

Measuring Performance

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Performance as Time

- Time between the start and the end of an operation
 - Also called running time, elapsed time, wall-clock time, response time, latency, execution time, ...
 - Most straightforward measure: “my program takes 12.5s on a Pentium 3.5GHz”
 - Can be normalized to some reference time
- Must be measured on a “dedicated” machine

Performance as Rate

- Used often so that performance can be independent on the “size” of the application
 - e.g., compressing a 1MB file takes 1 minute. compressing a 2MB file takes 2 minutes. The performance is the same.
- Millions of instructions / sec (MIPS)
 - $\text{MIPS} = \text{instruction count} / (\text{execution time} * 10^6)$
= clock rate / (CPI * 10^6)
 - But Instructions Set Architectures are not equivalent
 - 1 CISC instruction = many RISC instructions
 - Programs use different instruction mixes
 - May be ok for same program on same architectures

Performance as Rate

- Millions of floating point operations /sec (MFlops)
 - Very popular, but often misleading
 - e.g., A high MFlops rate in a stupid algorithm could have poor application performance
- Application-specific:
 - Millions of frames rendered per second
 - Millions of amino-acid compared per second
 - Millions of HTTP requests served per second
- Application-specific metrics are often preferable and others may be misleading
 - MFlops can be application-specific though
 - For instance:
 - I want to add two n-element vectors
 - This requires n Floating Point Operations
 - Therefore MFlops is a good measure

Measuring Performance Rates

- How do we measure performance rates?
- Time a section of code
- Count how many “items” are done in that section of the code
- Compute the rate as the number of items divided by the measured time
- Example:

```
start_timer(...)
for (i=0; i<1000000; i++)
  x = y * z + a
stop_timer(...)
```

 - Number of MFlop: 2 (1000000 additions, 1000000 multiplications)
 - Number of MFlops: 2 / time

“Peak” Performance?

- Resource vendors always talk about **peak performance rate**
 - computed based on specifications of the machine
 - For instance:
 - I build a machine with 2 floating point units
 - Each unit can do an operation in 2 cycles
 - My CPU is at 1GHz
 - Therefore I have a $1 * 2 / 2 = 1\text{GFlops}$ Machine
 - Problem:
 - In real code you will never be able to use the two floating point units constantly
 - Data needs to come from memory and cause the floating point units to be idle
- Typically, real code achieves only an (often small) fraction of the peak performance

Benchmarks

- Since many performance metrics turn out to be misleading, people have designed benchmarks
- Example: SPEC Benchmark
 - Integer benchmark
 - Floating point benchmark
- These benchmarks are typically a collection of several codes that come from “real-world software”
- The question “what is a good benchmark?” is difficult
 - If the benchmarks do not correspond to what you'll do with the computer, then the benchmark results are not relevant to you

How About GHz?

- This is often the way in which people say that a computer is better than another
 - More instruction per seconds for higher clock rate
- Faces the same problems as MIPS

Processor	Clock Rate	SPEC FP2000 Benchmark
IBM Power3	450 MHz	434
Intel PIII	1.4 GHz	456
Intel P4	2.4GHz	833
Itanium-2	1.0GHz	1356

- But usable within a specific architecture

Program Performance

- In this class we're not really concerned with determining the performance of a compute platform (whichever way it is defined)
- Instead we're concerned with improving a program's performance
 - For a given platform, take a given program
 - Run it and measure its wall-clock time
 - Enhance it, run it and quantify the performance improvement
 - i.e., the reduction in wall-clock time
 - For each version compute its performance
 - preferably as a relevant performance rate
 - so that you can say: the best implementation we have so far goes “this fast” (perhaps a % of the peak performance)

The UNIX `time` Command

- You can put `time` in front of any UNIX command you invoke
- When the invoked command completes, `time` prints out timing (and other) information

```
% time ls /home/casanova/ -la -R
0.520u 1.570s 0:20.58 10.1% 0+0k 570+105io 0pf+0w
```

- 0.520u 0.52 seconds of user time
- 1.570s 1.57 seconds of system time
- 0:20.56 20.56 seconds of wall-clock time
- 10.1% 10.1% of CPU was used
- 0+0k memory used (text + data)
- 570+105io 570 input, 105 output (file system I/O)
- 0pf+0w 0 page faults and 0 swaps

User, System, Wall-Clock?

- User Time: time that the code spends executing user code (i.e., non system calls)
- System Time: time that the code spends executing system calls
- Wall-Clock Time: time from start to end
- Wall-Clock \geq User + System
 - in our example: $20.56 \geq 0.52 + 1.57$
- Why?
 - because the process can be suspended by the O/S due to contention for the CPU by other processes
 - because the process can be blocked waiting for I/O

Using `time`

- It's interesting to know what the user time and the system time are
 - for instance, if the system time is really high, it may be that the code does too many calls to `malloc()`, for instance
 - But one would really need more information to fix the code (not always clear which system calls may be responsible for the high system time)
- Wall-clock - system - user \approx I/O + suspended
 - If the system is **dedicated**, suspended \approx 0
 - Therefore one can estimate the cost of I/O
 - If I/O is really high, one may want to look at reducing I/O or doing I/O better
- Therefore, time can give us insight into bottlenecks and gives us wall-clock time

Drawbacks of UNIX `time`

- The `time` command has poor resolution
 - “Only” milliseconds
 - Sometimes we want a higher precision, especially if our performance improvements are in the 1-2% range
- `time` times the whole code
 - Sometimes we’re only interested in timing some part of the code, for instance the one that we are trying to optimize
 - Sometimes we want to compare the execution time of different sections of the code

Timing with `gettimeofday`

- `gettimeofday` from the standard C library
- Measures the number of microseconds since midnight, Jan 1st 1970, expressed in seconds and microseconds

```
#include <sys/time.h>
struct timeval start;
...
gettimeofday(&start, NULL);
printf(“%ld,%ld\n”, start.tv_sec, start.tv_usec);
...
```

- Can be used to time sections of code
 - Call `gettimeofday` at beginning of section
 - Call `gettimeofday` at end of section
 - Compute the time elapsed in microseconds
 - e.g., $(\text{end.tv_sec} * 1000000.0 + \text{end.tv_usec} - \text{start.tv_sec} * 1000000.0 - \text{start.tv_usec}) / 1000000.0$

Other Ways to Time Code

- `ntp_gettime()` (Internet RFC 1589)
 - Sort of like `gettimeofday`, but reports estimated error on time measurement
 - Not available for all systems
 - Part of the GNU C Library
- Java: `System.currentTimeMillis()`
 - Known to have resolution problems, with resolution higher than 1 millisecond!
 - Solution: use a native interface to a better timer
- Java: `System.nanoTime()`
 - Added in J2SE 5.0
 - Probably not accurate at the nanosecond level
- Tons of “high precision timing in Java” on the Web

Dedicated Systems

- Measuring the performance of a code must be done dedicated system
 - No other user can start a process
 - The user measuring the performance only runs the minimum amount of processes
 - basically, a shell
 - “single-user mode” is typically considered overkill
- Nevertheless, one should always present measurement results as **averages over several experiments**
 - Because the (small) load imposed by the O/S is not deterministic
- In your assignments, always show averages over 10 experiments, or more if asked to do so explicitly
- In the class we will use machines in dedicated mode

Dedicated Systems

- For performance-oriented assignments, you’re better off using a multi-core / multi-processor machine
 - When I teach this class next year, I expect that everybody will have access to dual- or quad-cores
- If you have such a machine, great, you can use it and make sure you’re not running anything else on it when timing your code
- If you do not have such a machine then you will use one provided by me
 - Note that even if you have such a machine, it may be a good experience to use the one I am providing...