Searching, Merging, & Sorting  

Searching

given: sorted array \( X = (x_1, x_2, \ldots, x_n) \) s.t. \( x_i < x_{i+1} \)

and a value \( y \)

find: index \( i \) s.t. \( x_i \leq y < x_{i+1} \)  \( (x_0 = -\infty, x_{n+1} = +\infty) \)

Sequential binary search solves problem in \( \Theta(\log n) \) time

Can we do it faster w/ 1 < \( p \leq n \) processors? (Parallel Search)

basic idea: do “parallel comparison” by splitting \( X \)

into \( p \) equal pieces and restrict subsequent search in a subarray of size \( \frac{n}{p} \)

\( \Rightarrow \) recursive application finds \( i \) in time \( \approx \log_p n = O\left(\frac{\log n}{\log p}\right) \)

e.g. \( X = \begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\end{array} \)

\( y = 19 \) \( p = 3 \)

1st iter

\[
\begin{align*}
\ell &= 0, \quad r = 15 \\
&\text{P1 checks } X[\ell] = X[0] = -\infty \\
&\text{P2 checks } X[\ell + \frac{r-\ell}{p}] = X[5] = 10 \\
&\text{P3 checks } X[\ell + 2 \cdot \frac{r-\ell}{p}] = X[10] = 20 \\
&\text{decide } X[5] < y < X[10]
\end{align*}
\]

3rd iter

\[
\begin{align*}
\ell &= 9, \quad r = 10 \\
&\text{now } r-\ell \leq p \text{ check every elt.} \\
&\text{P1 checks } X[\ell] = X[9] = 18 \\
&\text{P2 checks } X[\ell + 1] = X[10] = 20 \\
&\text{decide } i = 9, \text{ i.e. } X[9] \leq y < X[10]
\end{align*}
\]

Note: Implement decide step w/ auxiliary array \( C[1..p] \) [the elt.]

- \( c(j) = \begin{cases} 1 & \text{if } y < X[p_j] \text{ checks} \\ 0 & \text{otherwise} \end{cases} \)

- 01 transition in \( C \) identifies subarray to recurse on

e.g. 11...100...00
Complexity

- $O(\log_p n)$ iterations, i.e. # times divide $n$ by $p$ to get subarray of size 1
- each iteration takes $O(1)$ time
  $\Rightarrow O(\log_p n) = O\left(\frac{\log n}{\log p}\right)$ time total

- each iteration takes $O(p)$ work
  $\Rightarrow O(p \cdot \frac{\log n}{\log p})$ work total

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

Lowerbound $\Omega(\log n - \log p)$ parallel time EREW PRAM

**NOTE:** In general, this is not a work optimal algorithm since sequential time is $O(\log n)$
- only work optimal if $p = O(1)$
Merging two sorted sequences

- Before (in Sect. 2.4) we saw $O(\log n)$ time, $O(n)$ work parallel algorithm
  $\Rightarrow$ work optimal

Ranking a short sequence in a long sorted sequence

$X = (x_1, x_2, x_3, \ldots, x_n)$ s.t. $x_i < x_{i+1}$
$Y = (y_1, y_2, \ldots, y_m)$ not necessarily sorted
$m = O(n^s)$ $0 < s < 1$

- Use parallel search algorithm to rank each element of $Y$ in $X$ separately
  - set $p = \lceil \frac{m}{\log n} \rceil = \Omega(n^{1-s})$ for each element
  - Can rank each element of $Y$ in $X$
    - time $O(\frac{\log n}{\log p}) = O(\frac{\log n}{\log n^{1-s}}) = O(\frac{1}{1-s} \frac{\log n}{\log n}) = O(1)$
    - work $O(p \cdot \frac{\log n}{\log p}) = O(p) = O(\frac{n}{m})$
  - do all rankings of elts of $Y$ in $X$ concurrently
    - time $O(1)$
    - work $O(m \cdot \frac{n}{m}) = O(n)$

CREW PRAM (needed for parallel search)
A Fast Merging Algorithm

\[ A = (a_1, a_2, \ldots, a_n) \quad a_i < a_{i+1} \quad \text{(assume all els distinct)} \]
\[ B = (b_1, b_2, \ldots, b_m) \quad b_i < b_{i+1} \]

Want \( \text{rank}(B; A) \) i.e. rank each elt \( b_i \) in \( A \)

Use partitioning strategy (2.4 in text)

1. rank \( \sqrt{m} \) els of \( B \) in \( A \) by parallel search
2. partition \( A \) into blocks s.t. each block of \( A \)
fits between two els of \( B \) (at most \( \sqrt{m} \) els apart)
   \( \Rightarrow \) problem reduced to ranking blocks of \( B \) in blocks of \( A \)
3. Stop recursion when \( m < 4 \) (rank \( m \) els w/p=n by parallel search)

**Complexity**

- **CREW PRAM b/c Parallel Search needs it**
  - ranking \( \sqrt{m} \) els in \( n \) els
    \[ p = \sqrt{n} \text{ for each call to parallel search} \]
    \[ O(\log n / \log \sqrt{m}) = O(1) \text{ time, } O(p \cdot \log^{\frac{\log n}{\log \sqrt{m}}}) = O(p) = O(\sqrt{m}) \text{ work} \]
    \( \Rightarrow \) Total time \( O(1) \), Total work \( O(\sqrt{m} \cdot \sqrt{n}) = O(m + n) \)
  - # recursive calls
    each time size of \( B \) reduced by \( \sqrt{m_i} \), \( M_i = \text{current size} \)
    \( \Rightarrow O(\log \log n) \) recursive calls
    
    **Total Time** = \( O(\log \log n) \) (each iter takes \( O(1) \) time)
    **Total Work** = \( O((n+m) \log \log n) \) (each iter takes \( O(nm) \) work)

**NOTE:** As before can make work optimal w/ accel. cascading.
Make smaller prob. of size \( \frac{n}{\log \log n} \) w/ slow work optimal algo then apply fast non-optimal algo
Sorting

Two algorithms that use the merge-sort strategy
- straightforward parallelization of sequential merge-sort
- Cole's parallel merge sort (pipelined divide-and-conquer)

Simple Parallel Merge Sort

\[
\text{MergeSort (A)} \\
T_1 := \text{MergeSort (1st half A)} \} \text{ in parallel} \\
T_2 := \text{MergeSort (2nd half A)} \} \\
T_3 := \text{Merge (T1 + T2)} \} \text{ use parallel merging algorithm} \\
\text{end At MergeSort */} \\
\]

Complexity: CREW PRAM (merging)
\[
T(n) = T(\frac{n}{2}) + M(n) = T(\frac{n}{2}) + O(\log \log n) = O(\log n \log \log n) \\
W(n) = 2 \cdot W(\frac{n}{2}) + O(n) = O(n \log n)
\]

* So a very simple work-optimal algorithm

We can use Cole's Parallel MergeSort to reduce time to \(O(\log n)\) while keeping work \(O(n \log n)\)
• in practice wouldn't want to implement Cole's algorithm
• it is an important theoretical result and introduces new pipe-lined technique
Cole's Parallel Merge Sort
- not in complete detail but it's important result so we want to give you flavor.
- In practice it's too complicated to yield good performance

Build binary tree on input (elts at leaves)
- Internal node \( v \) computes sorted list \( L[v] \) of input elts at leaves of subtree rooted at \( v \)

\[
L[-8, -7, -5, 3, 6, 12, 28, 51]
\]

\[
\begin{align*}
& L[-5, 12] \\
& L[-7, 51] \\
& L[6, 28] \\
& L[-8, 3]
\end{align*}
\]

Simple Parallel Merge Sort
- Forward (bottom to top) traversal of tree
- Completely compute \( L[v] \) at level \( i \) before going up to next

Cole's Parallel Merge Sort (pipelined or cascading merge sort)
- Forward traversal
- Compute \( L[v] \) over a number of stages
  - At stage \( s \), \( L_s[v] \) is an approximation of \( L[v] \) that is improved at next stage \( s+1 \)
  - A sample (every \( c \)th elt) of \( L_s[v] \) is propagated upward to be used to improve approximations above

def: Altitude \( alt(v) = h(T) - level(v) \) (\( alt(root) = h(T) \))
**initialize:** $L_o[v] = \emptyset$ if $v$ internal
$\quad L_o[v] =$ value at leaf $v$ if $v$ is leaf

The alg. updates $L[v]$ at stages $alt(v) \leq s \leq 3 \cdot alt(v)$
- after stage $3 \cdot alt(v)$ $L[v]$ is full or finished

$\Rightarrow$ algorithm runs in $3 \cdot h(T)$ stages
  which is $O(\log n)$ if each stage runs in $O(1)$ time

At stage $s$:

for all active nodes $v$ pardo  | $\forall alt(v) \leq s \leq 3 \cdot alt(v)$  

1. let $u + w$ be children of $v$
   $L'_{sv}[u] =$ Sample ($L_s[u]$)
   $L'_{sv}[w] =$ Sample ($L_s[w]$)

2. Merge $L'_{sv}[u]$ and $L'_{sv}[w]$ into sorted list $L_{sv}[v]$

end pardo

\[ \text{every } 4^{\text{th}} \text{ elt of } L_s \quad \text{ every other elt of } L_s \]

where $\text{Sample} (L_s[x]) = \begin{cases} 
\text{Sample}_y (L_s[x]) & \text{if } s \leq 3 \cdot \text{alt}(x) \\
\text{Sample}_2 (L_s[x]) & \text{if } s = 3 \cdot \text{alt}(x) + 1 \\
\text{Sample}_1 (L_s[x]) & \text{if } s = 3 \cdot \text{alt}(x) + 2 \\
\end{cases} \\
\text{every elt of } L_s$

- Correctness not difficult to see
- More complicated to see $O(1)$ time per stage
  - depends on relation between samples + ranks in two consecutive stages
  - by maintaining ranking info between kids/parents can do merging in step 2 in $O(1)$ time
Randomized Sorting Algorithms (Ch. 9 Sect 6)

random sampling - select random subset of a set of elts
sampling w/ replacement - always select elts from entire set

Randomized Quicksort (elts assumed distinct)
- two-way partitioning, very simple divide-and-conquer alg

RQSort(A)
1. if |A| < c
   then sort A sequentially + return
2. x := random elt of A
3. for 1 ≤ i ≤ n pardo
   if A(i) < x
       then mark(i) := 1
       else mark(i) := 0
   end for pardo
4. Rearrange A s.t. elts marked w/ 1 at front,
   then x, then elts marked w/ 0
   * let k be index of x in rearranged A
5. Recursively sort in parallel A(1:k-1) + A(k+1:n)
end /* RQSort */

Complexity
# iterations: worst-case O(n), average-case=best-case = O(\log n)
Steps 1-3: O(i) time, O(n) work
Step 4: prefix sums/compaction O(\log n) time, O(n) work
Total Time: O(\log^2 n)
Total Work: O(n \log n) EREW PRAM (duplicate x in O(\log n) time)
Randomized Sample Sort
- k-way divide-and-conquer

RSSort(A)
1. if |A| ≤ 30
   then sort A sequentially + return
2. choose a sample S of k random elts of A
3. Sort the elts in S = (s₁, s₂, ..., sₖ) s.t. sᵢ < sᵢ₊₁
4. Rearrange elts of A into k+1 buckets Bᵢ, 1 ≤ i ≤ k+1
   s.t. elts x ∈ Bᵢ satisfy sᵢ₋₁ < x ≤ sᵢ (s₀ = -∞, sₖ₊₁ = +∞)
5. Recursively sort elts in Bᵢ
end /* RRSort */

Analyze Complexity w/ k = √n

Claim: Each iteration takes O(log n) time, O(n log n) work w/ |Bᵢ| = n

Step 1: O(1) time, O(1) work

Step 2: O(1) time, O(√n) = O(k) work

Step 3: First, how to sort (brute force is sufficient)
(i) compare all O(k²) = O(√n²) = O(n) pairs of elts of S
(ii) store results in k × k (√n × √n) lookup table T
(iii) compute rank of each elt of S by a prefix sum computation on each row of T

Now complexity...
(i) + (ii): O(1) time, O(k²) = O(√n²) = O(n) work
(iii): O(log k) = O(log √n) = O(log n) time, O(k) = O(√n) work
   for each row of T, k = √n rows total
⇒ Total time O(log n)
Total work O(√n · √n) = O(n)
Step 4: Two Stages

Stage 1: Locate bucket for each elt of A by binary search on sorted S
- \(O(\log n)\) time, \(O(n\log n)\) work

Stage 2: Rearrange els so those in bucket 1 1st, 2 2nd, etc.
- Note: this is integer sorting where n ints belong to range \([1, 2, \ldots, \sqrt{n}]\)
- Briefly, build binary tree on input els (ex. 9.28 in text)
  - Each node has lists of els in each bucket
  - As proceed up tree, concatenate list from kids &
- Takes \(O(\log n)\) time, \(O(n\log n)\) work

\[\frac{1}{4}\times\text{claim x/}\]

# Iterations
- Expected \# els in each bucket is \(\sqrt{n} = n^{1/2}\) after \(I\) iter
- At \(i^{th}\) iteration \# els expected in bucket is \(n^{(\frac{1}{2})^i}\)
\[\Rightarrow\text{expected \# iters } O(\log \log n)\]

Total Time (expected)
- Time of \(i^{th}\) iteration = \(O(\log (n^{(\frac{1}{2})^i}))\)
- Total time = \(O(\sum \log n^{(\frac{1}{2})^i}) = O(\sum (\frac{1}{2})^i \log n) = O(\log n \sum (\frac{1}{2})^i) = O(\log n)\)

Total Work (expected)
- Work of \(i^{th}\) iteration = \(O(n^{(\frac{1}{2})^i} \log n^{(\frac{1}{2})^i})\)
- Total work = \(O(\sum n^{(\frac{1}{2})^i} \log n^{(\frac{1}{2})^i}) = O(n \log n)\)

Note: The text shows these bounds hold w/ high probability, i.e., probability \(1 - n^{-c}\), which is much greater than expected probability of \(\frac{1}{2}\).