Lecture 10 — Concurrency and Parallelism

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

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Part I

Limits

“Push it to the limit!”
(https://www.youtube.com/watch?v=kZu5iDTtNg0)

Writing Scalable Code

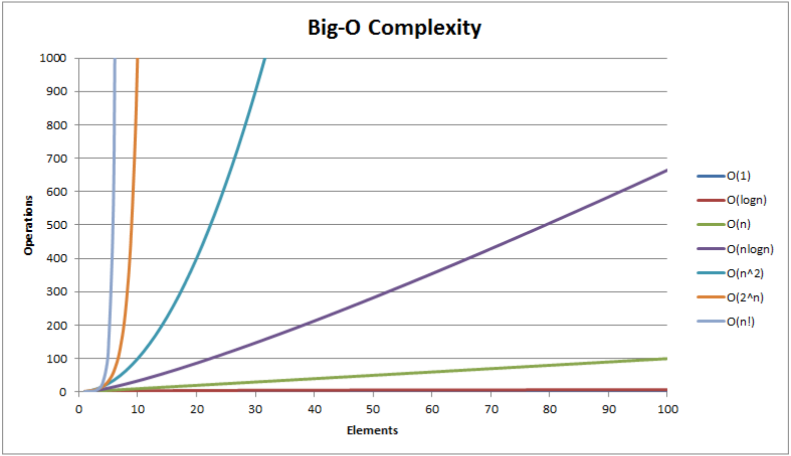
Think about the worst case run-time performance of the algorithm.

An algorithm that's $O(n^3)$ scales so much worse than one that's $O(n)$...

Trying to do an insertion sort on a small array is fine (actually... recommended); doing it on a huge array is madness.

Choosing a good algorithm is very important if we want it to scale.

Big-O Complexity



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.

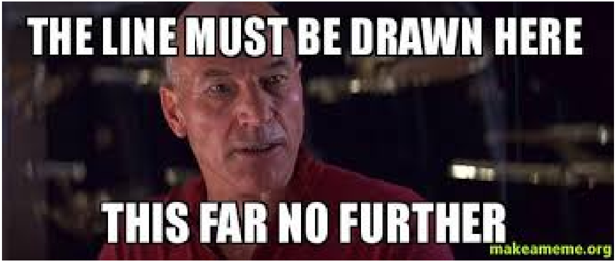
Concurrency

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

Main concern: synchronization.

Our main focus is parallelization.

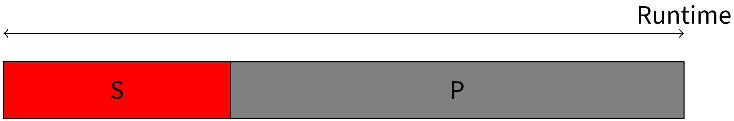
- Most programs have a sequential part and a parallel part; and,
- Amdahl's Law answers, "what are the limits to parallelization?"



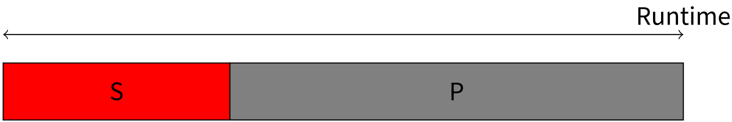
Visualizing Amdahl's Law

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

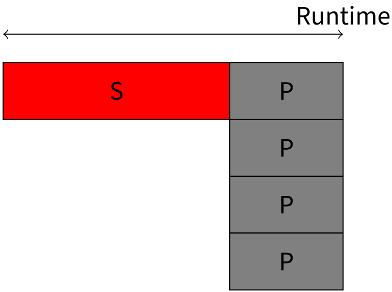
With 4 processors, best case, what can happen to the following runtime?



Visualizing Amdahl's Law



We want to split up the parallel part over 4 processors



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

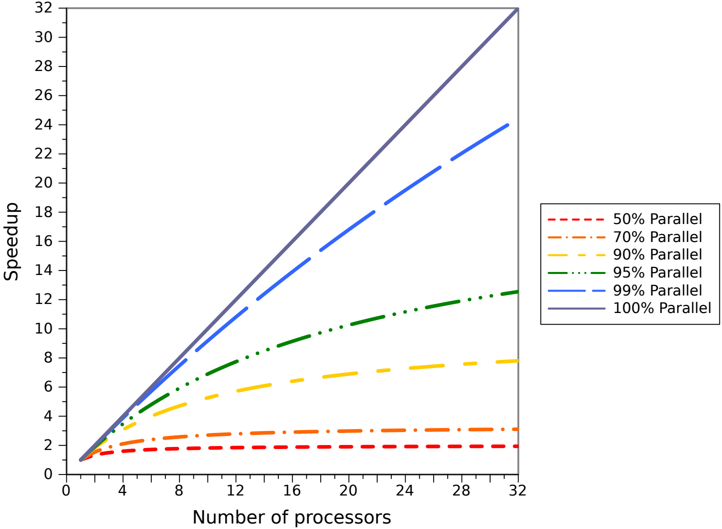
$$T_p = T_s \cdot (S + \frac{P}{N})$$

How much faster can we make the program?

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

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

Fixed-Size Problem Scaling, Varying Fraction of Parallel Code



Replace S with $(1 - P)$:

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

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

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.)

Driving Metaphor

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 II

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

```
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);
```

```
    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.
```

Part III

Threads vs Processes

Recall the difference between processes and threads:

- Threads are basically light-weight processes which piggy-back on processes' address space.

Traditionally (pre-Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads).

History

Developers mostly used `fork` before there was a standardized way to create threads (`clone` was non-standards-compliant).

For performance reasons, along with ease and consistency, we'll use Pthreads.

`fork` creates a new process.

- Drawbacks?
- Benefits?

Benefit: `fork` is Safer and More Secure Than Threads

1. Each process has its own virtual address space:
   - Memory pages are not copied, they are copy-on-write—
   - Therefore, uses less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process.

Drawback of Processes: Threads are Easier and Faster

- Interprocess communication (IPC) is more complicated and slower than interthread communication.
  - Need to use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
- Processes have much higher startup, shutdown and synchronization cost.
- And, Pthreads/C++11 threads fix issues with `clone` and provide a uniform interface for most systems **(Assignment 1)**.

Appropriate Time to Use Processes

If your application is like this:

- Mostly independent tasks, with little or no communication.
- Task startup and shutdown costs are negligible compared to overall runtime.
- Want to be safer against bugs and security holes.

Then processes are the way to go.

Threads vs Processes: Rust

```rust
use std::process::Command;

fn main() {
    for j in 0 .. 50000 {
        Command::new("/bin/false").
            spawn();
    }
}
```

```
1.530 s +/-  0.134 s
```

```rust
use std::thread;

fn main() {
    for j in 0 .. 50000 {
        thread::spawn(|| {
            false
        });
    }
}
```

```
630.5 ms +/-  21.5 ms
```

- We'll now look at thread and process-based parallelization.
- Although threads and processes differ, we don't care for now.

Pattern 1: Multiple Independent Tasks

Only useful to maximize system utilization.

- Run multiple tasks on the same system
  (e.g. database and web server).

  If one is memory-bound and the other is I/O-bound, for example, you'll get maximum utilization out of your resources.

  Example: cloud computing, each task is independent and can tasks can spread themselves over different nodes.

- Performance can increase linearly with the number of threads.

Pattern 2: Multiple Loosely-Coupled Tasks

Tasks aren't quite independent, so there needs to be some inter-task communication (but not much).

- Communication might be from the tasks to a controller or status monitor.

Refactoring an application can help with latency.
For instance: split off the CPU-intensive computations into a separate thread—your application may respond more quickly.

Example: A program (1) receives/forwards packets and (2) logs them. You can split these two tasks into two threads, so you can still receive/forward while waiting for disk. This will increase the throughput of the system.

Pattern 3: Multiple Copies of the Same Task

Variant of multiple independent tasks: run multiple copies of the same task (probably on different data).

- No communcation between different copies.

Again, performance should increase linearly with number of tasks.

Example: In a rendering application, each thread can be responsible for a frame (gain throughput; same latency).

Pattern 4: Single Task, Multiple Threads

Classic vision of "parallelization".

Example: Distribute array processing over multiple threads—each thread computes results for a subset of the array.

- Can decrease latency (and increase throughput), as we saw with Amdahl's Law.
- Communication can be a problem, if the data is not nicely partitioned.
- Most common implementation is just creating threads and joining them, combining all results at the join.

Pattern 5: Pipeline of Tasks

Seen briefly in computer architecture.

- Use multiple stages; each thread handles a stage.

  Example: a program that handles network packets: (1) accepts packets, (2) processes them, and (3) re-transmits them. Could set up the threads such that each packet goes through the threads.

- Improves throughput; may increase latency as there's communication between threads.

- In the best case, you'll have a linear speedup.

  Rare, since the runtime of the stages will vary, and the slow one will be the bottleneck (but you could have 2 instances of the slow stage).

Pattern 6: Client-Server

To execute a large computation, the server supplies work to many clients—as many as request it.

Client computes results and returns them to the server.

Examples: botnets, `SETI@Home`, GUI application (backend acts as the server).

Server can arbitrate access to shared resources (such as network access) by storing the requests and sending them out.

- Parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks

Pattern 7: Producer-Consumer

Variant on the pipeline and client-server models.
Producer generates work, and consumer performs work.

Example: producer which generates rendered frames; consumer which orders these frames and writes them to disk.

Any number of producers and consumers.

- This approach can improve throughput and also reduces design complexity

Combining Strategies

Most problems don't fit into one category; it's often best to combine strategies.

For instance, you might often start with a pipeline, and then use multiple threads in a particular pipeline stage to handle one piece of data.

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate**.

- build system, e.g. parallel make
- optical character recognition system

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate**.

- build system, e.g. parallel make
  - Multiple independent tasks, at a per-file granularity
- optical character recognition system
  - Pipeline of tasks
  - 2 tasks - finding characters and analyzing them

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
- producer-consumer (no rendering frames, please):

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
  - Computation of a mathematical function with independent sub-formulas.
- producer-consumer (no rendering frames, please):
  - Processing of stock-market data: a server might generate raw financial data (quotes) for a particular security. The server would be the producer. Several clients (or consumers) may take the raw data and use them in different ways, e.g. by computing means, generating charts, etc.