

CSE 332: Data Structures & Parallelism

Lecture 20: Parallel Prefix & Pack

Ruth Anderson

Autumn 2025

Administrative

- EX07 – On Graphs, programming: Due Fri Nov 14
- EX08 – On Shortest Paths, Due **Mon** Nov 17
- EX09 – On Fork Join programming, Due Fri Nov 21
- Resources!
 - **Conceptual Office Hours:** 11:30 Tues (Connor) and 11:30 Wed (Samarth) both in CSE1 006. A space to ask about **course content and topics only** *as opposed to direct help with exercises*.
 - [1-on-1 Meeting Requests](#) - Request a meeting with a staff member if you cannot make it to regularly scheduled office hours, or feel like you have an issue that requires a more in depth discussion.

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)**

The prefix-sum problem

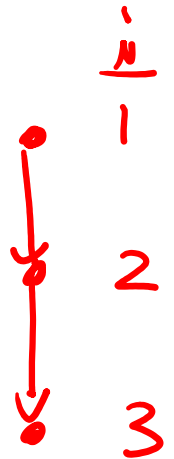
Given `int[] input`, produce `int[] output` where:

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

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

Sequential can be a CSE122 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[lo] .. input[hi-1]`
 - Build up from leaves; `parent.sum=left.sum+right.sum`
 - A leaf's sum is just its value; `input[i]`

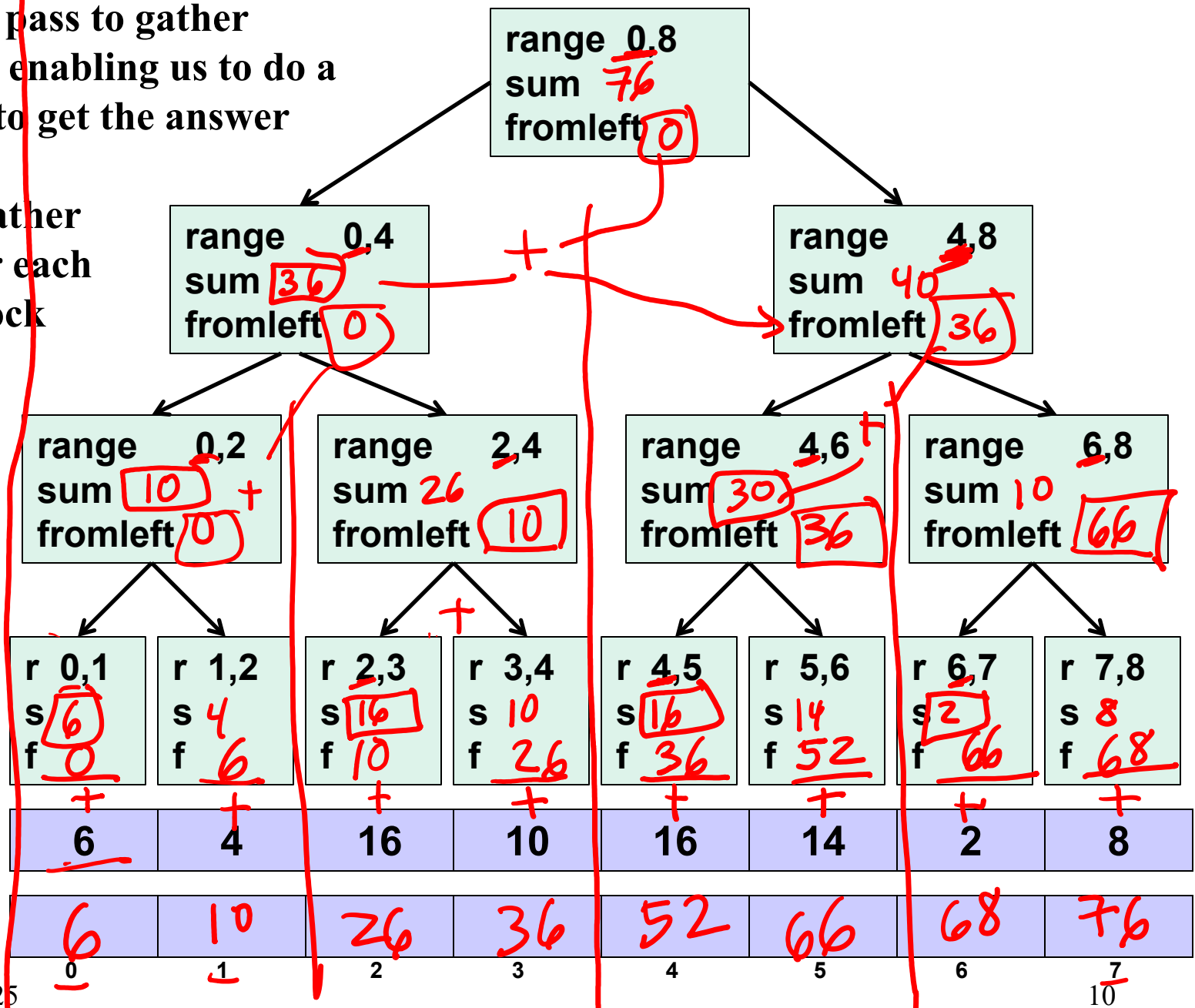
This is an easy fork-join computation: combine results by actually building a binary tree with all the sums of ranges

- 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

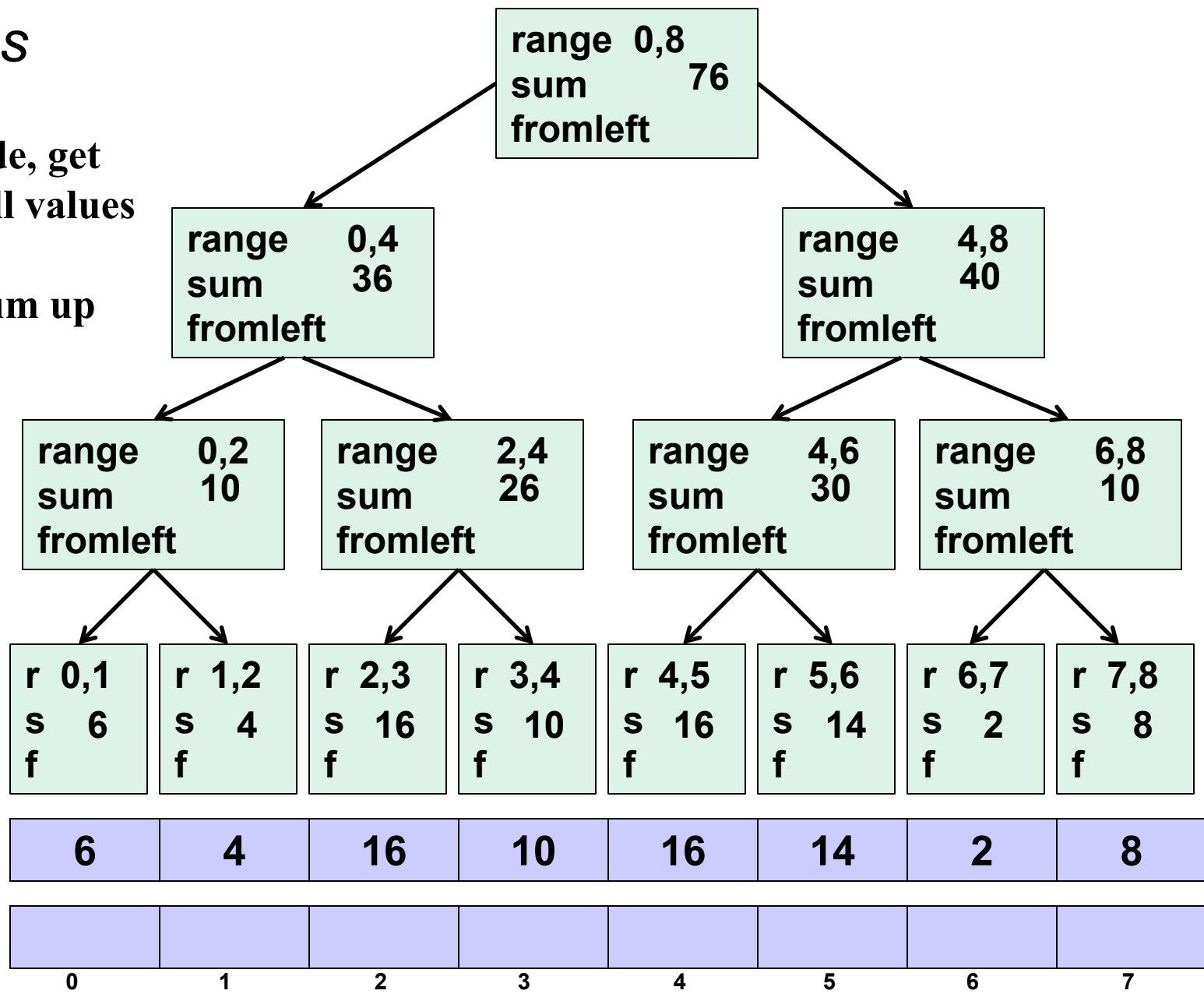
First we'll gather
the 'sum' for each
recursive block



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

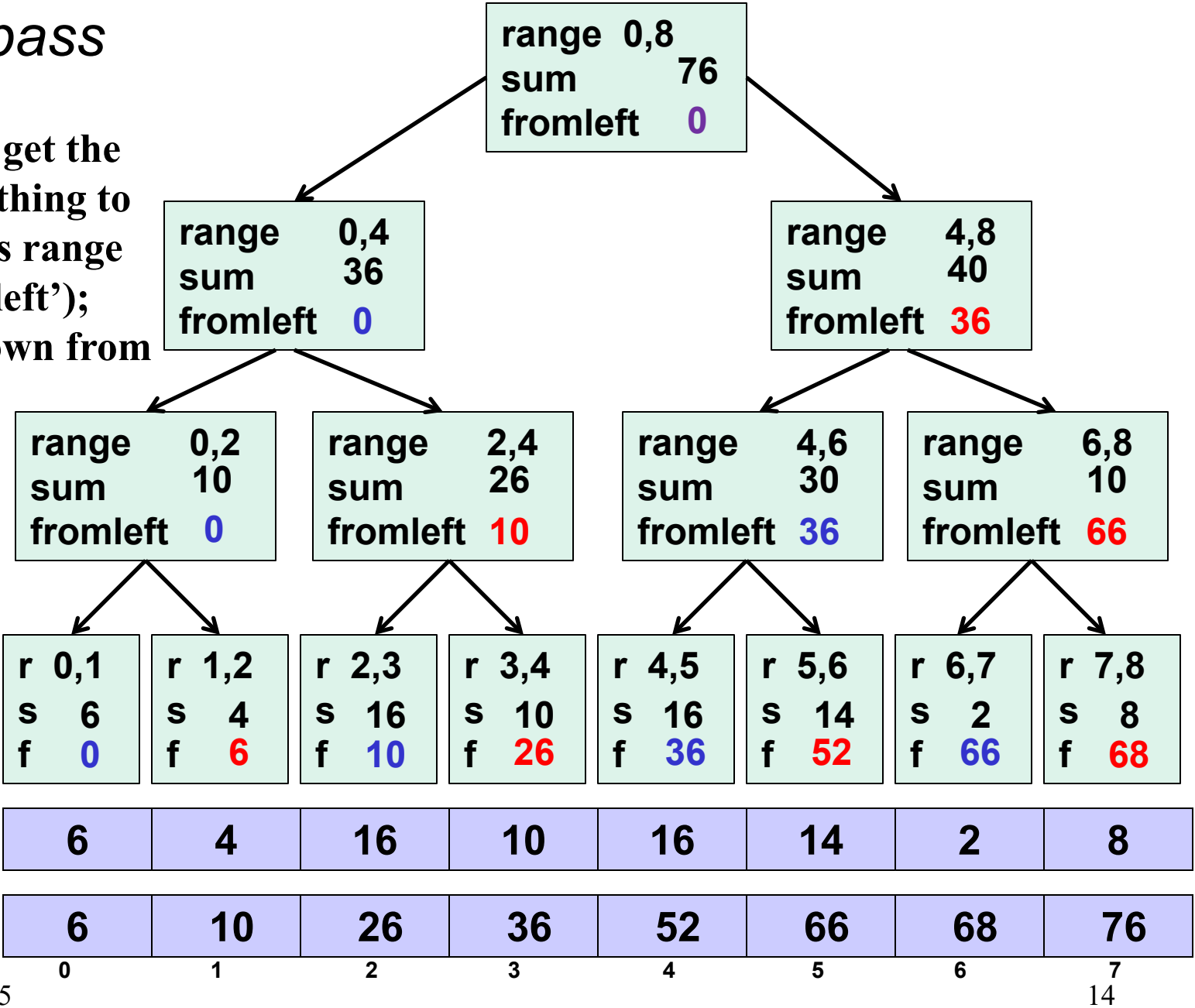
Analysis of first step: $O(n)$ work, $O(\log n)$ span

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

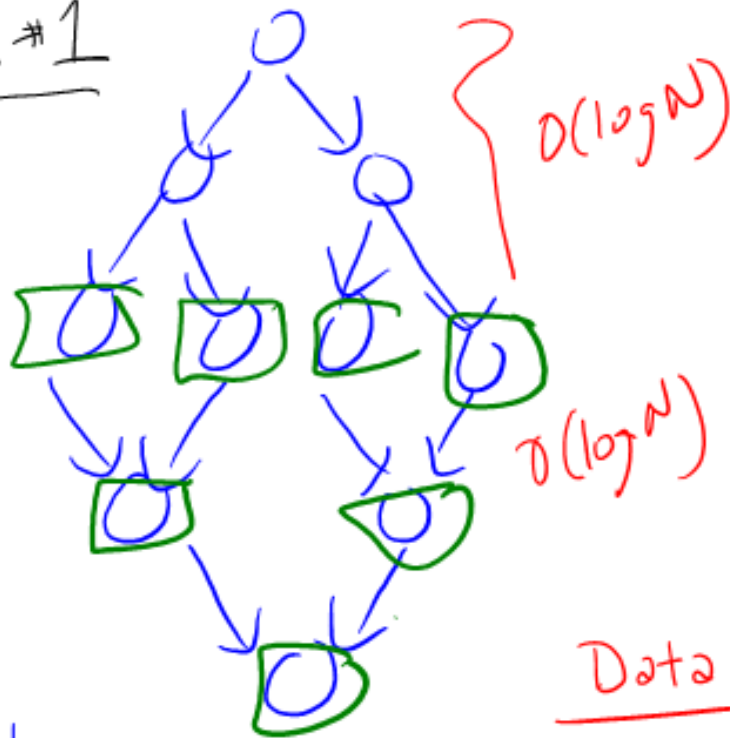
Total for algorithm: $O(n)$ work, $O(\log n)$ span

Second pass

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

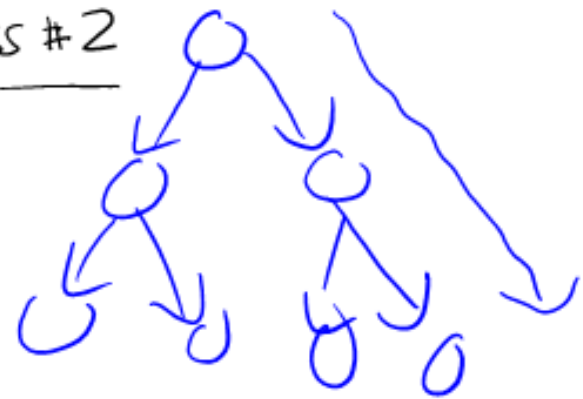


Pass #1



Fills in
"Sum"

Pass #2



Fills in
"From Left"

Data Structure



output

Parallel Prefix Sum Algorithm

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]
 0 1 2 3 4 5 6 7 8 9
 f: “is element > 10”

 output [17, 11, 13, 19, 24]
 0 1 2 3 4

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]

0 1 2 3 4 5 6 7 8 9

$O(\log N)$
span

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

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

0 1 2 3 4 5 6 7 8 9

$O(\log N)$
span

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

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

0 1 2 3 4

$O(\log N)$
span

```
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]
```

```
}
```

In this example,
Filter =
element > 10

*Parallel Pack =
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++){
    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. (see reading)