Measuring Performance

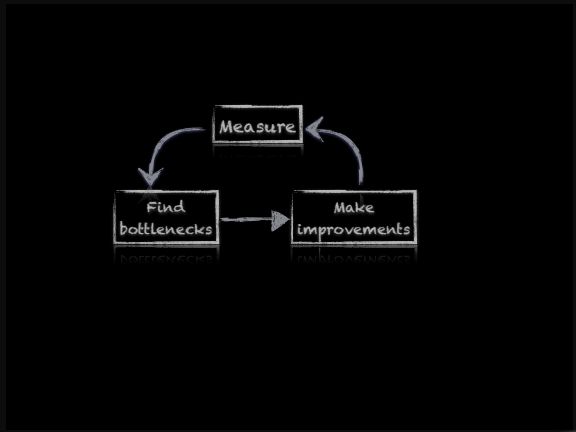
Ramses van Zon

PHY1610, Winter 2023

Measuring Performance

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 wate time optimizing parts that don’t matter.
- Find bottlenecks.

```
TickTock timer;
double timesteptime = 0.0;
double snapshottime = 0.0;
timer.tick();
initialize_wave(w);
timer.tock("initialization took");

// Output initial wave signal to files
timer.tick();
output_snapshot(0.0, w, fout);
nc_output_snapshot(0.0, w, ncout);
snapshottime += timer.silent_tock();
```

```
// Take timesteps
for (int s = 0; s < derivs.nsteps; s++) {

    // Evolve one time step
    timer.tick();
    advance_wave(w, params, derivs);
    timesteptime += timer.silent_tock();

    // Output wave signal to files
    if ((s+1)%derivs.nper == 0) {
        timer.tick();
        output_snapshot(s*derivs.dt,w,fout);
        nc_output_snapshot(s*derivs.dt,w,ncout);
        snapshottime += timer.silent_tock();
    }
}
std::cout
  <<"timesteps took "<<timesteptime<<"s\n"
  <<"file I/O took  "<<snapshottime<<"s\n";
```

Profiling

Two main approaches for Profiling

- Tracing vs. Sampling
- Instrumentation vs. Instrumentation-Free

The code on the right using "instrumentation": extra code needed to be added.

```
// Take timesteps
for (int s = 0; s < derivs.nsteps; s++) {

    // Evolve one time step
    timer.tick();
    advance_wave(w, params, derivs);
    timesteptime += timer.silent_tock();

    // Output wave signal to files
    if ((s+1)%derivs.nper == 0) {
        timer.tick();
        output_snapshot(s*derivs.dt,w,fout);
        nc_output_snapshot(s*derivs.dt,w,ncout);
        snapshottime += timer.silent_tock();
    }
}
std::cout
  <<"timesteps took "<<timesteptime<<"s\n"
  <<"file I/O took  "<<snapshottime<<"s\n";
```

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

```
// 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 -O2 sumsins.cpp
$ g++ -c -std=c++17 -O2 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.

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

```
$ 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
```

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 for Profiling

- 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.

A simple sampler : gprof

- gprof is a profiler that works by adding the options "-pg -g" to g++ (both in compilations and linking), the code will sample itself.
- Depending on the combination of versions of g++ and gprof, it may also require the `-gstabs` option.
- Rebuild and (re)run the application.
- A file called "`gmon.out`" is created as a side-effect now.
- `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 info.

```
$ make clean && make
g++ -c -pg -g -gstabs -std=c++17 -O2 -o wave1d.o wave1d.cpp
g++ -c -pg -g -gstabs -std=c++17 -O2 -o parameters.o parameters.cpp
...
g++ -O2 -pg -g -gstabs -o wave1d wave1d.o parameters.o ... ncoutput.o -lnetcdf_c++4 -lnetcdf
$ ./wave1d longwaveparameters.txt
Results written to 'longresults.txt'.
and also written to 'longresults.txt.nc'.
$ gprof ./wave1d gmon.out
...
$ gprof --line ./wave1d gmon.out
```

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
 23.50    1.92   0.81                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:44
 16.97    2.51   0.59                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:43
 15.52    3.04   0.54                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:42
  2.18    3.12   0.08                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:49
  2.18    3.19   0.08                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:50
  2.18    3.27   0.08                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:51
  1.45    3.32   0.05                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:41
  0.87    3.35   0.03                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:49
  0.73    3.37   0.03                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:48
  0.58    3.39   0.02                      one_time_step(Waves&, Params&, Derived&) (wavefields.cpp:47
  0.58    3.41   0.02                      ra::shared_shape<double, 1>::size() const (rarray:765 @ 403c
  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, st
  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::opera
  0.00    3.45   0.00    20    0.00    0.00 ra::shared_shape<double, 1>::decref() (rarray:868 @ 4031f0)
```

Memory Profiling

Most profilers use time as a *metric*, 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

ARM Forge

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 unload gcc/12   # for technical reasons gcc must be loaded after ddt
module load ddt
module load gcc/12
```

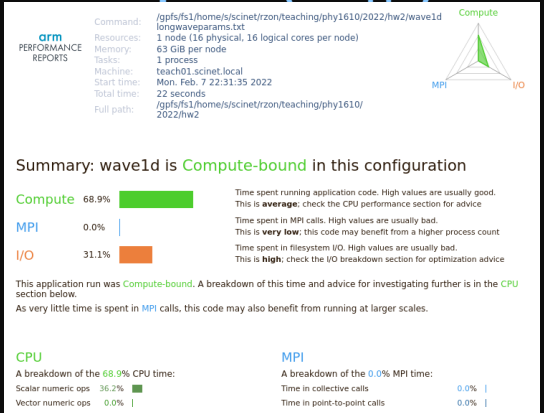
Performance Reports

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

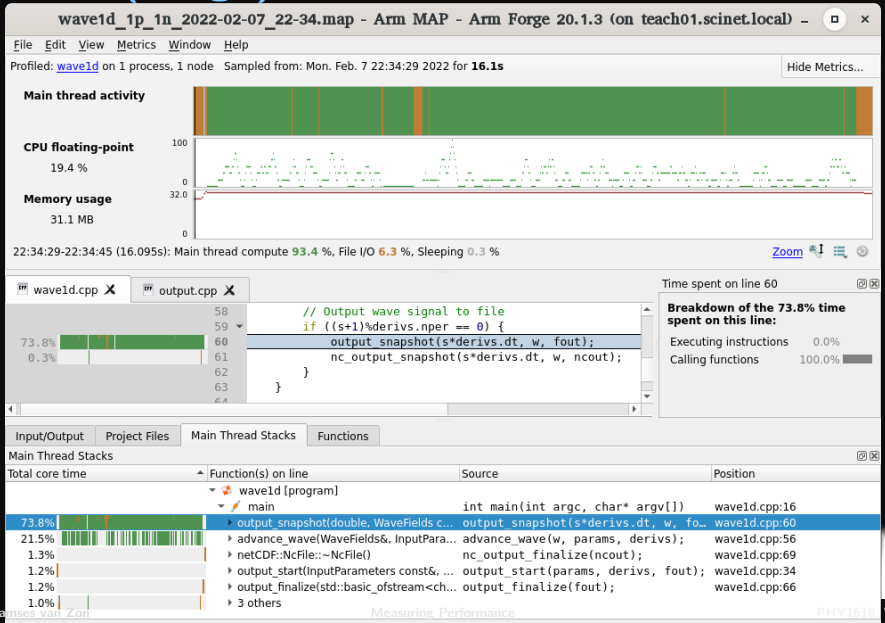
MAP

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

ARM Performance Reports (Forge)



ARM 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.
- ARM 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.

Using SciNet (and other HPC) clusters

Teach cluster: login and compute nodes

So far, you've been running short computations on the Teach login node.

But most cluster (incl. Teach) consists of at least ***two functionality different nodes**.

- The login node, teach01, is where you develop, edit, compile, prepare and submit jobs. You share its resources with other students and courses.
- Real computations should run on the compute nodes
- The login node and compute nodes have the same architecture, operating system, and software stack.
- To run on the compute nodes, you must submit a batch job.
- When running this way, your job get dedicated resources: no contention, less timing fluctuations.

Storage Systems and Locations on SciNet systems

Home and scratch

You have a home and scratch directory on the system, whose locations will be given by

`$HOME=/home/g/groupname/username`

`$SCRATCH=/scratch/g/groupname/username`

Use these convenient variables!

```
teach01:~$ pwd
/home/s/scinet/rzon
teach01:~$ cd $SCRATCH
teach01:~$ pwd
/scratch/s/scinet/rzon
```

Project

Users from groups with a RAC allocation will also have a project directory.

`$PROJECT=/project/g/groupname/username`

Storage Limits on SciNet

location	quota	expiration time	backed up	on login	on compute
$HOME	100 GB	never	yes	yes	read-only
$SCRATCH	25 TB	2 months	no	yes	yes
$PROJECT	by allocation	never	yes	yes	yes

- Compute nodes do not have local storage, but they have a lot of memory, which you can use as if it is local disk `/dev/shm`.
- Backup means a recent snapshot, not an achive of all data that ever was.

Testing

You really should test your code before you submit it to the cluster to know if your code is correct and what kind of resources you need.

- Small test jobs can be run on the login nodes.

  Rule of thumb: couple of minutes, taking at most about 1-2GB of memory, couple of cores.
- You can run the the `ddt` debugger on the login nodes after `module load ddt`.
- The ddt module also gives you the `map` performance profiler.
- Short tests that do not fit on a login node, or for which you need a dedicated node or set of cores, request an interactive debug job with the debugjob command

```
teach01:~$ debugjob -n C
debugjob: Requesting 1 nodes with N tasks for 240 minutes and 0 seconds
SALLOC: Granted job allocation 202753
SALLOC: Waiting for resource configuration
SALLOC: Nodes teach35 are ready for job
teach35:~$
```

Here, C is the number of cores.

The Scheduler

- The scheduler is a program that organizes the work load on the cluster.
- You submit a request to the scheduler, and it will find the right moment for your request to run.
- It does that by looking at the resources available, your priority, times and resources requested, ...
- It is quite a complicated process, with many variables to take into consideration.
- Even when we run 'interactively' (e.g. `debugjob`), we are requesting resources to the scheduler
- We refer to the 'resource request'+'script' as a jobs.

What we will see next is how to communicate with the scheduler and request resources to run our programs.

Submitting jobs

- Teach (as well as all other SciNet and CC systems) uses SLURM as its job scheduler.
- You submit jobs from a login node by passing a script to the sbatch command:

```
teach01:~$ cd $SCRATCH
teach01:scratch/rzon$ sbatch jobscript.sh
```

- This puts the job in the queue. It will run on the compute nodes in due course.

Keep in mind:

- You must use all requested cores.
- Maximum walltime is 24 hours.
- Jobs must write to your scratch directory (home is read-only on compute nodes).
- Compute nodes have no internet access

  Download data you need beforehand on a login node.
- Different clusters have different restrictions.

Example submission script (serial job)

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntask=1
#SBATCH --cpus-per-task=1
#SBATCH --time=1:00:00
#SBATCH --job-name serial_job
#SBATCH --output=serial_output_%j.txt
#SBATCH --mail-type=FAIL

module load gcc/9.2.0

./serial_code 1
```

```
teach01:scratch$ sbatch serial_job.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 (which it gives the name `serial_job`).
- In this case, SLURM looks for one core to be used for one task, for 1 hour.
- Submit from /scratch, as /home is read-only.
- Once SLURM founds a node with an unused core, the script is run there:
  - Loads modules;
  - Sets an environment variable;
  - Runs the `serial_code` app with argument "1".

Monitoring jobs - command line

Once the job is incorporated into the queue, there are some command you can use to monitor its progress.

- "`squeue`" to show the job queue ("`squeue --me`" for just your jobs);
- "`squeue -j JOBID`" to get information on a specific job
  (alternatively,"`scontrol show job JOBID`", which is more verbose).
- "`squeue --start -j JOBID`" to get an estimate for when a job will run.
- "`jobperf JOBID`" to get an instantaneous view of the cpu+memory usage of a running job's nodes.
- "`scancel JOBID`" to cancel the job.
  "`scancel -u USERID`" to cancel all your jobs (careful!).
- "`sinfo -p compute`" to look at available nodes.
- "`sacct`" to get information on your recent jobs.