



How to write an efficient script

Chin Yang Shapland

MRC Integrative Epidemiology Unit

6 June 2022

Introduction

This session will cover;

- ✿ Identifying bottlenecks
- ✿ The apply family
- ✿ Parallelizing a script
- ✿ High performance computing

Identifying bottlenecks: `system.time()`

```
system.time({  
  n <- 1000  
  r <- numeric(n)  
  for(i in 1:n) {  
    x <- rnorm(n)  
    r[i] <- mean(x)  
  }  
})  
  
##      user  system elapsed  
##      0.11    0.00    0.11
```

`user` is the time taken to execute the code, `system` is time for the system to open and close files, and `elapsed` is the stopwatch time, i.e. total time code took from start to finish.

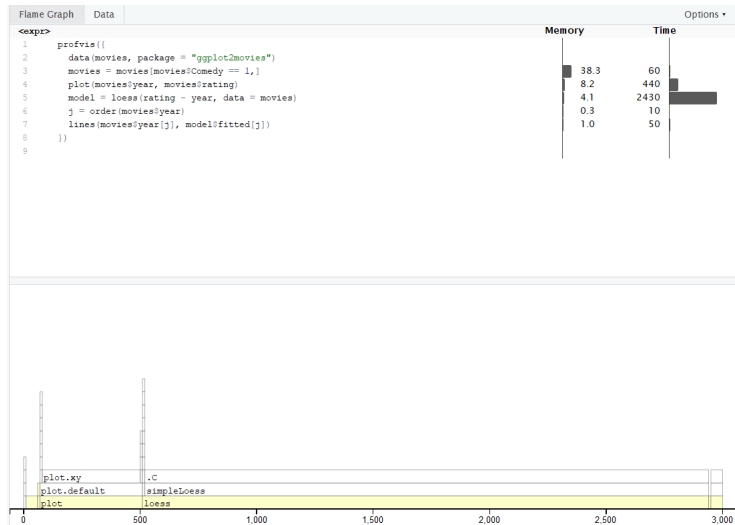
Identifying bottlenecks: The R Profiler

```
library(profvis)
library(bench)
library(ggplot2movies)

profvis({
  data(movies, package = "ggplot2movies")
  movies = movies[movies$Comedy == 1,]
  plot(movies$year, movies$rating)
  model = loess(rating ~ year, data = movies)
  j = order(movies$year)
  lines(movies$year[j], model$fitted[j])
})
```

Output

Outputs a tab called “Profile1”;



The apply family

are faster and safer than loops. We will cover

- ✿ `apply()`
- ✿ `lapply()`
- ✿ `sapply()`

but there are also others,

- ✿ `vapply()`
- ✿ `mapply()`
- ✿ `rapply()`
- ✿ `tapply()`

apply()

```
str(apply)  
## function (X, MARGIN, FUN, ..., simplify = TRUE)
```

X is the input, MARGIN should be row (1) or column (2), or both (c(1,2)), FUN is the function. Depending what the input is, it can return a vector, list, matrix or array.

apply() example

```
data_fun<-matrix(c(1:4,1:4,1:4,1:4),4,4)
```

```
data_fun
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    1    1    1  
## [2,]    2    2    2    2  
## [3,]    3    3    3    3  
## [4,]    4    4    4    4
```

```
apply(data_fun,2,sum)
```

```
## [1] 10 10 10 10
```

```
apply(data_fun,2,mean)
```

```
## [1] 2.5 2.5 2.5 2.5
```


apply() example with function

```
data_fun<-matrix(c(1:4,1:4,1:4,1:4),4,4)
```

```
data_fun
```

```
##           [,1] [,2] [,3] [,4]
## [1,]        1    1    1    1
## [2,]        2    2    2    2
## [3,]        3    3    3    3
## [4,]        4    4    4    4
```

```
apply(data_fun, 1:2, function(x) x/2)
```

```
##           [,1] [,2] [,3] [,4]
## [1,]    0.5  0.5  0.5  0.5
## [2,]    1.0  1.0  1.0  1.0
## [3,]    1.5  1.5  1.5  1.5
## [4,]    2.0  2.0  2.0  2.0
```

apply vs loops

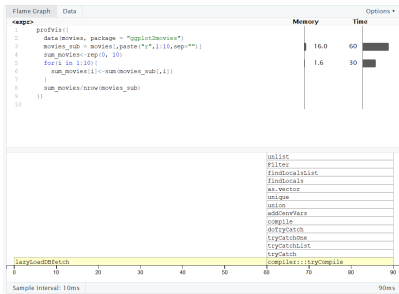
Using for loop

```
profvis({  
  data(movies, package = "ggplot2movies")  
  movies_sub = movies[,paste("r",1:10,sep="")]  
  sum_movies<-rep(0, 10)  
  for(i in 1:10){  
    sum_movies[i]<-sum(movies_sub[,i])  
  }  
  sum_movies/nrow(movies_sub)  
})
```

apply vs loops

```
# Using `apply()` function;  
profvis({  
  data(movies, package = "ggplot2movies")  
  movies_sub = movies[,paste("r",1:10,sep="")]  
  apply(movies_sub, 2, mean)  
})
```

apply vs loops



Inputs matters

Previous examples used matrices how about data frame?

```
data_df<-data.frame(1:4,1:4,1:4,1:4)
str(data_df)
## 'data.frame':    4 obs. of  4 variables:
##  $ X1.4   : int  1 2 3 4
##  $ X1.4.1: int  1 2 3 4
##  $ X1.4.2: int  1 2 3 4
##  $ X1.4.3: int  1 2 3 4

apply(data_df, 2, mean)
##    X1.4 X1.4.1 X1.4.2 X1.4.3
##    2.5    2.5    2.5    2.5
```

Inputs matters

```
data_df$Day<-as.factor(1:4)
str(data_df)
## 'data.frame':    4 obs. of  5 variables:
##  $ X1.4   : int   1 2 3 4
##  $ X1.4.1: int   1 2 3 4
##  $ X1.4.2: int   1 2 3 4
##  $ X1.4.3: int   1 2 3 4
##  $ Day    : Factor w/ 4 levels "1","2","3","4": 1 2 3 4

apply(data_df, 2, mean)
##   X1.4 X1.4.1 X1.4.2 X1.4.3   Day
##    NA    NA    NA    NA    NA
```

... options

... within the function have multiple options but the most option is to use for handling missing data

```
data_miss<-matrix(c(1:4,c(1:3,NA),1:4,1:4),4,4)
```

```
data_miss
```

```
##           [,1] [,2] [,3] [,4]
## [1,]        1    1    1    1
## [2,]        2    2    2    2
## [3,]        3    3    3    3
## [4,]        4   NA    4    4
```

```
apply(data_miss, 2, mean)
```

```
## [1] 2.5  NA 2.5 2.5
```

```
apply(data_miss, 2, mean, na.rm=T)
```

```
## [1] 2.5 2.0 2.5 2.5
```

lapply()

```
str(lapply)  
## function (X, FUN, ...)
```

X is the input, FUN is the function. Only returns list object.

lapply() example

```
list_fun<-list(a = 1:5, b = 10:15, c = 21:25)
list_fun
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 10 11 12 13 14 15
##
## $c
## [1] 21 22 23 24 25
```

lapply() example

```
lapply(list_fun,sum)
```

```
## $a
```

```
## [1] 15
```

```
##
```

```
## $b
```

```
## [1] 75
```

```
##
```

```
## $c
```

```
## [1] 115
```

lapply() example

```
lapply(list_fun,mean)
```

```
## $a
```

```
## [1] 3
```

```
##
```

```
## $b
```

```
## [1] 12.5
```

```
##
```

```
## $c
```

```
## [1] 23
```

lapply() example with function

```
lapply(list_fun, function(x) x/2)
## $a
## [1] 0.5 1.0 1.5 2.0 2.5
##
## $b
## [1] 5.0 5.5 6.0 6.5 7.0 7.5
##
## $c
## [1] 10.5 11.0 11.5 12.0 12.5
```

lapply() example with vectors

```
Random<-c("these", "are", "random", "example")
lapply(Random, nchar)
## [[1]]
## [1] 5
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 7
```

sapply()

This is a simplified form of lapply()

```
str(sapply)
```

```
## function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

X is the input, FUN is the function. If simplify is changed to F then it becomes lapply() and returns only list object. Otherwise it also outputs vector, matrix, array or list.

sapply() example

```
list_fun<-list(a = 1:5, b = 10:15, c = 21:25)
list_fun
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 10 11 12 13 14 15
##
## $c
## [1] 21 22 23 24 25

sapply(list_fun,sum)
##      a      b      c
##    15    75   115
```

sapply() example

```
list_fun<-list(a = 1:5, b = 10:15, c = 21:25)
list_fun
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 10 11 12 13 14 15
##
## $c
## [1] 21 22 23 24 25

sapply(list_fun,mean)
##      a      b      c
##  3.0 12.5 23.0
```


sapply() example

```
sapply(list_fun, function(x) x/2)
## $a
## [1] 0.5 1.0 1.5 2.0 2.5
##
## $b
## [1] 5.0 5.5 6.0 6.5 7.0 7.5
##
## $c
## [1] 10.5 11.0 11.5 12.0 12.5
```

Notice that this is given as a list rather than a vector, this is because each element have different lengths

Others

`vapply()`

similar to `sapply` but offers a third argument which sets a template for the output.

`mapply()`

multivariate version to `sapply` function, but it applies functions to the argument one by one.

`rapply()`

Similar to `lapply` but you can specify the structure of the output

`tapply()`

applies a function to numeric data distributed across various categories.

Parallelizing a script: when

When to parallelize?

“its important to evaluate the computational efficiency of requests, and work to ensure that additional compute resources brought to bear will pay off in terms of increased work being done.”

Parallelizing a script: how

When parallelizing jobs, one can:

- ✶ Use the multiple cores on a local computer through `mclapply`
- ✶ Use multiple processors on local (and remote) machines using `makeCluster` and `clusterApply` (Note this approach, one has to manually copy data and code to each cluster member using `clusterExport`. This is extra work, but sometimes gaining access to a large cluster is worth it.)

Parallel package: mclapply

```
library(parallel)
library(MASS)

starts <- rep(100, 40)
fx <- function(nstart) kmeans(Boston, 4, nstart=nstart)
numCores <- detectCores()
numCores
## [1] 8

system.time(
  results <- lapply(starts, fx)
)
##      user  system elapsed
##      1.06    0.03     1.11
```

Parallel package: mclapply

```
system.time(  
  results <- mclapply(starts, fx, mc.cores = numCores)  
)
```

```
##      user  system elapsed  
##    0.801    0.178    0.367
```

foreach package: %do% and %dopar%

```
for (i in 1:3) {  
  print(sqrt(i))  
}  
## [1] 1  
## [1] 1.414214  
## [1] 1.732051
```

In foreach package %do% evaluates the expression sequentially,

```
library(foreach)  
foreach (i=1:3, .combine=c) %do% {  
  sqrt(i)  
}  
## [1] 1.000000 1.414214 1.732051
```

foreach package with doParallel

while %dopar% evaluates it in parallel, but %dopar% has to work with doParallel package

```
library(foreach)
library(doParallel)
```

Use multicore, set to the number of our cores

```
registerDoParallel(numCores)
foreach (i=1:3, .combine=c) %dopar% { sqrt(i) }
## [1] 1.000000 1.414214 1.732051
```


%do%

Using the iris data set to do a parallel bootstrap from the doParallel vignette

```
x <- iris[which(iris[,5] != "setosa"), c(1,5)]
trials <- 10000

system.time({
  r <- foreach(icount(trials), .combine=rbind) %do% {
    ind <- sample(100, 100, replace=TRUE)
    result1 <- glm(x[ind,2]~x[ind,1],
                   family=binomial(logit))
    coefficients(result1)
  }
})

##      user  system elapsed
##  20.04    0.00   20.06
```

%dopar%

```
system.time({  
  r <- foreach(icount(trials), .combine=rbind) %dopar% {  
    ind <- sample(100, 100, replace=TRUE)  
    result1 <- glm(x[ind,2]~x[ind,1],  
                  family=binomial(logit))  
    coefficients(result1)  
  }  
})  
  
##      user  system elapsed  
##      2.98    0.47    6.47
```

Outputs

To simplify output, foreach has the .combine parameter that can simplify return values

```
# Default: Return a list  
foreach (i=1:3) %dopar% { sqrt(i) }  
## [[1]]  
## [1] 1  
##  
## [[2]]  
## [1] 1.414214  
##  
## [[3]]  
## [1] 1.732051
```

Outputs

Return a vector

```
foreach (i=1:3, .combine=c) %dopar% { sqrt(i) }
```

```
## [1] 1.000000 1.414214 1.732051
```

Return a data frame

```
foreach (i=1:3, .combine=rbind) %dopar% { sqrt(i) }
```

```
##           [,1]
```

```
## result.1 1.000000
```

```
## result.2 1.414214
```

```
## result.3 1.732051
```

Further tips parallel computing

- ✶ To clear up and stop parallel use;

```
stopImplicitCluster()
```

- ✶ To set the cluster seed if you want reproducible results, don't use `set.seed()`, use `clusterSetRNGStream()` instead

High performance computing

When all else fails running your script on a high powered server:

BlueCrystal Phase 4

```
#!/bin/bash
```

```
#SBATCH --job-name=job_name
```

```
#SBATCH --output=job_name
```

```
#SBATCH --error=job_name
```

```
#SBATCH --nodes=1
```

```
#SBATCH --ntasks-per-node=16
```

```
#SBATCH --time=2:0:0
```

```
module load languages/r/4.1.0
```

```
R CMD BATCH /your_dir/your_Rscript.R
```

Check your shell script!

Use **cat -v** command within BlueCrystal, as sometime when you write your shell script(.sh) in RStudio or notepad it leaves a bunch of none printable characters like this;

```
#!/bin/bash^M
^M
#SBATCH --job-name=job_name^M
#SBATCH --output=job_name^M
#SBATCH --error=job_name^M
#SBATCH --nodes=1^M
#SBATCH --ntasks-per-node=16^M
#SBATCH --time=2:0:0^M
^M
module load languages/r/4.1.0^M
^M
Rscript /your_dir/your_Rscript.R^M
```

Solution: Check your shell script!

To remove these characters, use a text editor in BlueCrystal, such nano, emacs or vi (very user friendly editors).

References

Help from HPC <https://www.bristol.ac.uk/acrc/high-performance-computing/hpc-documentation-support-and-training/>

Top 5 tips for efficient performance by Colin Gillespie
<https://csgillespie.github.io/efficientR/performance.html#top-5-tips-for-efficient-performance>

Quick Intro to Parallel Computing in R by Matt Jones
<https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html>

Join “code-clinic”! Email Louise Millard