CS 521: Systems Programming

Measuring Performance

Lecture 12

A Need for Speed

- Single-threaded CPU performance has been somewhat stagnant over the past years
- Most gains are being made in:
  - Power efficiency
  - Increased hardware efficiency (more instructions per *clock cycle*)
  - Adding more cores
- The last approach doesn't work unless we (as programmers) take advantage of those cores

Today's Reality

- These days, we get better performance through horizontal scaling
  - Rather than making one really fast processor (**vertical** scaling), we'll make a processor running at a reasonable speed with multiple cores
- And in some cases, we'll have multiple processors, or even multiple machines
  - Clusters

Examples

- Google doesn't buy the world's craziest, most expensive supercomputers
  - They buy **commodity hardware** in huge quantities
  - A single Google search may query tens or even hundreds of servers
- AMD's Ryzen/EPYC CPUs push multicore further
  - Let's say we want to build a 128-core CPU
    - The chance of a manufacturing defect is high
  - AMD's latest approach: take four 32-core CPUs and fuse them together (multi-chip module)

Who Cares About Performance?

- Mostly, this slowdown in CPU advancement is okay:
  - Most applications run "fast enough"
  - I don't have to buy a new laptop every year!
- But some use cases still need more power:
  - Climate models, large-scale, more realistic simulations
  - Machine Learning (deep learning)
  - Bioinformatics
  - Games! VR requires massive computational capabilities

Parallelism [1/2]

- Adding more cores is only helpful if we can actually make use of them!
- The bad news: parallelizing applications can be hard
- The basic idea behind parallel computing is:
  Divide and Conquer
- If we can split a problem up into many smaller problems, then each core (or each machine) can take care of part of the work

Parallelism [2/2]

- Some problems are **embarrassingly parallel**
  - If I told everyone to raise their hands, no coordination is necessary
- Unfortunately, not all problems are as easy to parallelize
  - Communication is required between CPUs and machines
    - This is where things get tricky

Measuring Parallel Performance

- There are two common metrics for measuring the performance of our parallel algorithms:
  - Speedup
  - Parallel Efficiency
- Evaluating these is crucial: if we're not gaining anything from parallelism, there's no reason to do it
- A closely related concept is **scalability**
  - How our algorithm performs when we give it more resources

Terminology

- We're kinda-sorta familiar with CPU ***cores***, right?
- A 4-core CPU means you can do… four things at once (at least in theory – the truth is way more complicated)
- For our purposes:
  - processor = core = processing element (PE) = rank
- We don't care ***too*** much about the details, as long as we can have these processors do work for us

Speedup

- The speedup of a parallel program is given by:
  - $S = \frac{T_{serial}}{T_{parallel}}$
- Or: how long the serial (original, non-parallel) program takes to run divided by the parallel run time
- Best speedup possible: $S = p$
  - Where p is the number of processors
  - This is fairly intuitive. If you add 100 processors, you'd hope for a 100x speedup.

Efficiency

- The parallel efficiency of a program is given by:
  - $E = \frac{S}{p} = \frac{T_{serial}}{pT_{parallel}}$
- The speedup divided by the number of processors
- Best efficiency possible: 1

Scalability [1/2]

- It's possible to write an algorithm that has high efficiency on two, four, eight cores, **but**:
  - Maybe when you try to run it on 16 cores efficiency starts to drop
- There are many reasons this can happen
  - Your algorithm requires a lot of communication
  - Your processes spend a lot of time blocked
  - In some cases, you're only as fast as the slowest worker

Scalability [2/2]

- A program that scales can use additional resources effectively
  - Doubling the number of processes should halve the execution time!
- We can measure this by calculating parallel efficiency
- If efficiency decreases as we add processes, then the algorithm is not scalable

Keeping Track of Time

- We've actually done a little bit of timing already this semester
  - The `time` command
- Not fine-grained
  - We can only test how long it takes to run the entire program
  - What happens when we prompt for a value? What about application startup time (from the OS)?
- We need to be able to track things at a finer level

The clock function (avoid!)

- C includes a clock function that tells us the **number of clock ticks** since the program started
  - Originally intended to be the number of CPU cycles but is implementation-specific
- Different hardware has different clock resolutions
  - Often `clock()` is a fairly low-resolution timer
- To translate the abstract notion of clock ticks into time, we can use `CLOCKS_PER_SEC`:
  - `clock() / CLOCKS_PER_SEC`

gettimeofday

- On Unix-based systems, this function provides the wall clock time, generally with $1\ \mu s$ precision
  - Note that this is also hardware dependent
- Wall clock time: the actual time taken for something to run
  - As opposed to CPU time
  - Check the output of `top` for CPU time
- Usually a much better option than clock(), if you have it
- `#include <timer.h>`

Timing

```
double get_time(void)
{
    struct timeval t;
    gettimeofday(&t, NULL);
    // Convert to ms before returning:
    return t.tv_sec + t.tv_usec / 1000000.0;
}
```

Estimating π

- To measure performance, we need something CPU-intensive to do
  - Printing 'hello world' ain't gonna cut it
- How about estimating π? Sure, we could just look it up and copy several digits into our code as a constant…
  - But that sounds **much** too easy. This is CS 521!
- Instead, we can do this with something called the *Madhava–Leibniz series*:
  - $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4}$

Madhava-Leibniz

- $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4}$
- The more iterations, the more accurate our estimate gets…
  - …however, please note that this is **NOT** a particularly efficient method
    - We're just using it to make our CPUs burn
- We can split these iterations up across multiple processes to speed things up
  - pthreads to the rescue! We can look at a few approaches…