Lecture 1 —Introduction

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

Department of Electrical and Computer Engineering
University of Waterloo

December 17, 2015

Course Syllabus

As our first order of business, let's go over the course syllabus.

Collaborative Course

The source material for the ECE 459 notes and slides is now open-sourced via Github.

If you find an error in the notes/slides, or have an improvement, go to `https://github.com/jzarnett/ece459` and open an issue.

If you know how to use `git` and LaTeX, then you can go to the URL and submit a pull request (changes) for me to look at and incorporate!

Performance!

I’m certain you know what “programming” means. But define “performance”.

Making a program “fast”.

Alright, but what does it mean for a program to be fast?

What is Fast?

Program execution as completion of some number of items – things to do.

We have two concepts:

(1) items per unit time (bandwidth – more is better)

(2) time per item (latency – less is better).

Improving on either of these will make your program “faster” in some sense.

Our Metrics

In a way they are somewhat related.

If we reduce the time per item from 5 s to 4 s it means an increase of 12 items per minute to 15 items per minute.

...if the conditions are right.

Hopefully we could improve both metrics; sometimes we'll have to pick one.

Bandwidth

This measures how much work can get done simultaneously.

Parallelization improves the number of items per unit time.

Sending a truck full of hard drives across the continent is high-bandwidth but also high-latency.

Latency

This measures how much time it takes to do any one particular task.

Also called response time.

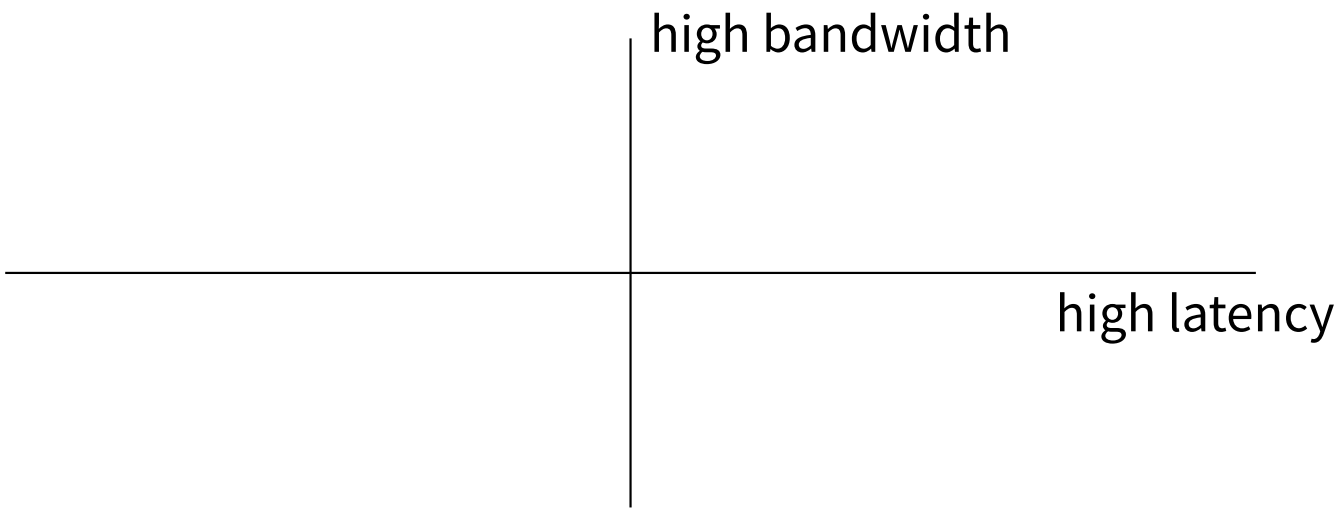
It doesn't tend to get measured as often as bandwidth, but it's especially important for tasks where people are involved.

Google cares, which is why they provide the `8.8.8.8` DNS servers.

Sadako and the Hundred Paper Airplanes

Say you need to make 100 paper airplanes. What's the fastest way of doing this?

Bandwidth vs Latency



We will focus on completing the items, not on transmitting information.

The above example makes the difference between bandwidth and latency clear.

Improving Latency

A good way of writing faster code is by improving single-threaded performance.

There is a limit to how much you can improve single-threaded performance.

Any improvements here may also help with the parallelized version.

On the other hand, faster sequential algorithms may not parallelize as well.

Profiling

You can’t successfully make your code faster if you don’t know why it’s slow.

Intuition seems to often be wrong here.

Run your program with realistic workloads under a profiling tool.

“Don’t guess; measure”.

Exercise Time!

Let's take a quick minute to visit `http://computers-are-fast.github.io/`

Self Assessment

Are the results surprising to you?

Did you do really well or really badly?

Chances are that you got some right and some wrong... and the ones that were wrong were not just a little wrong, but off by several orders of magnitude.

Moral of the story is: don’t just guess at what the slow parts of your code are.

It’s okay to have a theory as a starting point, but test your theory.

Do Less Work

A surefire way to be faster is to omit unnecessary work.

Two (related) ways of omitting work are:

(1) avoid calculating intermediate results that you don't actually need;

(2) compute results to only the accuracy that you need in the final output.

A Note About Logging

Producing text output to a log file or to a console screen is surprisingly expensive for the computer.

Sometimes one of the best ways to avoid unnecessary work is to spend less time logging and reporting.

It might make debugging harder, but code faster.

Caching

A hybrid between “do less work” and “be smarter” is caching.

Store the results of expensive, side-effect-free, operation (potentially I/O and computation) and reuse them as long as you know that they are still valid.

Caching is really important in certain situations.

Be Prepared

If you know something that the user is going to ask for in advance, you can have it at the ready to provide upon request.

Example: users want an Excel export of statistics on customs declarations.

Report generation takes a while, and it means a long wait.

Alternative: data pre-generated and stored in database (updated as necessary).

Then putting it into Excel is simple and the report is available quickly.

Be Smarter

You can also use a better algorithm.

This is probably “low hanging fruit” and by the time it’s time for P4P techniques this has already been done.

But if your sorting algorithm is $\Theta(n^3)$ and you can replace it with one that is $\Theta(n^2)$, it’s a tremendous improvement.

Even there are yet better algorithms out there.

Be Smarter

An improved algorithm includes better asymptotic performance as well as smarter data structures and smaller constant factors.

Compiler optimizations (which we'll discuss in this course) help with getting smaller constant factors.

We may also need to be aware of cache and data locality/density issues.

Checking out from the Library

Sometimes you can find this type of improvements in your choice of libraries.

Use a more specialized library which does the task you need more quickly.

It’s a hard decision sometimes.

Libraries may be better and more reliable than the code you can write yourself.

Or it might be better to write your own implementation that is optimized especially for your use case.

Throw Money at the Problem

Once upon a time, it was okay to write code with terrible performance on the theory that next year's CPUs would make it acceptably.

Spending a ton of time optimizing your code to run on today's processors was a waste of time.

Well, those days seem to be over; CPUs are not getting much faster these days (evolutionary rather than revolutionary change).

Spend Money to Save Money

What if the CPU is not the limiting factor: your code might be I/O-bound.
Buy some SSDs!

You might be swapping out to disk, which kills performance .
Add RAM.

Profiling is key here, to find out what the slow parts of execution are.

Spending a few thousand dollars on better hardware is often much cheaper than paying programmers to spend their time to optimize the code.

MOV R1,R3

What about outsmarting the compiler and writing assembly by hand?

This tends to be a bad idea these days.

Compilers are going to be better at generating assembly than you are.

Furthermore, CPUs may accept the commands in x86 assembly (or whatever your platform is) but internally they don't operate on those commands directly.

They rearrange and reinterpret and do their own thing.

Still, it's important to understand what the compiler is doing, and why it can't optimize certain things (we'll discuss that), but you don't need to do it yourself.

Anecdote Time

“The report generation has been running for three hours; I think it’s stuck.”

Nope, it reached a 30 minute time limit and got killed.

How do I speed up this task to get it under the 30 minute time limit?

Fly, My Pretties!

We can do more things at a time.

There are limits to how fast you can do each thing.

Often, it is easier to just throw more resources at the problem: use a bunch of CPUs at the same time.

We will study how to effectively throw more resources at problems.

In general, parallelism improves bandwidth, but not latency.

Unfortunately, parallelism does complicate your life, as we'll see.

Kinds of Parallelism

Different problems are amenable to different sorts of parallelization.

For instance, in a web server, we can easily parallelize simultaneous requests.

On the other hand, it's hard to parallelize a linked list traversal. (Why?)

Pipelining

A key concept is pipelining.

All modern CPUs do this, but you can do it in your code too.

Think of an assembly line: you can split a task into a set of subtasks and execute these subtasks in parallel.

Hardware

To get parallelism, we need to have multiple instruction streams executing simultaneously.

We can do this by increasing the number of CPUs: we can use multicore processors, SMP (symmetric multiprocessor) systems, or a cluster or machines.

We get different communication latencies with each of these choices.

We can also use more exotic hardware, like graphics processing units (GPUs).

Mo’ Parallelism, Mo’ Problems

You may have noticed that it is easier to do a project when it’s just you rather than being you and a team.

The same applies to code.

Here are some of the issues with parallel code.

Exceptions

Some domains are “embarassingly parallel”; these problems don’t apply.

It’s easy to communicate the problem to all of the processors and to get the answer back.

The processors don’t need to talk to each other to compute.

The canonical example is Monte Carlo integration.

Limitations

First, a task can’t start processing until it knows what it is supposed to process.

Coordination overhead is an issue.

If the problem lacks a succinct description, parallelization can be difficult.

Also, the task needs to combine its result with the other tasks.

Inherently Sequential Problems

"Inherently sequential" problems are an issue.

In a sequential program, it's OK if one loop iteration depends on the result of the previous iteration.

However, such formulations prohibit parallelizing the loop.

Sometimes we can find a parallelizable formulation of the loop, but sometimes we haven't found one yet.

Two Parts

Finally, code often contains a sequential part and a parallelizable part.

If the sequential part dominates, then executing the parallelizable part on infinite CPUs isn't going to speed up the task as a whole.

This is known as Amdahl's Law, and we'll talk about this soon.

To Avoid Complications...

It's already quite difficult to make sure that sequential programs work right.

Making sure that a parallel program works right is even more difficult.

The key complication is that there is no longer a total ordering between program events.

Instead, you have a partial ordering:

- Some events *A* are guaranteed to happen before other events *B*.
- Many events *X* and *Y* can occur in either the order *XY* or *YX*.

Races

A data race occurs when two threads or processes both attempt to simultaneously access the same data.

At least one of the accesses is a write.

This can lead to nonsensical intermediate states becoming visible.

Avoiding data races requires coordination between the participants to ensure that intermediate states never become visible (typically using locks).

Deadlock

Deadlock occurs when none of the threads or processes can make progress.

There is a cycle in the resource requests.

To avoid a deadlock, the programmer needs to enforce an ordering in the locks.

But Will it Scale?

It gets worse. Performance is great, but it's not the only thing.

We also care about scalability: the trend of performance with increasing load.

A program generally has a designed load (e.g., *x* transactions per hour).

A properly designed program will be able to meet this intended load.

If the performance deteriorates rapidly with increasing load (that is, the number of operations to do), we say it is not scalable.

Scaling Up

If we have a good program design it can be fixed.

If we have a bad program design, then no amount of programming for performance techniques are going to solve that (“rearranging deck chairs on the Titanic”).

Even the most scalable systems have their limits, of course, and while higher is better, nothing is infinite.