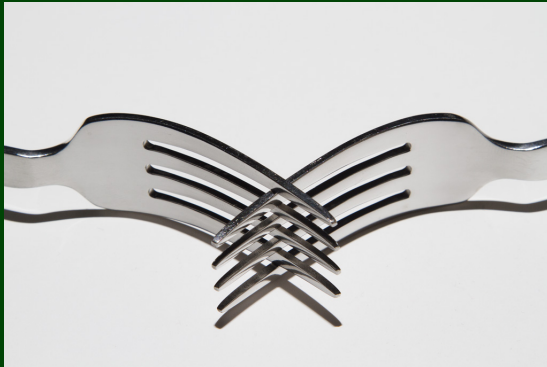


# CSE 332

## Data Abstractions

# Analysis of Parallel Programs



# Outline

- 1 Parallel Primitives
- 2 Paralellism with Other Data Structures
- 3 Analyzing Parallel Algorithms

## Largest Factors

Last time, we found the number of primes in a range. This time, let's actually find the primes.

```
1 protected void compute() {
2     if (hi - lo <= CUTOFF) {
3         seqReplaceWithLargestFactor(arr, lo, hi);
4         return;
5     }
6
7     int mid = lo + (hi - lo) / 2;
8     PrimeFinderTask left = new PrimeFinderTask(arr, lo, mid);
9     PrimeFinderTask right = new PrimeFinderTask(arr, mid, hi);
10
11     left.fork();
12     right.compute();
13     left.join();
14 }
```

This problem was different than the previous ones. The goal was to apply a function to every element of an array rather than to return a result.

## Reductions

Last time, we saw several problems of the form:

**INPUT:** An array

**OUTPUT:** A combination of the array by an associative operation

The general name for this type of problem is a **reduction**. Examples include: `max`, `min`, `has-a`, `first`, `count`, `sorted`

## Maps

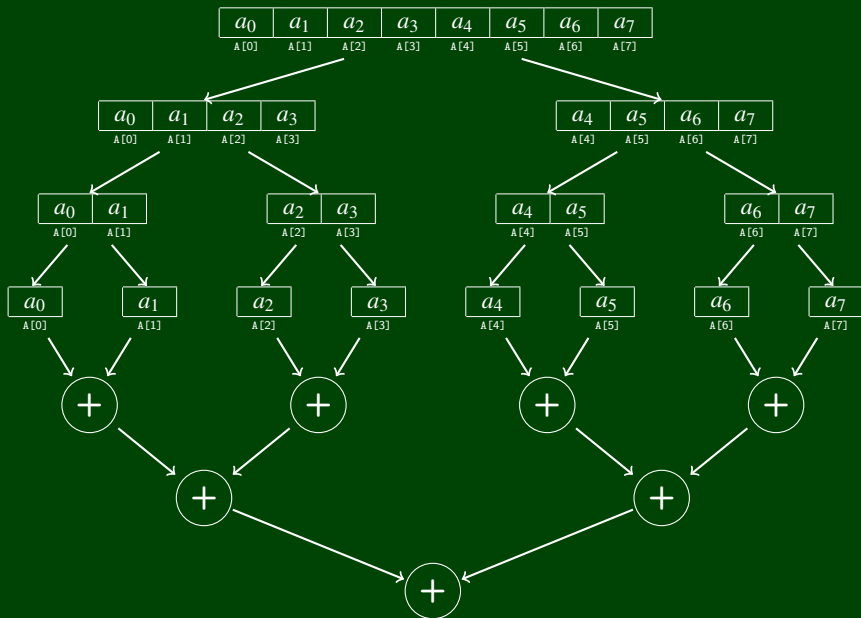
We just saw a problem of the form:

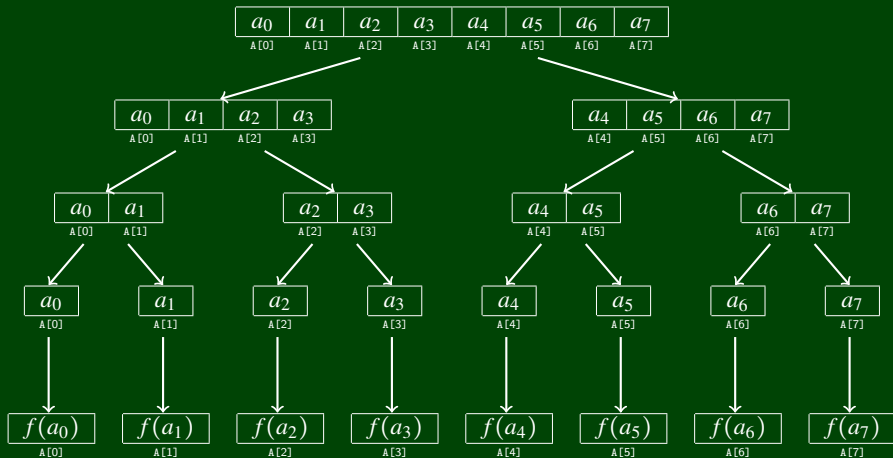
**INPUT:** An array

**OUTPUT:** Apply a function to every element of that array

The general name for this type of problem is a **map**. You can do this with any function, because the array elements are independent.

These two types of problems are “**parallel primitives**” in the same way loops and if statements are “programming primitives”. Next lecture, we’ll add two more primitives.





You may have heard of Google's MapReduce (or the open-source version Hadoop).

- Idea: 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 declarative queries for databases



So far, we've only tried to apply parallelism to an `Array` (or, equivalently, an `ArrayList`). What about the other data structures we know? In particular, how does `ForkJoin` do on:

- `LinkedLists`?
- `BinaryTrees`?
- (Balanced) `BinaryTrees`?
- *n*-ary `Trees`?

Let's think about this with our toy problem of “sum up all the elements of the input”.

We wrote code that treated the array like a `LinkedList` last lecture.

```
1 compute() {
2     if (not the end of the list) {
3         fork a thread to do the rest of the elements;
4     }
5
6     do my work
7
8     join with the thread after me
9 }
```

The only gain we're going to get with `LinkedLists` is if the **map function** is very expensive. Then we'll at least get most of those going at once.

Naturally, as with standard algorithms on unbalanced trees, since they degenerate to linked lists, we have the same problem.

The idea here is to divide-and-conquer **each child** instead of array sub-ranges:

```
1 compute() {
2     left.fork();    // Handles the entire left subtree
3     right.compute(); // Handles the entire right subtree
4
5     return left.join() + rightResult;
6 }
```

But what about the sequential cut-off?

Either store the **number of nodes** in each subtree or approximate it with **the height**

Consider the **MAXIMUM** problem from a few lectures ago. The best we could do in sequential-land was  $\Omega(n)$ , but with parallelism, we can find the maximum element in  $\Theta(\lg n)$  time (with enough processors... )!

With sequential algorithms, we often considered  $T(n)$  (the runtime of the algorithm). Now, we'll consider a more general notion:

Let  $T_P(n)$  be the runtime of an algorithm **using  $P$  processors**.

There are two important runtime quantities for a parallel algorithm:

- How long it would take if it were fully sequential (work)
- How long it would take if it were as parallel as possible (span)

## Definition (Work)

We say work  $(n) = T_1(n) = T(n)$  is the cumulative work that **all** processors must complete.

## Definition (Span)

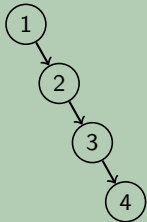
We say span  $(n) = T_\infty(n)$  is the largest amount of work **some** processor must complete.

For each “type” of tree, figure out  $\text{work}(-)$  and  $\text{span}(-)$  of  $\text{findMax}$  in terms of the number of nodes,  $n$ .

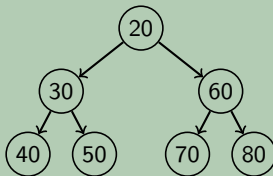
## A (Parallel) Algorithm

```
1 int findMax(Node current) {  
2     if (current is a leaf) {  
3         return current.data;  
4     }  
5  
6     return max(findMax(left), findMax(right));  
7 }
```

### Degenerate Tree



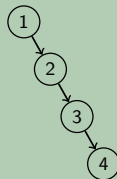
### Perfect Tree



## A (Parallel) Algorithm

```
1 int findMax(Node current) {  
2     if (current is a leaf) {  
3         return current.data;  
4     }  
5  
6     return max(findMax(left), findMax(right));  
7 }
```

## Degenerate Tree



To calculate work, we just do our standard analysis. First, we make a recurrence:

$$\text{work}(n) = \begin{cases} 0 & \text{if } n = 0 \\ \mathcal{O}(1) & \text{if } n = 1 \\ \text{work}(0) + \text{work}(n-1) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

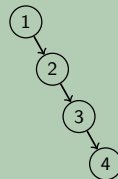
Solving this recurrence gives us:

$$\text{work}(n) = \sum_{i=0}^n 1 = \Theta(n)$$

## A (Parallel) Algorithm

```
1 int findMax(Node current) {  
2     if (current is a leaf) {  
3         return current.data;  
4     }  
5  
6     return max(findMax(left), findMax(right));  
7 }
```

## Degenerate Tree



To calculate span, we assume all calls are in parallel. We look for the **longest dependence chain**. We make a recurrence:

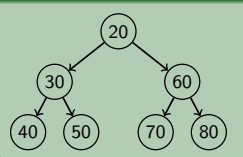
$$\text{span}(n) = \begin{cases} 0 & \text{if } n = 0 \\ \mathcal{O}(1) & \text{if } n = 1 \\ \max(\text{span}(0), \text{span}(n-1)) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

This ends up being the same recurrence as for  $\text{work}(-)$ . Notice for the degenerate tree  $\text{work}(n) = \text{span}(n)$ . This proves our intuition that we don't get much of a (any!) speed-up with parallelism for linked lists!

## A (Parallel) Algorithm

```
1 int findMax(Node current) {  
2     if (current is a leaf) {  
3         return current.data;  
4     }  
5  
6     return max(findMax(left), findMax(right));  
7 }
```

## Perfect Tree



To calculate work, we just do our standard analysis. First, we make a recurrence:

$$\text{work}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ 2 \times \text{work}(n/2) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

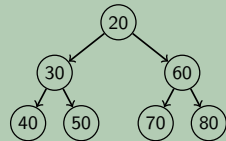
Master Theorem says this recurrence is  $\Theta(n)$ .



## A (Parallel) Algorithm

```
1 int findMax(Node current) {  
2     if (current is a leaf) {  
3         return current.data;  
4     }  
5  
6     return max(findMax(left), findMax(right));  
7 }
```

## Perfect Tree



To calculate span, we just do our standard analysis. First, we make a recurrence:

$$\text{span}(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ \max(\text{span}(n/2), \text{span}(n/2)) + \mathcal{O}(1) & \text{otherwise} \end{cases}$$

Master Theorem says this recurrence is  $\Theta(\lg n)$ .

Again, this proves our intuition that parallelizing tree algorithms helps.

**But what does it mean for work to be  $\Theta(n)$  and span to be  $\Theta(\lg n)$ ?**

## Definition (Speed-Up)

The **speed-up** given  $P$  processors is  $\frac{T_1}{T_P}$ .

If the speed-up is  $P$  as we vary  $P$ , it's called a **perfect linear speed-up**.

## Definition (Parallelism)

**Parallelism** is the **maximum possible speed-up**. In other words, parallelism is the speed-up when we take  $P = \infty$ .

**We want to decrease span without increasing work!**

Consider  $T_P$ . We know the following:

- $T_P \geq \frac{T_1}{P}$ , because that would only happen when all processors are always busy.
- $T_P \geq T_\infty$ , because  $T_\infty$  is the length of the critical path which the algorithm must go through.

So, **asymptotically**, we know:

$$T_P \in \mathcal{O}\left(\frac{T_1}{P} + T_\infty\right)$$

## The Good News!

The ForkJoin Framework gives an expected-time guarantee of asymptotically optimal! (Want to know how? Take an advanced course!)  
But this is only true given some assumptions about your code:

- The program splits up the work into small and approximately equal pieces
- The program combines the pieces efficiently

## Maximum in a Perfect Tree

When calculating the maximum element in a tree, we had:

- $\text{work}(n) \in \Theta(n)$
- $\text{span}(n) \in \Theta(\lg n)$

So, we expect the algorithm to take  $\mathcal{O}\left(\frac{n}{P} + \lg n\right)$

## Another Example

Suppose we have the following work and span:

- $\text{work}(n) \in \Theta(n^2)$
- $\text{span}(n) \in \Theta(n)$

So, we expect the algorithm to take  $\mathcal{O}\left(\frac{n^2}{P} + n\right)$

Every program has:

- parts that parallelize easily/well
- parts that don't parallelize at all

For example, we **can't** parallelize reading a linked list.

**The non-parallelizable parts of a program are a huge bottleneck**

Split the work up into two pieces: the “parallelizable” piece and the “non-parallelizable” piece. Let  $S$  be the inherently sequential work.

$$T_1 = S \times \text{work}(n) + (1 - S) \times \text{work}(n)$$

Suppose we get a perfect linear speed-up on the parallelizable work:

$$T_P = S \times \text{work}(n) + \frac{(1 - S) \times \text{work}(n)}{P}$$

So, the speed-up is:

$$\frac{T_1}{T_P} = \frac{1}{S + \frac{1-S}{P}}$$

### The Bad News

Suppose 33% of a program is sequential. Then, the **absolute best speed-up** we can get is:

$$\frac{T_1}{T_\infty} = \frac{1}{0.33} = 3$$

**That means infinitely many processors won't help us get more than a 3 times speed-up!**

Amdahl tells us that if a **particular algorithm** has too many sequential computations, it's better to find a **more parallelizable** algorithm than to just add more processors.

We'll see next time that unexpected problems can be solved in parallel!

## 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**