A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency

Lecture 3
Parallel Prefix, Pack, and Sorting

Steve Wolfman, based on work by Dan Grossman

MOTIVATION: *Pack*

AKA, **`filter`**, like getting all elts less than the pivot in QuickSort.

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

Example: **`input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`**
`f: is elt > 10`
`output [17, 11, 13, 19, 24]`

Parallelizable? Sure, using a list concatenation reduction.

Efficiently parallelizable on arrays?
Can we just put the output straight into the array *at the right spots*?

MOTIVATION: *Pack as map, reduce, prefix combo??*

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

Example: **`input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`**
`f: is elt > 10`

Which pieces can we do as maps, reduces, or prefixes?

MOTIVATION: *Parallel prefix sum to the rescue (if only we had it!)*

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.size(); i++){
  if(bits[i])
    output[bitsum[i]-1] = input[i];
}
```


The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where **`output[i] = input[0]+input[1]+…+input[i]`**

Sequential version is straightforward:

```
vector<int> prefix_sum(const vector<int>& input){
  vector<int> output(input.size());
  output[0] = input[0];
  for(int i=1; i < input.size(); i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Example:

input	42	3	4	7	1	10
output						

The prefix-sum problem

Given a list of integers as input, produce a list of integers as output where **`output[i] = input[0]+input[1]+…+input[i]`**

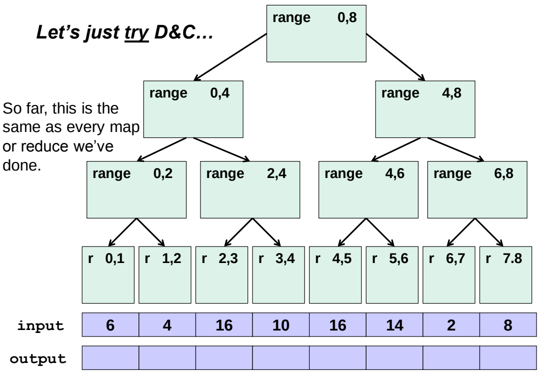
Sequential version is straightforward:

```
Vector<int> prefix_sum(const vector<int>& input){
  vector<int> output(input.size());
  output[0] = input[0];
  for(int i=1; i < input.size(); i++)
    output[i] = output[i-1]+input[i];
  return output;
}
```

Why isn't this (obviously) parallelizable? Isn't it just map or reduce?
Work:
Span:

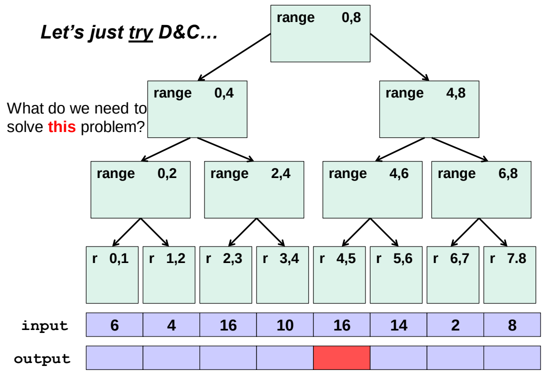
Let's just try D&C…

So far, this is the same as every map or reduce we've done.



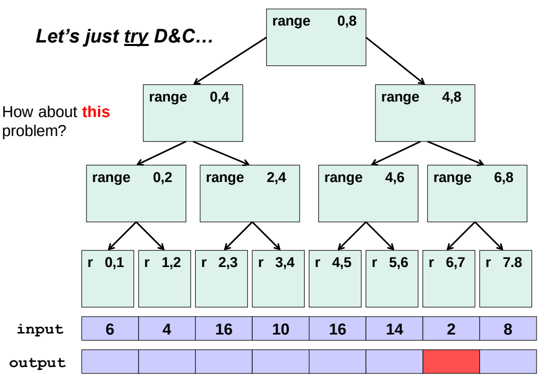
Let's just try D&C…

What do we need to solve **this** problem?



Let's just try D&C…

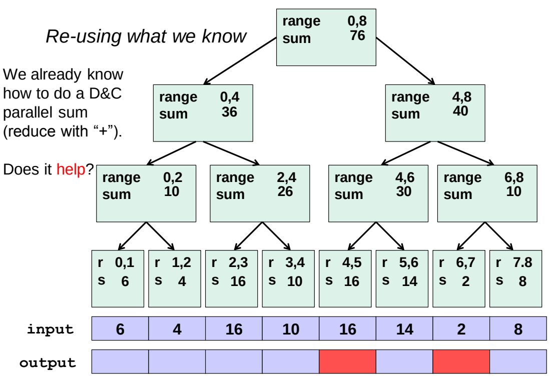
How about **this** problem?



Re-using what we know

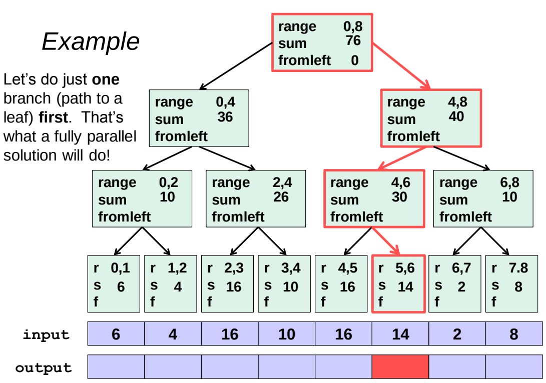
We already know how to do a D&C parallel sum (reduce with "+").

Does it help?



Example

Let's do just **one** branch (path to a leaf) **first**. That's what a fully parallel solution will do!



 Algorithm from [Ladner and Fischer, 1977]

Parallel prefix-sum

The parallel-prefix algorithm does two passes:

1. build a "sum" tree bottom-up
2. traverse the tree top-down, accumulating the sum from the left

The algorithm, step 1

1. Step one does a parallel sum to build a binary tree:
   – Root has sum of the range [**0,n**)
   – An internal node with the sum of [**lo,hi**) has
     • Left child with sum of [**lo,middle**)
     • Right child with sum of [**middle,hi**)
   – A leaf has sum of [**i,i+1**), i.e., **input[i]** (or an appropriate larger region w/a cutoff)

How? Parallel sum but explicitly build a tree:

```
return left+right;  ⇒  return new Node(left->sum + right->sum,
                                        left, right);
```

Step 1: Work? Span?

The algorithm, step 2

2. Parallel map, passing down a **fromLeft** parameter
   – Root gets a **fromLeft** of **0**
   – Internal nodes pass along:
     • to its left child the same **fromLeft**
     • to its right child **fromLeft** plus its left child's **sum**
   – At a leaf node for array position **i**, **output[i]=fromLeft+input[i]**

(already calculated in step 1!)

How? A map down the step 1 tree, leaving results in the output array.
Notice the *invariant*: **fromLeft** is the sum of elements left of the node's range

Step 2: Work? Span?

Parallel prefix-sum

The parallel-prefix algorithm does two passes:
1. build a "sum" tree bottom-up
2. traverse the tree top-down, accumulating the sum from the left

Step 1:	Work: ***O***(*n*)	Span: ***O***(**lg** *n*)
Step 2:	Work: ***O***(*n*)	Span: ***O***(**lg** *n*)
Overall:	Work?	Span?

Paralellism (work/span)?

In practice, of course, we'd use a sequential cutoff!

Parallelizing Quicksort

Recall quicksort was sequential, in-place, expected time *O*(*n* **lg** *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 do we parallelize this?
What span do we get?

$T_\infty(n) =$

Parallelizing Quicksort

Recall quicksort was sequential, in-place, expected time *O*(*n* **lg** *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?
Parallelize the recursive calls as we usually do in fork/join D&C.
Parallelize the partition by doing two packs (filters) instead.

Analyzing $T_\infty(n) = \mathtt{lg}\ n + T_\infty(n/2)$

Turns out our techniques from way back at the start of the term will work just fine for this:

$T_\infty(n)$	$= \mathtt{lg}\ n + T_\infty(n/2)$	if n > 1
	= 1	otherwise