Parallel Prefix

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar	Kristin Li	Winston Jodjana
Maggie Jiang	Hans Zhang	Michael Duan
Jeffery Tian	Annie Mao	

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-07-29A

Announcements

- Project 3 is out!
  - Fill out the partner surveys! We are only 1/3 of the way there
  - Checkpoint 1 has been pushed back a week
  - Please please, program together and don't delegate work
  - Everything in the project builds on each other
- Section this week is parallel programming!
  - Be ready to pair program in breakout rooms together
- Use this URL to request late days on projects:
  - https://grinch.cs.washington.edu/late

pollev.com/332summer :: tinyurl.com/332-07-29A

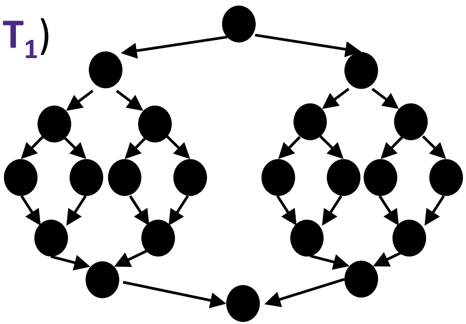
Lecture Outline

- **Amdahl's Law**
- Parallel Prefix – Prefix Sum
- Parallel Pack

pollev.com/332summer :: tinyurl.com/332-07-29A

Review: Work and Span

- Let T_P be the *running time* if there are **P** *processors* available
- Two important definitions:
  - **Work**: How long it'd take with 1 processor (ie, T_1)
    - Just "sequentialize" the recursive forking
    - Sum of all nodes in the graph
    - Simple map/reduction:
      - (assuming equal work done in every node and cutoff=1)



  - **Span**: How long it'd take with infinitely many processors (ie, T_∞)
    - Sum of all the nodes *on the longest path* in the graph
    - Simple map/reduction:
      - (assuming equal work done in every node and cutoff=1)

Review: Speed-up, Parallelism, and Optimality

- **Speed-up**, using **P** processors: T_1 / T_P
- **Perfect linear speed-up** occurs when $T_1 / T_P = P$
  - Perfect linear speed-up means doubling **P** halves running time
- **Parallelism**: T_1 / T_∞
  - Maximum possible speed-up; adding processors won't help

- We know T_P MUST BE greater than or equal to:
  - T_1 / P *(why?)*
  - T_∞ *(why?)*

- So an *asymptotically optimal* execution must be:

$$O(\ (T_1/P) + T_\infty\ )$$

  - First term dominates for small **P**, second for large **P**

Amdahl's Law

- Let the work (**T_1**) be 1 unit of time and **S** be the unparallelizable portion of execution time:

$$T_1 = 1 = S + (1-S)$$

- Suppose *perfect linear speed-up* on the parallelizable portion. Then:

$$T_P = S + (1-S)/P$$

- Amdahl's Law states the speed-up with P processors is:

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

- and the parallelism (maximum possible speed-up) is:

$$T_1 / T_\infty = 1 / S$$

Implications of Amdahl's Law

Speedup:	$T_1 / T_P = 1 / (S + (1-S)/P)$
Max Parallelism:	$T_1 / T_\infty = 1 / S$

- In "the good old days" (1980-2005), ~12 years = 100x speedup
- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1. What portion of the program must be parallelizable to get 100x speedup?
  - *For 256 processors to get at least 100x speedup, we need*

    $$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

  - *Which means* $\mathbf{S} \leq .0061$ *(i.e., 99.4% must be parallelizable)*

The Challenge Posed by Amdahl's Law

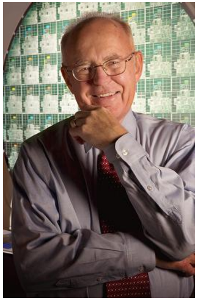
- Amdahl's Law tells us unparallelized parts become a bottleneck very quickly
  - But it *doesn't* tell us additional processors are worthless

- … because we can find new parallel algorithms
  - Some things that seem sequential turn out to be parallelizable
  - Eg: How can we parallelize a 'running sum' array?

input	6	4	16	10	16	15	2	8
output	6	10	26	36	52	67	69	77

- We can also change the problem we're solving
  - Eg: Video games use tons of parallel processors; they are not rendering 10-year-old graphics faster

Moore and Amdahl

- Moore's "Law" is an **observation** about the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl's Law is a **mathematical theorem**
  - Diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

Lecture Outline

- Amdahl's Law

- **Parallel Prefix: Prefix-Sum**
  - This was our example "unparallelizable" problem
  - It turns out there's a "key trick" that reveals surprising parallelization
  - Enables other things like packs (aka filters)

pollev.com/332summer :: tinyurl.com/332-07-29A

The Prefix-Sum Problem (1 of 2)

- Given `int[] input`, produce `int[] output` where:
  `output[i] = input[0]+input[1]+…+input[i]`

input	6	4	16	10	16	15	2	8
output	6	10	26	36	52	67	69	77

The Prefix-Sum Problem (2 of 2)

input	6	4	16	10	16	15	2	8
output	6	10	26	36	52	67	69	77

- Sequential solution feels like 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;
}
```

O(n)

- Doesn't seem parallelizable!
  - Work: O(n), Span: O(n)
  - There's a different algorithm with Work: O(n), Span: O(log n) 😮

Parallel Prefix-Sum: Overview

1968? 1973?

Recent

- Local bragging:
  - Algorithm due to R. Ladner and M. Fischer *at UW in 1977*
  - Richard Ladner joined the UW faculty in 1971 and hasn't left

- Parallel-prefix sum algorithm has two passes:
  - Each pass is O(n) work and O(log n) span
  - So – as with array summing – parallelism is n/log n: exponential!

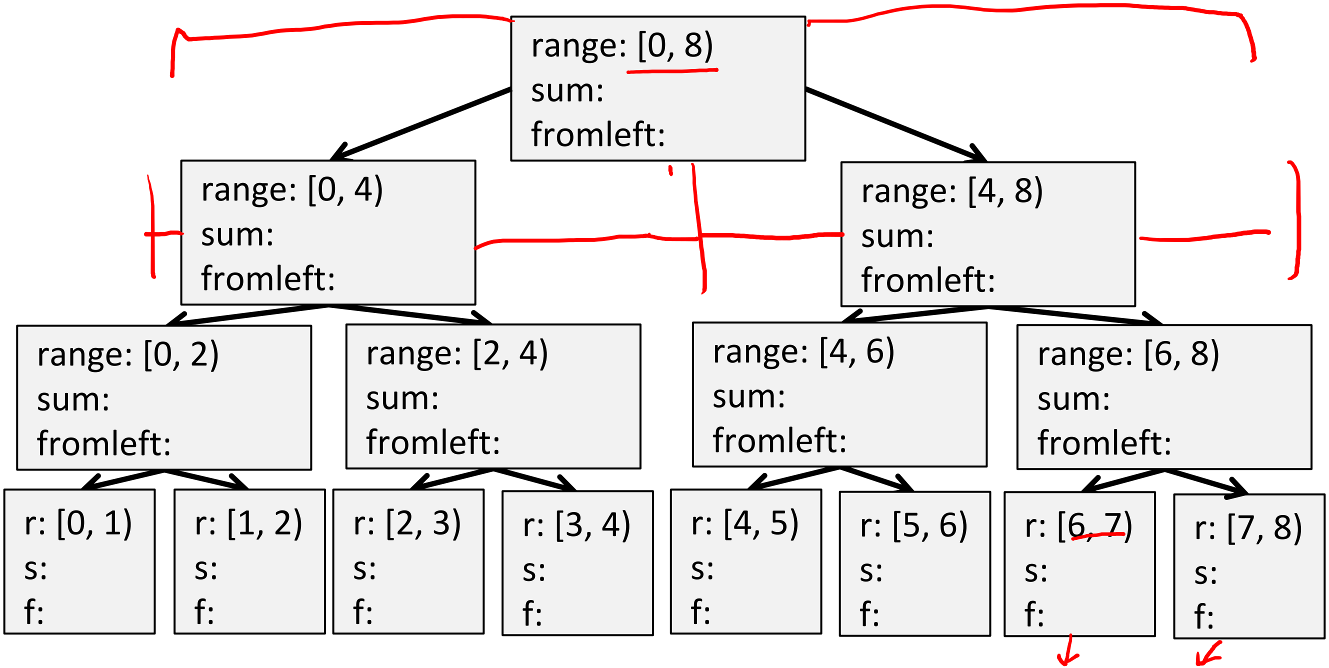
Parallel Prefix-Sum: The "Up" Pass: Overview

- This first pass builds a *binary tree* from the bottom: the "up" pass

- Parallel Prefix-Sum's binary tree:
  - Internal nodes have a range and sum of [lo, hi)
    - … and the root has [0, n+1)
  - Left child has range and sum of [lo, middle)
  - Right child has range and sum of [middle, hi)
  - A leaf has range and sum of [i,i+1); the sum is simply input[i]

- Unlike parallel-sum, we actually *create the tree;* we need it for the next pass (the "down" pass)
  - Doesn't have to be an actual tree; could use an array (eg, binary heap)

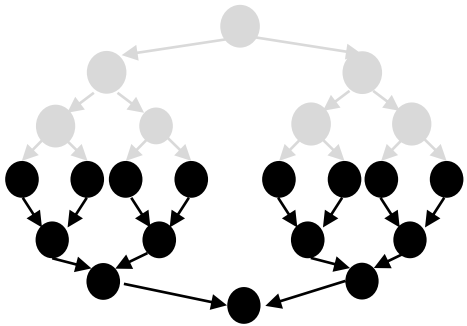
Parallel Prefix-Sum's Tree



input	6	4	16	10	16	15	2	8
output								

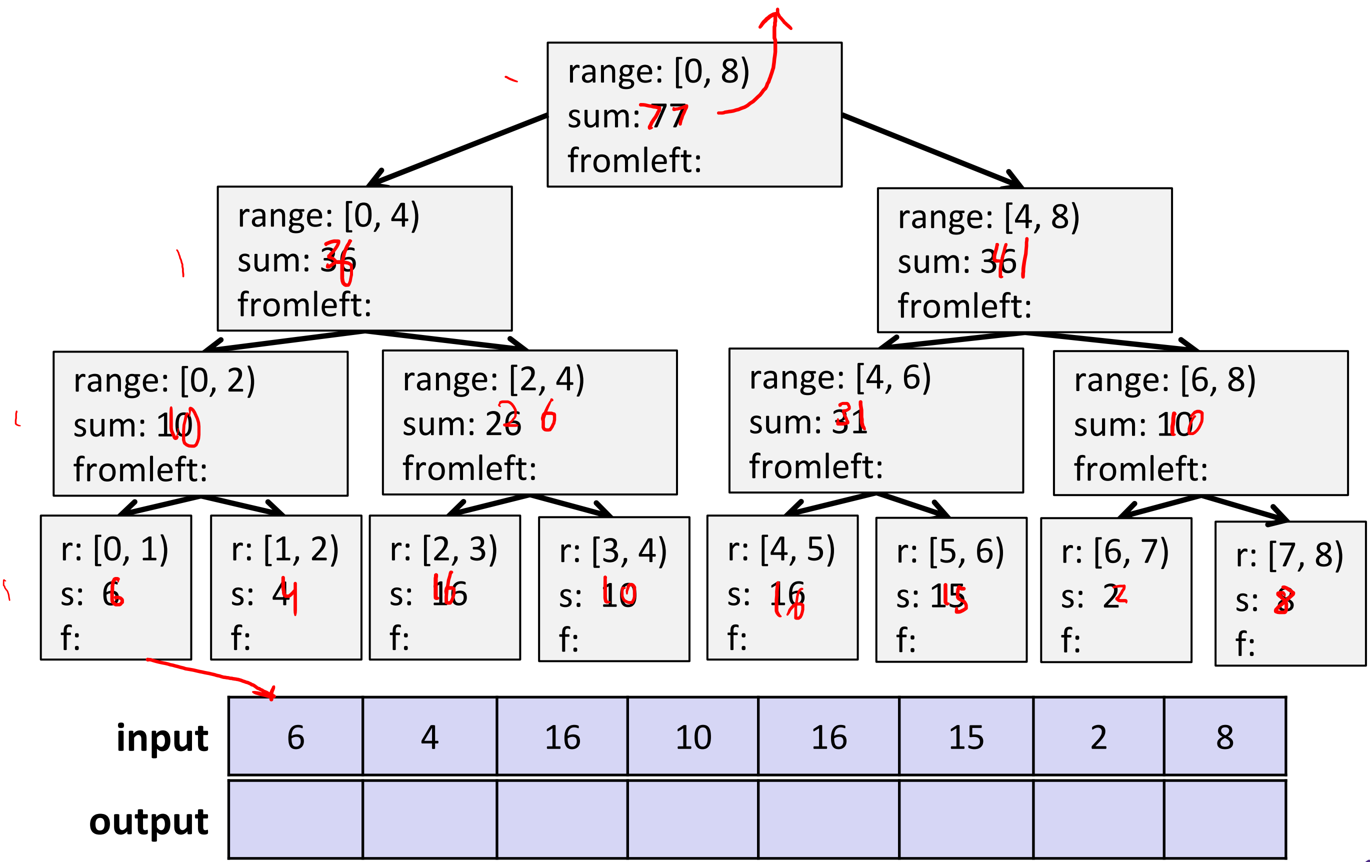
Parallel Prefix-Sum: The "Up" Pass: Details

- Parent has range and sum of [lo, hi)
  - left has [lo, middle), and right has [middle, hi)



- Build sum from the bottom of the tree:
  - A leaf's sum is just its value: input[i]
- Easy fork-join computation!
  - Save the partial sums from our parallel-sum algorithm
  - Tree is built from bottom-up, in parallel
- Analysis of the up pass:
  - Work: O(n)
  - Span: O(log n)

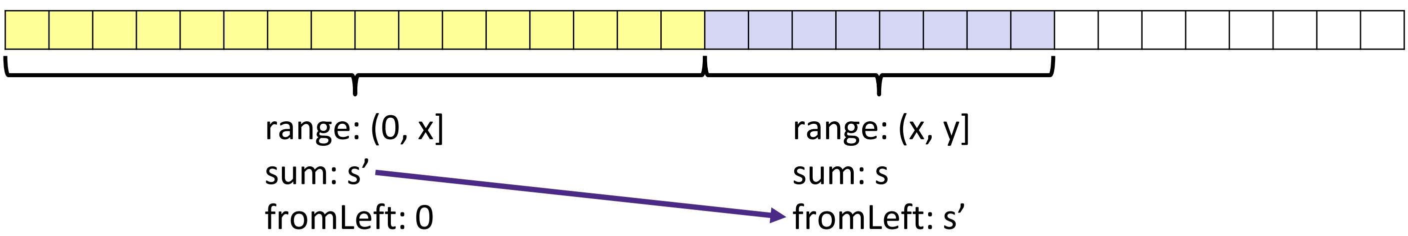
Parallel Prefix-Sum's Example: The "Up" Pass



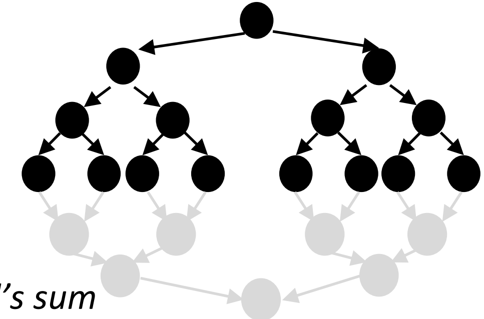
Parallel Prefix-Sum: The "Down" Pass: Overview

- This second pass *processes* the binary tree: the "down" pass

- All nodes have a range and sum of [lo, hi); now we populate their fromLeft fields
  - Invariant: fromLeft is sum of elements left of the node's range: [0, lo)



Parallel Prefix-Sum: The "Down" Pass: Details



- Propagate fromLeft down:
  - Root starts with a fromLeft of 0 *(why?)*
  - Internal 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*
  - At the leaf, *must also* `output[i] = fromLeft + input[i]`

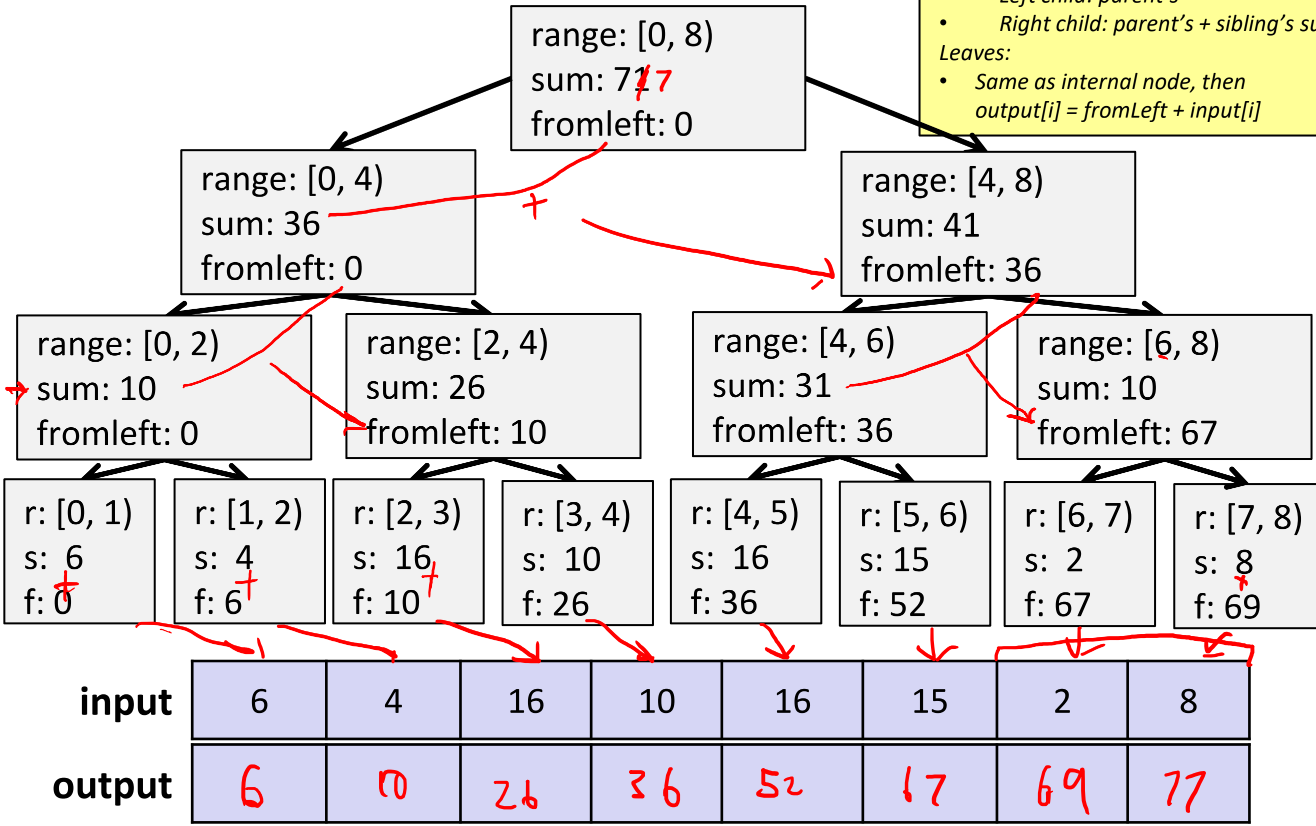
- Also an easy fork-join computation!
  - Traverse the tree built in step 1
  - Don't produce an explicit result; the leaves will assign to `output`

- Analysis of down pass: Work: O(n), Span: O(log n)
- Total for algorithm: Work: O(n), Span: O(log n)

Parallel Prefix-Sum's Example: The "Down" Pass

Internal nodes:
- *Left child: parent's*
- *Right child: parent's + sibling's sum*

Leaves:
- *Same as internal node, then output[i] = fromLeft + input[i]*



Sequential Cutoff for Prefix-Sum

- Adding a sequential cut-off isn't too bad:

1. Propagating up the sums:
   - Leaf node just holds the sum of a range of values (i.e., sequentially compute sum for that range)
   - The tree itself will be shallower

2. Propagating down the fromLefts:
   - Have leaf compute prefix sum sequentially over its [lo,hi), then:

```
output[lo] = fromLeft + input[lo];
for(i=lo+1; i < hi; i++)
  output[i] = output[i-1] + input[i]
```

Generalized Parallel-Prefix-Sum = Parallel-Prefix

- Sum-array was an example of a common pattern

- Prefix-sum is also a pattern that arises in 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

sum → max of range

sum to the left → max to the left

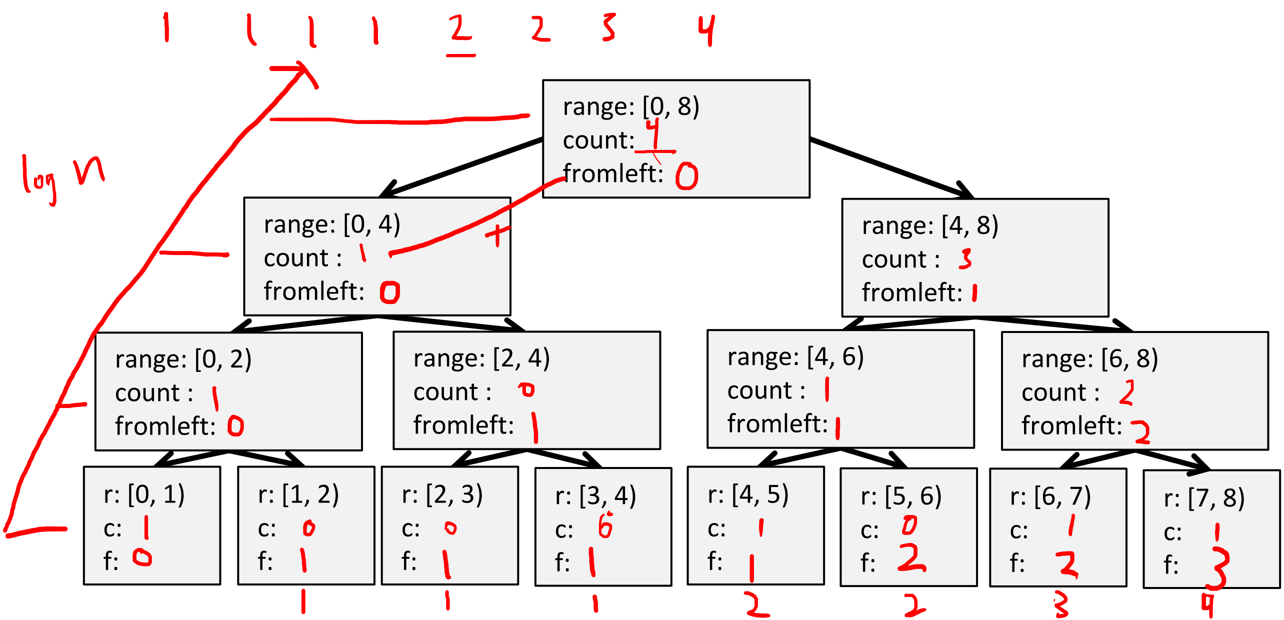
You now know the "one weird trick": parallel-prefix!

right.fromLeft = max(left.max, parent.fromLeft)

Your Turn!

- Write your answers on a piece of scratch paper:
  - Given the following array, fill in the Parallel Prefix tree such that it creates a new array with `arr[i]` containing the counts of values >10 in the index to the left of, and including, `i`:

`[17, 4, 6, 8, 11, 5, 13, 19]`



Lecture Outline

- Amdahl's Law
- Parallel Prefix – Prefix Sum
- **Parallel Pack**

pollev.com/332summer :: tinyurl.com/332-07-29A

Pack (aka "Filter")

- Given an array `input`, produce an array `output` containing only elements such that `f(element)` is true
  - E.g.: `input: [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`
    - `f: "is element > 10"`
    - `output: [17, 11, 13, 19, 24]`

- Parallelizable?
  - Yes: determining *whether* an element belongs in the output is easy
  - No: determining *where* an element belongs in the output is hard; seems to depend on previous results....

We Already Know Parallel-Pack!

In this example, filter = element > 10?

- Parallel-Pack = Parallel-Map + Parallel-Prefix + Parallel-Map!
  1. **Parallel map** to compute a bit-vector for filtered 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 output:

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

```
output = new array of size bitsum[n-1]
FORALL (i=0; i < input.length; i++){


}
```

We Already Know Parallel-Pack!

In this example,
filter = element > 10?

- Parallel-Pack = Parallel-Map + Parallel-Prefix + Parallel-Map!

1. **Parallel map** to compute a bit-vector for filtered 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 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];
}
```

Parallel-Pack Comments

Parallel-Pack:
1. Parallel-map: compute bit-vector
2. Parallel-prefix: compute bit-sum
3. Parallel-map: produce output

- First two steps can be combined into a prefix-sum
  - Different base case for the prefix sum
  - No effect on asymptotic complexity
- Combine third step into the down pass of the prefix-sum
  - Again, no effect on asymptotic complexity
- Analysis: $O(n)$ work, $O(\texttt{log}\ n)$ span
  - 2 or 3 passes, but both are constants ☺
- Parallelized packs will help us parallelize quicksort…