Measuring Performance

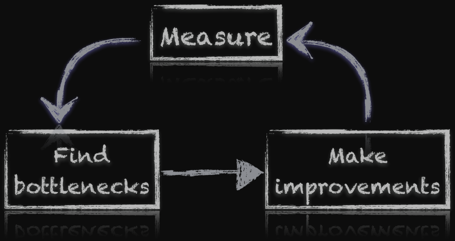
Ramses van Zon

PHY1610, Winter 2025

Measuring Performance a.k.a. Profiling

Profiling

- is a form of *runtime application analysis* that *measures* a performance metric, e.g. the memory or the duration of a program or part thereof, the usage of particular instructions, or the frequency and duration of function calls.
- Like debuggers for finding bugs, *profilers* are *evidence-based* methods to find performance problems.
- Most commonly, profiling information serves to aid program optimization.
- We cannot improve what we don't measure!



Profiling

- Where in the program is time being spent?
- Find and focus in the ‘expensive’ parts.
- Don’t waste time optimizing parts that don’t matter.
- Find bottlenecks.

Two main ways of profiling

Tracing

Events happening during code execution are logged.

- Need to know what events you want logged.
- Depending on how it's done, can slow down code.
- Depending on the tool, may be hard to interpret.

Sampling

At periodic intervals, the state of the system is logged.

- Detects where program spends its time.
- Statistical; needs enough samples.
- May not detect time in system calls.

To instrument or not to instrument

Instrumentation

This refers to anything that changes the build process.

- Adding extra code to your source code to make profiling happen.
- Changing how to build the program.
- Changing how to execute the program.

Instrumentation-free

No need to change the source code.

May need to change how the program is built.

May need to change how the program is run.

In both cases, data is stored during runtime, and a program is needed afterward to display the results.

Instrumentation

- You can instrument regions of the code
- Simple, but incredibly useful
- Runs every time your code is run
- Can trivially see if changes make things better or worse

Tick tock example

```
// sumsins.cpp
#include <cmath>
#include <iostream>
#include "ticktock.h"
int main()
{
  TickTock stopwatch; // holds timing info
  stopwatch.tick();   // starts timing
  // compute
  double b = 0.0;
  for (int i=0; i<=10000000; i++)
       b += sin(i);
  // report
  std::cout << "The sum of sin(i) for i=0..10M"
            << " is " << b << "\n";
  stopwatch.tock("To compute this took");
}
```

```
$ g++ -c -std=c++17 -O3 sumsins.cpp
$ g++ -c -std=c++17 -O3 ticktock.cc
$ g++ sumsins.o ticktock.o -o sumsins
$ ./sumsins
The sum of sin(i) for i=0..10M is 1.95589
To compute this took      0.1318 sec
```

This actually just uses the `std::chrono` standard C++ library under the hood, but offers a simpler way to time portions of code.

git clone https://github.com/vanzonr/ticktock

Instrumentation-free profiling with OS utilities

Let's start by looking at some utilities provided by the Linux OS that we can use for profiling.

- `time`
  Measure duration of the whole run of an application
- `top`, `htop`
  Monitor CPU, memory and I/O utilization while the application is running.
- `ps`, `vmstat`, `free`
  (One-time) information on a running processes
- ...

Time : timing the whole program

- time is a built-in command in the bash shell.
- Very simple to use. It can be run from the Linux command line on any command.

Suppose we have an application waved1d to be run as ./wave1d longwaveparams.txt.

We can just prepend time to the command:

```
$ time ./wave1d longwaveparams.txt

[ program output ]

real     0m16.715s  # Elapsed "walltime"
user     0m16.105s  # Actual user time (of all cores)
sys      0m0.252s   # System/OS time, e.g. I/O
```

- In a serial program:
  `real = user + sys`
- In parallel, at most:
  `user = nprocs x real`
- Can be run on tests to identify *performance regressions*

Top: Watching a program run

- Run a command in one terminal.
- Run top or `top -u $USER` in another terminal on the same node (type 'q' to exit).

```
top - 20:26:34 up 6 days,  2:52,  8 users,  load average: 0.47, 0.81, 1.06
Tasks: 380 total,   2 running, 378 sleeping,   0 stopped,   0 zombie
%Cpu(s):  6.5 us,  0.6 sy,  0.0 ni, 92.7 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 65945184 total, 52059848 free,  1759912 used, 12125424 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 57586756 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
12241 rzon      20   0  104376   8696   6228 R  97.7  0.0   0:05.96 wave1d
12244 rzon      20   0  173104   2656   1696 R   0.3  0.0   0:00.02 top
 6199 rzon      20   0  186868   2760   1100 S   0.0  0.0   0:01.09 sshd
 6200 rzon      20   0  127364   3364   1816 S   0.0  0.0   0:00.10 bash
```

- Refreshes every 3 seconds.
- htop is an alternative to top with a nicer default display.
- ps, vmstat and free can give the same information, but just at a single time and non-interactively.

Pro-tip: type "zxcVm1t0" after starting top for a more insightful display.

Sampling

Concept

- As the program executes, every so often ($\sim$ 100ms) a timer goes, off, and the current location of execution is recorded
- Shows where time is being spent

Benefits:

- Allow us to get finer-grained (more detailed) information about where time is being spent
- Very low overhead
- No instrumentation, i.e., no code modification

Disadvantages:

- Requires sufficiently long runtime to get enough samples.
- Does not tell us *why* the code was there.

An effective profiler sampler : gprof

- gprof is a profiler that works by adding the options -pg -g to g++.
  (both in compilations and linking).
- Rebuild and (re)run the application.
- The code will then sample itself when it is run.
- In addition, functions calls (if not inlined) will be counted.
- During the run, this raw information is stored in a file called "`gmon.out`" .
- `gmon.out` needs to be analysed by the `gprof` command.
- The `gprof` command takes at least two arguments: the executable and the gmon.out file name.
  This will show how much of its time the program spend in each function.
- It also can take an option `--line` argument, to show line-by-line timings.

Gprof example

```
$ module load gcc/12.3 rarray/2.8.0
$ make
g++ -c -pg -g -Og -std=c++17 -Wall -Wfatal-errors -o wave1d.o wave1d.cpp
...
g++ -pg -g -Og -o wave1d wave1d.o parameters.o ...
$ ./wave1d longwaveparams.txt
```

Note that the Makefile needs to be changed to add the -pg flags.

Optimization flags also needs to be changed, particularly for line-resolve timing.

- `-Og` is usually safe.
- To use `-O2` or `-O3` but you may need to disable some optimizations, e.g. `-fno-inline-functions-called-once -fno-inline-small-functions -fno-omit-frame-pointer`

Process the results with a command like:

- `gprof --line ./wave1d gmon.out | less`

Output of gprof –line

```
$ gprof --line ./wave1d gmon.out | less
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 32.20      1.11     1.11                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:42 @ 4
 23.50      1.92     0.81                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:44 @ 4
 16.97      2.51     0.59                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:43 @ 4
 15.52      3.04     0.54                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:42 @ 4
  2.18      3.12     0.08                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:49 @ 4
  2.18      3.19     0.08                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:50 @ 4
  2.18      3.27     0.08                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:51 @ 4
  1.45      3.32     0.05                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:41 @ 4
  0.87      3.35     0.03                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:49 @ 4
  0.73      3.37     0.03                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:48 @ 4
  0.58      3.39     0.02                             one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:47 @ 4
  0.58      3.41     0.02                             ra::shared_shape<double, 1>::size() const (rarray:765 @ 403c32)
  0.44      3.43     0.02                             std::ostream::operator<<(double) (ostream:221 @ 403c12)
  0.29      3.44     0.01                             std::ostream::operator<<(double) (ostream:221 @ 403beb)
  0.15      3.44     0.01                             output_snapshot(double, Waves&, std::basic_ofstream<char, std::
  0.15      3.45     0.01                             std::ostream::operator<<(double) (ostream:221 @ 403c06)
  0.15      3.45     0.01                             std::basic_ostream<char, std::char_traits<char> >& std::operato
  0.00      3.45     0.00       20     0.00     0.00  ra::shared_shape<double, 1>::decref() (rarray:868 @ 4031f0)
```

Other ways to run gprof

- `gprof ./wave1d gmon.out`

  Gives profile by function

- `gprof -A --all-lines --line --annotated-source=evolve.cpp ./wave1d gmon.out`

  Annotates the lines with the number of times they are hit (not real time).

- `gprof -q ./wave1d gmon.out`

  Shows the call graph, ordered by cumulative time.

Caveats

- `gprof` measures time spent in your code. It can miss time spent in library calls.
- `gprof --line` orders by self-time, but often the cumulative time is more important.
- `gprof -A --all-lines --line ./wave1d gmon.out`

Memory Profiling

Most profilers use *time* or *events* as metrics, but what about *memory*?

Valgrind

- Massif: Memory Heap Profiler
  - `valgrind --tool=massif ./mycode`
  - `ms_print massif.out`
- Cachegrind: Cache Profiler
  - `valgrind --tool=cachegrind ./mycode`
  - Kcachegrind (gui frontend for cachegrind)

https://valgrind.org

Linaro Forge

Linaro Forge (formerly ARM Forge) is a commercial suite of developer tools: a debugger DDT, a profiler MAP and a performance report utility (perf-report).

Get them on the Teach cluster or on Niagara with:

```
module load ddt-cpu
```

Performance Reports

- Compile with debugging on, ie -g (but **not** -pg)
- `perf-report  ./wave1d longwaveparameters.txt`
- Generates `.txt` and `.html` files

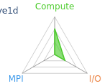
MAP

- Compile with debugging on, ie -g (but **not** -pg)
- `map` or `map  ./wave1d longwaveparameters.txt`
- Can run without a gui with the `--profile` parameter.

Linaro Performance Reports (Forge)

arm PERFORMANCE REPORTS

Command:	/gpfs/fs1/home/s/scinet/rzon/teaching/phy1610/2022/hw2/wave1d longwaveparams.txt
Resources:	1 node (16 physical, 16 logical cores per node)
Memory:	63 GiB per node
Tasks:	1 process
Machine:	teach01.scinet.local
Start time:	Mon. Feb. 7 22:31:35 2022
Total time:	22 seconds
Full path:	/gpfs/fs1/home/s/scinet/rzon/teaching/phy1610/ 2022/hw2



Summary: wave1d is Compute-bound in this configuration

Compute	68.9%	Time spent running application code. High values are usually good. This is **average**; check the CPU performance section for advice
MPI	0.0%	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	31.1%	Time spent in filesystem I/O. High values are usually bad. This is **high**; check the I/O breakdown section for optimization advice

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 68.9% CPU time:

Scalar numeric ops	36.2%
Vector numeric ops	0.0%
Memory accesses	63.8%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 0.0% MPI time:

Time in collective calls	0.0%
Time in point-to-point calls	0.0%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	0.00 bytes/s

No time is spent in MPI operations. There's nothing to optimize here!

I/O

A breakdown of the 31.1% I/O time:

Time in reads	0.0%
Time in writes	100.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	47.3 MB/s

Most of the time is spent in write operations with a low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to ...

Threads

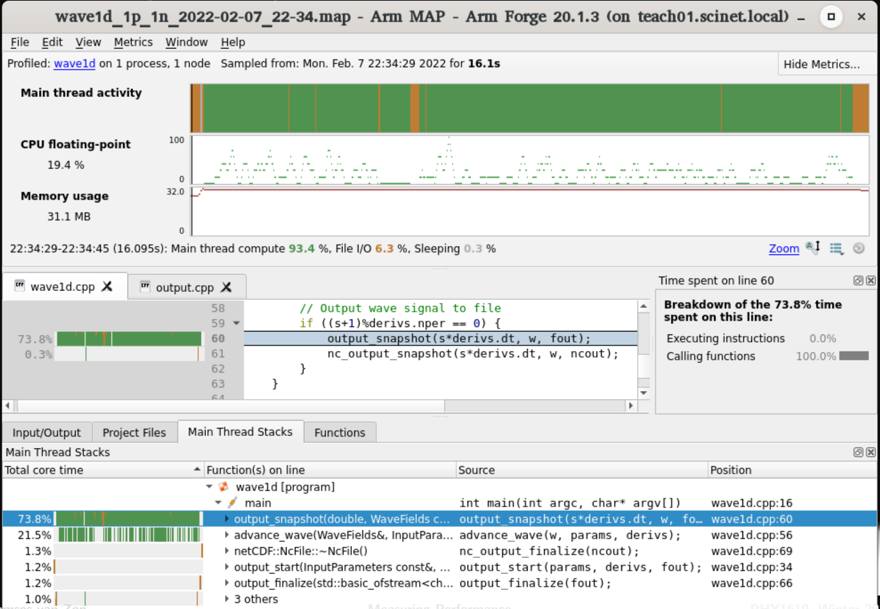
A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	4.3%
System load	6.5%

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of ...

Linaro MAP (Forge)



Profiling Summary

- Two main approches: tracing vs sampling
- Put your own timers in the code in/around important sections, find out where time is being spent.
  - if something changes, you'll know in what section
- `gprof` is easy to use and excellent at finding where most of the time in your code is spent.
- Know the 'expensive' parts of your code and spend your programming time accordingly.
- `valgrind` is good for all things memory; performance, cache, and usage.
- Linaro Forge (with MAP, DDT, perf-report) is a great tool, if you have it available use it!
- The "write less code" advice applies here too: use already optimized libraries.