

A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency

Lecture 3 Parallel Prefix, Pack, and Sorting

Steve Wolfman, based on work by Dan Grossman

Learning Goals

- Judge appropriate contexts for and apply the parallel map, parallel reduce, and parallel prefix computation patterns.
- And also... lots of practice using map, reduce, work, span, general asymptotic analysis, tree structures, sorting algorithms, and more!

Outline

Done:

- Simple ways to use parallelism for counting, summing, finding
- (Even though in practice getting speed-up may not be simple)
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- Parallel prefix
- Parallel pack (AKA filter)
- Parallel sorting
 - quicksort (not in place)
 - mergesort

The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where $\text{output}[i] = \text{input}[0] + \text{input}[1] + \dots + \text{input}[i]$

Sequential version is straightforward:

```
Vector<int> prefix_sum(const vector<int>& input) {  
    vector<int> output(input.size());  
    output[0] = input[0];  
    for(int i=1; i < input.size(); i++)  
        output[i] = output[i-1] + input[i];  
    return output;  
}
```

Example:

input	42	3	4	7	1	10
output						

The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where $\text{output}[i] = \text{input}[0] + \text{input}[1] + \dots + \text{input}[i]$

Sequential version is straightforward:

```
Vector<int> prefix_sum(const vector<int>& input) {  
    vector<int> output(input.size());  
    output[0] = input[0];  
    for(int i=1; i < input.size(); i++)  
        output[i] = output[i-1] + input[i];  
    return output;  
}
```

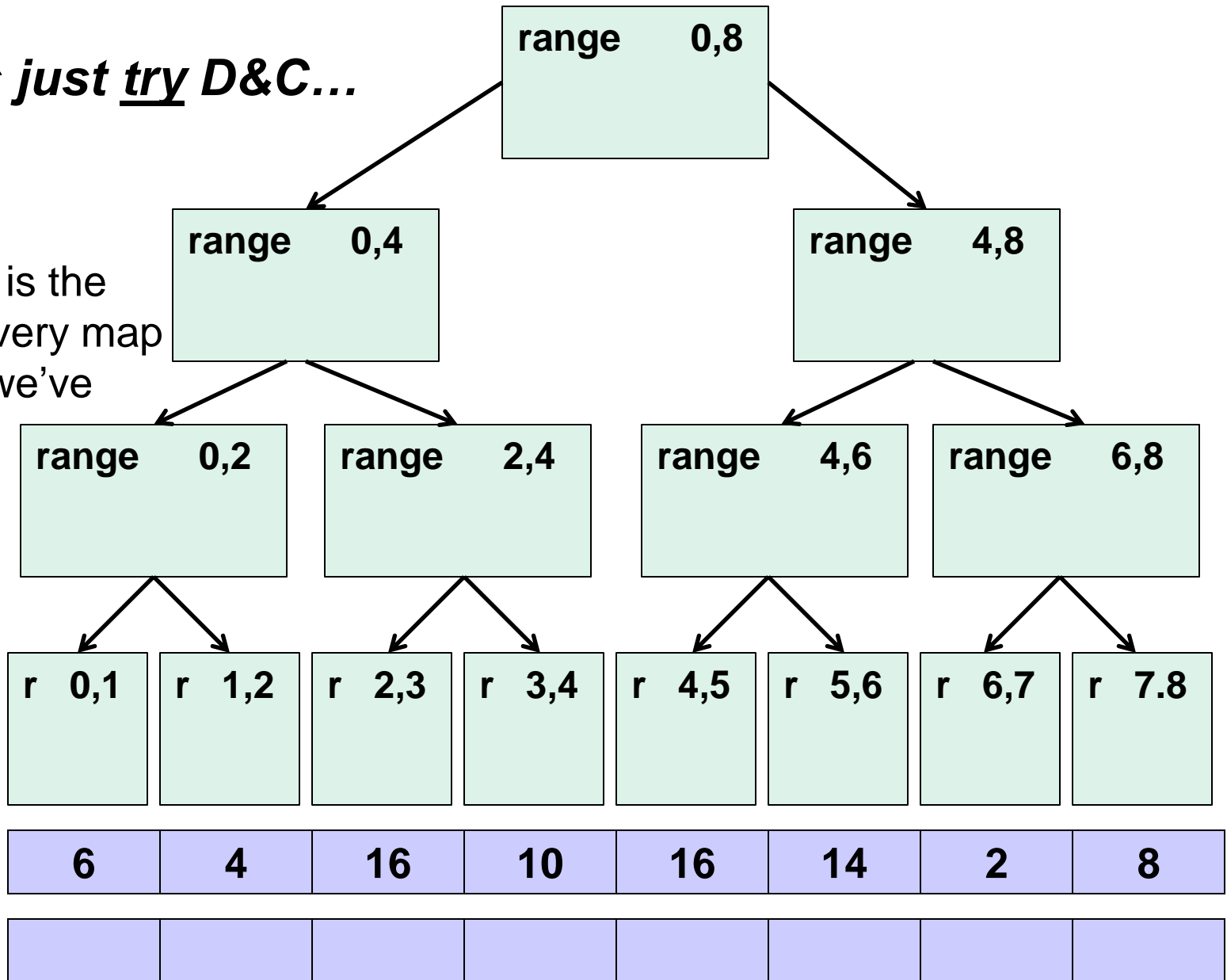
Why isn't this (obviously) parallelizable? Isn't it just map or reduce?

Work:

Span:

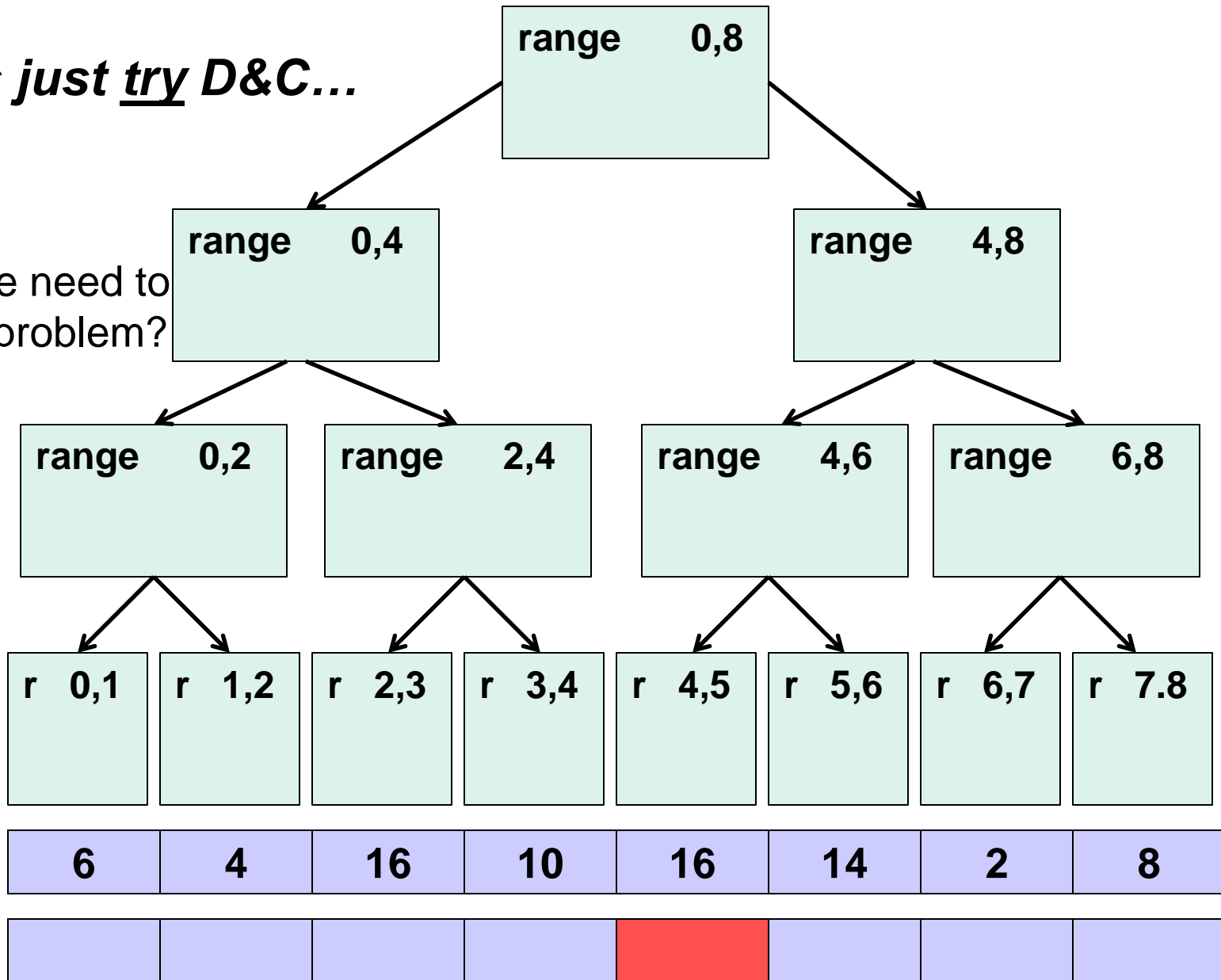
Let's just try D&C...

So far, this is the same as every map or reduce we've done.



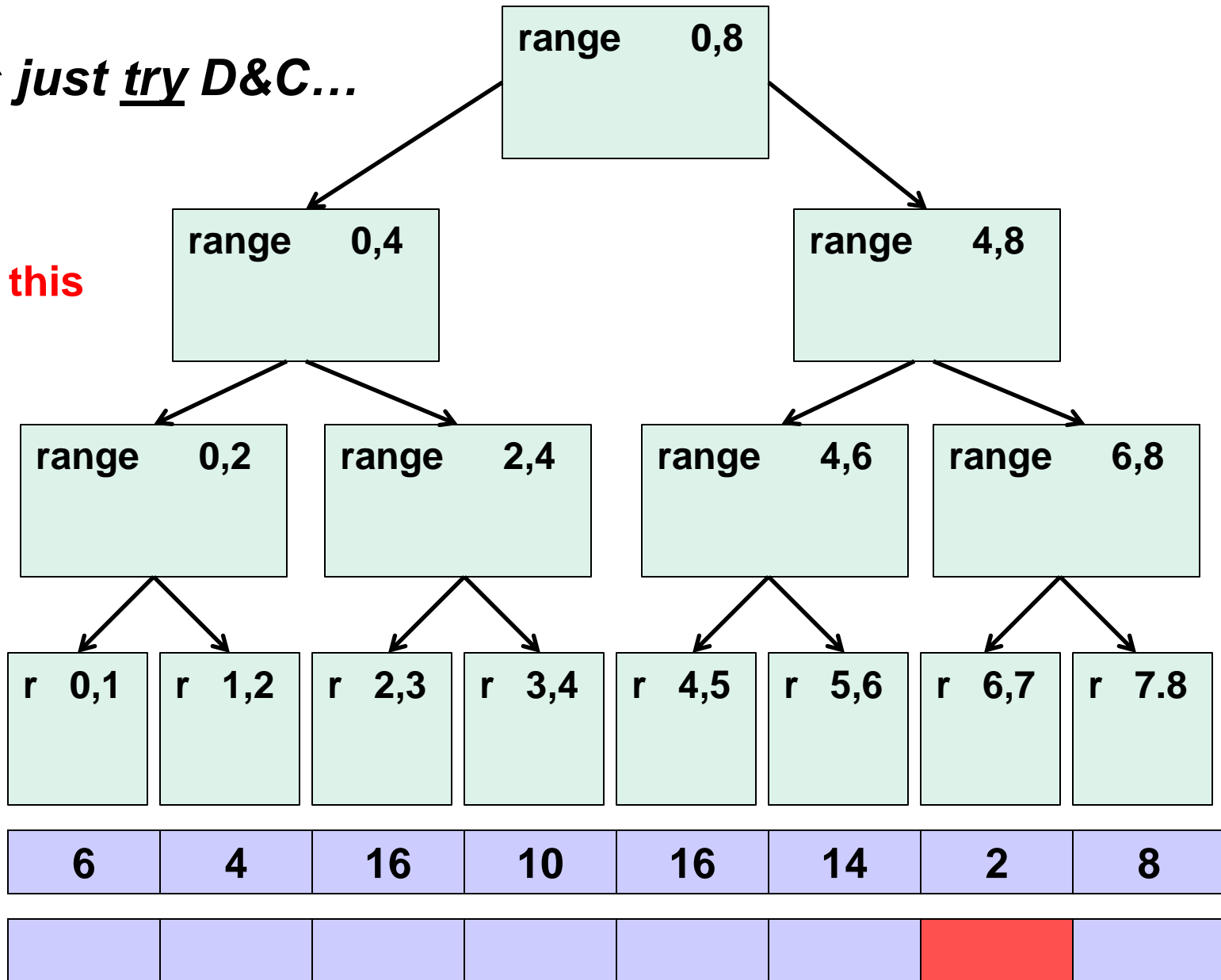
Let's just try D&C...

What do we need to solve **this** problem?



Let's just try D&C...

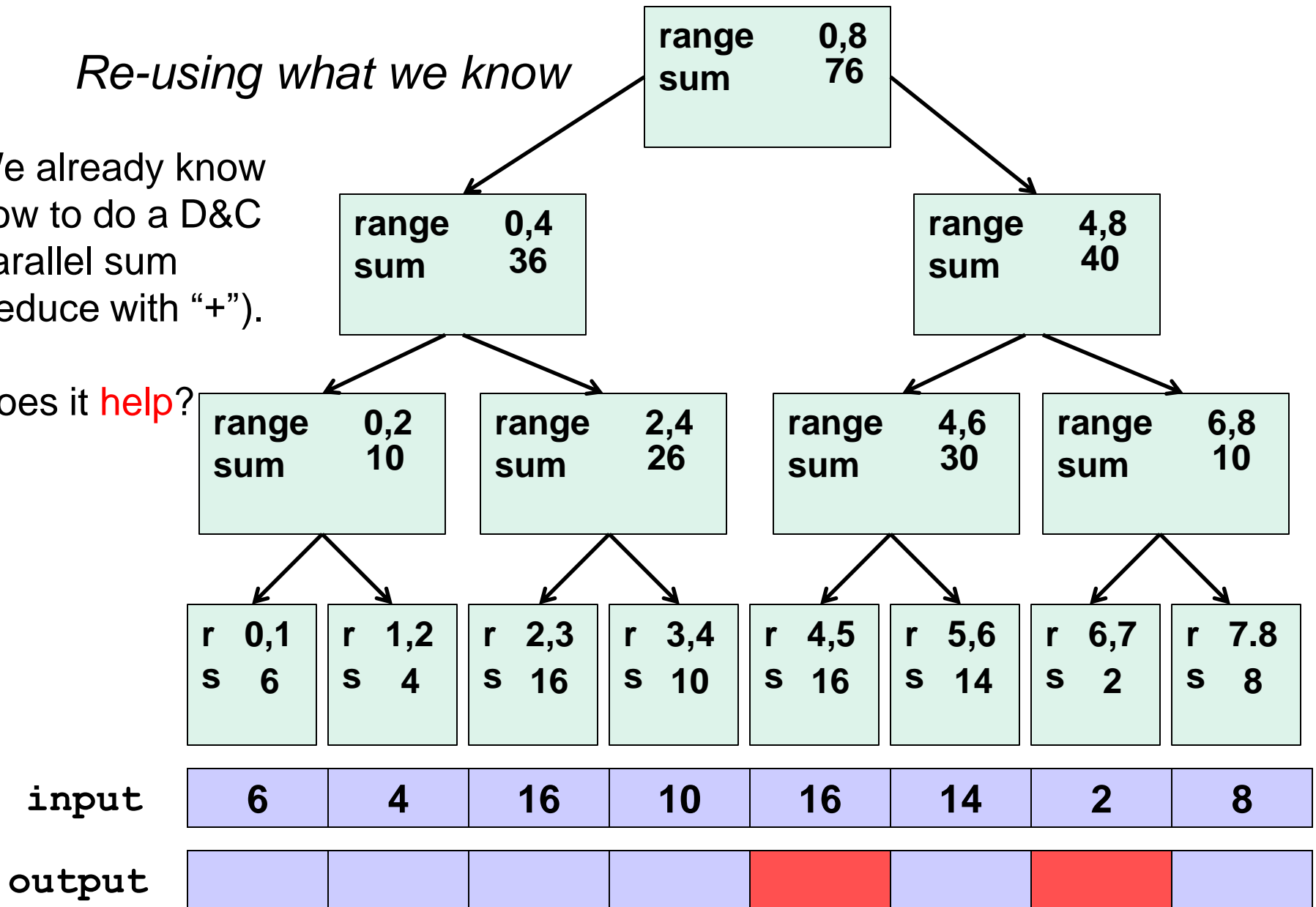
How about **this** problem?



Re-using what we know

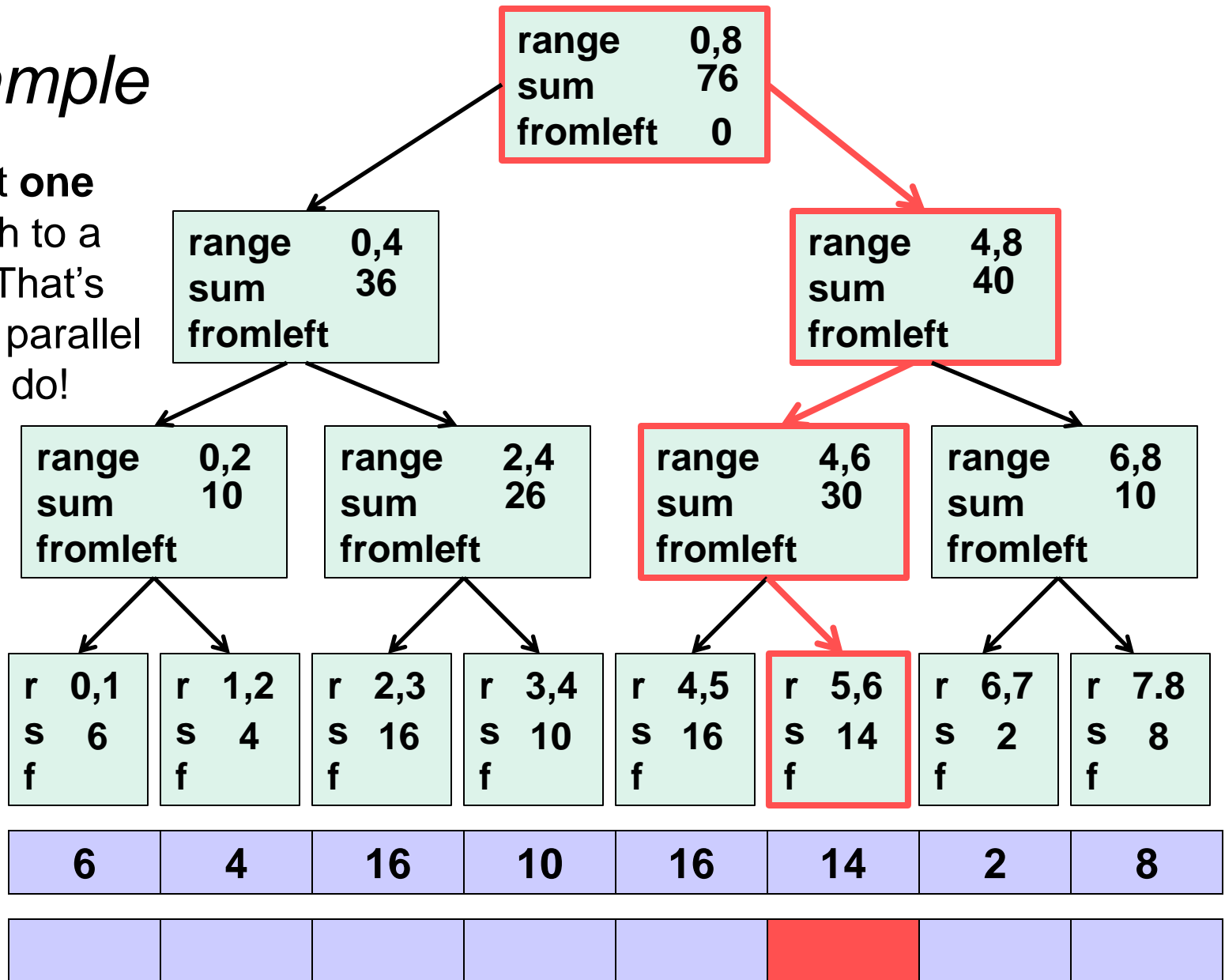
We already know how to do a D&C parallel sum (reduce with “+”).

Does it **help**?



Example

Let's do just **one** branch (path to a leaf) **first**. That's what a fully parallel solution will do!



Parallel prefix-sum

The parallel-prefix algorithm does two passes:

1. build a “sum” tree bottom-up
2. traverse the tree top-down, accumulating the sum from the left

The algorithm, step 1

1. Step one does a parallel sum to build a binary tree:
 - Root has sum of the range $[0, n)$
 - An internal node with the sum of $[lo, hi)$ has
 - Left child with sum of $[lo, middle)$
 - Right child with sum of $[middle, hi)$
 - A leaf has sum of $[i, i+1)$, i.e., `input[i]`

How? Parallel sum but explicitly build a tree:

```
return left+right;   $\Rightarrow$   return new Node(left->sum + right->sum,  
                                           left, right);
```

Step 1:

Work?

Span?

The algorithm, step 2

2. Parallel map, passing down a **fromLeft** parameter
 - Root gets a **fromLeft** of 0
 - Internal node along:
 - to its left child the same **fromLeft**
 - to its right child **fromLeft** plus its left child's **sum**
 - At a leaf node for array position **i**,
output[i] = fromLeft + input[i]

(already calculated
in step 1!)

How? A map down the step 1 tree, leaving results in the output array.

Notice the *invariant*: **fromLeft** is the sum of elements left of the node's range

Step 2:

Work?

Span?

Parallel prefix-sum

The parallel-prefix algorithm does two passes:

1. build a “sum” tree bottom-up
2. traverse the tree top-down, accumulating the sum from the left

Step 1:	Work: $O(n)$	Span: $O(\lg n)$
---------	--------------	------------------

Step 2:	Work: $O(n)$	Span: $O(\lg n)$
---------	--------------	------------------

Overall:	Work?	Span?
----------	-------	-------

Paralellism (work/span)?

Parallel prefix, generalized

Can we use parallel prefix to calculate the minimum of all elements to the left of i ?

In general, what property do we need for the operation we use in a parallel prefix computation?

Outline

Done:

- Simple ways to use parallelism for counting, summing, finding
- (Even though in practice getting speed-up may not be simple)
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- Parallel prefix
- Parallel pack (AKA filter)
- Parallel sorting
 - quicksort (not in place)
 - mergesort

Pack

AKA, `filter` 😊

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

Example: `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
 `f: is elt > 10`
 `output [17, 11, 13, 19, 24]`

Parallelizable? Sure, using a list concatenation reduction.

Efficiently parallelizable on arrays?

Can we just put the output straight into the array *at the right spots*?

Pack as map, reduce, prefix combo??

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

Example: `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
`f: is elt > 10`

Which pieces can we do as maps, reduces, or prefixes?

Parallel prefix to the rescue

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.size(); i++){
    if(bits[i])
        output[bitsum[i]-1] = input[i];
}
```

Pack Analysis

Step 1:	Work?	Span?
(compute bit-vector w/a parallel map)		

Step 2:	Work?	Span?
(compute bit-sum w/a parallel prefix sum)		

Step 3:	Work?	Span?
(emplace output w/a parallel map)		

Algorithm:	Work?	Span?
Parallelism?		

Outline

Done:

- Simple ways to use parallelism for counting, summing, finding
- (Even though in practice getting speed-up may not be simple)
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- Parallel prefix
- Parallel pack (AKA filter)
- **Parallel sorting**
 - quicksort (not in place)
 - mergesort

Parallelizing Quicksort

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

	Best / expected case <i>work</i>
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	$2T(n/2)$

How do we parallelize this?

What span do we get?

$T_{\infty}(n) =$

How good is $O(\lg n)$ Parallelism?

Given an infinite number of processors, $O(\lg n)$ faster.

So... sort 10^9 elements 30 times faster?! That's not much ☹

Can't we do better? What's causing the trouble?

(Would using $O(n)$ space help?)

Parallelizing Quicksort

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

	Best / expected case <i>work</i>
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	$2T(n/2)$

How do we parallelize this?

What span do we get?

$T_{\infty}(n) =$

Analyzing $T_{\infty}(n) = 1g \ n + T_{\infty}(n/2)$

Turns out our techniques from way back at the start of the term will work just fine for this:

$$\begin{array}{ll} T_{\infty}(n) &= 1g \ n + T_{\infty}(n/2) && \text{if } n > 1 \\ &= 1 && \text{otherwise} \end{array}$$


Parallel Quicksort Example

- 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 limit extra space to one array of size n, as in mergesort)

Outline

Done:

- Simple ways to use parallelism for counting, summing, finding
- (Even though in practice getting speed-up may not be simple)
- Analysis of running time and implications of Amdahl's Law

Now: Clever ways to parallelize more than is intuitively possible

- Parallel prefix
- Parallel pack (AKA filter)
- Parallel sorting
 - quicksort (not in place)
 - mergesort

mergesort

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

- | | |
|---|-----------------------------|
| 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 $T(n) = O(n) + 1T(n/2) \in O(n)$

- Again, parallelism is $O(\lg n)$
- 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 n elements. In parallel:

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

Parallelizing the merge

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

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

Parallelizing the merge

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

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

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

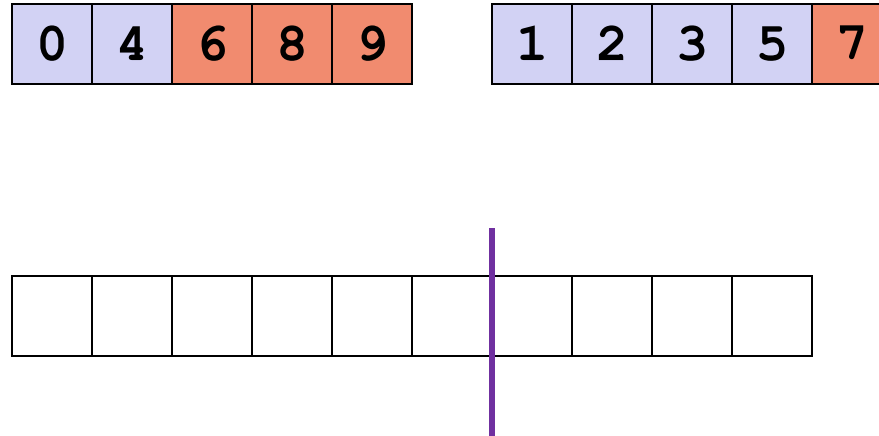
Parallelizing the merge

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

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

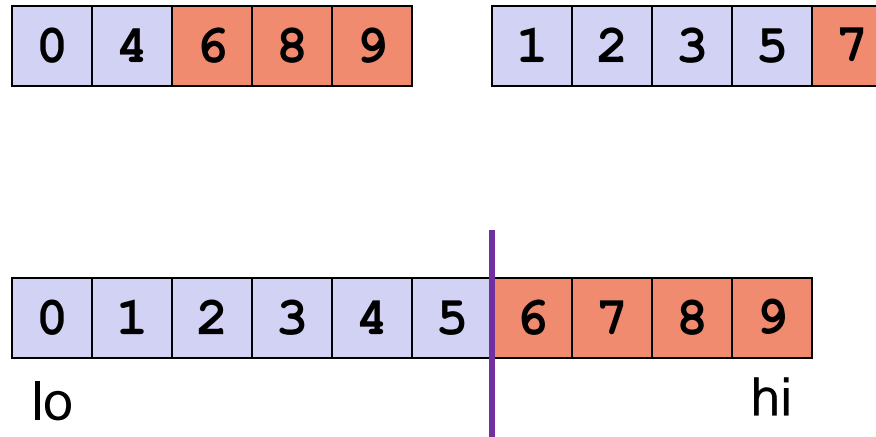
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: $O(\lg n)$ to do binary search on the sorted small half

Parallelizing the merge



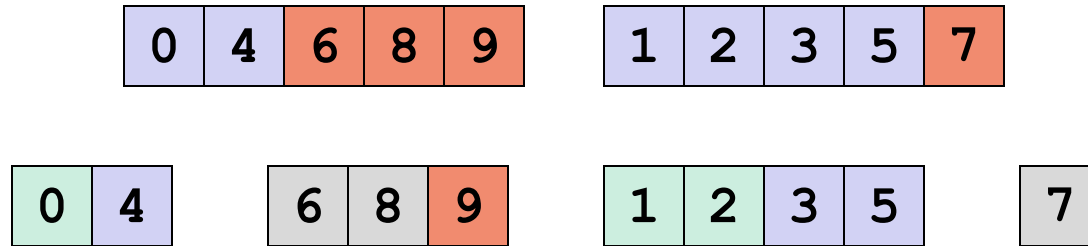
1. Get median of bigger half: $O(1)$ to compute middle index
2. Find how to split the smaller half at the same value as the left-half split: $O(\lg n)$ to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$

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 as the left-half split: $O(\lg n)$ to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array: $O(1)$
4. Do two submerges in parallel

The Recursion



When we do each merge in parallel, we split the bigger one in half and use binary search to split the smaller one

Analysis

- Sequential recurrence for mergesort:

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

- 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 $O(\lg n)$ 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 $(1/4)n$ and $(3/4)n$
 - When subarrays same size and “smaller” splits “all” / “none”

Analysis continued

For just a parallel merge of n elements:

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

So for mergesort with parallel merge overall:

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

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

- Not quite as good as quicksort, but worst-case guarantee
- And as always this is just the asymptotic result

Looking for Answers?

The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where $\text{output}[i] = \text{input}[0] + \text{input}[1] + \dots + \text{input}[i]$

Sequential version is straightforward:

```
Vector<int> prefix_sum(const vector<int>& input) {  
    vector<int> output(input.size());  
    output[0] = input[0];  
    for(int i=1; i < input.size(); i++)  
        output[i] = output[i-1] + input[i];  
    return output;  
}
```

Example:

input	42	3	4	7	1	10
output	42	45	49	56	57	67

The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where $\text{output}[i] = \text{input}[0] + \text{input}[1] + \dots + \text{input}[i]$

Sequential version is straightforward:

```
Vector<int> prefix_sum(const vector<int>& input) {  
    vector<int> output(input.size());  
    output[0] = input[0];  
    for(int i=1; i < input.size(); i++)  
        output[i] = output[i-1] + input[i];  
    return output;  
}
```

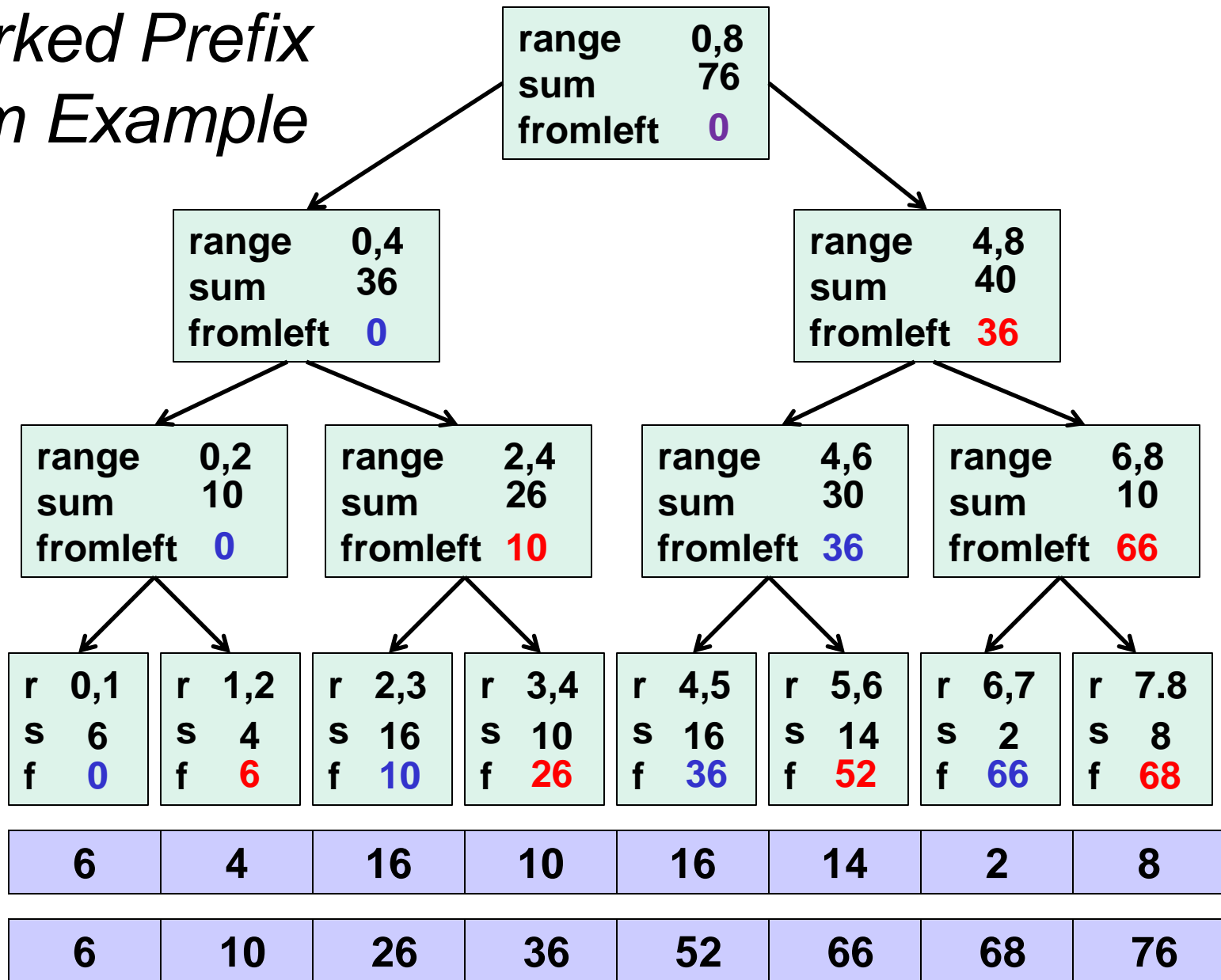
Why isn't this (obviously) parallelizable? Isn't it just map or reduce?

Work: $O(n)$

Span: $O(n)$ b/c each step depends on the previous.

Joins everywhere!

Worked Prefix Sum Example



Parallel prefix-sum

The parallel-prefix algorithm does two passes:

1. build a “sum” tree bottom-up
2. traverse the tree top-down, accumulating the sum from the left

Step 1:	Work: $O(n)$	Span: $O(\lg n)$
---------	--------------	------------------

Step 2:	Work: $O(n)$	Span: $O(\lg n)$
---------	--------------	------------------

Overall:	Work: $O(n)$	Span? $O(\lg n)$
----------	--------------	------------------

Paralellism (work/span)? $O(n/\lg n)$

Parallel prefix, generalized

Can we use parallel prefix to calculate the minimum of all elements to the left of i ?

Certainly! Just replace “sum” with “min” in step 1 of prefix and replace fromLeft with a fromLeft that tracks the smallest element left of this node’s range.

In general, what property do we need for the operation we use in a parallel prefix computation?

ASSOCIATIVITY! (And not commutativity, as it happens.)

Pack Analysis

Step 1: Work: $O(n)$ Span: $O(\lg n)$

Step 2: Work: $O(n)$ Span: $O(\lg n)$

Step 3: Work: $O(n)$ Span: $O(\lg n)$

Algorithm: Work: $O(n)$ Span: $O(\lg n)$

Parallelism: $O(n/\lg n)$

Parallelizing Quicksort

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

	Best / expected case <i>work</i>
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	$2T(n/2)$

How should we parallelize this?

Parallelize the recursive calls as we usually do in fork/join D&C.

Parallelize the partition by doing two packs (filters) instead.

Parallelizing Quicksort

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

- | | Best / expected case <i>work</i> |
|--|----------------------------------|
| 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 | $2T(n/2)$ |

How do we parallelize this? First pass: parallel recursive calls in step 3.

What span do we get?

$$T_{\infty}(n) = n + T_{\infty}(n/2) = n + n/2 + T_{\infty}(n/4) = \\ n/1 + n/2 + n/4 + n/8 + \dots + 1 \in \Theta(n)$$

(We replace the $O(n)$ term in $O(n) + T_{\infty}(n/2)$ with just n for simplicity of analysis.)

Analyzing $T_{\infty}(n) = \lg n + T_{\infty}(n/2)$

Turns out our techniques from way back at the start of the term will work just fine for this:

$$\begin{aligned} T_{\infty}(n) &= \lg n + T_{\infty}(n/2) && \text{if } n > 1 \\ &= 1 && \text{otherwise} \end{aligned}$$

We get a sum like:

$$\lg n + (\lg n) - 1 + (\lg n) - 2 + (\lg n) - 3 + \dots 3 + 2 + 1$$

Let's replace $\lg n$ by x :

$$x + x-1 + x-2 + x-3 + \dots 3 + 2 + 1$$

That's our "triangle" pattern: $O(k^2) = O((\lg n)^2)$