ECE459: Programming for Performance Winter 2011

Lecture 5 — January 18, 2011

Patrick Lam *version 1*

We've talked (a bit) about modern processors and profiling. The major topic for the rest of the course is parallelization: how do we leverage the multicore aspect of modern processors to increase throughput? First, though, we'll talk about how far parallelization can take us.

Limits to parallelization

I mentioned briefly in Lecture 1 that programs often have a sequential part and a parallel part. We'll quantify this observation today and discuss its consequences.

Amdahl's Law. One classic model of parallel execution is Amdahl's Law. In 1967, Gene Amdahl argued that improvements in processor design for single processors would be more effective than designing multi-processor systems. Here's the argument. Let's say that you are trying to run a task which has a serial part, taking time S, and a parallelizable part, taking time P, then the total amount of time T_s on a single-processor system is:

$$T_s = S + P.$$

Now, moving to a parallel system with N processors, the parallel time T_p is instead:

$$T_p = S + \frac{P}{N}.$$

As N increases, T_p is dominated by S, limiting the potential speedup.

We can restate this law in terms of speedup, which is generally the original time T_s divided by the sped-up time T_p:

$$S_p = \frac{T_s}{T_p}.$$

If we let f be the parallelizable fraction of the computation, i.e. set f to P/T_s, and we let S_f be the speedup we can acheive on f, then we get[1]:

$$S_p(f, S_f) = \frac{1}{(1-f) + \frac{f}{S_f}}.$$

Plugging in numbers. If $f = 1$, then we can indeed get good scaling, since $S_p = S_f$ in that case; running on an N-processor machine will give you a speedup of N. Unfortunately, usually $f < 1$. Let's see what happens.

[1] `http://www.cs.wisc.edu/multifacet/amdahl/`

f	$S_p(f, 18)$
1	18
0.99	15
0.95	10
0.5	2

To get the 2× speedup for $f = 0.5$, you'd need to use 8 cores.

Amdahl's Law tells you how many cores you can hope to leverage in your code, if you can estimate f. I'll leave this as an exercise to the reader: fix some f and set a tolerance, i.e. you don't care about a speedup less than x. Use the equations to figure out how many cores give that speedup.

The graph looks like this:

To get reasonable parallelization with a tolerance of 1.1, f needs to be at least 0.8.

Consequences of Amdahl's Law. For over 30 years, most performance gains did indeed come from increasing single-processor performance. The main reason that we're here today is that, as we saw in the video in Lecture 2, single-processor performance gains have hit the wall.

By the way, note that we didn't talk about the cost of synchronization between threads here. That can drag the performance down even more.

A more optimistic point of view

In 1988, John Gustafson pointed out[2] that Amdahl's Law only applies to fixed-size problems, but that the point of computers is to deal with bigger and bigger problems.

In particular, you might vary the input size, or the grid resolution, number of timesteps, etc. When running the software, then, you might need to hold the running time constant, not the problem size: you're willing to wait, say, 10 hours for your task to finish, but not 500 hours. So you can change the question to: how big a problem can you run in 10 hours?

According to Gustafson, scaling up the problem tends to increase the amount of work in the parallel part of the code, while leaving the serial part alone. As long as the algorithm is linear, it is possible to handle linearly larger problems with a linearly larger number of processors.

Of course, Gustafson's Law works when there is some "problem-size" knob you can crank up. As a practical example, observe Google, which deals with huge datasets.

[2] `http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html`

Multicore Processors

Let's continue on that optimistic note and talk about multicore processors. As I've alluded to earlier, they came about because clock speeds just aren't going up anymore. We'll discuss technical details today.

Each processor *core* executes instructions; a processor with more than one core can therefore simultaneously execute multiple (unrelated) instructions.

Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
plam@plym:~/courses/p4p/lectures$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
...
processor : 1
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
```

If you actually opened my computer, though, you'd only find one chip. The chip is pretending to have two *virtual CPUs*, and the operating system can schedule work on each of these CPUs. In general, you can't look at the chip and figure out how many cores it contains.

Threads and CPUs. In your operating systems class, you've seen implementations of threads ("lightweight processes"). We'll call these threads *software threads*, and we'll program with them throughout the class. Each software thread corresponds to a stream of instructions that the processor executes. On a old-school single-core, single-processor machine, the operating system multiplexes the CPU resources to execute multiple threads concurrently; however, only one thread runs at a time on the single CPU.

On the other hand, a modern chip contains a number of *hardware threads*, which correspond to the virtual CPUs. These are sometimes known as *strands*. The operating system still needs to multiplex the software threads onto the hardware threads, but now has more than one hardware thread to schedule work onto.

What's the term for swapping out the active thread on a CPU?

Next time, we'll talk about processor design, caches, and translation look-aside buffers, as they affect parallel programming.