ECE459: Programming for Performance Winter 2011

Lecture 24 — March 31, 2011

Patrick Lam *version 1*

Course Summary

The main theme in this course has been leveraging parallelism to improve performance. Here's a quick, nonexhaustive summary of what we've seen.

Bandwidth and Latency. We started off by looking at what it means to program for performance. Almost all of the techniques we explore in this class improve performance by improving bandwidth, in particular by running code on parallel hardware. We've touched on a few ways to improve latency; for instance, we used a faster, approximate algorithm in Assignment 4. Unfortunately, there doesn't tend to be a generic way to improve latency.

Profiling. To productively optimize code, you've got to know what to optimize, so we've returned to the topic of profiling a number of times in this course; in particular, I asked you to do it in Assignments 1 and 3. We also studied tools for profiling, including the venerable `gprof` as well as newer `oprofile`-style system-level tools. I also talked about DTrace, which enables you to write custom queries about system behaviour, and particularly about system performance.

Amdahl's Law, Gustafson's Law. The problem with parallelization is that there's always some sort of serial part. We saw Amdahl's Law, which estimates limits to speedup caused by serial parts, and Gustafson's Law, which points out that you can use parallelization to solve bigger problems.

Parallelism Implementations/Modern Hardware. Moving on, we started talking about how modern commodity computers make parallelism available through virtual CPUs and by using multiple cores. There was also a refresher on caches and TLBs, which greatly affect performance.

Parallelization Patterns. The next topic was a number of design patterns for leveraging parallelism, including pipelines and producer-consumer designs. Recall the difference between data parallelism (which especially dominates on GPUs, but also appears in some of the other parallelism constructs we've seen) and task parallelism (as in OpenMP tasks).

Dependencies and speculation. The main barrier to parallelizing everything is dependencies in the code: some code relies on other code having previously executed (or not having executed yet). We saw the different types of potentially problematic dependencies: RAW, WAR and WAW. Recall that WAR and WAW can be eliminated with renaming.

Speculation is another way to break dependencies; if you can predict the value that you're going to read, then you can get a head start on a future calculation. Architectures do this extensively, and you can predict in software as well.

Race conditions and synchronization primitives. Parallel programmers worry about races, simultaneous accesses to the same resource without proper locking discipline. I gave an example of a race condition, and recalled various synchronization primitives, like locks and barriers.

Paralllelization, including automatic parallelization. I gave a quick `pthreads` refresher (+ thread pools) and gave a broad overview of how to parallelize code. Then I talked about how and when compilers will automatically parallelize code for you. In Assignment 3, you looked at both automatic parallelization and manual parallelization, the latter using explicit OpenMP directives.

OpenMP. Typically, OpenMP is going to be a better bet than relying on automatic parallelization. It was initially designed for FORTRAN loops, but you can also use it for C programs as well. OpenMP tasks also go beyond the old loop-based model. You have to be careful to make sure that the code doesn't contain dependencies between loop iterations.

Memory consistency and memory barriers. One way to deal with unwanted dependencies is to introduce memory barriers; they state that reads or writes can't migrate beyeond the memory barrier. Without memory barriers, architectures and compilers can reorder memory accesses and produce surprising code. (This is not the same as the concept of barrier synchronization for threads).

Compiler optimizations. Speaking of compilers, we ran through a list of compiler optimizations. During compilation, the compiler performs sophisticated analyses on the program to enable semantics-preserving optimizations. The simpler your code, the more likely that the compiler can optimize. The compiler is pretty good at figuring out what to inline, especially when you run it in profile-guided mode. One optimization that people still do by hand sometimes is loop unrolling, which gives the compiler other optimization opportunities and can also enable SIMD parallelization.

GPU Programming. It's now relatively easy to program GPUs using OpenCL (or CUDA, for that matter). We saw some example code and talked about the principles behind coding for GPUs (i.e. you have to move data yourself when you want it). You practiced porting some N-body simulation code to GPUs in Assignment 4.

Distributed Systems: clusters, MPI, and MapReduce. Moving beyond one computer, we talked about cluster setups (including compute clouds). Distributed systems go further in the message-passing versus shared-memory continuum, and communication between nodes is now quite high-latency. MPI is a way to program distributed systems. MapReduce is another paradigm.

High-Performance Languages. Finally, we looked at three languages being designed specifically for massively parallel hardware: X10, Fortress and Chapel; and surveyed their HPC features.