CSE 332: Data Structures and Parallelism

Spring 2022

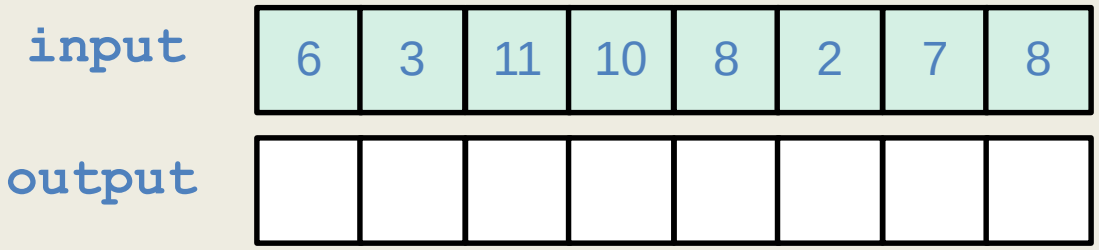
Richard Anderson

Lecture 18: Parallel Algorithms

Announcements

Parallelizable?

- Prefix-sum:

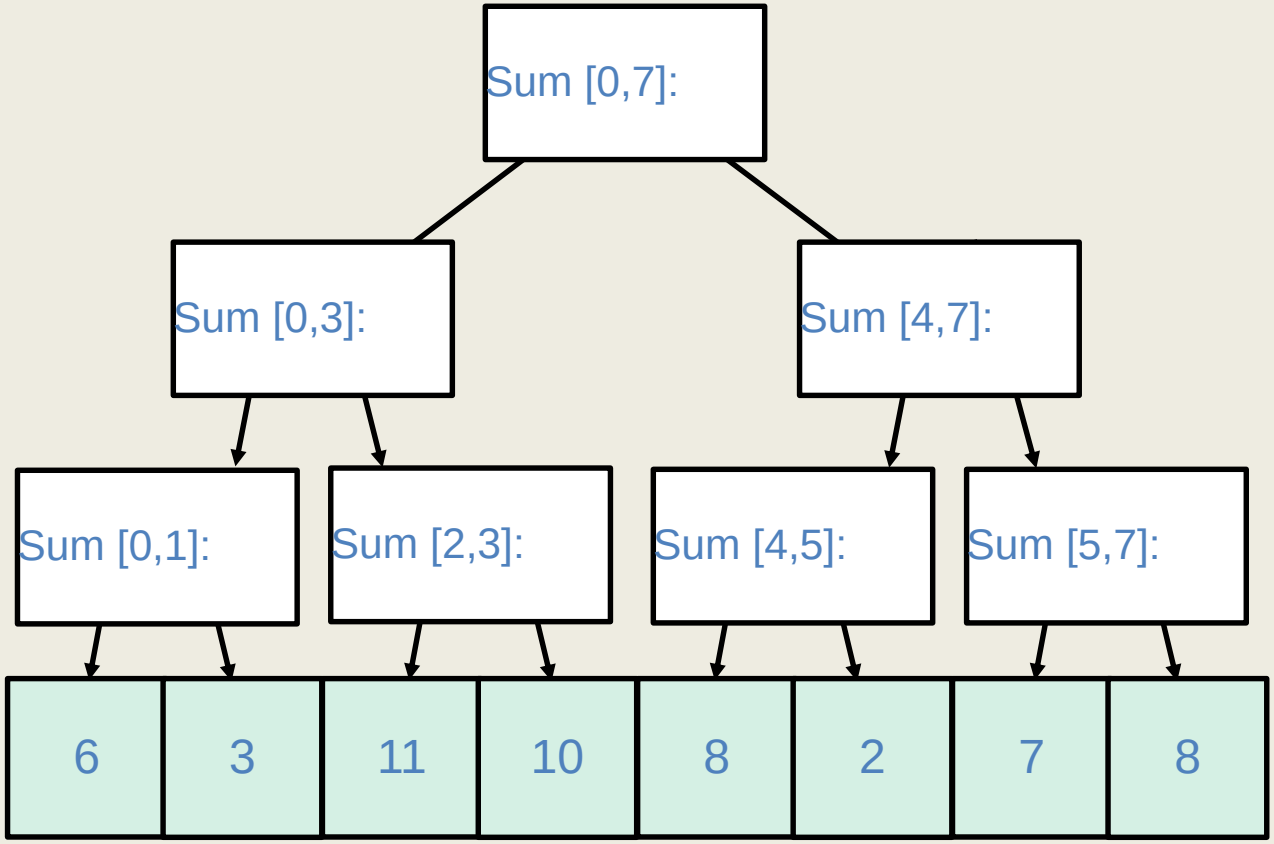


- $\text{output}[j] = \sum_{i=0}^{j} \text{input}[i]$

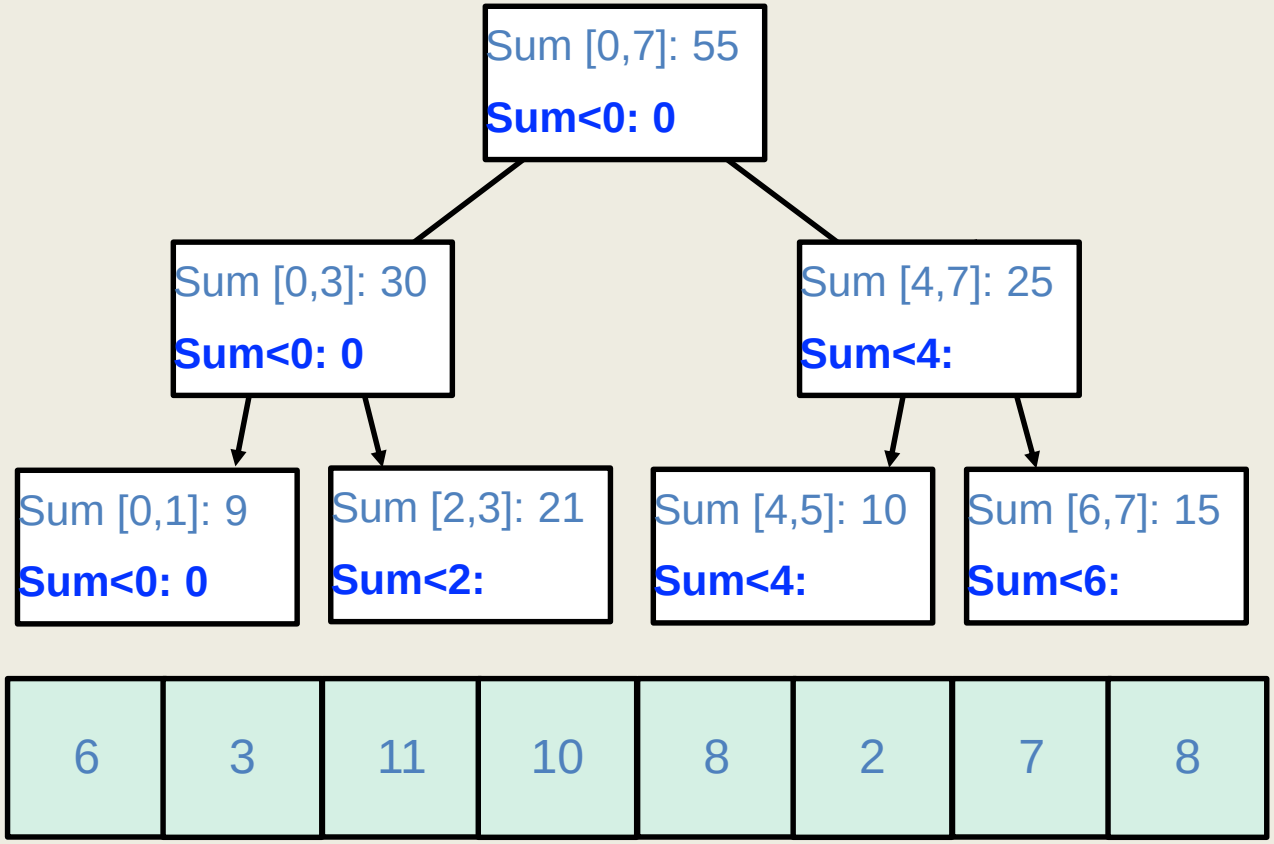
Parallel prefix-sum

- The parallel-prefix algorithm does two passes
  - Each pass has $O(n)$ work and $O(\log n)$ span
  - So in total there is $O(n)$ work and $O(\log n)$ span
- First pass builds a tree bottom-up: the "up" pass
  - Compute the some over each subtree
- Second pass traverses the tree top-down: the "down" pass
  - Pass the values compute in left

First Pass: Sum

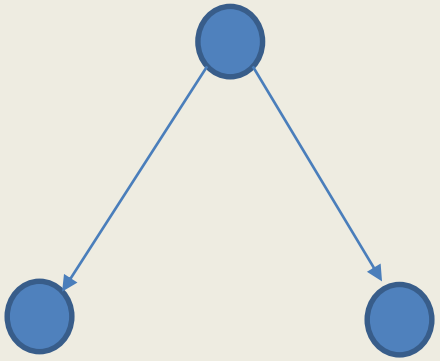


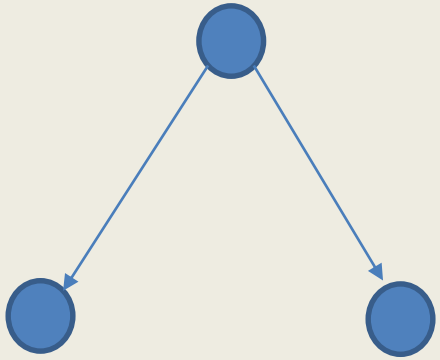
2nd Pass: Use Sum for Prefix-Sum



A nodes computation

sum = left.sum + right.sum





left.fromLeft = fromLeft

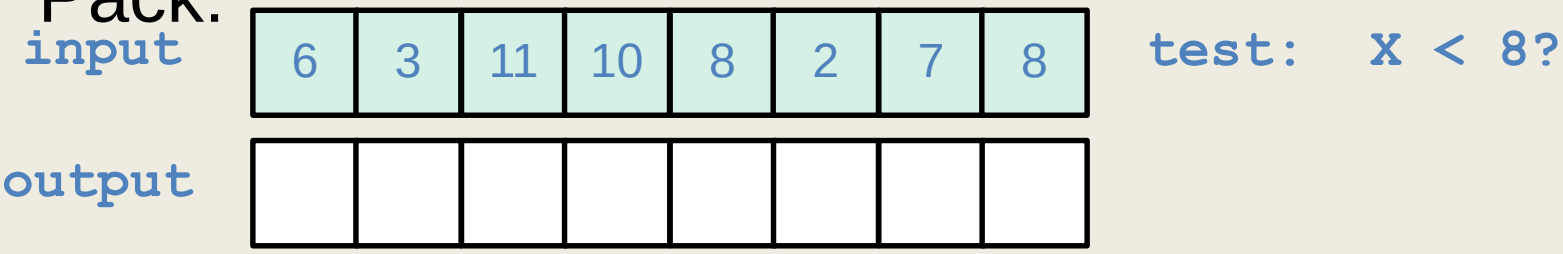
right.fromLeft = fromLeft + left.sum

Parallel Prefix, Generalized

- Prefix-sum is another common pattern (prefix problems)
  - maximum element to the left of i
  - is there an element to the left of i satisfying some property?
  - count of elements to the left of i satisfying some property
  - …
- We can solve all of these problems in the same way

Pack

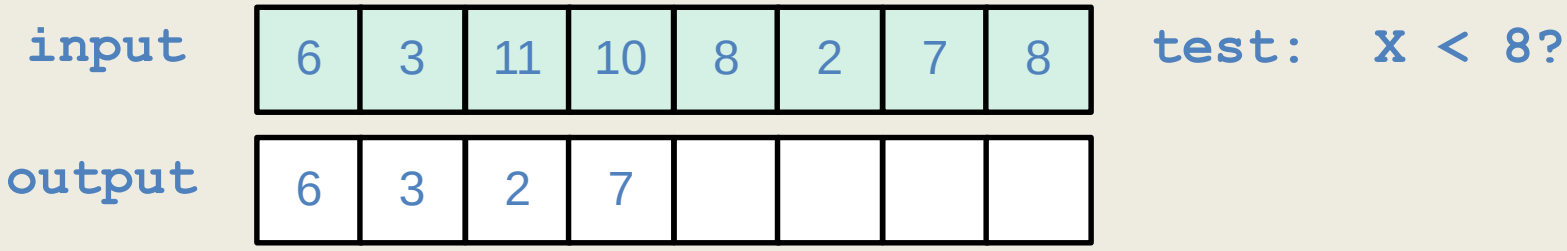
- Pack:



- Output array of elements satisfying **`test`**, in original order

Parallel Pack?

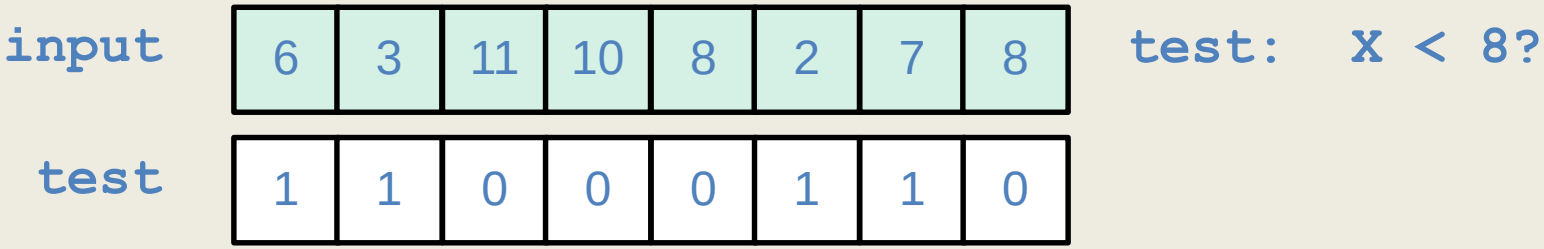
- Pack



- Determining **which** elements to include is **easy**
- Determining **where** each element goes in output is **hard**
  - seems to depend on previous results

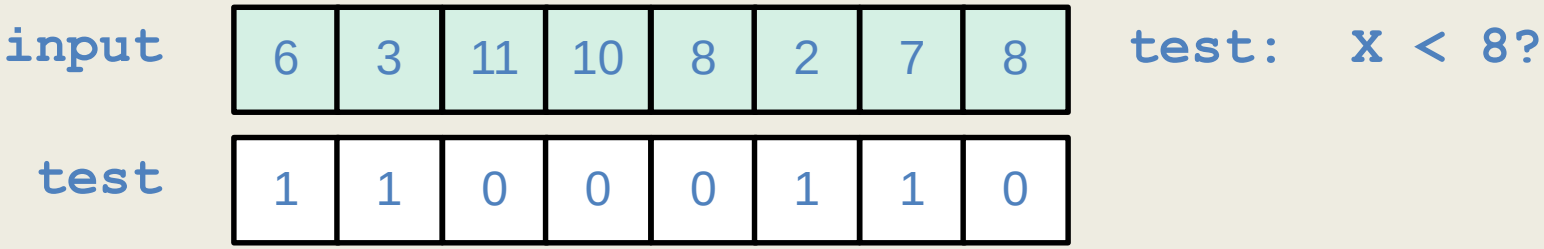
Parallel Pack

1. map test input, output [0,1] bit vector



Parallel Pack

1. map test input, output [0,1] bit vector



2. transform bit vector into array of indices into result array

Parallel Pack

1. map test input, output [0,1] bit vector

input	6	3	11	10	8	2	7	8	**test: X < 8?**
test	1	1	0	0	0	1	1	0	

2. prefix-sum on bit vector

pos	**1**	**2**	2	2	2	**3**	**4**	4

3. map input to corresponding positions in output

output	6	3	2	7				

- `if (test[i] == 1) output[pos[i]] = input[i]`

Parallel Pack Analysis

- Parallel Pack
  1. map: O( ) span
  2. sum-prefix: O( ) span
  3. map: O( ) span
- Total: O( ) span

Sequential Quicksort

- Quicksort (review):
  1. Pick a pivot — O(1)
  2. Partition into two sub-arrays — O(n)
     A. values less than pivot
     B. values greater than pivot
  3. Recursively sort A and B — 2T(n/2), approx

- Complexity
  - T(n) = n + 2T(n/2) T(0) = T(1) = 1
  - O(n logn)

- How to parallelize?

Avoiding bad cases for quicksort

- Quicksort can be $\Omega(n^2)$ with bad pivot choices
- If input is random then Quicksort is O(n log n) with high probability
- If pivots are random then Quicksort is O(n log n) with high probability
- Pick 5 elements at random, choose the middle as a pivot

Parallel Quicksort

- Quicksort
  1. Pick a pivot — O(1)
  2. Partition into two sub-arrays — O(n)
     A. values less than pivot
     B. values greater than pivot
  3. Recursively sort A and B in parallel — T(n/2), avg

- Complexity (avg case)
  - T(n) = n + T(n/2) T(0) = T(1) = 1
  - Span: O( )
  - Parallelism (work/span) = O( )

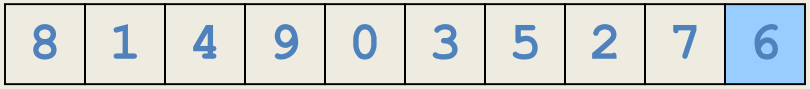
Parallel Partition

- Partition into sub-arrays
  A. values less than pivot
  B. values greater than pivot

- What parallel operation can we use for this?

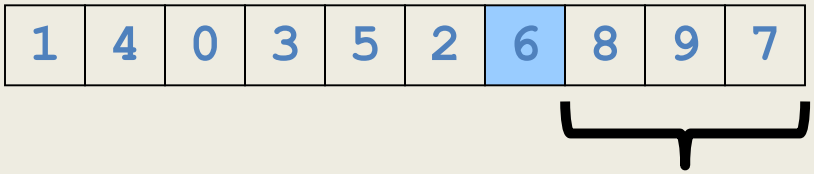
Parallel Partition

- Pick pivot



- Pack (test: **<6**)

- Right pack (test: **>=6**)



Parallel Quicksort

- Quicksort
  1. Pick a pivot — O(1)
  2. Partition into two sub-arrays — O( ) span
     A. values less than pivot
     B. values greater than pivot
  3. Recursively sort A and B in parallel — T(n/2), avg

- Complexity (avg case)
  - T(n) = O( ) + T(n/2) T(0) = T(1) = 1
  - Span: O( )
  - Parallelism (work/span) = O( )

20

Implementation

- Recommend random selection of pivot
- Choose sequential cutoff
  - Change over to sequential quick sort
- Constant factors in partitioning are higher for the parallel version

Sequential Mergesort

- Mergesort (review):
  1. Sort left and right halves — 2T(n/2)
  2. Merge results — O(n)

- Complexity (worst case)
  - T(n) = n + 2T(n/2) T(0) = T(1) = 1
  - O(n logn)

- How to parallelize?
  - Do left + right in parallel, improves to O(n)
  - To do better, we need to…

Parallel Mergesort

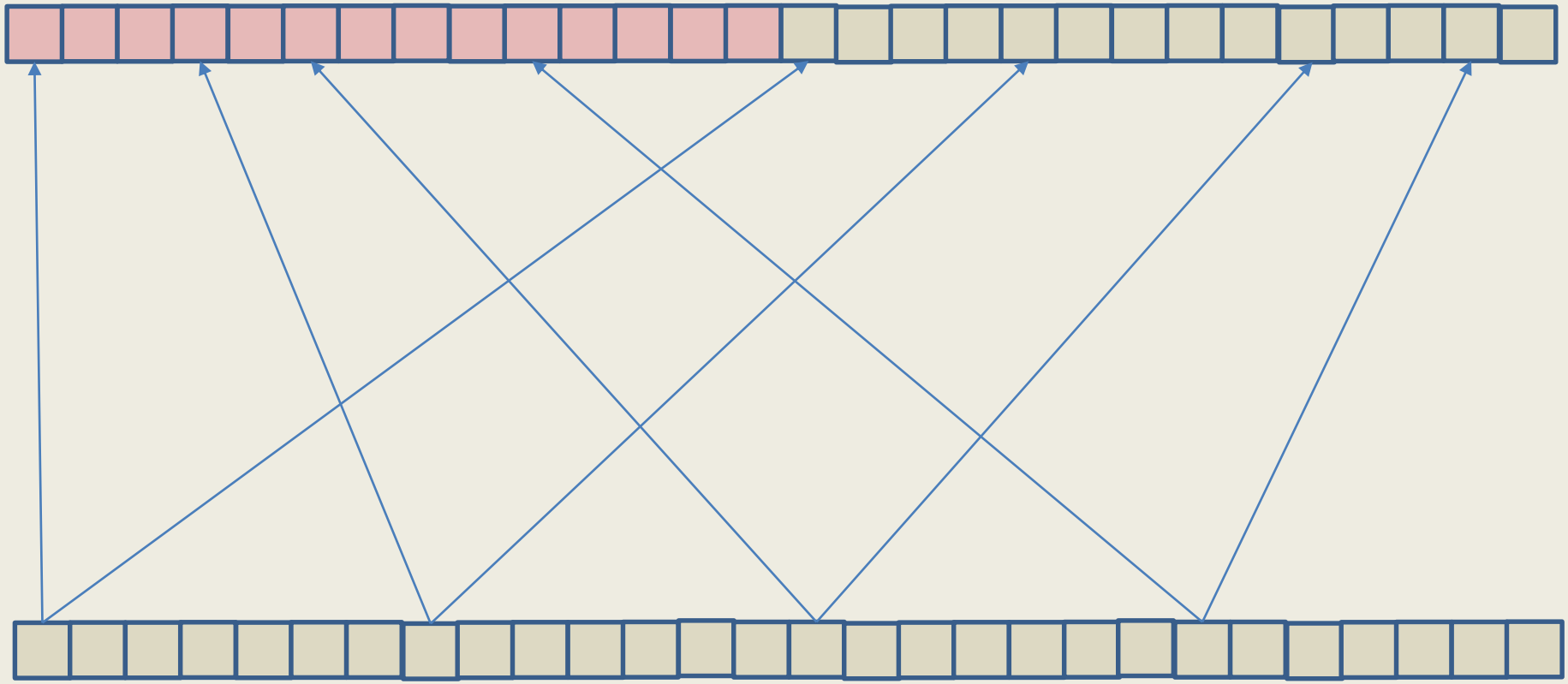
- MergeSort(Arr, lo, hi)
  - Threads to compute MS(Arr, lo, mid), MS(Arr, mid, hi)
  - Merge Arr[lo,mid] and Arr[mid,hi] into Arr[lo,hi]

- Can stop at a sequential cut off

Parallel Merge

- How to merge two sorted lists in parallel?

Parallel Merge



Parallel Merge: n items with p threads

- Each thread needs to know where to start in the two arrays being merged
- If starting points are given, select the next n/p items

- Finding the starting points can be done in O(log n) time using a modified binary search

Finding the starting point

- Given two sorted arrays A, B, find the item of rank k in the combined arrays
- Compare A[k/2] and B[k/2]
  - If A[k/2] < B[k/2] discard first k/2 items of A, otherwise discard first k/2 items of B
- Look for item of rank k/2 in remaining items
- Logarithmic process

Parallel Quicksort and Mergesort

- Both algorithms can be implemented as efficient parallel algorithms
- With p processors, a speedup of p is achievable provided p << n
- Speedup comes from:
  - Doing much of the work on sorting items below the sequential cutoff
  - Taking advantage of parallelism in the combine steps to avoid a sequential bottlenec,