Statistical programming using R

Lecture 7 – Miscellaneous

Introduction

This lecture is not examinable. It is not compulsory. It includes topics that should make your life a bit easier, tools that are helpful but do not fit in other lectures.

The first topic is *R Markdown*. Presentation of your research is very important. You already know some tools, e.g. MS Word, LaTeX. You can prepare very good documents using R.

Next topic is about different ways limited resources we have while working with large data sets. R as programming language has a big problem with performance. It can use only only one core on the CPU while all modern computers have 4-6-8 cores even in the most "modest" configurations. There are tools helping to employ full power of your computers. Another limiter resource is memory (RAM) – there is never enough memory. There are tools to solve this problem too.

R Markdown

Good presentation is very important. You are as good as your report. If your report looks poor, no one will read it and no one will know you are an excellent researcher.

The most popular application to make reports is Microsoft Word. Millions of people use it everyday to prepare their documents. MS Word follows WYSIWYG approach – What You See Is What You Get. People spend a lot of time to make their reports look nice. In general, work in MS Word can be split between creating content and visual appearance as 50/50.

You should know another tool and another approach – LaTeX. LaTeX focuses mostly on content and visual appearance will be applied/added later and automatically according to some rules. The process might be as simple as adding a journal profile and you get your report formatted according to design rules of this journal.

Markdown [https://www.markdownguide.org/] is a free and open-source markup language. You can create almost any types of documents using Markdown. Similar to LaTeX, Markdown focuses on content only. You don't see how the final document will look like until you compile the document. Unlike LaTeX, Markdown is very simple.

Markdown is supported by all operating systems and by many different applications including cloud-based as web servers Reddit or Github. Obviously, there is a support for Markdown in R and RStudio can be used to create many different types of documents using Markdown. You just select in the menu File -> New File -> R Markdown and start creating your content.

Special feature of R Markdown is an ability to combine text content and R code. Code can be executed and provide results and visualisations for the report that will be embedded in the document automatically.

All my lecture notes where created in using R Markdown in RStudio. It is a bit difficult to make a quick and nice R Markdown document about creating R Markdown documents, hence I will use an online manual for RStudio R Markdown for presentation.

- Home page of R Markdown – https://rmarkdown.rstudio.com/
- Brief step-by-step tutorial – https://rmarkdown.rstudio.com/lesson-1.html
- Reference guide (very important) – https://rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf

High performance computing

Sometime we need to do analysis that is BIG. "Big" can mean large data set or a lot of calculations or both. In general, R is not so good in terms of performance. It needs data loaded into a memory (random access memory – RAM) and it can use only one processor (central processing unit – CPU) for calculations. Fortunately, there are packages that can extend functionality of R and allow to go beyond R limitations.

Parallel processing

All modern computers have multiple CPUs. Even the most basic laptop has between 4 and 8 CPU cores that can work in parallel. Here are examples of a standard sequential calculations:

```
# prepare data
mydata <- seq(1,10)

# using apply functions - calculate square root of each number
sapply(mydata, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

```
# the same task using functionality of purrr package
library(purrr)
mydata %>% map_dbl(sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

```
# the same task with package foreach
suppressMessages(library(foreach))
foreach(i = mydata, .combine = c) %do% {
  sqrt(i)
}
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

Function `foreach()` creates a for-loop. Using loops should be discouraged in R as loops are inefficient. However, sometimes it is impossible to avoid a loop and sometimes we do need to run many thousands of iterations. One of the benefits of the package `foreach` is the output format. You can have results of the iterations stored as any type of structure based on the argument `.combine`.

We can try to run above three functions and benchmark time required for the same operation.

```
# prepare larger data set
mydata <- list(seq(1, 10^8))
mydata <- rep(mydata, 7)

library(rbenchmark)  # load a package for benchmark function

benchmark(

  # apply function
  sapply(mydata, mean),

  # purr package
  mydata %>% map_dbl(mean),

  # foreach package
  foreach(i = mydata, .combine = c) %do% { mean(i) },

  replications = 1,
  columns = c("test", "elapsed", "relative")
)
```

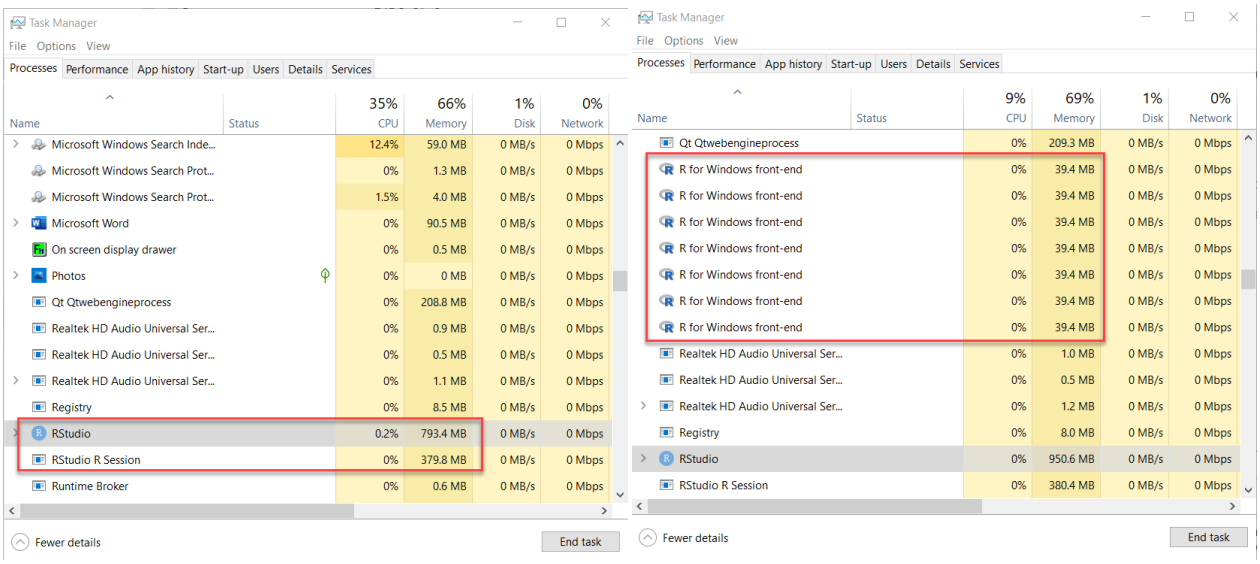
```
##                                                    test elapsed relative
## 3 foreach(i = mydata, .combine = c) %do% {\n    mean(i)\n}    2.46    1.000
## 2                                 mydata %>% map_dbl(mean)    2.47    1.004
## 1                                     sapply(mydata, mean)    2.47    1.004
```

Results are the same as all these functions do the same job and mostly in the same way. They use only one CPU core. However, my laptop has 8 cores. Hence, theoretically speaking I can run the same task 8 times quicker. I just need to make a cluster and explain R how to use it. Fortunately, I don't need to do it manually – there is a package for that.

```
# check how many cores available
parallel::detectCores(all.tests = FALSE, logical = TRUE)
```

```
## [1] 6
```

```
# load package for parallel processing
suppressMessages(library(doSNOW))

# make a cluster of 7 cores
cl <- makeCluster(7, type="SOCK")

# start the cluster
registerDoSNOW(cl)
```



"Workers" are ready and I can run the same `foreach()` function as above but change it to parallel processing

```
#### parallel processing ####

# check time for foreach function
system.time(
  foreach(i = mydata, .combine = c) %dopar% { mean(i) }
)
```

```
##    user  system elapsed
##    0.02    0.00    0.54
```

```
# stop the cluster after finishing the job
stopCluster(cl)
```

Unlike `foreach` package that can be used on single and multiple CPUs, `purrr` can run on one CPU only. However, there is a package `furrr` that replicates functionality of `purrr` but for multiple CPUs.

```
# load the library
suppressMessages(library(furrr))

# set the plan of calculation - multiple sessions on 7 workers
plan("future::multisession", workers = 7)

system.time(
  # future_map function repeats map function from purrr
  mydata %>% future_map_dbl(mean)
)
```

```
##    user  system elapsed
##    0.06    0.00    1.23
```

```
# close off multi CPU cluster
plan(sequential)
```

Both functions `foreach()` and `future_map()` were quicker that single CPU functions but not seven times quicker as we might expect. Parallel processing has a lot of computational overheads related to cluster management, loading data, extracting and combining results. So, there is no real value to use multiple CPUs if the task is relatively small. If the task is large, requires a lot of iterations and complex calculations in each iteration, then overheads become relatively small compared to main calculations. As a result, speed improvement becomes significant and close to the number of core employed.

Each extra session of R acts as an independent virtual machine with R running on it. Hence, if you want to do some data analysis, you have to load the data set and required packages inside each R session. If the data set size is $2Gb$, then to run data analysis on 7 cores in parallel we need at least $7 * 2 = 14Gb$ memory. That might be beyond of what we have available. So, parallel processing has its downsides and limitations.

Managing large data sets

To analyse any data set in R, it should be loaded into a memory. Also, some extra memory should be available for running functions and to store results. What to do if there is not enough memory to load the data? Well, we can leave data on the hard drive and work with is as it is loaded in the RAM. Hard drive is not that quick as RAM. Even the quickest solid-state hard-drives are 100 times slower than the RAM. However, if the data set is really large, then using hard drive is the only way to process data.

As you should guess, there is a package – `ff`. It allows to create a special type of objects: ff-vector, ff-matrix, ff-array, ff-dataframe. Most of the data loaded into these objects are stored on the hard drive, while there is a relatively small amount of memory occupied by the data related information. Then there is a package `ffbase` for basic manipulations with ff objects.

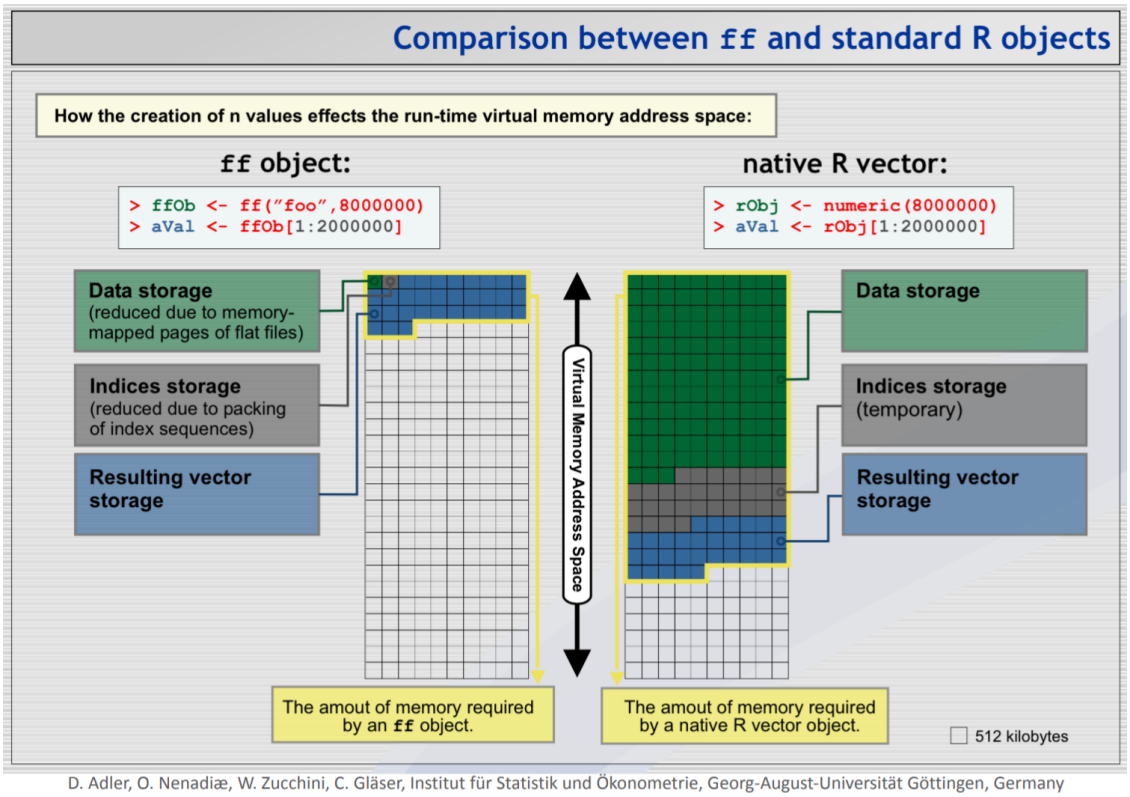


Figure 1: "ff object versus R object"

When we run any function on ff-object, data are loaded into memory by chunks, so it is possible to analyse the entire data set even if it is too large for the computer memory. Here is an example and comparison of

the normal R object and ff-object.

```
# clean up memory
rm(list = ls()); gc()
```

```
##           used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  630008 33.7    1091302 58.3  1091302 58.3
## Vcells 1201127  9.2    8388608 64.0  1927773 14.8
```

```
# check memory use by R
memory.size()
```

```
## [1] 70.34
```

```
# create very large integer vector
x <- integer(2^30)

# check used memory again - more than 4Gb, that's a lot
memory.size()
```

```
## [1] 4157.17
```

```
# delete x and clean up memory again
rm(x); gc()
```

```
##           used (Mb) gc trigger   (Mb)  max used   (Mb)
## Ncells  630068 33.7    1091302   58.3   1091302   58.3
## Vcells 1201153  9.2  518223748 3953.8 538102126 4105.4
```

```
#### try ff object

#load the package
suppressMessages(library(ff))
suppressMessages(library(ffbase))

# check memory use by R
memory.size()
```

```
## [1] 66.49
```

```
# create very large integer vector as ff object
x <- ff(length = 2^30, vmode = "integer")

# check used memory again - almost no increase in memory usage as the object has all zeros
memory.size()
```

```
## [1] 67.75
```

```
# calculate mean of ff vector - it should be zero, it takes time to calculate
mean.ff(x)
```

```
## [1] 0
```

```
# check memory again - it is high as we loaded data from hard drive
# however it is a way lower than the full data set
memory.size()
```

```
## [1] 381.96
```

Parallel processing of large data sets

As you have seen before, parallel processing requires a lot of memory. At the same time, package `ff` allows to save some memory usage by storing the main body of data on the hard drive. That data might be shared between workers in the cluster. All workers can have access to the same data set without the need to load data into each worker individually.

Below is a practical example with a real data set. There are 12 variables and 1.5 million observations. This is a small data set, it does not need `ff`. Still we try to use `ff` for loading and processing data.

First run is for "normal" parallel processing without `ff` package – with the full data set loaded in memory

```
# load data set - not too large, 122.6 Mb
df <- read.csv(file="yelp_reviews.csv", header=TRUE)

library(foreach)
library(doSNOW)

# prepare the cluster
cl <- makeCluster(7, type="SOCK")
registerDoSNOW(cl)

# run 2000 random samples, build linear model for each and check time
system.time(
  res <- foreach(i=seq(1,2000), .combine = rbind, .packages = c("ff")) %dopar% {
    temp_df <- df[sample(nrow(df), 100000), ]
    temp_fit <- lm(stars ~ pos_words + neg_words, data = temp_df)
    temp_fit$coefficients
  }
)

stopCluster(cl)

# calculate expectation of coefficients
apply(res, 2, mean)
```

Here is the result:

```
   user  system elapsed
   5.41    2.11   80.25

(Intercept)   pos_words   neg_words
 3.67981123  0.08763288 -0.21834192
```

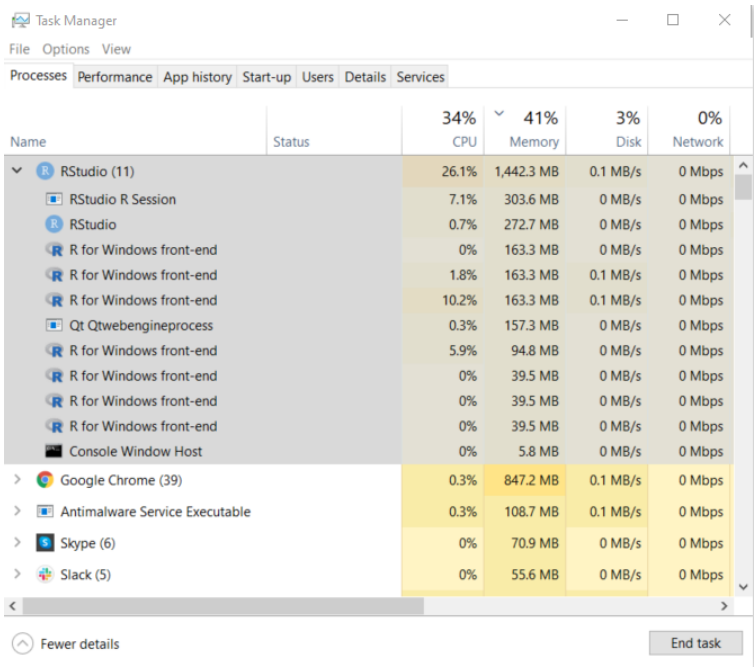


Figure 2: "Loading data in workers"

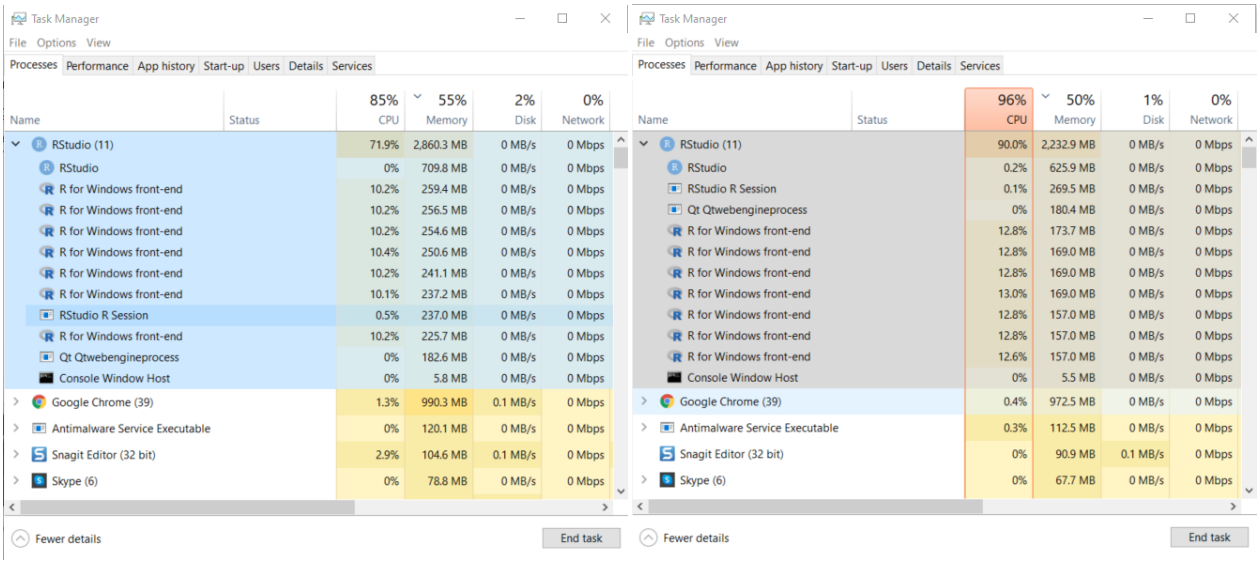
Now, the same code but with loading data set in ff-dataframe. There are no any other changes to the code.

```
library(ff)

# load data set into ff object - size in memory is 36.2 Mb
df <- read.csv.ffdf(file="yelp_reviews.csv", header=TRUE)

library(foreach)
library(doSNOW)

# prepare the cluster
cl <- makeCluster(7, type="SOCK")
registerDoSNOW(cl)

# run 2000 random samples and build linear model for each and check time
system.time(
  res <- foreach(i=seq(1,2000), .combine = rbind, .packages = c("ff")) %dopar% {
    temp_df <- df[sample(nrow(df), 100000), ]
    temp_fit <- lm(stars ~ pos_words + neg_words, data = temp_df)
    temp_fit$coefficients
  }
)

stopCluster(cl)

# calculate expectation of coefficients
apply(res, 2, mean)
```

The result:

```
   user  system elapsed
   2.28    0.37  162.08

(Intercept)   pos_words   neg_words
 3.67927004  0.08765284 -0.21827166
```

The same code working with ff-dataframe took twice longer to run, which is not unexpected as data was loaded from the hard drive. However, the second attempt used less memory.



Data analysis in the cloud

There is one more way to deal with large data – to rent a server in the cloud, have RStudio server installed and running there. User experience is the same as working on your own computer. However, you can rent a server with up to $24Tb$ of RAM and this is relatively cheap. There are many different providers. My personal choice is Amazon AWS – https://aws.amazon.com/console/ .

It is possible to rent a "naked" Linux server and install everything you need but it would take some time. It is much easier to use pre-configured image with R, RStudio and some other staff for data analysis. Here are instructions and links for such images for Amzon EC2 – https://www.louisaslett.com/RStudio_AMI/ .

You just need to select location for the server as different images available in different locations. Click on the link and you'll be transferred to Amazon EC2 and can select desired server configuration. After the server starts, you can start working on your project.

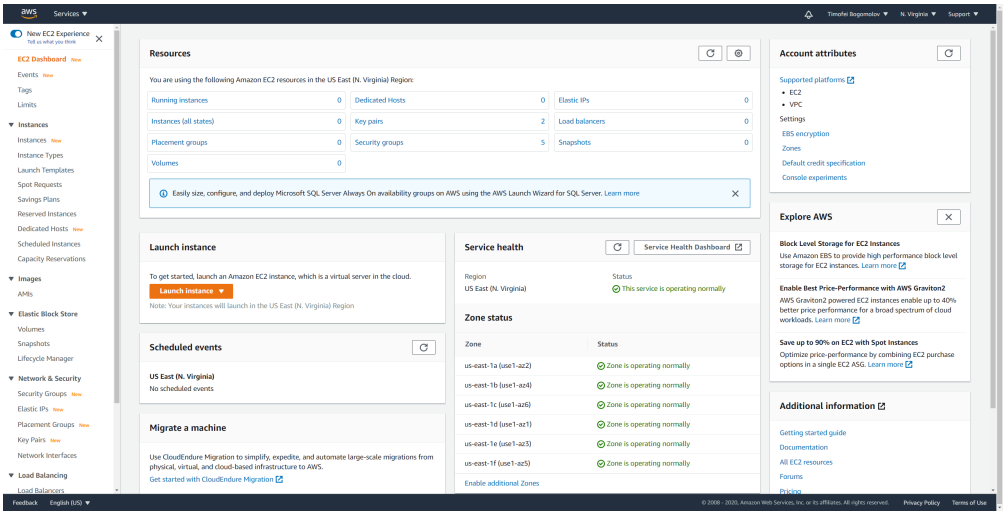


Figure 3: "Amazon EC2 dashboard"