

Parallel Prefix

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- ❖ Ex 3,4,5 grades out – regrade requests open until Friday noon
- ❖ Don't forget to submit the midterm tonight!
- ❖ Section tomorrow

Lecture Outline

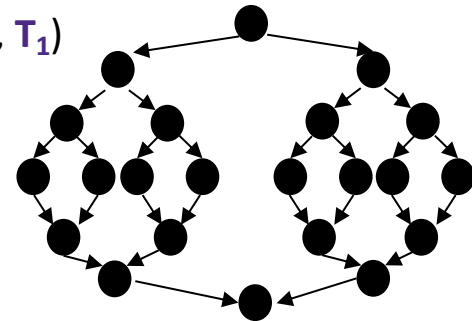
- ❖ **Review of Parallelism Analysis**
- ❖ Parallelized Prefix-Sum
 - The Prefix-Sum problem
 - How can we parallelize it?
- ❖ Parallel Pack

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

And Now for the Good / Bad News ...

- ❖ In practice, it's common that a program has:
 - a) Parts that **parallelize** well:
 - E.g. maps/reduces over arrays and trees
 - b) ... and parts that **don't parallelize** at all:
 - E.g. reading a linked list
 - E.g. waiting on input
 - E.g. computations where each step needs the results of previous step

- ❖ These unparallelizable parts turn out to be a big bottleneck, which brings us to Amdahl's Law ...

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:

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

Amdahl's Law Example

- ❖ Recall: $T_1 = 1 = S + (1-S)$ and $T_p = S + (1-S)/P$
- ❖ Suppose: $T_1 = 1/3 + 2/3 = 1$ (eg, $T_1 = 100s = 33s + 67s$)
- ❖ Then: $T_p = 33 \text{ sec} + (67 \text{ sec})/P$
 $T_3 = 33 \text{ sec} + (67 \text{ sec})/3 =$
 $T_6 = 33 \text{ sec} + (67 \text{ sec})/6 =$
 $T_{67} = 33 \text{ sec} + (67 \text{ sec})/67 =$
- ❖ If 33% of a program is sequential, a billion processors won't give a speedup over 3!!!
- ❖ No matter how many processors you use, your speedup is bounded by the sequential portion of the program

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*
$$100 \leq 1 / (S + (1-S)/256)$$
 - *Which means $S \leq .0061$ (i.e., 99.4% must be parallelizable)*

Lecture Outline

- ❖ Review of Parallelism Analysis
- ❖ **Parallelized Prefix-Sum**
 - The Prefix-Sum problem
 - How can we parallelize it?
- ❖ Parallel Pack

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

The Prefix-Sum Problem (1 of 2)

❖ 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	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;
}
```

- ❖ 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?



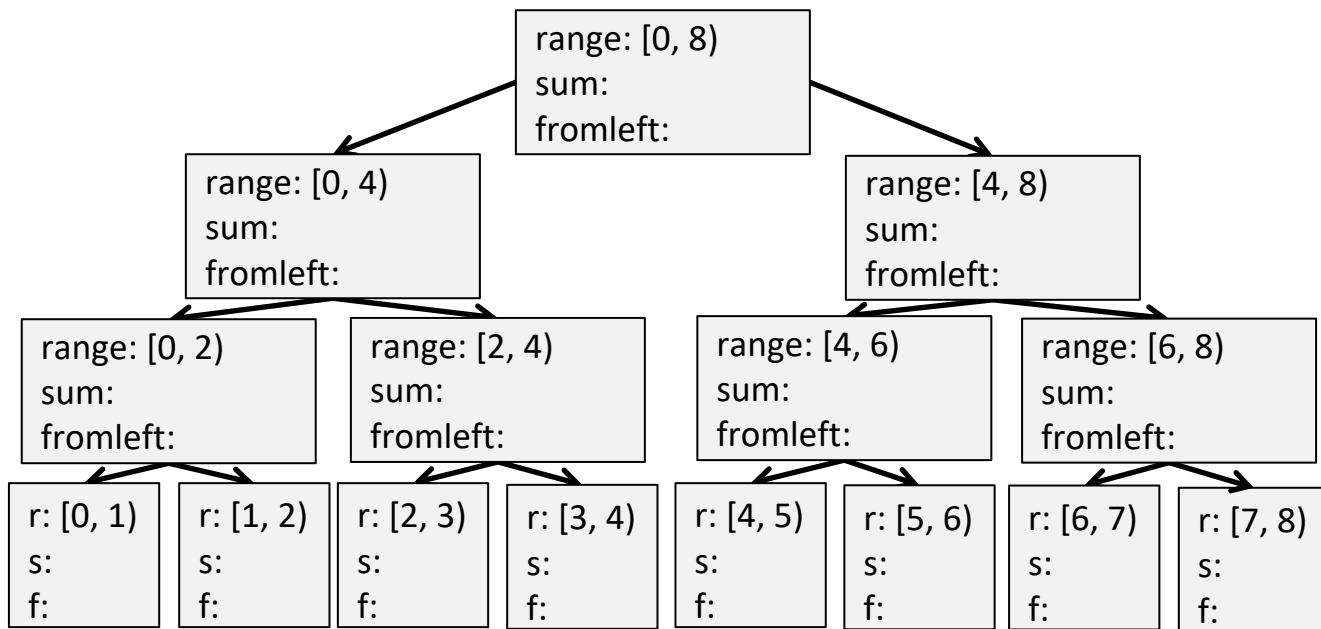
Recent

- ❖ 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)

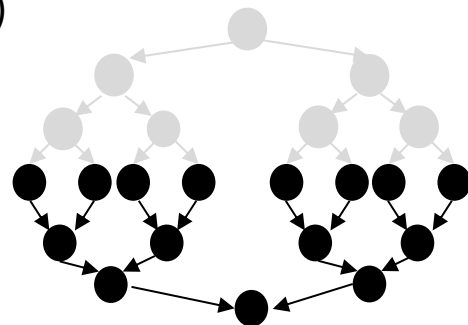
Parallel Prefix-Sum's Tree



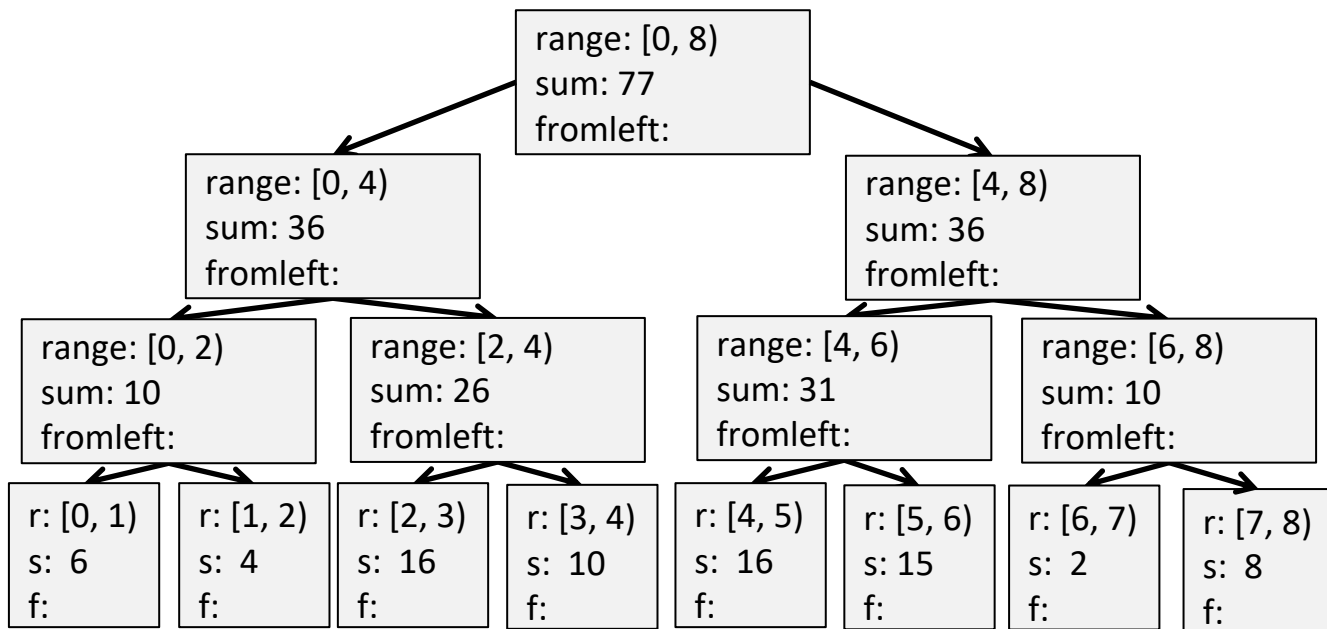
input	6	4	16	10	16	15	2	8
output								

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:
 - Span:



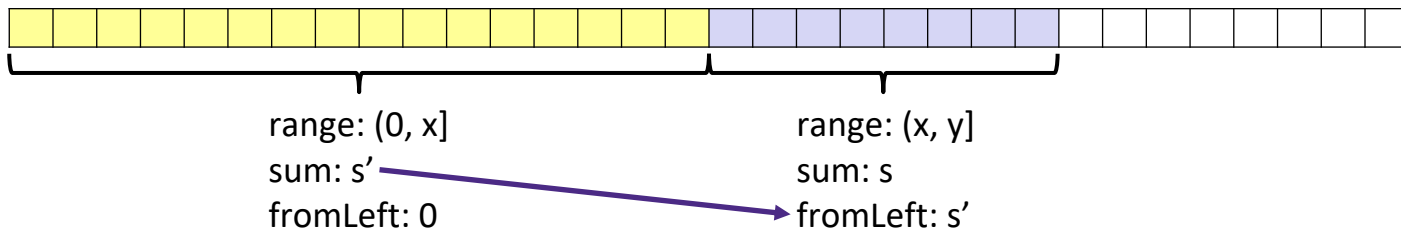
Parallel Prefix-Sum's Example: The "Up" Pass



input	6	4	16	10	16	15	2	8
output								

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: $[0, lo)$

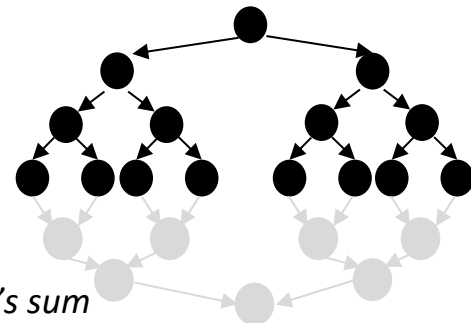


Parallel Prefix-Sum: The “Down” Pass: Details

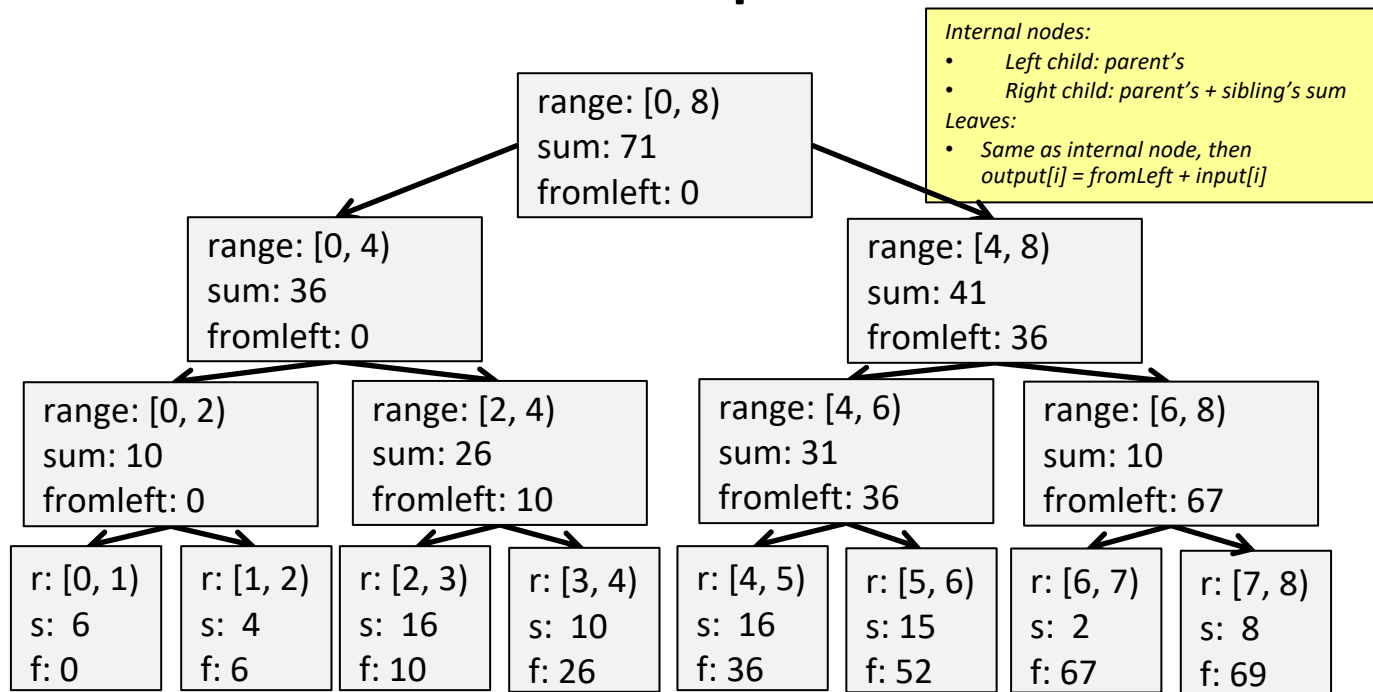
- ❖ Propagate fromLeft down:
 - Root starts with a fromLeft of 0 (*why?*)
 - Internal node takes its fromLeft value and
 - Passes its left child *the same* fromLeft
 - Passes its right child *its fromLeft plus its left child's sum*
 - At the leaf, *must also* $output[i] = fromLeft + input[i]$

- ❖ 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: _____, Span: _____
- ❖ Total for algorithm: Work: _____, Span: _____



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



input	6	4	16	10	16	15	2	8
output								

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

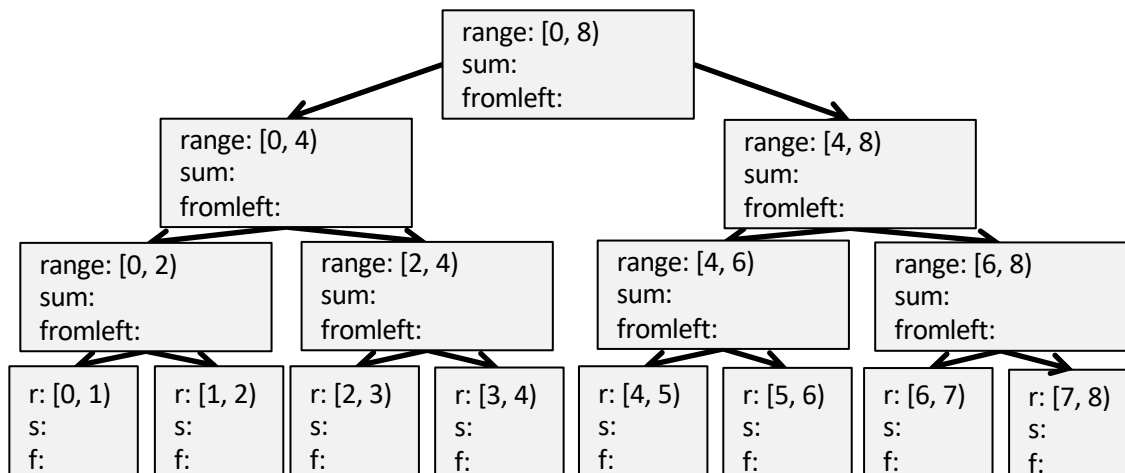
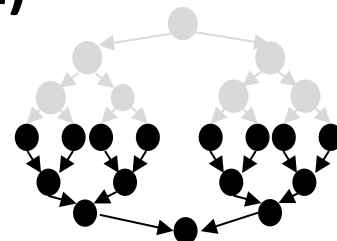
- ❖ Sum-array was an example of a common pattern
- ❖ Prefix-sum is also a pattern that arises in many problems:
 - Minimum, maximum of all elements **to the left of i**
 - Is there an element **to the left of i** satisfying some property?
 - Count of elements **to the left of i** satisfying some property

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



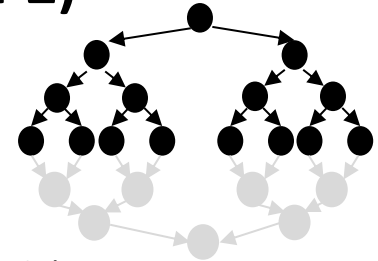
Parallel Prefix-Sum: Summary (1 of 2)

- ❖ Parent has range and sum of $[lo, hi)$
 - left has $[lo, middle)$, and right has $[middle, hi)$
- ❖ “Up” Pass: build sum from the bottom of the tree:
 - A leaf’s sum is just its value: $input[i]$
- ❖ Output of the “up” pass is this binary tree:

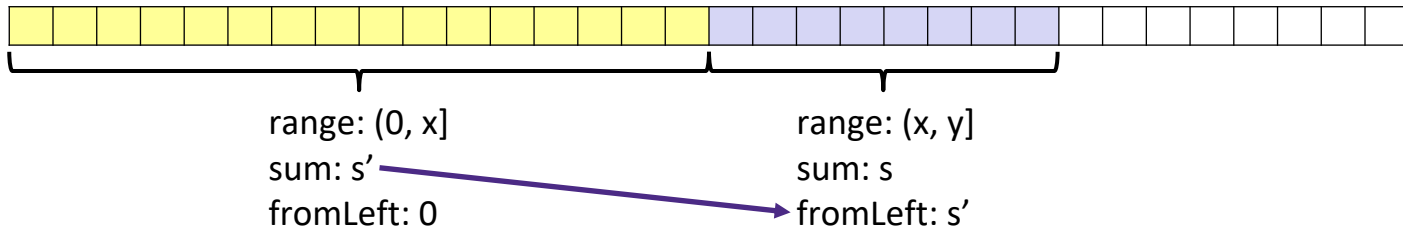


Parallel Prefix-Sum: Summary (2 of 2)

- ❖ The “down” pass: process the binary tree to populate the fromLeft fields



- fromLeft is sum of elements left of the node's range: $[0, lo)$



- 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 $output[i] = fromLeft + input[i]$
- ❖ Total for algorithm: Work: $O(n)$, Span: $O(\log n)$

Parallel-Prefix

- ❖ 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

Lecture Outline

- ❖ Review of Parallelism Analysis
- ❖ Parallelized Prefix-Sum
 - The Prefix-Sum problem
 - How can we parallelize it?
- ❖ **Parallel Pack**

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++){

}
```

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, $O(\log n)$ span
 - 2 or 3 passes, but both are constants 😊
- ❖ Parallelized packs will help us parallelize quicksort...