Why does my code take a week to run?

Optimizing and profiling your code

slides available at
http://www.as.utexas.edu/~bwmulligan/GSPS_Profiling.pdf

Brian W. Mulligan
UT Austin
Grad. Student & Post-doc seminar
4 Mar 2016

Hands-on

The best way to learn is to do it.

For this talk, each task / component will be accompanied with examples for you to run / test on your own computer.

Examples in c++ & python

Disclaimer

Some of the information given here may not be valid in non c/c++ based languages (python and java are c/c++ based).

There may be particular tricks or methods that may not apply to other languages.

I work almost exclusively from command line when running / compiling. Using Jupyter or browser based stuff may be a bit different.

Outline

- Simple profiling
- Common simple optimizations
- Advanced profiling

Simple profiling

- Use `time` to measure how long a program takes to run
- Use high-precision timing routines to measure individual subroutines or components
- Use medium precision timing routines and do something a lot of times

Using `time`

```
$time sleep 2
```

```
0.000u 0.001s 0:02.04 0.0%     0+0k 64+0io 1pf+0w
```

```
$time multitest
```

```
8.029u 0.005s 0:08.10 99.0%    0+0k 32+0io 0pf+0w
```

Gives clock time (0:02.04, 0:08.10) required to run, as well as CPU processing load (8.029u)

(sleep doesn't load the CPU)

Using timers

Python

```
Import time

start=time.time()

    [code to time here]

runtime=time.time() - start
```

~millisecond accuracy

c++

```
#include <ctime>
double time(void)
{
  timespec tTime_Curr;
  clock_gettime(CLOCK_MONOTONIC_RAW,&tTime_Curr);
  return (double)(tTime_Curr.tv_sec + tTime_Curr.tv_nsec
  * 1.0e-9);
}

double start,runtime;
start = time();
  [code to time]
runtime=time() - start;
```

~nanosecond accuracy

Templates available:
www.as.utexas.edu/~bwmulligan/timing_template.cpp
www.as.utexas.edu/~bwmulligan/timing_template.py

Hint for testing timing

- When working with simple routines, try to avoid using constant inputs
  - Compilers / interpreters may automatically optimize the code
- Use random number inputs instead.

```
import random
x = random.random() * random.random()


#include <stdlib.h>
x = rand() / (double)(RAND_MAX);
```

Optimization: print statements

Get rid of print statements

- Print, printf, cout <<, etc.

Console output is horribly slow. The less output the faster your code will go.

Example:

Create two for loops, one which performs an operation such as x = exp(random.random()), and one which does the same operation and prints the result every time. Do at 10000 iterations of each loop.

Compare the execution time of the two loops.

Optimization: loops

Combine consecutive for loops that have the same or similar range

e.g.

```
for i in range (1,1000):
  x = 1 + a
for i in range (1,1000):
  y = 4.3 * b
for i in range (1,1000):
  z = exp(c)
```

Example: create the three for loops above, with a,b, and c as random variates. Determine the time to execute the three loops separately, then time them combined into a single loop

Optimization: order of data

When working with large datasets, access the data in the order it is stored.

C based languages store in “row first” order (i.e. `a[0][0]`, `a[0][1]`, and `a[0][2]` are contiguous in memory, `a[0][0]` and `a[1][0]` are separated by the width of `a`)

Example: create a set of nested loops that fill a large (1024x1024) array with random numbers in [i][j] order, and another in [j][i] order and compare the time to complete the task.

Note: FORTRAN is “column first”.

Optimization: minimize math operations

Floating point operations are expensive, especially divides.

Remove constant sets of operations from inside of loops or blocks of code

i.e.

```
for i in range(1,1000):
  vol = 4 * math.pi / 3. * r ** 3
```

Better:

```
c = 4 * math.pi / 3.
for i in range(1,1000):
  vol = c * r ** 3
```

Example: Test the above two methods for computing a volume, with radius as a random variate

Optimization:
Don't use ** or pow

The ** or (math.)pow are expensive operations, equivalent to about 10-100 FLOPS

When performing integer power operations, multiply them out.

e.g.

```
x = pow(z,5) // slow
x = z * z * z * z * z // fast
```

- Example: try the above two methods of computing z^5

Optimizing: Don't use python

Equivalent c/c++ program is 100-1000x faster than python.

If the code takes more than 5m to run and is being used often and/or by many people, write it in c/c++ or FORTRAN

If the code is a “one-off” but takes more than 10-15m to run, will probably be better in c/c++ (depends on how much longer it will take you to write c/c++ code).

`numba` can create a compiled version of a python program; significant speedup running this instead of through python interpreter.

Optimizing: Don't spin your wheels

Focus on high gain tasks when optimizing.

e.g. Memory access order is way more important than a for loop with 10 iterations and an unnecessary divide.

Look for subroutines that are called a large number of times, large arrays, or for loops that are nested or have many iterations.

Use a profiler to help identify the code to focus your effort on.

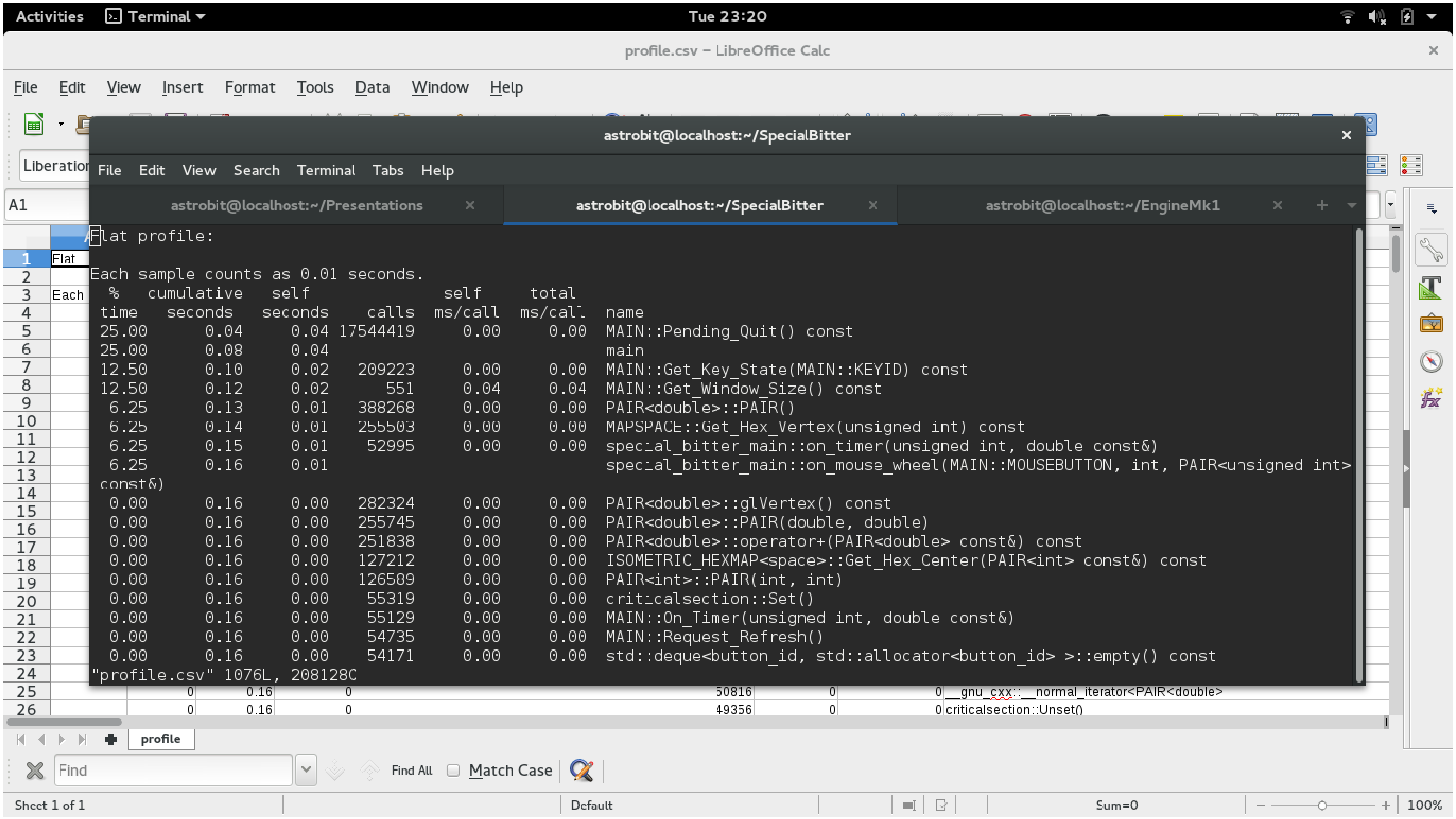
Profiling

Report the frequency of usage and/or CPU time used by individual components / modules

Outputs:

- % of total time spent in a given subroutine
- Time spent in a subroutine
- # of calls to a subroutine
- Call history of subroutine

Example (gprof)



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.00      0.04     0.04 17544419     0.00     0.00  MAIN::Pending_Quit() const
 25.00      0.08     0.04                             main
 12.50      0.10     0.02   209223     0.00     0.00  MAIN::Get_Key_State(MAIN::KEYID) const
 12.50      0.12     0.02      551     0.04     0.04  MAIN::Get_Window_Size() const
  6.25      0.13     0.01   388268     0.00     0.00  PAIR<double>::PAIR()
  6.25      0.14     0.01   255503     0.00     0.00  MAPSPACE::Get_Hex_Vertex(unsigned int) const
  6.25      0.15     0.01    52995     0.00     0.00  special_bitter_main::on_timer(unsigned int, double const&)
  6.25      0.16     0.01                             special_bitter_main::on_mouse_wheel(MAIN::MOUSEBUTTON, int, PAIR<unsigned int>
 const&)
  0.00      0.16     0.00   282324     0.00     0.00  PAIR<double>::glVertex() const
  0.00      0.16     0.00   255745     0.00     0.00  PAIR<double>::PAIR(double, double)
  0.00      0.16     0.00   251838     0.00     0.00  PAIR<double>::operator+(PAIR<double> const&) const
  0.00      0.16     0.00   127212     0.00     0.00  ISOMETRIC_HEXMAP<space>::Get_Hex_Center(PAIR<int> const&) const
  0.00      0.16     0.00   126589     0.00     0.00  PAIR<int>::PAIR(int, int)
  0.00      0.16     0.00    55319     0.00     0.00  criticalsection::Set()
  0.00      0.16     0.00    55129     0.00     0.00  MAIN::On_Timer(unsigned int, double const&)
  0.00      0.16     0.00    54735     0.00     0.00  MAIN::Request_Refresh()
  0.00      0.16     0.00    54171     0.00     0.00  std::deque<button_id, std::allocator<button_id> >::empty() const
"profile.csv" 1076L, 208128C
```

Activities
Terminal
Tue 23:20
profile.csv – LibreOffice Calc
File Edit View Insert Format Tools Data Window Help
astrobit@localhost:~/SpecialBitter
File Edit View Search Terminal Tabs Help
astrobit@localhost:~/Presentations
astrobit@localhost:~/SpecialBitter
astrobit@localhost:~/EngineMk1
25 0 0.16 0 50816 0 0 __gnu_cxx::__normal_iterator<PAIR<double>
26 0 0.16 0 49356 0 0 criticalsection::Unset()
profile
Find
Find All
Match Case
Sheet 1 of 1
Default
Sum=0
100%

How to use profiling data

Look for functions called large # of times and using significant fraction of total time.

If the subroutine is small, replace function call with code – save f.c. overhead.

If subroutine is large, apply methods discussed earlier.

Look for functions called few times but using a significant fraction of total time.

How to profile

C++:

Compile:

```
g++ -g program.cpp -o program -pg
```

Run:

run program as usual. Gmon.out will be created.

Profile:

```
gprof program
```

Python:

```
python -m cProfile program.py
```

Note: gprof and cProfile are “default” profilers. There are many others available that you may like better that may give their output in a more user-friendly way.

Exercise: profile and optimize

http://www.as.utexas.edu/~bwmulligan/prof_ex.py

http://www.as.utexas.edu/~bwmulligan/prof_ex.cpp

C++ specific

- Pass by reference instead of by value for classes or doubles
  - pointer (reference) = 4 bytes (32-bit systems) or 8 bytes (64-bit systems)
  - double = 8 bytes
  - class = n bytes (n probably >> 8)
- Avoid allocating and deallocating memory
  - perform new and delete ops as infrequently as possible
- use stl (map, vector, string, etc.)
- Don't write your own operator = if you don't have to
- Avoid casting from int to double and *vice versa*
- arrays that are sized in multiples of 512 (for doubles) or 1024 (for int) can be slightly more efficient

Things to try at home

- Compare integer and floating point math.
- Compare various math functions (e.g. exp, pow, log, sin, etc.).
- Compare matrix operations using manual code, numpy, blas, linpack, or your other favorite linear algebra package.
- Compare saving data in memory then dumping to a file vs outputing small blocks to file as you go.

Additional resources

C++ optimization suggestions

- http://www.tantalon.com/pete/cppopt/main.htm

python optimization suggestions

- https://wiki.python.org/moin/PythonSpeed/PerformanceTips

Use numba or numpy

- http://numba.pydata.org/
- http://www.numpy.org/

Profilers

- https://en.wikipedia.org/wiki/List_of_performance_analysis_tools