STATS 769
Code Efficiency

Paul Murrell

The University of Auckland

September 30, 2019

Overview

- This section of the course (two lectures) explores ways to make code more efficient.

- The point of this section is to learn how to measure the time taken to run code and explore where the time is being spent.
- Another point of this section is to move from writing code that works to writing code that works faster (when it makes sense).

Overview

- Do we have a problem?
  Measuring execution time.
- If we have a problem, where is it?
  Profiling.
- If we have a problem, what can we do about it?
  Writing faster code.

Stop and think

- Speed of computation is not the only consideration.
- Your code must get the right answer(!)
- Your code must be understandable (by you and others).
- Your code must be shareable/runnable (for others).
- Reusing existing code is not only convenient, but will be more robust (if that code has been used lots of times by others).
- Human time is more expensive than computer time.

Practical tips

- As with measuring memory usage, if you are unsure, start small and extrapolate before trying the real thing.

- You must be able to stop a runaway process.
- Ctrl-c to "interrupt" a process.
- Ctrl-z to "background" a process.
- `ps aux | grep <UPI>` to find a process.
- `kill -9 <PID>` to "kill" a process.

Measuring execution time

- system.time()
- system.time(replicate())

- User time is CPU time taken by your code
- System time is CPU time taken by the OS on behalf of your code
- Elapsed time is total "wall clock" time from start to finish
- We are typically interested in user time plus system time.

- The **microbenchmark** package offers greater accuracy and convenience (and plots).

Measuring performance

- Real time can be longer if the CPU is doing other things (multiple processes), or is waiting for something else (network traffic or writing files to disk)
- Real time can be shorter if there is more than one CPU

- The CPU and RAM load can have an impact (especially in a multi-user, shared-resource environment)
- The hardware and operating system can have an impact
- The garbage collector can have an impact in R

Profiling

- Measure the time taken in each expression within a function (recursively)
- Where in your code is the most time being spent?

- The `Rprof(filename)` function starts profiling (and records results in a file called `filename`)
- `Rprof(NULL)` stops profiling
- The `summaryRprof(filename)` function displays the results (using the information from `filename`)
- The sampling interval can be controlled with `Rprof(filename, interval=)`

Profiling

- The **profvis** package provides a nice graphical display of profiling results
- `profvis::profvis()` output will be embedded automatically within an R Markdown file that is processed to an HTML format.

- Profiling is stochastic (the profiling information is based on a sample of the call stack)
- Different profiling runs on the same code will produce (slightly) different results
- A function call that executes faster than the sampling interval may not appear in the results
- Call `gc()` before you profile (to make the effect of garbage collection more stable)

Compilation

- The **compiler** package.
- The `cmpfun()` function to compile a function.
- The `compile()` function to compile an expression.

- Core R packages are already compiled
- JIT (just-in-time) compilation of R code is on by default in recent R versions (from 3.4.0)

Writing faster code

- Some functions have faster implementations than others e.g., colMeans() over apply()
- When writing a loop and storing the results, pre-allocate memory, do not "grow" it.
- Vectorised code will typically run faster than a loop.
- Some data structures are much faster to work with than others e.g., matrices vs data frames.

Writing faster code

- `read.csv()` is much faster if it does not have to guess data types.
- Binary files can be much faster to read than text files (as well as being much smaller).
- `data.table::fread()` is much faster than `read.csv()`.
- Data manipulation can also be a lot faster with `data.table`.

Measuring performance from the shell

- `time -p <CMD>`
- NOTE that shell tools piped together are working in parallel!

Other options

- Using a faster machine these days means using more cores rather than a single faster CPU
  (see next topic on parallel computing)

- Using a lower-level language can produce amazing speed ups compared to R
  e.g., write C code and call it from R with the **Rcpp** package
  (see final topic on other systems)

Resources

- "Tidying and profiling R code" in the "Writing R Extensions" manual
  `http://cran.stat.auckland.ac.nz/doc/manuals/r-devel/R-exts.html#Tidying-and-profiling-R-code`
- The "Performance" chapter of "Advanced R" by Hadley Wickham
  `http://adv-r.had.co.nz/Performance.html`
- "FasteR! HigheR StrongeR!" by Noam Ross
  `http://www.noamross.net/blog/2013/4/25/faster-talk.html`