Data Programming with R

Isabella Gollini

Lecture 11 - Performance enhancement and parallel R

Performance enhancement and parallel R

- How to write fast code
- Examples
- Memory issues and profiling
- Parallel R
- Big Data

Writing fast code

- The most important thing is writing code that is correct. Speed should always be a secondary issue.
- When writing fast code we often have to trade-off between the amount of work we want the processor to do and the amount of memory we want to use.
- Often the neatest way of writing fast code is to use mathematics - a simple matrix trick can save you hours (even days) of computation time.
- Other quick tricks for making your R code run faster:
  - Optimise the code through vectorisation and careful avoidance of loops.
  - Parallelise your code (covered this week).
  - Write the most processor-intensive parts of your code in another language, e.g. C or C++ (see lecture 12).
- To optimise your code it is helpful to understand how R allocates memory when you create and manipulate objects.

Useful functions

- Some functions we will cover today:
  - `system.time` will estimate the amount of time a particular command or function takes to run.
  - `Rprof` will attempt to quantify the amount of time your code spends in different functions.
  - `tracemem` will tell you where an object is stored in R's memory.
  - `foreach`, `mclapply` and `ClusterApply` will help parallelise your code to (potentially) make it run faster.

Avoiding for loops

Consider the following two simple functions:

```r
fun1 <- function(x, y) z <- x + y
fun2 <- function(x, y) {
  z <- vector(length = length(x))
  for(i in 1:length(x)) z[i] <- x[i] + y[i]
}
x <- y <- 1:1000000
system.time(fun1(x, y))
```

```
##    user  system elapsed
##   0.005   0.003   0.007
```

```
system.time(fun2(x, y))
```

```
##    user  system elapsed
##   0.124   0.002   0.128
```

Timings can vary between runs. If you want to get a proper estimate run it 10+ times.

You can use the function `benchmark` that is contained in the `rbenchmark` package.

```
rbenchmark::benchmark(fun1(x, y), fun2(x, y), replications = 100)
```

```
##         test replications elapsed relative user.self sys.self user.child
## 1 fun1(x, y)          100   0.382    1.000     0.336    0.045          0
## 2 fun2(x, y)          100  10.522   27.545    10.400    0.092          0
##   sys.child
## 1         0
## 2         0
```

Notes on for loops

- The reason the `for` loop in `fun2` is slow is that we are actually making numerous function calls:
  - `for()` is itself a function
  - The `:` is actually a function. `1:10` is equivalent to `':'(1,10)`
  - Each step of the loop involves running more function calls to both `'['` and `'+'`
- Note that in `fun1` there is only a single call to `+`. The looping is done in the interpreted code (most likely in C).

Extended examples

SC 1

- Snap! - Finding powers of a matrix

Snap

We can compare the two snap functions created in lecture 7. The function `snap1` uses a `for` loop and `snap2` uses `apply`

```
snap1 <- function(numhands) {
  snaptimes <- vector(length = numhands)
  for(i in 1:numhands) {
    p1hand <- sample(cards)
    p2hand <- sample(cards)
    snaptimes[i] <- which(p1hand - p2hand == 0)[1]
  }
  table(snaptimes, useNA = 'always')
}
```

```
snap2 <- function(numhands) {
  snap <- function(handdiff) which(handdiff == 0)[1]
  bighand1 <- apply(matrix(rep(cards, numhands), 52, numhands), 2, sample)
  bighand2 <- apply(matrix(rep(cards, numhands), 52, numhands), 2, sample)
  snaptimes2 <- apply(bighand1 - bighand2, 2, snap)
  table(snaptimes2, useNA = 'always')
}
```

More Snap

Timings:

```
cards <- 1:52
rbenchmark::benchmark(snap1(10000), snap2(10000), replications = 10,
  columns = c("test", "replications", "elapsed", "relative",
    "user.self", "sys.self"))
```

```
##           test replications elapsed relative user.self sys.self
## 1 snap1(10000)           10   1.502    1.000     1.397    0.098
## 2 snap2(10000)           10   2.347    1.563     2.236    0.104
```

- `snap2` is almost twice as slow - but we're using `apply`!?
- `snap2` actually involves 3 calls to `apply`, but `snap1` just has a single loop with small variables being over-written - this is again the trade-off between CPU and RAM.
- `apply` is actually written in R code and uses `for` loops (unlike `sapply` and `lapply`) so can be slow.

Finding powers of a matrix

In the `polyreg` example in Lecture 8, we needed to create a matrix where each column is raised to a power (here `deg`). We can do this a few different ways:

```
powers1 <- function(x, deg) {
  pw <- matrix(x, nrow = length(x))
  prod <- x
  for(i in 2:deg) {
    prod <- prod * x
    pw <- cbind(pw, prod)
  }
  return(pw)
}
```

```
powers2 <- function(x, deg) {
  pw <- matrix(nrow = length(x), ncol = deg)
  pw[,1] <- prod <- x
  for(i in 2:deg) {
    prod <- prod * x
    pw[,i] <- prod
  }
  return(pw)
}
```

```
powers3 <- function(x, deg) {
  Xmat <- sweep(matrix(rep(x, deg), ncol = deg, nrow = length(x)),
                2, 1:deg, '^') }
```

```
x <- runif(1000000)
rbenchmark::benchmark(powers1(x, 8), powers2(x, 8), powers3(x, 8), replications = 10)
```

```
##            test replications elapsed relative user.self sys.self
## 1 powers1(x, 8)           10   1.534    1.543     1.055    0.476
## 2 powers2(x, 8)           10   0.994    1.000     0.620    0.372
## 3 powers3(x, 8)           10   3.438    3.459     3.112    0.316
##   user.child sys.child
## 1          0         0
## 2          0         0
## 3           0           0
```

`powers2` was the fastest! - Why? Because the loop was short and required very little memory.

Notes about speed

- Most of the time `apply` is faster than a `for` loop.
- However, if you need to use repeated calls to `apply`, or with particularly slow functions being apply'd things will slow down and the memory load will increase.
- If you're going to create a vector/matrix/array, it's best to declare the full size up front rather than using e.g. `cbind` or `rbind`.
- Timings can also be very variable and strongly depend on the computer you're using. It's often convenient to run timings a few times to get a stable result.
- Think carefully about the memory/processor trade off when trying to write fast code.

Memory issues and profiling

- Problems with assigning vectors
- `Rprof`
- Profiling the powers functions
- What does `Rprof` do?

Vector assignment

A simple command such as:

```
z[3] <- 8
```

is a bit more complicated than it looks. Remember that `[` is actually a function and what is really being run is:

```
z <- '[<-'(z, 3, 8)
```

. . . as this is a replacement function

What's really happening in the background is that a whole new version of `z` is created in memory with the new value `8` in the third position. When you type `z` R gives the new value of `z` it created rather than the old one

Thus though we are only changing one element, an entire new variable is created!

Copy on change

If we set two vectors to be equal, as in:

```
y <- z
```

Then `y` and `z` occupy the same memory location. You can find this use the `tracemem` function:

```
tracemem(z)
```

```
## [1] "<0x7fba0034c478>"
```

```
tracemem(y)
```

```
## [1] "<0x7fba0034c478>"
```

This is nice and efficient, but as soon as we change one element of either `z` or `y`, we will be using extra memory. This can be problematic if the vectors are big.

Once you've changed one element however, it won't take up any more memory to change other elements.

```r
y[1] <- 2
tracemem(z)
```

```
## [1] "<0x7fba0034c478>"
```

```r
tracemem(y)
```

```
## [1] "<0x7fba007b3408>"
```

Profiling code

R has a helpful function for when code is running slowly called `Rprof`.

This gives you a report of how much time your main function spent in the different functions that it calls.

The way we use it is:

```r
Rprof()
# Your code/functions
Rprof(NULL)
summaryRprof()
```

We can use it to profile our different `powers` functions to see where they get stuck.

`Rprof` produces a `list`, the first tag is `by.self` which is time spent in the function itself and the second is `by.total` which is time spent in the function and other functions that it calls

Profiling the powers functions

```r
Rprof()
invisible(powers1(x, 8))
Rprof(NULL)
summaryRprof()$by.self
```

```
##           self.time self.pct total.time total.pct
## "cbind"        0.12       75       0.12        75
## "powers1"      0.04       25       0.16       100
```

`powers1` spent most of the time in `cbind` (nearly 90%) - this must be a slow function and we should try and remove it.

Note: `invisible` used here to stop printing out the matrix.

```r
Rprof()
invisible(powers2(x, 8))
Rprof(NULL)
summaryRprof()$by.self
```

```
##           self.time self.pct total.time total.pct
## "powers2"      0.08      100       0.08       100
```

`Rprof` doesn't provide any useful information about `powers2`.

Profiling `powers3`.

```r
Rprof()
invisible(powers3(x, 8))
Rprof(NULL)
summaryRprof()$by.self
```

```
##                 self.time self.pct total.time total.pct
## "sweep"              0.18    64.29       0.28    100.00
## "aperm.default"      0.06    21.43       0.06     21.43
## "matrix"             0.02     7.14       0.04     14.29
## "is.atomic"          0.02     7.14       0.02      7.14
```

`powers3` spends about two-thirds of its time in `sweep` which creates arrays and permutes them - not quite as fast as I would have liked.

What does `Rprof` do?

- Every 0.02 seconds, `Rprof` finds out which functions are currently being called and writes them to a text file called `Rprof.out` - you'll find it in your working directory. Here are some sample lines from profiling `powers3`:

```
"matrix" "sweep" "powers3"   "matrix" "sweep" "powers3"   "array" "aperm" "sweep" "powers3"
...
```

- So, in the last line inspected here, the function `powers3` had called `sweep` which had called `aperm` which had called `array`.
- Two final notes: `Rprof` doesn't always help - sometimes it's hard to read or doesn't identify useful blockages.
- See the package `profvis` for more advanced tools.

Parallel R

SC 2

- Why parallel? -`parallel`
- `foreach`
- `snow`
- Why doesn't parallelisation always help?

Why parallel?

- Many computers these days have multiple cores to which we can send programming instructions.
- If the problem we are working on can be broken down into mutually exclusive parts then performing steps in parallel can greatly increase the speed of our code.
- In this section we will go through a few different packages which allow us to 'parallelise' our R code.
- Note: many of you will be working on computers with 2 or more cores. If you're not you may struggle with running some of the code in this section.

The `parallel` package contains a function for working out how many cores you have available:

```r
library(parallel)
detectCores()
```

```
## [1] 4
```

A motivating example

Consider a simple example where we want to find the average of the column sums of a matrix:

```
n <- 5000000 # Number of rows
m <- 5 # Number of columns
M <- matrix(rnorm(n * m), nrow = n, ncol = m)
mycolsum <- function(i) {
  return(sum(M[,i]))
}
```

Non-parallel version

```
try1 <- sum(sapply(1:m, mycolsum)) / m
```

The `foreach` package

The `foreach` package (which also uses `doParallel`) is possibly the simplest implementation of parallel processing in R.

It works the same as a for loop but gives each element of the loop to a different core.

```
library(doParallel)
registerDoParallel(cores = 2)
library(foreach)
try2 <- sum(unlist(foreach(j = 1:m) %dopar% mycolsum(j))) / m
```

The `registerDoParallel` command tells R how many cores we want to use, while `foreach` does all the hard work in allocating jobs to cores.

The `parallel` package

The parallel package includes simple-to-use tools to parallelise some tasks without too much user input.

The key functions are `pvec` which parallelises vector functions and `mclapply` which is a multi-core version of `lapply`:

```
try3 <- sum(unlist(mclapply(1:m, mycolsum, mc.cores = 2))) / m
```

Note that the arguments to `mclapply` are exactly the same as `lapply`, aside from the final argument to `mc.cores`.

The `snow` package

- The `snow` (simple network of workstations) package is much more advanced than `foreach` and `parallel` and has many more functions and allows for far greater flexibility.
- With `snow` you have to follow a general structure in setting up your code:
  1. Use the `makeCluster` function to tell it how many cores you want to use and what they're called.
  2. You split the number of function calls required across the number of cores you want to use - this gives you greater control than `foreach` or `mclapply`.
  3. You run the function `clusterApply` with the function you want to run, the cores you have specified and the order in which the function calls should be given to each core.

```
library(snow)
cl <- makeCluster(type = 'SOCK', rep('localhost', 2))
num.cores <- length(cl)
mycolsum2 <- function(i, M) {
  return(sum(M[,i]))
}
chunks <- split(1:m, 1:num.cores)
try4 <- sum(unlist(clusterApply(cl, chunks, mycolsum2, M))) / m
```

- The `makeCluster` function here specifies that you want to use your own computer (`'localhost'`).
- Side note: some of the above code gives warnings which you can safely ignore.

Who's the fastest?

```
ftry1 <- function(m) sum(sapply(1:m, mycolsum)) / m
ftry2 <- function(m) sum(unlist(foreach(j = 1:m) %dopar% mycolsum(j))) / m
ftry3 <- function(m) sum(unlist(mclapply(1:m, mycolsum, mc.cores = 2))) / m
ftry4 <- function(m) sum(unlist(clusterApply(cl, chunks, mycolsum2, M))) / m
rbenchmark::benchmark(ftry1(m), ftry2(m), ftry3(m), ftry4(m), replications = 10)
```

```
##       test replications elapsed relative user.self sys.self user.child
## 1 ftry1(m)           10   1.637    1.000     1.149    0.483      0.000
## 2 ftry2(m)           10   2.553    1.560     0.052    0.058      1.900
## 3 ftry3(m)           10   2.421    1.479     0.012    0.037      1.874
## 4 ftry4(m)           10  22.254   13.594    11.864    1.128      0.000
##   sys.child
## 1     0.000
## 2     2.268
## 3     2.254
## 4     0.000
```

Why didn't parallelisation help?

- Logically, we might expect that using 2 cores will produce code that is twice as fast as using 1 core.
- Unfortunately it doesn't usually work out that way - the different function calls need to be sent to each processor, and then often collected together again at the end.
- When the different jobs take variable amounts of time this can lead to a lot of latency, slowing things down too - `snow` has a function `clusterApplyLB` which can help balance the loads on different cores.
- The `snow` package also allows you to send different functions, packages, etc, to different cores.

Big Data

- What can you do in R?
- Sampling big data
- sampling chunking
- R packages for big data
- Hadoop and MapReduce.

What can you do with big data?

- The term 'big data' has become popular over the last few years.
- This is largely due to the growth of information collected by governments and companies - very few people are qualified or even able to analyse it.
- We refer to big data as anything than can't fit in the memory of the computer you're using.
- There are various technical and non-technical solutions to dealing with big data. In the following slides we will go through five of them

Solution 1: sampling

- Statisticians have been dealing with big data for centuries - they solved the problem by taking samples.
- If the big data are of the order of millions of observations, then a sample of a few thousand will yield predictions to within a margin or error of a few percent.
- We have already met the function sample to take samples from data.
- You can also leave out observations with the `read.table` family of functions via the `skip` argument.
- However, this doesn't solve the problem of data that are 'fat' - with many more columns than rows.

Solution 2: maths

- As with all computational tasks knowing a little bit of maths (especially matrix algebra) will enable a huge speed up.
- For example, fitting a linear regression (e.g. via maximum likelihood with `optim`) using millions of observations will be very slow, but the matrix solution for the regression coefficients after doing a little bit of algebra are:

$$(X^TX)^{-1}X^Ty$$

- These matrices all have dimension p, where p is the number of explanatory variables rather than n, so provided $p << n$ then this will be fast
- Lesson: learn more matrix algebra!
- Use this very useful reference: The Matrix Cookbook http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/3274/pdf/imm3274.pdf

Solution 3: chunking

- You don't have to load in all your data in one go.
- We've already mentioned the skip argument in `read.table`: we might use this to calculate statistics (e.g. the sum of a variable) as we go, to save us loading in the whole object.
- We can also keep data in binary format which tends to be smaller.
- R has some useful packages for loading in big data sets, for example `ff` and `ffbase`, which have functions `save.ffdf` and `load.ffdf`.
- These functions load and save objects in pieces in a directory - they then have methods to create means, variances, etc for all the standard types of analysis you might like to do.

Solution 4: Use an R package

- R has several packages that are designed analyse big data sets:
  - `biglm` fits standard `glm` models (e.g. the logistic regression models we have covered) to very large data sets. It has its own summary and print methods so requires almost no extra knowledge.
  - The `pdbR` prject (Programming with Big Data in R - https://pbdr.org/) is a set of R packages that allows you to install R on a high performance computer. This will then take advantage of better algorithms for parallelisation.
  - `h2o` R Interface for H2O an open-source software for scalable implementations of GLM, random forests, and deep learning. http://docs.h2o.ai/
- A general list of packages for high performance computation is here: http://cran.r-project.org/web/views/HighPerformanceComputing.html

Solution 5: advanced tools.

- You don't have to just use R or RStudio on your machine. You could use different versions of R, e.g.:
  - RStudio on Amazon EC2 - http://www.louisaslett.com/RStudio_AMI/
  - Microsoft R Open (formerly Revolution R Open) is an enhanced R version from Microsoft: https://mran.microsoft.com/open
- `Hadoop` and `MapReduce` are programming frameworks for big data sets.

    - `Hadoop` allows the distributed processing of large data sets across clusters of computers.
    - `MapReduce` works by splitting up the data, analysing it and recombining it.

Lessons from this week

- Use maths, parallel processing and apply-type functions (sometimes) to speed up your code.
- `for` loops are bad in general (but not always so).
- If you have code that is running too slowly then look for bottlenecks with `Rprof`.
- You can almost always analyse big data with R, just choose the right analysis method.