Parallelism 2 Analysis, Amdahl's Law

Data Structures and Parallelism

Divide and Conquer SumThread

```
Class SumThread extends LibraryThreadObject{
      //constructor, fields unchanged.
      void run(){
            if(hi-lo == 1)
                  ans = arr[lo]
            else{
                  SumThread left = new SumThread(arr, lo, (hi+lo)/2);
                  SumThread right = new SumThread(arr, (hi+lo)/2, hi);
                  left.fork(); right.fork();
                  left.join(); right.join();
                  ans = left.ans + right.ans;
} } }
```

Divide And Conquer SumThread

```
int sum(int[] arr){
  SumThread t = new SumThread(arr, 0, arr.length);
  t.run(); //this first call isn't making a new thread
  return t.ans;
}
```

Divide And Conquer Optimization

Imagine calling our current algorithm on an array of size 4.

How many threads does it take?

7

It shouldn't take that many threads to add a few numbers.

And every thread introduces A LOT of overhead.

We'll want to **cut-off** the parallelism when the threads cause too much overhead.

Similar optimizations are used for (sequential) merge and quick sort

Cut-offs

Are we really saving that much?

Suppose we're summing an array of size 2^{30}

And we set a cut-off of size-100

- i.e. subarrays of size 100 are summed without making any new threads.

What fraction of the threads have we just eliminated?

99.9% !!!! (for fun you should check the math)

One more optimization

A small tweak to our code will eliminate half of our threads

```
left.fork();
right.fork();
left.join();
right.join();
```

Old version.
Too many threads

New version.
Good amount of threads

```
left.fork();
right.run();
left.join();
```

Current thread actually executes the right hand side.
Ordering of these commands is very important!

Analysis

None of our optimizations will make a difference in the O() analysis

But they will make a difference in practice.

Other Engineering Decisions

Getting every ounce of speedup out requires a lot of thought.

Choose a good sequential threshold

- Depends on the library
- For ours, a few hundred to one-thousand operations in the non-parallel call is recommended.

Library needs to warm up

Wait for more processors?

Memory Hierarchy

- Won't focus on this, but it can have an effect.

ForkJoin Library

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
Import java.util.concurrent.RecursiveAction;
```

Two possible classes to extend

`RecursiveTask<E>`

- Returns an E object

`RecrusiveAction`

- Doesn't return anything.

First thread created by:

```
ForkJoinPool.commonPool().invoke( ThreadObject );
```

ForkJoin Library summary

Start a new thread: `fork()`

Wait for a thread to finish: `join()`

- `join()` will return an object, if you extended `RecursiveTask`

Your Thread objects need to write a `compute()` method

Calling `compute()` does NOT start a new thread in the JVM.

ArraySum in ForkJoin

```
class SumThread extends RecursiveTask<Integer>{
    int lo; int hi; int[] arr; int cutoff;
    public SumThread(int l, int h, int[] a, int c){
        lo = l;
        hi = h;
        arr = a;
        cutoff = c;
    }
    …
```

```
        protected Integer compute(){
                if(hi-lo < cutoff){
                        int ans=0;
                        for(int i=lo; i<hi; i++)
                                ans += arr[i];
                        return new Integer(ans);
                }
                else{
                        SumThread left = new SumThread(arr, lo, (hi+lo)/2);
                        SumThread right = new SumThread(arr, (hi+lo)/2, hi);
                        left.fork();
                        Integer rightAns = right.compute();
                        Integer leftAns = left.join();
                        return newInteger(leftAns + rightAns);
} } }
```

Starting ArraySum in ForkJoin

```
public static Integer sum(int[] arr){
    SumThread thd =
        new SumThread(0, arr.length, arr, 500);
    return ForkJoinPool.commonPool().invoke( thd );
}
```

Reduce

It shouldn't be too hard to imagine how to modify this code to:

1. Find the maximum element in an array.
2. Determine if there is an element meeting some property.
3. Find the left-most element satisfying some property.
4. Count the number of elements meeting some property.
5. Check if elements are in sorted order.
6. [And so on…]

Reduce

You'll do similar problems in section tomorrow.

The key is to describe:

1. How to compute the answer at the cut-off.
2. How to merge the results of two subarrays.

We say parallel code like this "reduces" the array

We're reducing the arrays to a single item

Then combining with an **associative** operation.

e.g. sum, max, leftmost, product, count, or, and, …

Doesn't have to be a single number, could be an object.

Reduce – Terminology

An operation like we've seen is often called "reducing" the input.

Don't confuse this operation with reductions from 311/Exercise 7.

I'll abuse grammar and call the parallelism things "reduces" instead of reductions.

Map

Another common pattern in parallel code

Just applies some function to each element in a collection.

- No combining!

Easy example: vector addition

The fact that GPUs do certain maps quickly is a big reason why deep learning is so hot right now.

Maps & Reduces

Maps and Reduces are “workhorses” of parallel programming.

Google’s MapReduce framework relies on these showing up frequently.
- or Hadoop the open-scource version)

- **Usually** your reduces will extend `RecursiveTask<E>`
- **Usually** your maps will extend `RecursiveAction`

Analysis

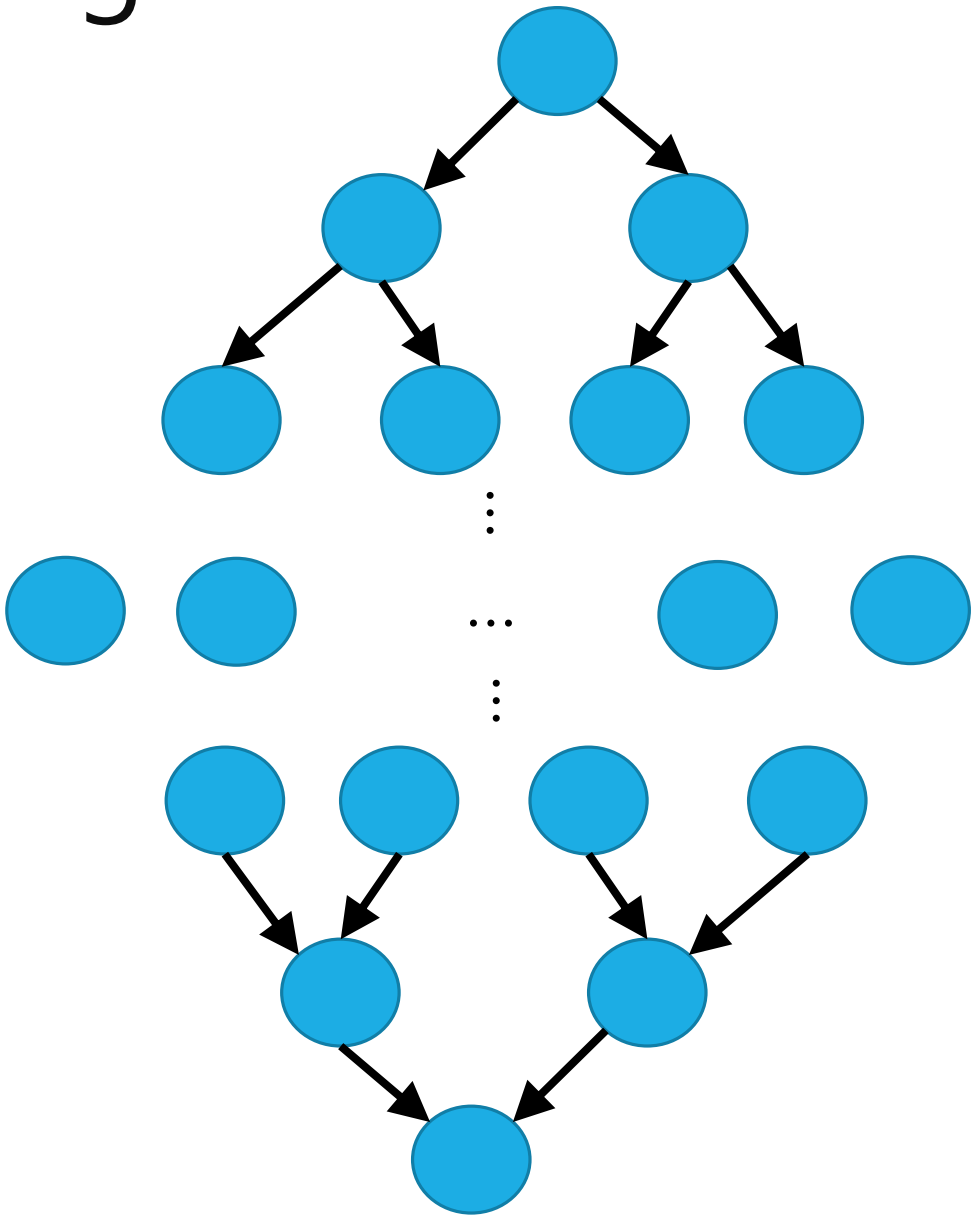
Big idea: let P, the number of processors, be another variable in our big-O analysis.

Let T_P be the big-O running time with P processors.

What is T_P for summing an array?

Useful Diagram

One node per
O(1) operation



Edge from u to v
if v waits for u.
I.e. v can't start
until u finishes.

Question: why are
there no cycles in
this graph?

Analysis

Big idea: let P, the number of processors, be another variable in our big-O analysis.

Let T_P be the big-O running time with P processors.

What is T_P for summing an array?

$O\left(\frac{n}{P} + \log n\right)$

Definitions

Work: T_1 it's $O(n)$ for summing an array.

Probably going to be equivalent to the running time of the code without parallelism.

Span: T_∞ $O(\log n)$ for summing an array

Longest path in graph of computation.

More Definitions

Speedup: for P processors: $\frac{T_1}{T_P}$

ideally: speedup will be close to P (“perfect linear speedup”)

Parallelism: $\frac{T_1}{T_\infty}$

the speedup when you have as many processors as you can use

Optimal T_P

We can calculate T_1, T_∞ theoretically.

But we probably care about T_P for, say, $P = 4$.

T_P can't beat (make sure you understand why):

- T_1/P
- T_∞

So optimal running time (asymptotically)

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

ForkJoin Framework has **expected** time guarantee of that O()

Assuming you write your code well enough.

Uses randomized scheduling.

Amdahl's Law

Now it's time for some bad news.

In practice, your program won't **just** sum all the elements in an array.

You will have a program with

Some parts that parallelize well

- Can turn them into a map or a reduce.

Some parts that won't parallelize at all

- Operations on a linked list.
- Reading a text file.
- A computation where each step needs the result of the previous steps.

Amdahl's Law

Let the work be 1 unit of time.

Let S be the portion of the code that is unparallelizable.

$T_1 = S + (1 - S) = 1.$

At best we can get perfect linear speedup on the parallel portion

$T_P \geq S + \frac{1-S}{P}$

So overall speedup with P processors

$$\frac{T_1}{T_P} \leq \frac{1}{S+(1-S)/P}$$

Parallelism: $\frac{T_1}{T_\infty} \leq \frac{1}{S}$

Amdahl's Law

Suppose our program takes 100 seconds.

And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors

6 processors

22 processors

67 processors

1,000,000 processors (approximately).

Amdahl's Law

Suppose our program takes 100 seconds.

And S is 1/3 (i.e. 33 seconds).

What is the running time with

3 processors: 33 + 67/3 ≈ 55 seconds

6 processors: 33 + 67/6 ≈ 44 seconds

22 processors: 33 + 67/22 ≈ 36 seconds

67 processors 33 + 67/67 ≈ 34 seconds

1,000,000 processors (approximately). ≈ 33 seconds

Amdahl's Law

This is BAD NEWS

If 1/3 of our program can't be parallelized, we can't get a speedup better than 3.

No matter how many processors we throw at our problem.

And while the first few processors make a huge difference, the benefit diminishes quickly.

Amdahl's Law and Moore's Law

In the Moore's Law days, 12 years was long enough to get 100x speedup.

Suppose in 12 years, the clock speed is the same, but you have 256 processors.

What portion of your program can you hope to leave unparallelized?

$$100 \leq \frac{1}{S+\frac{1-S}{256}}$$

[wolframalpha says] $S \leq 0.0061$.

Amdahl's Law: Moving Forward

Unparallelized code becomes a bottleneck quickly.

What do we do? Design smarter algorithms!

Consider the following problem:

Given an array of numbers, return an array with the "running sum"

3	7	6	2	4

3	10	16	18	22

More clever algorithms

Doesn’t look parallelizable…

But it is!

Think about how you might do this (it’s NOT obvious)

We’ll go through it on Friday (or possibly Monday)!