Parallel computing with **R** (session 1)

M2 Statistics and Econometrics

Thibault Laurent

Toulouse School of Economics, CNRS

Last update: 2023-09-28

Table of contents

1. Principle of parallel computing

2. How coding a program in parallel?

3. Balancing the distribution of tasks

4. Reproduccing the results: choice of the seed

Before starting

Packages and software versions

Install the following packages:

```r
install.packages(c("microbenchmark", "rpart", "snow", "snowFT", "ti
                 dependencies = TRUE)
```

And load them:

```r
require("parallel")
require("snow")
require("snowFT")
require("VGAM")
```

This document has been compiled under this version of **R**

```r
R.Version()$version.string
```

```
## [1] "R version 4.3.1 (2023-06-16)"
```

This session was inspired by this tutorial.

1. Principle of parallel computing

Principle of parallel computing

Parallel computing is particularly adapted for this kind of algorithm:

1. Create a function `myfunc()` that is sampling from the original data with a given seed and returns the value of an estimator:

```
myfunc <- function(data, i){
 set.seed(i)
 data_boot <- data[sampling(1:n, replace = T), ]
   ...
 return(estimator)
}
```

2. Repeat N times `myfunc()` with a different seed for each simulation:

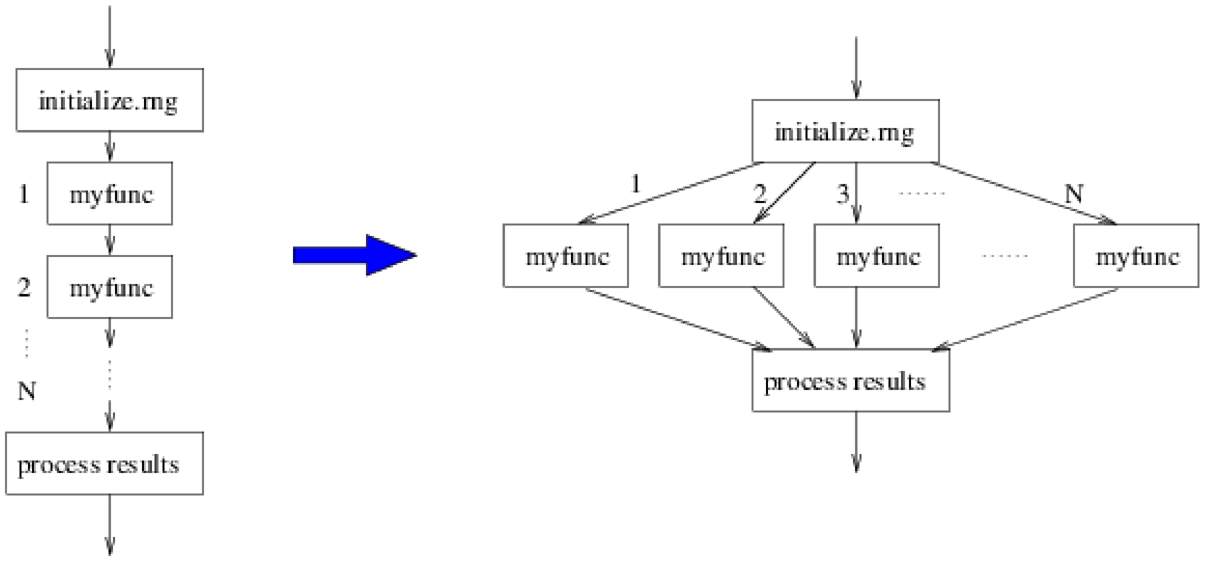
```
for (iteration in 1:N) {
  result[iteration] <- myfunc(...)
}
```

3. Summarize the N results obtained:

```
process(result,...)
```

Sequential to parallel

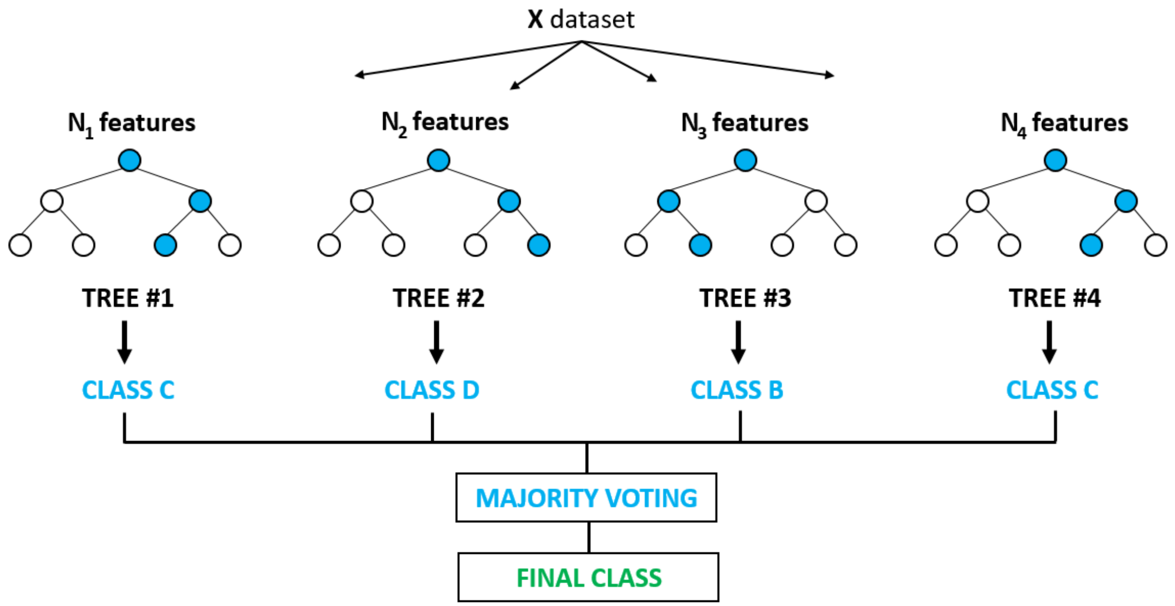
- Sequential: execute iteration 1, then iteration 2, etc. sequentially
- Parallel: execute the N iterations at the same time in parallel



Remark: in practice, the number of availabale cores is lower than N. In that case, we split the iterations into groups of size equal to the number of cores available and execute sequentially the sub-iterations inside the cores

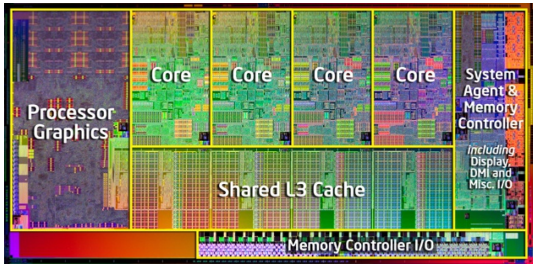
Example of algorithm: Random forest

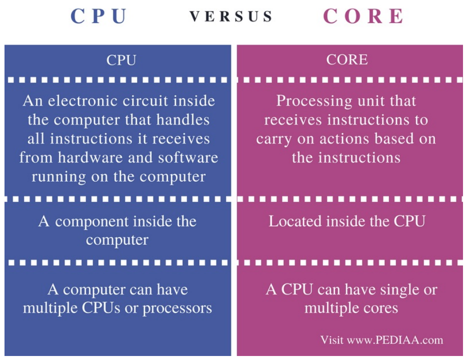
Repeat N regression or classification trees with respect to the N samples:



How many cores are available on my machine?

What is difference between CPU and cores ?





CPU VERSUS CORE

CPU	CORE
An electronic circuit inside the computer that handles all instructions it receives from hardware and software running on the computer	Processing unit that receives instructions to carry on actions based on the instructions
A component inside the computer	Located inside the CPU
A computer can have multiple CPUs or processors	A CPU can have single or multiple cores

Visit www.PEDIAA.com

Function *detectCores()* from **parallel** package permits to detect the number of cores and threads

```
library("parallel")
detectCores(logical = FALSE) # number of cores
detectCores() # number of logical cores or threads
```

👩‍🎓 Training

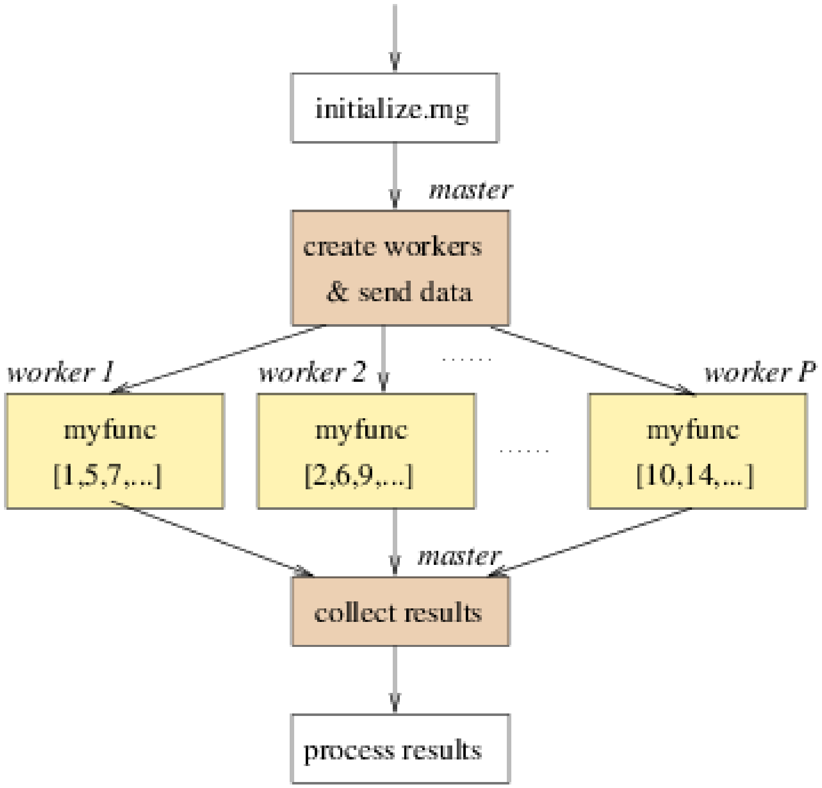
Exercise 1

- How many cores do you have on your machine ?
- How many threads do you have on your machine ?
- Which are the methods that you have seen in the Big Data course that could be used in parallel computing ?

What is a master program?

- **Definition:** The master program specifies the division of tasks and summarizes the parallelized results.
- **Example:** in the previous algorithm, we have N process to replicate. Let suppose that $N = 100$ and the number of cores available is equal to 4. The master program must indicate how to divide tasks across the cores. For example:
    - The core 1 will execute iterations 1, 5, 9, …, 93, 97
    - The core 2 will execute iterations 2, 6, 10, …, 94, 98
    - The core 3 will execute iterations 3, 7, 11, …, 95, 99
    - The core 4 will execute iterations 4, 8, 12, …, 96, 100
- Once the tasks computed, the master program has to summarize the results obtained.

Summary of a master program



Many packages permit to do parallel computing with **R**. We will focus on **parallel** and **snow**.

What does the function *set.seed()* do?

When sampling, the need is the possible desire for reproducible results. *set.seed()* function in **R** is used to reproduce results i.e. it produces the same sample again and again. It takes as input argument an **integer** which is then used in the algorithm to generate randomness. The function is followed by a simulation function like *sample()*, *rnorm()*, *runif()*, *rbinom()*, etc. For example, if we draw 5 numbers among 49 and then simulate a white noise:

```r
set.seed(200907)
loto <- sample(1:49, 5)
white_noise <- rnorm(5)
```

If you repeat again the same syntax (starting by *set.seed()* with the same **integer**) followed by the same simulation functions, the two simulated vectors **loto** and **white_noise** will be identical. It can be useful to reproduce results which use simulations.

More informations in this video.

Be carreful with *set.seed()*!

In the algorithm of bootstrap, each simulation should be different from another one. Hence, do not use *set.seed()* with a fix number inside the loop **for**. In the following example, we compute the "bootstrap" mean of the variable "Sepal.Length", but the seed is fixed to the same number. Hence, the sample drawn is the same for any simulation.

```
B <- 10
res_mean <- numeric(B)
for (b in 1:B) {
  set.seed(123)
  samp <- sample(1:nrow(iris), replace = T)
  res_mean[b] <- mean(iris[samp, "Sepal.Length"])
}
res_mean
```

```
##  [1] 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774 5.774
```

Solution:

- a) execute *set.seed()* before the loop
- b) possibility to replace 123 by **b** inside the **for** loop

👩‍🎓 Training

Exercise 2.1

- Compare the two solutions a) and b) proposed at previous page

Exercise 2.2

The aim is to use a bootstrap algorithm and compute the "bootstrap" mean of **my_vec**. We fix the number of replication to $B = 1000$.

```
set.seed(200907)
my_vec <- rnorm(100000)
```

- Implement the program in a sequential way; represent the histogram of the "bootstrap" values, the "bootstrap" mean and the true value.
- Describe the algorithm for parallel computing with respect to the number of cores available on your machine.

2. How coding a program in parallel?

a. Example

Bagging algorithm

We split **iris** data into two data sets: a training data set called **train_sets** of size 120 and a test data set called **test_sets** with 30 observations (10 observations for each level of **Species**).

```
set.seed(5656)
id_pred <- c(sample(which(iris$Species == "setosa"), 10, replace =
             sample(which(iris$Species == "versicolor"), 10, replac
             sample(which(iris$Species == "virginica"), 10, replace
test_sets <- iris[id_pred, ]
train_sets <- iris[-id_pred, ]
```

Goal: we want to predict **Species** variable on the test data set by using a bagging algorithm.

We remind that the bagging algorithm consists in:

1. bootstrapping B training samples (here $B = 100$),
2. doing a classification or regression tree on each "bootstrapped" sample
3. predicting B times on the test samples.
4. summarizing on the B results to get the final predictions

Classification tree on a Bootstrap sample

We consider the function *my_tree()* which:

1. bootstraps the **train** data set,
2. computes a classification tree (argument **formula** gives the model to be estimated)
3. returns the predictions on the **test** data set.

The first argument of *my_tree()* is the value of the seed to be used before sampling the data.

```
my_tree <- function(b, train, test, formula) {
  set.seed(b)
  # bootstrap the observations
  train_sets_bootstrap <- train[sample(1:nrow(train), replace = T),
  # classification tree on the bootstrap sample
  res_rf <- rpart::rpart(formula, data = train_sets_bootstrap)
  # prediction
  res <- predict(res_rf, newdata = test, type = "class")
  return(as.character(res))
}
# Example:
my_tree(123, train = train_sets, test = test_sets, formula = Specie
```

Starting from the non // version: **for** loop (1)

In the non parallel case, we can use either a **for** loop solution either the function *sapply()*. With the **for** loop solution, we use the argument **b** in the **for** loop as the first argument of the function:

```r
B <- 100
my_pred <- matrix("0", B, nrow(test_sets))
for (b in 1:B) {
 my_pred[b, ] <- my_tree(b, train = train_sets, test = test_sets, f
}
```

Then, for each observation in the test sample, we summarize the predictions obtained. For doing this, we compute a function which returns the mode of a vector of character.

```r
my_mode <- function(x) {
      ux <- unique(x)
      ux[which.max(tabulate(match(x, ux)))]
}
```

Starting from the non // version: **for** loop (2)

For example, for the first observation in the test data set :

```
my_mode(my_pred[1, ])
```

```
## [1] "versicolor"
```

We can do it for each observation by using function *apply()*:

```
my_final_pred <- apply(my_pred, 2, my_mode)
```

Finally, we can compare with the observed data:

```
table(test_sets$Species, my_final_pred)
```

```
##             my_final_pred
##              setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0          9         1
##   virginica       0          1         9
```

Starting from the non // version: *sapply()*

In the 1st argument, we give a vector of size B corresponding to the values of **b** to be evaluated by the function *my_tree()*. Here, we choose integer values from 1 to B.

```
B <- 100
res_non_par <- sapply(1:B, FUN = my_tree,
          train = train_sets, test = test_sets, formula = Species ~
```

Remark: the optional argument **train**, **test** and **formula** appear in the function *sapply()* after the argument **FUN**.

Finally, we summarize the bagging predictions:

```
# final prediction
my_final_pred <- apply(res_non_par, 1, my_mode)
table(test_sets$Species, my_final_pred)
```

Comparing computational time with R

The function *system.time()* can be used, but it is noisy due to the CPU which is always running different applications. Repeating the same instruction will return two different computational times:

```
system.time(sapply(1:B, FUN = my_tree, train = train_sets, test = t
system.time(sapply(1:B, FUN = my_tree, train = train_sets, test = t
```

The principle of function *microbenchmark()* is to repeat the same instructions several times and doing statistics on the computational time results:

```
library(microbenchmark)
mbm_1 <- microbenchmark(
  `for loop` = {for (b in 1:B) {
    my_pred[b, ] <- my_tree(b, train = train_sets, test = test_sets
  }},
  `sapply` = sapply(1:B, FUN = my_tree, train = train_sets, test =
    times = 10L)
```

Remark: **for** loop and *sapply()* give similar results in computational time

Summary statistics on the computational time

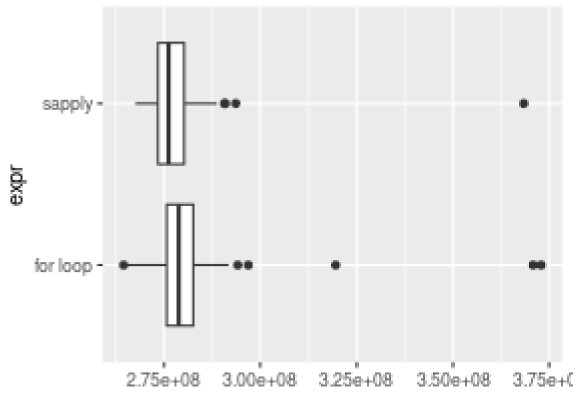
When you print the previous object, you get directly some statistics:

```
print(mbm_1)
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq      max neval
 for loop 264.4870 275.6725 281.2750 278.8226 282.6041 372.9308   100
   sapply 267.5311 273.3300 277.9667 276.1686 280.2284 368.4190   100
```

To plot a parallel boxplot, one can use the **ggplot2** syntax :

```
library(tidyverse)
ggplot(mbm_1) +
  aes(x = time, y = expr) +
  geom_boxplot()
```



Parallel case

With **parallel** package:

- we first allocate the number of cores to be used with *makeCluster()*.
- Then, we use function *clusterApply()*. Excepted the 1st argument which indicates which cluster to be used, the syntax is exactly the same than *sapply()*.
- To finish, it is necessary to free the cores with *stopCluster()*:

```
P <- 4
cl <- makeCluster(P)
mbm_2 <- microbenchmark::microbenchmark(
  `parallel computing` = {res_par <- parallel::clusterApply(cl, 1:B
      train = train_sets, test = test_sets, formula = Species ~ .)}
  times = 10L
)
stopCluster(cl)
```

Remark: the objects **train_sets**, **test_sets**, and **formula** are send in the cores job by job.

Summarizing the results

The results is a **list** object. Before computing the predictions, we need first to transform the list in a more convenient format. This can be done with a **for** loop.

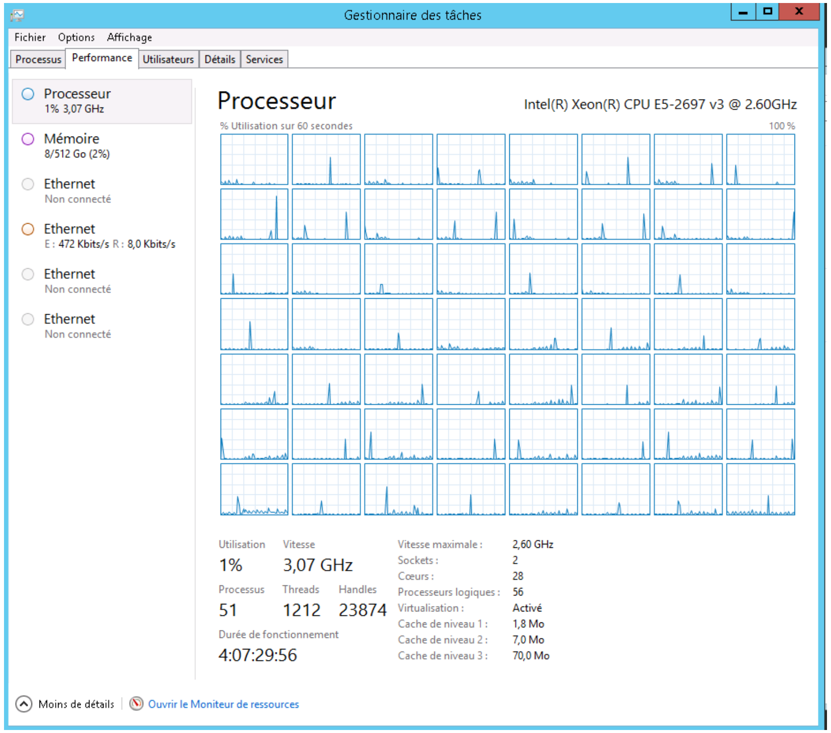
```r
# Loop 1: aggregate the results on the B simulation
my_matrix <- matrix("0", 100, 30)
for (b in 1:100)
     my_matrix[b, ] <- as.character(res_par[[b]])
```

Once it has been done, we can summarize the results of the B replications to get the predictions.

```r
# Loop 2: predict by the most observed levels
resultats <- data.frame(id = id_pred)
resultats$bagging <- apply(my_matrix, 2, function(x) names(which.ma
```

What's happening during the process?

User can check than the cores are running. On **Windows**, click on "Gestionnaire des tâches/performance". On **Linux**, use command **top** in the terminal.


Gestionnaire des tâches
Fichier Options Affichage
Processus Performance Utilisateurs Détails Services
Moins de détails
Ouvrir le Moniteur de ressources

Processeur 1% 3,07 GHz
Mémoire 8/512 Go (2%)
Ethernet Non connecté
Ethernet E : 472 Kbits/s R : 8,0 Kbits/s
Ethernet Non connecté
Ethernet Non connecté

Processeur Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
% Utilisation sur 60 secondes 100 %

Utilisation 1%
Vitesse 3,07 GHz
Processus 51
Threads 1212
Handles 23874
Durée de fonctionnement 4:07:29:56

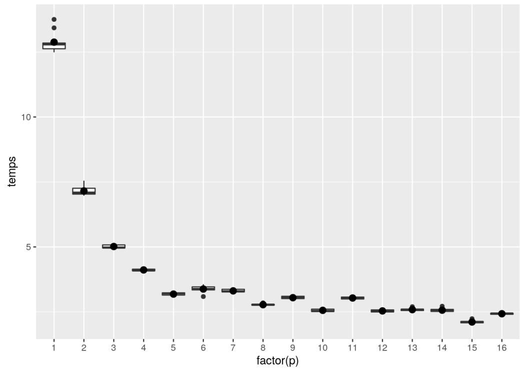
Vitesse maximale :	2,60 GHz
Sockets :	2
Cœurs :	28
Processeurs logiques :	56
Virtualisation :	Activé
Cache de niveau 1 :	1,8 Mo
Cache de niveau 2 :	7,0 Mo
Cache de niveau 3 :	70,0 Mo

Recommendations

- Launch your parallel code once it has been tested and was successful with a single core.
- If you get an error message, do not forget to use *stopCluster()* to free cores.
- If you kill you master session, it could happen that the allocated cores are still running. Moreover, *stopCluster()* is probably not sufficient. Click on "Gestionnaire des tâches/processus" and right-click on each of the open **R** processes and kill them.
- If you need to load big objects, these objects will be duplicated in each core. In other terms, the total RAM needed corresponds to $P \times RAM_{\text{needed for 1 core}}$, where P is the number of cores. If the total amount of RAM is 100% of the capacity, parallel computing is probably not optimal.

How many cores should I use?

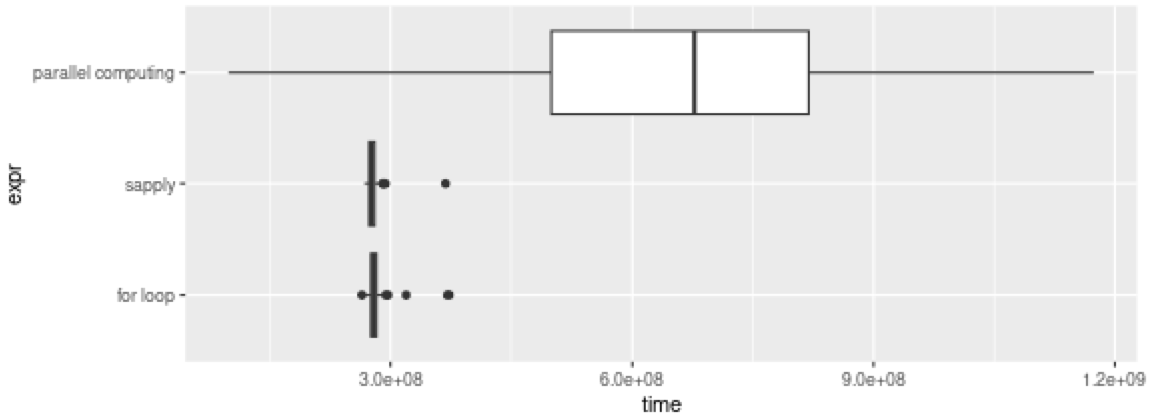
- The time saving is not linear depending on the number of cores. Indeed, there are some flows between the master program and the cores that are costly in computation time (typically, the transfer of data).
- The time saving is decreasing but there could be an asymptote. In that case, it is not necessary to use all the cores.



- It could happen that the computational time with several cores is higher than with only one core. In that case, it probably means that the program is poorly implemented

Computational time

With our example, parallel computing is not efficient… This is because there are many flows of data between the master program and the cores.



We are going to see how we could improve it?

2. How coding a program in parallel?

b. Improving my parallel code

How to evaluate expressions across the cores?

Doing parallel computing on P cores is equivalent to opening P new **R** session. Hence, there are no objects known neither packages loaded in the cores. For example, let consider a variation of *my_tree()*

```
my_tree_2 <- function(b) {
  set.seed(b)
  train_sets_bootstrap <- train_sets[sample(1:nrow(train_sets), rep
  res_rf <- rpart(formula, data = train_sets_bootstrap)
  res <- predict(res_rf, newdata = test_sets, type = "class")
  return(res)
}
```

If we send this function like this in a parallel way, there are two issues:

- package **rpart** must be loaded for using *rpart()* function
- the three objects **train_sets**, **test_sets** and **formula** are unknown in the cores (even if they have beed defined in the main session).

Evaluate objects and packages in each core

It is possible to evaluate objects and packages in each core before executing *my_tree_2()*. For doing this, we use function *clusterEvalQ()* which contains some expressions to be evaluated in each core.

```
cl <- makeCluster(P)
clusterEvalQ(cl, {
  library("rpart")
  set.seed(5656)
  id_pred <- c(sample(1:50, 10, replace = F),
               sample(51:100, 10, replace = F),
               sample(101:150, 10, replace = F))
  test_sets <- iris[id_pred, ]
  train_sets <- iris[-id_pred, ]
  formula <- Species ~ .
})
mbm_3 <- microbenchmark::microbenchmark(
  `parallel computing 2` = {res_par <- clusterApply(cl, 1:100, fun
  times = 100L
)
stopCluster(cl)
```

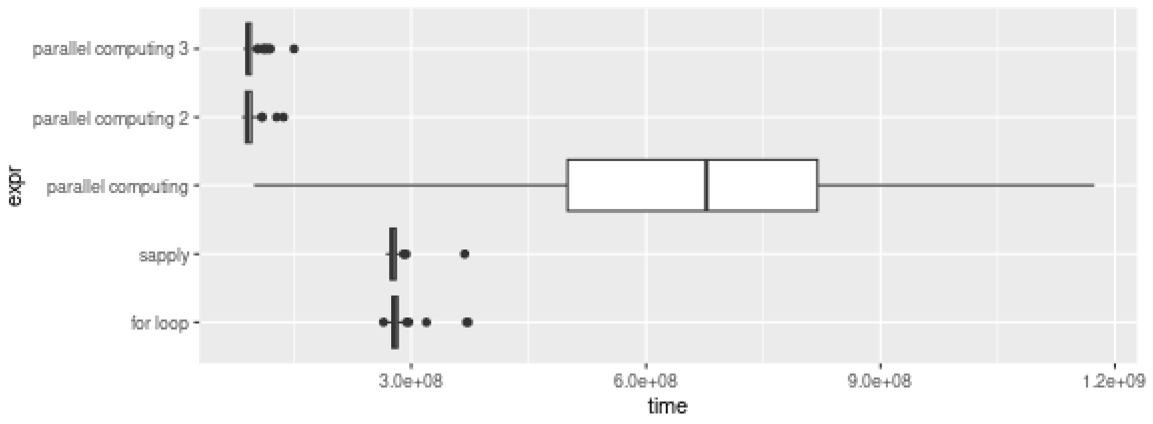
Load objects and packages in each core

In the previous example, we have send expressions to be evaluated in each core. It is also possible to export directly objects from the main session to the different cores thanks to the function *clusterExport()*.

```
cl <- makeCluster(P)
clusterExport(cl, c("test_sets", "train_sets"))
clusterEvalQ(cl, {
  library("rpart")
  formula <- Species ~ .
  })
mbm_4 <- microbenchmark::microbenchmark(
  `parallel computing 3` = {clusterApply(cl, 1:100, fun = my_tree_2
  times = 100L
)
stopCluster(cl)
```

Comparaison of the computational times

In this example, it is interesting to use parallel computing, if and only if we load the data in the cores (thanks to *clusterExport()* and *clusterEvalQ*). Otherwise, transferring flows of data job by job across the cores, is more costly than using only one core.



Other packages for parallel computing

- **snowFT**: takes into account the issue of random seed (we present an example at the end of this session)
- **foreach** and **doParallel** (see vignette for more information): interesting because based on **for** loop syntax.

```
require("doParallel")
registerDoParallel(cores = P) # allocate the cores
getDoParWorkers() # print the number of cores
system.time(
  res_par_foreach <- foreach(i = r_values)
   %dopar% myfun(i, mean = 5, sd = 10)
  )
# free the cores
registerDoParallel(cores = 1)

# print the result
unlist(res_par_foreach)
```

- **doMPI**: for a MPI achitecture

👩‍🎓 Training

Exercise 3.1

The aim of this exercise is to implement the Random Forest (RF) algorithm by using parallel computing on the **iris** data seen previously.

Remark: RF is similar to Bagging. The only difference is that at each replication b, the model on the train set, select randomly a number m of explanatory variables. Here we choose $m = 2$.

Compare the predictions between bagging and RF

3. Balancing the distribution of tasks

Problematic

When we split the tasks between cores, it could happen that some tasks take more time than others. For example, let consider the function *rnmean()* which returns the average mean of a Gaussian sample of size r where **r** is the input argument.

```
rnmean <- function(r, mean = 0, sd = 1) {
        mean(rnorm(r, mean = mean, sd = sd))
}
```

If we apply *rnmean()* on heterogeneous values of r, there is an imbalance between tasks with respect to the values of r. We create here heterogeneous values for **r**:

```
N <- 80
set.seed(50)
(r.seq <- sample(ceiling(exp(seq(10, 14, length = N))), N))
```

```
##  [1]   36547  291350  215019  622671   31396   47075   52092   60637  508
## [10]  356757  136324   86430   34742  375286  483404  105834   40441  117
## [19]   74250  802059  226187  100609  655010  184718   29846  843715  194
## [28]  887535   25640   67099   28373  150852  982120  263291  724815  158
## [37]  933630 1143229   78107   49520  143404   22027  322399  306482   44
```

Applications on 4 cores

If we parallelize the tasks between 4 cores with respect to the vector **r.seq** :

```
cl <- makeCluster(4)
my_res <- clusterApply(cl, r.seq, fun = rnmean)
stopCluster(cl)
```

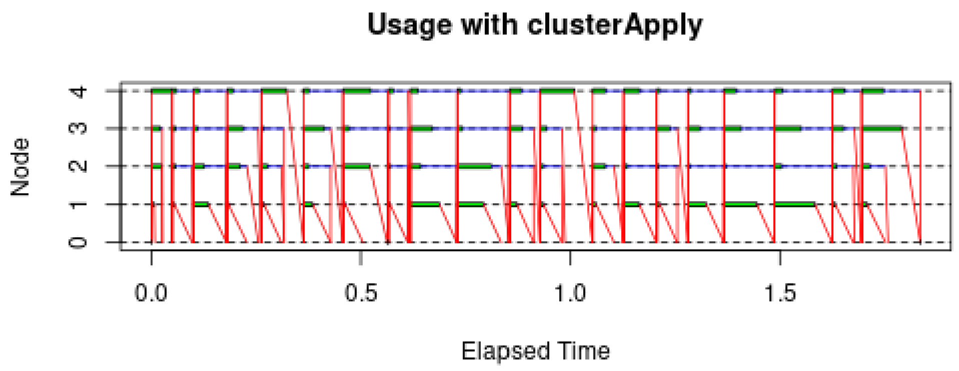
The division of tasks will be done like this:

- core 1 will make the computations for the following values of **r.seq**: 1021451 (1st position), 867586 (5th position), 88235 (9th position), etc.
- core 2 will make the computations for the following values of **r.seq**: 49828 (2nd position), 25933 (6th position), 183956 (10th position), etc.
- core 3 will make the computations for the following values of **r.seq**: 254990 (3rd position), 39005 (7th position), 112719 (11th position), etc.
- core 4 will make the computations for the following values of **r.seq**: 1202605 (4th position), 74944 (8th position), 576832 (12th position), etc.

Computational time per core

We can plot the computational time job by job (green line) and core by core. It can be done with function *snow.time()* from package **snow**. Red line corresponds to transfer data and blue line to a time break. Here, we can see that jobs must start at the same time in the cores.

```
cl <- makeCluster(P)
ctime <- snow::snow.time(snow::clusterApply(cl, r.seq, fun = rnmean
plot(ctime, title = "Usage with clusterApply")
```

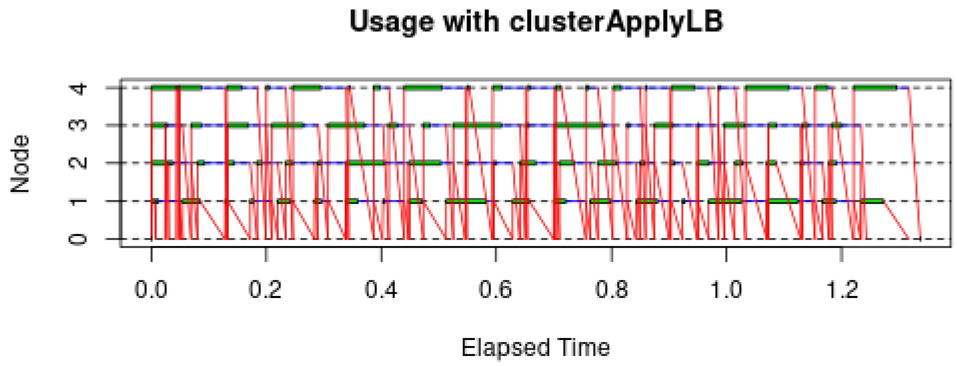


```
stopCluster(cl)
```

Use *clusterApplyLB()*

The use of *clusterApplyLB()* instead of *clusterApply()* permits to resolve this issue. The execution of the parallelized functions start independently between cores.

```
cl <- makeCluster(P)
ctimeLB <- snow.time(clusterApplyLB(cl, r.seq, fun = rnmean))
plot(ctimeLB, title = "Usage with clusterApplyLB")
```



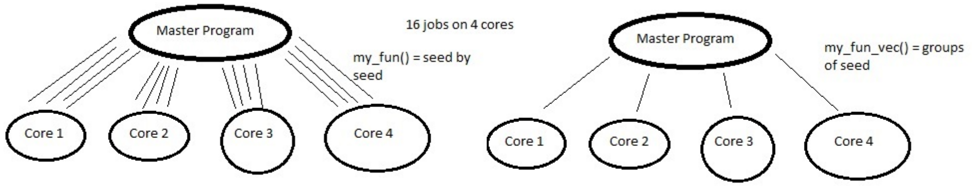
```
stopCluster(cl)
```

Number of jobs > number of cores

In our previous examples, one job corresponds to the execution of a particular function *my_fun()*. Let suppose that we have 4 cores and we want to execute 16 times *my_fun*. When we use the following instruction, we actually do 16 transfer of informations (figure on the left).

```
cl <- makeCluster(4)
clusterApply(cl, 1:16, fun = my_fun)
stopCluster(cl)
```

The idea is to reducce the number of flows by vectorizing the function *my_fun()*



```
cl <- makeCluster(4)
clusterApply(cl, list(`job_1` = 1:4, `job_2` = 5:8, `job_3` = 9:12,
stopCluster(cl)
```

Applications

In the previous example, there are 40 jobs which means 40 flows of information between master core and other cores. The program could be implemented differently and call only 4 jobs, each job containing the results of 10 simulations.

For doing that, the function *rnmean()* must be adapted for being able to treat several values of **r**, by using function *sapply()* for instance. The 1st argument is now a vector :

```
rnmean_vect <- function(vec_r, mean = 0, sd = 1) {
        sapply(vec_r,
         function(x) mean(rnorm(x, mean = mean, sd = sd)))
}
```

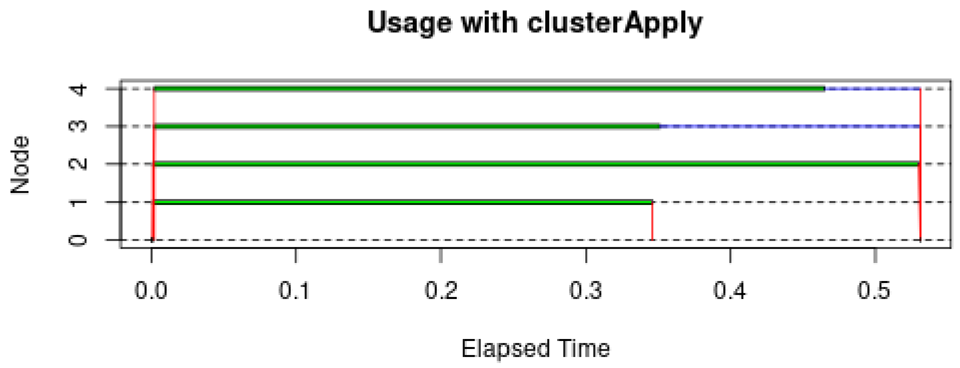
Then the master program must also be adapted; indeed, it may send in each core a vector of values to be evaluated. For doing this, we prepare a list of vectors:

```
r.seq_list <- list(r.seq[1:20], r.seq[21:40], r.seq[41:60], r.seq[6
```

Best solution

Finally, we call parallel computing. It is not necessary to use *clusterApplyLB()* instead of *clusterApply()* because the number of jobs is equal to the number of cores. Finally, we observe the best computational time.

```
library("snow")
cl <- makeCluster(P)
ctime <- snow.time(clusterApply(cl, r.seq_list, fun = rnmean_vect))
plot(ctime, title = "Usage with clusterApply")
```



```
stopCluster(cl)
```

When // computing is not a good solution?

Let consider the following "big" object:

```
n <- 10000000
big_file <- data.frame(chiffre = 1:n, lettre = paste0("caract", 1:n
      date = sample(seq.Date(as.Date("2017-10-01"), by = "day", len
object.size(big_file)
```

The aim is to compute a new binary variable with 1 if **chiffre** is an even and 0 otherwise.

Solution 1: create a vector of boolean with **%%** and *ifelse()*

```
mbm_1 <- microbenchmark::microbenchmark({
  big_file$new <- ifelse(big_file$chiffre %% 2 == 0, 1, 0)}, times
```

Solution 2: create a vector of boolean with **%%** and *as.numeric()*

```
mbm_2 <- microbenchmark::microbenchmark({
  big_file$new <- as.numeric(big_file$chiffre %% 2 == 0)}, times =
```

Bad example of // computing

Solution 3: use // computing with *foreach()*. We define the function to be parallelize

```
compare <- function(x)
  x %% 2 == 0
```

Then we *foreach()* (only on the 1000 first thousand values because it is too much costly otherwise)

```
require("doParallel")
P <- 4
registerDoParallel(cores = P)
system.time(
  res <- foreach(i = 1:1000) %dopar%
    compare(big_file$chiffre[i])
)
```

Remark: here 1 job corresponds to the evaluation of 1 value of the vector. As we have seen previously, it is not efficient because there are too much flows of information.

Good example of // computing

Instead of sending 1 job per core, we evaluate several jobs. In the following program, we evaluate in the 1st core the values from 1 to 2500000, in the 2nd core the values from 2500001 to 5000000, etc.:

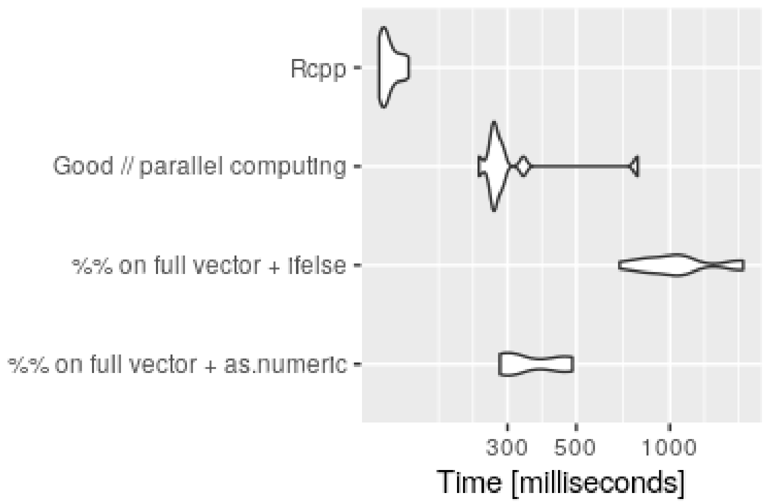
```
require("doParallel")
registerDoParallel(cores = P)
mbm_3 <- microbenchmark::microbenchmark({
res <- foreach(i = 1:4) %dopar%
  compare(big_file[(1 + 2500000 * (i - 1)):(2500000 * i), "chiffre"
}, times = 10L
)
```

Solution 4: Use C++

```cpp
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
IntegerVector compare_cpp(IntegerVector x) {
    int n = x.size();
    IntegerVector res(n);
    for(int i = 0; i < n; i++) {
      if(x(i) % 2 == 0) {
      res(i) = 1;
      } else {
      res(i) = 0;
      }
    }
    return res;
}
```

```r
require("Rcpp")
mbm_4 <- microbenchmark::microbenchmark(
  big_file$new <- compare_cpp(big_file$chiffre),
  times = 10L)
```

Comparison of computational time



In this example, using **C++** is the best solution which is often the case when a program uses a huge number of **loop**.

Parallel computing does not really improve the computational time compared to the vectorial solution, because the flows of information between master program and cores are important.

👩‍🎓 Training

Exercise 4.1

The aim of this exercise is to improve the Random Forest (RF) algorithm programmed previously by limiting the number of jobs. Here, we impose that the number of jobs equals the number of cores. It requires to vectorize the function `my_tree()`. Compare the computational time with the previous version of the algorithm.

4. Reproduccing the results: choice of the seed

How to define the seed

The function *set.seed()* must be called inside the function to be paralleled. In parallel computing, we have to choose the values of the seed such that each job has an unique value. In the following function, we define two arguments which can be modified in a job: **x** the seed and **r** the size of the vector to be simulated

```r
rnmean <- function(x, r, mean = 0, sd = 1) {
  set.seed(x)
  return(mean(rnorm(r, mean = mean, sd = sd)))
}
```

That is why we use hereafter *clusterMap()* (like *mapply()*) which permits to parallel a function with two arguments:

```r
r_values <- rep(c(10, 1000, 100000, 10000000), each = 10)
cl <- makeCluster(P)
# Good example:
mbm_1 <- microbenchmark::microbenchmark(
  clusterMap(cl, fun = rnmean, x = 1:100, r = r_values, mean = 0, s
# Bad example:
clusterMap(cl, fun = rnmean, x= rep(1:25, times = 4), r = r_values,
stopCluster(cl)
```

Package snowFT (1)

Function *performParallel()* from package **snowFT** permits to determine the different seeds in the replicates with respect to the seed given in the master program. In this case, it is not necessary to include *set.seed()* inside the function to be parallelize. However the unique seed is given in argument **seed** of function *performParallel()*.

```
rnmean <- function(r, mean = 0, sd = 1) {
        return(mean(rnorm(r, mean = mean, sd = sd)))
}

library("snowFT")
seed <- 1
mbm_2 <- microbenchmark::microbenchmark(
  performParallel(P, r.seq, fun = rnmean, seed = seed), times = 10L
)
```

This package seems to give similar results in a computational point of view

More informations about **snowFT**

When using package **snowFT**, the objects to export and packages to load in the cores are given in the arguments **initexpr** and **export**. For example:

```r
myfun_pareto <- function(r) {
  return(mean(rpareto(r, scale = scale, shape = shape)))
}

seed <- 1
scale <- 1
shape <- 10
r_values <- rep(c(10, 1000, 100000, 10000000), each = 10)

res <- performParallel(P, r.seq, fun = myfun_pareto,
                       seed = seed,
                       initexpr = require("VGAM"),
                       export = c("scale", "shape"))
```