Optimisation of computing resource usage

for code authors

Gianluca Petrillo

SLAC National Accelerator Laboratory, U.S.A.

ICARUS Collaboration meeting, September 20, 2018

Topics in this talk

I’ll talk of “optimisation”:

≜ using the minimum amount of computing resources (memory, processing power, storage, network bandwidth) needed to complete a task

1. will go through a few recent examples of optimisation
2. will tell about do-it-yourself resource profiling

The tools

CPU time can be measured at two levels:

- *art* `TimeTracker` service reports time for each module
- *ARM Forge map* performs sampling profiling[1]
  - reports how long each function takes to execute
  - licensed by Fermilab, available only on FNAL GPVM

memory can also be measured at two levels:

- *art* `MemoryTracker` service reports memory usage after each code module
- `valgrind massif` tool
  - detailed tracking of what allocates how much memory
  - ×50 slow down compared to regular run

The reports of *none* of these tools are straightforward to interpret.
Discussing the findings with experienced collaborators helps a lot.

[1] That is to keep asking your program “what are you doing now?” (every 10 ms).

Recent optimisation: photon visibility map

- we ask `LArSoft` to use a lookup table to tell which fraction of photons is visible from each point of the TPC
- building that table takes a lot of time and many parallel jobs
- it also used to take gigabytes of memory
  - because it kept track of the whole TPC volume
- by limiting each job to a small part of the TPC volume, the memory required for each job decreased to... negligible

Lesson learned

- a careful workflow choice can make the difference
- some coding was required to make it pay though

Diagnostic tool: `valgrind massif`.

Recent optimisation: services configuration

- each job needs the right set of *art*/`LArSoft` services
- we used to load “all” of them
- e.g., would load the 1.5 GB photon visibility library for nothing
- the service configuration has been reorganised so that presets fit most common situations: `icarus_basic_services`, `icarus_wirecalibration_services`, `icarus_detsim_services`, ...

Lesson learned

- start with no `LArSoft` service, add them as crashes tell you to (tedious and very effective)

Diagnostic tool: `MemoryTracker`.

Configuration files `services_icarus.fcl` and `services_icarus_simulation.fcl` (`fcl/services` path in icaruscode source tree) have some documentation at top of the file.

Recent optimisation: Gaussian hit fitting

- `GausHitFinder` algorithm parametrises a time slice of TPC channel waveform with a superposition of Gaussian "hits"
- on an event with 100k hits, it would takes *hours* and GB of memory
- the code was on each fit creating a new fit function from a string, which ROOT would compile with Cling...

```
TF1 Gaus("Gaus",equation.c_str(),0,roiSize);
```

- replaced with a pool of prebuilt fit functions instead
- now it takes minutes

Lesson learned

- be mindful of side effects when creating ROOT objects

Disgnostic tool: *ARM Forge map*. Inconsistent report needed some creativity.

Recent optimisation: ICARUS hit fitting

- `ICARUSHitFinder` algorithm parametrises time slices of TPC channel waveform with terms based on the form $\frac{e^{-(x-p_1)/p_2}}{1-e^{-(x-p_3)/p_4}}$
- one fit function was computing the *same* exponentials in a loop:

```
for (int js=0; js < floor(par[7*jp+6]); js++) {
  fitval += (1.+js*par[7*jp+7])*(
    par[7*jp+1] +par[7*jp+2]*TMath::Exp(-(x[0]-par[7*jp+3])/par[7*jp+4])
    /(1+TMath::Exp(-(x[0]-par[7*jp+3])/par[7*jp+5]))
    )/(par[7*jp+6]);
}
```

- rewritten the function factorising the repeating terms
- on an event with 100k hits, used to take *hours*; now, a couple of minutes

Lesson learned

- pay attention to the form of the math formulas

Disgnostic tool: *ARM Forge map*, blaming `TMath::Exp` of taking 90% of CPU.

Recent optimisation: PMT signal simulation

- `SimPMTIcarus` algorithm simulates PMT waveforms adding one template photoelectron shape for each scintillation photon reaching the PMT
- photons were added one at a time, which would take too long:

```
for(auto const& ph : photons)
  AddPhoton(ph,fFullWaveforms[photons.OpChannel()]);
```

- rewritten into a two step algorithm:
  1. collect the number of photons arriving at each PMT sampling tick
  2. add for each tick all the photons at once scaling the template
- enabled use special instructions (SIMD), reduced precision (double → single)
- running time halved

Lesson learned

- rethink the code and be willing to pay a bit with memory

Disgnostic tool: *ARM Forge map*, precisely pointing to a $+=$ operation.

Conclusions

- making your code use just the minimum resources is not easy
- yet, at a certain point in the development, it makes sense to spend *half a day* in understanding if there are problems
  - → take a look at a test checklist suggestion
  - → the software/reconstruction group will give you support
- production team should be given one week of time to test *frozen code* before starting the production
  - *without testing, wasted time is typically more than one week*
  - production has, at various times, been seriously slowed down by these issues

Once again: if it's overpowering you, ask the software group for help.

Thank you for your attention

Many thanks to all the people in the software and production groups!

Checklist to test the code

when: when the structure of the code is complete and you are shifting into tuning the physics

input: an unforgiving sample; e.g., if the code is at all supposed to be run on cosmic background, test input sample should contain that background

configuration: use `prof` qualifier; two job configuration (FHiCL) files:

1. preparation: *everything* your code needs as input; run only once!
2. test: only your module(s), services your code needs, `TimeTracker`/`MemoryTracker`, *no ROOT output*

tools: two *local* runs (`icarusgpvm01` or `icarusbuild01`):

- a regular one, with enough events for a 1/2 hour run
- a run profiled with *ARM Forge map*, 2–5 events

alarm bells: → more than 2 GB of resident (RSS) memory: why is that?

→ a single function taking 80% of the time

- can it be made faster?
- can it be called fewer times? or its result be cached?