ECE459: Programming for Performance Winter 2013

Lecture 1 — January 8, 2013

Patrick Lam *version 1*

Making Programs Faster

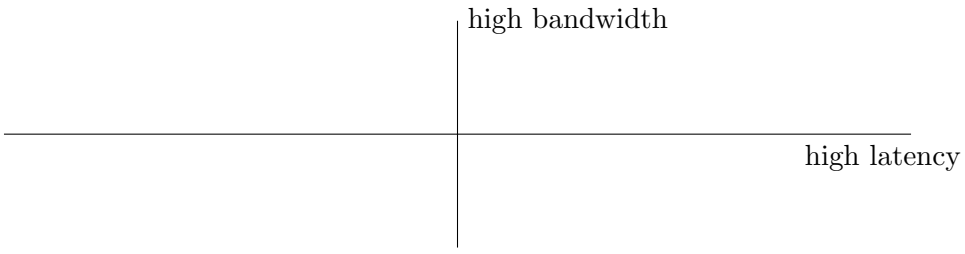
Let's start by defining what it means for programs to be fast. Basically, we have two concepts: items per unit time (bandwidth), and time per item (latency). Increasing either of these will make your program "faster" in some sense.

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 doing work, not on transmitting information, but the above example makes the difference between bandwidth and latency clear.

Improving latency: doing each thing faster

Although we'll be mostly focussing on parallelism in this course, a good way of writing faster code is by improving single-threaded performance. Unfortunately, there is often 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.) Here are 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. In particular, make sure that you're not losing a lot of time doing non-critical tasks, e.g. those that you didn't really need to do.

This can be remarkably difficult. For instance, I've hacked on the Soot program analysis framework[1], which has a terrible startup time, but I haven't been able to improve it.

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.

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.

Be smarter. You can also use a better algorithm. (I suspect that this isn't that important in a lot of code, but I have no evidence to support that.) Better algorithms include 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. Gove's book discusses this issue in what I feel is exhaustive detail.

Improve the hardware. While CPUs are not getting much faster these days, your code might be I/O-bound, so you might be able to win by going to solid-state drives; or you might be swapping out to disk, which kills performance (add RAM). Profiling is key here.

On using assembly. This tends to be a bad idea these days. Compilers are going to be better at generating assembly than you are. 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.

[1] `http://www.sable.mcgill.ca/soot`

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 simultanously. 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 dounf 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.