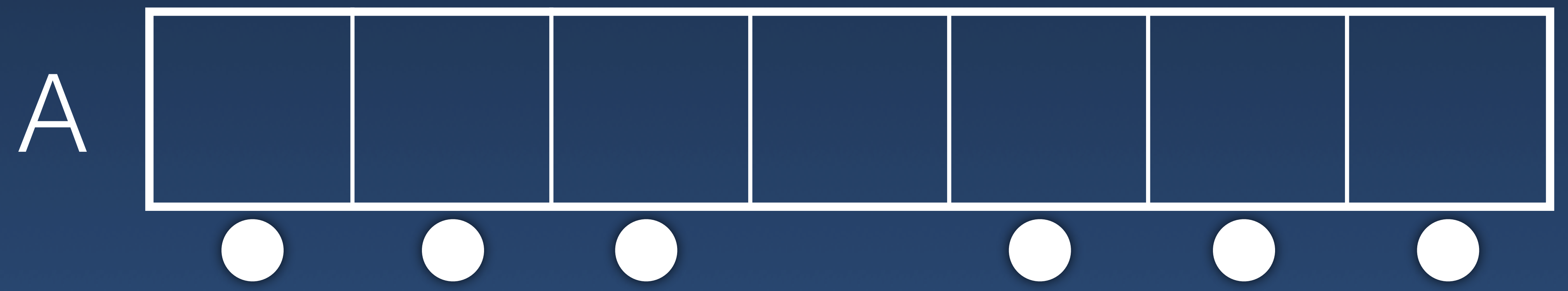
Subodh Kumar

Min-find

Input: array A with n elements

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

CRCW



Subodh Kumar

Min-find

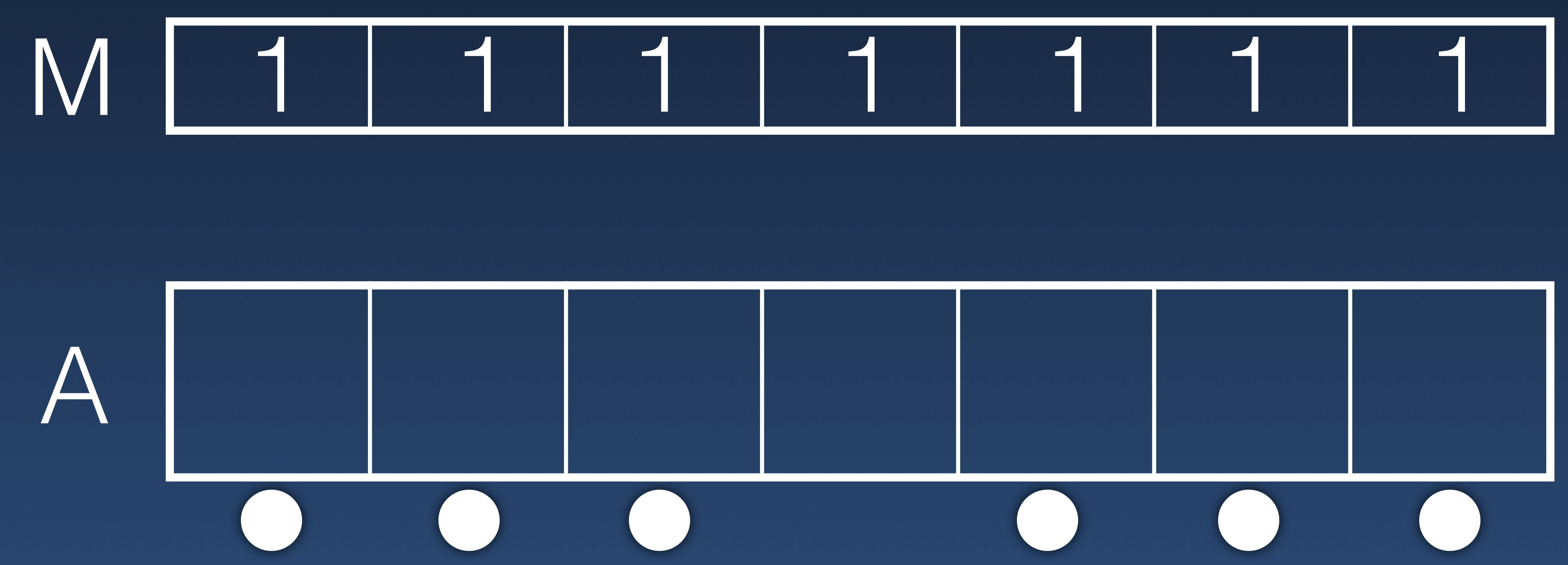
Input: array A with n elements

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

forall i in [0:n)

M[i] = 1

CRCW



Subodh Kumar

Min-find

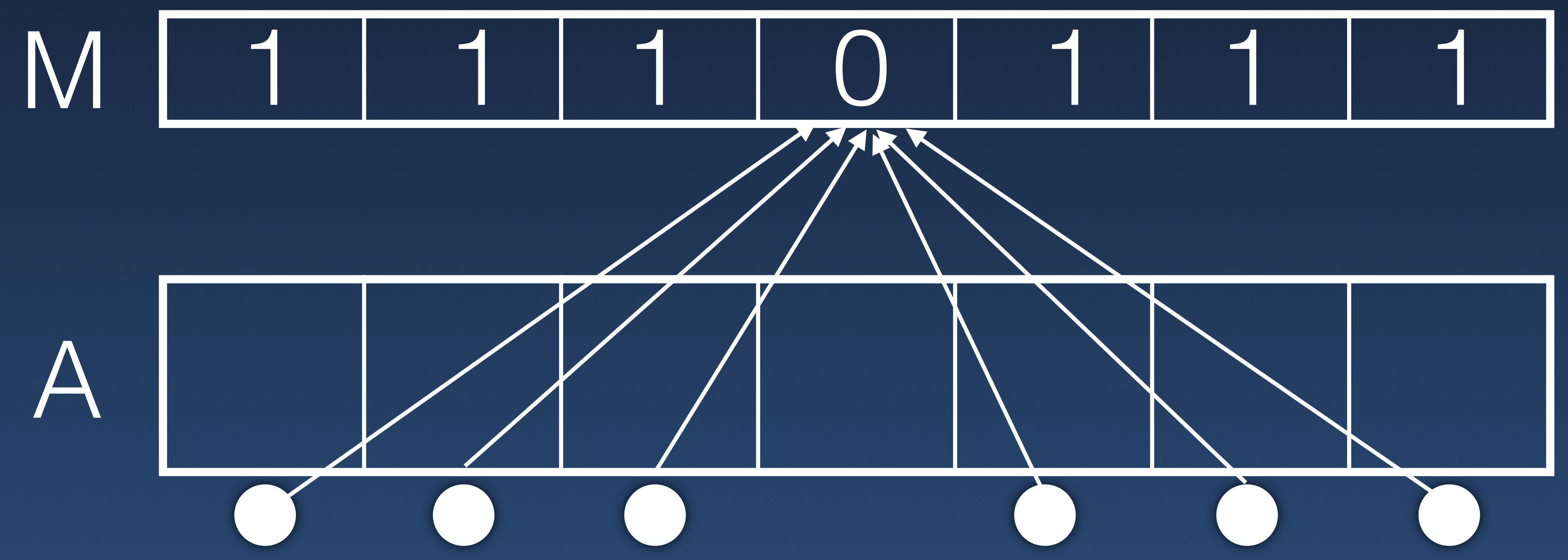
Input: array A with n elements

CRCW

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

forall i in [0:n)

M[i] = 1



Subodh Kumar

Min-find

Input: array A with n elements

CRCW

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

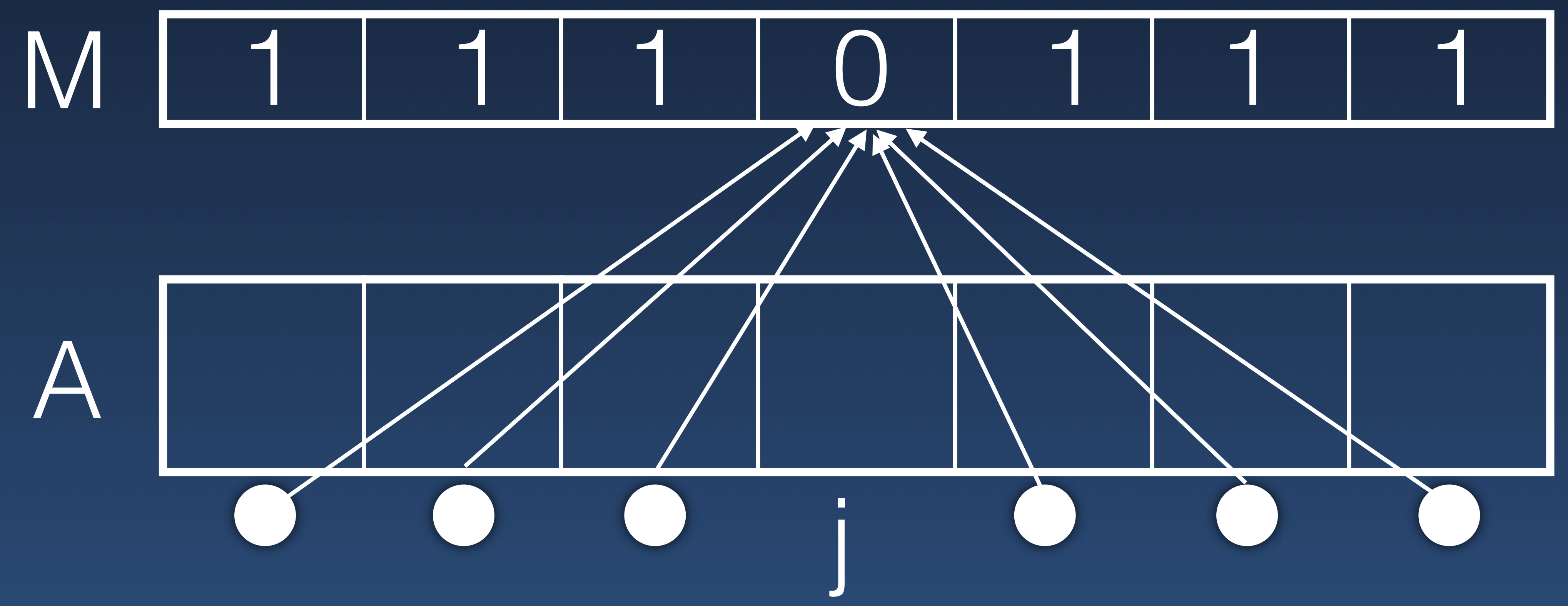
forall i in [0:n)

M[i] = 1

forall i,j in [0:n)

if i≠j && A[i] < A[j]

M[j] = 0



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)

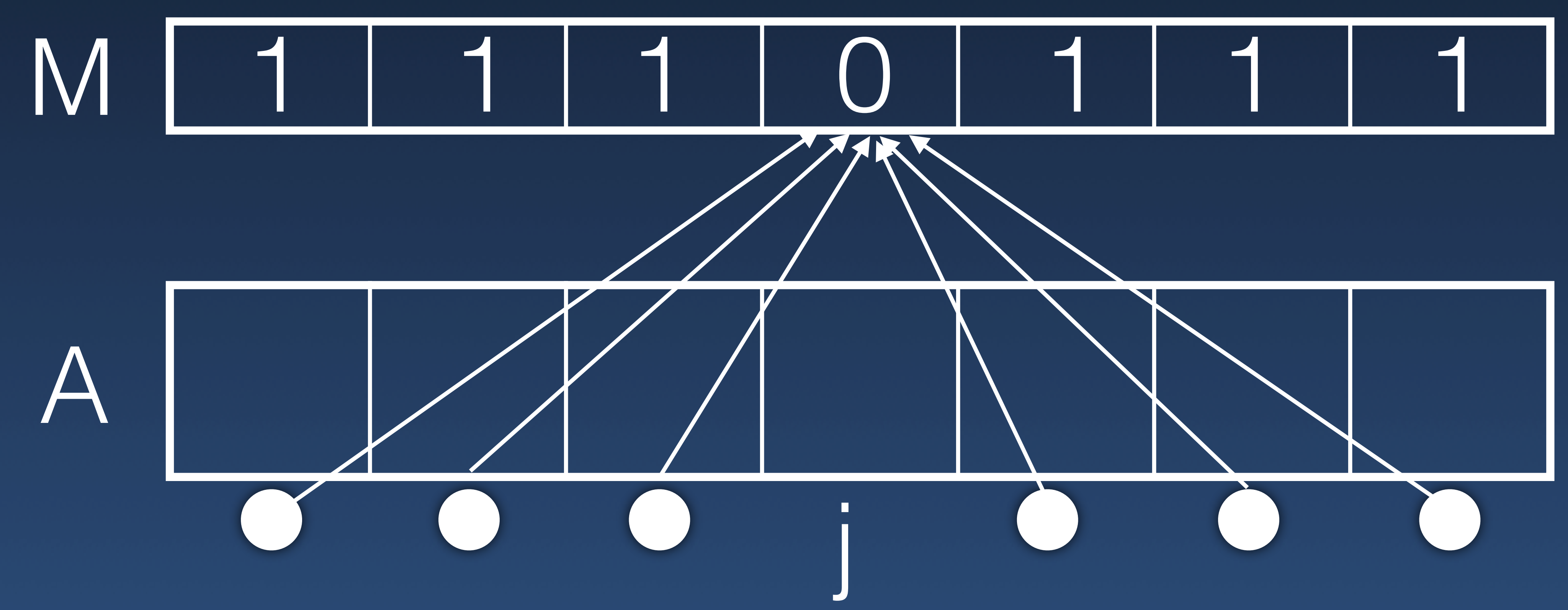
if i≠j && A[i] < A[j]

M[j] = 0

forall i in (0:n]

if M[i]=1

min = A[i]



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

Optimal Min-find

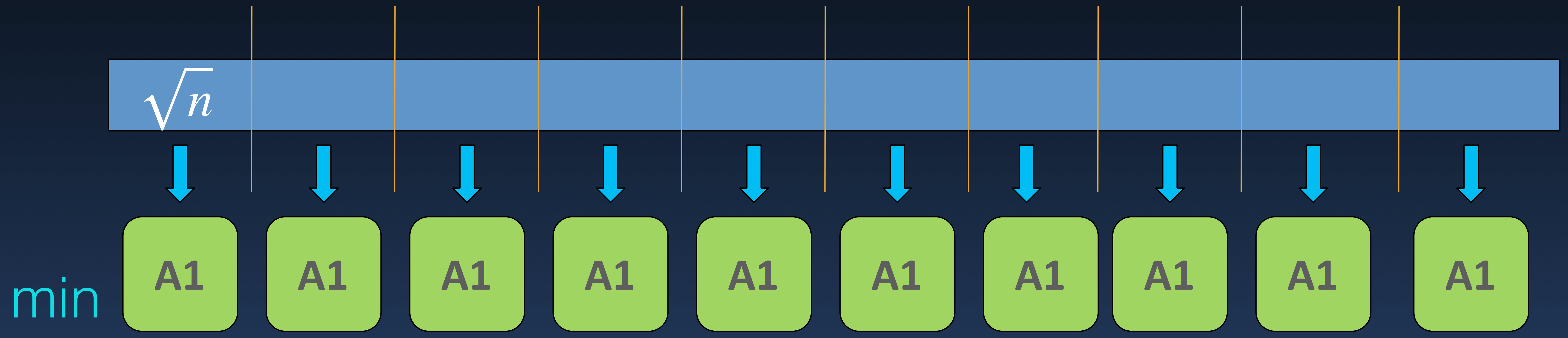
- Balanced Binary tree
  - O(log n) time
  - O(n) work => Optimal
- Make the tree branch quicker
  - Number of children of node u = $\sqrt{n_u}$
    - if the number of leaves in u's subtree is n_u
  - Works well if the operation at each node is fast, say, O(1)
- Use Accelerated cascading

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

Algorithm A2

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

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

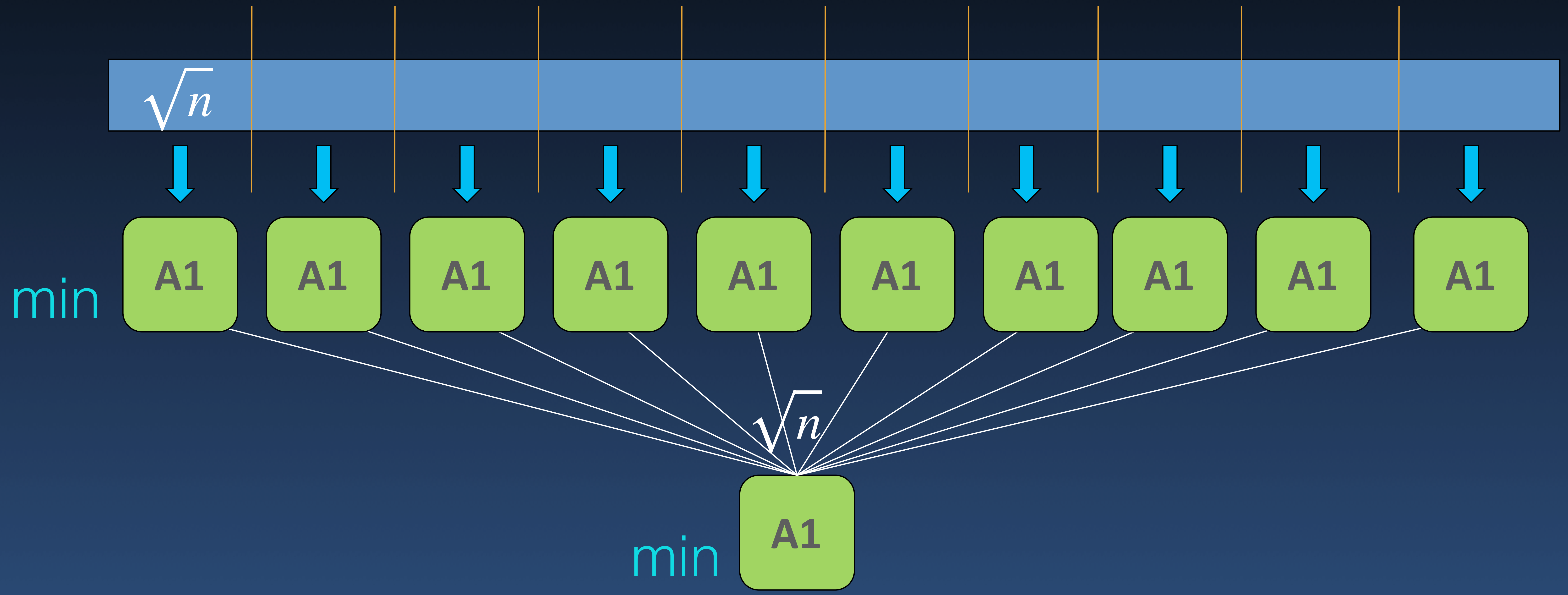


Algorithm A2

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

Step 2: Apply A1 to each block

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



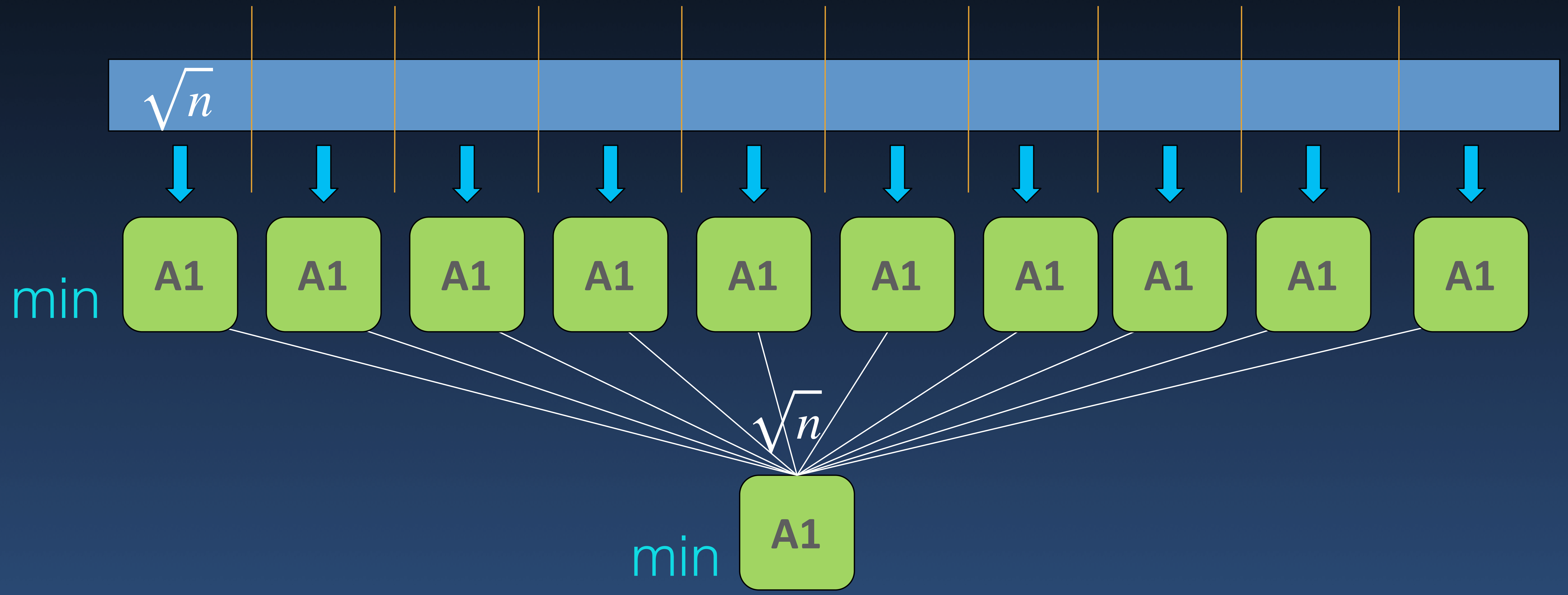
<u>Algorithm A2</u>

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

Step 2: Apply A1 to each block

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

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



Algorithm A2

	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

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

Algorithm A3

Step 1: Partition into disjoint blocks of size

Step 2: Apply A2 to each block

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

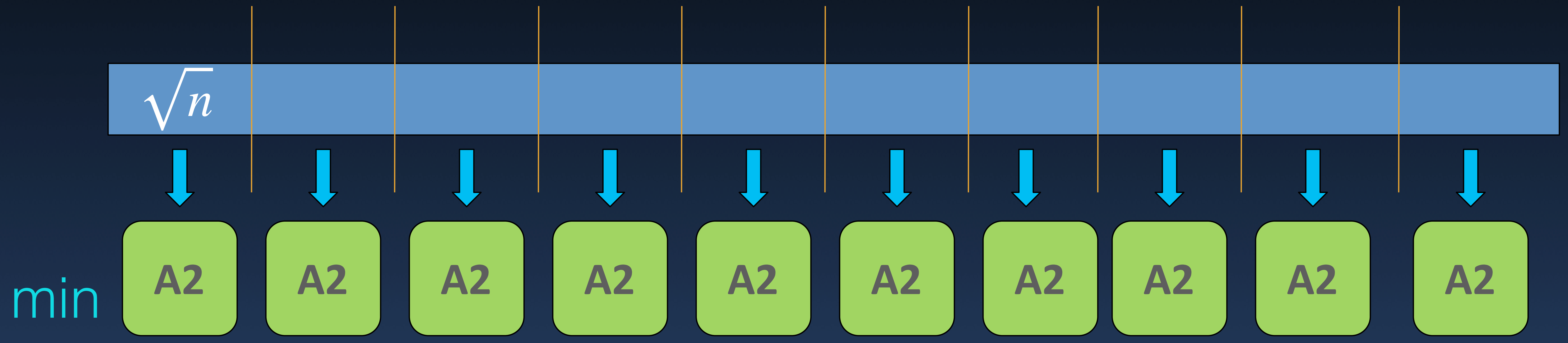
A2 work

$n\sqrt{n}$

n

Subodh Kumar

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



<u>Algorithm A3</u>

Step 1: Partition into disjoint blocks of size

Step 2: Apply A2 to each block

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

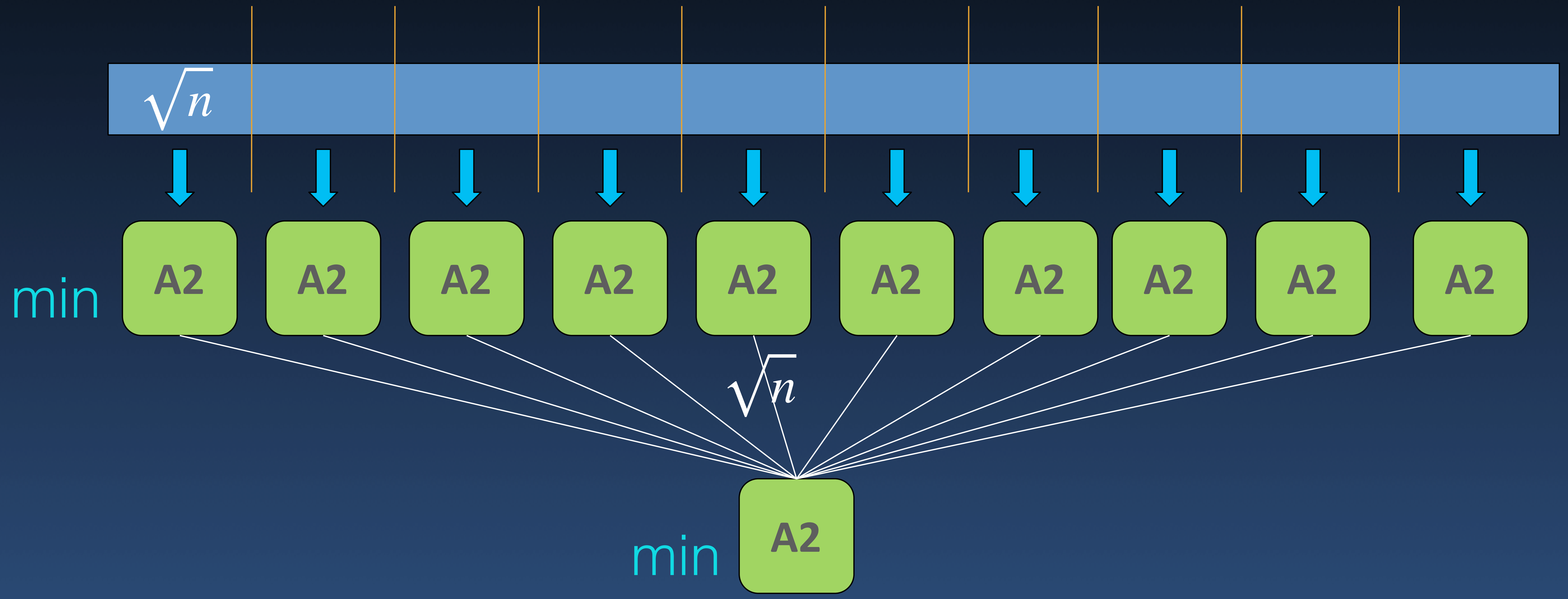
A2 work

$n\sqrt{n}$

n

Subodh Kumar

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



Algorithm A3

Step 1: Partition into disjoint blocks of size

Step 2: Apply A2 to each block

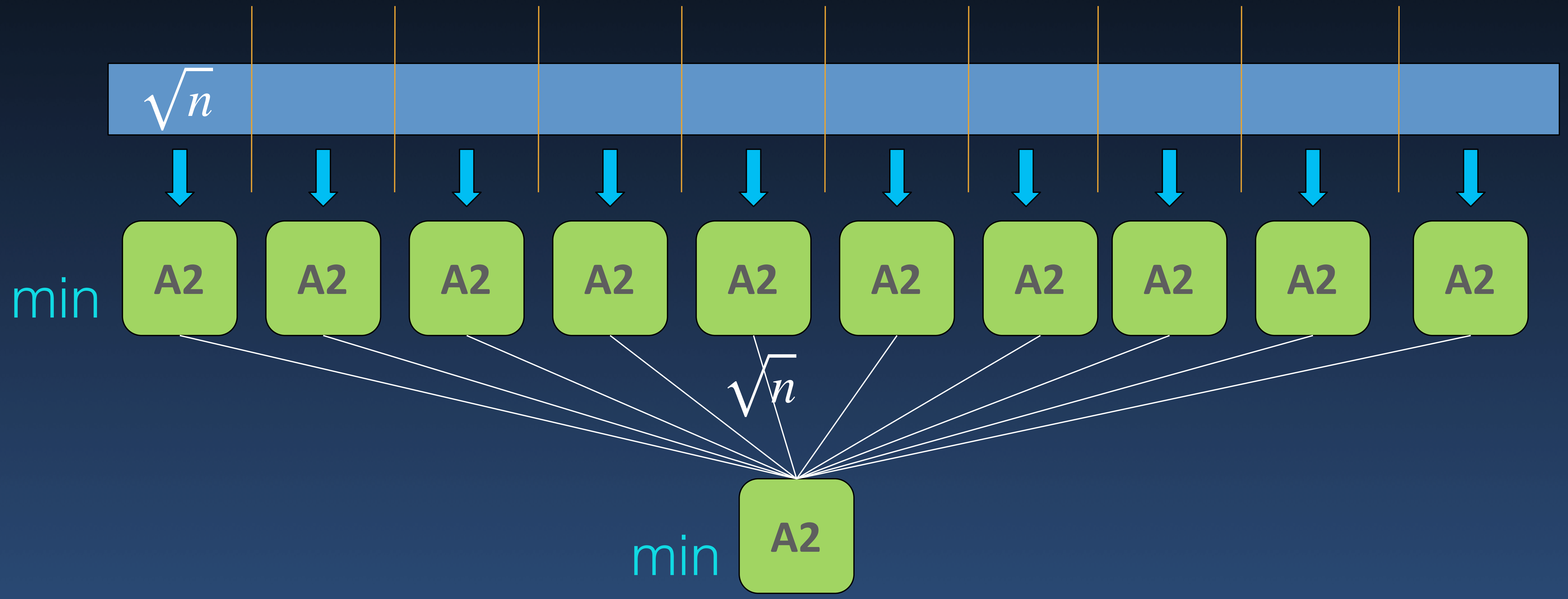
Step 3: Apply A2 to the results from the step 2

A2 work

$n\sqrt{n}$

n

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 $\boldsymbol{n}^{1/2}$ 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 $\boldsymbol{n}^{1/2}$ minima

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

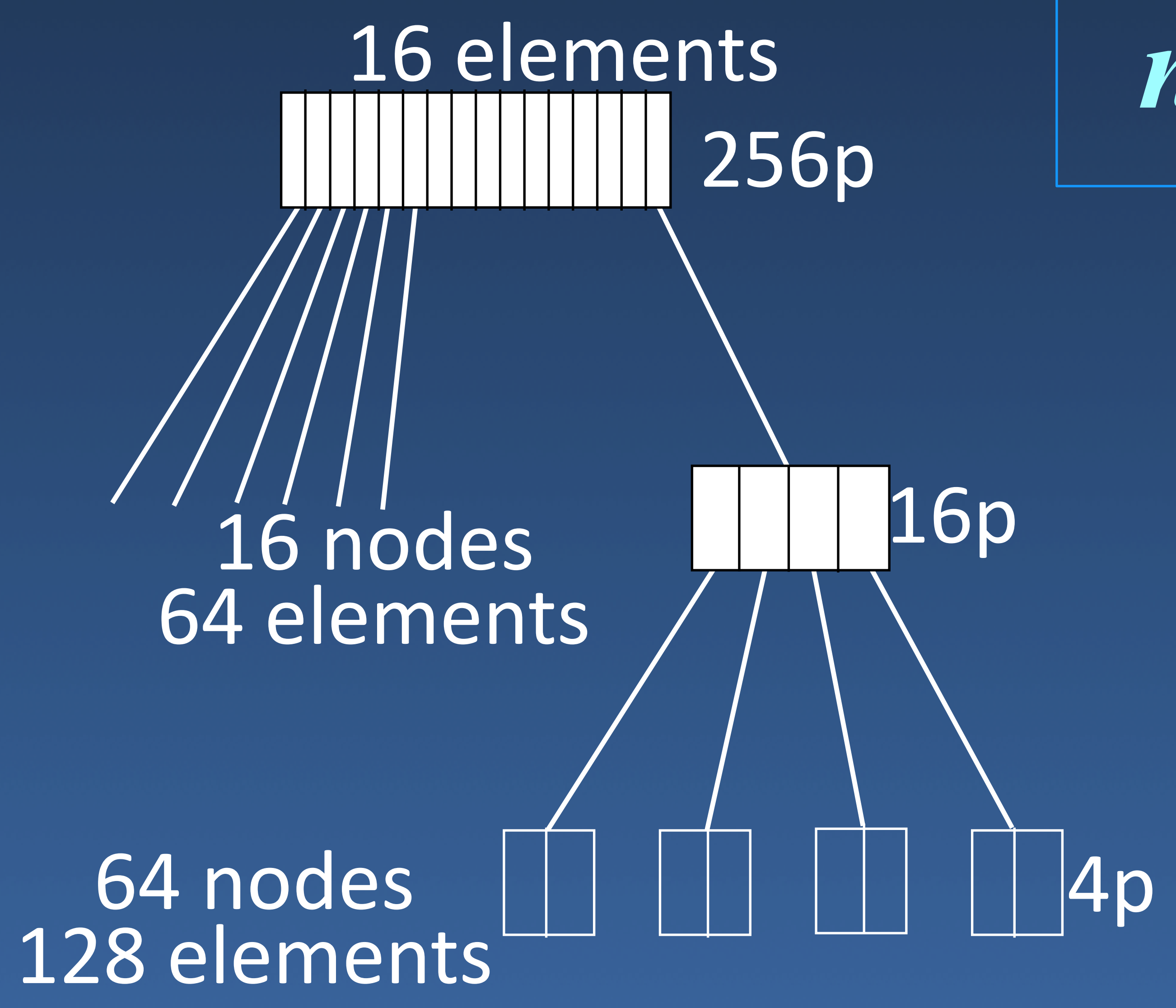
Algorithm A_∞ takes ?? with $n^{1+\epsilon}$ processors

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

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



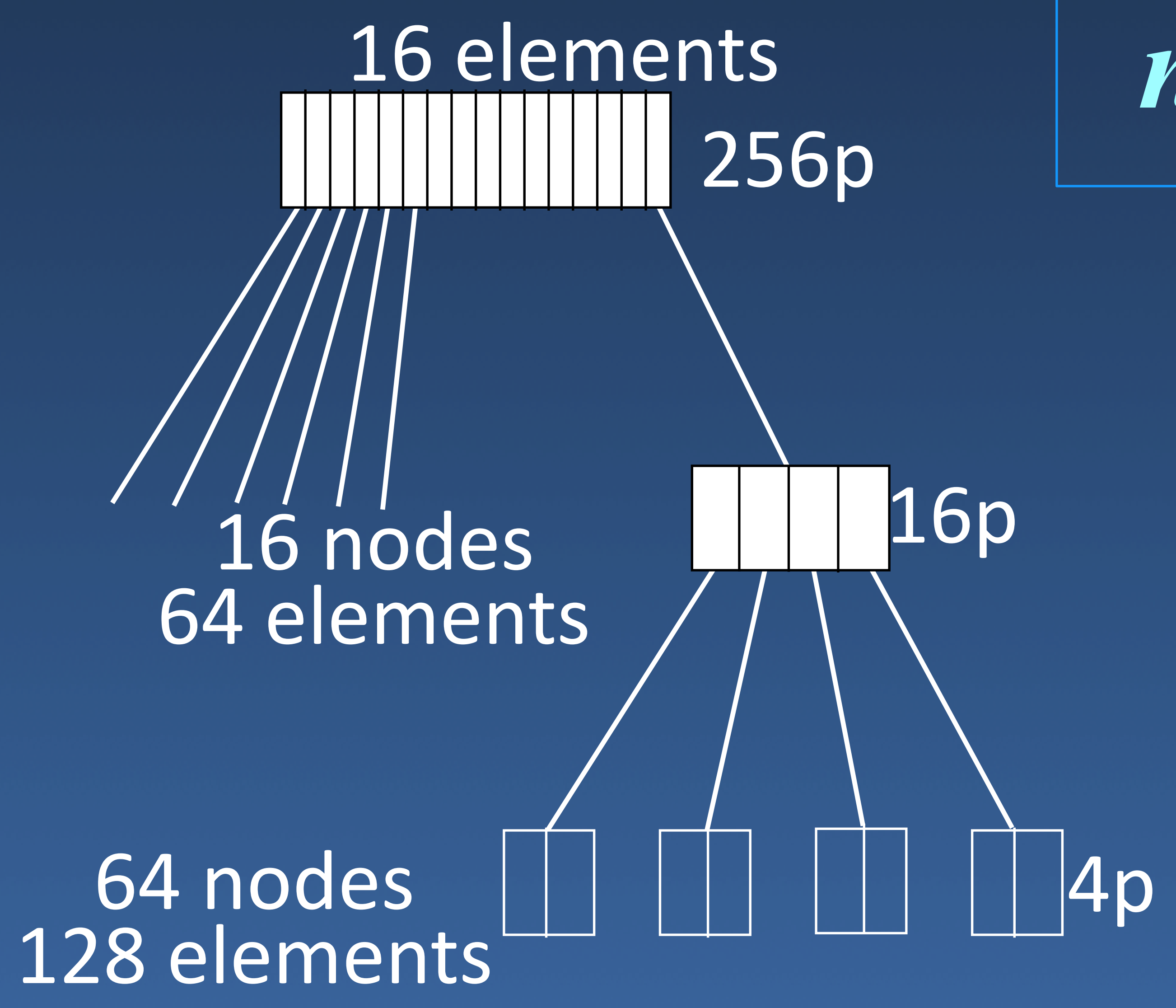
A_1		A_2		A_3 ..						
n^2	->	$n^{1+1/2}$	->	$n^{1+1/4}$	->	$n^{1+1/8}$	->	$n^{1+1/2^k}$	..	$\sim n^{1+\varepsilon}$

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

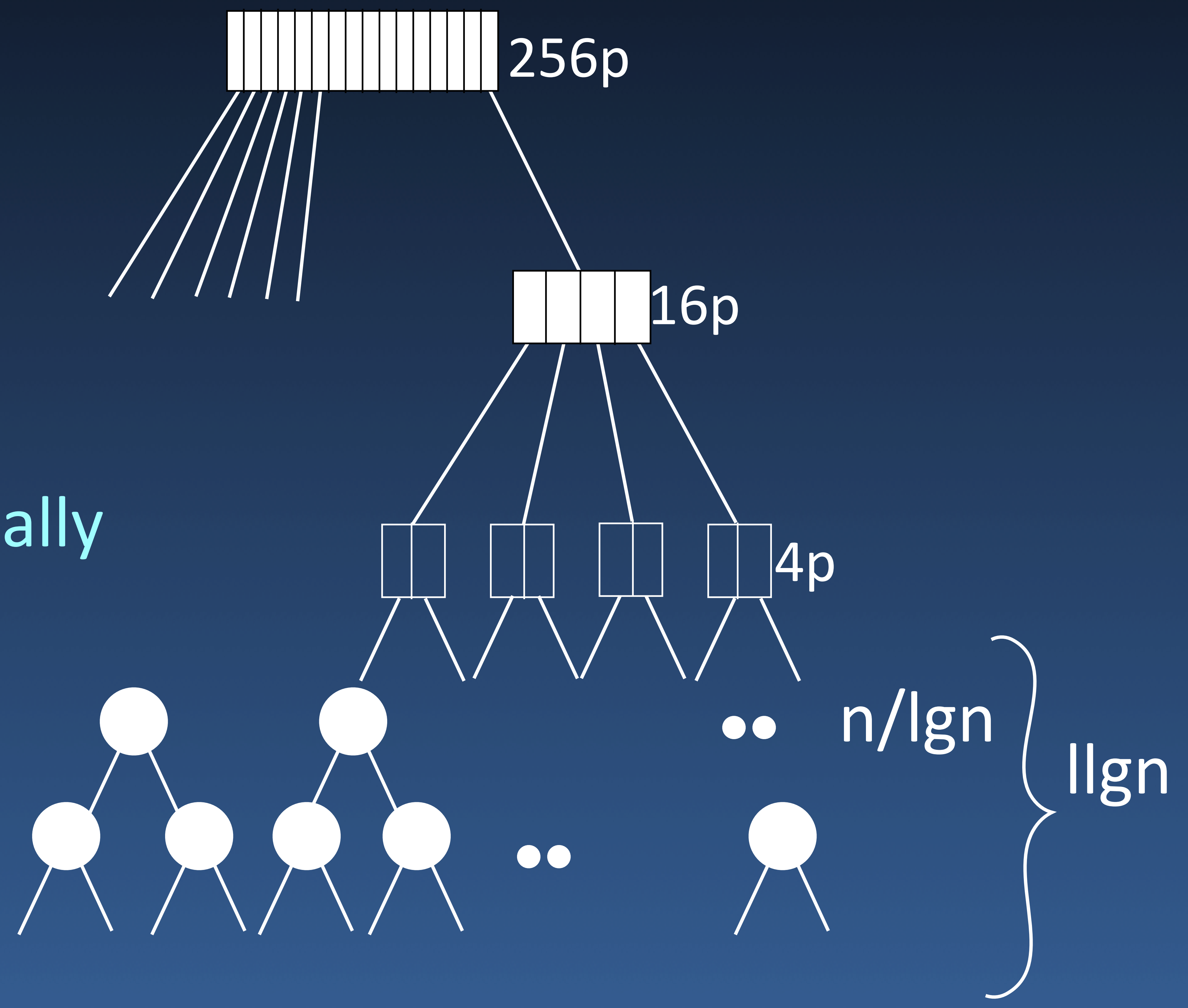
$\boxed{\text{n loglog n work, loglog n time}}$

Min-Find Review

- Constant-time algorithm
  - ➡ $O(n^2)$ work
- $O(\log n)$ Balanced tree approach
  - ➡ $O(n)$ work (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 = $O(\log \log n)$

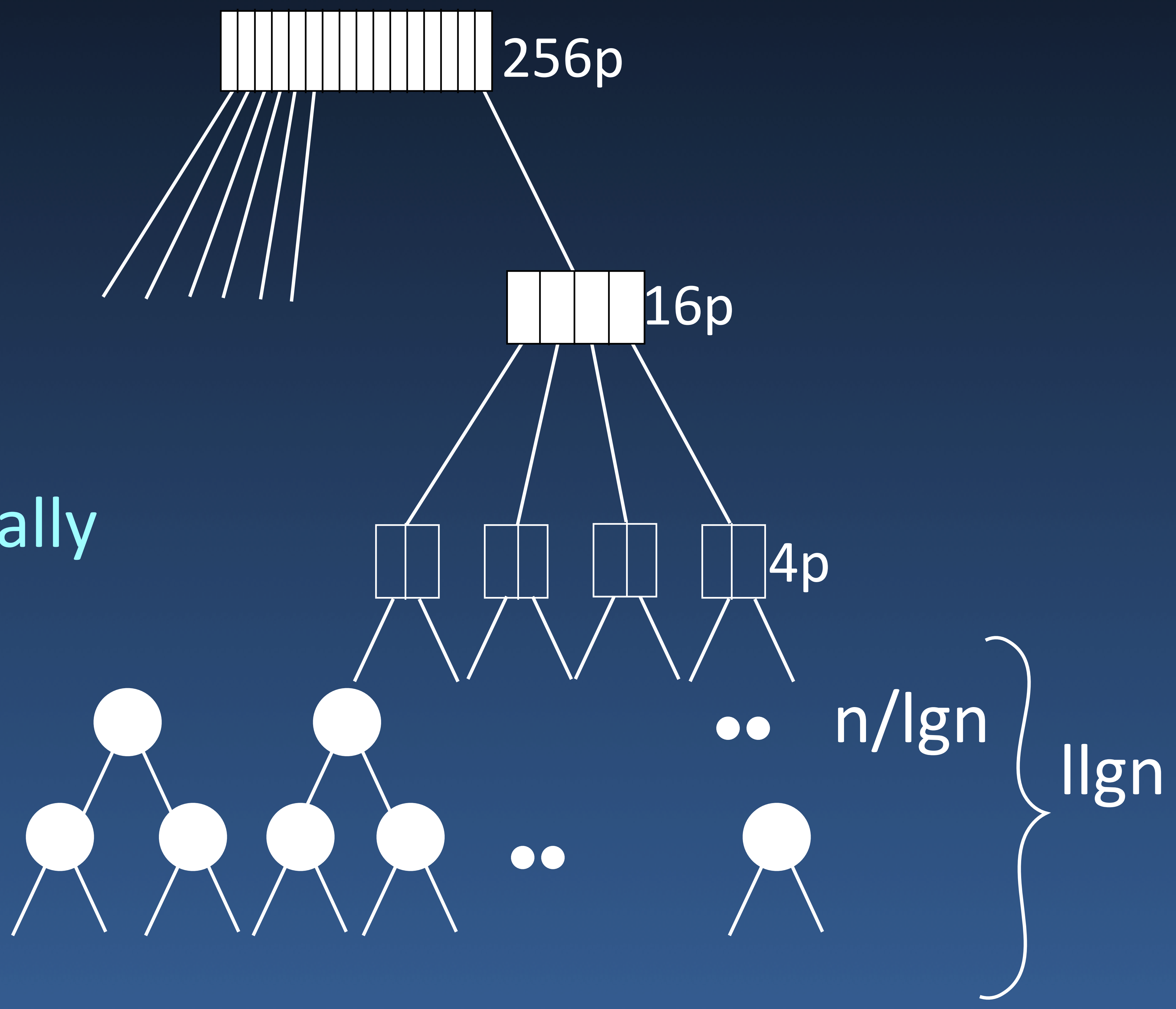
Accelerated Cascading

- 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 loglog n levels
  - Then switch to O(n loglog n)-work algorithm



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

- 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

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]
}
```

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

```
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]
}
```

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

(i - psum[i])

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]
}
```

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

(i - psum[i]) + psum[n-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
- Time per round = O(log n)
- Work per round = O(n)

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

```
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]
}
```

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

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

Subodh Kumar

Review

- Merge, Minima-find, Quicksort
- Combining non-work optimal fast algorithm with optimal less fast algorithm
  - ➡ Accelerated Cascading