**ECE459: Programming for Performance** Winter 2016

# Lecture 1 — Programming for Performance

*Jeff Zarnett, based on original by Patrick Lam*

## Performance!

By this point, I'm certain you know what "programming" means, but we need to take a minute right off the top to define "performance". This course is not about how to program when other people are watching (fun as that can be, as the popularity of Hackathons shows). What it's really about is making a program "fast". Alright, but what does it mean for a program to be fast?

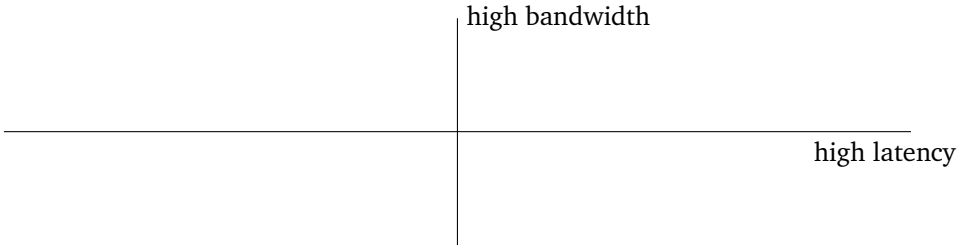
Let's think about the program execution as completion of some number of items – things to do. We have two concepts: items per unit time (bandwidth – more is better), and time per item (latency – less is better). Improving on either of these will make your program "faster" in some sense. In a way they 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, but sometimes we'll have to pick one.

**Items per unit time.** This measures how much work can get done simultaneously; we refer to it as bandwidth. Parallelization – or doing many things at once – improves the number of items per unit time. We might measure items per time in terms of transactions per second or jobs per hour. You might still have to wait a long time to get the result of any particular job, even in a high-bandwidth situation; sending a truck full of hard drives across the continent is high-bandwidth but also high-latency.

**Time per item.** This measures how much time it takes to do any one particular task: we can call this the latency or 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 a lot about latency, which is why they provide the `8.8.8.8` DNS servers.

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

Here's another example, containing various communications technologies:




We will focus on completing the items (doing useful work), not on transmitting information, but the above example makes the difference between bandwidth and latency clear.

## Improving Latency

Although we'll mostly focus on parallelism in this course, a good way of writing faster code is by improving single-threaded performance. Unfortunately, there will be a limit to how much you can improve single-threaded performance; however, any improvements here may also help with the parallelized version. On the other hand, faster sequential algorithms may not parallelize as well. But let's take a look at some ways you can improve latency.

**Profile the code.** You can't successfully make your code faster if you don't know why it's slow. Intuition seems to often be wrong here, so run your program with realistic workloads under a profiling tool and figure out where all the time is going. This is a specific instance of one of my favourite rules of engineering: "Don't guess; measure".

Let's take a quick minute to visit `http://computers-are-fast.github.io/` and take a quiz on how fast computers can do certain operations [**?**]. 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 to avoid calculating intermediate results that you don't actually need; and computing results to only the accuracy that you need in the final output.

Interesting to note: 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, yes, but once the code is correct (or close enough), removing the logging and debugging statements can actually make a difference.

A hybrid between "do less work" and "be smarter" is caching, where you store the results of expensive, side-effect-free, operations (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 from that other job of mine: users often want an Excel export of various statistics on their customs declarations. If the user asks for the report, generating it takes a while, and it means a long wait. If, however, the report data is pre-generated and stored in the database (and updated as necessary) then putting it in the Excel output file 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. 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, as does being aware of the cache and data locality/density issues.

Sometimes you can find this type of improvements in your choice of libraries: you might use a more specialized library which does the task you need more quickly. The build structure can also help, i.e. which parts of the code are in libraries and which are in the main executable. 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.

**Improve the hardware.** 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, and 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). But sometimes the CPU is not the limiting factor: your code might be I/O-bound, so you might be able to improve things dramatically by going to solid-state drives; or 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. When it comes down to it, spending a few thousand dollars on better hardware is often much cheaper than paying programmers to spend their time to optimize the code.

**On using assembly.** Not that long ago, compilers were not very smart and expert programmers could outsmart the compiler and produce better 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.** Recently I was presented with a ticket that read as follows: "the report generation has been running for three hours; I think it's stuck." Turns out the report had not been running for that long, it reached a 30 minute time limit and the server had killed the task (and it just looked like it was running). So now I have a puzzle: how do I speed up this task to get it under the 30 minute time limit?

How does the report work? It selects the transactions for a given period from the database. Then for each transaction, it looks up the latest article data, recomputes the transaction's worth based on the most up to date currency exchange rate, and then stores the updated transaction in the database again.

Step one was to bring up the profiler and look at a few things. The slow steps were primarily database operations: retrieving of exchange rates, retrieving the article data, and then storing all the transactions. Right, with this data, it's time to apply some strategies here.

Caching played a big role: the exchange rate data doesn't change for the report (it is run retroactively, with a date on the end of the last month, so the exchange rates are defined for that day rather than floating). So retrieving the exchange rate 500 times can be cut down to once per currency. Caching was also important for the articles; an article might be used dozens of times, so loading it from the database repeatedly is also a waste of time. Also, I could select all the articles at once rather than each one as encountered.

How about doing less work? For one thing, instead of pulling all the fields of the article from the database, why not just get the five that are actually needed? And in saving the transactions, what if we only update the parts that changed rather than update the full transaction and all its parts.

Ultimately, these techniques combined brought the report time down under 30 minutes and it can now run to completion.

## Doing more things at a time

Rather than, or in addition to, doing each thing faster, we can do more things at a time.

### Why parallelism?

While it helps to do each thing faster, there are limits to how fast you can do each thing. The (rather flat) trend in recent CPU clock speeds illustrates this point. 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.

**Different 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).

## Difficulties with using parallelism

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.

First, some domains are "embarassingly parallel" and these problems don't apply to them; for these domains, it's easy to communicate the problem to all of the processors and to get the answer back, and the processors don't need to talk to each other to compute. The canonical example is Monte Carlo integration, where each processor computes the contribution of a subrange of the integral.

I'll divide the remaining discussion into limitations and complications.

**Limitations.** Parallelization is no panacea, even without the complications that I describe below. Dependencies are the big problem.

First of all, a task can't start processing until it knows what it is supposed to process. Coordination overhead is an issue, and if the problem doesn't have a succinct description, parallelization can be difficult. Also, the task needs to combine its result with the other tasks.

"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.

Finally, code often contains a sequential part and a parallelizable part. If the sequential part takes too long to execute, then executing the parallelizable part on even an infinite number of processors 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.

**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 $B$, but many events $X$ and $Y$ can occur in either the order $XY$ or $YX$. This makes your code harder to understand, and complicates testing, because the ordering that you witness might not be the one causing the problem.

Two specific problems are data races and deadlocks.

- A *data race* occurs when two threads or processes both attempt to simultaneously access the same data, and at least one of the accesses is a write. This can lead to nonsensical intermediate states becoming visible to one of the participants. Avoiding data races requires coordination between the participants to ensure that intermediate states never become visible (typically using locks).
- A *deadlock* occurs when none of the threads or processes can make progress on the task because of a cycle in the resource requests. To avoid a deadlock, the programmer needs to enforce an ordering in the locks.

Another complication is stale data. Caches for multicore processors are particularly difficult to implement because they need to account for writes by other cores.

## Scalability

It gets worse. Performance is great, but it's not the only thing we're interested in. We also care about *scalability*: the trend of performance with increasing load. A program generally has a designed load (e.g., we are expecting to handle $x$ 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* [**?**]. This is undesirable, of course, and for the most part 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").

The things we're going to look at in this course are ways to meet $x$ or even raise the value of $x$. Even the most scalable systems have their limits, of course, and while higher is better, nothing is infinite. There's only so much we can do to push it, but chances are we can make some serious progress if we make the effort.