Performance

- Measuring
- Analyzing
- Reporting
What is performance?

Computers are systems of components that interact with each other.

Complex and hard to model analytically

Performance analysis is an experimental discipline of computer science involving:

- Measurement
- Interpretation
- Communication
- Cost – money, time, energy?
Goals of performance analysis

Compare alternatives when buying computers
Determining the impact of a feature when designing a system or upgrading
Set runtime expectations
Performance debugging/optimizations
Basic methodologies

- **Measurements**
  - Not very general
  - Hard to change system parameters
  - Time consuming
  - Intrusion of probes

- **Simulation**
  - Easy to change system parameters
  - Hard to model every detail
  - Needs validation

- **Analytical modeling**
  - Hard
Measuring execution time

Why not just use a stopwatch?

In a modern (time-sharing) operating system your code may not execute the entire time.

Processes are interrupted by I/O, kernel activity and other processes

The **wall clock time**, is the classic stopwatch time

The **CPU time** is the accumulated time the process actually ran on a CPU.

This can further be divided into **system** and **user** CPU time
Unix `time` Command

$ time make osevent
  gcc -O2 -Wall -g -c clock.c
  gcc -O2 -Wall -g -c options.c
  gcc -O2 -Wall -g -c load.c
  gcc -O2 -Wall -g -o osevent osevent.c . . .

0.820u 0.300s 0:01.32 84.8% 0+0k 0+0io 4049pf+0w

0.82 seconds user time
  Corresponds to 82 timer intervals (ticks)
0.30 seconds system time
  Corresponds to 30 timer intervals (ticks)
1.32 seconds wall time
84.8% of total was used running these processes
(.82+0.3)/1.32 = .848
"Time" on a Computer System

real (wall clock) time

= user time \textit{(time executing instructions in the user process)}

= system time \textit{(time executing instructions in kernel on behalf of user process)}

= some other user’s time \textit{(time executing instructions in different user’s process)}

+ + = real (wall clock) time

\textit{We will use the word “time” to refer to user time.}

cumulative user time
Two Fundamental Time Scales

Processor: \(~10^{-9}\) sec.

External events: \(~10^{-2}\) sec.

• Implication
  – Can execute many instructions while waiting for external event to occur
  – Can alternate among processes without anyone noticing
How Much Time Does Program X Require?

CPU time

- How many total seconds are used when executing X?
- Measure used for most applications
- Small dependence on other system activities
  - NOT independent, though

Actual ("Wall") Time

- How many seconds elapse between the start and the completion of X?
- Depends on system load, I/O times, etc.

Important Factors

How does time get measured?

Many processes share computing resources

- Transient effects when switching from one process to another
- The effects of alternating among processes become noticeable
Activity Periods: Light Load

Most of the time spent executing one process

Periodic interrupts, e.g. every 10ms

Keep system from executing only one process

Other interrupts

• Due to I/O activity

• Inactivity periods

• System time spent processing interrupts

~250,000 clock cycles
Interval Counting

OS Measures Runtimes Using Interval Timer

Maintain 2 counts per process

- User time
- System time

Each time you get a timer interrupt, increment counter for executing process

- This is called a clock tick
- User time if running in user mode
- System time if running in kernel mode
(a) Interval Timings

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>B</th>
<th></th>
<th>A</th>
<th></th>
<th>B</th>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Au</td>
<td>Au</td>
<td>Au</td>
<td>As</td>
<td>Bu</td>
<td>Bu</td>
<td>Bu</td>
<td>Bu</td>
<td>Bu</td>
<td>A</td>
</tr>
</tbody>
</table>

A: 110u + 40s  
B: 70u + 30s

(b) Actual Times

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>A</th>
<th></th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A: 120.0u + 33.3s  
B: 73.3u + 23.3s
Accuracy of Interval Counting

Worst Case Analysis

Timer Interval = T

Single process segment measurement can be off by ±T

No bound on error for multiple segments

- Could consistently underestimate, or consistently overestimate

• Estimated time = 70ms
• Min Actual = 60 + ε
• Max Actual = 80 − ε
Average Case Analysis

Over/underestimates tend to balance out

As long as total run time is sufficiently large
  • Min run time $\sim$1 second
  • 100 timer intervals

Consistently miss overhead due to timer interrupts
Accuracy of process timer

Intel Pentium III, Linux, process timer

Process timer unusable below 100 ms
Most modern systems have built-in registers that are incremented every clock cycle

- Very fine grained
- Sometimes maintained as part of process state
  - In Linux, counts elapsed global time
- Problems in SMP, dynamic frequency scaling

Special assembly code instruction to access

On (recent model) Intel machines:

- 64 bit counter.
- RDTSC instruction
Counters and overflow

Consider a 32-bit unsigned int

Ticks are microseconds (10^{-6} s):

- Maximum time = 2^{32} cycles = 1.2 h

Ticks are milliseconds (10^{-3} s):

- Maximum time = 2^{32} ms = 49 days

A 64-bit unsigned int in microseconds gives a maximum time of > 500,000 yr
Multitasking Effects

Cycle Counter Measures Elapsed Time

Keeps accumulating during periods of inactivity

- System activity
- Running other processes

Key Observation

Cycle counter never underestimates program run time

Possibly overestimates by large amount
“What is taking so much time?”

“What which section of my program should be made better?”

- Profilers
  - gprof
  - fancy profilers like Shark
- Timing utilities
  - time.h
The process interval timers can also be accessed from within your code

**times(2)** will return the number of clock ticks since an arbitrary point in the past (userCPU+sysCPU). Use `sysconf(_SC_CLK_TCK)` to determine the number of clock ticks per second.

**clock(3)** will return userCPU time used so far in clocks. Use `CLOCKS_PER_SEC` to convert to seconds.

```c
#include <sys/times.h>

struct tms {
    clock_t tms_utime; /* user time */
    clock_t tms_stime; /* system time */
    ...,
};

clock_t times(struct tms *buf);
```

```c
#include <time.h>

#define CLOCKS_PER_SEC something

clock_t clock(void);
```
Timing, gettimeofday()

Standard UNIX interval timer

Returns Wall-clock time

“Microsecond” resolution

#include <sys/time.h>
#include <time.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);

<sys/time.h>:
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
#include <time.h>
#include <sys/time.h>

static struct timeval start_time, end_time;
……
double start_count, end_count, elapsed_time;

gmtimeofday(&start_time, NULL);
/* Stuff to be measured */
gmtimeofday(&end_time, NULL);

start_count = (double)start_time.tv_sec + 1.e-6 * (double) start_time.tv_usec;
end_count = (double)end_time.tv_sec + 1.e-6 * (double) end_time.tv_usec;
elapsed_time = (end_count - start_count);
printf("The total elapsed time is: %f seconds\n", elapsed_time);
Indicators should if possible be independent of the experiment. Traditional indicators are e.g.

- CPU time
- Numerical accuracy
- Number of iterations
- Scope, applicability
- Portability
- Storage requirement
- Operation count (flops)

Relative indicators or ranking can be misleading
Reporting computational experiments

Presentation of algorithms
Complete description of the algorithm
Specification of the domain of applicability of the algorithm
If possible: Computational complexity, in time and space
If relevant: Convergence results
If relevant: Rate of convergence, order of accuracy
Presentation of implementation

- Programming language used
- Compiler name and options used
- Computer environment, system and OS (*not* “a Linux machine with an AMD processor”)
- Input data
- Settings

Is there a “gold standard”?

Run it in your specific environment!
Presentation of experiments

- Objective of the experiment
- Description of problem generator/input data set
- (Why) Are the results general?
Reporting computational experiments

Presentation of results, e.g. CPU time:

- Description of how the timings were produced
- What parts of the code are included in the measurement?
- What is the variability of the timings?
- How is the variability handled?
Guidelines for measuring execution time

Use a high resolution timer, such as gettimeofday()

Check the amount of time your program spends in the OS using the time command

Measure at least 6-7 runs

Pick the shortest time as a representative or an average

Beware of outliers!

If you have room, report all samples
Measuring FLOPS

1. Count the number of operations in your source code
2. Time the operation
3. Calculate the rates

Example,

Matrix(n*k)-vector product: n*((k-1)+n)
Scalar Product, example

```c
sum = 0.0;
for( i=0; i<N; i++ ) {
    sum += a[i]*b[i];
}
printf("Scalar product of a and b is %f\n",sum);
```

~N additions
~N multiplications
----------------------
~2*N FLOPS
Profiler Basics

Generate an execution profile, by sampling the execution

Stop the program at regular intervals and sample what the program is doing

More insights can be obtained using instrumentation

Adds bookkeeping code

Gives you a view of where in your code time is spent

Can be on functional level (prof or gprof) or basic block level (tcov)
Manual

You add it yourself (e.g. static variable counting the number of calls)

Timing printouts in your code

Compiler-assisted

Compiler inserts sampling code

Used by “gprof” -pg and “prof” -p

Runtime instrumentation

No extra code, but still profiling overhead

Intel Vtune, Sun Analyzer, Valgrind
Call Graph

main()

solve_PDE()

print_results()

do_iteration()
A call-graph is composed of functions

Typically, we have parents and children for each parent.

Inclusive CPU time is the time spent in a parent and all its children.

Exclusive (self) CPU time is the time spent only executing code in the parent.
Simplest tool
Compile and link with -p
Run your code
mon.out is produced
Run $ prof <executable>
Results in a short list of your most expensive functions
<table>
<thead>
<tr>
<th>%Time</th>
<th>Seconds</th>
<th>Cumsecs</th>
<th>#Calls</th>
<th>msec/call</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.3</td>
<td>11.58</td>
<td>11.58</td>
<td>1</td>
<td>11580.00</td>
<td>matrix_Determinant</td>
</tr>
<tr>
<td>25.0</td>
<td>8.96</td>
<td>20.54</td>
<td>140825570</td>
<td>0.0001</td>
<td>_mcount</td>
</tr>
<tr>
<td>12.7</td>
<td>4.57</td>
<td>25.11</td>
<td></td>
<td></td>
<td>_libc_threads_interface</td>
</tr>
<tr>
<td>7.9</td>
<td>2.82</td>
<td>27.93</td>
<td>37384625</td>
<td>0.0001</td>
<td>_free_unlocked</td>
</tr>
<tr>
<td>7.1</td>
<td>2.54</td>
<td>30.47</td>
<td>28671511</td>
<td>0.0001</td>
<td>__pow</td>
</tr>
<tr>
<td>5.9</td>
<td>2.13</td>
<td>32.60</td>
<td>37384627</td>
<td>0.0001</td>
<td>malloc</td>
</tr>
<tr>
<td>4.1</td>
<td>1.46</td>
<td>34.06</td>
<td></td>
<td></td>
<td>.div</td>
</tr>
<tr>
<td>2.2</td>
<td>0.78</td>
<td>34.84</td>
<td>37384625</td>
<td>0.0000</td>
<td>free</td>
</tr>
<tr>
<td>1.6</td>
<td>0.59</td>
<td>35.43</td>
<td></td>
<td></td>
<td>_libc_pthread_getspecific</td>
</tr>
</tbody>
</table>
Slightly more advanced profiler
Compile with and link with -pg
Run your program
Generates gmon.out
$ gprof <executable>
Also gives you break-downs per function, call-graph
<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>65.8</td>
<td>72.99</td>
<td>72.99</td>
<td>1</td>
</tr>
<tr>
<td>10.4</td>
<td>84.50</td>
<td>11.51</td>
<td>1</td>
</tr>
<tr>
<td>6.4</td>
<td>91.56</td>
<td>7.06</td>
<td>108505237</td>
</tr>
<tr>
<td>2.5</td>
<td>94.33</td>
<td>2.77</td>
<td>19958400</td>
</tr>
<tr>
<td>2.2</td>
<td>96.78</td>
<td>2.45</td>
<td>37384625</td>
</tr>
<tr>
<td>2.0</td>
<td>98.96</td>
<td>2.18</td>
<td>74769266</td>
</tr>
<tr>
<td>1.7</td>
<td>100.88</td>
<td>1.92</td>
<td>28671511</td>
</tr>
<tr>
<td>1.7</td>
<td>102.74</td>
<td>1.86</td>
<td>74769266</td>
</tr>
<tr>
<td>1.6</td>
<td>104.49</td>
<td>1.75</td>
<td>37384631</td>
</tr>
<tr>
<td>1.3</td>
<td>105.93</td>
<td>1.44</td>
<td>149538532</td>
</tr>
<tr>
<td>1.1</td>
<td>107.11</td>
<td>1.18</td>
<td>149538532</td>
</tr>
<tr>
<td>1.0</td>
<td>108.18</td>
<td>1.07</td>
<td>17426232</td>
</tr>
<tr>
<td>0.8</td>
<td>109.03</td>
<td>0.85</td>
<td>37384627</td>
</tr>
<tr>
<td>0.7</td>
<td>109.84</td>
<td>0.81</td>
<td>37384625</td>
</tr>
<tr>
<td>0.5</td>
<td>110.42</td>
<td>0.58</td>
<td>17426201</td>
</tr>
<tr>
<td>0.4</td>
<td>110.88</td>
<td>0.46</td>
<td>17426227</td>
</tr>
<tr>
<td>0.0</td>
<td>110.88</td>
<td>0.00</td>
<td>121</td>
</tr>
<tr>
<td>index</td>
<td>%time</td>
<td>self</td>
<td>descendents</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28671511</td>
<td>11.51</td>
<td>24.94</td>
<td>1/1</td>
</tr>
<tr>
<td></td>
<td>2.77</td>
<td>5.19</td>
<td>19958400/19958400</td>
</tr>
<tr>
<td></td>
<td>0.85</td>
<td>6.47</td>
<td>37384623/37384627</td>
</tr>
<tr>
<td></td>
<td>0.81</td>
<td>5.06</td>
<td>37384623/37384625</td>
</tr>
<tr>
<td></td>
<td>1.92</td>
<td>0.00</td>
<td>28671511/28671511</td>
</tr>
<tr>
<td></td>
<td>1.87</td>
<td>0.00</td>
<td>28671511/108505237</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28671511</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CME212 – Introduction to Large Scale Computing in Engineering
High Performance Computing and Programming
Profiler-Assisted Optimization

Use a profiler to check where your code spends its time

Spend your optimization efforts on “hotspots”

Remember: premature optimization is the root of all evil

Be aware of the sampling resolution of your profiler
Profiler-Guided Optimization

PGO supported in Intel, Microsoft (some ed.) and Portland compilers

Allows:

- Increase code memory locality
- Smart decisions on inlining
- Improved branching
- Good experiments just got a lot harder
Performance bugs

When things don't make sense, consider:

- Change of algorithms in libraries.
  - E.g. FFT for prime number length vectors is performed as matrix-vector multiplication ($O(n^2)$ instead of $O(n \log(n))$)
- Complexity hidden in code layers or library calls
- Memory access patterns
- (Unhelpful) optimizations
- Unnecessary or erroneous “NaN”, “Inf”, denormalized numbers etc
Take-home message

• Use profilers and timing utilities to:
  • Understand what the computer is doing
  • Expose performance bottlenecks
  • Target your optimization effort
  • Target your parallelization effort

• Do not optimize code before it is correct
Performance Analysis

- Read Roofline.pdf