Outline

1. Parallel Primitives
2. Parallelism with Other Data Structures
3. Analyzing Parallel Algorithms

More Parallel Primes-ish

Largest Factors
Last time, we found the number of primes in a range. This time, let's find the largest factors for each number in an input array.

```java
protected void compute() {
    if (hi - lo <= CUTOFF) {
        seqReplaceWithLargestFactor(arr, lo, hi);
        return;
    }
    int mid = lo + (hi - lo) / 2;
    LargestFactorTask left = new LargestFactorTask(arr, lo, mid);
    LargestFactorTask right = new LargestFactorTask(arr, mid, hi);
    left.fork();
    right.compute();
    left.join();
}
```

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.

Maps and Reductions

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 and Reductions

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.

Google MapReduce and Hadoop

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
Parallelism on Other Data Structures

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".

Parallelism on LinkedLists

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

```java
public class LinkedList {
    public void compute() {
        if (not the end of the list) {
            fork a thread to do the rest of the elements;
        }
        do my work
        join with the thread after me
    }
}
```

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.

Parallelism on Balanced Trees

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

```java
public class BalancedTree {
    public void compute() {
        left.fork(); // Handles the entire left subtree
        right.compute(); // Handles the entire right subtree
        return left.join() + rightResult;
    }
}
```

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.

Work and Span

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

Let $T(n)$ 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.

---

**Analyzing a Parallel Algorithm: Work of Degenerate Tree**

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

**A (Parallel) Algorithm**

```java
int findMin(Node current) {
    if (current is a leaf) {
        return current.data;
    }
    return min(current.data, findMin(left), findMin(right));
}
```

**Degenerate Tree**

1
2
3
4
5

**Perfect Tree**

20
30
40
50
60
70
80

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

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

Solving this recurrence gives us:

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

---

**Analyzing a Parallel Algorithm: Span of Degenerate Tree**

A (Parallel) Algorithm

```java
int findMin(Node current) {
    if (current is a leaf) {
        return current.data;
    }
    return min(current.data, findMin(left), findMin(right));
}
```

**Degenerate Tree**

1
2
3

To calculate span, we assume all calls are in parallel. We look for the longest dependence chain. We make a recurrence:
To calculate work, we just do our standard analysis. First, we make a recurrence:

\[
\text{work}(n) = \begin{cases} 
O(1) & \text{if } n = 0 \\
2 \times \text{work}(\frac{n}{2}) + O(1) & \text{otherwise}
\end{cases}
\]

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

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

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

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!
**Speed-up, Parallelism, and \( T_P \)**

**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!

Okay, but we don’t have \( \infty \) processors...

Consider \( T_P \). We know the following:
- \( T_P \geq \frac{T_1}{P} \), the case where all the processors are always busy.
- \( T_P \geq T_\infty \), \( T_\infty \) is the length of the critical path which the algorithm must go through.

**Okay, but we don’t have \( \infty \) processors...**

Consider \( T_P \). We know the following:
- \( T_P \geq \frac{T_1}{P} \), the case where all the processors are always busy.
- \( T_P \geq T_\infty \), \( T_\infty \) is the length of the critical path which the algorithm must go through.

So, in an optimal execution, **asymptotically**, we know:

\[
T_P \in \Theta \left( \frac{T_1}{P} + T_\infty \right)
\]
Okay, but we don’t have \( \infty \) processors.

Consider \( T_P \). We know the following:

- \( T_P \geq \frac{T_1}{P} \), the case where all the processors are always busy.
- \( T_P \geq T_w \), \( T_w \) is the length of the critical path which the algorithm must go through.

So, in an optimal execution, **asymptotically**, we know:

\[
T_P \in \Theta \left( \frac{T_1}{P} + T_w \right)
\]

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

Applying Our Asymptotic Bound

**Minimum in a Perfect Tree**

When calculating the minimum 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 \( O\left( \frac{n}{P} + \frac{n}{\lg n} \right) \)

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 \( O\left( \frac{n^2}{P} + n \right) \)

Amdahl’s Law

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.

\[
T_1 = S \times \text{work}(n) + (1 - S) \times \text{work}(n)
\]
Amdahl’s Law
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:

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

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

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

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

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

So, Let’s Give 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!