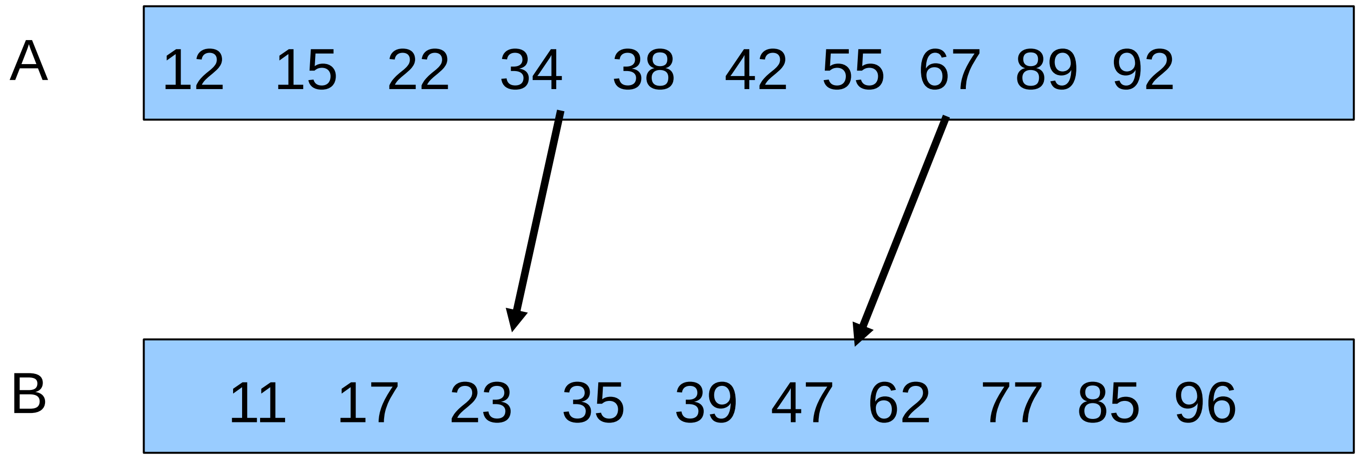
Other Design Paradigms

- Partitioning
  - Similar to divide and conquer
  - But no need to combine solutions
  - Can treat problems independently and solve in parallel.
  - Example: Parallel merging, searching.

Merging in Parallel by Partitioning



- Two sorted arrays A and B to be merged into C.
- Let A be a sorted array. Let Rank(x, A) be the number of elements smaller than x in A.
- Claim: Rank(x, C) = Rank(x, A) + Rank(x, B)
- For x in A, Rank(x,A) is immediately available. To find Rank(x, B) can use binary search in parallel.

Quick Example

A = [8 10 12 24]

B = [15 17 27 32]

Element	8	10	12	24	15	17	27	32
Rank in A	0	1	2	3	3	3	4	4
Rank in B	0	0	0	2	0	1	2	3
Rank in C	0	1	2	5	3	4	6	7

C = [8 10 12 15 17 24 27 32]

Merging in Parallel by Partitioning

- Time for each binary search is O(log n)
- Total time for merging = O(log n), the total work is O(n log n).
  - Not work optimal as compared to the best possible sequential time complexity of O(n).
- Can reduce the total work to O(n).
  - Induce equal-sized partitions in the arrays
  - Rank one element, say the first element, from each partition
  - Use these ranks to find the ranks of the other elements, sequentially.

An Improved Optimal Algorithm

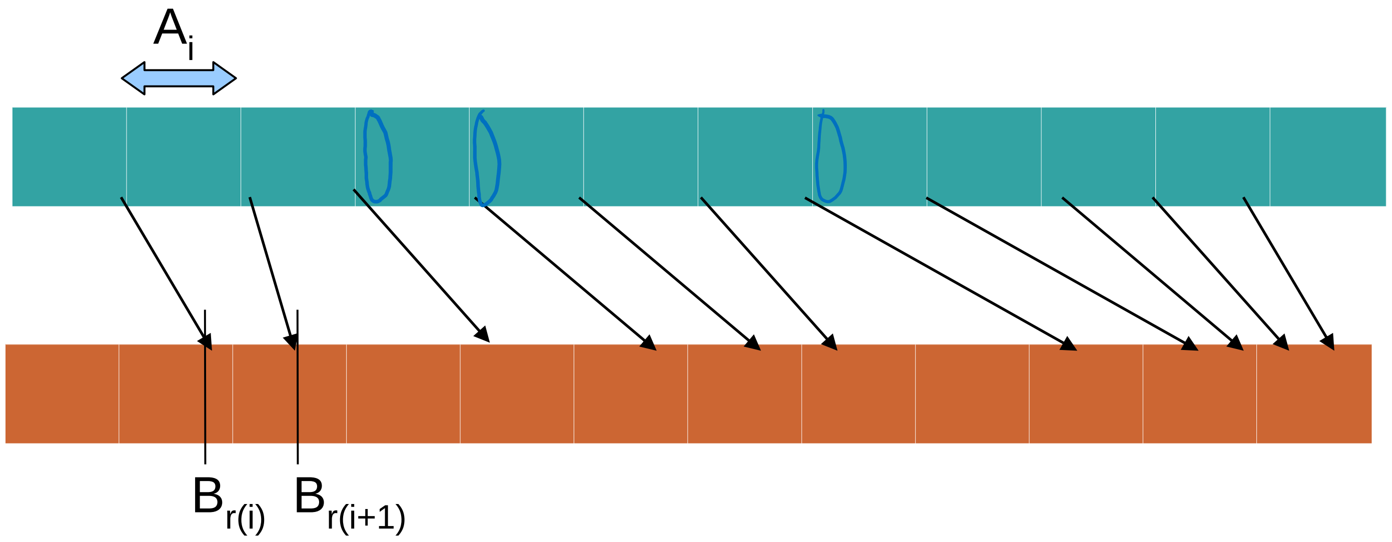
- General technique
  - Solve a smaller problem in parallel
  - Extend the solution to the entire problem.
- For the first step, the problem size to be solved is guided by the factor of non-optimality of an existing parallel algorithm.

An Improved Parallel Algorithm

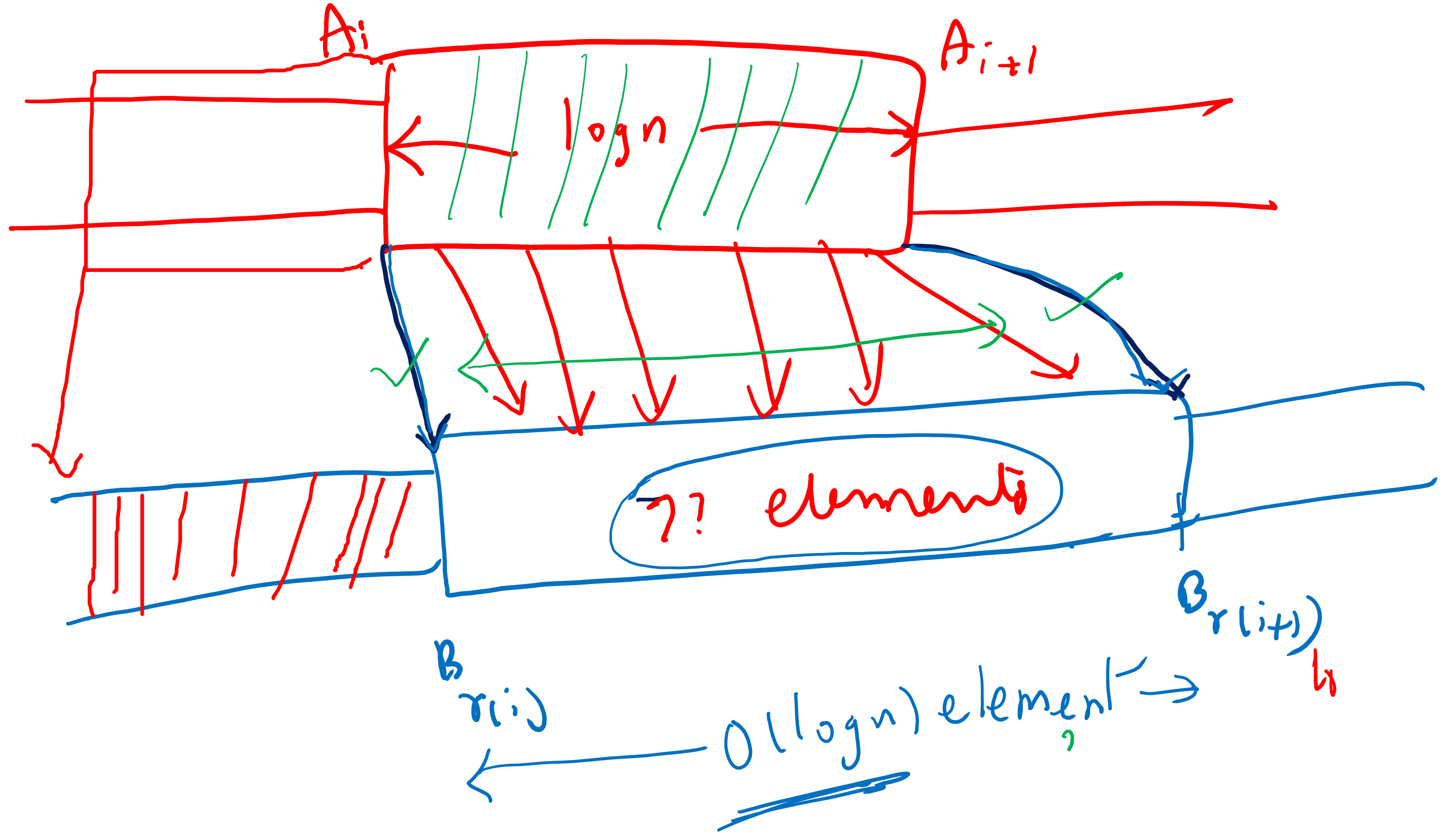
- Our simple parallel algorithm is away from work optimality by a factor of O(log n).
- So, we should solve a problem of size O(n/log n).
- For this purpose, we pick every log n^{th} element of A, and similarly in B.
- Use the simple parallel algorithm on these elements of A and B.
  - Binary search however in the entire A and B.

An Improved Parallel Algorithm

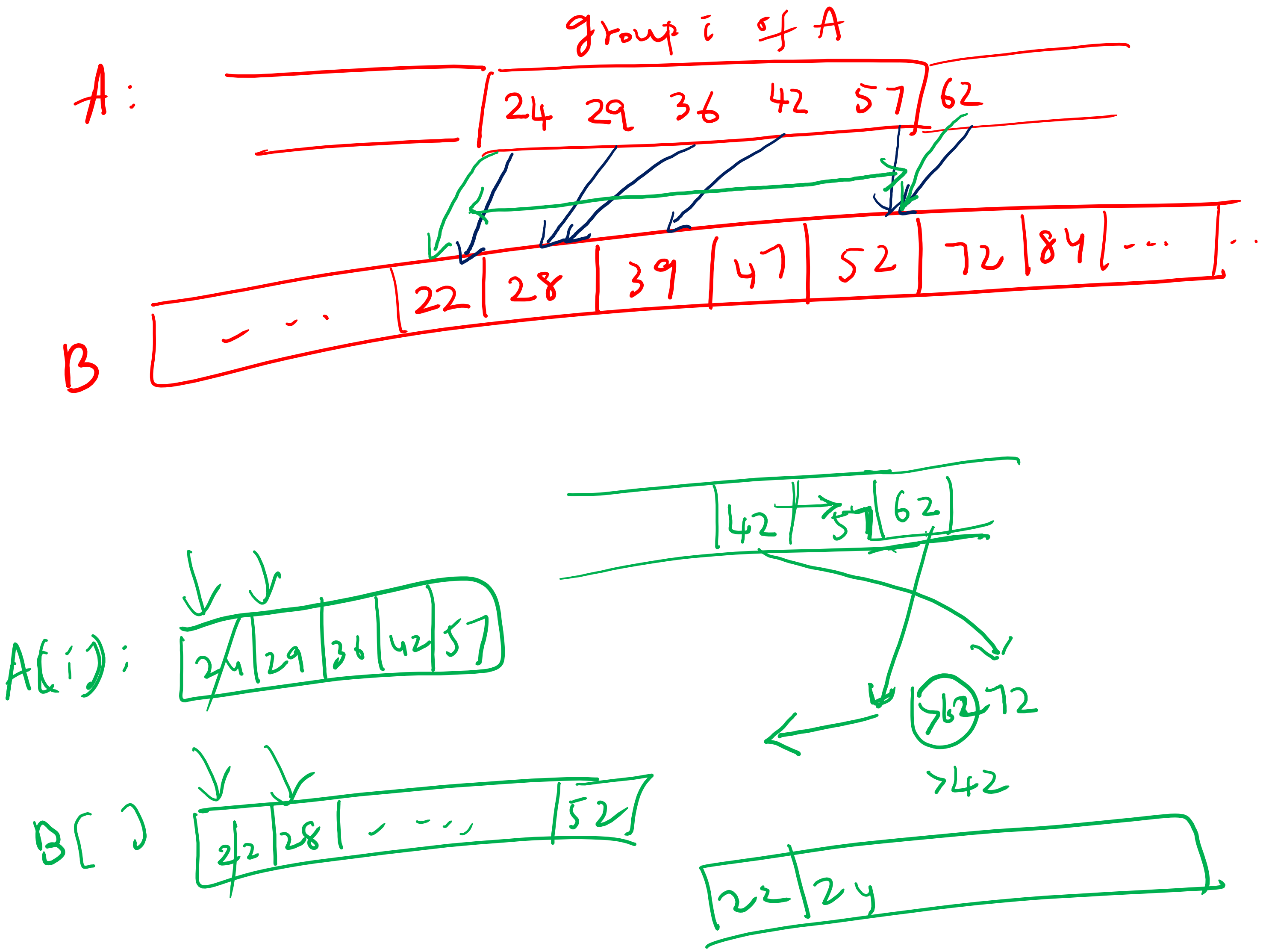
$\frac{n}{\log n}$



- Let $A_1, A_2, \ldots, A_{n/\log n}$ be the elements of A ranked in B.
- These ranks induce partitions in B.
  - Define $[B_{r(i)} \ldots B_{r(i+1)}]$ as the portion of B so that [A(i)...A(i+1)] have ranks in.
- Can therefore merge [A(i)...A(i+1)] with $[B_{r(i)} \ldots B_{r(i+1)}]$ sequentially.



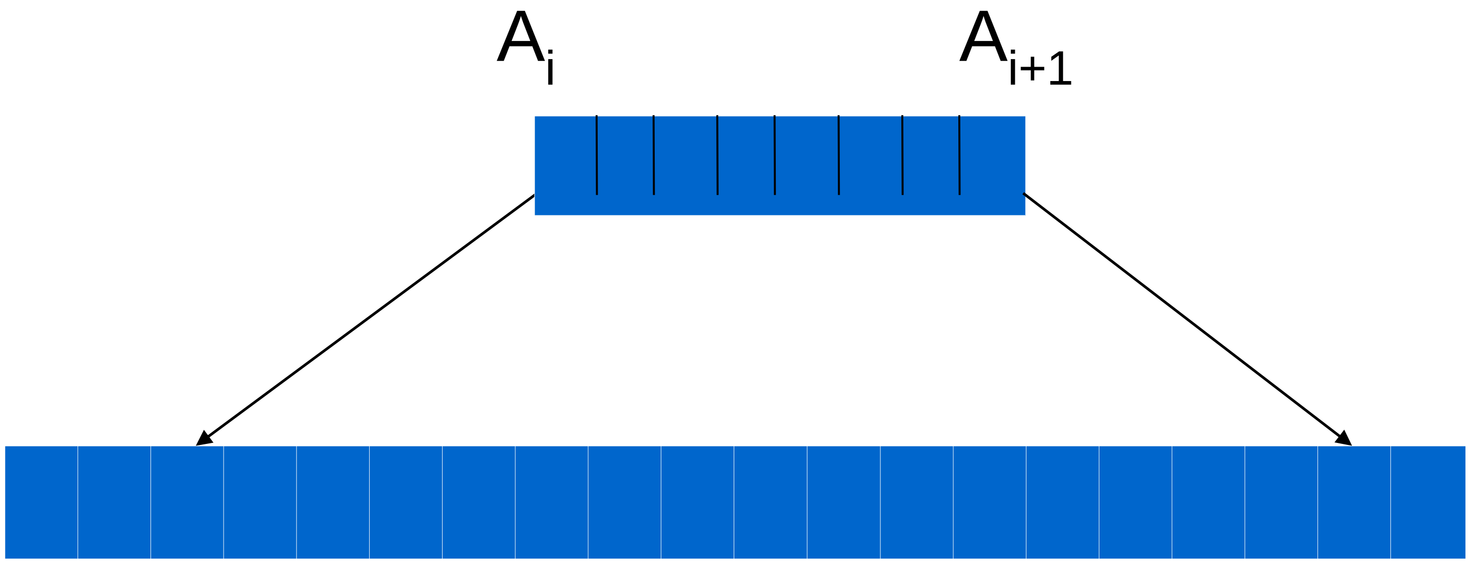
Ai
Ai+1
log n
?? elements
Br(i)
Br(i+1)
O(log n) elements


group i of A
A:
24 29 36 42 57 62
B
22 28 39 47 52 72 84
A[i]:
24 29 36 42 57
B[ ]
22 28 52
42 57 62
>62 72
>42
22 24

An Improved Parallel Algorithm

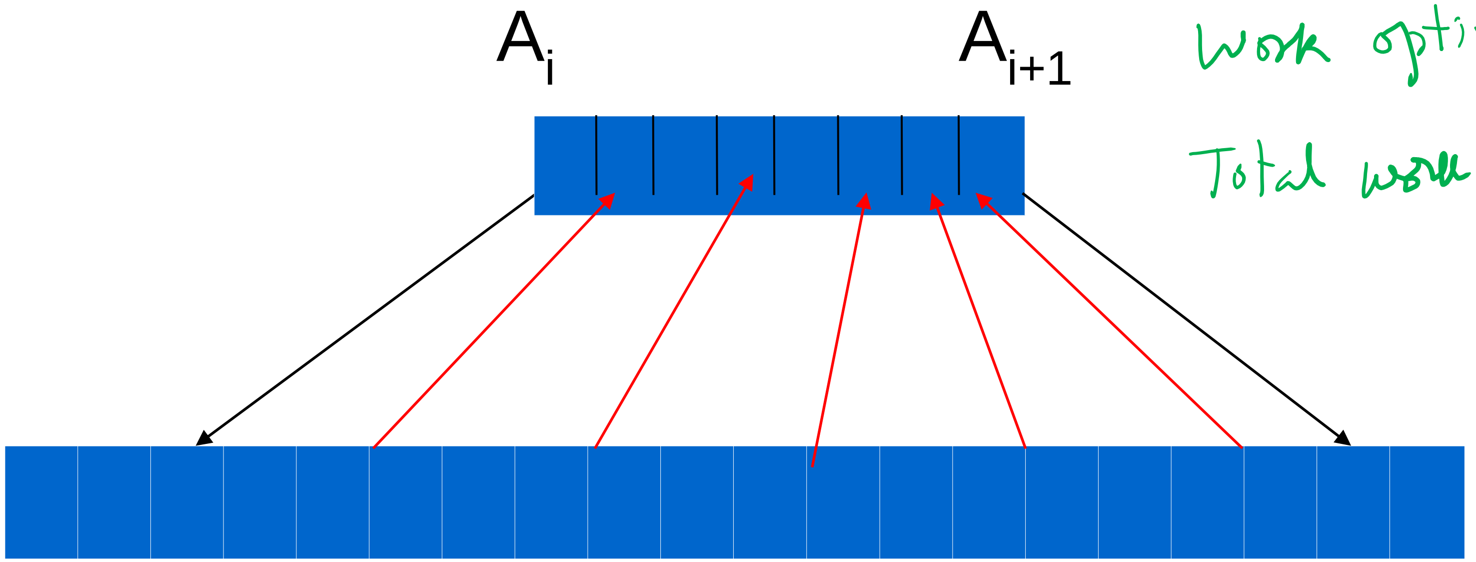
- Such sequential merges can happen in parallel, at each index of A[i].
- Time taken for the sequential merge is $O(\log n + B_{r(i+1)} - B_{r(i)})$.
- Time:
  - Binary search: O(log n), with n/log n processors.
  - Sequential merge: O(log n), subject to certain conditions. There are also n/log n such merges in parallel.
- Work:
  - There are n/log n binary searches in parallel. Work = O(n).
  - For the sequential merges too, work = O(n).

An Improved Parallel Algorithm



- What if $[B_{r(i)} \ldots B_{r(i+1)}]$ has a size of more than log n?
- The situation can be addressed
  - Pick equally spaced, no more than log n, spaced items in $[B_{r(i)} \ldots B_{r(i+1)}]$.
  - Rank these in $[A_i \ldots A_{i+1}]$.

An Improved Parallel Algorithm



Work optimal

Total work = $O(n)$

- What if $[B_{r(i)} \ldots B_{r(i+1)}]$ has a size of more than log n?
- The situation can be addressed
  - Pick equally spaced, no more than log n, spaced items in $[B_{r(i)} \ldots B_{r(i+1)}]$.
  - Rank these in $[A_i \ldots A_{i+1}]$.

Final Result

- Can merge two sorted arrays of size n in time O(log n) with work O(n).
  - Need CREW model, for binary searches.
- Can improve further, we will see later.
- The technique to achieve optimality is a general technique, with several applications. We will see more applications of this later.

A Further Improvement

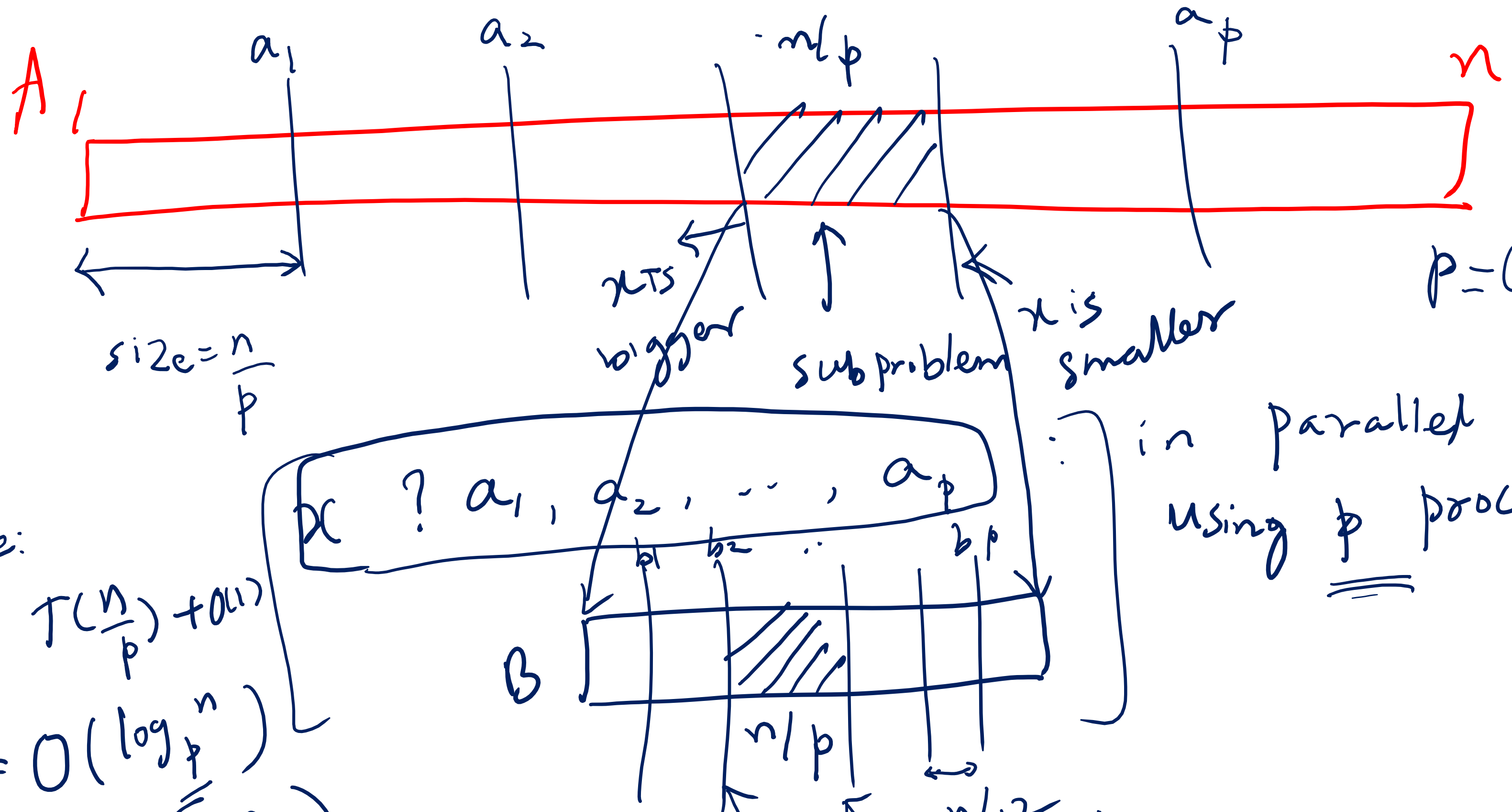
- Where is the scope for improvement?
- Each binary search takes O(log n) time, and we also have O(n/log n) subproblems each of size O(log n).
- To get further improvements, we should look at both aspects.
- Can we search faster? Parallel?

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
  - Either all 0's
  - A shift from 1's to 0's

x : element to be searched

$$W(n) = W\left(\frac{n}{p}\right) + p$$
$$= O\left(p \log_p n\right)$$



$p = O(1)$

in parallel using p processors

Time:

$$T(n) = T\left(\frac{n}{p}\right) + O(1)$$
$$= O(\log_p n)$$
$$= O\left(\frac{\log n}{\log p}\right)$$

x ? $b_1, b_2, \ldots, b_p$

in parallel using p processors

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!

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

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.

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

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

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