Parallelism 2 Analysis, Amdahl's Law

CSE 332 25Sp
Lecture 20

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	Ex 7 (DFS, coding) due Ex 9 (reductions, gs) out			Bring laptop!	TODAY Ex 8 (Dijkstra, gs) due Ex 10,11 (parallel prog) out
Next Week	Ex 9 (reductions, gs) due				Ex 10 (parallel, prog) due

Optional readings (Grossman) covers next few weeks of parallelism and concurrency

ParallelSum: Take 2

```
int sum(int[] arr){
      int len = arr.length;
      int ans = 0;
      SumThread[] ts = new SumThread[4];
      for(int i=0; i<4; i++)
            ts[i] = new SumThread(arr, i*len/4 (i+1)*len/4);
            ts[i].fork()
      for(int i=0; i<4; i++)
            ts[i].join()
            ans += ts[i].ans;
      return ans;
}
```

Optimizing: Number of Threads

The last version of ParallelSum will work.

I.e. it will always get the right answer

And it will use 4 threads.

But…

What if we get a new computer with 6 processors?

- We'll have to rewrite our code

What if the OS decides "no, you only get 2 processors right now."

What if our threads take wildly different amounts of time?

Optimizing: Number of Threads

The counter-intuitive solution:

Even more parallelism!

Divide the work into more smaller pieces.

If you get more processors, you take advantage of all of them.

If one thread finishes super fast, throw the next thread to that processor.

"Load Imbalance"

Engineering Question:

- Let's say we change our `ParallelSum` code so each thread adds 10 elements.
- Is that a good idea? What's the running time of the code going to be?

Thread Creation

If we create $n/10$ threads, each summing 10 elements…

Creating $n/10$ threads one-right-after-the-other takes $\Theta(n)$ time.

(Same with joining the threads together at the end).

This is a linear time algorithm now. Can we do better?

Divide and Conquer: Parallelism

What if we want a bunch of threads, but don't want to spend a bunch of time making threads?

Parallelize thread creation too!

Divide and Conquer SumThread

```
Class SumThread extends SomeThreadObject{
  //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.start(); right.start();
     left.join(); right.join();
     ans = left.ans + right.ans;
    }
  }
}
```

Divide And Conqure SumThread

```
int sum(int[] arr){
  SumThread t = new SumThread(arr, 0, arr.length);
  t.run(); //this 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 make

6

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 can be 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% !!!! (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

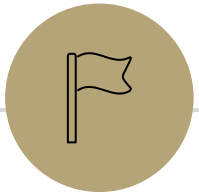
```
left.fork();
right.run();
left.join();
```

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.



ForkJoin Framework

ForkJoin Framework

Method	Use
`compute`	Thread objects override (void/V) method `compute` When `fork` is called, `compute` method is executed A bit like main(…)---default starting point
`fork`	Starts a new thread executing `compute` method
`join`	Calling `otherThread.join()` pauses this thread until `otherThread` has completed its `compute`.
`RecursiveTask<V>`	Class which we extend to make threads to return a result of type `V`. `compute` returns `V` for this object
`Recursive Action`	Class we extend when we don't return a result
`ForkJoinPool`	Object that manages threads
`invoke`	Pool method to start first thread object.

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---Magic Incantations

```
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:

```
static final ForkJoinPool POOL = new ForkJoinPool();
POOL.invoke(firstThd);
```

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

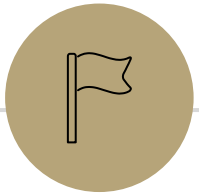
```
class SumThread extends RecursiveTask<Integer>{
  int lo; int hi; int[] arr; int cutoff;
  static final ForkJoinPool POOL = new ForkJoinPool();
  public SumThread(int l, int h, int[] a, int c){
        lo = l;
        hi = h;
        arr = a;
        cutoff = c;
    }
```

…

```
…
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);
      }
}

…
```

```
…
  public static Integer sum(int[] arr){
    SumThread thd = new SumThread(0, arr.length, arr, 500);
    POOL.invoke( thd );
  }
}//end class SumThread
```



The “reduce” pattern

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 did similar problems yesterday.

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 lecture 18.

We'll call the parallelism-related operation "a reduce." or "a reduce operation" (and the algorithm design thing "a reduction")

This convention isn't universal (you'll find resources calling the parallel code "a reduction")

Map

Another common pattern in parallel code

Just applies some function to each element in a collection.

- No combining!

`PowMod` from yesterday was a map

Easy example: vector addition

These operations are common enough that some processors do vector computations in parallel at the hardware level.

- That's a major reason why GPUs are so popular for ML applications.

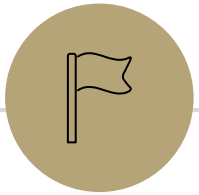
Maps & Reduces

Maps and Reduces are “workhorses” of parallel programming.

Google’s MapReduce framework relies on these showing up frequently.

- or Hadoop (the open-source version)

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



Analysis

Analysis

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

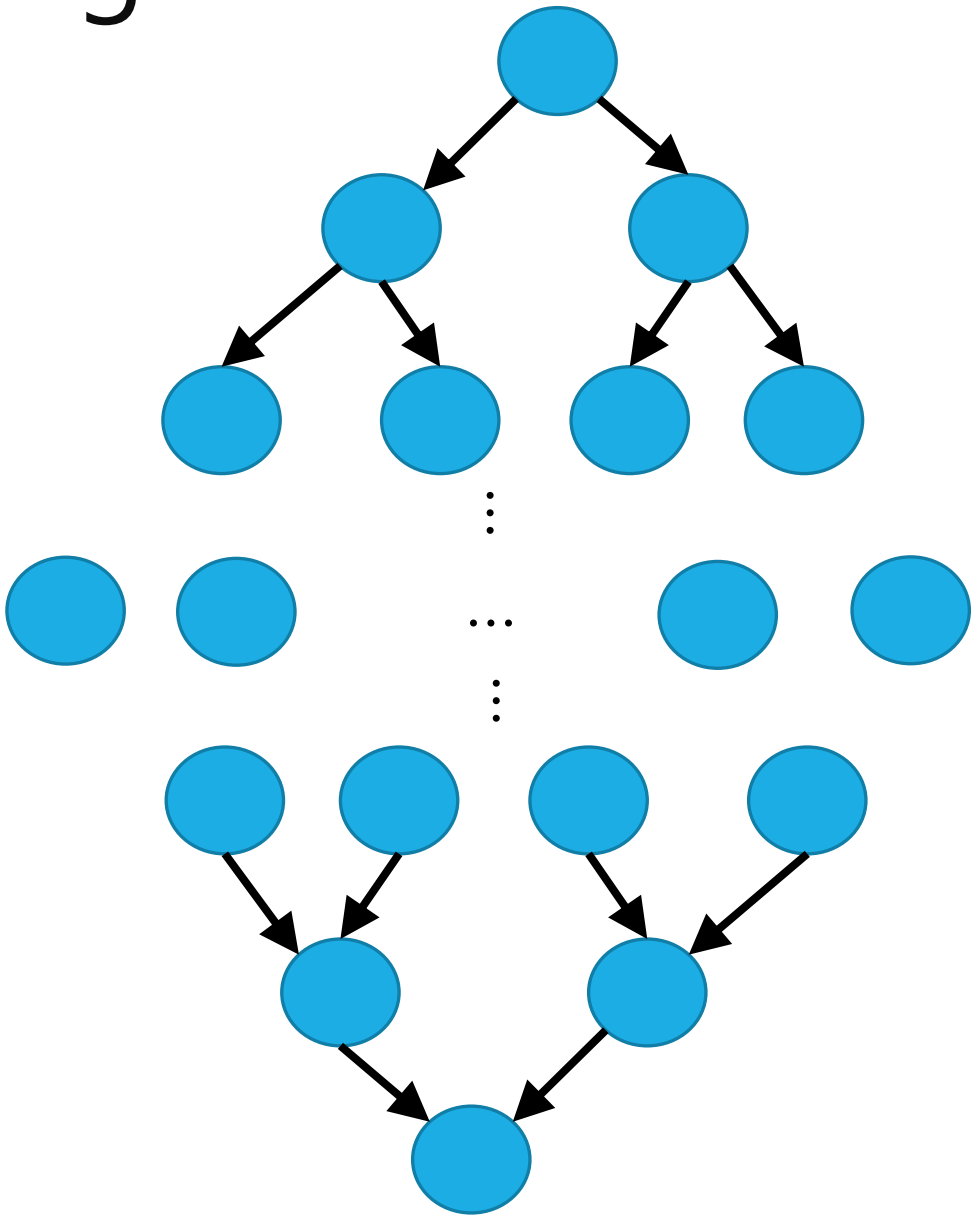
Let T_P be the big-O running time with P processors.

What is T_P for summing an array?

We'll need to do computations in a new way.

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.

Might be due to `join` or might be sequential code.

Useful Diagram

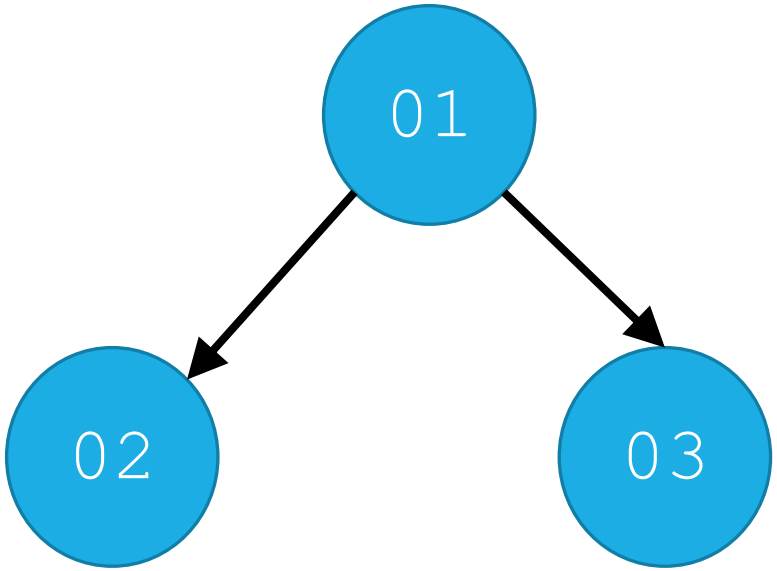
One node per
O(1) operation

What does the
dependency
graph look like for
that snippet?

```
01: x=x+5
02: y=x+7
03: z=x+13
```

Useful Diagram

One node per
O(1) operation




```
01: x=x+5
02: y=x+7
03: z=x+13
```

Useful Diagram

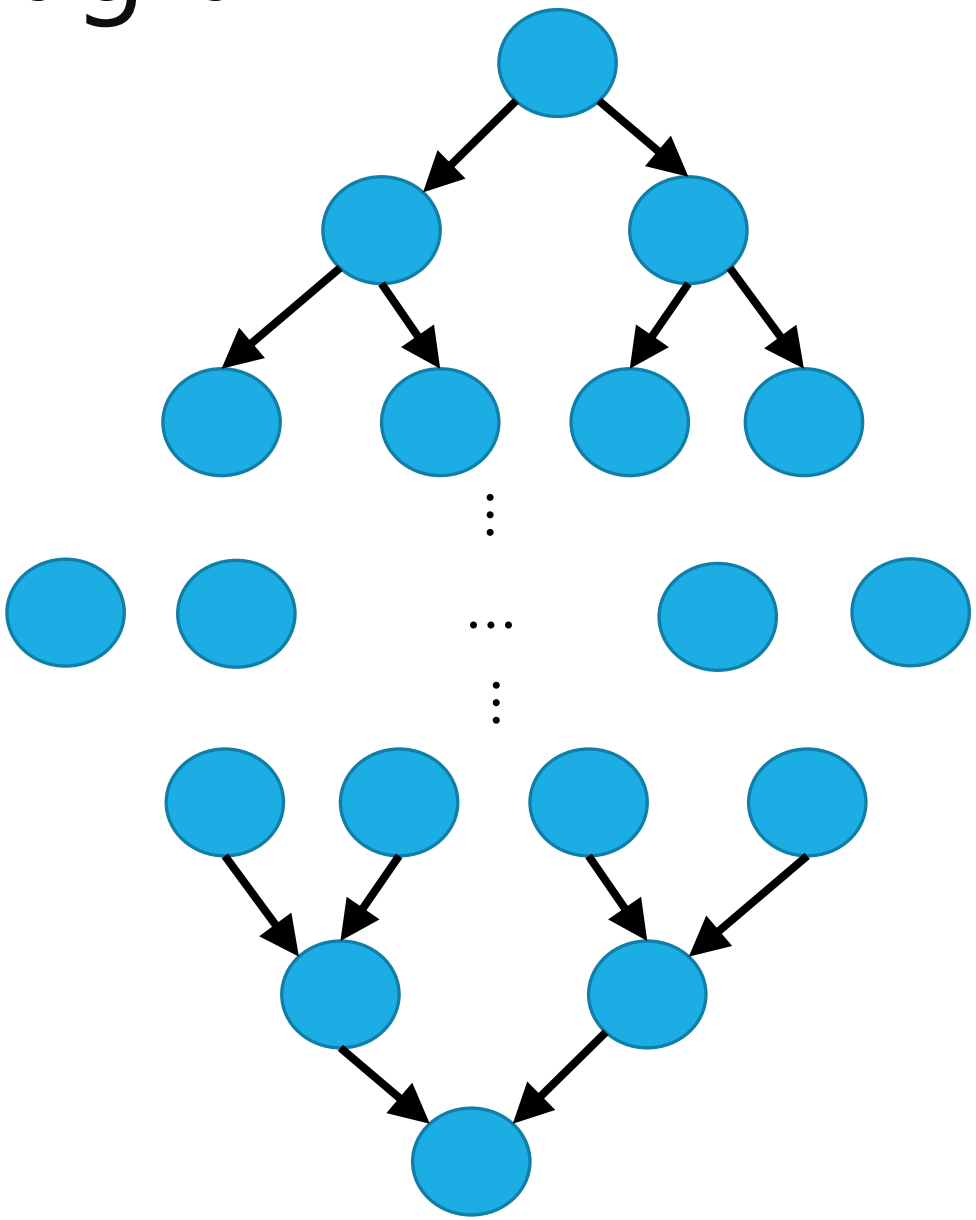
For parallel code:

Fork will usually “split” a node into two nodes (even if only one new thread is created), parent to two children (or many if many forks)

Join will usually “combine” two nodes into one node (even if only one thread is joined), two sources to one destination (or many if many joins)

Useful Diagram

One node per
O(1) operation



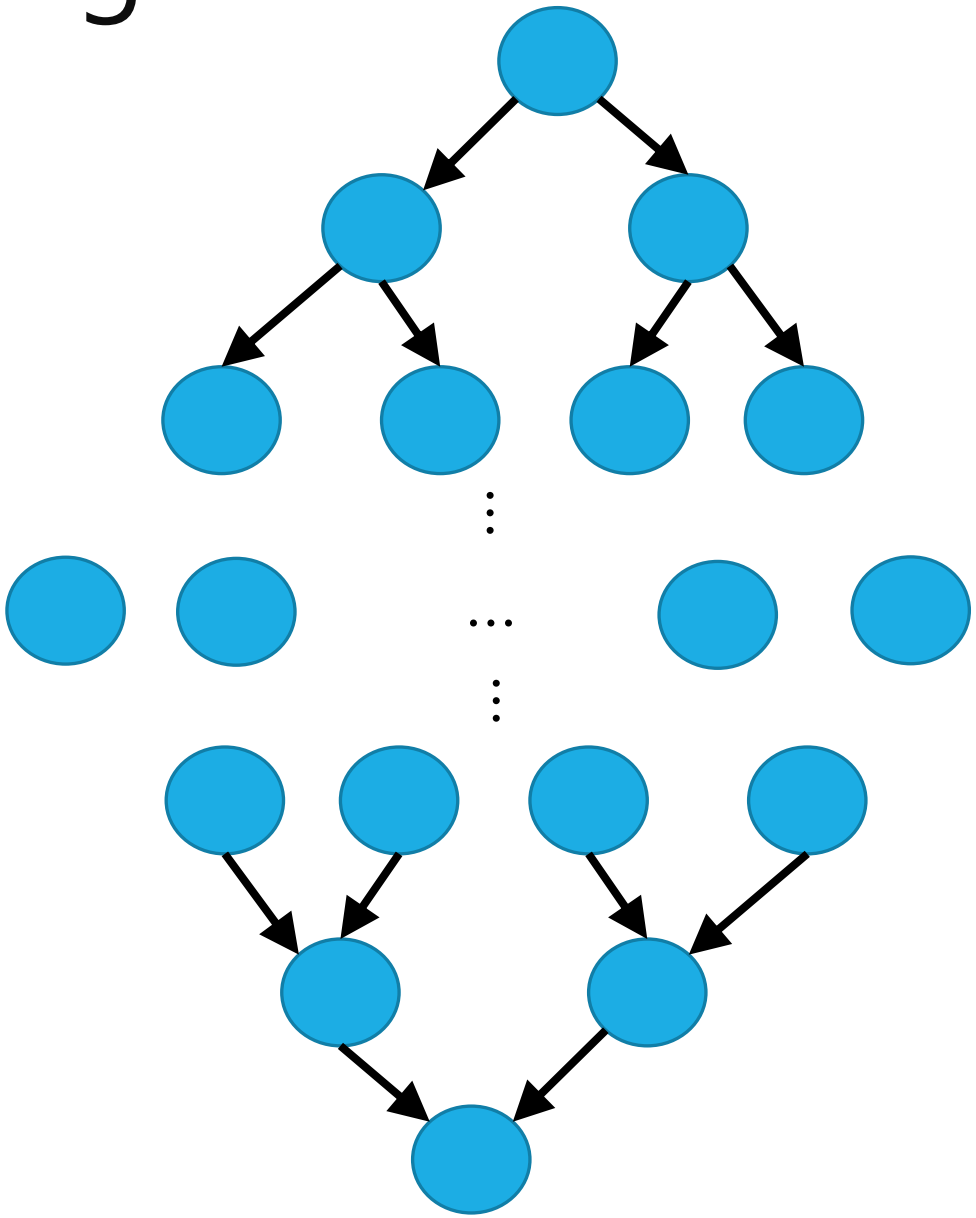
Divide to create threads.

Base Case computations

`Join` and combine to create final answer.

Useful Diagram

One node per
O(1) operation



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 (but not necessarily!!!) going to be equivalent to the running time of the code you would write if you had never heard of parallelism.

Definition is running time of **parallel** code if you had a single processor.

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

$T_?$ is running time with ? Processors, so span is "you always have a processor available"

Longest path in graph of computation. "critical path"

More Definitions

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

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

E.g. if $T_1 = 100$sec And $T_4 = 25$sec, then speedup$= \frac{100}{25} = 4$

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

the speedup when you have as many processors as you can use (there's a point at which another one won't actually help).

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_∞

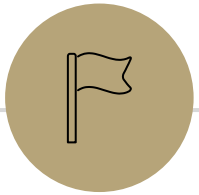
So optimal running time (asymptotically)

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

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

Assuming you write your code well enough.

Uses randomized scheduling.



Amdahl’s Law

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. DATA STRUCTURES MATTER!!!
- 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 ("sequential").

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

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

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

Therefore 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

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

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

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

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 and Moore’s Law

Moore’s Law was “a business decision”

- How much effort/money/employees are dedicated to improving processors so computers got faster.

Amdahl’s Law is a theorem

- You can prove it formally.

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 Monday!