Lecture 9 — Algorithms, Concurrency, and Parallelism

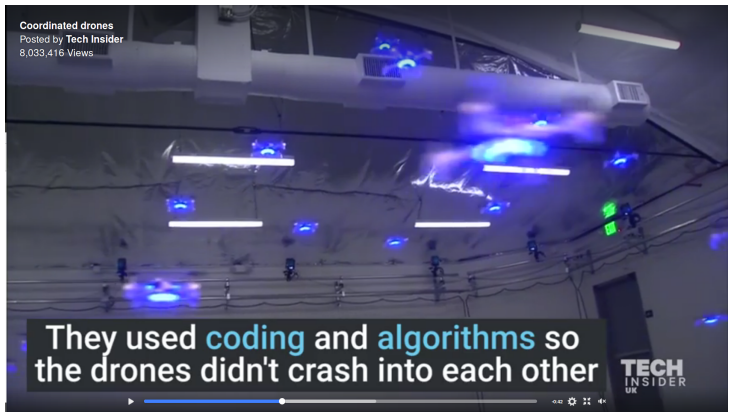
Jeff Zarnett & Patrick Lam
jzarnett@uwaterloo.ca, patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 7, 2025

Part I

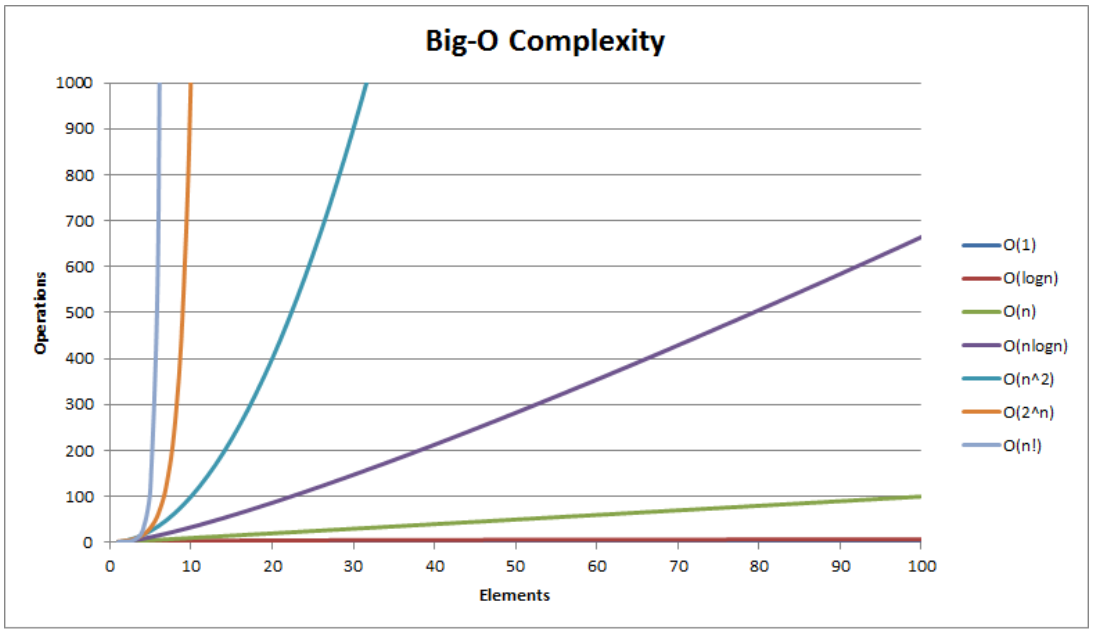
Algorithms




```
if(goingToCrashIntoEachOther) {
   dont();
}
```

Algorithmic Complexity

Remember we often care about worst-case run-time performance:



But you know this already!

You're not here to hear me tell you to use Quicksort instead of Bubblesort.

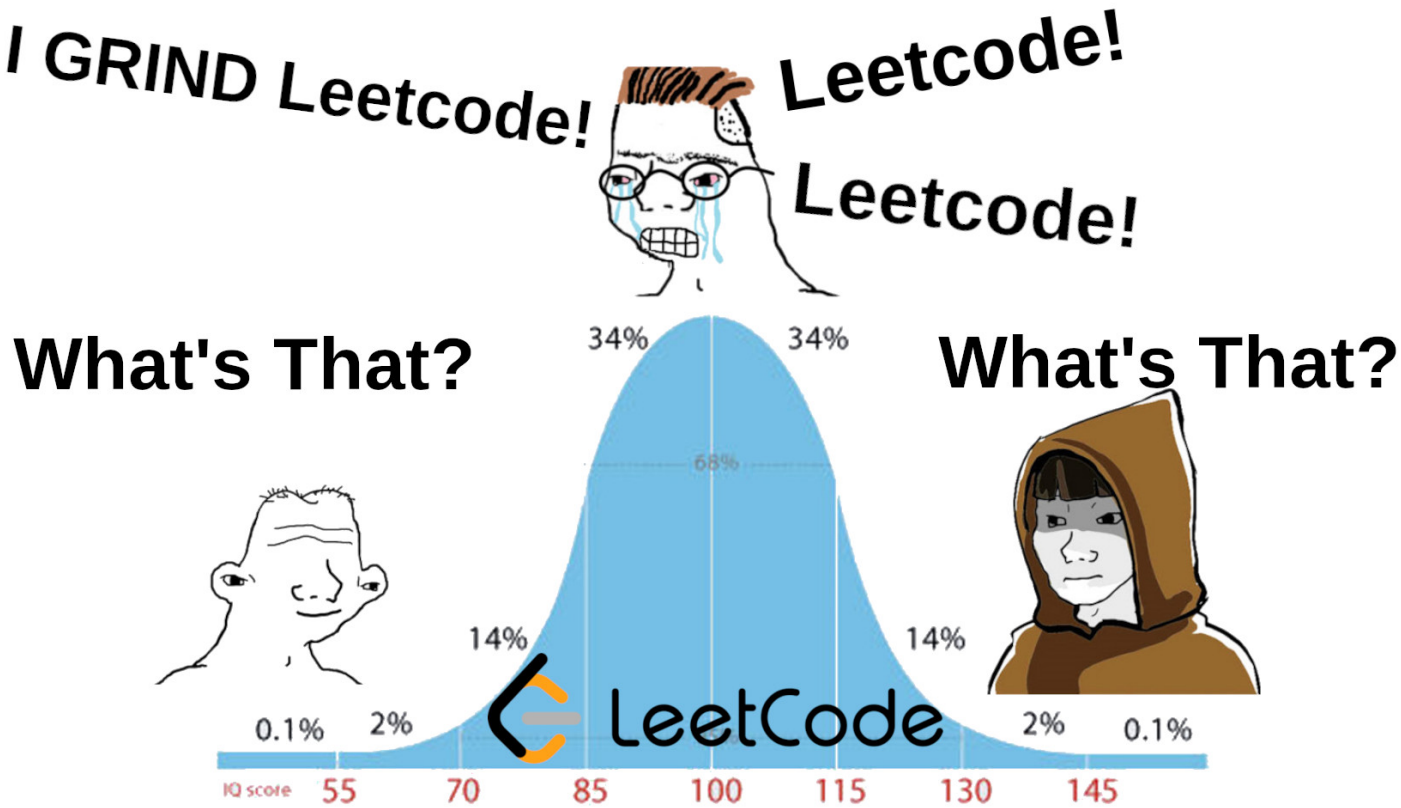
Is Sorting a Bad Example?

In reality, you use a library- or language-provided sort.

That is, just call `.sort()` on the collection and it's done!

Ah, but what about... leetcode?

Just Grind Leetcode and Git Gud



Is this how to get a job these days? (Spoiler alert: often.)

Good Ideas in the Grind

There are some ideas from how to grind leetcode that help in other situations.

More time is likely to be spent on architecture or understanding requirements.

But let’s take some ideas from what I tell people for interview prep.

Start with Simple

In many (non-leetcode) scenarios, simple and correct suffices!

Either *n* is small enough or this code is not performance-critical.

Next Step: Refine

Refine is improve what you've got. See *Cracking the Coding Interview* for more!

Look for:

- Bottlenecks
- Unnecessary Work
- Duplicated Work

Use the Force, Luke

In a coding-interview, you have a well-defined set of pieces of information.
Chances are, you need to use all of them to find optimal solution.

In other situations, it may not be provided; investigate!

You may need to be the change you want to see.

Other Thoughts

Big-O complexity assumes a large enough *n* that the other terms don't matter.

That's fine for an interview, but is not always true in real life.

Sorting a large array or putting it into a Hashmap is likely to be optimal if you intend to use it many times, but maybe is not worthwhile if we search once.

Ask for Help

"Is it hard for you to just ask for some help?"

Me:

Algorithmic Improvements Limits

Remember also that algorithmic complexity improvements always have limits.

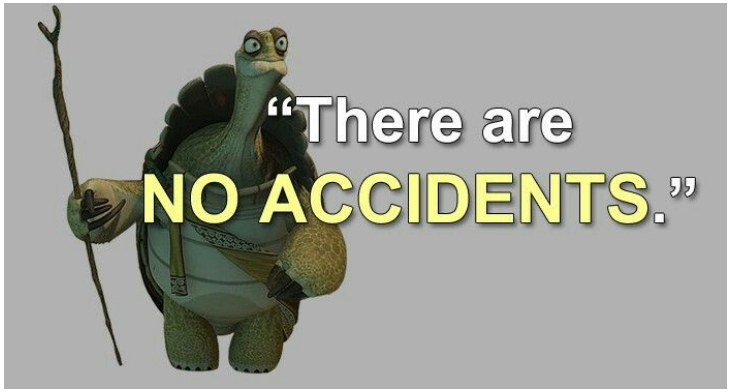
To successfully mark the final exam, the teaching team really does have to look at every question of every exam paper.

No amount of cleverness in the process can get around the fact that the only way to properly mark the exam is to look at every answer from each student.

Accidentally Quadratic

Many problems really are linear at the core; a problem arises when we combine two linear things and get quadratic behaviour.

Oh no! This is a situation that can be described as “accidentally quadratic”.



Older Code, But it Checks Out

The 2019 Rust-specific example is about Robin-Hood Hashing.

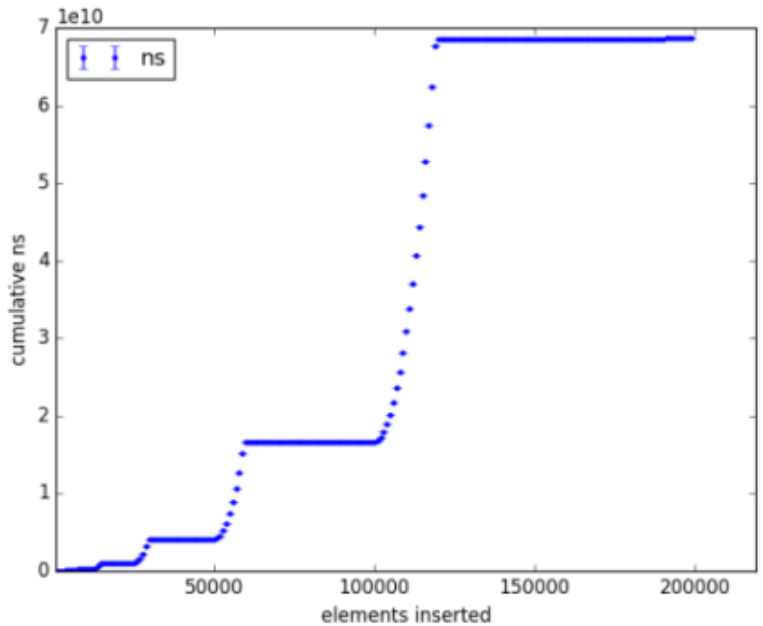
Remember open addressing and linear probile?

Moving the data to a new hash table is a problem...

Copy the data to a table half the size of the original.

As it gets close to full: quadratic behaviour!

Linear for each element in 1^{st} table, linear to find a free space in 2^{nd}.



Sometimes is Always

Algorithmic analysis is, remember, often worst-case scenario.

Don’t over-index on this example:

“WeLL aCtuALLy”

It just means that sometimes the normally-best approach isn’t best this time.

Hello Marking, My Old Friend

If we can’t do any better than linear (or maybe even quadratic), what do we do?

The exam marking example gives a hint: divide & conquer!

Or, in other words, parallelism.

Part II

Parallelism

Parallelism versus Concurrency

Before we talk about parallelism, let's distinguish it from concurrency.

Parallelism

Two or more tasks are parallel
if they are running at the same time.

Main goal: run tasks as fast as possible.

Main concern: dependencies.

Concurrency

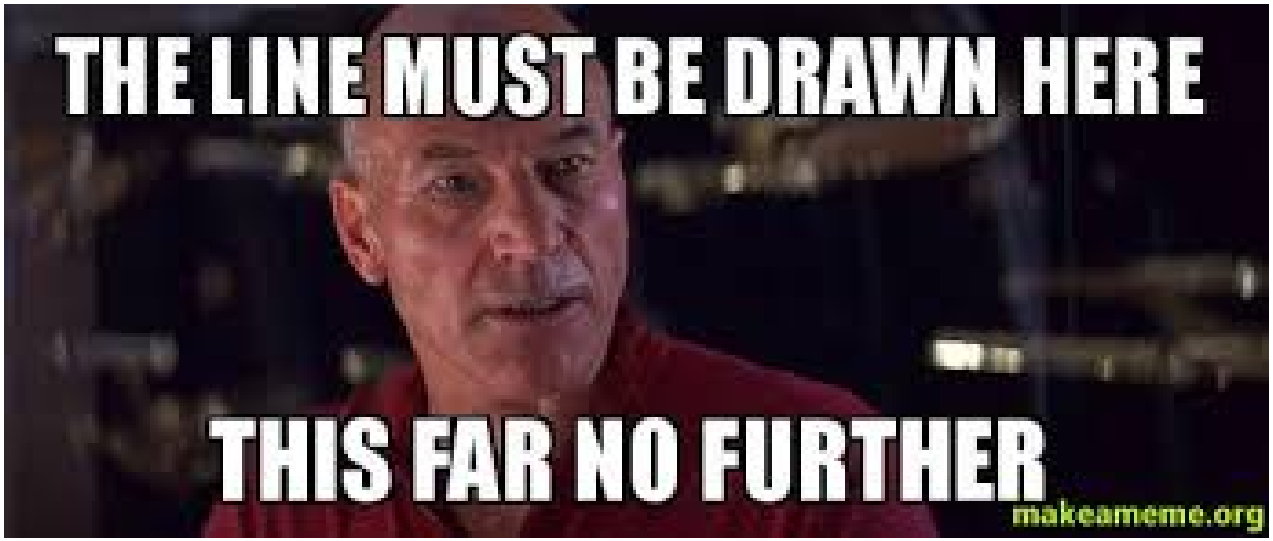
Two or more tasks are concurrent
if the ordering of the two tasks is not predetermined.

Main concern: synchronization.

Limitations of Speedups

Our main focus is parallelization.

- Most programs have a sequential part and a parallel part; and,
- Amdahl's Law answers, "what are the limits to parallelization?"

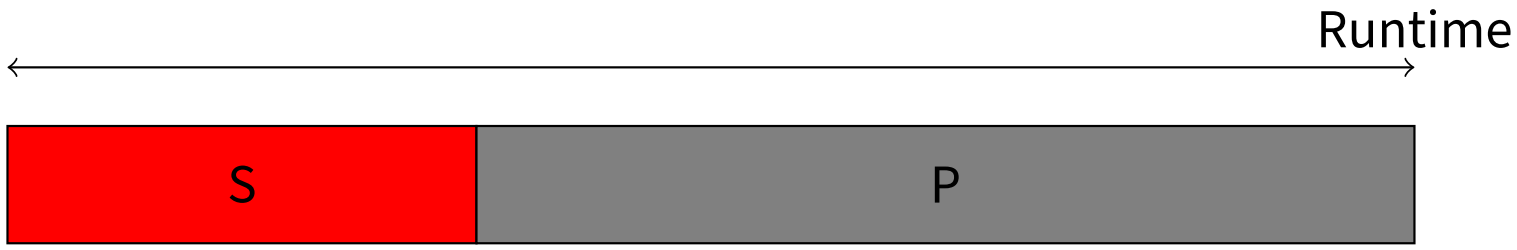


Visualizing Amdahl's Law

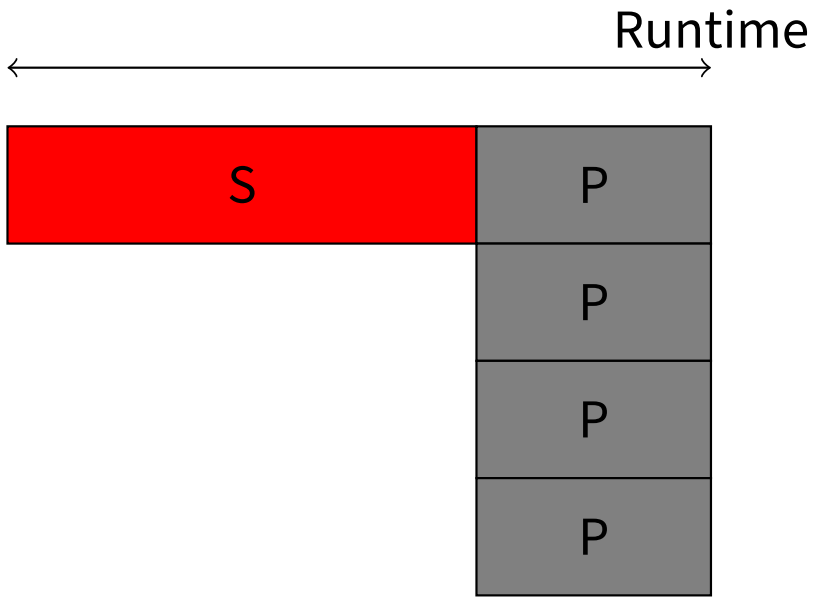
S: fraction of serial runtime in a serial execution.
P: fraction of parallel runtime in a serial execution.
Therefore, $S + P = 1$.

With 4 processors, best case, what can happen to the following runtime?

Visualizing Amdahl's Law



We want to split up the parallel part over 4 processors



Obey the Law

T_s: time for the program to run in serial

N: number of processors/parallel executions

T_p: time for the program to run in parallel

- Under perfect conditions, get N speedup for P

$$T_p = T_s \cdot (S + \frac{P}{N})$$

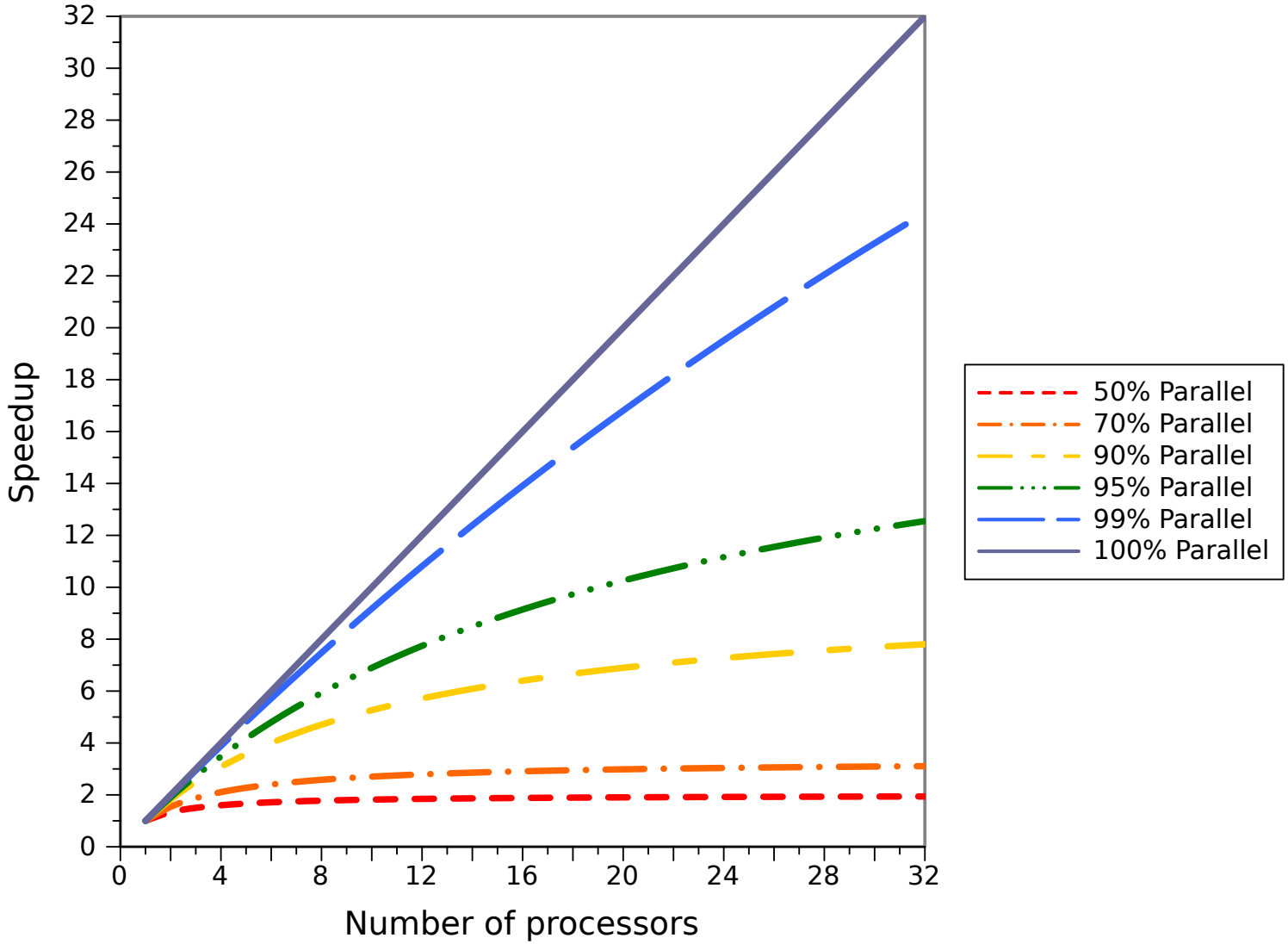
Obey the Law

How much faster can we make the program?

$$
\begin{aligned}
\mathit{speedup} &= \frac{T_s}{T_p} \\
&= \frac{T_s}{T_S \cdot (S + \frac{P}{N})} \\
&= \frac{1}{S + \frac{P}{N}}
\end{aligned}
$$

(assuming no overhead for parallelizing; or costs near zero)

Fixed-Size Problem Scaling, Varying Fraction of Parallel Code



Replace S with $(1 - P)$:

$$speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

$$maximum\ speedup = \frac{1}{(1-P)}, \text{since } \frac{P}{N} \to 0$$

As you might imagine, the asymptotes in the previous graph are bounded by the maximum speedup.

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

Initialization and recombination code in this routine requires 400 ms.

So with one processor executing, it would take about 4.6 s to execute the loop.

Split it up and execute on two processors: about 2.3 s to execute the loop.

Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s.

Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%.

Amdahl's Law on the 5 s Task

Applying this formula to the example:

Processors	Run Time (s)
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

Assumptions behind Amdahl's Law

We assume:

- problem size is fixed (we'll see this soon);
- program/algorithm behaves the same on 1 processor and on *N* processors;
- that we can accurately measure runtimes—
  i.e. that overheads don't matter.

Amdahl's Law Generalization

The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law.

$f_1, f_2, \ldots, f_n$: fraction of time in part n

$S_{f_1}, S_{f_n}, \ldots, S_{f_n}$: speedup for part n

$$speedup = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \ldots + \frac{f_n}{S_{f_n}}}$$

Application (1)

Consider a program with 4 parts in the following scenario:

		Speedup	
Part	Fraction of Runtime	Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

We can implement either Option 1 or Option 2.
Which option is better?

Application (2)

"Plug and chug" the numbers:

Option 1

$$speedup = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

Option 2

$$speedup = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

Empirically estimating parallel speedup P

Useful to know, don't have to commit to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}$$

- Quick way to guess the fraction of parallel code
- Use $P_{\text{estimated}}$ to predict speedup for a different number of processors

Summary of Amdahl's Law

Important to focus on the part of the program with most impact.

Amdahl's Law:

- estimates perfect performance gains from parallelization (under assumptions); but,
- only applies to solving a fixed problem size in the shortest possible period of time

Gustafson's Law: Formulation

n: problem size
$S(n)$: fraction of serial runtime for a parallel execution
$P(n)$: fraction of parallel runtime for a parallel execution

$$\begin{aligned} T_p &= S(n) + P(n) = 1 \\ T_s &= S(n) + N \cdot P(n) \end{aligned}$$

$$speedup = \frac{T_s}{T_p}$$

Gustafson's Law

$$speedup = S(n) + N \cdot P(n)$$

Assuming the fraction of runtime in serial part decreases as n increases, the speedup approaches N.

Yes! Large problems can be efficiently parallelized. (Ask Google.)

Amdahl's Law

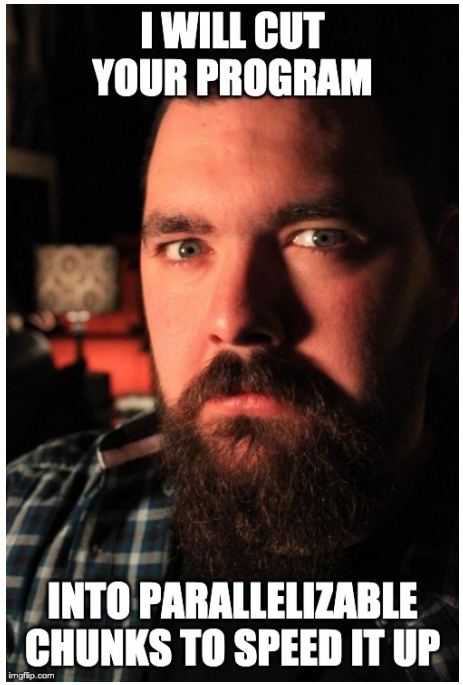
Suppose you're travelling between 2 cities 90 km apart. If you travel for an hour at a constant speed less than 90 km/h, your average will never equal 90 km/h, even if you energize after that hour.

Gustafson's Law

Suppose you've been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.

Part III

Parallelization Techniques



Lock It Down

The more locks and locking we need, the less scalable the code is going to be.

You may think of the lock as a resource. The more threads or processes that are looking to acquire that lock, the more “resource contention” we have.

And thus more waiting and coordination are going to be necessary.

“We have a new enemy...”

Assuming we’re not working with an embedded system where all memory is statically allocated in advance, there will be dynamic memory allocation.

The memory allocator is often centralized and may support only one thread allocating or deallocating at a time (using locks to ensure this).

This means it does not necessarily scale very well.

There are some techniques for dynamic memory allocation that allow these things to work in parallel.

Everybody Into the Pool

If we have a pool of workers, the application just submits units of work, and then on the other side these units of work are allocated to workers.

The number of workers will scale based on the available hardware.

This is neat as a programming practice: as the application developer we don't care quite so much about the underlying hardware.

Let the operating system decide how many workers there should be, to figure out the optimal way to process the units of work.

Rust Thread Pool

```rust
use std::collections::VecDeque;
use std::sync::{Arc, Mutex};
use threadpool::ThreadPool;
use std::thread;

fn main() {
    let pool = ThreadPool::new(8);
    let queue = Arc::new(Mutex::new(VecDeque::new()));
    println!("main thread has id {}", thread_id::get());

    for j in 0 .. 4000 {
        queue.lock().unwrap().push_back(j);
    }
    queue.lock().unwrap().push_back(-1);
```

Rust Thread Pool

```
    for i in 0 .. 4 {
        let queue_in_thread = queue.clone();
        pool.execute(move || {
            loop {
                let mut q = queue_in_thread.lock().unwrap();
                if !q.is_empty() {
                    let val = q.pop_front().unwrap();
                    if val == -1 {
                        q.push_back(-1);
                        println!("Thread {} got the signal to exit.",
                            thread_id::get());
                        return;
                    }
                    println!("Thread {} got: {}!", thread_id::get(), val);
                }
            }
        });
    }
    pool.join();
}
```

Thread Pool Output

```
main thread has id 4455538112
Thread 123145474433024 got: 0!
Thread 123145474433024 got: 1!
Thread 123145474433024 got: 2!

...

Thread 123145478651904 got: 3997!
Thread 123145478651904 got: 3998!
Thread 123145478651904 got: 3999!
Thread 123145476542464 got the signal to exit.
Thread 123145484980224 got the signal to exit.
Thread 123145474433024 got the signal to exit.
Thread 123145478651904 got the signal to exit.
```