



# CSE 332: Data Structures & Parallelism

## Lecture 16: Parallel Prefix, Pack, and Sorting

Ruth Anderson

Winter 2021

# Outline

Done:

- Simple ways to use parallelism for counting, summing, finding
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- **Parallel prefix:**
  - This “key trick” typically underlies surprising parallelization
  - Enables other things like **packs (aka filters)**
- **Parallel sorting:** quicksort (not in place) and mergesort
  - Easy to get a little parallelism
  - With cleverness can get a lot

# The prefix-sum problem

Given `int[] input`, produce `int[] output` where:

$$\text{output}[i] = \text{input}[0] + \text{input}[1] + \dots + \text{input}[i]$$

input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76

Sequential can be a CSE142 exam problem:

```
int[] prefix_sum(int[] input) {
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i=1; i < input.length; i++)
        output[i] = output[i-1]+input[i];
    return output;
}
```

Does not seem parallelizable

- Work:  $O(n)$ , Span:  $O(n)$
- *This algorithm* is sequential, but a *different algorithm* has Work:  $O(n)$ , Span:  $O(\log n)$

# *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
  - So like with array summing, parallelism is  $n/\log n$ 
    - An exponential speedup
- First pass builds a tree bottom-up: the “up” pass
- Second pass traverses the tree top-down: the “down” pass

# *Local bragging*

Historical note:

- Original algorithm due to R. Ladner and M. Fischer at UW in 1977
- Richard Ladner joined the UW faculty in 1971 and hasn't left



1968? 1973?



recent

# *Parallel Prefix: The Up Pass*

We build want to build a binary tree where

- Root has sum of the range  $[x,y)$
- If a node has sum of  $[lo,hi)$  and  $hi > lo$ ,
  - Left child has sum of  $[lo,middle)$
  - Right child has sum of  $[middle,hi)$
  - A leaf has sum of  $[i,i+1)$ , which is simply  $input[i]$

It is critical that we actually create the tree as we will need it for the down pass

- We do not need an actual linked structure
- We could use an array as we did with heaps

**Analysis of first step:** Work =                      Span =

# *The algorithm, part 1*

Specifically.....

1. Propagate 'sum' up: Build a binary tree where
  - Root has sum of `input[0] .. input[n-1]`
  - Each node has sum of `input[l0] .. input[hi-1]`
    - Build up from leaves; `parent.sum=left.sum+right.sum`
  - A leaf's sum is just its value; `input[i]`

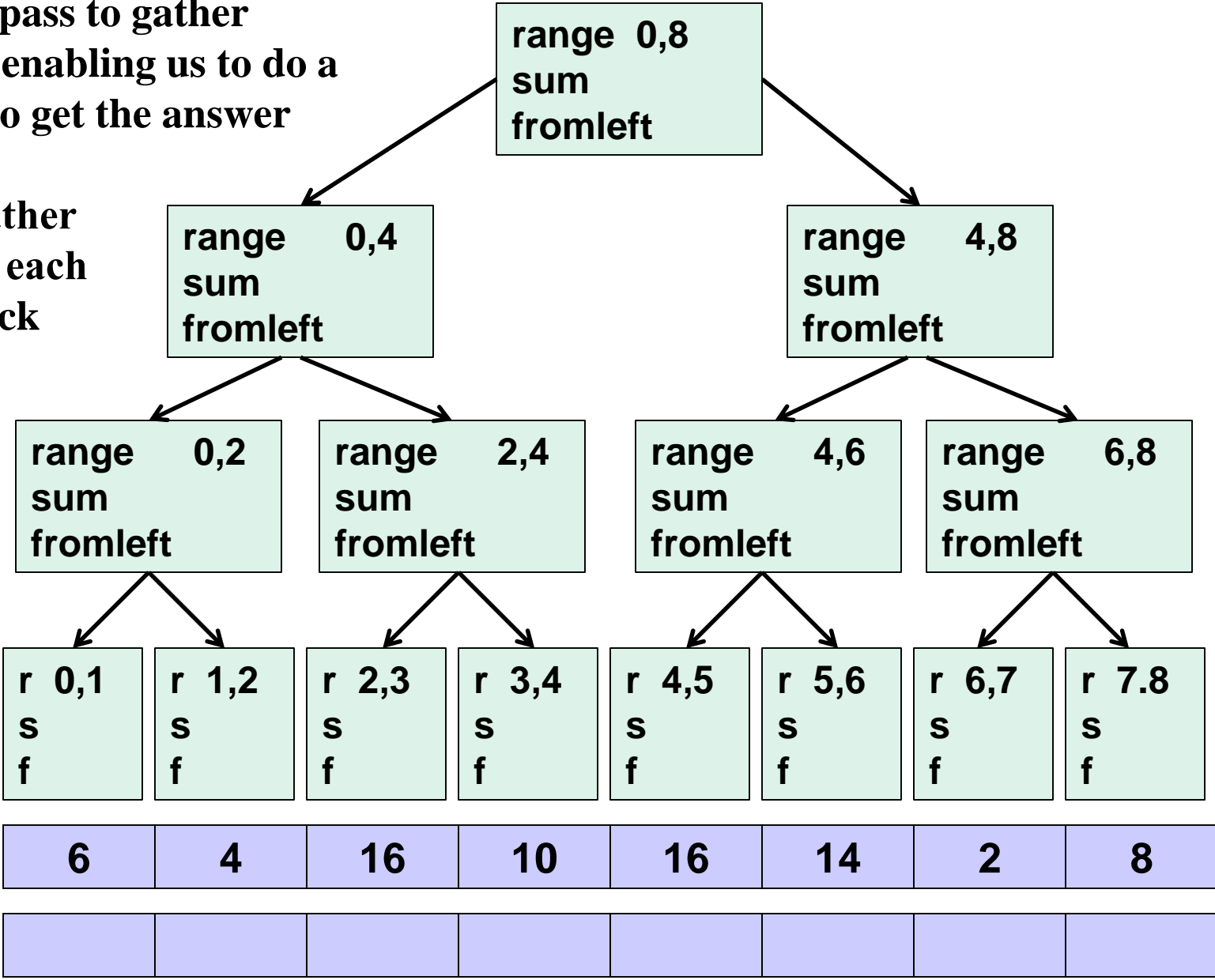
This is an easy fork-join computation: combine results by actually building a binary tree with all the sums of ranges

- Tree built bottom-up in parallel
- Could be more clever; ex. Use an array as tree representation like we did for heaps

**Analysis of first step:  $O(n)$  work,  $O(\log n)$  span**

**The (completely non-obvious) idea:**  
**Do an initial pass to gather information, enabling us to do a second pass to get the answer**

**First we'll gather the 'sum' for each recursive block**

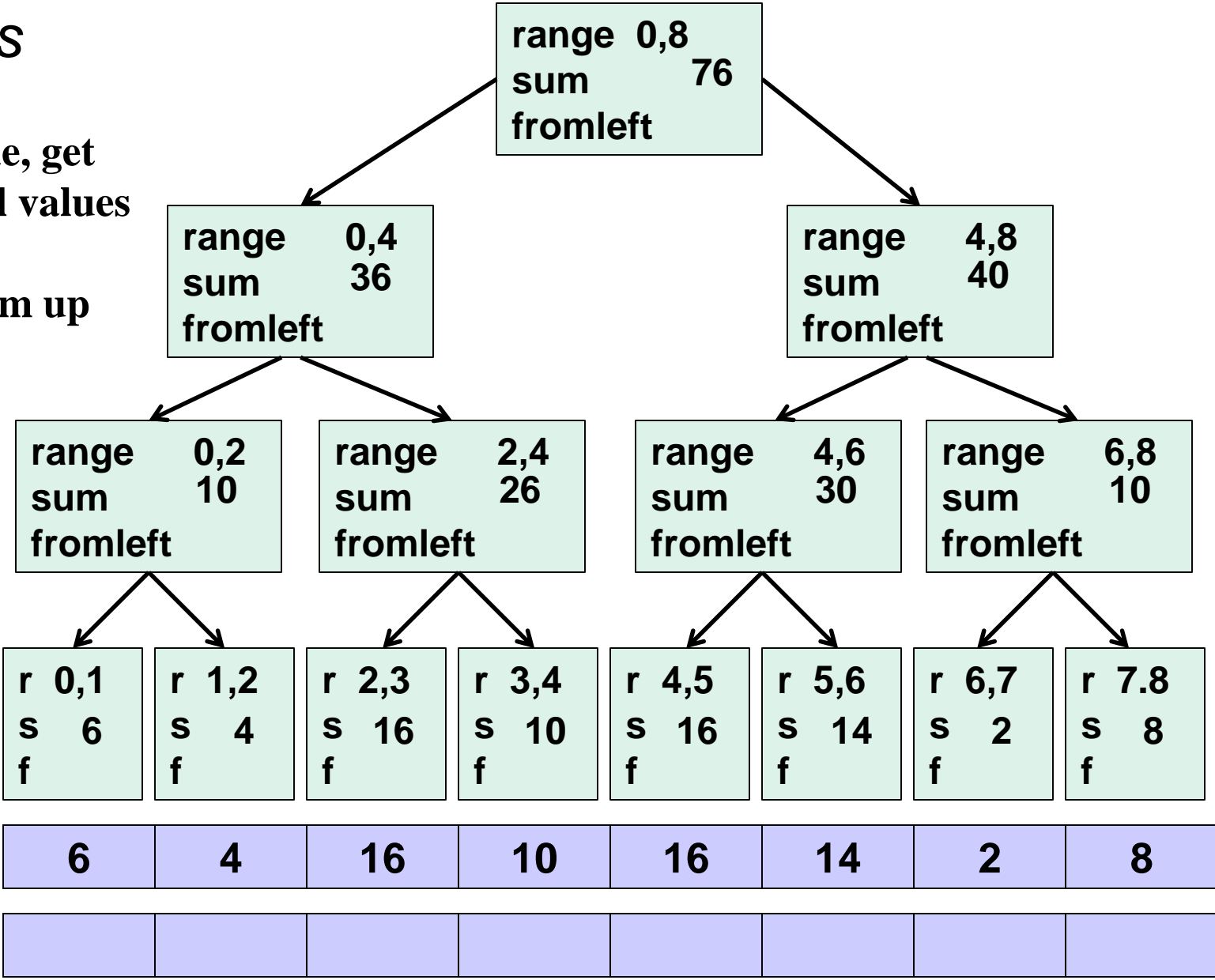




# First pass

For each node, get the sum of all values in its range; propagate sum up from leaves

Will work like parallel sum, but recording intermediate information



## *The algorithm, part 2*

2. Propagate 'fromleft' down:

- Root given a **fromLeft** of 0
- Node takes its **fromLeft** value and
  - Passes its left child the same **fromLeft**
  - Passes its right child its **fromLeft** plus its left child's **sum** (as stored in part 1)
- At the leaf for array position **i**,  
**output[i]=fromLeft+input[i]**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result (the leaves assign to **output**)

- Invariant: **fromLeft** is sum of elements left of the node's range

Analysis of first step:  $O(n)$  work,  $O(\log n)$  span

Analysis of second step:

**Total for algorithm:**

## *The algorithm, part 2*

2. Propagate 'fromleft' down:

- Root given a **fromLeft** of 0
- Node takes its **fromLeft** value and
  - Passes its left child the same **fromLeft**
  - Passes its right child its **fromLeft** plus its left child's **sum** (as stored in part 1)
- At the leaf for array position **i**,  
**output[i]=fromLeft+input[i]**

This is an easy fork-join computation: traverse the tree built in step 1 and produce no result (the leaves assign to **output**)

- Invariant: **fromLeft** is sum of elements left of the node's range

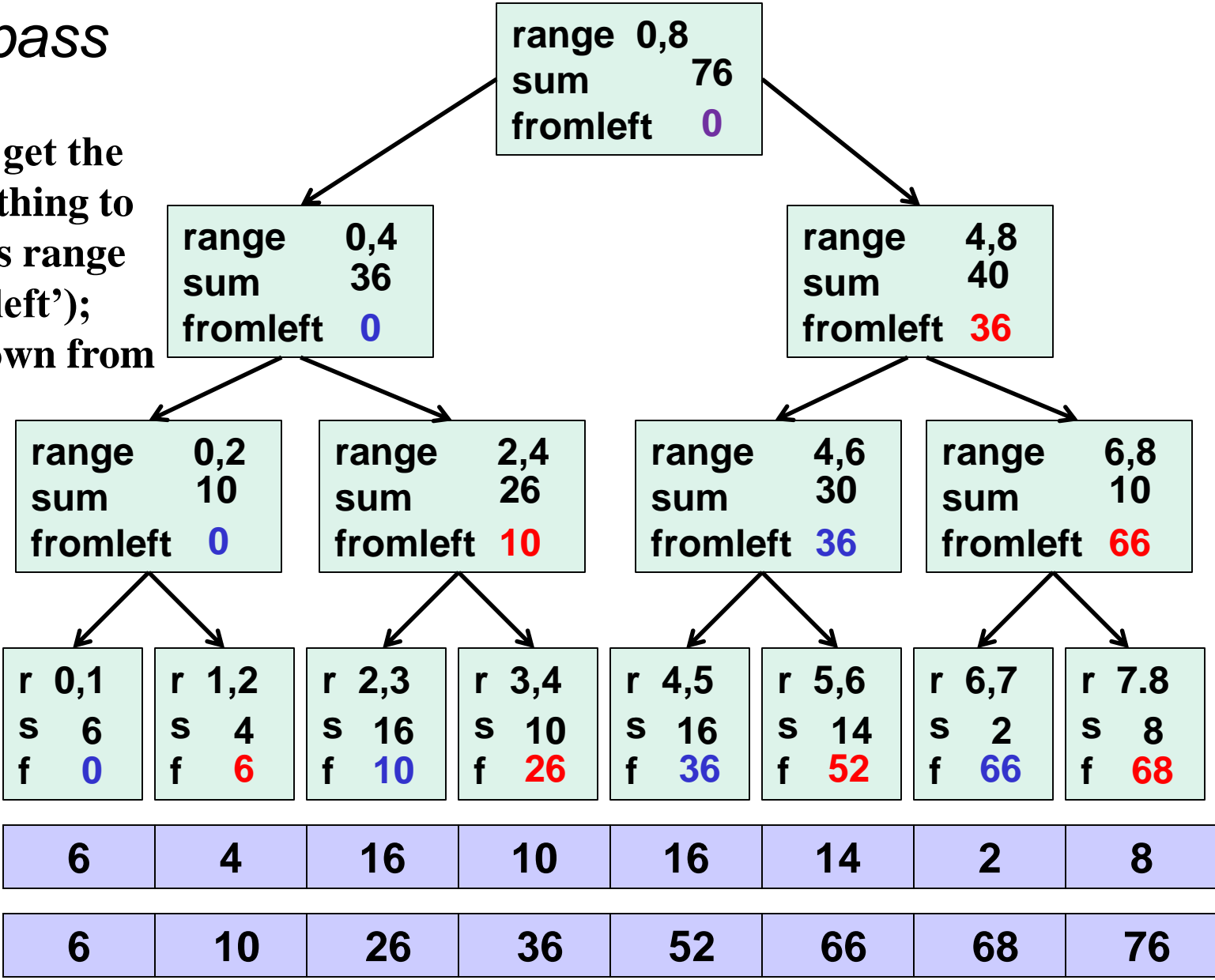
**Analysis of first step:**  $O(n)$  work,  $O(\log n)$  span

**Analysis of second step:**  $O(n)$  work,  $O(\log n)$  span

**Total for algorithm:**  $O(n)$  work,  $O(\log n)$  span

# Second pass

Using 'sum', get the sum of everything to the left of this range (call it 'fromleft'); propagate down from root



# *Sequential cut-off*

Adding a sequential cut-off isn't too bad:

- **Step One:** Propagating Up the **sums**:
  - Have a leaf node just hold the sum of a range of values instead of just one array value (Sequentially compute sum for that range)
  - The tree itself will be shallower
- **Step Two:** Propagating Down the **fromLefts**:
  - Have leaf compute prefix sum sequentially over its [lo,hi):  
`output[lo] = fromLeft + input[lo];`  
`for (i=lo+1; i < hi; i++)`  
`output[i] = output[i-1] + input[i]`

# *Parallel prefix, generalized*

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that arises in many, many problems

- Minimum, maximum of all elements **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
  - This last one is perfect for an efficient parallel pack...
  - Perfect for building on top of the “parallel prefix trick”

# *Pack (think “Filter”)*

[Non-standard terminology]

Given an array **input**, produce an array **output** containing only elements such that **f(element)** is **true**

Example: **input** [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

**f**: “is element > 10”

**output** [17, 11, 13, 19, 24]

Parallelizable?

- Determining whether an element belongs in the output is easy
- But determining where an element belongs in the output is hard; seems to depend on previous results....

In this example,  
Filter =  
element > 10

# *Parallel Pack = (Soln)*

## *parallel map + parallel prefix + parallel map*

1. **Parallel map** to compute a **bit-vector** for true elements:

input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

bits [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

2. **Parallel-prefix sum** *on the bit-vector*:

bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

3. **Parallel map** to produce the output:

output [17, 11, 13, 19, 24]

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){

}
```



# *Pack comments*

- First two steps can be combined into one pass
  - Just using a different base case for the prefix sum
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity
- Analysis:  $O(n)$  work,  $O(\log n)$  span
  - 2 or 3 passes, but 3 is a constant 😊
- Parallelized packs will help us parallelize quicksort...

# Sequential Quicksort review

Recall quicksort was sequential, in-place, expected time  $O(n \log n)$

- |  | <b>Best / expected case work</b> |
|--|----------------------------------|
| 1. Pick a pivot element                | <b><math>O(1)</math></b>         |
| 2. Partition all the data into:        | <b><math>O(n)</math></b>         |
| A. The elements less than the pivot    |                                  |
| B. The pivot                           |                                  |
| C. The elements greater than the pivot |                                  |
| 3. Recursively sort A and C            | <b><math>2T(n/2)</math></b>      |

Recurrence (assuming a good pivot):

$$T(0)=T(1)=1$$

$$T(n)=\underline{\hspace{10em}}$$

Run-time:  $O(n \log n)$

How should we parallelize this?

# Review: Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

$T(n) = O(1) + T(n-1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n-1)$	exponential
$T(n) = O(n) + T(n-1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n \log n)$

Note big-Oh can also use more than one variable

- Example: can sum all elements of an  $n$ -by- $m$  matrix in  $O(nm)$

# *Parallel Quicksort (version 1)*

	<b>Best / expected case <i>work</i></b>
<b>1. Pick a pivot element</b>	<b><math>O(1)</math></b>
<b>2. Partition all the data into:</b>	<b><math>O(n)</math></b>
<b>A. The elements less than the pivot</b>	
<b>B. The pivot</b>	
<b>C. The elements greater than the pivot</b>	
<b>3. Recursively sort A and C</b>	<b><math>2T(n/2)</math></b>

First: Do the two recursive calls in parallel

- **Work:**
- **Span:** now recurrence takes the form:

**Span:**

# Doing better

- $O(\log n)$  speed-up with an infinite number of processors is okay, but a bit underwhelming
  - Sort  $10^9$  elements 30 times faster
- Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized
  - The Internet has been known to be wrong 😊
  - But we need auxiliary storage (no longer in place)
  - In practice, constant factors may make it not worth it, but remember Amdahl's Law...(exposing parallelism is important!)
- Already have everything we need to parallelize the partition...

# *Parallel partition (not in place)*

**Partition all the data into:**

- A. The elements less than the pivot**
- B. The pivot**
- C. The elements greater than the pivot**

- This is just two packs!
  - We know a pack is  $O(n)$  work,  $O(\log n)$  span
  - Pack elements less than pivot into left side of **aux** array
  - Pack elements greater than pivot into right side of **aux** array
  - Put pivot between them and recursively sort
  - With a little more cleverness, can do both packs at once but no effect on asymptotic complexity
- With \_\_\_\_\_ span for partition, the total span for quicksort is  
 $T(n) =$

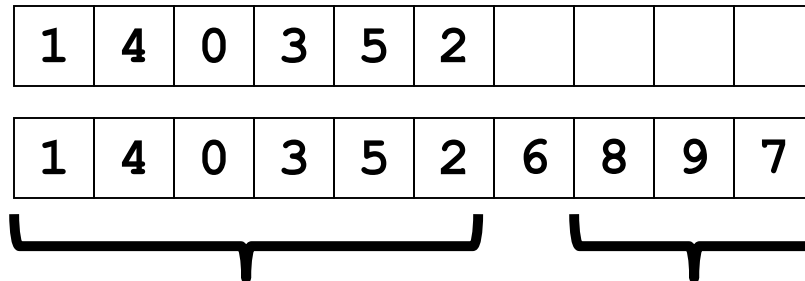
# Parallel Quicksort Example (version 2)

- Step 1: pick pivot as median of three

8	1	4	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

- Steps 2a and 2c (combinable): pack less than, then pack greater than into a second array
  - Fancy parallel prefix to pull this off (not shown)

1	4	0	3	5	2				
1	4	0	3	5	2	6	8	9	7



- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (like in mergesort)

# Parallelize Mergesort?

Recall mergesort: sequential, **not**-in-place, worst-case  $O(n \log n)$

- |                                  |           |
|----------------------------------|-----------|
| 1. Sort left half and right half | $2T(n/2)$ |
| 2. Merge results                 | $O(n)$    |

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the **Span** to  $T(n) = O(n) + 1T(n/2) = O(n)$

- Again, **Work** is  $O(n \log n)$ , and
- parallelism is  $\text{work/span} = O(\log n)$
- To do better, *need to parallelize the merge*
  - The trick won't use parallel prefix this time...



# Parallelizing the merge

Need to merge two *sorted* subarrays (may not have the same size)

0	1	4	8	9
---	---	---	---	---

2	3	5	6	7
---	---	---	---	---

**Idea:** Suppose the larger subarray has  $m$  elements. In parallel:

- Merge the first  $m/2$  elements of the larger half with the “appropriate” elements of the smaller half
- Merge the second  $m/2$  elements of the larger half with the rest of the smaller half

# *Parallelizing the merge (in more detail)*

Need to merge two **sorted** subarrays (may not have the same size)

**Idea:** Recursively divide subarrays in half, merge halves in parallel

0	4	6	8	9
---	---	---	---	---

1	2	3	5	7
---	---	---	---	---

Suppose the larger subarray has  $m$  elements. In parallel:

- Pick the **median** element of the larger array (here 6) in constant time
- In the other array, use binary search to find the first element greater than or equal to that median (here 7)

Then, in parallel:

- Merge half the larger array (from the median onward) with the upper part of the shorter array
- Merge the lower part of the larger array with the lower part of the shorter array

# *Example: Parallelizing the Merge*

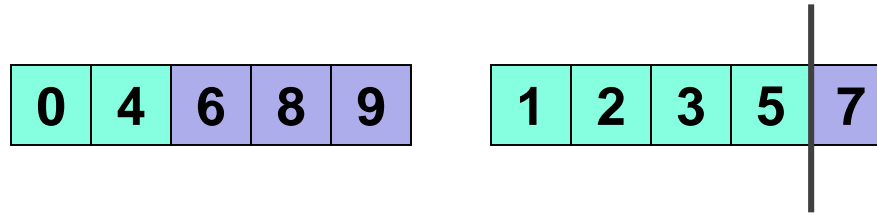
<b>0</b>	<b>4</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>5</b>	<b>7</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

# *Example: Parallelizing the Merge*



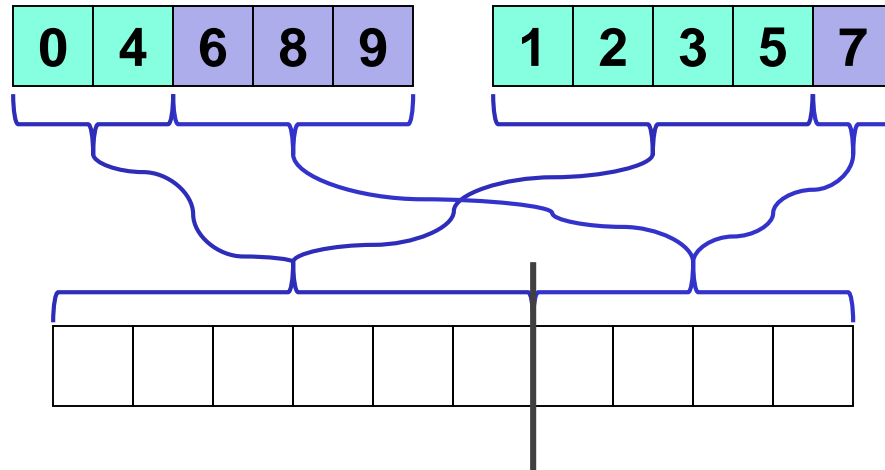
1. Get median of bigger half:  $O(1)$  to compute middle index

# Example: Parallelizing the Merge



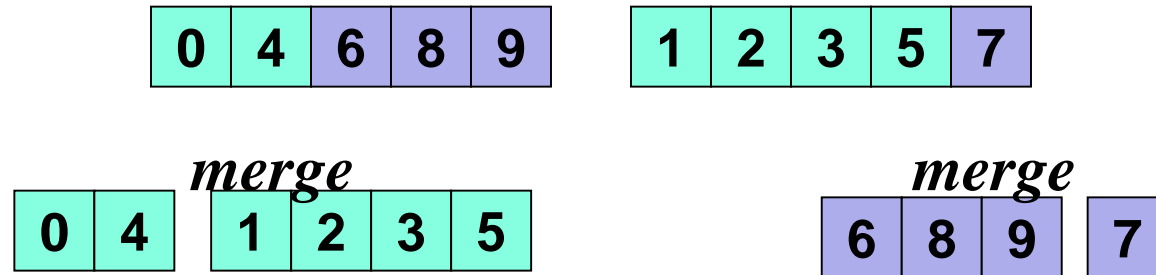
1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  $O(\log n)$  to do binary search on the sorted small half

# Example: Parallelizing the Merge



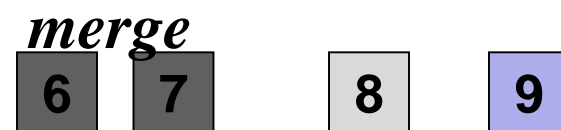
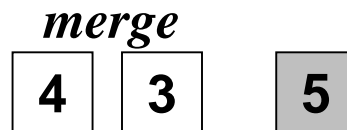
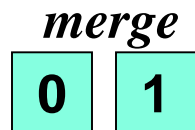
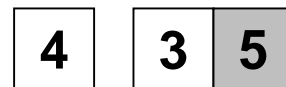
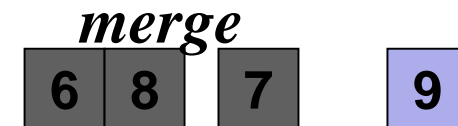
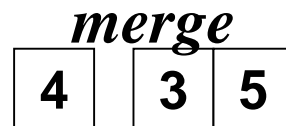
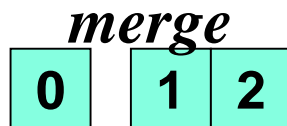
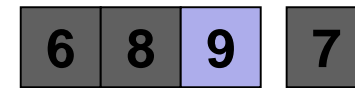
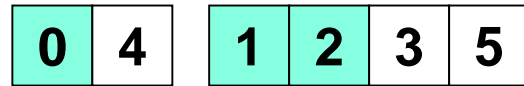
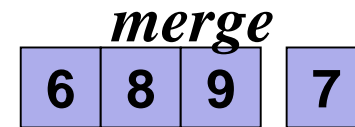
1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  $O(\log n)$  to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array:  $O(1)$

# Example: Parallelizing the Merge



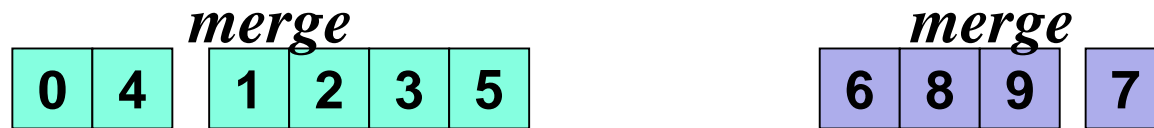
1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  $O(\log n)$  to do binary search on the sorted small half
3. Two sub-merges conceptually splits output array:  $O(1)$
4. Do two submerges in parallel

# Example: Parallelizing the Merge



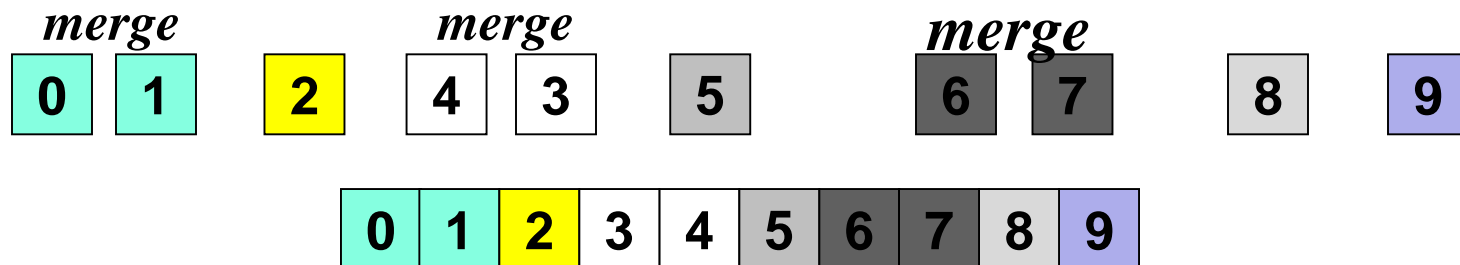


# Example: Parallelizing the Merge



When we do each merge in parallel:

- we split the bigger array in half
- use binary search to split the smaller array
- And in base case we do the copy



# *Parallel Merge Pseudocode*

```
Merge(arr[], left1, left2, right1, right2, out[], out1, out2 )  
    int leftSize = left2 – left1  
    int rightSize = right2 – right1  
    // Assert: out2 – out1 = leftSize + rightSize  
    // We will assume leftSize > rightSize without loss of generality  
  
    if (leftSize + rightSize < CUTOFF)  
        sequential merge and copy into out[out1..out2]  
  
    int mid = (left2 – left1)/2  
    binarySearch arr[right1..right2] to find j such that  
        arr[j] ≤ arr[mid] ≤ arr[j+1]  
  
    Merge(arr[], left1, mid, right1, j, out[], out1, out1+mid+j)  
    Merge(arr[], mid+1, left2, j+1, right2, out[], out1+mid+j+1, out2)
```

# Analysis

- Sequential mergesort:

$$T(n) = 2T(n/2) + O(n) \quad \text{which is } O(n \log n)$$

- Doing the *two recursive calls in parallel* but a sequential merge:

**Work:** same as sequential

**Span:**  $T(n) = 1T(n/2) + O(n)$  which is  $O(n)$

- Parallel merge makes **work** and **span** harder to compute...

- Each merge step does an extra  $O(\log n)$  binary search to find how to split the smaller subarray
- To merge  $n$  elements total, do two smaller merges of possibly different sizes
- But worst-case split is  $(3/4)n$  and  $(1/4)n$ 
  - Happens when the two subarrays are of the same size ( $n/2$ ) and the “smaller” subarray splits into two pieces of the most uneven sizes possible: one of size  $n/2$ , one of size 0

“larger”

0	4	6	8
---	---	---	---

“smaller”

1	2	3	5
---	---	---	---

# *Analysis continued*

For **just** a parallel merge of  $n$  elements:

- **Work** is  $T(n) = T(3n/4) + T(n/4) + O(\log n)$  which is  $O(n)$
- **Span** is  $T(n) = T(3n/4) + O(\log n)$ , which is  $O(\log^2 n)$
- (neither bound is immediately obvious, but “trust me”)

So for **mergesort** with parallel merge overall:

- **Work** is  $T(n) = 2T(n/2) + O(n)$ , which is  $O(n \log n)$
- **Span** is  $T(n) = 1T(n/2) + O(\log^2 n)$ , which is  $O(\log^3 n)$

So parallelism (work / span) is  $O(n / \log^2 n)$

- Not quite as good as quicksort’s  $O(n / \log n)$ 
  - But (unlike Quicksort) this is a worst-case guarantee
- And as always this is just the asymptotic result