ECE459: Programming for Performance Winter 2015

Lecture 14 — February 4, 2015

Patrick Lam *version 0*

Back to parallelization patterns.

Pattern 5: Pipeline of Tasks. We've seen pipelining in the context of computer architecture. It can also work for software. For instance, you can use pipelining for packet-handling software, where multiple threads, as above, might confound the order. If you use a three-stage pipeline, then you can have three packets in-flight at the same time, and you can improve throughput by a factor of 3 (given appropriate hardware). Latency would tend to remain the same or be worse (due to communication overhead).

Some notes and variations on the pipeline: 1) if a stage is particularly slow, then it can limit the performance of the entire pipeline, if all of the work has to go through that stage; and 2) you can duplicate pipeline stages, if you know that a particular stage is going to be the bottleneck.

Pattern 6: Client-Server. Botnets work this way (as does `SETI@Home`, etc). To execute some large computation, a server is ready to tell clients what to do. Clients ask the server for some work, and the server gives work to the clients, who report back the results. Note that the server doesn't need to know the identity of the clients for this to work.

A single-machine example is a GUI application where the server part does the backend, while the client part contains the user interface. One could imagine symbolic algebra software being designed that way. Window redraws are an obvious candidate for tasks to run on clients.

Note that the single server can arbitrate access to shared resources. For instance, the clients might all need to perform network access. The server can store all of the requests and send them out in an orderly fashion.

The client-server pattern enables different threads to share work which can somehow be parcelled up, potentially improving throughput. Typically, the parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks. It's also a design pattern that's easy to reason about.

Pattern 7: Producer-Consumer. The producer-consumer is a variant on the pipeline and client-server models. In this case, the producer generates work, and the consumer performs work. An example is a producer which generates rendered frames, and a consumer which orders these frames and writes them to disk. There can be any number of producers and consumers. This approach can improve throughput and also reduces design complexity.

Combining Strategies. If one of the patterns suffices, then you're done. Otherwise, you may need 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. Or, as I alluded to earlier, you can replicate pipeline stages to handle different data items simultaneously.

Note also that you can get synergies between different patterns. For instance, consider a task which takes 100 seconds. First, you take 80 seconds and parallelize it 4 ways (so, 20 seconds). This reduces the runtime to 40 seconds. Then, you can take the serial 20 seconds and split it into two threads. This further reduces runtime to 30 seconds. You get a 2.5× speedup from the first transformation and 1.3× from the second, if you do it after the first. But, if you only did the second parallelization, you'd only get a 1.1× speedup.

How to Parallelize Code

Here's a four-step outline of what you need to do.

1. Profile the code.
2. Find dependencies in hotspots. For each dependency chain in a hotspot, figure out if you can execute the chain as multiple parallel tasks or a loop over multiple parallel iterations. Think about changing the algorithm, if that would help.
3. Estimate benefits.
4. If they're not good enough (e.g. far from linear speedup), step back and see if you can parallelize something else up the call chain, or at a higher level of absraction. (Think Mandelbrot sets and computing different points in parallel).

Try to reduce the amuont of synchronization that you have to do (waiting for parallel tasks to finish), because that always slows you down.

Thread Pools

We talked about "single task, multiple threads" last week. The idea behind a *thread pool* is that it's relatively expensive to start a thread; it costs resources to keep the threads running at the operating system level; and the threads won't run optimally anyway, because they'll spend too much time swapping state in and out of the cache. Instead, you start an appropriate number of threads, which each grab work from a work queue, do the work, and report the results back. Web servers are a good application of thread pools[1].

A key question is: how many threads should you create? This depends on which resources your threads use; if you are writing computationally-intensive threads, then you probably want to have fewer threads than the number of virtual CPUs. You can also use Amdahl's Law to estimate the maximum useful number of threads, as discussed previously.

Here's a longer discussion of thread pools:

`http://www.ibm.com/developerworks/library/j-jtp0730.html`

[1] Apache does this: `http://httpd.apache.org/docs/2.0/mod/worker.html`. Also see an assignment where the students write thread-pooled web servers: `http://www.cse.nd.edu/~dthain/courses/cse30341/spring2009/project4/project4.html`.

Modern languages provide thread pools; Java's `java.util.concurrent.ThreadPoolExecutor`[2], C#'s `System.Threading.ThreadPool`[3], and GLib's `GThreadPool`[4] all implement thread pools.

GLib. GLib is a C library developed by the GTK team. It provides many useful features that you might otherwise have to implement yourself in C. Consider using GLib's thread pool in your assignment 2, unless you want to implement the work queue yourself. (I see no reason to do that!)

Automatic Parallelization

We'll now talk about automatic parallelization. The vision is to take your standard sequential C program and convert it into a parallel C program which leverages multiple cores, CPUs, machines, etc. This was an active area of research in the 1990s, then tapered off in the 2000s (because it's a hard problem!); it is enjoying renewed interest now (but it's still hard!)

What can we parallelize? The easiest kind of program to parallelize is the classic Fortran program which performs a computation over a huge array. C code—if it's the right kind—is a bit worse, but still tractable, given enough hints to the compiler. For us, the right kind of code is going to be array codes. Some production compilers, like the non-free Intel C compiler `icc`, the free-as-in-beer Solaris Studio compiler[5], and the free GNU C compiler `gcc`, include support for parallelization, with different maturity levels.

I did some live coding in class, but let's just move all of that discussion to Lecture 10, which will contain a unified treatment of parallelizing the example.

[2] `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html`
[3] `http://msdn.microsoft.com/en-us/library/3dasc8as%28v=vs.80%29.aspx`
[4] `http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html`
[5] `http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html`