IMPERIAL

Profiling

Advanced Programming

Thomas M Davison

thomas.davison@imperial.ac.uk

Why profile?

- As you have seen in previous modules, often we want to optimise or speed up code
- Useful to know where the slow parts of our code are (the bottlenecks)
- Profiling gives us a powerful way to find those bottlenecks, to point out to us where we need to work on improving our code

- Typically works by sampling the code (e.g. 100 times per second), and recording which function is currently being executed

What do we need to do?

- Compile in “debugging” mode
  - MSVC – compile in “debug”, not “release”, for full symbol support
  - gcc/g++ – compile with the “–g” flag

- This means all the names of the functions are easy to see for a profiler

- We’ll see a few ways we can use a profiler on Mac and Windows

Profiling on MacOS with gcc

- Not many good options available on Macs using Apple silicon (M1, M2, etc)
- One free option that does work (although requires a little work):
  - gperftools (Originally called Google Performance Tools)
  - Install with Homebrew: `brew install gperftools`
  - Need to add another option to g++ compilation command to link the profiler: `-lprofiler`
  - Depending on how you have your library paths set up, you may also need to specify these locations: `-L /opt/homebrew/lib -I /opt/homebrew/include`
  - Full compilation command might look like:

```
g++ -o main main.cpp -L /opt/homebrew/lib -I /opt/homebrew/include -lprofiler -g
```

Changes to code

- Need to add an extra include statement:
  - `#include <gperftools/profiler.h>`
- Also need to add commands to tell the profiler which section of the code to sample

```
int main() {
    ProfilerStart(”my_main.prof");  // Can also control this file with
                                    // CPUPROFILE environment variable
    . . . // Your code here
    ProfilerStop();
    return 0;
}
```

How to run the code

- Run the code you have compiled as normal
- `./main`
- This will produce a file (defined by the `ProfilerStart` command or `CPUPROFILE` env var)
    - E.g. `my_main.prof`

- Next, we need to use the `pprof` command to interpret this file:

```
pprof --text main my_main.prof >> my_main.txt
```

How do I interpret this file?

6 columns in the file, might look like:

```
      flat  flat%   sum%        cum   cum%
      1485   5.7%  93.2%       9487  36.4%   distance
```

1. Number of profiling samples in this function
2. Percentage of profiling samples in this function
3. Percentage of profiling samples in the functions printed so far
4. Number of profiling samples in this function and its callees
5. Percentage of profiling samples in this function and its callees
6. Function name

Can also produce graphical output

- Built in graphical output from gperftools does not work on Apple silicon
- Requires another package on MacOS: `brew install qcachegrind`
- Run program as before
- Run pprof with `--callgrind` flag (instead of `--text`)

```
pprof --callgrind main my_main.prof >> my_main.callgrind
```

- Now, open with qcachegrind

```
qcachegrind my_main.callgrind
```

Live coding...

Other options on MacOS

- On MacOS, can also use Instuments.app from Xcode
- Large install! Can take a long time to download/install if you haven’t done it already
- Allows graphical interpretation of call graph

Windows

- Short story: **Use MSVC!**

- Built in profiling options
- Need to build in “debug” mode to see function names (symbols) or in “release” mode, can load symbols later (more accurate!)
- Can see many of the same things we have seen above

- Debug -> Performance Profiler (or Alt-F2)

- Let’s see this live…