# Parallel Scans & Prefix Sums

COS 326

David Walker

Princeton University

Slide credits: Dan Grossman, UW

http://homes.cs.washington.edu/~djg/teachingMaterials/spac

# One More

So far we've seen a number of parallel divide-and-conquer algorithms

Today: One more key algorithm

- Parallel prefix:
  - Another "relentlessly sequential" algorithm parallelized
  - And its generalization to a parallel scan
- Application:
  - Parallel quicksort
  - Easy to get a little parallelism
  - With cleverness can get a lot

# The prefix-sum problem

val prefix_sum : int array -> int array

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

The simple sequential algorithm: accumulate the sum from left to right

- Sequential algorithm: Work: $O(n)$, Span: $O(n)$
- Goal: a parallel algorithm with Work: $O(n)$, Span: O(log n)

# Parallel prefix-sum

The trick: *Use 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 of sums bottom-up*

- the "up" pass

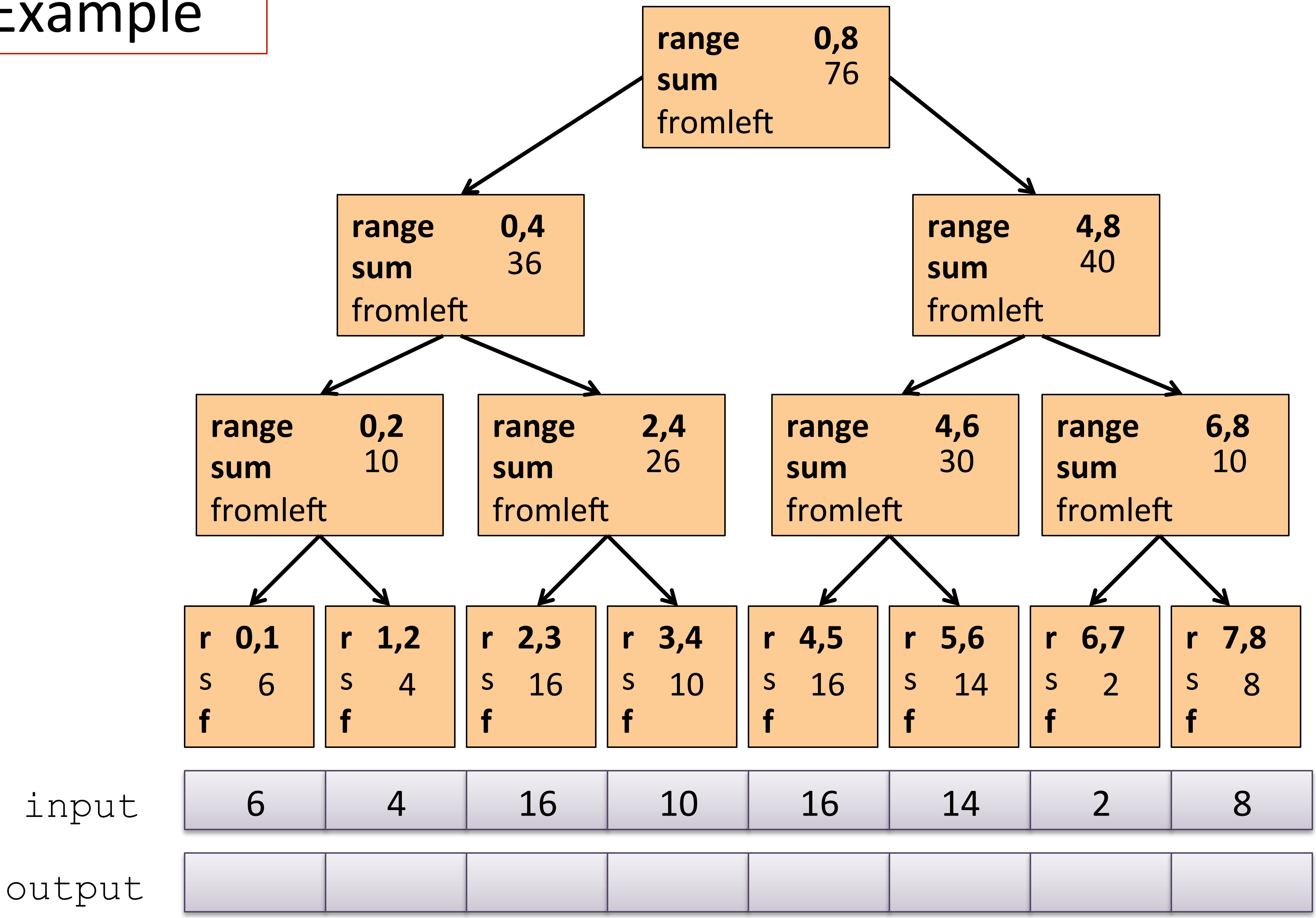
Second pass *traverses the tree top-down to compute prefixes*

- the "down" pass

Historical note:

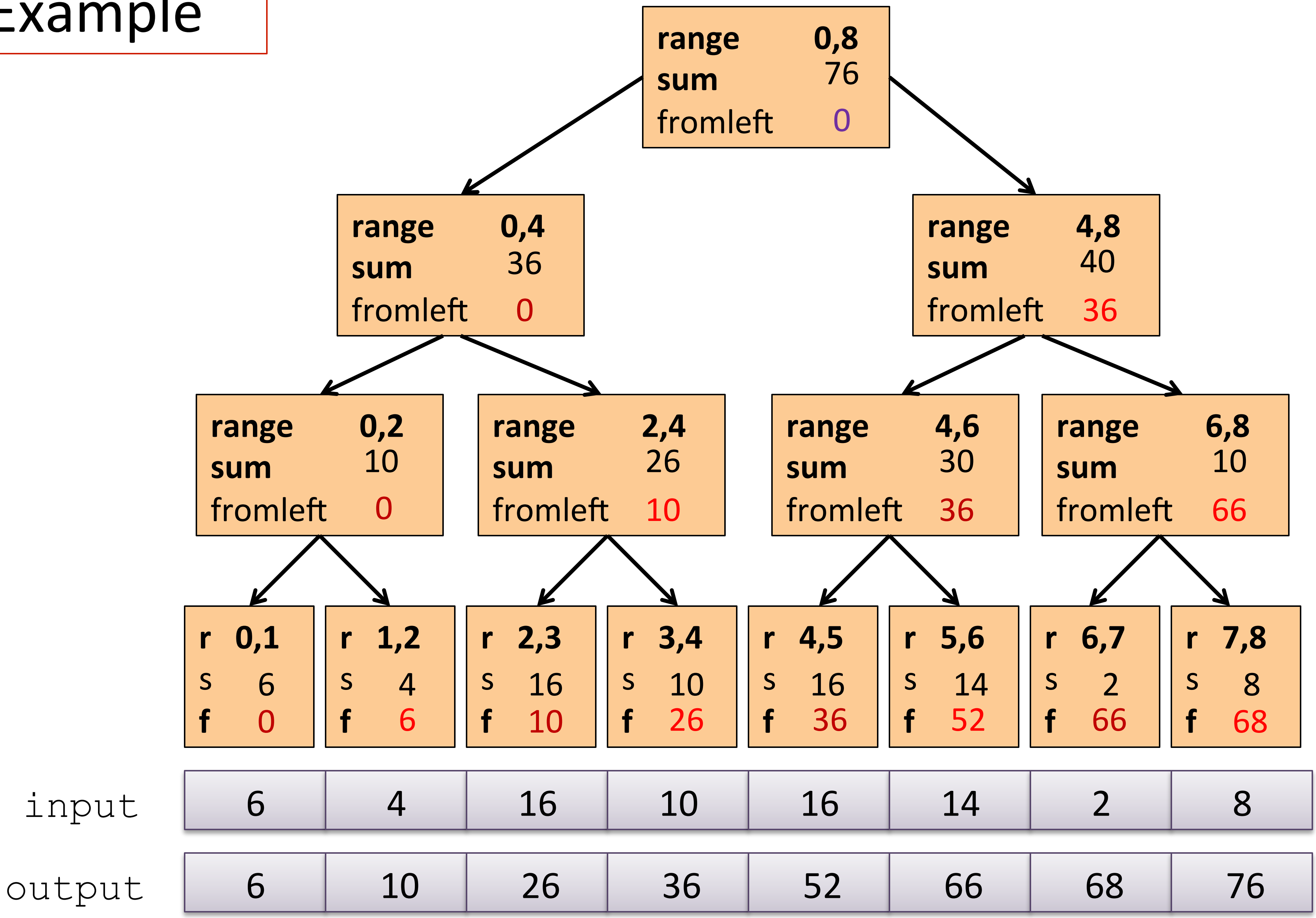
- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977

# Example



range 0,8
sum 76
fromleft
range 0,4
sum 36
fromleft
range 4,8
sum 40
fromleft
range 0,2
sum 10
fromleft
range 2,4
sum 26
fromleft
range 4,6
sum 30
fromleft
range 6,8
sum 10
fromleft
r 0,1
s 6
f
r 1,2
s 4
f
r 2,3
s 16
f
r 3,4
s 10
f
r 4,5
s 16
f
r 5,6
s 14
f
r 6,7
s 2
f
r 7,8
s 8
f
input
6
4
16
10
16
14
2
8
output

# Example


range 0,8
sum 76
fromleft 0
range 0,4
sum 36
fromleft 0
range 4,8
sum 40
fromleft 36
range 0,2
sum 10
fromleft 0
range 2,4
sum 26
fromleft 10
range 4,6
sum 30
fromleft 36
range 6,8
sum 10
fromleft 66
r 0,1 s 6 f 0
r 1,2 s 4 f 6
r 2,3 s 16 f 10
r 3,4 s 10 f 26
r 4,5 s 16 f 36
r 5,6 s 14 f 52
r 6,7 s 2 f 66
r 7,8 s 8 f 68
input
6 4 16 10 16 14 2 8
output
6 10 26 36 52 66 68 76

# The algorithm, pass 1

1. Up: 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**), i.e., **input[i]**

This is an easy parallel divide-and-conquer algorithm: "combine" results by actually building a binary tree with all the range-sums

- Tree built bottom-up in parallel

Analysis: $O(n)$ work, $O(\mathbf{log}\ n)$ span

# The algorithm, pass 2

2. Down: Pass down a value **`fromLeft`**
   - 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 parallel divide-and-conquer algorithm: traverse the tree built in step 1 and produce no result

- Leaves assign to **`output`**
- Invariant: **`fromLeft`** is sum of elements left of the node's range

Analysis: $O(n)$ work, $O(\mathbf{log}\ n)$ span

# Sequential cut-off

For performance, we need a sequential cut-off:

- Up:

  just a sum, have leaf node hold the sum of a range

- Down:

```
output.(lo) = fromLeft + input.(lo);
for i=lo+1 up to hi-1 do
  output.(i) = output.(i-1) + input.(i)
```

# Parallel prefix, generalized

Just as map and reduce are the simplest examples 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 filter …
  - Perfect for building on top of the "parallel prefix trick"

# Parallel Scan

scan (o) <x1,  ..., xn>

==

<x1,  x1 o x2,  ...,  x1 o ... o xn>

like a fold, except return
the folded prefix at each step

pre_scan (o) base <x1,  ..., xn>

==

<base, base o x1, ..., base o x1 o ... o xn-1>

sequence with o applied to all items
to the left of index in input

# Filter

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

Example: let f x = x > 10

```
    filter f <17, 4, 6, 8, 11, 5, 13, 19, 0, 24>
 == <17, 11, 13, 19, 24>
```

Parallelizable?

- Finding elements for the output is easy
- *But getting them in the right place seems hard*

# 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>
```

# Quicksort review

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

| | 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) |

How should we parallelize this?

# Quicksort

| | 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) |

Easy: Do the two recursive calls in parallel

- Work: unchanged. Total: $O(n \log n)$
- Span: now $T(n) = O(n) + 1T(n/2) = O(n)$

# Doing better

We get a $O(\mathtt{log}\ n)$ speed-up with an *infinite* number of processors. That is a bit underwhelming

– Sort $10^9$ elements 30 times faster

(Some) Google searches 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

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 filters!

- We know a parallel filter is $O(n)$ work, $O(\log n)$ span
- Parallel filter elements less than pivot into left side of **`aux`** array
- Parallel filter 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 filters at once but no effect on asymptotic complexity

With $O(\log n)$ span for partition, the total best-case and expected-case span for quicksort is

$$T(n) = O(\log n) + 1T(n/2) = O(\log^2 n)$$

# Example

Step 1: pick pivot as median of three

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

Steps 2a and 2c (combinable): filter less than, then filter greater than into a second array

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 0 | 3 | 5 | 2 | | | | |

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

Step 3: Two recursive sorts in parallel

- Can copy back into original array (like in mergesort)

# More Algorithms

- To add multi precision numbers.
- To evaluate polynomials
- To solve recurrences.
- To implement radix sort
- To delete marked elements from an array
- To dynamically allocate processors
- To perform lexical analysis. For example, to parse a program into tokens.
- To search for regular expressions. For example, to implement the UNIX grep program.
- To implement some tree operations. For example, to find the depth of every vertex in a tree
- To label components in two dimensional images.

*See Guy Blelloch "Prefix Sums and Their Applications"*

# Summary

- Parallel prefix sums and scans have many applications
  - A good algorithm to have in your toolkit!
- Key idea: An algorithm in 2 passes:
  - Pass 1: build a sum (or "reduce") tree from the bottom up
  - Pass 2: compute the prefix top-down, looking at the left-subchild to help you compute the prefix for the right subchild

# END