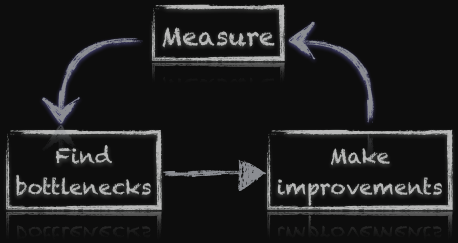
Measuring Performance a.k.a. Profiling

Ramses van Zon

PHY1610, Winter 2026

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

```cpp
// sumsins.cpp
#include <cmath>
#include <print>
#include "ticktock.h"
int main()
{
  TickTock stopwatch; // holds timing info
  stopwatch.tick();   // starts timing
  // compute
  double b = 0.0;
  long n = 100'000'000;
  for (int i=0; i<=n; i++)
       b += sin(i);
  // report
  std::print("The sum of sin(i) for i=0..{}", n);
  std::println(" is {:.5}", b);
  stopwatch.tock("To compute this took");
}
```

```
$ g++ -c -std=c++23 -O3 sumsins.cpp
$ g++ -c -std=c++23 -O3 ticktock.cc
$ g++ sumsins.o ticktock.o -o sumsins
$ ./sumsins
The sum of sin(i) for i=0..100000000 is 1.7136
To compute this took      1.342 sec
```

`ticktock` 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 `gameof1d` to be run as `./gameof1d 1000 100000 0.35 > output.dat`.

We can just prepend "time" to the command:

```
$ time ./gameof1d 1000 100000 0.35 > output.dat

real    0m5.711s  # Elapsed "walltime"
user    0m5.587s  # Actual user time (of all cores)
sys     0m0.075s  # 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 - 16:21:13 up 22 days,  6:13, 23 users,  load average: 0.93, 1.69, 1.16
Tasks: 1000 total,   2 running, 995 sleeping,   3 stopped,   0 zombie
%Cpu(s):  2.5 us,  0.4 sy,  0.0 ni, 96.9 id,  0.0 wa,  0.1 hi,  0.0 si,  0.0 st
MiB Mem : 193005.4 total, 138114.3 free,   6237.7 used,  48653.4 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used. 183231.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
2050140 lcl_uot+  20   0    6036   3648   3424 R 100.0   0.0   0:16.04 gameof1d
2050163 lcl_uot+  20   0   22536   5188   3652 R  11.8   0.0   0:00.03 top
1937481 lcl_uot+  20   0   48400   7792   5424 S   0.0   0.0   0:00.44 sshd
2046547 lcl_uot+  20   0  232920   5156   3940 S   0.0   0.0   0:00.02 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 (~ 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

```
$ git clone ~lcl_uotphy1610s1466/gameof1d
$ module load gcc/14.3 rarray/2.8.2 # catch2/3.3.1
$ make clean
$ make CXXFLAGS='-std=c++23 -Og -g -pg' LDFLAGS='-Og -g -pg'
g++  -std=c++23 -Og -g -pg -c -o gameof1d.o gameof1d.cpp
...
g++ -Og -g -pg -o gameof1d gameof1d.o readparams.o fillcells.o updatecells.o outputcells.o
$ ./gameof1d 1000 100000 0.35 > output.dat
```

Note that the CXXFLAGS and LDFLAGS from the Makefile needs to be overwriiten to add the `-pg` flags.

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

- `-Og` is usually safe.
- Possible 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 ./gameof1d gmon.out | less`

Output of gprof –line

```
$ gprof --line ./gameof1d gmon.out | less
```

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 10.17      0.65     0.65                             std::__format::_Sink_iter<char> std::__format::__do_vformat_to<std
  8.45      1.19     0.54                             std::__format::_Arg_value<std::basic_format_context<std::__format:
  7.67      1.68     0.49                             _init
  3.76      1.92     0.24                             decltype(auto) std::visit_format_arg<std::__format::__do_vformat_t
  3.29      2.13     0.21                             next_cell_state(ra::rarray<bool, 1> const&, int) (updatecells.cpp:
  2.50      2.29     0.16                             next_cell_state(ra::rarray<bool, 1> const&, int) (updatecells.cpp:
  2.03      2.42     0.13 100001000     1.30     2.10  void std::__cxx11::basic_string<char, std::char_traits<char>, std
  1.88      2.54     0.12                             std::vprint_nonunicode(std::ostream&, std::basic_string_view<char,
  1.72      2.65     0.11 100101001     1.10     1.10  std::vprint_nonunicode(std::ostream&, std::basic_string_view<char
  1.41      2.74     0.09 100301003     0.90     0.90  std::__format::_Sink_iter<char> std::__format::__do_vformat_to<st
  1.41      2.83     0.09                             ra::detail::shared_buffer<bool>::cbegin() const (rarray:198 @ 40f4
  1.41      2.92     0.09                             auto std::make_format_args<std::basic_format_context<std::__format
  1.33      3.01     0.09 100001000     0.85     0.85  std::__format::_Seq_sink<std::__cxx11::basic_string<char, std::ch
  1.25      3.09     0.08 100001000     0.80     0.80  std::__cxx11::basic_string<char, std::char_traits<char>, std::all
  1.25      3.17     0.08 100001000     0.80     0.80  auto std::__format::__do_vformat_to<std::__format::_Sink_iter<cha
  1.25      3.25     0.08                             next_cell_state(ra::rarray<bool, 1> const&, int) (updatecells.cpp:
  ...
```

Ways to run gprof

```
$ gprof ./gameof1d gmon.out
```

Gives profile by function

```
$ gprof -A --all-lines --line --annotated-source=updatecells.cpp ./gameof1d gmon.out
```

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

```
$ gprof -q ./gameof1d 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 ./gameof1d gmon.out` gives too much

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 Trillium with:

```
module load ddt-cpu
```

Performance Reports

Compile with debugging on, ie -g (but **not** -pg)

```
$ make clean
$ make CXXFLAGS='-std=c++23 -Og -g' LDFLAGS='-Og -g'
```

Run with

```
perf-report ./gameof1d 1000 100000 0.35 > output.dat
```

Generates `.txt` and `.html` files

MAP

Compile the same way.

Run with

```
$ map ./gameof1d 1000 100000 0.35
```

Can run without a gui with the `--profile` parameter.

Linaro Performance Report (Forge)

Linaro Performance Reports

Command:	/home/l/lcl_uotphy1610/lcl_uotphy1610s1466 /gameof1d/stuff/gameof1d 1000 1000000 0.35
Resources:	1 node (40 physical, 40 logical cores per node)
Memory:	188 GiB per node
Tasks:	1 process
Machine:	teach-login02
Architecture:	x86_64
CPU Family:	cascadelake-x
Start time:	Wed. Mar. 11 16:48:32 2026
Total time:	87 seconds (about 1 minute)
Full path:	/home/l/lcl_uotphy1610/lcl_uotphy1610s1466/gameof1d/stuff

Summary: gameof1d is Compute-bound in this configuration

Compute	99.6%	86.4s	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	0.0%	0.0s	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	0.4%	0.3s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

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 99.6% (86.4s) CPU time:

Scalar numeric ops	14.0%	12.3s
Vector numeric ops	0.0%	0.0s
Memory accesses	69.5%	60.1s

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% (0.0s) MPI time:

Time in collective calls	0.0%	0.0s
Time in point-to-point calls	0.0%	0.0s
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 0.4% (0.3s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	0.3s
Effective process read rate	0.00 bytes/s	
Effective process write rate	2.92 GB/s	

Most of the time is spent in write operations with a high effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	2.5%	
System load	6.3%	

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	42.1 MiB	
Peak process memory usage	42.1 MiB	
Peak node memory usage	3.0%	

The peak node memory usage is very low. Larger problem sets can be run before scaling to multiple nodes.

Energy

A breakdown of how energy was used:

CPU	not supported %	
System	not supported %	
Mean node power	not supported W	
Peak node power	0.00 W	

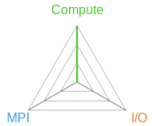
Energy metrics are not available on the system.
CPU metrics: Error parsing /sys/class/powercap/intel-rapl:0/energy_uj

Measuring Performance

Linaro Performance Report (Forge) - zoom

Linaro Performance Reports

Command:	/home/l/lcl_uotphy1610/lcl_uotphy1610s1466 /gameof1dstuff/gameof1d 1000 1000000 0.35
Resources:	1 node (40 physical, 40 logical cores per node)
Memory:	188 GiB per node
Tasks:	1 process
Machine:	teach-login02
Architecture:	x86_64
CPU Family:	cascadelake-x
Start time:	Wed. Mar. 11 16:48:32 2026
Total time:	87 seconds (about 1 minute)
Full path:	/home/l/lcl_uotphy1610/lcl_uotphy1610s1466/gameof1dstuff



Summary: gameof1d is Compute-bound in this configuration

Compute	99.6%	86.4s	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	0.0%	0.0s	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	0.4%	0.3s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

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 99.6% (86.4s) CPU time:

MPI

A breakdown of the 0.0% (0.0s) MPI time:

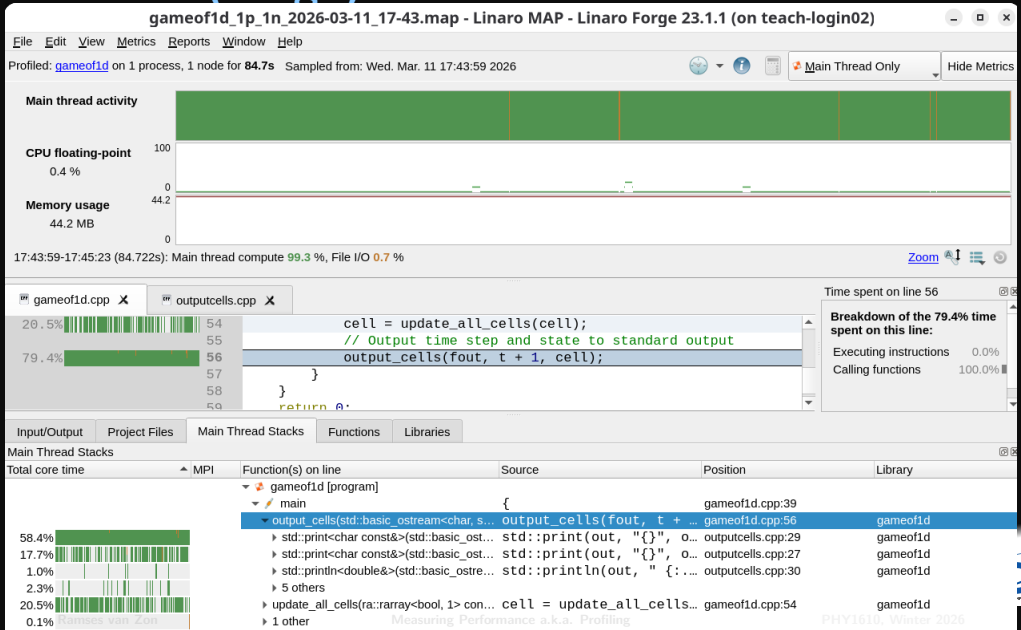
Careful!

It says that it's Compute bound.

But the gprof output pointed at I/O as the bottleneck.

Because:
I/O formatting and conversions to text requires computations!

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.
- Linaro Forge (with MAP, DDT, perf-report) is a great tool, if you have access, use it!
- The "write less code" advice applies here too: use already optimized libraries.

What about ‘the others’?

When running on a shared cluster, timings depend on what other users are doing.

Would like to reserve a part of the cluster exclusively for a particular computation!

- Makes computation time predictable.
- Makes profiling reliable.

Can we do that? Yes!

Using a scheduler

- When you log in, you are on a login nodes.
- These are shared.
- You can edit, compile and run quick tests there.
- But the real compute with dedicated resources has to be done on compute nodes.

Once you have compiled and tested your code or workflow on a login node, and confirmed that it behaves correctly, you are ready to submit jobs to the cluster.

You submit jobs from a login node to the scheduler queue by passing a script to the sbatch command:

```
teach-login01:~$ sbatch jobscript.sh
```

- The scheduler will run your jobs will run on one or more of Teach compute nodes.
- When and where your job runs is determined by the scheduler.
- Teach uses SLURM as its job scheduler.

Example

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=3:00:00
#SBATCH --job-name serialjob
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/14

time ./gameof1d 1000 100000 0.35 > output.dat
```

```
teach-login01:~$ sbatch serialjob.sh
```

- First line indicates that this is a bash script.
- Lines starting with `#SBATCH` go to SLURM.
- sbatch reads these lines as a job request
- In this case, SLURM looks for 1 core to be put on one node to be run for 3 hours.
- Once it found such a node, script is run:
  - Loads module(s)
  - Run the code and times it.

Example #2

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --time=3:00:00
#SBATCH --job-name serialjob4
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/14

CASES="0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9"
parallel -j 4 ./gameof1d 1000 100000 ::: $CASES
```

```
teach-login01:~$ sbatch serialjob4.sh
```

- First line indicates that this is a bash script.
- Lines starting with #SBATCH go to SLURM.
- sbatch reads these lines as a job request
- In this case, SLURM looks for 4 cores to be put on the same one node to be run for 3 hours.
- Once it found such a node, script is run:
  - Loads module
  - Has gnu-parallel load-balance 8 cases over 4 cores.

Using the Scheduler

Some of the most common sbatch parameters are:

`-t`	`--time`	amount of time
`-N`	`--nodes`	number of nodes
`-n`	`--ntasks`	number of tasks
	`--ntasks-per-node`	number of tasks per node
`-c`	`--cpus-per-task`	number of threads per task
`-G`	`--gpus-per-node`	number of gpus per node
	`--mem`	amount of memory
`-A`	`--account`	scheduler account to use
	`--reservation`	use reserved nodes

Using the Scheduler

Commands to interact with the scheduler

`sbatch`	submit job
`squeue`	see queued jobs and their status
`scancel`	cancel a job
`salloc`	get short interactive job on a compute node
`debugjob -n P`	get short interactive job on a compute node

More on how to submit jobs at:
https://docs.scinet.utoronto.ca/index.php/Teach