Task of processing a table consisting of records whose keys come from a linearly ordered set arises in many practical situations.

- Searching for a key in a table
- Sorting the keys of a table

Tasks involve repeated comparisions & data movement operations.

We assume each key is an atomic unit that cannot be manipulated as an integer or as a string of bits.

Optimal algorithms exist for searching, merging, sorting.

Several algorithmic techniques have been developed;

1> Parallel Search

2> Partitioning

3> Pipelined or cascading divide & conquer.

4> Bitonic Sorting.

Searching

given sorted array $X = (x_1, x_2 - - - - x_n)$ st. $x_i \leq x_{i+1}$

and a value y.

find: index i st $x_i \leq y_i < x_{i+1}$ $(x_0 = -\infty, x_{n+1} = +\infty)$

$[-\infty < x < +\infty, \ \forall x \in X]$.

Sequential binary search solves problem in $O(\log n)$ time.

Can we do it faster given $1 < p \leq n$ processors?

(Parallel Search).

Basic Idea: Do "parallel/concurrent comparisons" by splitting X into p equal pieces and restrict subsequent search in a subarray of size $\frac{n}{p}$ (we assume $p \mid n$, if not replace by $\lfloor n/p \rfloor$).

if we find 'i' in time ~

⇒ If we apply this recursively

$$\log_p n = O\left(\frac{\log n}{\log p}\right).$$

ample

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$X =$	$-\infty$	2	4	6	8	10	12	14	16	18	20	22	24	26	28	$+\infty$

$y = 19,\ p = 3.$

$r = 15,\ l = 0$ (done by P_1).

1st iteration

$r - l > P.$

$l = 0,\ r = 15$

P1 checks $X[l] = X[0] = -\infty$

P2 checks $X[l + \frac{r-l}{p}] = X[5] = 10$

P3 checks $X[l + 2\frac{r-l}{p}] = X[10] = 20$

decide $X[5] < y < X[10]$.

2nd iteration.

$r - l > P$

$l = 5,\ r = 10.$

P1 checks $X[l] = X[5] = 10$

P2 checks $X[l + \frac{r-l}{P}] = X[7] = 14$

P3 checks $X[l + 2\frac{r-l}{P}] = X[9] = 18.$

decide $X[9] < y < X[10]$.

3rd iteration

$l = 9,\ r = 10.$

now $r - l \leq P$ check every element by making $(r-l)$ comparisions in parallel.

P1 checks $X[l] = X[9] = 18$

P2 checkes $X[l+1] = X[10] = 20.$

decide $i = 9$, ie $X[9] \leq y < X[10]$.

Note; Implementation of decide step can be done with an auxilliary array $C[1 \ldots p]$.

$$c(j) = \begin{cases} 1 & \text{if } y < X[\text{the element } P_j \text{ checks}] \\ 0 & \text{otherwise} \end{cases}$$

Look for 01 transition in C by checking $c_j < c_{j+1}$. This identifies the subarray to recurse on.

Complexity

- $S_i = r - l$.

$$S_{i+1} = \left(l + j\frac{r-l}{p}\right) - \left(l + (j-1)\frac{r-l}{p}\right)$$

$$= \frac{r-l}{p} = \frac{S_i}{p}.$$

$\therefore$ $S_{i+1} = \frac{n}{p^i}$ ⇒ if i is the number of iterations required so that size of final array is 1 ⇒ $i = \frac{\log n}{\log p} = \log_p n$.

CREW PRAM since all need to access y, l, r.

EREW PRAM. $\Omega(\log n - \log p)$ parallel time EREW. No advantage over sequential algorithm.

~~Not~~ **Optimality**
- each iteration takes $O(1)$ time
- each iteration takes $O(p)$ work

⇒ # operations $= O(\log_p n)$

$\therefore$ Time $= O(\log_p n)$

Work $= O\left(p \frac{\log n}{\log p}\right)$,

Not work optimal unless $p = O(1)$.

Merging two sorted sequences

Before we saw $O(\log n)$ time, $O(n)$ work parallel algorithm, ⇒ work optimal.

Ranking a short sequence (unsorted) in a long sorted sequence,

$X = (x_1, x_2 - - x_n)$ st. $x_i < x_{i+1}$.

$Y = (y_1 \cdot y_2 - - y_m)$ not necessarily sorted.

$m = O(n^s)$, $0 < s < 1$ (s is a constant).

Strategy / Idea.

Use parallel search algorithm to rank each element of Y in X separately.

- Set $p = \lfloor n/m \rfloor = \Omega(n^{1-s})$ for each element of Y.
- To rank each element of Y in X thus,

  time $O\left(\frac{\log n}{\log p}\right) = O\left(\frac{\log n}{\log n^{1-s}}\right) = O\left(\frac{1}{1-s}\right) = O(1)$.

  Work $O\left(p \cdot \frac{\log n}{\log p}\right) = O(p) = O\left(\frac{n}{m}\right)$.
- To do all rankings of elements of Y in X concurrently,

$$\boxed{\begin{array}{ll} \text{time} & O(1) \\ \text{Work} & O\left(m \cdot \frac{n}{m}\right) = O(n). \end{array}}$$

CREW PRAM is needed for parallel search.

A fast Merging Algorithm.

$A = (a_1 . a_2 \dots a_n)$ $a_i < a_{i+1}$

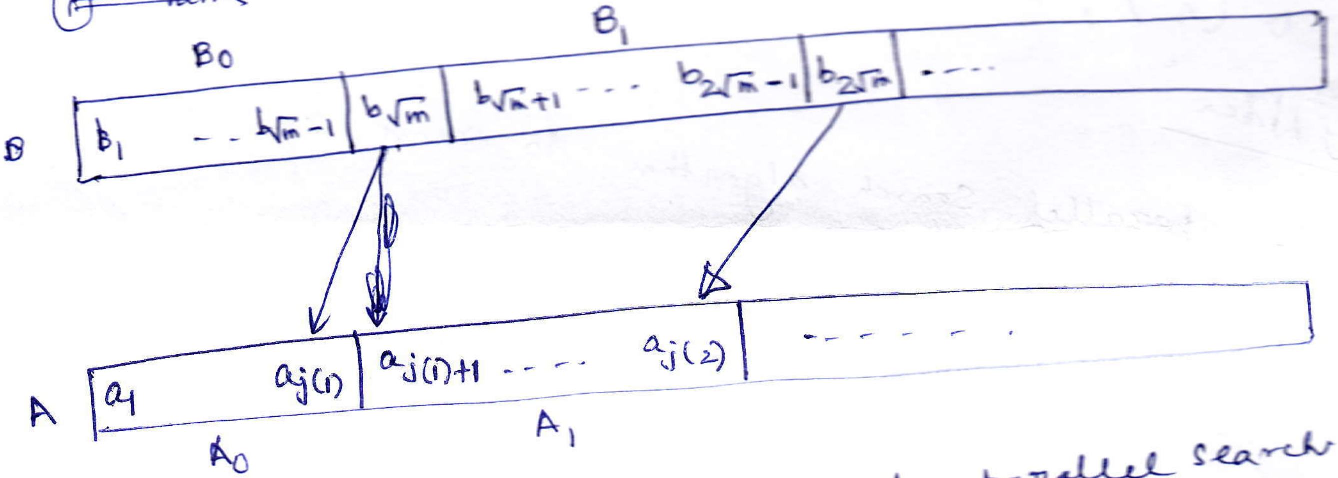
$B = (b_1 b_2 \dots b_m)$ $b_i < b_{i+1}$.

Want to rank $(B : A)$ i.e rank each element b_i in A..

[Reminder: rank $(x : X)$ is the number of elements of X that are less than or equal to x]

Use partitioning strategy (previously each partition size was $(\log m)$. — now $\sqrt{m}$).

① ~~rank~~



① rank $\sqrt{m}$ elements of B in A by parallel search.

② Partition A into blocks st. each block of A fits between two elements of B at most $\sqrt{m}$ elements apart. (Like A1 fits between $b_{\sqrt{m}}$ & $b_{2\sqrt{m}}$).

Thus problem reduced to ranking blocks of B in blocks of A.

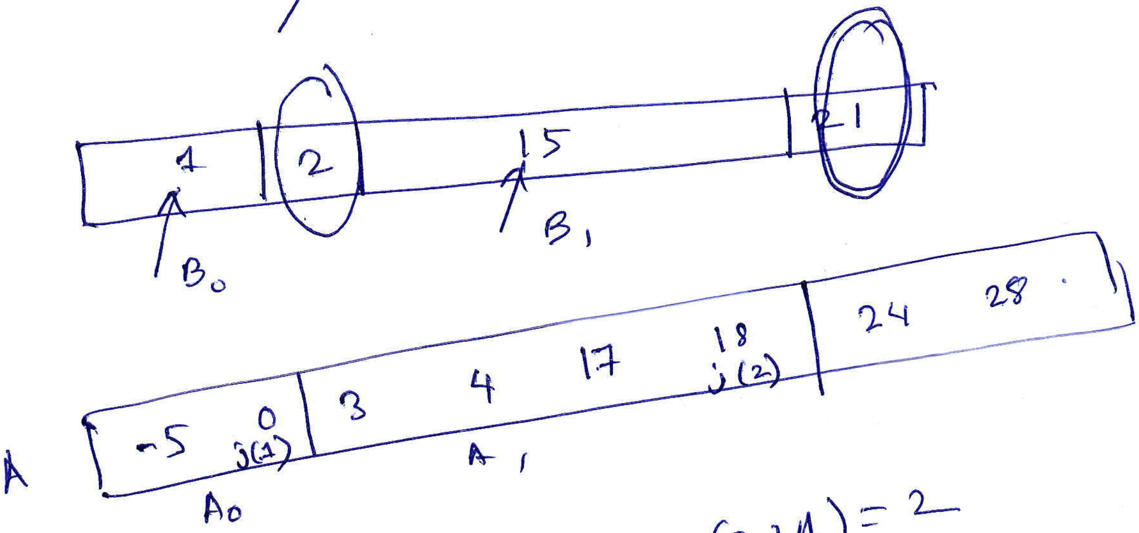
③ Stop recursion when $m < 4$ (rank m elements with $p = 3$ by parallel search).

Ex

$A = (-5 \quad 0 \quad 3 \quad 4 \quad 17 \quad 18 \quad 24 \quad 28)$

$B = (\overset{b_1}{1} \quad \overset{b_2}{2} \quad \overset{b_3}{15} \quad \overset{b_4}{21})$.

$m=4 \Rightarrow \sqrt{m} = 2$.



$j(0) = 0$.

$j(1) = \text{rank}(b_2 : A) = \text{rank}(2, A) = 2$

$j(2) = \text{rank}(b_4 : A) = \text{rank}(21, A) = 6$.

$\text{rank}(B_0 : A_0) = \text{rank}(1 : A_0) = 2$

$\text{rank}(B_1 : A_1) = \text{rank}(15 : A_1) = 2$.

$\text{rank}(1 : A) = j(0) + \text{rank}(1 : A_0) = 2$.

$\text{rank}(15 : A) = j(1) + \text{rank}(15 : A_1) = 2 + 2 = 4$.

∴ rank(B : A) is computed.

Likewise rank(A : B) can be computed.

Complexity   CREW PRAM because Parallel Search needs ~~it~~ —.

◦ ranking $\sqrt{m}$ elements in n elements.

$p = \sqrt{n}$ for each call to parallel search

Time: $O(\log n / \log \sqrt{n}) = O(1)$ time.

Work: $O\left(p \frac{\log n}{\log \sqrt{n}}\right) = O(p) = O(\sqrt{n})$.

Total time : $O(1)$, Total work : $O(\sqrt{m}\ \sqrt{n})$
$= O(m+n)$.

\# recursive calls

each time size of B reduces by $\sqrt{m}$.

$$T(m) = T(\sqrt{m}) + O(1)$$
$$= T(m^{1/4}) + 2O(1)$$
$$\doteq T(m^{1/2^k}) + k\,O(1)$$
$$3 = m^{1/2k} \Rightarrow \log 3 = \frac{1}{2^k} \log m.$$
$$\therefore 2^k = O(\log m)$$
$$\therefore k = O(\log \log m).$$

Total time = $O(\log\log m)$ [each iteration takes $O(1)$ time]

Total work : $O((n+m)\log\log m)$ [each iteration takes $O(n+m)$ work].

Simple parallel Merge Sort

→ size n

Merge Sort (A)

T1 = merge Sort (1st half of A) } in parallel
T2 = merge Sort (2nd half of A)
T3 = merge (T1 & T2) } use parallel merging $O(\log\log n)$ time, $O(n)$ work.

Complexity

$$T(n) = T(n/2) + M(n) = T(n/2) + O(\log\log n)$$
$$= O(\log n \log\log n).$$

$$W(n) = 2W(n/2) + O(n) = O(n \log n).$$

$\boxed{\text{Can we reduce this to } O(\log n)}$?

— Cole's Algorithm.