Lecture 17—Midterm Results, A3 Discussion, Profiling

ECE 459: Programming for Performance

March 12, 2013

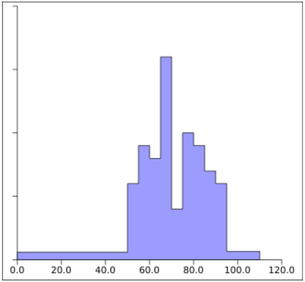
Previous Lecture

Midterm Solutions

Midterm Results

Note: I noticed that Morteza sometimes forgot to give points for Q1 (iii). Double-check your exam.

Raw mean: 69.96%
Raw median: 69.4%



Scaling

We can see two clumps:
"A": 75+
"B": 60–70

Here's the formula I used:

$$P_s = 5 + \frac{100}{90} M_r.$$

P_s is your midterm mark, in percent.
M_r is the raw mark.

Part I

About Assignment 3

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Assignment Problem: Beier-Neely image morphing

> *"an image processing technique typically used as an animation tool for the metamorphosis of one image to another."*

Thaddeus Beier and Shawn Neely. "Feature-Based Image Metamorphosis". SIGGRAPH 1992, pp. 35–42.

Introduction

Image morphing:

computes intermediate images between source and dest.

More complicated than cross-dissolving between pictures
(interpolating pixel colours).

Morphing $=$ warping $+$ cross-dissolving.

Domain Discussion Omitted

Sorry, I had no time to learn and explain how code works.

We'll see it again on Thursday.

Algorithms

All of the C++ built-in algorithms work with anything using the standard container interface.

- `max_element`—returns an iterator with the largest element; you can use your own comparator function.
- `min_element`—same as above, but the smallest element.
- `sort`—sorts a container; you can use your own comparator function.
- `upper_bound`—returns an iterator to the first element greater than the value; only works on a **sorted** container (if the default comparator isn't used, you have to use the same one used to sort the container).
- `random_shuffle`—does `n` random swaps of the elements in the container

Some Hurdles

If you've worked with C++ before, you probably know the awful compiler messages and pages of template expansions.

- You can use `clang` if you have a compiler error, and let it show you the error instead `-std=c++11`
- For the profiler messages, it might get pretty bad. Look for one of the main functions, or if it's a weird name, look where it's called from.
- You can use Google Perf Tools to help break it down—more fine grained.

Example Profiler Function Output

```
[32] std::_Hashtable<unsigned int, std::pair<unsigned
     int const, std::unordered_map<unsigned int,
    double, std::hash<unsigned int>, std::equal_to<
    unsigned int>, std::allocator<std::pair<unsigned
    int const, double> > > >, std::allocator<std::
    pair<unsigned int const, std::unordered_map<
    unsigned int, double, std::hash<unsigned int>,
    std::equal_to<unsigned int>, std::allocator<std::
    pair<unsigned int const, double> > > > >, std::
    _Select1st<std::pair<unsigned int const, std::
    unordered_map<unsigned int, double, std::hash<
    unsigned int>, std::equal_to<unsigned int>, std::
    allocator<std::pair<unsigned int const, double> >
     > > >, std::equal_to<unsigned int>, std::hash<
    unsigned int>, std::__detail::_Mod_range_hashing,
     std::__detail::_Default_ranged_hash, std::
    __detail::_Prime_rehash_policy, false, false,
    true>::clear()
```

is actually `distance_map.clear()` (from last year's assignment), which is automatically called by the destructor.

Things You Can Do

Well, it's a basic implementation, so there should be a lot you can do.

- You can introduce threads, using pthreads (or C++11 threads, if you're feeling lucky), OpenMP, or whatever you want.
- Play around with compiler options.
- Use better algorithms or data strutures
  (maybe Qt is inefficient!)
- The list goes on and on.

Things You Need to Do

Profile!

- Keep your inputs constant between all profiling results so they're comparable.
- Baseline profile with no changes.
- You will pick your two best performance changes to add to the report.
  - You will include a profiling report before the change and just after the change (and only that change!)
  - More specific instructions in the handout.
- There may or may not be overlap between the baseline and the baseline for each change.
- My recommendation: use your initial baseline as the "before" for your first change, and the "after" of the first change for the baseline of your second change.
- Whatever you choose, it should be convincing.

Things To Notice

Your program will be run on `ece459-1` (or equivalent), with a 10 second limit.

We will have some type of leaderboard, so the earlier you have some type of submission, the better.

A Word

This assignment should be **enjoyable**.

Part II

Profiling

Introduction to Profiling

So far we've been looking at small problems.

Must **profile** to see what takes time in a large program.

Two main outputs:

- flat;
- call-graph.

Two main data gathering methods:

- statistical;
- instrumentation.

Profiler Outputs

Flat Profiler:

- Only computes the average time in a particular function.
- Does not include other (useful) information, like callees.

Call-graph Profiler:

- Computes call times.
- Reports frequency of function calls.
- Gives a call graph: who called what function?

Data Gathering Methods

Statistical:

Mostly, take samples of the system state, that is:

- every 2ns, check the system state.
- will cause some slowdown, but not much.

Instrumentation:

Add additional instructions at specified program points:

- can do this at compile time or run time (expensive);
- can instrument either manually or automatically;
- like conditional breakpoints.

Guide to Profiling

When writing large software projects:

- First, write clear and concise code.
  Don't do any premature optimizations—focus on correctness.
- Profile to get a baseline of your performance:
  - allows you to easily track any performance changes;
  - allows you to re-design your program before it's too late.

Focus your optimization efforts on the code that matters.

Things to Look For

Good signs:

- Time is spent in the right part of the system.
- Most time should not be spent handling errors; in non-critical code; or in exceptional cases.
- Time is not unnecessarily spent in the operating system.

gprof introduction

Statistical profiler, plus some instrumentation for calls.

Runs completely in user-space.

Only requires a compiler.

gprof usage

Use the `-pg` flag with `gcc` when compiling and linking.

Run your program as you normally would.

- Your program will now create a `gmon.out` file.

Use gprof to interpret the results: `gprof <executable>`.

`gprof` example

A program with 100 million calls to two math functions.

```
int main() {
    int i,x1=10,y1=3,r1=0;
    float x2=10,y2=3,r2=0;

    for(i=0;i<100000000;i++) {
        r1 += int_math(x1,y1);
        r2 += float_math(y2,y2);
    }
}
```

- Looking at the code, we have no idea what takes longer.
- Probably would guess floating point math taking longer.
- (Overall, silly example.)

Example (Integer Math)

```
int int_math(int x, int y){
    int r1;
    r1=int_power(x,y);
    r1=int_math_helper(x,y);
    return r1;
}

int int_math_helper(int x, int y){
    int r1;
    r1=x/y*int_power(y,x)/int_power(x,y);
    return r1;
}

int int_power(int x, int y){
    int i, r;
    r=x;
    for(i=1;i<y;i++){
        r=r*x;
    }
    return r;
}
```

Example (Float Math)

```
float float_math(float x, float y) {
    float r1;
    r1=float_power(x,y);
    r1=float_math_helper(x,y);
    return r1;
}

float float_math_helper(float x, float y) {
    float r1;
    r1=x/y*float_power(y,x)/float_power(x,y);
    return r1;
}

float float_power(float x, float y){
    float i, r;
    r=x;
    for(i=1;i<y;i++) {
        r=r*x;
    }
    return r;
}
```

Flat Profile

When we run the program and look at the profile, we see:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- One function per line.
- **% time:** the percent of the total execution time in this function.
- **self:** seconds in this function.
- **cumulative:** sum of this function's time + any above it in table.

Flat Profile

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 32.58      4.69     4.69 300000000    15.64    15.64  int_power
 30.55      9.09     4.40 300000000    14.66    14.66  float_power
 16.95     11.53     2.44 100000000    24.41    55.68  int_math_helper
 11.43     13.18     1.65 100000000    16.46    45.78  float_math_helper
  4.05     13.76     0.58 100000000     5.84    77.16  int_math
  3.01     14.19     0.43 100000000     4.33    64.78  float_math
  2.10     14.50     0.30                             main
```

- **calls:** number of times this function was called
- **self ns/call:** just self nanoseconds / calls
- **total ns/call:** average time for function execution, including any other calls the function makes

Call Graph Example (1)

After the flat profile gives you a feel for which functions are costly, you can get a better story from the call graph.

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.30   14.19                 main [1]
                0.58    7.13 100000000/100000000     int_math [2]
                0.43    6.04 100000000/100000000     float_math [3]
-----------------------------------------------
                0.58    7.13 100000000/100000000     main [1]
[2]     53.2    0.58    7.13 100000000         int_math [2]
                2.44    3.13 100000000/100000000     int_math_helper [4]
                1.56    0.00 100000000/300000000     int_power [5]
-----------------------------------------------
                0.43    6.04 100000000/100000000     main [1]
[3]     44.7    0.43    6.04 100000000         float_math [3]
                1.65    2.93 100000000/100000000     float_math_helper [6]
                1.47    0.00 100000000/300000000     float_power [7]
```

Reading the Call Graph

The line with the index is the current function being looked at **(primary line)**.

- Lines above are functions which called this function.
- Lines below are functions which were called by this function (children).

Primary Line

- **time:** total percentage of time spent in this function and its children
- **self:** same as in flat profile
- **children:** time spent in all calls made by the function
  - should be equal to self + children of all functions below

Reading Callers from Call Graph

Callers (functions above the primary line)

- **self:** time spent in primary function, when called from current function.
- **children:** time spent in primary function's children, when called from current function.
- **called:** number of times primary function was called from current function / number of nonrecursive calls to primary function.

Reading Callees from Call Graph

Callees (functions below the primary line)

- **self:** time spent in current function when called from primary.
- **children:** time spent in current function's children calls when called from primary.
  - self + children is an estimate of time spent in current function when called from primary function.
- **called:** number of times current function was called from primary function / number of nonrecursive calls to current function.

Call Graph Example (2)

```
index % time    self  children    called     name

                2.44    3.13 100000000/100000000     int_math [2]
[4]     38.4    2.44    3.13 100000000          int_math_helper [4]
                3.13    0.00 200000000/300000000     int_power [5]
-----------------------------------------------
                1.56    0.00 100000000/300000000     int_math [2]
                3.13    0.00 200000000/300000000     int_math_helper [4]
[5]     32.4    4.69    0.00 300000000          int_power [5]
-----------------------------------------------
                1.65    2.93 100000000/100000000     float_math [3]
[6]     31.6    1.65    2.93 100000000          float_math_helper [6]
                2.93    0.00 200000000/300000000     float_power [7]
-----------------------------------------------
                1.47    0.00 100000000/300000000     float_math [3]
                2.93    0.00 200000000/300000000     float_math_helper [6]
[7]     30.3    4.40    0.00 300000000          float_power [7]
```

We can now see where most of the time comes from, and pinpoint any locations that make unexpected calls, etc.

This example isn't too exciting; we could simplify the math.

Part III

gperftools

Introduction to gperftools

Google Performance Tools include:

- CPU profiler.
- Heap profiler.
- Heap checker.
- Faster `malloc`.

We'll mostly use the CPU profiler:

- purely statistical sampling;
- no recompilation; at most linking; and
- built-in visual output.

Google Perf Tools profiler usage

You can use the profiler without any recompilation.

- Not recommended—worse data.

```
LD_PRELOAD="/usr/lib/libprofiler.so" \
CPUPROFILE=test.prof ./test
```

The other option is to link to the profiler:

- `-lprofiler`

Both options read the `CPUPROFILE` environment variable:

- states the location to write the profile data.

Other Usage

You can use the profiling library directly as well:

- `#include <gperftools/profiler.h>`

Bracket code you want profiled with:

- `ProfilerStart()`
- `ProfilerEnd()`

You can change the sampling frequency with the `CPUPROFILE_FREQUENCY` environment variable.

- **Default value:** 100

pprof Usage

Like gprof, it will analyze profiling results.

```
% pprof test test.prof
    Enters "interactive" mode
% pprof --text test test.prof
    Outputs one line per procedure
% pprof --gv test test.prof
     Displays annotated call-graph via 'gv'
% pprof --gv --focus=Mutex test test.prof
    Restricts to code paths including a .*Mutex.* entry
% pprof --gv --focus=Mutex --ignore=string test test.prof
    Code paths including Mutex but not string
% pprof --list=getdir test test.prof
    (Per-line) annotated source listing for getdir()
% pprof --disasm=getdir test test.prof
    (Per-PC) annotated disassembly for getdir()
```

Can also output dot, ps, pdf or gif instead of gv.

Text Output

Similar to the flat profile in gprof

```
jon@riker examples master % pprof --text test test.prof
Using local file test.
Using local file test.prof.
Removing killpg from all stack traces.
Total: 300 samples
      95  31.7%  31.7%      102  34.0% int_power
      58  19.3%  51.0%       58  19.3% float_power
      51  17.0%  68.0%       96  32.0% float_math_helper
      50  16.7%  84.7%      137  45.7% int_math_helper
      18   6.0%  90.7%      131  43.7% float_math
      14   4.7%  95.3%      159  53.0% int_math
      14   4.7% 100.0%      300 100.0% main
       0   0.0% 100.0%      300 100.0% __libc_start_main
       0   0.0% 100.0%      300 100.0% _start
```

Text Output Explained

Columns, from left to right:

Number of checks (samples) in this function.
Percentage of checks in this function.

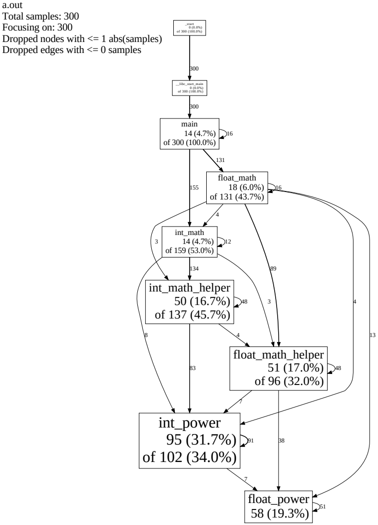
- Same as **time** in `gprof`.

Percentage of checks in the functions printed so far.

- Equivalent to **cumulative** (but in %).

Number of checks in this function and its callees.
Percentage of checks in this function and its callees.
Function name.

Graphical Output



Graphical Output Explained

Output was too small to read on the slide.

- Shows the same numbers as the text output.
- Directed edges denote function calls.
- Note: number of samples in callees =
  number in "this function + callees" -
  number in "this function".
- **Example:**
  `float_math_helper`, 51 (local) of 96 (cumulative)
  96 - 51 = 45 (callees)
  - callee `int_power` = 7 (bogus)
  - callee `float_power` = 38
  - callees total = 45

Things You May Notice

Call graph is not exact.

- In fact, it shows many bogus relations which clearly don't exist.
- For instance, we know that there are no cross-calls between `int` and `float` functions.

As with `gprof`, optimizations will change the graph.

You'll probably want to look at the text profile first, then use the `-focus` flag to look at individual functions.

Summary

- Talked about midterm and A3 (more on Thursday).
- Saw how to use `gprof`
  (one option for Assignment 3).
- Profile early and often.
- Make sure your profiling shows what you expect.

- We'll see other profilers we can use as well:
  - OProfile
  - Valgrind
  - AMD CodeAnalyst
  - Perf