Measuring Performance

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Measuring Performance

- At this point in the semester we have already discussed performance issues having to do with concurrency quite a bit
- And we have measured performance for some of the code we developed
  - Typically in “seconds elapsed”
- In this module we’ll talk about performance issues that are important and not necessarily connected to concurrency
  - i.e., improving the performance of a single thread is important in a multi-threaded program
- But first, let’s talk a bit about performance measures

Running Fast is Good

- One almost always wants programs that run fast
  - Lower the “time to result” for one run
  - Increase the “number of runs one can do in a day”
- This can be achieved in three main ways:
  - Get faster hardware
    - Not always successful / possible
  - Use judicious data structures and algorithms (ICS?11)
    - e.g., don’t do a linear search on a sorted array, use a binary search
    - e.g., use a heap instead of a linked list
  - Produce “fast code”
    - e.g., use multi-threading
    - e.g., use performance enhancing “tricks”
- A key question: How much effort/money am I willing to put in for how much of an increase in performance?
  - Depends on who you are and what you do

Fast Hardware?

- Microprocessor manufacturers market processors based on cost, performance, and power efficiency
- One measure is peak performance
  - Computed based on specifications
  - For instance:
    - I build a machine with 4 floating point arithmetic units
    - Each unit can do an operation in 2 cycles
    - My CPU is at 1GHz
    - Therefore I have a processor that can compute 2 Million of floating point operations per second

Peak vs. Real Performance

- Peak performance can never be reached
  - For instance, in the example in the previous slide, data to be computed on has to be brought from memory to the processor, meaning that do do anything useful you can’t do it at peak performance
- But it’s a coarse measure often used to compare computer systems
- Let’s look at the Top500 site: https://top500.org
  - Latest list
  - RPeak: Peak Performance in PFlop/sec
  - Rmax: Maximum performance achieved by the Linpack Benchmark in PFlop/sec
    - Linear System Solve

Benchmarks

- Since peak performance can be misleading, one typically uses benchmarks
  - We just saw the LINPACK benchmark used for the Top500 list
- There are many popular benchmarks, some of which you may have heard of
  - These benchmarks are typically a collection of several codes that come from ore are inspired by “real-world software”
  - They measure integer performance, floating point performance, memory performance, gaming performance, etc.
- The question “what is a good benchmark?” is difficult
  - If the benchmark does not correspond to what you’ll do with the computer once you purchase it, then the benchmark results are not relevant to you

Program Performance

- In this class, we haven’t focused on determining the performance of a computer (whichever way it is defined)
- Instead, we have assumed that you have a computer
  - e.g., purchased based on peak performance considerations, purchased based on published benchmark results, purchased based on cost, purchased randomly
- You have a program that you need to run on this computer, and you want to measure the program’s performance
- First question: What are performance measures?

Elapsed Time

- The only measure we have used in this class so far is elapsed time
- The most commonly understood performance measure
  - Also called wall-clock time, running time, response time, execution time, execution latency, makespan, …
- Allows us to make statements like: “My program runs in 45.12 seconds on a 2.5GHz Intel Core i7”

Performance as a Rate

- Measure how many “things” the code did, and divide by the elapsed time
- Millions of Instructions per Seconds (MIPS)
  - But not all instructions are equal on different processor families, so this can be misleading
- Millions of Floating Point Operations per Seconds (Mflops)
  - Used routinely for “scientific” applications
  - That was the metric we saw on the Top500 Web site
- Application-specific:
  - Example: “My program renders 42.4 movie frames per second”
  - For our ics432imgapp code, this could be number of oil painting filtered 200x200 jpeg files per second

How to Time Program Execution

- All previous performance metrics involve taking time measurements
- The most basic approach is the `time` UNIX command
- It Reports:
  - Elapsed time
  - User time (time spent in user code)
  - System time (time spent in kernel code)
- It has a pretty coarse resolution, but is available everywhere
- You saw it in ICS332

The perf Command on Linux

- The **`perf`** command works pretty much like **`time`**, but has much better resolution (but it doesn't differentiate user/system times)
  - `sudo apt install linux-tools-common linux-tools-generic`
- Usage: **`perf stat <some command>`**
- Reports:
  - msec of execution, total clocks cycles, page-faults, CPU stalls, total number of instructions, etc.
- Let's try it (not on a VM)…

Timing Sections of Code

- Timing the whole execution is not always what you want
- When you're working on a small section of the code, you want to see the time taken only in that section
  - So that you can measure small improvements that you make, which would "disappear" in the whole elapsed time
- This is doable in all programing languages, using a "tell me what time it is now" function?

Timing a Section of Code

```
double begin = get_time();
for (int i=0; i < 100; i++) {
    render_frame(i);
}
double stop = get_time();
double elapsed_time = stop - begin;
double frame_render_rate = i/elapsed_time;
```

Timing in C: clock()

- All UNIX-like systems have it
- It’s easy, but does not have great resolution
- Example:

```
#include <time.h>
clock_t begin = clock();
// some section of code
clock_t end = clock();
double elapsed_time_in_seconds = (double)(end
- begin) / CLOCKS_PER_SEC;
```

Timing in C: gettimeofday()

- All UNIX-like systems have it
- Better resolution than clock(), but still not great
  - Sufficient for our homework
- Example:

```
#include <sys/time.h>
struct timeval begin, end;
gettimeofday(&begin, NULL);
// some section of code
gettimeofday(&end, NULL);
double elapsed_time_in_seconds =
      (10.0E+6 * (end.tv_sec - begin.tv_sec) +
      (end.tv_usec - begin.tv_usec)) / 10.0E+6;
```

Timing in C: clock_gettime

- Available on Linux (not other UNIX-like OSes), better resolution than previous ones

```
#include <time.h>
struct timespec begin, end;
clock_gettime(CLOCK_MONOTONIC, &begin);
// some section of code
clock_gettime(CLOCK_MONOTONIC, &end);
double elapsed_time_in_seconds =
      (10.0E+9 * (end.tv_sec - begin.tv_sec) +
      (end.tv_nsec - begin.tv_nsec)) / 10.0E+9;
```

Timing in Java: System.currentTimeMillis()

```
#include <time.h>
long begin = java.lang.System.currentTimeMillis();
// some section of code
long end = java.lang.System.currentTimeMillis();
double elapsed_time_in_seconds =
      (end - begin) / 10.0E+3;
```

- Known to not have great resolution

Timing in Java: System.nanoTime()

```
#include <time.h>
long begin = java.lang.System.nanoTime();
// some section of code
long end = java.lang.System.nanoTime();
double elapsed_time_in_seconds =
      (end - begin) / 10.0E+9;
```

- Designed to be used as a timer (not as a meaningful way to ask “what time is it?”)
- Not accurate to the nano second (more like microseconds)
- Not thread-safe

Good Performance?

- You have a code that was given to you or that you wrote yourself
- You compile it with your favorite compiler
- And you get some performance measure
- You can compare this to the peak performance, and you’re typically really far

- How can performance be improved?

Performance Bottlenecks

- A running application is a system that has many components
  - Hardware: CPU cores, memory, memory bus, caches, network card, drive, video card, etc.
  - Software: methods you wrote, libraries you use, operating system
- Pick a component and “magically” make it faster, if the application performance increases, that component was a performance bottleneck
- Now, because we don’t have magic, we have to reason about the execution find out what’s a bottleneck (there are tools)

Removing a Bottleneck

- Brute force: Hardware Upgrade
  - Sometimes necessary, but can only get you so far and may be costly
    - e.g., memory technology

- Software upgrade
  - The bottleneck is there because the code uses a “resource” heavily, or in non-intelligent manner, or is simply not well-written

Identifying a Bottleneck

- The first step in identifying a bottleneck is to determine which part of the code takes a lot of time
- How do we know which part of the code takes the most time?
  - If you've not written the code you may not know at all
  - If you've written the code you may have some idea (although experience shows that most programmers don't)
  - The most expensive part may be in some library function you haven't written and may not even know about!
- You could put clock_gettime() calls everywhere, but that gets really cumbersome for large projects
- The standard approach: use a profiler
- Show of hands: who has used a profiler before?

What is a Profiler?

- A profiler is a tool that monitors the execution of a program and that reports the amount of time spent in different functions
- Every "serious" language has profilers (Java, Python, etc.)
  - Some free, some commercial
- On UNIX/Linux there are several options for compiled code
  - gprof: the "dinosaur"
    - basic, limited, flawed, but still used widely
  - valgrind/callgrind
    - much more evolved (nice visualization), but runs slowly
  - gperftools from Google
    - gained a lot in popularity
- Java has many profilers, Python includes a profiler (cProfile)
- Information on how to install/run the above is all over the Web

Let’s Run gperftools

- I’ve installed gperftools in a Docker image
- Let’s download a ray-tracer: https://github.com/PurpleAlien/Raytracer/archive/refs/heads/master.zip
  - Modify the Makefile to link with -lprofiler and add the -g compiler flag
  - Run it as: LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=prof.out ./raytrace scene.txt
  - Export the output to format X: pprof --X <absolute path to executable> prof.out > output.X
    - Let’s do this for X=text and X=pdf output…

The “code acceleration” Loop

- Run the profiler
- Identify functions that are time-consuming
  - Accelerating a function that accounts for 1% of the execution time will have little impact
- Modify the code to make these functions faster
  - Modify their code
  - Call them less
  - Or make the call that a hardware upgrade would help
- Repeat until you’ve run out of ideas :)

Conclusion

- In this course we have not done a "make a random piece of code" faster
  - I used to do that a long time ago in this course!
- Instead we've implemented all our own code, and it's pretty clear which part needs optimization (image filtering!)
  - And our code is really small anyway
- So we have not used a profiler
  - Even though we can profile our code at any time of course, using a Java-specific profiler
- But you shouldn't graduate without knowing what a profiler is!