CSE 374: Lecture 28

Profiling and memory

Administrative Notes

- HW6 & HW7
  - While you have time to submit still, you'll be doing us a kindness if you get things in earlier.
- Exam is posted after class today
  - Only multiple choice / short answer
- Office Hours
  - Between ~10:30 & 12 on Monday (I have a meeting before & after, but will be on zoom for as long as possible - check Canvas for link).
  - Tuesday - limited zoom time by request

Course Take-aways

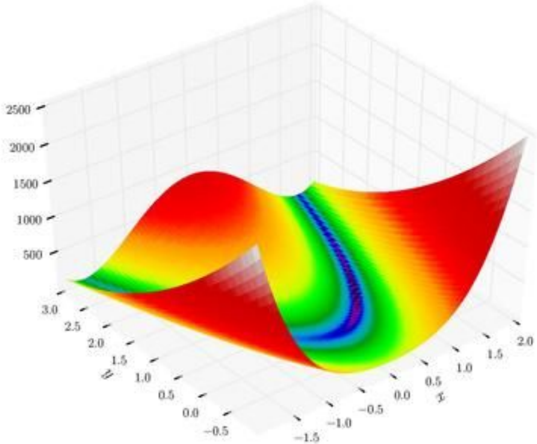
- **Confidence**
- You know enough to believe that you can figure out solutions
- You know some basic commands to get started
- You know some ideas to start searching
- You know about ‘man’
- You have some good resources (Pocket Guide/cplusplus.com)
- You know about processes and how to link them
- You’ve practice breaking down problems to come up with a tool chain
- You’ve seen enough C & C++ to picture alternate computer languages
- You’ve practice with memory to understand how the computer might work

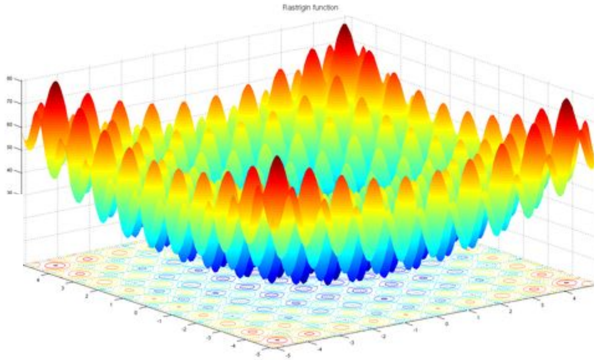
When to use which language?

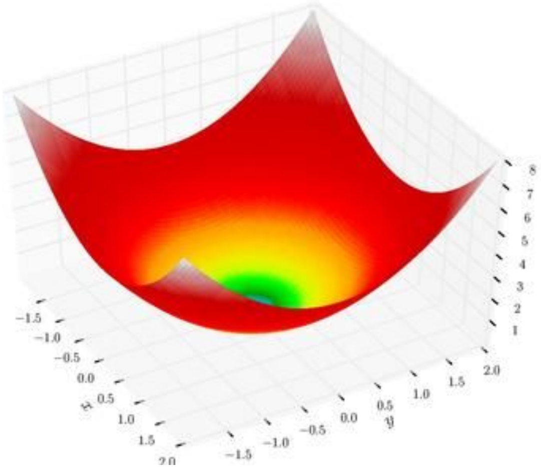
Particle Swarm Optimization

- Particle swarm optimization (PSO) is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995, inspired by social behavior of bird flocking or fish schooling.
- Used to find the global optimum of potentially non-convex functions.
  - Optimize control settings (intelligent control)
  - Fit data to functions (machine learning)
  - Find low energy solutions
    - Low energy often matches the natural solution (protein structures)
- Function optimization is usually an iterative algorithm
  - Coding inefficiencies add up.

Finding minima







Code

```
float* optimize (float(*obj)(float*), float* mins, float* maxs);


int main() {

  float* opt;
  …
  printf("Starting PSO on Sphere\n");
  opt = optimize(spherefunc, mins, maxs);
  …
  return 0;
}

// spherefunc min at 0,0
float spherefunc(float* pos) {
   return pos[0]*pos[0] + pos[1]*pos[1];
}
```

Code

```
float* optimize (float(*obj)(float*), float* mins, float* maxs);


int main() {

  float* opt;
  …
  printf("Start
  opt = optimiz
  …
  return 0;
}

// spherefunc m
float spherefun
   return pos[0]*pos[0] + pos[1]*pos[1];
}
```

Review Opportunity:

`float(*obj)(float*)`

-- function pointer from float* to float

Not to be confused with code or algorithm optimization

Minimize memory usage, computation time, or both

- Examination of behavior of a running program
- Tally of memory allocation
- Record of run time, including breakdown of where the time is spent.

Can use a variety of techniques (hardware interrupts, code tooling, performance counters)

Trace: stream of recorded events, proportional to execution time

Profile: statistical summary of event, proportional to code size

Basics

1. Write code
2. Run test cases (benchmarks)
3. Python clint.py
4. Valgrind

Benchmarking v. Profiling

Benchmarking collects statistics on specific sample problems

(Ex. objective functions are standard benchmark functions for optimization)

- ➔ Number of iterations until convergence
- ➔ Likelihood of finding solution
- ➔ Run time
- ➔ Memory usage

Benchmarking can be very useful for measuring performance on subsequent deliveries

Profiling Tools

- Investigate run-time behavior of code at different points
- Checks time taken by instructions from machine language to high-level functions
  - actual time
  - number of calls to the instruction
- Flat profiler - computes average call times, does not break down calls
- Call graph profiler - shows chains based on called functions

Insertion v. Sampling profilers

Insertion:

- Place specific profiling code in program
- Can be used on various platforms
- Accurate
- Requires recompilation and relinking
- Will affect performance

Sampling:

- Monitoring or snap-shotting at specific intervals
- No modification of code
- Less accurate - limited by sampling rate
- Very small methods often missed
- Not great for memory

$gprof

Gnu profiling tool

Compile with `$gcc -pg` flag

`$./mainopt`

Creates gmon.out

Run profiler with

`$gprof ./mainopt`

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
64.87     0.22     0.22  2040000     0.00     0.00  update_vel
14.74     0.27     0.05  2008462     0.00     0.00  rastrigin
11.79     0.31     0.04  2040000     0.00     0.00  update_pos
 5.90     0.33     0.02      102     0.20     3.34  optimize
 2.95     0.34     0.01    51000     0.00     0.00  update_gb
 0.00     0.34     0.00    20132     0.00     0.00  rosenbrock
 0.00     0.34     0.00    20131     0.00     0.00  spherefunc
 0.00     0.34     0.00      102     0.00     0.00  initialize_opt
```

$valgrind --tool=callgrind

```
$valgrind --tool=callgrind ./mainopt
```

Creates callgrind.out.X

You can read output file

But its tricky; try:

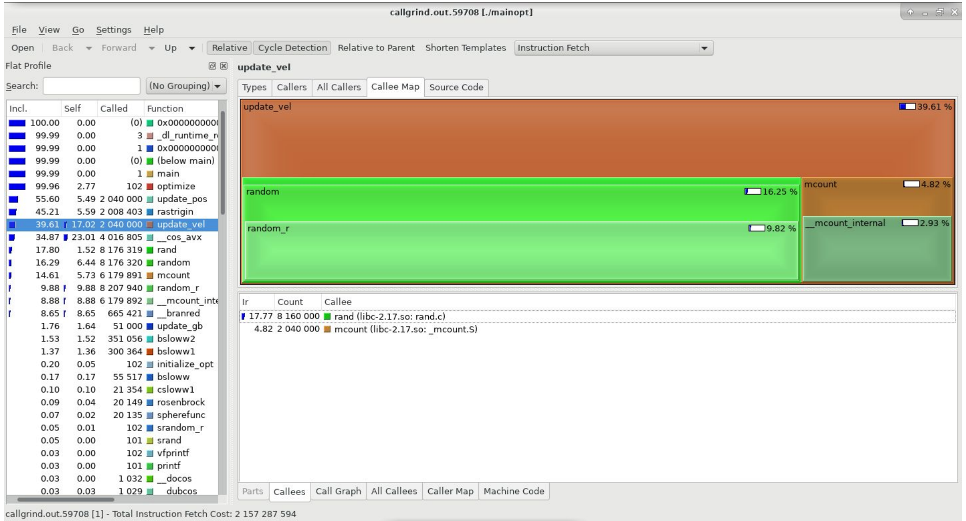
```
$kcachegrind callgrind.out.X
```

(Must install cachegrind:

```
$sudo yum install kcachegrind
```

Or, on Ubuntu:

```
$sudo apt-get install kcacehgrind
```


callgrind.out.59708 [./mainopt]
File View Go Settings Help
Open Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch
Flat Profile
Search: (No Grouping)
update_vel
Types Callers All Callers Callee Map Source Code
update_vel 39.61 %
random 16.25 %
random_r 9.82 %
mcount 4.82 %
__mcount_internal 2.93 %
Parts Callees Call Graph All Callees Caller Map Machine Code

Incl.	Self	Called	Function
100.00	0.00	(0)	0x00000000000
99.99	0.00	3	_dl_runtime_r
99.99	0.00	1	0x00000000000
99.99	0.00	(0)	(below main)
99.99	0.00	1	main
99.96	2.77	102	optimize
55.60	5.49	2 040 000	update_pos
45.21	5.59	2 008 403	rastrigin
39.61	17.02	2 040 000	update_vel
34.87	23.01	4 016 805	__cos_avx
17.80	1.52	8 176 319	rand
16.29	6.44	8 176 320	random
14.61	5.73	6 179 891	mcount
9.88	9.88	8 207 940	random_r
8.88	8.88	6 179 892	__mcount_inte
8.65	8.65	665 421	__branred
1.76	1.64	51 000	update_gb
1.53	1.52	351 056	bsloww2
1.37	1.36	300 364	bsloww1
0.20	0.05	102	initialize_opt
0.17	0.17	55 517	bsloww
0.10	0.10	21 354	csloww1
0.09	0.04	20 149	rosenbrock
0.07	0.02	20 135	spherefunc
0.05	0.01	102	srandom_r
0.05	0.00	101	srand
0.03	0.00	102	vfprintf
0.03	0.00	101	printf
0.03	0.00	1 032	__docos
0.03	0.03	1 029	dubcos

Ir	Count	Callee
17.77	8 160 000	rand (libc-2.17.so: rand.c)
4.82	2 040 000	mcount (libc-2.17.so: _mcount.S)

callgrind.out.59708 [1] - Total Instruction Fetch Cost: 2 157 287 594

$valgrind --tool=callgrind

```
$valgrind --tool=callgrind ./mainopt
```

Creates callgrind.out.X

You can read output file

But its tricky; try:

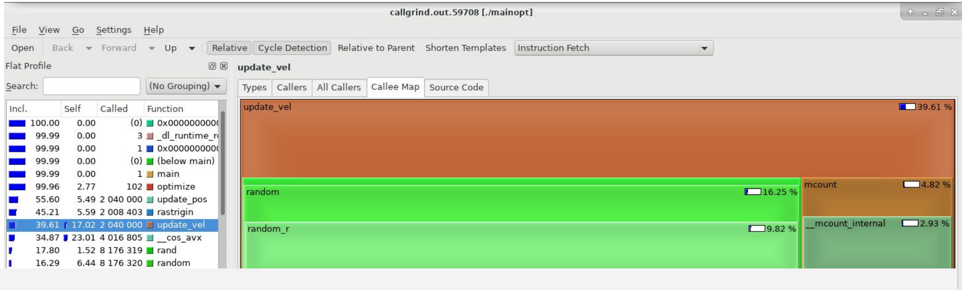
```
$kcachegrind callgrind.out.X
```

(Must install cachegrind:

```
$sudo yum install kcachegrind
```

Or, on Ubuntu:

```
$sudo apt-get install kcacehgr
```



Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000
99.99	0.00	3	_dl_runtime_r
99.99	0.00	1	0x0000000000
99.99	0.00	(0)	(below main)
99.99	0.00	1	main
99.96	2.77	102	optimize
55.60	5.49	2 040 000	update_pos
45.21	5.59	2 008 403	rastrigin
39.61	17.02	2 040 000	update_vel
34.87	23.01	4 016 805	__cos_avx
17.80	1.52	8 176 319	rand
16.29	6.44	8 176 320	random

Review Opportunity:

```
$sudo yum install kcachegrind
```

What does sudo do?

What about yum install?

Observe

- Which methods are being called the most
  - these may not necessarily be the "slowest" methods!
- Which methods are taking the most time relative to the others
  - common problems
    - inefficient unbuffered I/O
    - poor choice of data structure
    - recursion call overhead
    - unnecessary re-computation of expensive information, or unnecessary multiple I/O of same data

Please fill in your course reviews!

Thank You