CSE 332 Data Abstractions: Introduction to Parallelism and Concurrency

Kate Deibel

Summer 2012

Where We Are

Last time, we introduced fork-join parallelism

- Separate programming threads running at the same time due to presence of multiple cores
- Threads can fork off into other threads
- Said threads share memory
- Threads join back together

We then discussed two ways of implementing such parallelism in Java:

- The Java Thread Library
- The ForkJoin Framework

Ah yes… the comfort of mathematics…

ENOUGH IMPLEMENTATION: ANALYZING PARALLEL CODE

Key Concepts: Work and Span

Analyzing parallel algorithms requires considering the full range of processors available

- We parameterize this by letting $\mathbf{T_P}$ be the running time if **P** processors are available
- We then calculate two extremes: work and span

Work: $\mathbf{T_1}$ → How long using only 1 processor

- Just "sequentialize" the recursive forking

Span: $\mathbf{T_\infty}$ → How long using infinity processors

- The longest dependence-chain
- Example: $O(\log n)$ for summing an array
  - Notice that having > $n/2$ processors is no additional help (a processor adds 2 items, so only n/2 needed)
- Also called "critical path length" or "computational depth"

The DAG

A program execution using **`fork`** and **`join`** can be seen as a DAG

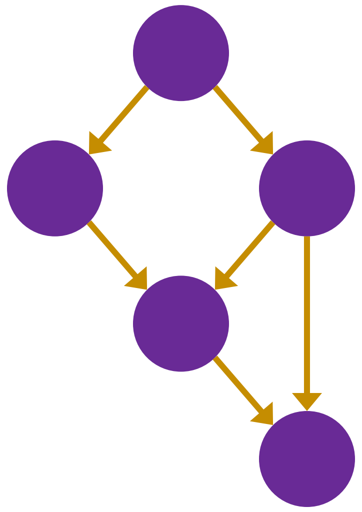
- Nodes: Pieces of work
- Edges: Source must finish before destination starts

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

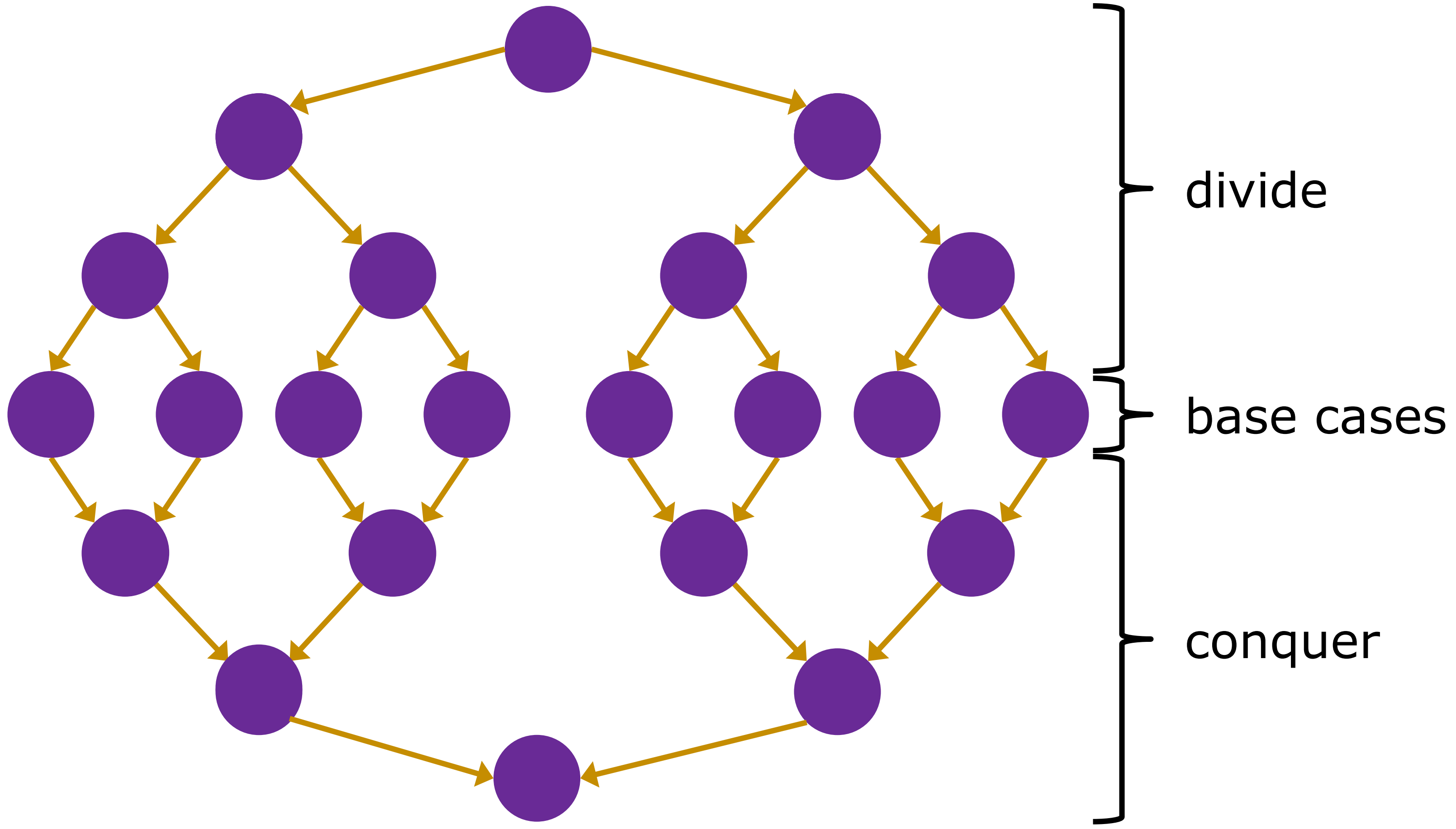
- Node just ended
- Last node of thread joined on



Our Simple Examples

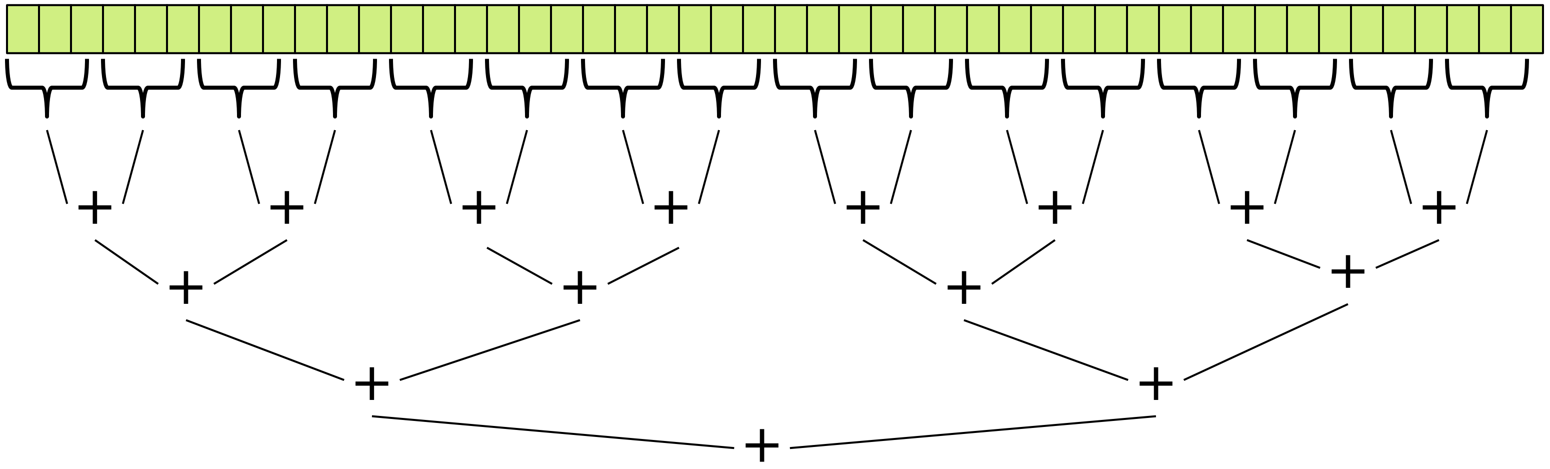
`fork` and **`join`** are very flexible, but divide-and-conquer use them in a very basic way:

- A tree on top of an upside-down tree



What Else Looks Like This?

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)

Examples

- 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., first 17)
  - What should the recursive tasks return?
  - How should we merge the results?
- 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 Data Parallelism

A map operates on each element of a collection independently to create a new collection of the same size

- No combining results
- For arrays, this is so trivial some hardware has direct support (often in graphics cards)

Canonical example: Vector addition

```
int[] vector_add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  result = new int[arr1.length];
  FORALL(i=0; i < arr1.length; i++) {
    result[i] = arr1[i] + arr2[i];
  }
  return result;
}
```

Maps in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
  int lo; int hi; int[] res; int[] arr1; int[] arr2;
  VecAdd(int l,int h,int[] r,int[] a1,int[] a2){ … }
  protected void compute(){
    if(hi – lo < SEQUENTIAL_CUTOFF) {
       for(int i=lo; i < hi; i++)
         res[i] = arr1[i] + arr2[i];
    } else {
      int mid = (hi+lo)/2;
      VecAdd left = new VecAdd(lo,mid,res,arr1,arr2);
      VecAdd right= new VecAdd(mid,hi,res,arr1,arr2);
      left.fork();
      right.compute();
      left.join();
    }
  }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2){
  assert (arr1.length == arr2.length);
  int[] ans = new int[arr1.length];
  fjPool.invoke(new VecAdd(0,arr.length,ans,arr1,arr2);
  return ans;
}
```

Maps and Reductions

Maps and reductions are the "workhorses" of parallel programming

- By far the two most important and common patterns
- We will discuss two more advanced patterns later

We often use maps and reductions to describe parallel algorithms

- We will aim to learn to recognize when an algorithm can be written in terms of maps and reductions
- Programming them then becomes "trivial" with a little practice (like how for-loops are second-nature to you)

Digression: MapReduce on Clusters

You may have heard of Google’s "map/reduce"

- Or the open-source version Hadoop

Perform maps/reduces on data using many machines

- The system takes care of distributing the data and managing fault tolerance
- You just write code to map one element and reduce elements to a combined result

Separates how to do recursive divide-and-conquer from what computation to perform

- Old idea in higher-order functional programming transferred to large-scale distributed computing
- Complementary approach to database declarative queries

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?

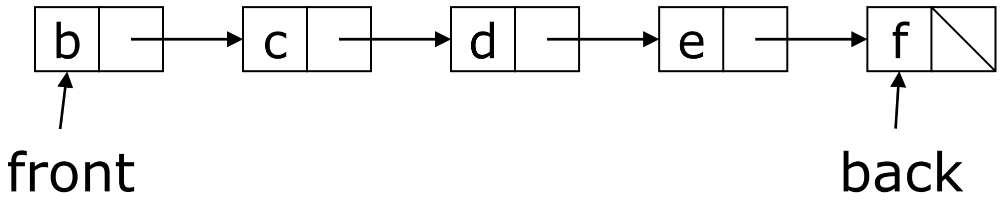
- Each node stores number-of-descendants (easy to maintain)
- Or approximate it (e.g., AVL tree height)

Parallelism also correct for unbalanced trees but you obviously do not get much speed-up

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 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)

Analyzing algorithms

Like all algorithms, parallel algorithms should be:

- Correct
- Efficient

For our algorithms so far, their correctness is "obvious" so we’ll focus on efficiency

- Want asymptotic bounds
- Want to analyze the algorithm without regard to a specific number of processors
- The key "magic" of the ForkJoin Framework is getting expected run-time performance asymptotically optimal for the available number of processors
  - Ergo we analyze algorithms assuming this guarantee

Connecting to Performance

Recall: $\mathbf{T_P}$ = run time if **P** processors are available

We can also think of this in terms of the program's DAG

Work = $\mathbf{T_1}$ = sum of run-time of all nodes in the DAG

- Note: costs are on the nodes not the edges
- That lonely processor does everything
- Any topological sort is a legal execution
- $O(n)$ for simple maps and reductions

Span = $\mathbf{T_\infty}$ = run-time of most-expensive path in DAG

- Note: costs are on the nodes not the edges
- Our infinite army can do everything that is ready to be done but still has to wait for earlier results
- $O(\log n)$ for simple maps and reductions

Some More Terms

Speed-up on **P** processors: $\mathbf{T_1 / T_P}$

Perfect linear speed-up: If speed-up is **P** as we vary **P**

- Means we get full benefit for each additional processor: as in doubling **P** halves running time
- Usually our goal
- Hard to get (sometimes impossible) in practice

Parallelism is the maximum possible speed-up: $\mathbf{T_1/T_\infty}$

- At some point, adding processors won’t help
- What that point is depends on the span

Parallel algorithms is about decreasing span without increasing work too much

Optimal T_P: Thanks ForkJoin library

So we know $\mathbf{T_1}$ and $\mathbf{T_\infty}$ but we want $\mathbf{T_P}$ (e.g., **P**=4)

Ignoring memory-hierarchy issues (caching), $\mathbf{T_P}$ cannot

- Less than $\mathbf{T_1 / P}$ why not?
- Less than $\mathbf{T_\infty}$ why not?

So an *asymptotically* optimal execution would be:

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

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

The ForkJoin Framework gives an *expected-time guarantee* of asymptotically optimal!

- Expected time because it flips coins when *scheduling*
- How? For an advanced course (few need to know)
- Guarantee requires a few assumptions about your code…

Division of Responsibility

Our job as ForkJoin Framework users:

- 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

$$\mathbf{T_P = \mathit{O}((T_1 / P) + T_\infty)}$$

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

Algorithms seen so far (e.g., sum an array):

If $\mathbf{T_1} = O(n)$ and $\mathbf{T_\infty} = O(\log n)$

→ $\mathbf{T_P} = O(n/\mathbf{P} + \log n)$

Suppose instead:

If $\mathbf{T_1} = O(n^2)$ and $\mathbf{T_\infty} = O(n)$

→ $\mathbf{T_P} = O(n^2/\mathbf{P} + n)$

Of course, these expectations ignore any overhead or memory issues

Things are going so smoothly…

Parallelism is awesome…

Hello stranger, what's your name?

Murphy? Oh @!♪%★$☹*!!!

AMDAHL'S LAW

Amdahl's Law (mostly bad news)

In practice, much of our programming typically has parts that parallelize well

- Maps/reductions over arrays and trees

And also parts that don't parallelize at all

- Reading a linked list
- Getting/loading input
- Doing computations based on previous step

To understand the implications, consider this:

"Nine women cannot make a baby in one month"

Amdahl's Law (mostly bad news)

Let ***work*** (time to run on 1 processor) be 1 unit time

If **S** is the portion of execution that cannot be parallelized, then we can define T_1 as:

$$\mathbf{T_1 = S + (1\text{-}S) = 1}$$

If we get perfect linear speedup on *the parallel portion*, then we can define T_P as:

$$\mathbf{T_P = S + (1\text{-}S)/P}$$

Thus, the overall speedup with **P** processors is (Amdahl's Law):

$$\mathbf{T_1 / T_P = 1 / (S + (1\text{-}S)/P)}$$

And the parallelism (infinite processors) is:

$$\mathbf{T_1 / T_\infty = 1 / S}$$

Why this is such bad news

Amdahl's Law: $T_1 / T_P = 1 / (S + (1-S)/P)$

$T_1 / T_\infty = 1 / S$

Suppose 33% of a program is sequential

- Then a billion processors won't give a speedup over 3

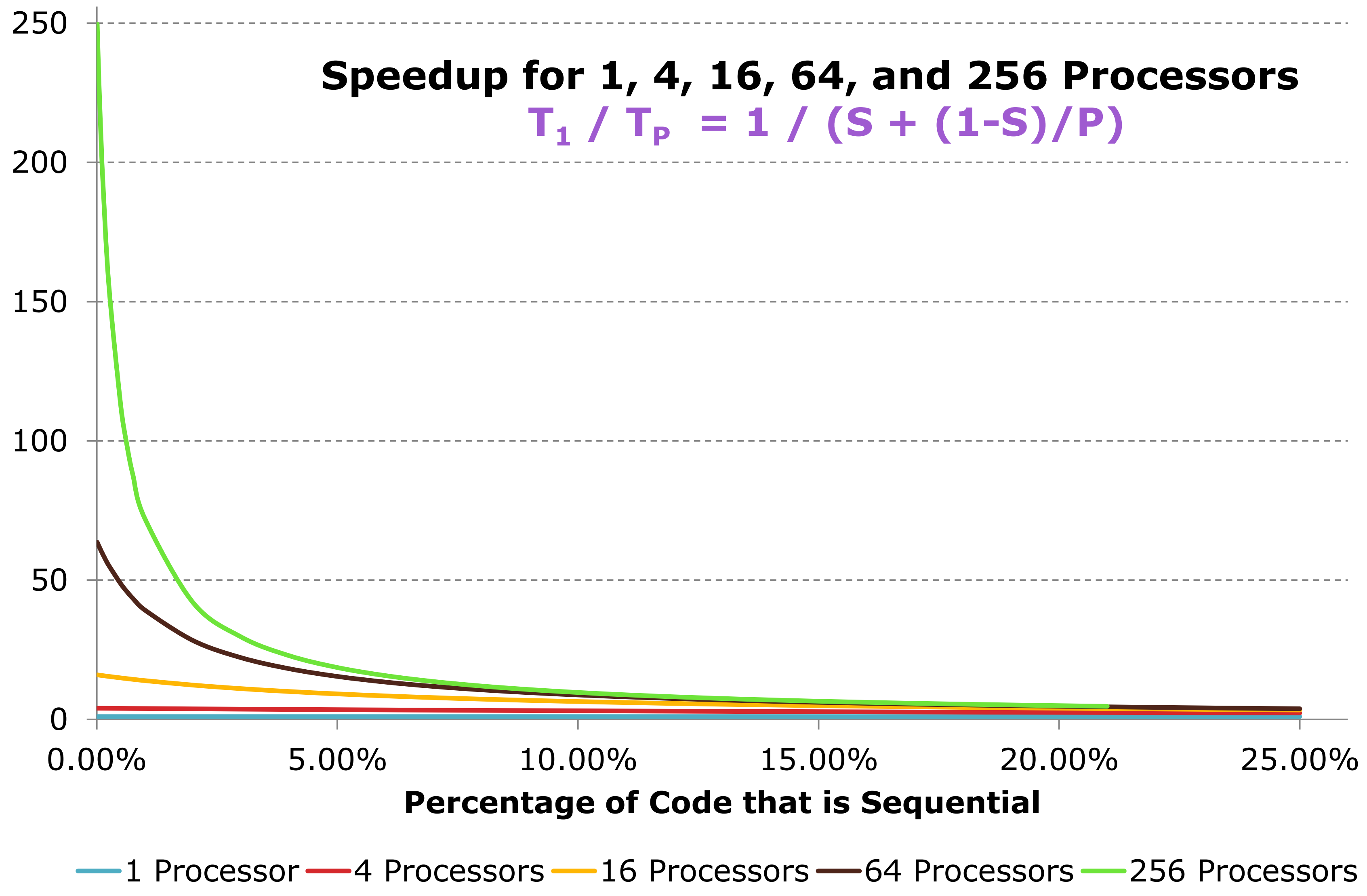
Suppose you miss the good old days (1980-2005) where 12 years or so was long enough to get 100x speedup

- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of just 1
- For the 256 cores to gain ≥100x speedup, we need

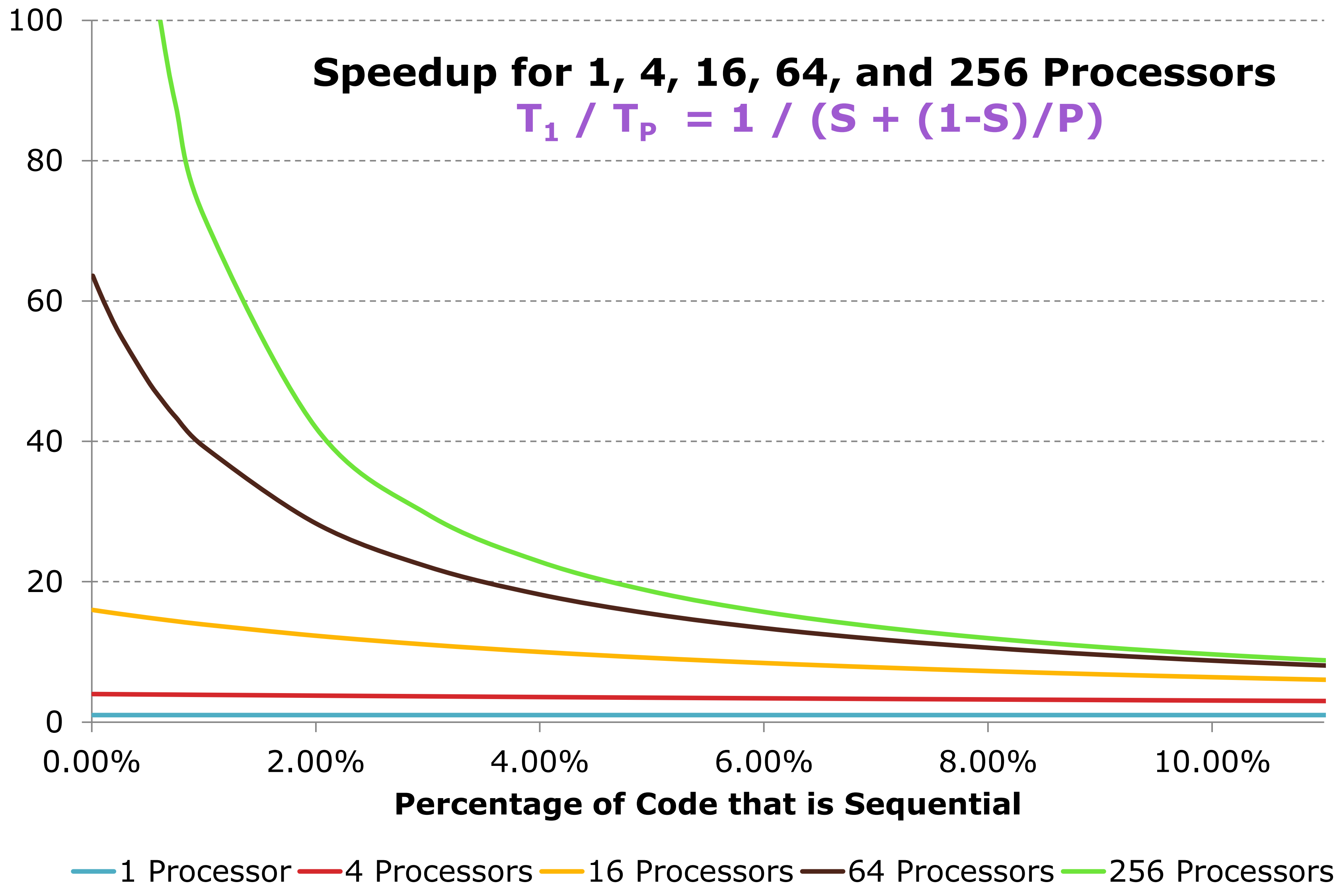
  $$100 \leq 1 / (\mathbf{S} + (1-\mathbf{S})/256)$$

  Which means **S** ≤ .0061 or 99.4% of the algorithm must be perfectly parallelizable!!

A Plot You Have To See



A Plot You Have To See (Zoomed In)



All is not lost

Amdahl's Law is a bummer!

- Doesn't mean additional processors are worthless!!

We can always search for new parallel algorithms

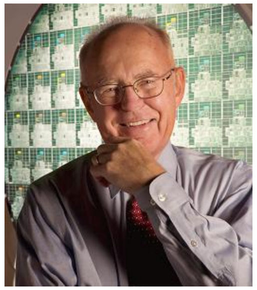
- We will see that some tasks may seem inherently sequential but can be parallelized

We can also change the problems we're trying to solve or pursue new problems

- Example: Video games/CGI use parallelism
  - But not for rendering 10-year-old graphics faster
  - They are rendering more beautiful(?) monsters

A Final Word on Moore and Amdahl

Although we call both of their work laws, they are very different entities



Moore's "Law" is an *observation* about the progress of the semiconductor industry:

- Transistor density doubles every ≈18 months

Amdahl's Law is a mathematical theorem

- Diminishing returns of adding more processors

Very different but incredibly important in the design of computer systems

If we were really clever, we wouldn't constantly say parallel because after all we are discussing parallelism so it should be rather obvious but this comment is getting too long and stopped being clever ages ago…

BEING CLEVER: PARALLEL PREFIX

Moving Forward

Done:

- "Simple" parallelism for counting, summing, finding
- Analysis of running time and implications of Amdahl’s Law

Coming up:

- Clever ways to parallelize more than is intuitively possible
- Parallel prefix:
  - A "key trick" typically underlying surprising parallelization
  - Enables other things like packs
- Parallel sorting: mergesort and quicksort (not in-place)
  - Easy to get a little parallelism
  - With cleverness can get a lot of parallelism

The Prefix-Sum Problem

Given `int[] input`, produce `int[] output` such that:

`output[i]=input[0]+input[1]+…+input[i]`

A sequential solution is a typical CS1 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-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: $O(n)$
- Span: $O(n)$

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 $O(\texttt{log}\ n)$ span
- In total there is $O(n)$ work and $O(\texttt{log}\ n)$ span
- Just like array summing, parallelism is n / $\texttt{log}\ n$
- 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:

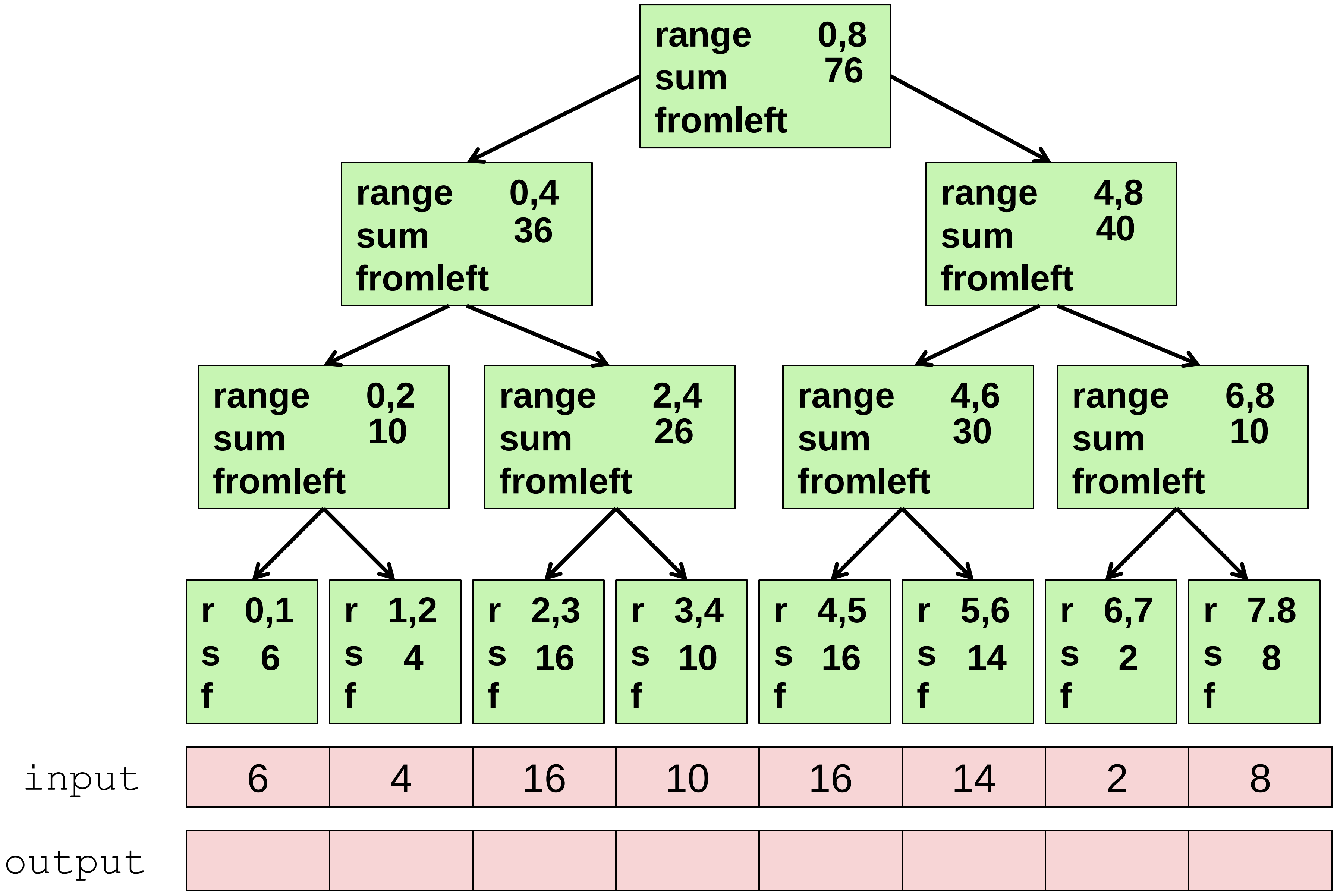
buildRange(arr,lo,hi)

- If lo+1 == hi, create new node with sum arr[lo]
- Else, create two new threads:
  buildRange(arr,lo,mid) and
  buildRange(arr,mid+1,high)
  where mid = (low+high)/2
  and 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

Parallel Prefix: The Down Pass

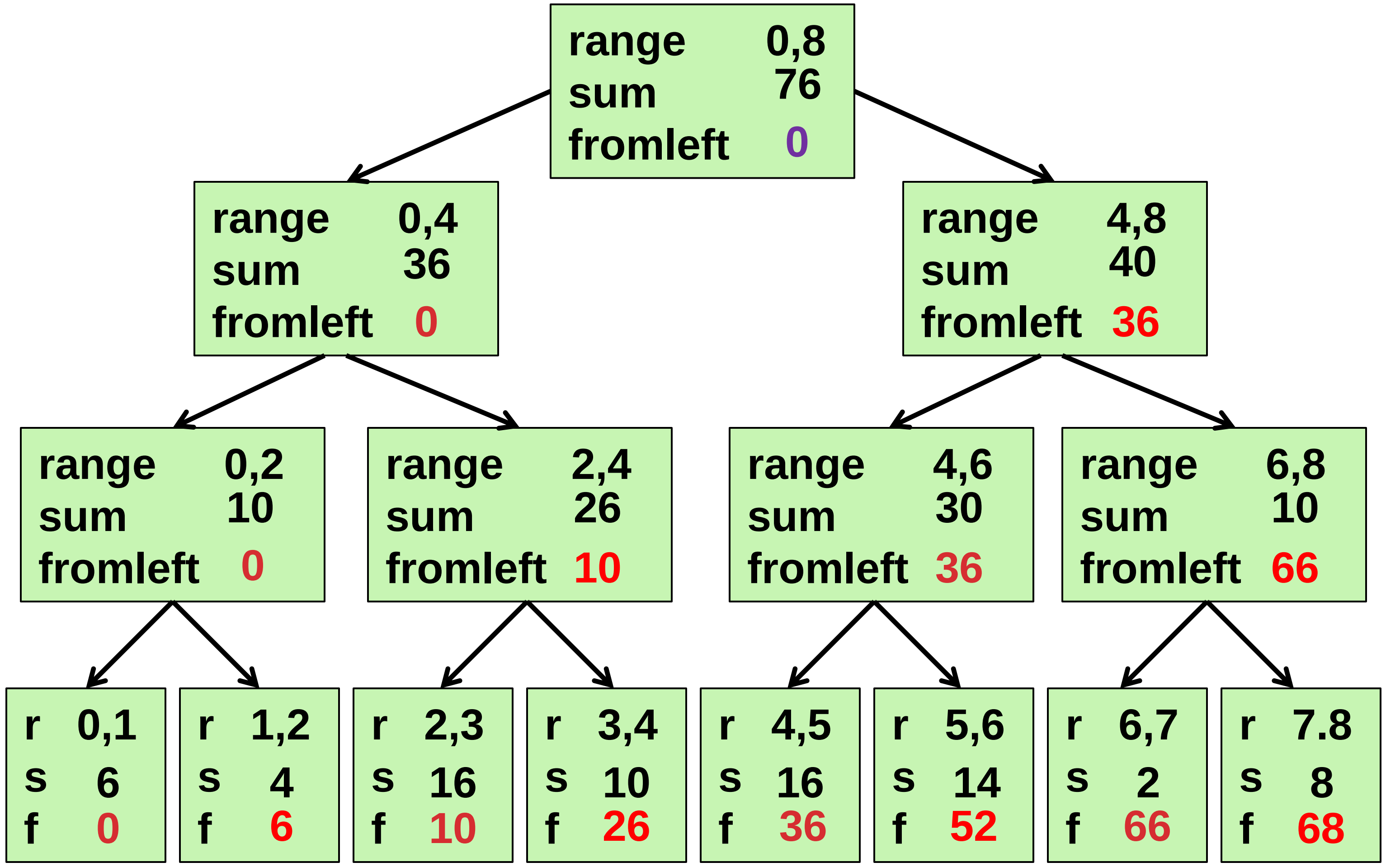
Note that this parallel algorithm does not return any values

- Leaves assign to **`output`** array
- This is a map, not a reduction

Performance Analysis:

- Work: O(n)
- Span: O(**`log`** n)

Down Pass Example



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

Sequential Cut-Off

Adding a sequential cut-off is easy as always:

- Up Pass:
  Have leaf node hold the sum of a range instead of just one array value

- Down Pass:

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

Generalizing Parallel Prefix

Just as sum-array was the simplest example of a common pattern, 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

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

```
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];
}
```

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: $O(n)$ work, $O(\texttt{log}\ n)$ span

- Multiple passes, but this is a constant

Parallelized packs will help us parallelize sorting

Welcome to the Parallel World

We will continue to explore this topic and its implications

In fact, the next class will consist of 16 lectures presented simultaneously

- I promise there are no concurrency issues with your brain
- It is up to you to parallelize your brain before then

The interpreters and captioner should attempt to grow more limbs as well