departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Parallel Algorithms

Concurrency and Parallelism — 2018-19

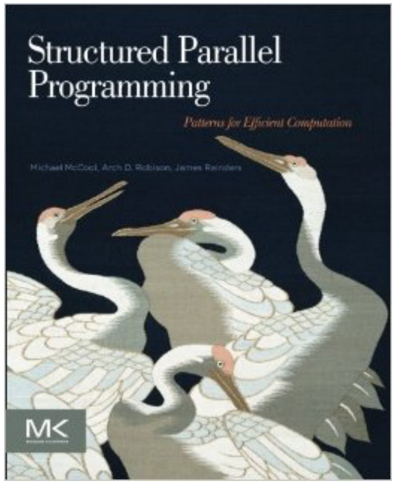
Master in Computer Science

(Mestrado Integrado em Eng. Informática)

Joao Lourenço <joao.lourenco@fct.unl.pt>

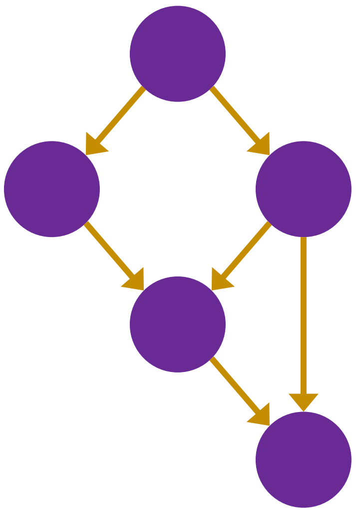
Outline

- Parallel computations as DAGs
  - Parallel computing by divide-and-conquer
  - Maps and reductions on tree-like DAGs
  - The Prefix-Sum (Scan) problem and its parallel solution
  - An implementation for the Pack parallel pattern
  - Bibliography:
    - **Chapter 3, 4 and 5** of book
      McCool M., Arch M., Reinders J.;
      Structured Parallel Programming: Patterns for Efficient Computation;
      Morgan Kaufmann (2012);
      ISBN: 978-0-12-415993-8



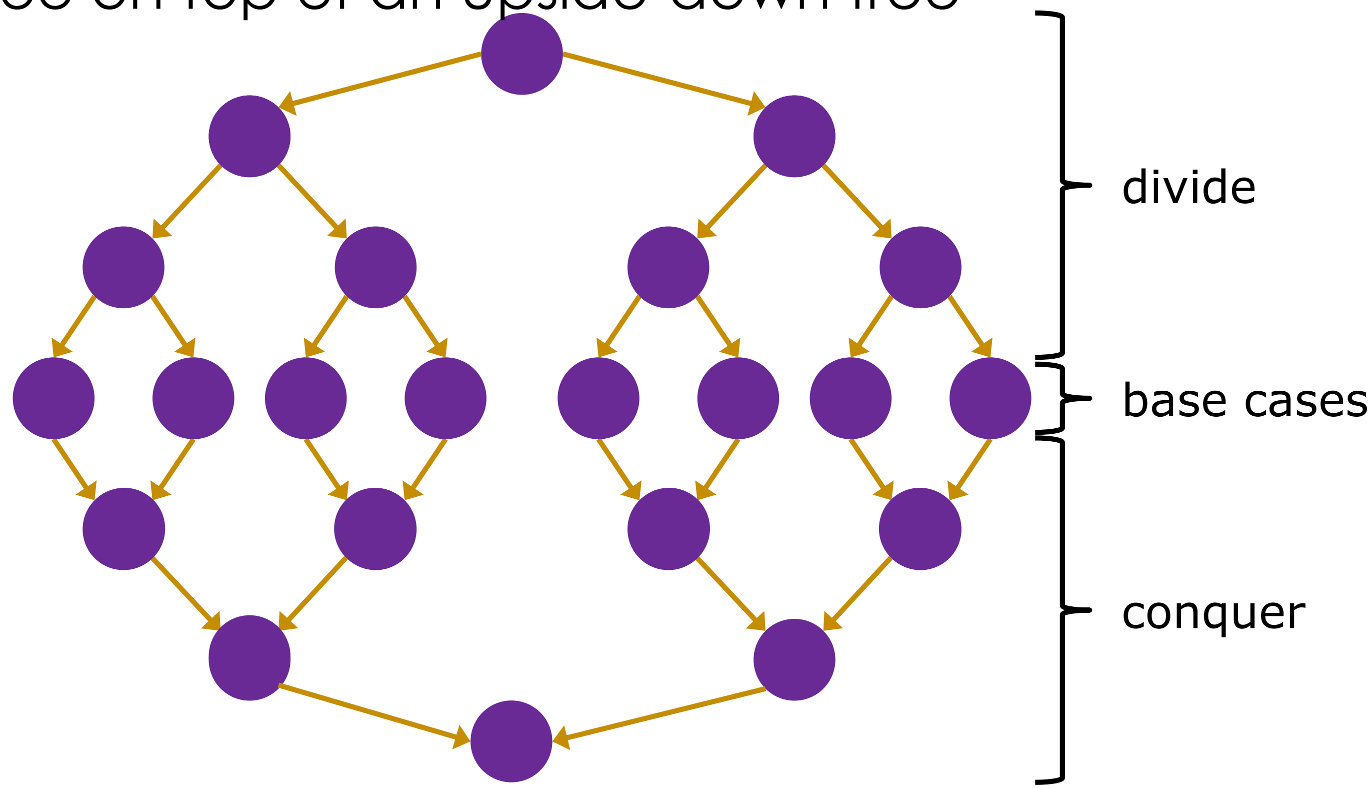
The DAG

- A program execution using fork and join can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts
- A fork "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A join "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on



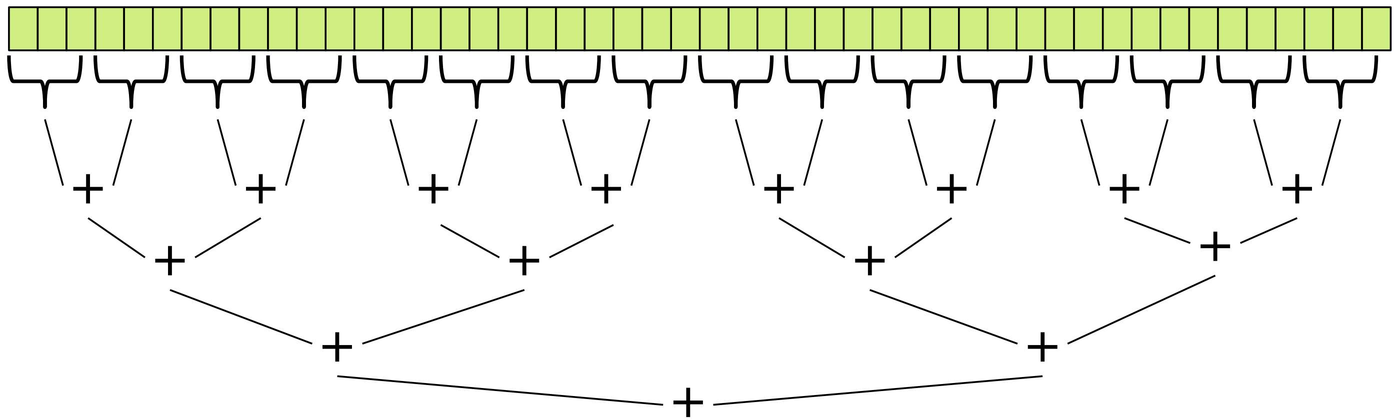
A simple example

- **fork** and **join** are very flexible, but divide-and-conquer use them in a very basic way:
  - A tree on top of an upside-down tree



Another example: reduce

- Summing an array went from O(n) sequential to O(log n) parallel (assuming a lot of processors and very large n)



- Anything that can use results from two halves and merge them in O(1) time has the same properties and exponential speed-up (in theory)

Applications of “reduce”

- Maximum or minimum element
- Is there an element satisfying some property?
  - e.g., is there a 17?
- Left-most element satisfying some property?
  - e.g., index of first 17
- Corners of a rectangle containing all points (a “bounding box”)
- Counts
  - e.g., # of strings that start with a vowel
  - This is just summing with a different base case

More Interesting DAGs?

- Of course, the DAGs are not always so simple (and neither are the related parallel problems)

- Example:
- Suppose combining two results might be expensive enough that we want to parallelize each one
- Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

Reductions

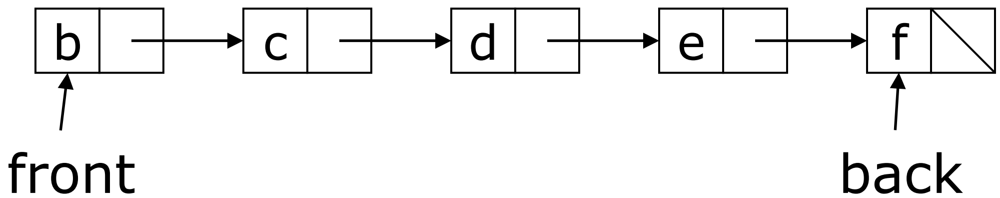
- Such computations of this simple form are common enough to have a name: reductions (or reduces?)
- Produce single answer from collection via an associative operator
  - Examples: max, count, leftmost, rightmost, sum, ...
  - Non-example: median
- Recursive results don't have to be single numbers or strings and can be arrays or objects with fields
  - Example: Histogram of test results
- But some things are inherently sequential
  - How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

Maps and Reductions on Trees

- Work just fine on balanced trees
  - Divide-and-conquer each child
  - Example:
    Finding the minimum element in an unsorted but balanced binary tree takes O(log n) time given enough processors
- How to do you implement the sequential cut-off?
  - Each node stores number-of-descendants (easy to maintain)
  - Or approximate it (e.g., AVL tree height)
- Parallelism also correct for unbalanced trees but obviously one gets worse speed-ups

Linked Lists

- Can you parallelize maps or reduces over linked lists?
  - Example: Increment all elements of a linked list
  - Example: Sum all elements of a linked list



- Nope. Once again, data structures matter!
- For parallelism, balanced trees are generally better than lists so that we can get to all the data exponentially faster $O(\log n)$ vs. $O(n)$
  - Trees have the same flexibility as lists compared to arrays (i.e., no shifting for insert or remove)

Division of Responsibility

- Parallel Framework users (e.g., Cilk+, Java ForkJoin)
  - Pick a good parallel algorithm and implement it
  - Its execution creates a DAG of things to do
  - Make all the nodes small(ish) and approximately equal amount of work
- The framework-writer's job:
  - Assign work to available processors to avoid idling
  - Keep constant factors low
  - Give the expected-time optimal guarantee assuming framework-user did his/her job
- $T_P = O((T_1 / P) + T_\infty)$

Examples: $T_P = O((T_1 / P) + T_\infty)$

- Sum an array
  - $T1 = O(n)$ and $T_\infty = O(\log n)$ => $T_P = O(n / P + \log n)$
- Suppose
  - $T1 = O(n^2)$ and $T_\infty = O(n)$ => $T_P = O(n^2 / P + n)$
- Of course, these expectations ignore any overhead or memory issues

The Prefix (Scan) Sum Problem

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

- A sequential solution in a typical 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;
}
```

The Prefix (Scan) Sum 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;
}
```

- Above algorithm does not seem to be parallelizable!
  - Work (T_1): $O(n)$ Span (T_∞): $O(n)$
- It isn't. The above algorithm is sequential.
- But a different algorithm gives a span of $O(\log n)$

Parallel Prefix-Sum

- The parallel-prefix algorithm does two passes
  - Each pass has O(n) work and O(log n) span
  - In total there is O(n) work and O(log n) span
  - Just like array summing, parallelism is O(n / log n)
  - An exponential speedup
- The first pass builds a tree bottom-up
- The second pass traverses the tree top-down

Historical note:
Original algorithm due to R. Ladner and M. Fischer at the UW in 1977

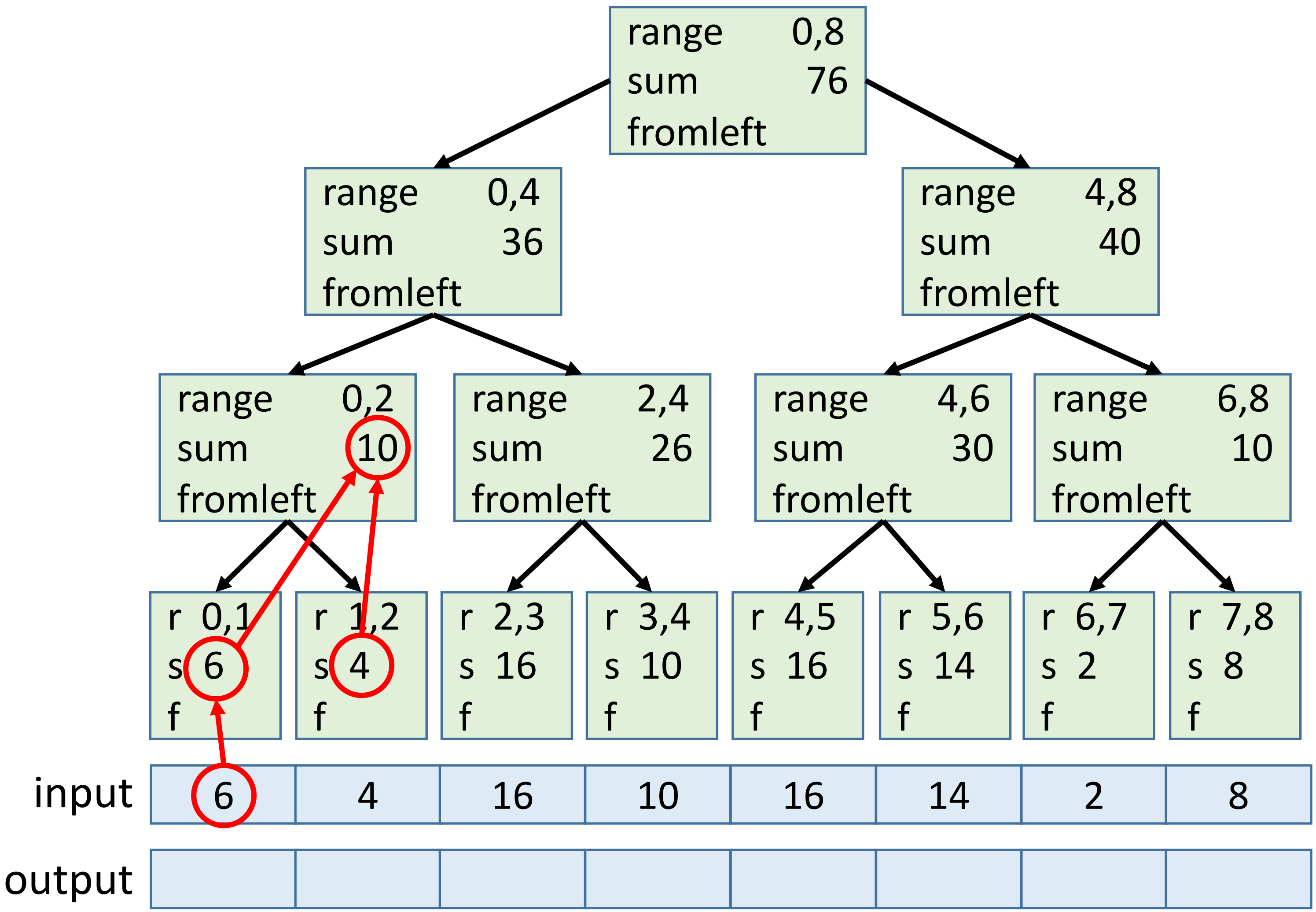
Parallel Prefix: The Up Pass

- We build want to 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], which is simply input[i]
- It is critical that we actually create the tree as we will need it for the down pass
  - We do not need an actual linked structure
  - We could use an array as we did with heaps

Parallel Prefix: The Up Pass

- This is an easy fork-join computation:
- buildRange(arr,lo,hi)
  - If lo+1 == hi, create new node with sum arr[lo]
  - Else, create two new threads:
    - buildRange(arr,lo,mid)
    - buildRange(arr,mid+1,high)
    - Where mid = (low+high)/2
  - When threads complete, make new node with
    - sum = left.sum + right.sum
- Performance Analysis:
  - Work: O(n)
  - Span: O(log n)

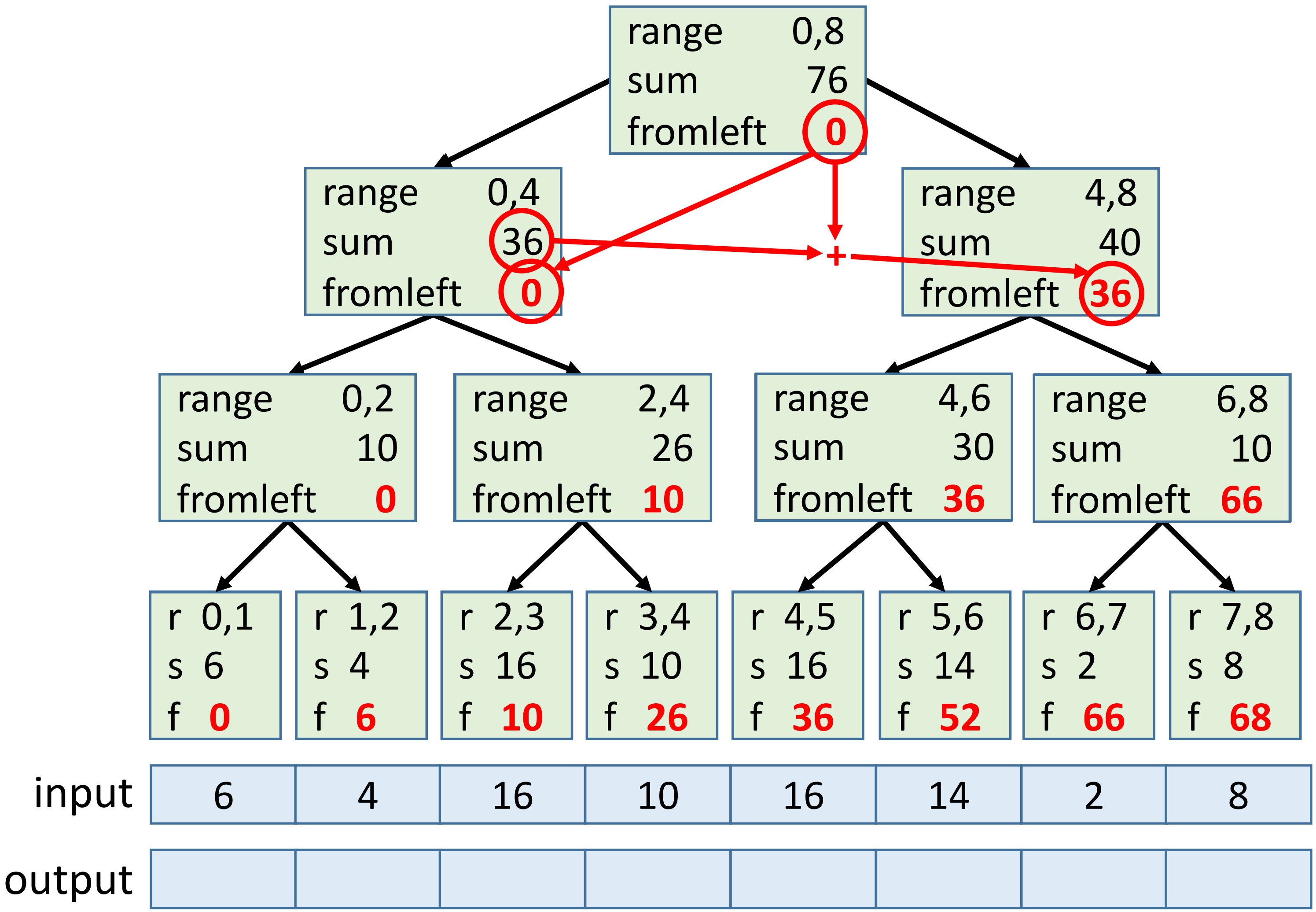
Up Pass Example



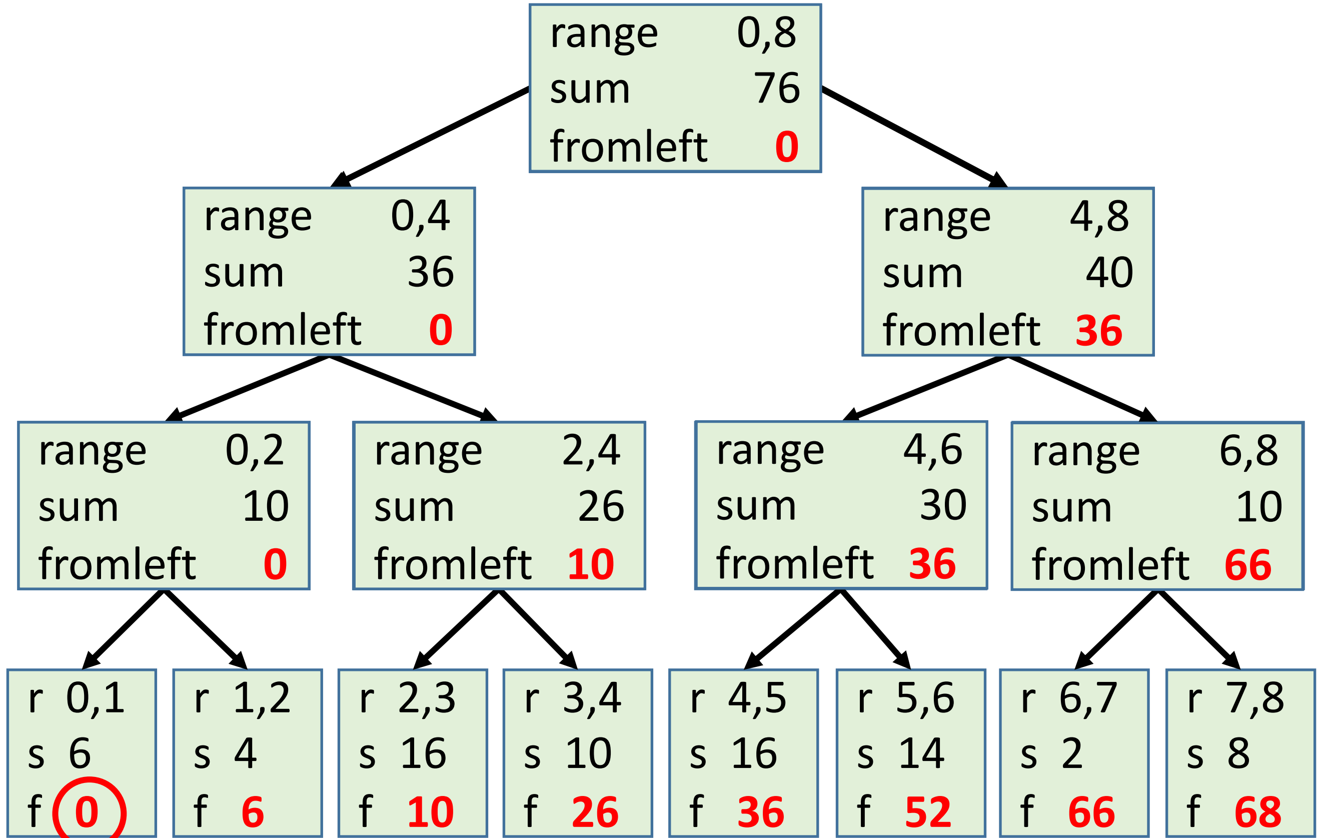
Parallel Prefix: The Down Pass

- We now use the tree to get the prefix sums using an easy fork-join computation
- Starting at the root:
  - Root is given a fromLeft of 0
  - Each node takes its fromLeft value and:
    - Passes to the left child: fromLeft
    - Passes to the right child: fromLeft + left.sum
  - At leaf for position i, output[i]=fromLeft+input[i]
- Invariant: **fromLeft is sum of elements left of the node's range**

Down Pass Example



Down Pass Example



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

Parallel Prefix: The Down Pass

- Note that this parallel algorithm does not return any values
  - Leaves assign to output array
  - **This is a map, not a reduction**

- Performance Analysis:
  - Work: O(n)
  - Span: O(log n)

Generalizing Parallel Prefix

- Prefix-sum illustrates a pattern that can be used 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!
- That last one is perfect for an efficient parallel pack that builds on top of the "parallel prefix trick"

Pack (Think Filtering)

- Given an array `input` and boolean function `f(e)` produce an array `output` containing only elements `e` such that `f(e)` is `true`
- Example:
  input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
  f(e): is e > 10?
  output [17, 11, 13, 19, 24]
- Is this parallelizable? Of course!
  - Finding elements for the output is easy
  - But getting them in the right place seems hard

Pack: Parallel Map + Parallel Prefix + Parallel Map

1. Use a 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

```
bitsum   [ 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];
}
```

Pack Comments

- First two steps can be combined into one pass
  - Will require changing base case for the prefix sum
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity
- Analysis: Work: O(n); Span: O(log n)
  - Multiple passes, but this is a constant

The END

- Sources:
  - Parallel Computing, CIS 410/510, Department of Computer and Information Science
  - https://courses.cs.washington.edu/courses/cse332/12su/slides/lecture12-parallelism-work-span.pdf