Parallel Prefix Sum – Scan

Outline

- Prefix computation problem
  - Given an array of values, compute the running sums
    - In general, addition is replaced by any associative operation
  - Easy to solve sequentially, not clear how to parallelize
- Parallel prefix computation
  - Divide and conquer algorithm that exposes parallelism that is not obvious from get-go
- Applications of parallel prefix computation
  - Many seemingly sequential problems can be parallelized in this way

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)

(Inclusive) Prefix-Sum (Scan) Definition

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$*, and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

Inclusive Scan Application Example

- Assume we have a 100-inch sandwich to feed 10
- We know how many inches each person wants
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sandwich quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan and cut in parallel
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- **Scan is a simple and useful parallel building block**
  - Convert recurrences from sequential :

```
for(j=1;j<n;j++)
    out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- **Useful for many parallel algorithms:**

•Radix sort
•Quicksort
•String comparison
•Lexical analysis
•Stream compaction
•Polynomial evaluation
•Solving recurrences
•Tree operations
•Histograms
•Etc.

Other Applications

- Assigning space in farmers market
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- …

A Inclusive Sequential Prefix-Sum

Given a sequence $[x_0, x_1, x_2, ...]$

Calculate output $[y_0, y_1, y_2, ...]$

Such that $y_0 = x_0$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

…

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i=1; i < Max_i; i++)
    y[i] = y[i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)

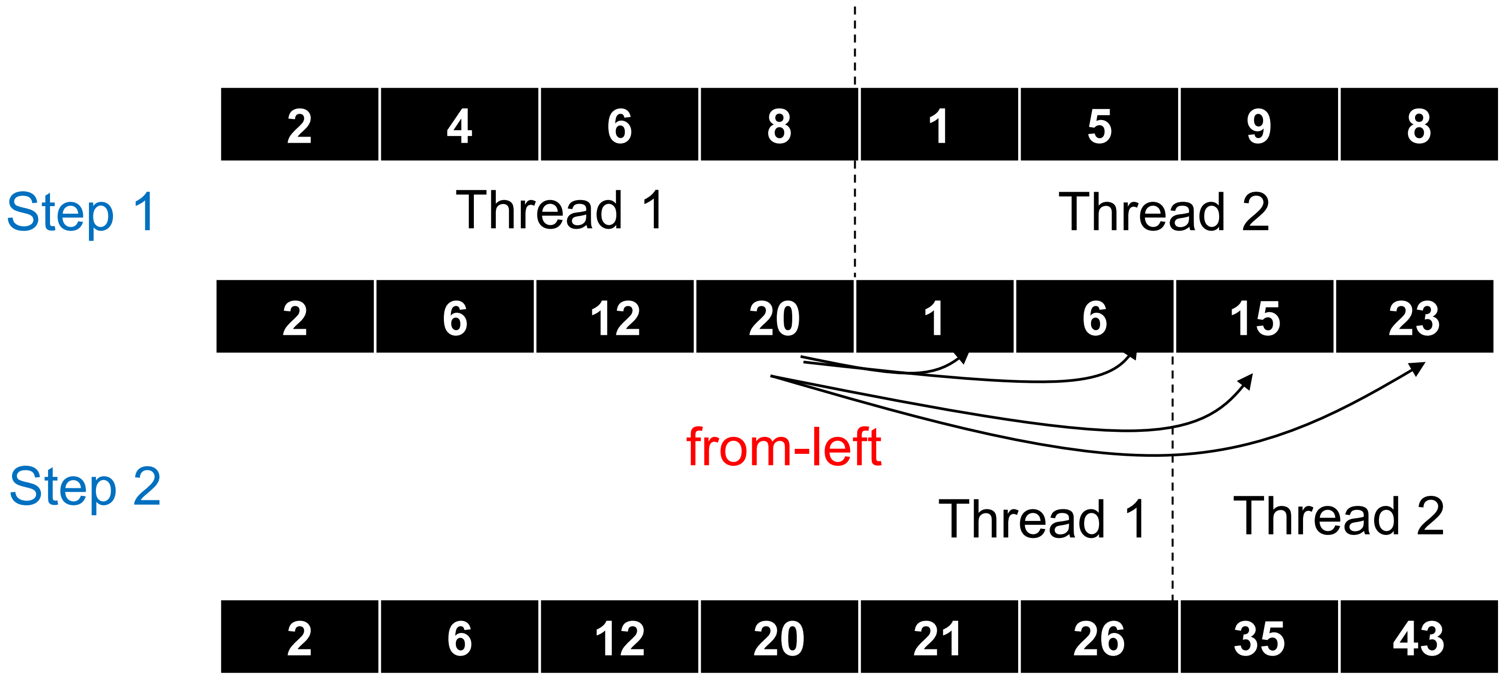
A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread add up all x elements needed for the y element

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

Parallel programming is easy as long as you don't care about performance.

How to parallelize?



- Assume two threads
- Step 1: threads compute prefix sum for left and right halves of array in parallel using some algorithm (say sequential algorithm)
- Step 2: add final element from first half (called from-left) to each element of second half in parallel
- Check: both steps are parallel, no ping-ponging of cache lines because of block distribution in second step

Recursive Python program

main.py

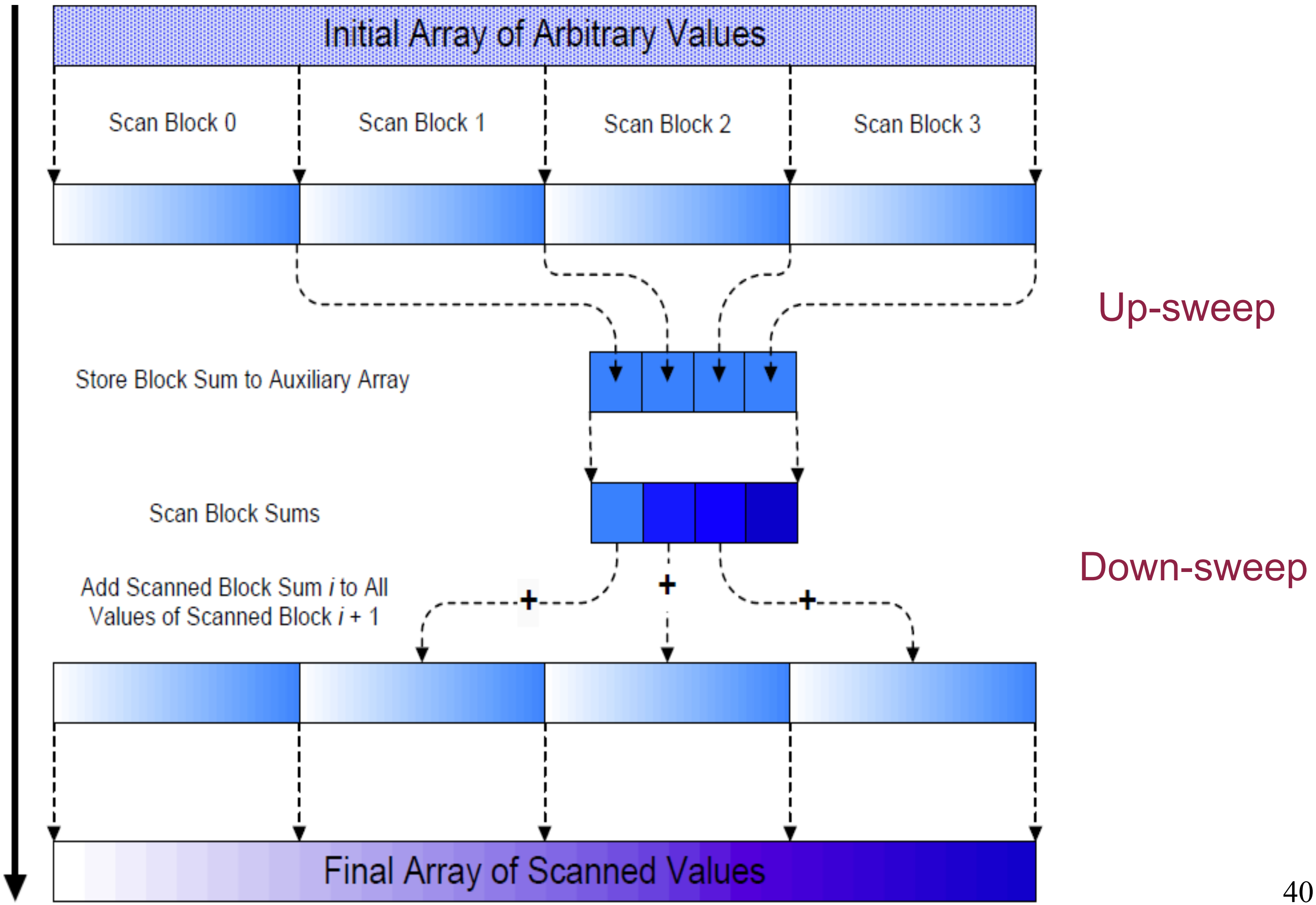
Run

```python
import math
a = [3,1,7,0,4,1,6,3,3,1,7,0,4,1,6]
#performs scan of array segment a[low,hi)
def scan(a,low,hi):
    if (hi <= low+1): #nothing to do if fewer than 2 elements
        return
    else:
        if (hi == low+2): #two element array; update neighbor
            a[low+1] = a[low+1]+a[low]
        else:
            #bisect array
            cut = low + math.floor((hi - low)/2)
            #scan left half of array
            scan(a,low, cut)
            #scan right half of array
            scan(a, cut, hi)
            #update right half of array
            for i in range(cut,hi):
                a[i] = a[i] + a[cut-1]
scan(a,0,len(a))
print(a)
```

Shell

```
[3, 4, 11, 11, 15, 16, 22, 25, 28, 29, 36, 36, 40, 41, 47]
>
```

Generalize to t (=4) threads



In the limit

- Assume large array, unbounded # of processors
- Up-sweep:
  - Divide input array into segments of length 2
  - Collect from-left values from each segment into another array like in previous slide
  - This array will be large too so perform previous two steps recursively on this array as well
  - Recursion stops when from-left array is size 1
- Down-sweep:
  - Update from-left arrays successively

Parallel prefix

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

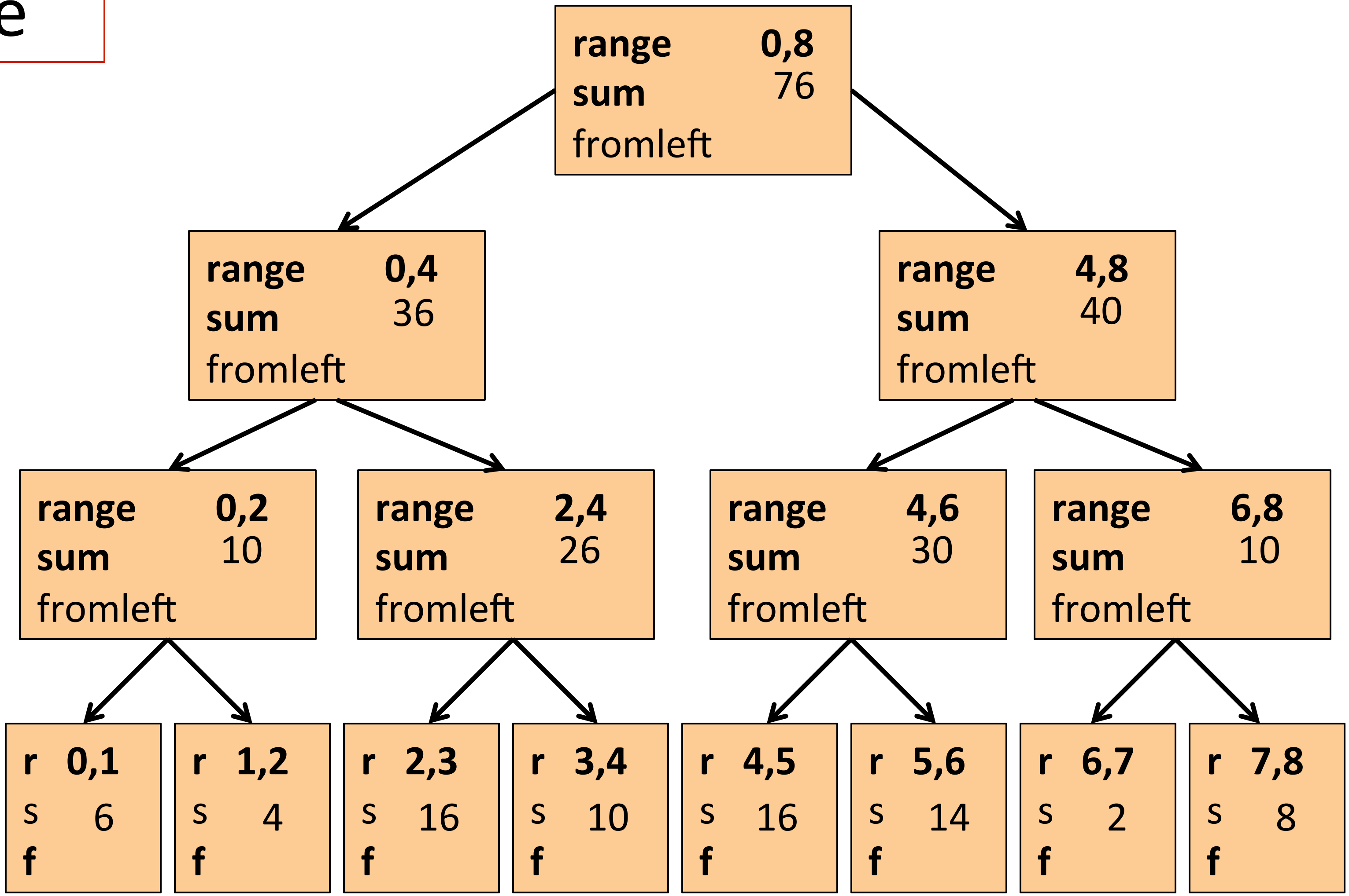
Second pass *traverses the tree top-down to compute prefixes*

- the "down" pass

Historical note:

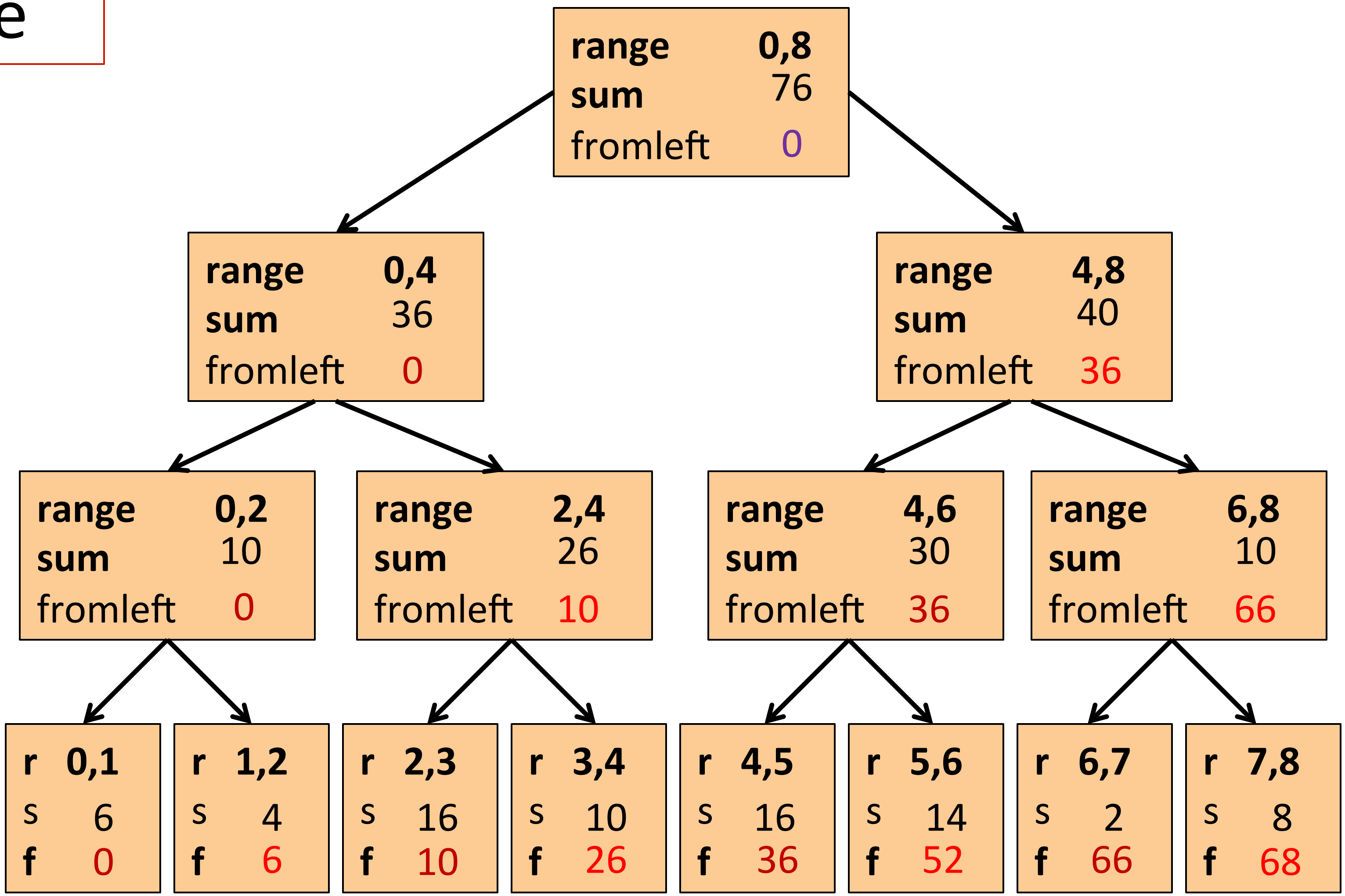
- Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977

Example



input	6	4	16	10	16	14	2	8
output								

Example



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, $O(\log n)$ 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 $O(n \texttt{ log } n)$

	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 $O(\log n)$ 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