CSE 332: Data Structures & Parallelism
Lecture 15: Analysis of Fork-Join Parallel Programs

✓✓ → $mid = low + \frac{hi-lo}{2}$

Arthur Liu

Summer 2022

The Parallelism Part of this class

- Introduction of Parallelism Ideas
  - Java's Thread
  - ForkJoin Library
- General Parallelism Algorithms
  - Reduce, Map
  - Analysis (span, work)
- Clever Parallelism Ideas
  - Parallel Prefix
  - Parallel Sorts
- Synchronization
  - The need for locks (Concurrency)
- Other Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings

The prefix-sum problem

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

`output[i]` = **`input[0]+input[1]+…+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;
}
```

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?

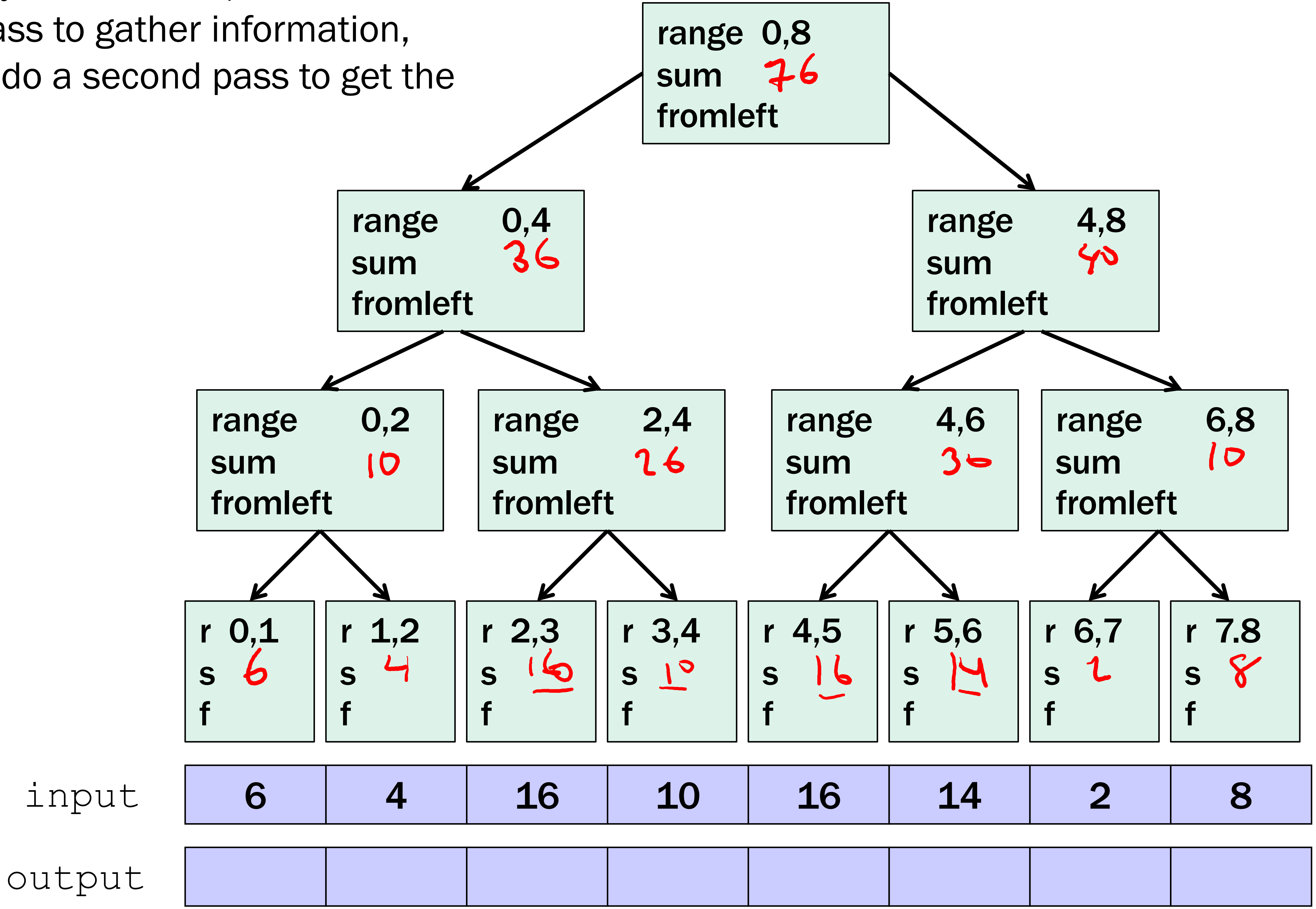
recent

The algorithm, part 1

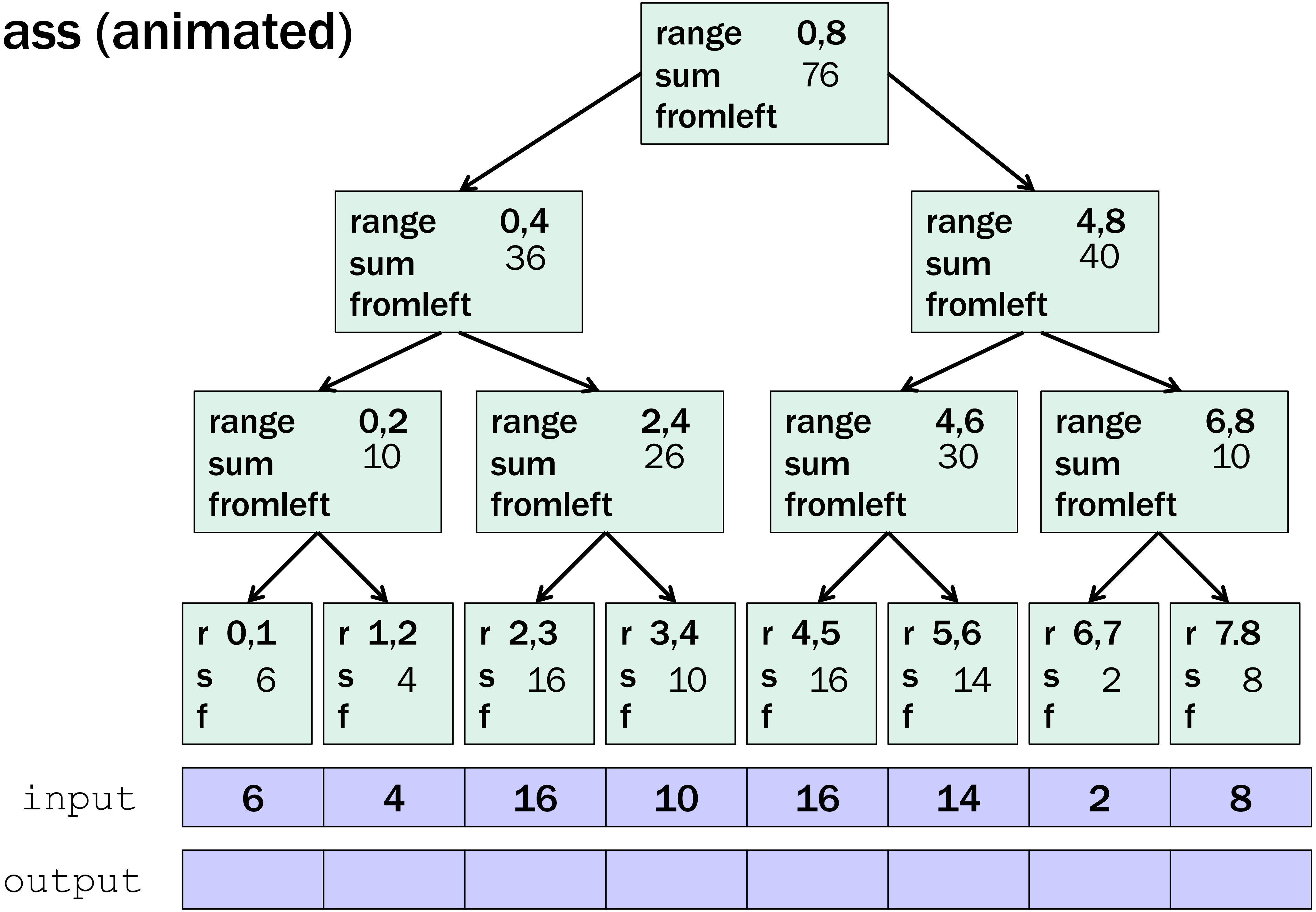
1. Propagate ‘sum’ up: Build a binary tree where
    - Root has sum of **`input[0]..input[n-1]`**
    - Each node has sum of **`input[lo]..input[hi-1]`**
        - Build up from leaves; parent.sum=left.sum+right.sum
    - A leaf’s sum is just it’s value; **`input[i]`**

This is an easy fork-join computation: same as sum algorithm of array but this time store answers in tree as we move up

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



First pass (animated)



The algorithm, part 2

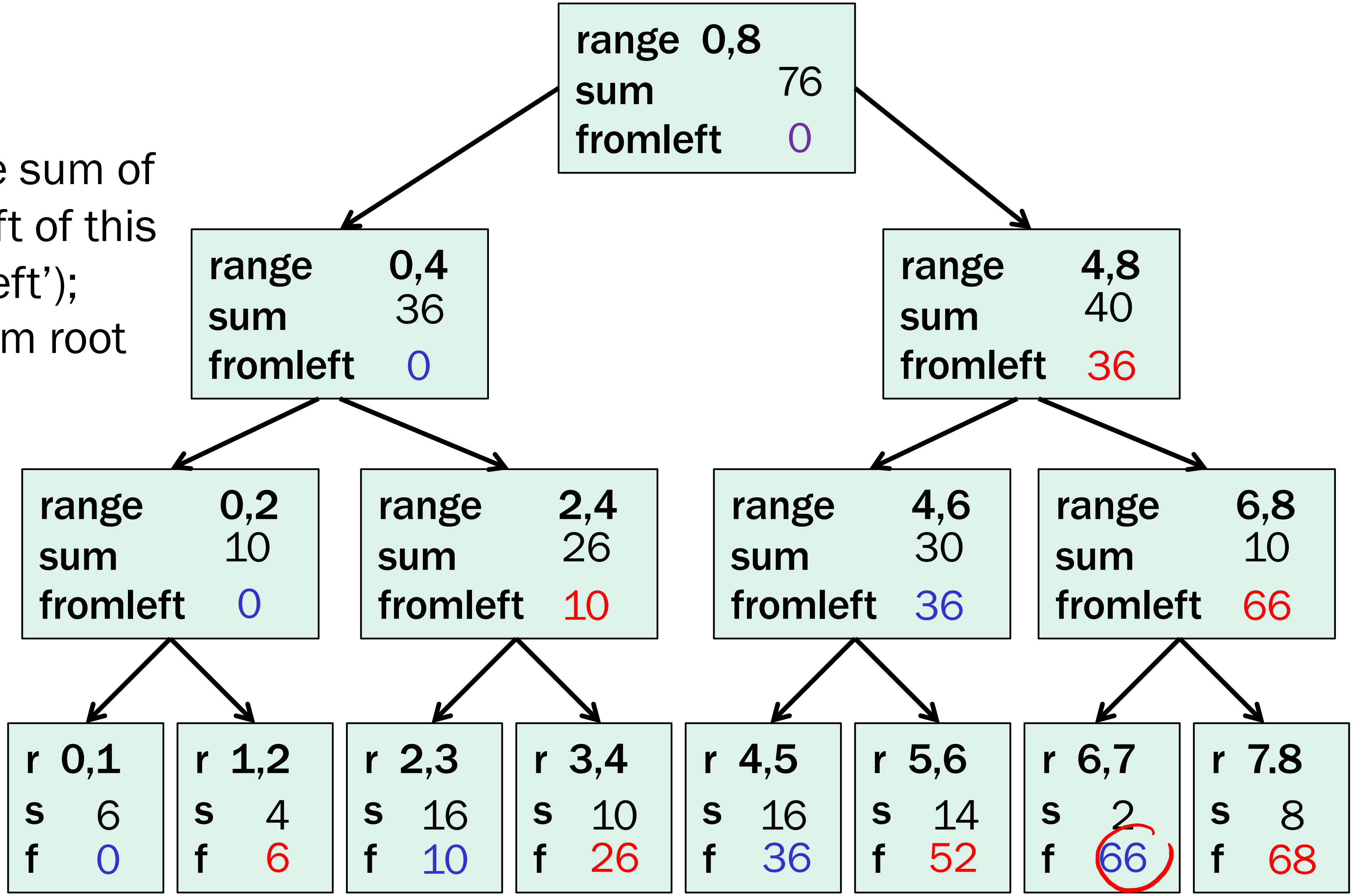
2. Propagate ‘fromleft’ down:
   - Root given a **`fromLeft`** of **`0`**
   - Node takes its **`fromLeft`** value and
     - Passes its left child the same **`fromLeft`**
     - Passes its right child its **`fromLeft`** plus its left child’s **`sum`** (as stored in part 1)
   - At the leaf for array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

This is also an easy fork-join computation: traverse the tree built in step 1 and fill in the fromLeft field using saved information

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

Second pass

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



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

Analysis of Algorithm

Original boring 142 algorithm: *O(n)*

Analysis of our fancy prefix sum algorithm:

Analysis of first step: span

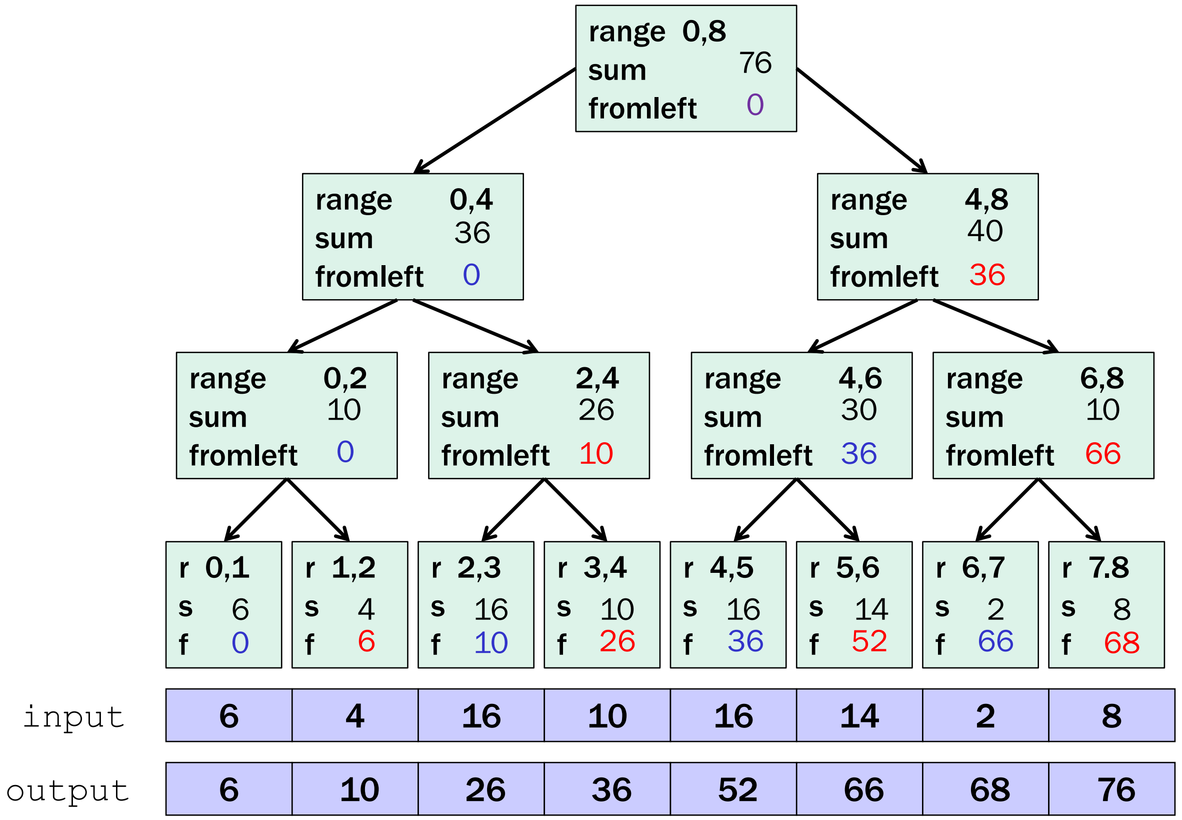
O(n) work O(log n)

Analysis of second step:

O(n) work O(log n) span

Total for algorithm:

O(n) work O(log n) span



Analysis of Algorithm

Original boring 142 algorithm: $O(n)$

Analysis of our fancy prefix sum algorithm:

Analysis of first step:

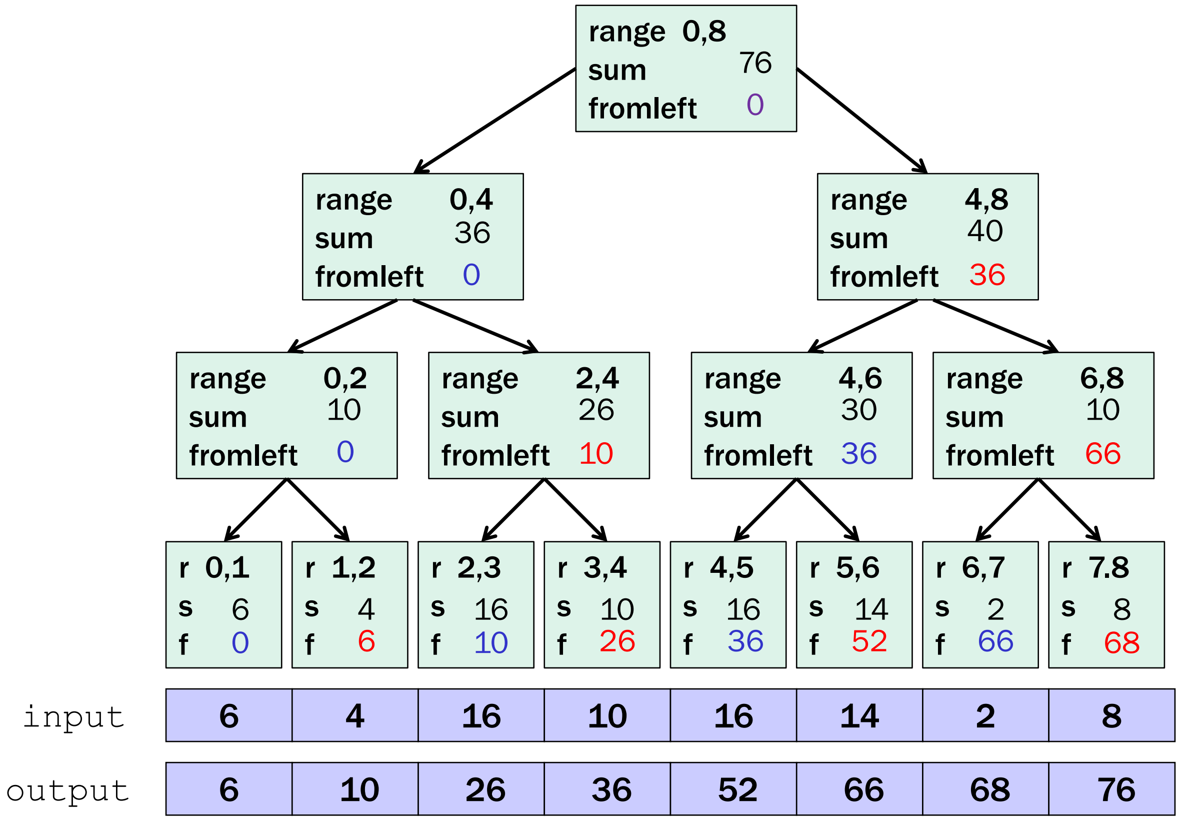
$O(n)$ **work,** $O(\texttt{log}\ n)$ **span**

Analysis of second step:

$O(n)$ **work,** $O(\texttt{log}\ n)$ **span**

Total for algorithm:

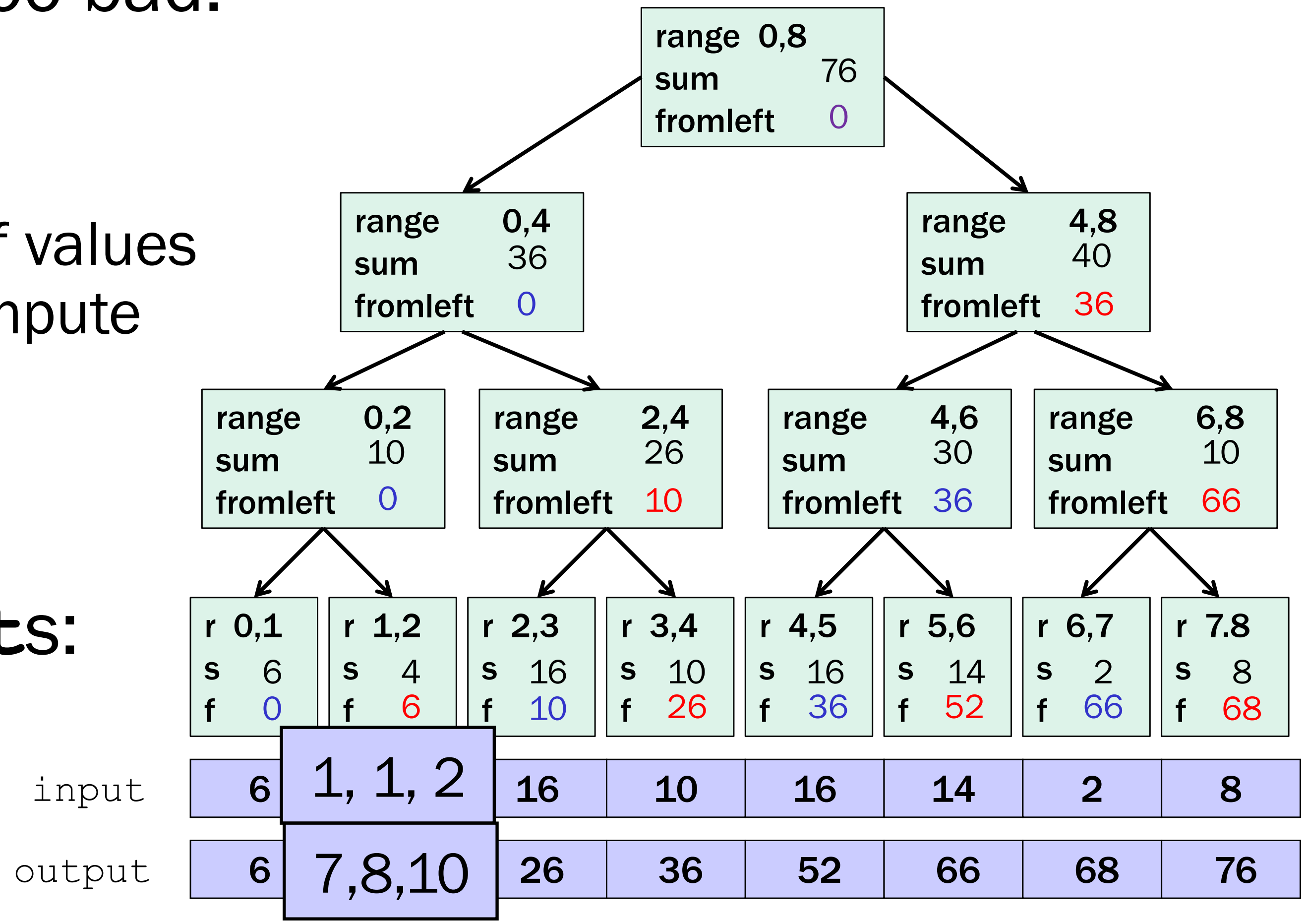
$O(n)$ **work,** $O(\texttt{log}\ n)$ **span**



Sequential cut-off

Optimizing: 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`:
  - At leaf, compute prefix sum over its [lo,hi):



On the topic of optimization, do we need to actually have a tree?

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")

Given an array **`input`**, produce an array **`output`** containing <u>only</u> 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 *<u>whether</u>* an element belongs in the output is easy
- But determining *<u>where</u>* an element belongs in the output is hard; seems to depend on previous results....

Solution! Parallel Pack = parallel map + parallel prefix + parallel map

In this example,
Filter =
element > 10

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++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[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…

The Parallelism Part of this class

- Introduction of Parallelism Ideas
  - Java’s Thread
  - ForkJoin Library
- General Parallelism Algorithms
  - Reduce, Map
  - Analysis (span, work)
- Clever Parallelism Ideas
  - Parallel Prefix
  - Parallel Sorts (Next)
- Synchronization
  - The need for locks (Concurrency)
- Other Synchronization Issues
  - Race Conditions: Data Races & Bad Interleavings

Quick Quick Sort Analysis Note

- For all of our quick sort analysis, we'll do best case.
- The average case is the same as best case.

- Worst case is still going to be the same (bad) $\Theta(n^2)$ with parallelism or not.

Sequential Quicksort review

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

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

Recurrence (assuming a good pivot):

$$T(n) = \begin{cases} O(1) \\ 2T(n/2) + O(n) \end{cases}$$

Run-time:

$O(n \log n)$

Parallel **Quicksort VERSION 1**

Best / expected case *work*

1. Pick a pivot element	O(1)
2. Partition all the data into:	O(n)
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	2T(n/2)

Idea: Do the two recursive calls in parallel

Work: $T_1(n) = 2T_1\left(\frac{n}{2}\right) + O(n)$

Span: $T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + O(n)$

Parallel Quicksort VERSION 1

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

Idea: Do the two recursive calls in parallel

Work:

$$T_1(n) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + O(n) & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases} = O(n \log n)$$

Span:

$$T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + c_1 \cdot n & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases} = O(n)$$

Parallel Quick Sort

With infinitely many processors, we can speed up quicksort from

$O(n \log n)$ to…

$O(n)$.

So…yeah….

We can do better!

In exchange for using auxiliary arrays (i.e. a not in-place sort).

Probably not better today. But maybe eventually…

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, $O(\texttt{log}\ n)$ span
  - Pack elements less than pivot into left side of **`aux`** array
  - Pack elements greater than pivot into right size 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

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 (NOTE: no longer in-place!)
  - 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

Parallel Quicksort VERSION 2

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

Idea: Do the partition with some parallel prefix packing

Work:

Span:

$$T_\infty = \begin{cases} O(1) \\ T_\infty\left(\frac{n}{2}\right) + O(\log n) \end{cases}$$

Parallel Quicksort VERSION 2

Best / expected case *work*

1. Pick a pivot element — O(1)
2. Partition all the data into: — O(n)
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C — 2T(n/2)

Idea: Do the partition with some parallel prefix packing

Work: same but worse constants

$$\mathrm{T}_1(\mathrm{n}) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + O(n) & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases} = O(n \log n)$$

Span:

$$T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + O(\log n) & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases}$$

Closed form: $T_\infty(n) = O\left(\log^2(n)\right)$

Parallelize Mergesort?

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

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 $T(n) = \boldsymbol{O(n) + 1T(n/2) = O(n)}$

Again, Work is O(nlogn), and

parallelism is 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 (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)

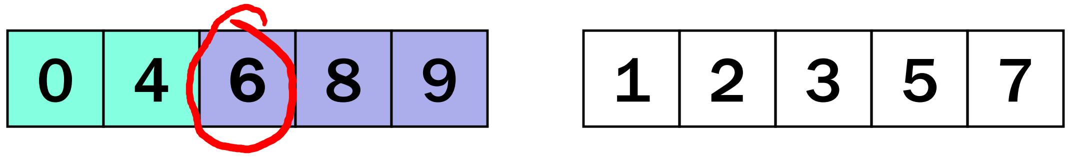
Then, in parallel:

- 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

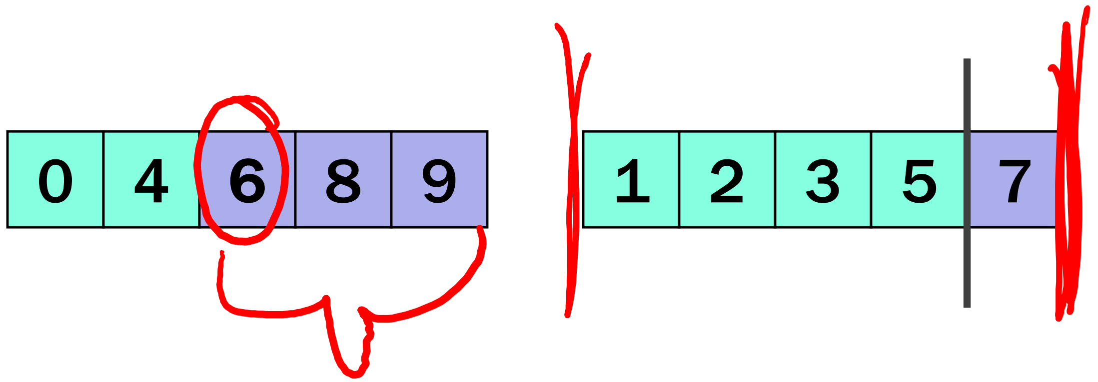
0	4	6	8	9		1	2	3	5	7

Example: Parallelizing the Merge



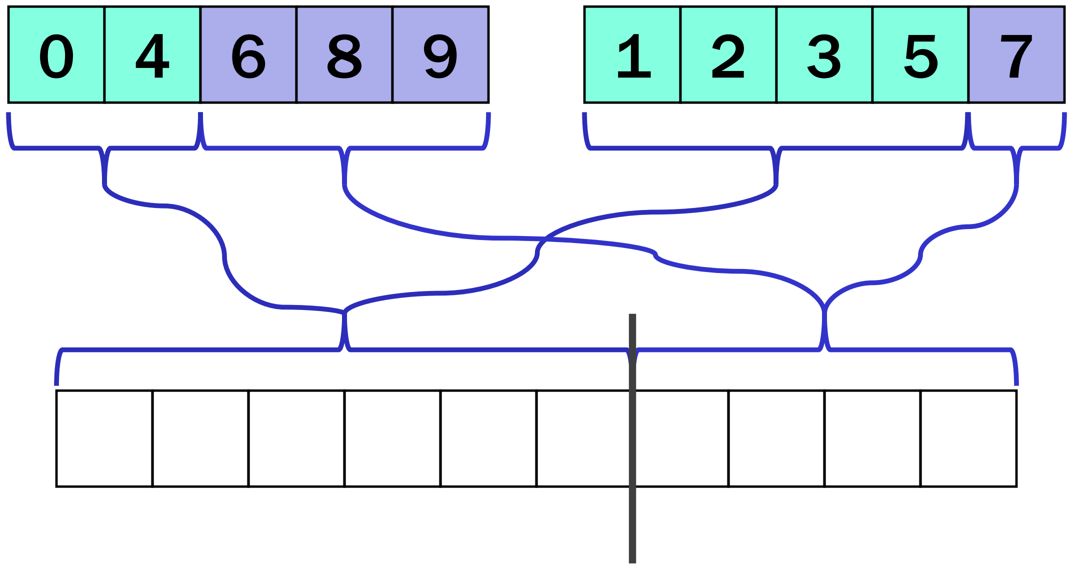
1. Get median of bigger half: $O(1)$ to compute middle index

Example: Parallelizing the Merge



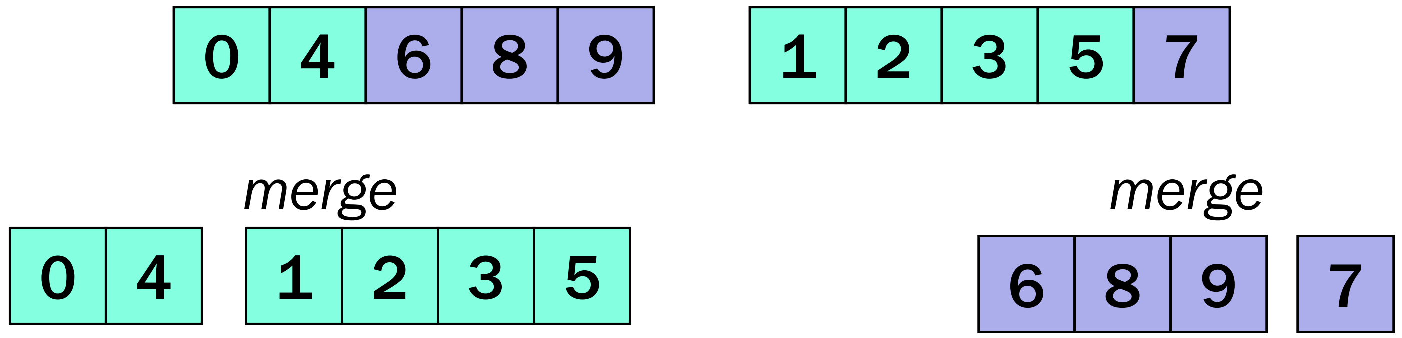
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value: O(**`log`** n) to do binary search on the sorted small half

Example: Parallelizing the Merge



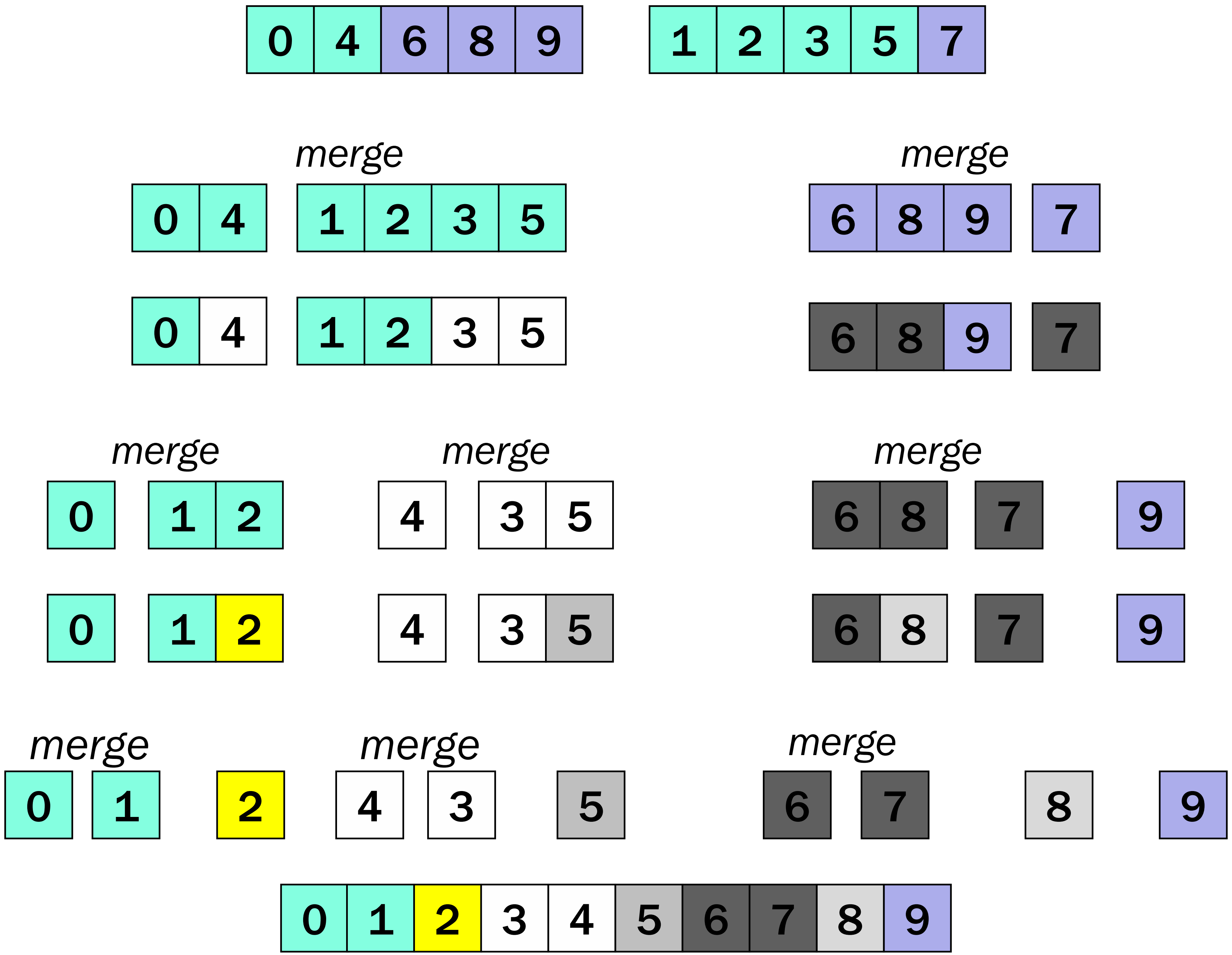
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value:
   O(`log` n) to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$

Example: Parallelizing the Merge

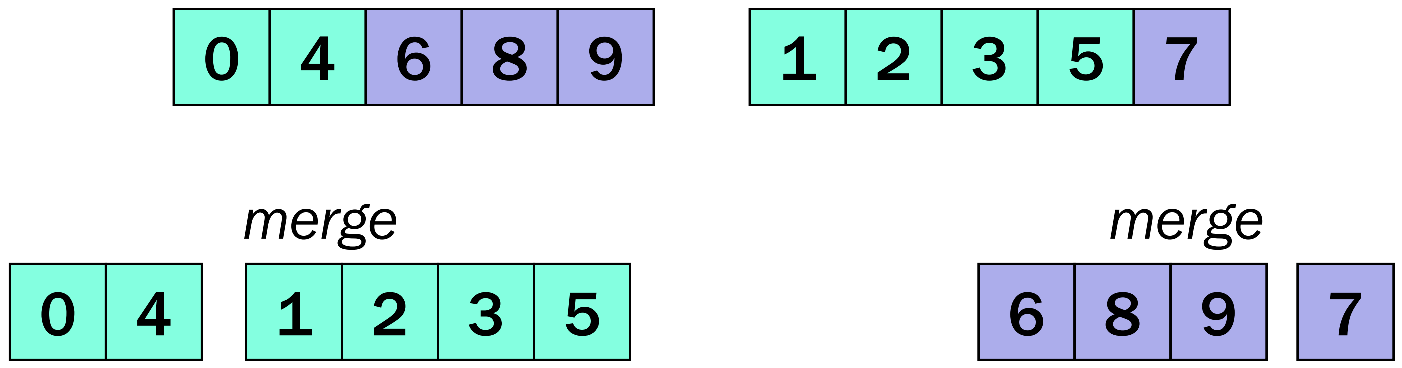


1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value: O(**`log`** n) to do binary search on the sorted small half
3. Two sub-merges conceptually splits output array: $O(1)$
4. Do two submerges in parallel

Example: Parallelizing the Merge

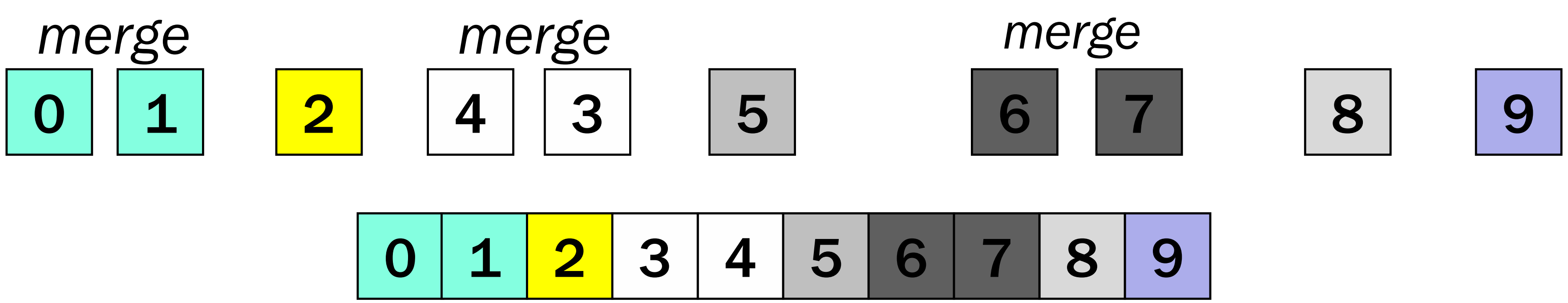


Example: Parallelizing the Merge



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 Sort

Let's just analyze the ***merge***:

What's the worst case?

One subarray has ¾ of the elements, the other has ¼ .

This is why we start with the median of the larger array.

Work: $\mathrm{T}_1(\mathrm{n}) =$

Span: $T_\infty(n) =$

Parallel Merge Sort

Let's just analyze the merge:

What's the worst case?

One subarray has ¾ of the elements, the other has ¼ .

This is why we start with the median of the larger array.

$$\text{Work: } T_1(n) = \begin{cases} T_1\left(\frac{3n}{4}\right) + T_1\left(\frac{n}{4}\right) + O(\log n) & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases}$$

$$\text{Span: } T_\infty(n) = \begin{cases} T_\infty\left(\frac{3n}{4}\right) + O(\log n) & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases}$$

Parallel Merge Sort

Let's just analyze the merge:

What's the worst case?

One subarray has ¾ of the elements, the other has ¼ .

This is why we start with the median of the larger array.

Work: $T_1(n) = O(n)$

Span: $T_\infty(n) = O(\log^2 n)$

Parallel Merge Sort

- Now the full mergesort algorithm:

$$\text{Work: } T_1(n) = \begin{cases} 2T_1\left(\frac{n}{2}\right) + \boxed{O(n)} & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases}$$

$$\text{Span: } T_\infty(n) = \begin{cases} T_\infty\left(\frac{n}{2}\right) + \boxed{O(\log^2 n)} & \text{if } n \geq \text{cutoff} \\ O(1) & \text{if } n < \text{cutoff} \end{cases}$$

Parallel Merge Sort

- Now the full mergesort algorithm:

- Work: $T_1(n) = O(n \log n)$

- Span: $T_\infty(n) = O(\log^3 n)$

Quicksort

$O(\log^2 n)$

1. $\log n \log n$

2. $\log(\log(n))$