Techniques for Parallel Algorithm Design

CS 263

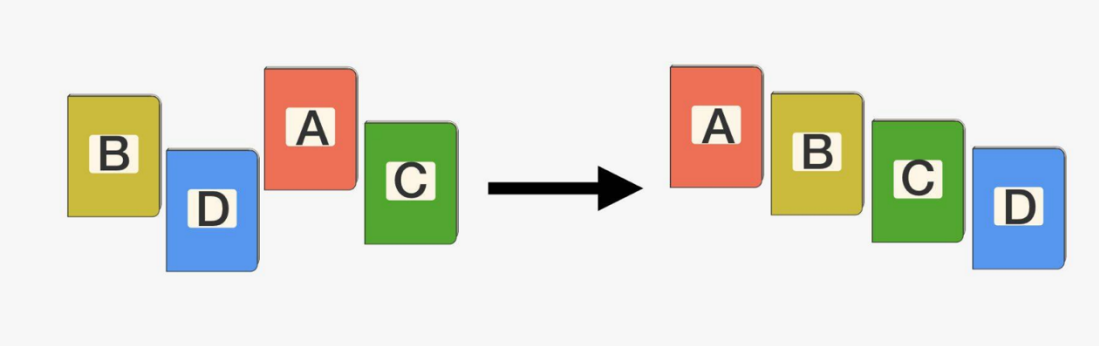
Michael T. Goodrich

University of California, Irvine

Some slides adapted from content by Joseph Jaja, Yun Gu, and Yihan Sun

Parallel Sorting

- Given an array of elements $\boldsymbol{A}$ with size $\boldsymbol{n}$, and a total order defined, create an array $\boldsymbol{B}$ of the same elements in $\boldsymbol{A}$, with:
  - $B[0] \leq B[1] \leq B[2] \leq \cdots \leq B[n-1]$

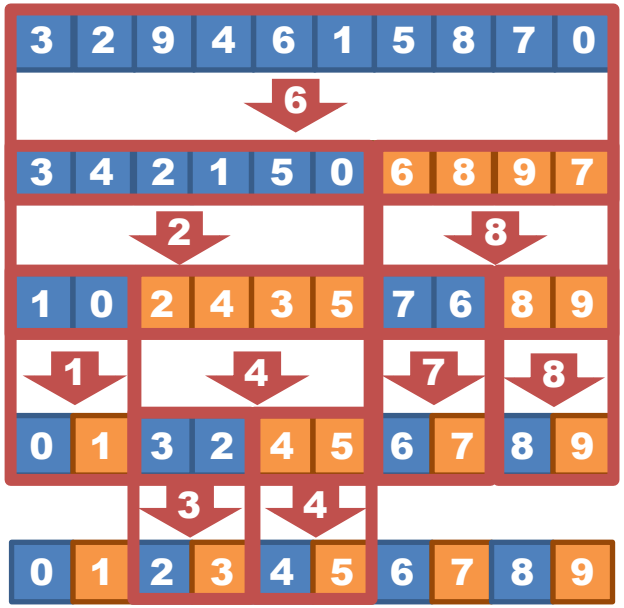


Sorting Algorithms

- Given an array of elements $\boldsymbol{A}$ with size $\boldsymbol{n}$, and a total order defined, create an array $\boldsymbol{B}$ of the same elements in $\boldsymbol{A}$, with:
  - $B[0] \leq B[1] \leq B[2] \leq \cdots \leq B[n-1]$
- Recall sequential sorting bounds
  - Lower bound in the comparison model: $\Omega(n \log n)$ comparisons needed (and achievable)
  - Implies that a parallel algorithm requires $\Omega(n \log n)$ work

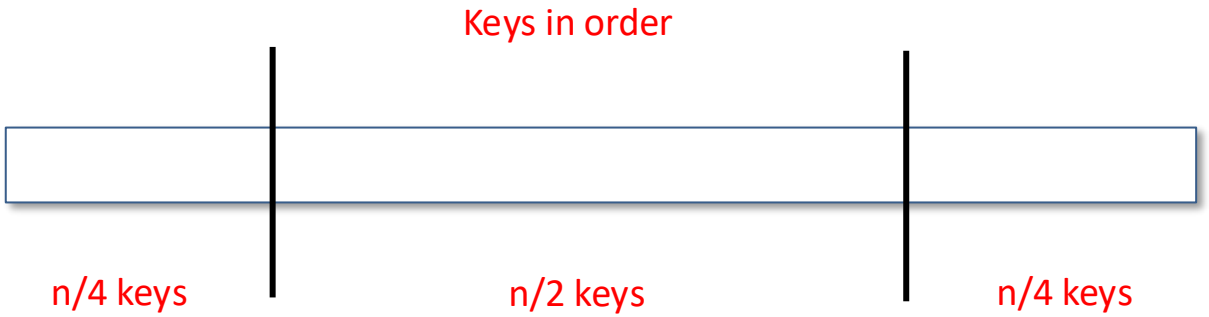
Recall Randomized Quicksort

- Find a random pivot x in the array
- Put all elements in A that are $\leq$ than x on the left, and all elements in A that are > than x on the right



- **Runs in O(n log n) time whp**

Quicksort cost analysis



- Pivot is chosen uniformly at random
- 1/2 chance that pivot falls in middle range, in which case sub-problem size is at most 3n/4
- Expected #rounds: $O(\log n)$ (also w.h.p., with high probability)
  - $x = O(f(n))$ w.h.p. means that $\Pr[x > cf(n)] \geq 1 - \frac{1}{n^c}$
  - E.g., the probability that a quicksort doesn't finish in $5\log n$ rounds is no more than $1/n^5$
- Each round need $O(n)$ time (partition)
- In total $O(n \log n)$ time
- Important observation: The depth of the recursion tree is O(log n)

Sequential quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively

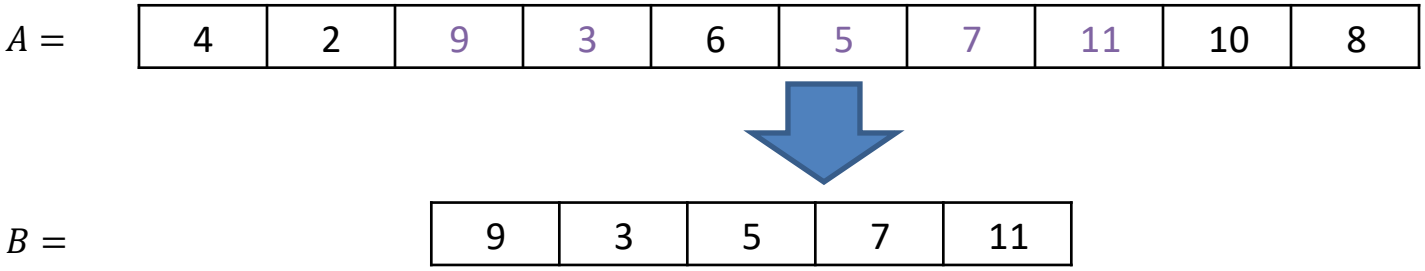
Parallel quicksort

- Use a pivot and partition the array into two parts
- Sort each of them recursively, in parallel

Parallel filtering / packing

- Given an array $\boldsymbol{A}$ of elements and a predicate function $\boldsymbol{f}$, output an array $\boldsymbol{B}$ with elements in $\boldsymbol{A}$ that satisfy $\boldsymbol{f}$

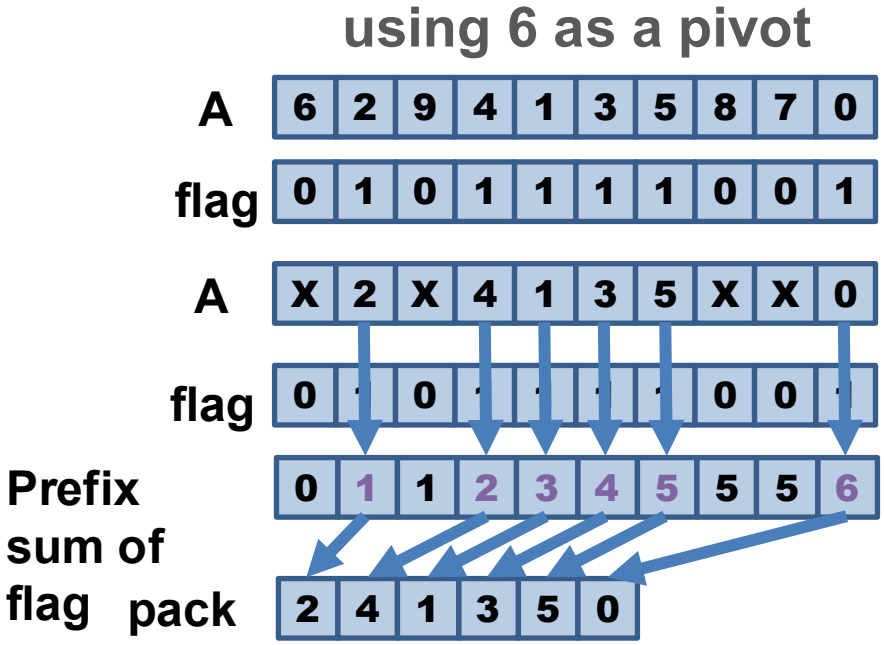
$$f(x) = \begin{cases} true & if\ x\ is\ odd \\ false & if\ x\ is\ even \end{cases}$$



We can use parallel prefix to solve this in O(log n) span and O(n) work.

Using prefix-sum filter for partition

- How to move elements around? The filter algorithm!



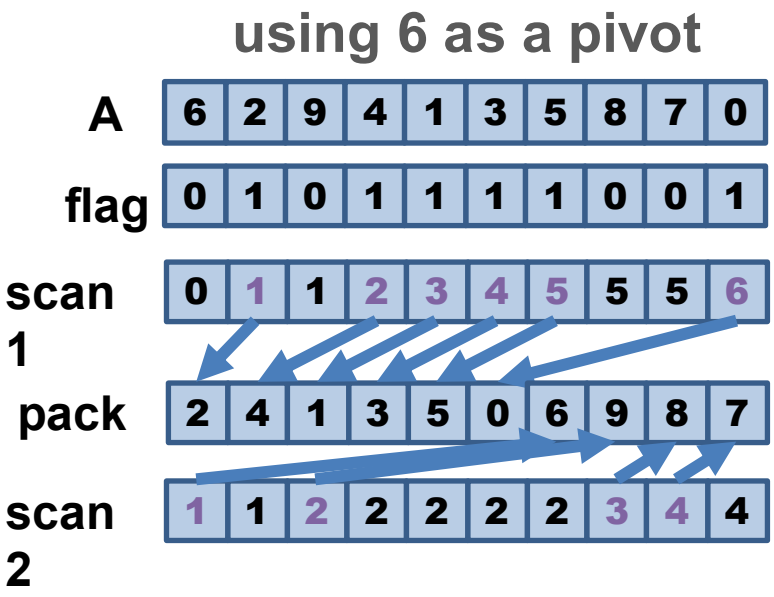

```
filter(A, flag, n) {
  ps = scan(flag);
  parallel_for(i=1 to n) {
    if (ps[i]!=ps[i-1])
      B[ps[i]] = A[i];
  }
}
```

- **$O(n)$ work for one round**

Using filter to partition

- To get all elements smaller than the pivot and all elements larger than the pivot
  - We can run two separate filters
  - Two rounds of I/O and global data movement
- Parallel partition
  - After doing the first scan, we know the result of the second scan!

Using filter for partition



scan1[]: the prefix sum of 1s
scan2[]: the prefix sum of 0s
=> scan1[i] + scan2[i] = i
=> scan2[i] = i - scan1[i]

Parallel quicksort

- Using the filter algorithm to do partition

- Finishes in $\boldsymbol{O(\log n)}$ rounds in expectation (also w.h.p.)
- Each round need $\boldsymbol{O(n)}$ work and $\boldsymbol{O(\log n)}$ depth
- $\boldsymbol{O(n \log n)}$ work and $\boldsymbol{O(\log^2 n)}$ depth in total

Sequential mergesort

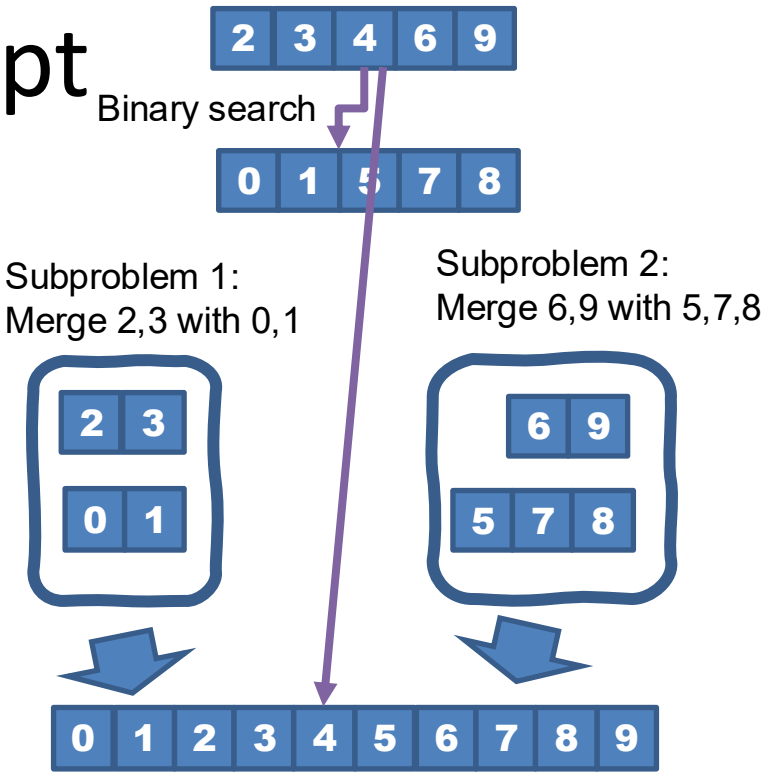
- Split the array evenly in two
- Sort each of them recursively
- Merge them back

Parallel mergesort

- Split the problem size evenly in two
- Sort each of them recursively, in parallel
- Merge them back

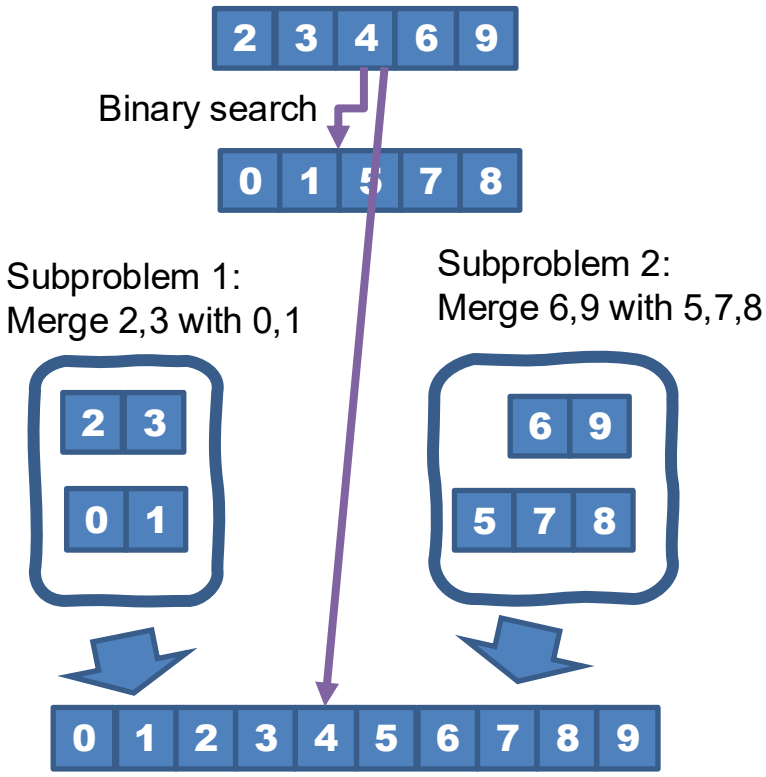
A parallel merge algorithm: First attempt

- Find the median m of one array
- Binary search it in the other array
- Put m in the correct slot
- Recursively, in parallel do:
  - Merge the left two sub-arrays into the left half of the output
  - Merge the right ones into the right half of the output



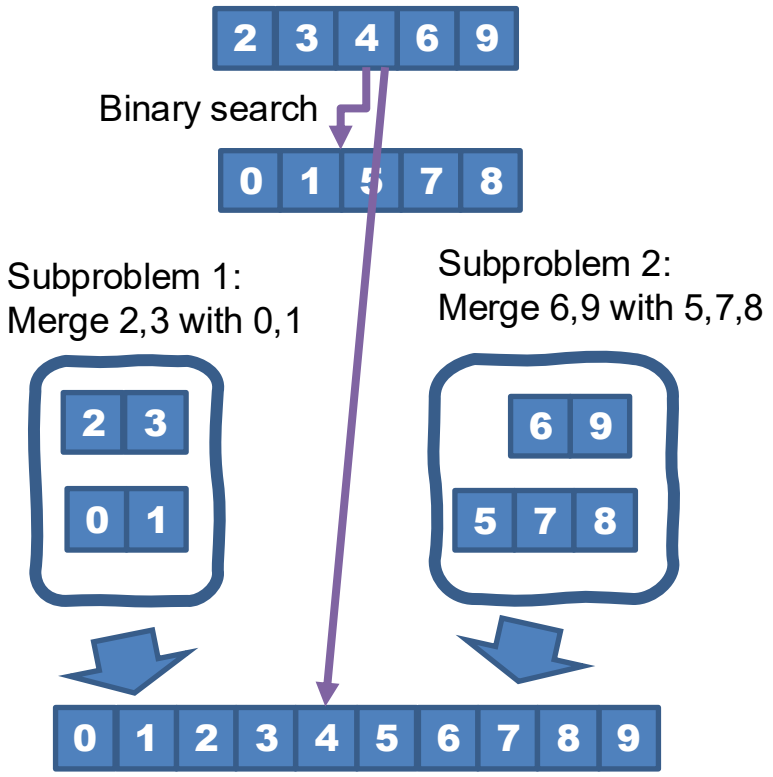
A parallel merge algorithm

```
//merge array A of length n1 and array B of length
n2 into array C.
Merge(A', n1, B', n2, C) {
  if (A' is empty or B' is empty) base_case;
  m = n1/2;
  m2 = binary_search(B', A'[m]);
  C[m+m2+1] = A'[m];
  in parallel:
    merge(A', m, B', m2, C);
    merge(A'+m+1, n1-m-1, B'+m2+1, n2-m2-1, C+m+m2);
  return C;
}
```



A parallel merge algorithm

- In each recursive call the only work is the binary search
- Assume the original input arrays are $\boldsymbol{A}$ and $\boldsymbol{B}$. They are both of the same size $\boldsymbol{n}$.
- Assume in each recursive call, we are dealing with $\boldsymbol{A'} \subseteq \boldsymbol{A}$ and $\boldsymbol{B'} \subseteq \boldsymbol{B}$, they can have different sizes.
- Array $\boldsymbol{A'}$ from $\boldsymbol{A}$ is always perfectly partitioned, but it's not the case for array $\boldsymbol{B'}$. But, as long as $\boldsymbol{A'}$ is empty, we reach the base case.
- So in log n rounds we reach the base case. In each round the cost is also O(log n)



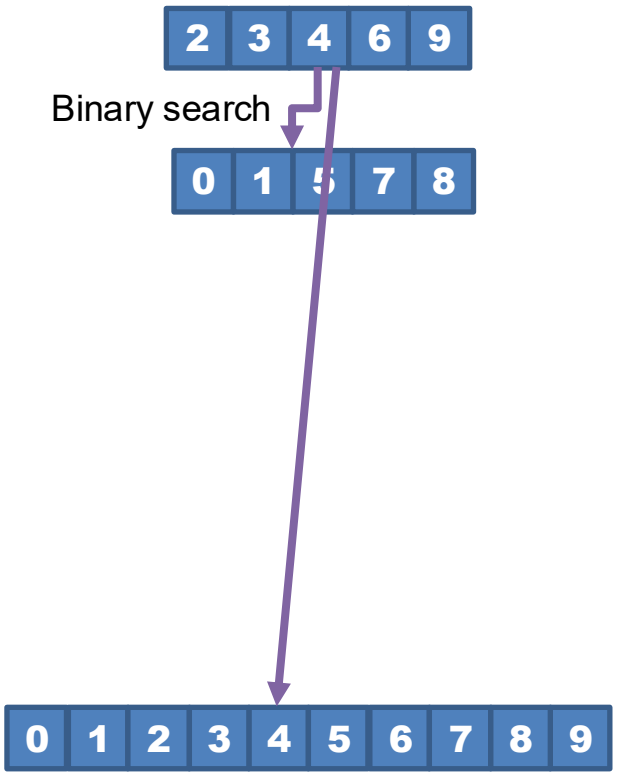
- $D(N) = O(\log^2 N)$

Parallel merge sort

- Parallel merge: $\boldsymbol{O(n)}$ work and $\boldsymbol{O(\log^2 n)}$ depth
- Finishes in $\boldsymbol{O(\log n)}$ rounds
- Total work: $\boldsymbol{O(n \log n)}$, depth: $\boldsymbol{O(\log^3 n)}$

A parallel merge algorithm: Second attempt

- For each element in each array, binary search for it in the other array
- Put the element in the correct slot
- O(log n) time, but O(n log n) work

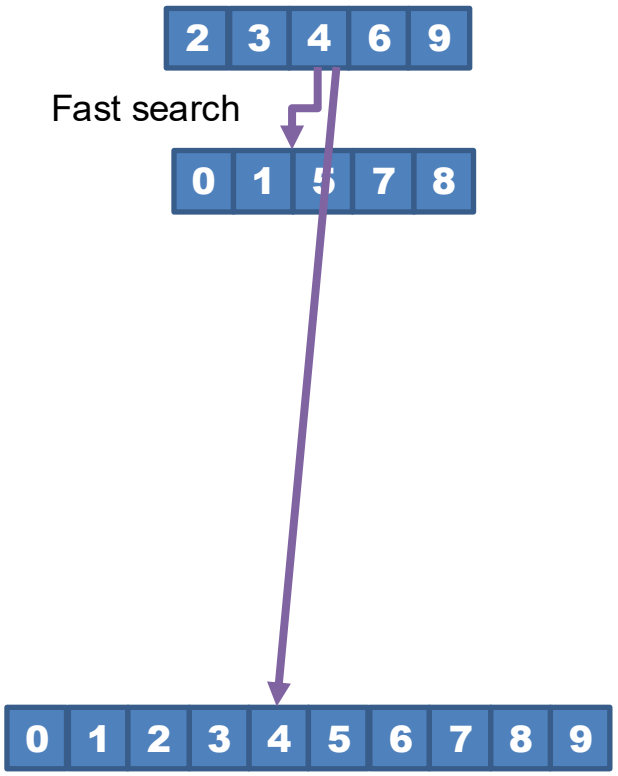


Parallel merge sort: Second Attempt

- Parallel merge: $\boldsymbol{O(n \log n)}$ work and $\boldsymbol{O(\log n)}$ depth
- Finishes in $\boldsymbol{O(\log n)}$ rounds
- Total work: $\boldsymbol{O(n \log^2 n)}$, depth: $\boldsymbol{O(\log^2 n)}$

A parallel merge algorithm: Third attempt (for CREW PRAM)

- For each element in each array, do a fast search to compute its rank in merged array
- Put the element in the correct slot



Parallel merging: $O(n^2)$ work and $O(1)$ depth

- **For two arrays of size n, there are $O(n^2)$** pairs of elements
  - For each element x_i, compare x_i against all of the other elements.
  - There is a unique pair (y_j, y_{j+1}) such that $x_i < y_j$ and $xi > y_{j+1}$.
  - This gives us all information needed – write x_i to output B[i+j].
- The work is $\boldsymbol{O(n^2)}$ since we need to compare all the pairs
- The depth is $\boldsymbol{O(1)}$
- It uses lots of concurrent reads, but no concurrent writes

Valiant's Merge Algorithm

- Use this fast merging technique along with square-root-way divide-and-conquer.

1. Choose $n^{1/2}$ evenly spaced elements from array A and $n^{1/2}$ evenly spaced elements from array B.
2. Merge these two sampled lists in O(1) time using the constant-time merge method.
3. Recursively merge the $O(n^{1/2})$ pairs of overlapping subproblems of size $O(n^{1/2})$ each.

CREW PRAM Merging Algorithm

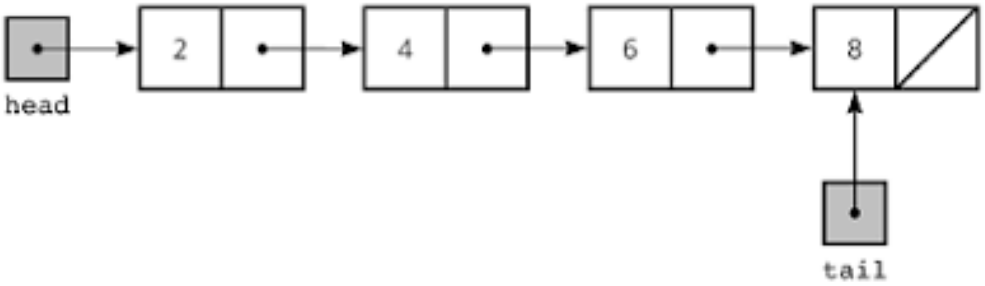
- Time: $T(n) = T(n^{1/2}) + 1$, which implies $T(n)$ is $O(\text{loglog } n)$.
- Work: $W(n) = n^{1/2}\, W(n^{1/2}) + n$, which means $W(n)$ is $O(n \text{ loglog } n)$.
- We can get the work down to $O(n)$ by stopping when the problem size is $O(\text{loglog } n)$ and doing the merges sequentially.

Parallel Mergesort

- Use the fast merge algorithm as a part of a binary parallel mergesort.
- O(log n) levels of recursion, which take O(log log n) time each.
- Total work: O(n log n).

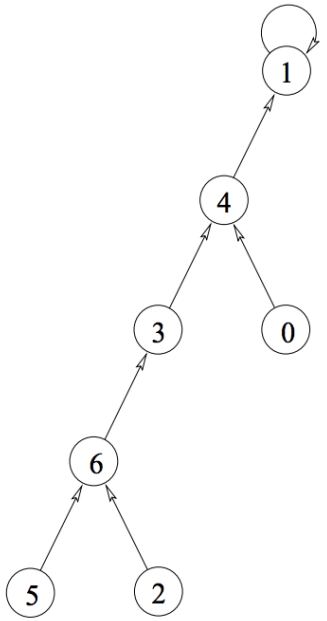
Linked Lists

- Linked lists are simple and important data structures



- Sometimes we want to know the rank of each node (e.g., the distance to the head/tail)

List Ranking



(a) The input tree $P = [4, 1, 6, 4, 1, 6, 3]$.

- Input array P, P[i]=j means that the i-th element's parent is the j-th element
- In practice the input can be a linked list with next/parent pointers
- Follow the pointers until reaching the root

Source: "Parallel Algorithms" by Guy E. Blelloch and Bruce M. Maggs

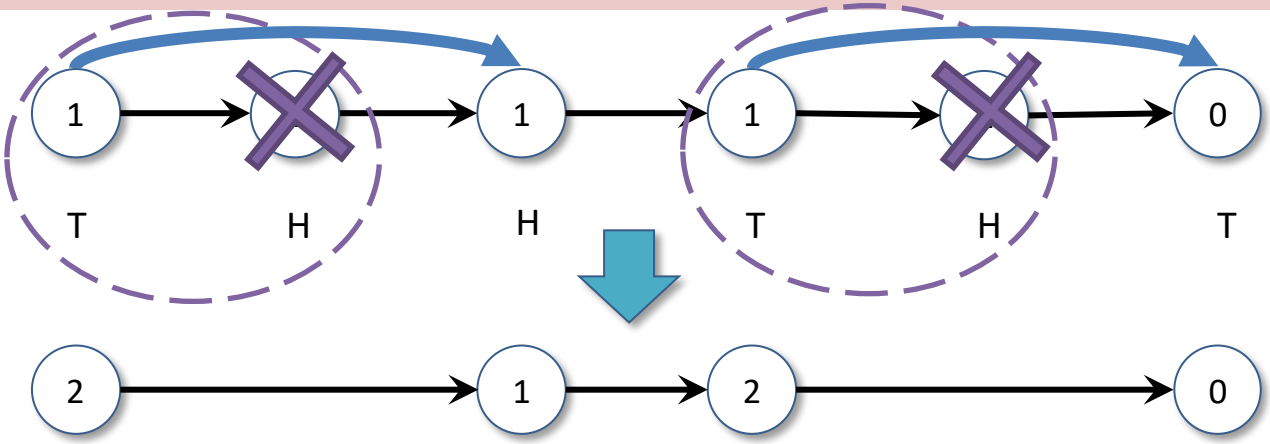
Work-Efficient List Ranking

Idea: reduce the problem size by a constant factor per round, and apply the algorithm recursively

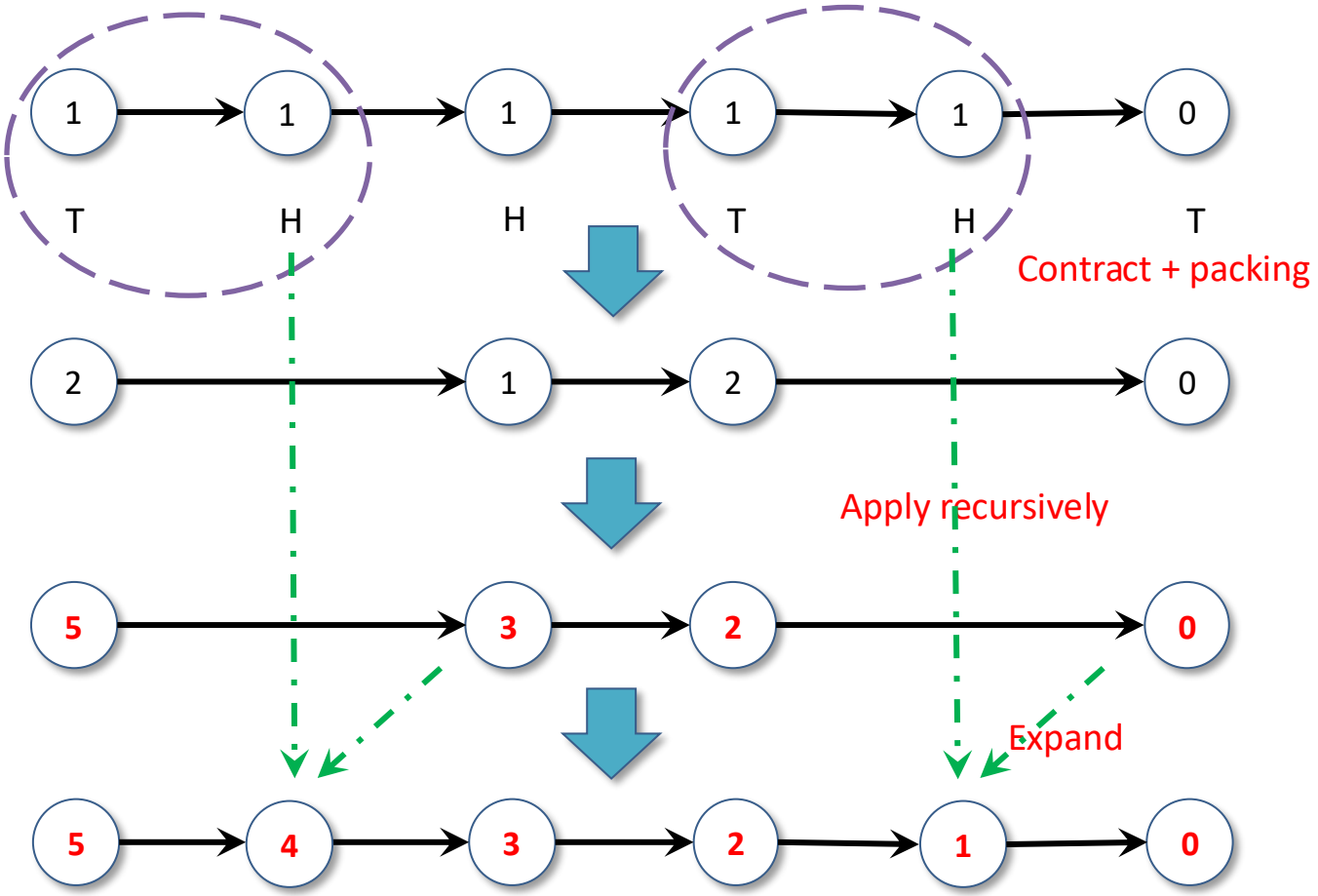
ListRanking(list P)

1. If list has two or fewer nodes, then return //base case
2. Every node flips a fair coin
3. For each vertex u (except the last vertex), if u flipped Tails and P[u] flipped Heads then u will be paired with P[u]
   A. rank(u) = rank(u)+rank(P[u])
   B. P[u] = P[P[u]]
4. Recursively call ListRanking on smaller list
5. Insert contracted nodes v back into list with rank(v) = rank(v) + rank(P[v])

Remove an element if:
It is head
Its previous element is a tail



Work-Efficient List Ranking



Source: MIT 6.886 by Julian Shun

Work-Depth Analysis

Remove an element if:
It is head
Its previous element is a tail

- Number of pairs per round is reduced by (n-1)/4 in expectation
  - For all nodes u except for the last node, probability of u flipping Head and its previous element flipping Tails is 1/4
  - => A node gets removed with probability 1/4
- Each round takes linear work and O(log n) depth
- Expected work: W(n) ≤ W(3n/4) + O(n)
- Expected depth: D(n) ≤ D(3n/4) + O(log n)

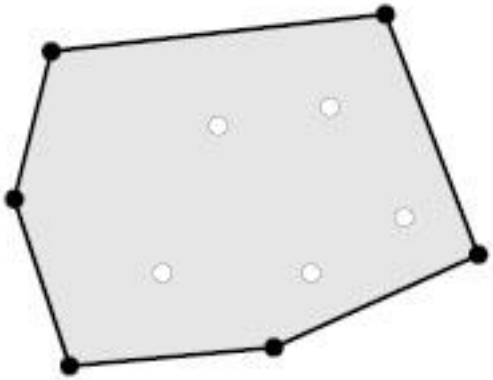
$D = O(\log n)$ in arbitrary-forking

W = O(n)
D = $O(\log^2 n)$

Source: MIT 6.886 by Julian Shun

Convex Hull

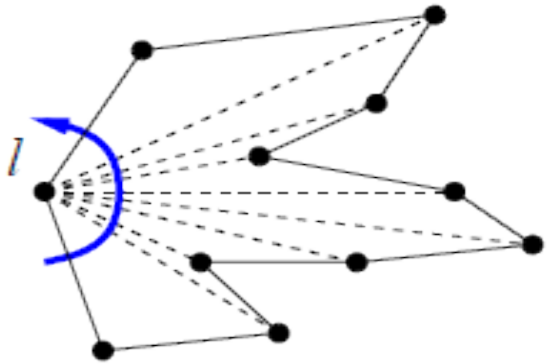
- Given a set of points in a plane P , convex hull is the largest convex polygon whose vertices are all in P.



A set of points and its convex hull.
Convex hull vertices are black; interior points are white.

Serial Algorithms

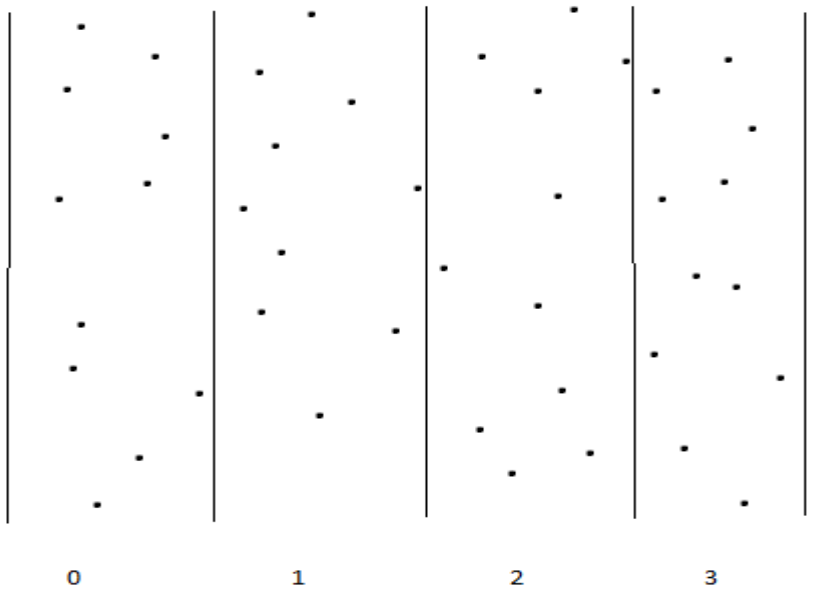
- Brute force
- Divide and Conquer
- Graham Scan
  - Select the left most point as pivot
  - Sort the rest of the points by polar angles with respect to the pivot
  - March around this points and build the hull
  - Add edges when left turn and backtrack when right



Parallel Algorithm

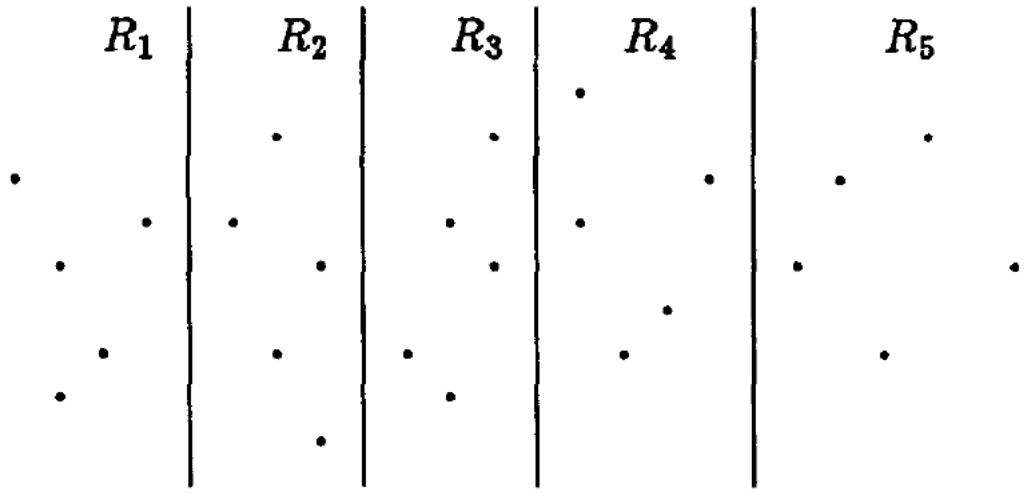
Divide and Conquer

- Divide the plane containing the points among the processors
- Sequentially find the local convex hull
- Merge the convex hull from neighboring processors



Convex Hulls and Divide-and-Conquer

- Divide into $n^{1/2}$ groups of size $n^{1/2}$ each.
- Recursively solve each problem
- Merge the subproblems
- $T(n)=T(n^{1/2}) + O(\log n)$
- $T(n)=O(\log n)$



Divide and Conquer

Merging $n^{1/2}$ upper convex hull

Assign each upper convex hull $n^{1/2}$ processors to find the upper common tangents in O(log n) time, and then determine the edges which belong to the solution.