Projet Final

Julien Lanthier

Akankhya Mohapatra

Zakaria Rayadh

04/25/2021

```
# General Libraries
library(ggplot2)
library(kableExtra)

# for foreach %dopar%
library(foreach)
library(doParallel)
# for mclapply
library(parallel)

#Random Forest Libraries
library(randomForest)
library(randomForestSRC)
library(ranger)

library(torch)
# device_big_ops = "cpu"
device_big_ops = "cuda"
```

Introduction

During the last decade, we have seen a massive inflow of data across the globe and across servers. Whether the data comes from people using their cellphone or from daily companies' operations, multiple organizations are trying to get access to this new important resource in order to help them make better decisions.

New state of the art technologies often run complex artificial intelligence models using voluminous amount of data to obtain insights that can lead to better costumer retention and marketing oportunities. Behind these complex algorithms are simple matrix operations that computers can handle easily nowadays. Before talking about GPUs, TPUs supercomputers and clusters that allows to optimize these operations, let's start with the moment when modern commercialized computer where first introduced.

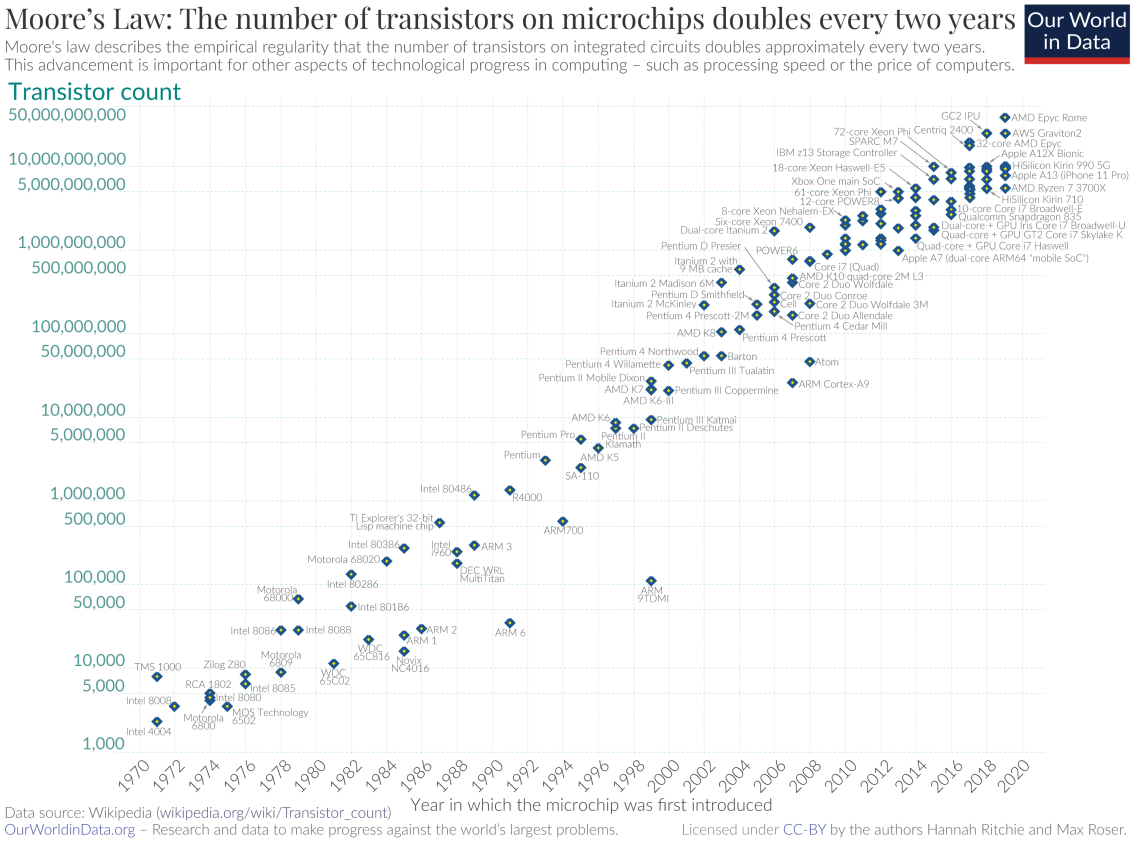
Computers

The initial idea of a computer was already brought up in 1937 by Alan Turing *Turing (1937)* who introduced the Turing machines concept. The main idea of those machine were that they could store information on a *Tape* that could be later accessed to retrieve information.

Nowadays, a computer is defined as a machine that is used to store and process data according to a given set of instructions (program). Computer hardware is composed of several parts including motherboards, memory (RAM) and most importantly the Central Processing Unit (CPU) which will be a very important topic in this report.

The CPU resides in most of the devices we use today and it is responsible for the processing and the execution of instructions. Those instructions includes the one that we directly give to the computers and also all the instructions that allow the environment to stay functionnal. The CPU is the equivalent of the human brain for computers. Throughout the

years, billions of dollars have been invested in research and development to build new CPU capable of increasing the speed of our computers. In 1965, Gordon Moore made a prediction that computing power would increase at an exponential pace *Moore (2006)*. Until this day, his affirmation is still relevant. In fact, the number of transistors which dictates the speed of the CPU is doubling every two 2 years *Roser and Ritchie (2013)*.



Moores Law Illustration

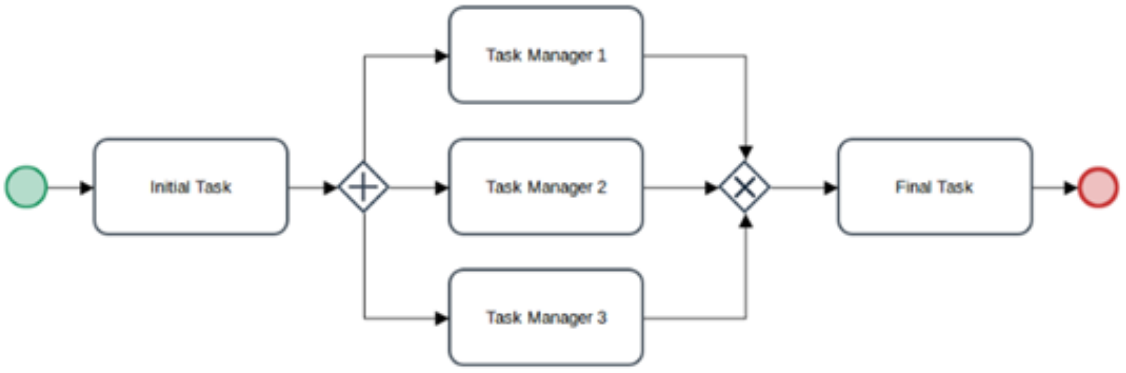
The problem here is that the growth rate of the data volume is bigger than the growth rate of computing power, this causes problem to many companies leveraging data in their daily operations. To counter this constraint, they use a long existing method called parallelization that allows them to assign specific tasks to multiple computers or to assign it to multiple processing units inside the same computer. One of the first noticable introduction of parallel computing to the computer science world was brought in the early 70's when researchers from the University of Illinoi built the ILLIAC IV computer that was later used by the NASA *Bouknight et al. (1972)*.

Multi-Processing

Currently, the vast majority of modern computers run on more than 2 processing units. It is one of the most important specification that is looking at when people are shopping for computer. Depending on the kind of usage the buyers are looking for, they can go with 2, 4, 6 and even more than 8 cores. For computational expensive task such as video editing, gaming, or computer science, it is recommended to use 6 cores or more. This gives the ability to run multiple tasks on several CPU in parallel resulting in a faster computation time. When people need to tackle very computing expensive tasks they will usually pool multiple computers into one entity that will be able to distribute and split the task to all the computers. Those entities are usually what people call clusters. Nowadays, since the connections to link computers with each other are very fast, it is also very common to talk about clusters as supercomputers. As of 2020, the supercomputer that had the most core was the Fugaku with 7,630,848 cores *Kodama et al. (2020)*.

Parallel Task

The best way to understand a parallel task is to compare it with a real-life situation. We can take as an example a school project where the final task is to present a 30 page report. One way of tackling this job is by doing every individual part of the report (introduction, methodology, analysis, etc.) in a sequential manner (by the same person) meaning the total time of the work will be the sum of all it's individual tasks duration. This method is fairly simple and does not demand much coordination at the cost of being very slow to accomplish and also not fully using its ressources (only 1 student in the team is working).



Task Management

However, we can use a methodology that is much more effective when dealing with big projects by separating tasks between 3 students in a parallel way. Each student works individually on one task of the assignment with the objective to combine their work at then end. The total length of the project will be dependent on the longest task duration as well as the coordination time of combining all the work together.

In computer science terminology, the coordination time is called the overhead time and the student can be represented as clusters, workers, cores or threads.

Another important factor in a parallel task consists of a good distribution of jobs. For a parallel task to be fully optimized, each core needs to be assigned a similar length else there will be a bottleneck effect and many cores will patiently wait for their next task, wasting precious resources.

Finally, when building a parallel task, we need to specify how to combine the information gathered by every worker. In R, most of the parallel libraries let us decides the way to combine all the tasks output. For example, we can receive the information as a list, as matrix, as a vector or even as an aggregation of all the different outputs.

Parallelization in Data Science

Most of data science projects consists of working with relatively large datasets. Whether it is during data exploration or while building regression models, parallel computation can almost always help to speed up processes. Take for example this famous formula that allow to find the weights of the parameters of a linear regression model: $(X^T X)^{-1} X^T Y$

This formula can easily be parallelized by separating the formula in 2 parts. We could first let one worker work on the first part of the equation and then assign the second part of the equation to the second worker. Afterwards those two results could be combined again after they have been processed. Worker 1: $(X^T X)^{-1}$ Worker 2: $X^T Y$ On top of that, we could even go at a deeper level than that since in matrix multiplication we can decompose the product of each vectors into separate tasks which can also result in significative speedup.

Parallel computation is also found in multiple machine learning libraries. For example, the Caret library in R uses this technique when doing cross-validation and they often to allow users to use parallel computation in their functions.

As for data exploration, we often want to apply specific function to all rows or columns in the dataset using for loops. Having massive dataset may scare off users to apply these functions due to the time constraints, but multiple solutions exist to tackle this inconvenience. We will explain more in detail how to overcome this problem in the following sections.

Before jumping into the analysis, we will first describe the existing libraries that every data scientist should know to speed up their workflow.

Libraries

Parallel is the first library to get familiar with. It is a simple library containing functions that can be use in parallel tasks.

One of the most important building block of the *parallel* library is the function detectCores(). This function gives the number of CPU cores and threads available on the computer.

To find the number of threads available:

```
print(detectCores())
```

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

To find the number of cores available:

```
nClust = detectCores(logical = FALSE)
print(nClust)
```

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

This information will be very important later on when we will create clusters.

The difference between a thread and a core is that modern physical cores contain multiple threads, allowing them to run more than one task at a time, but when specifying to run parallel task in R we have to allocate the tasks to different cores.

Secondly, the makeCluster() function creates a set of N copies of R running in parallel that are able to communicate with each other *Team (2020)*. It is important to mention the number of clusters inside parallel functions and it is recommended to leave one free core for other operations.

Making a cluster that will be used by parallel processes:

```
makeCluster(nClust-1)
```

```
## socket cluster with 3 nodes on host 'localhost'
```

Once the environment is set up and ready for parallel computation, we can use libraries such as doParallel *Corporation and Weston (2020)* which provides a parallel backend to run parallel tasks.

There is a tutorial available in the vignette of the foreach package *Microsoft and Weston (2020)* that brightly explains how to exploit this package. It explains how for each loops can be used in replacement of simple for loop functions. It also goes in depth into several parallel task examples including a quick sort algorithm.

The doParallel library allows users to use mainstream functions such as lapply but in a parallel manner using ParLapply. A small online blog (https://www.r-bloggers.com/2016/07/lets-be-faster-and-more-parallel-in-r-with-doparallel-package/ (https://www.r-bloggers.com/2016/07/lets-be-faster-and-more-parallel-in-r-with-doparallel-package/)) showcase the time advantage to use such libraries by trying to calculate all the prime numbers between 10 and 10,000. Using the following line of codes they were able to get more then 50% decrease in computation time.

Create a function to find prime numbers

```
#Function to get prime numbers
getPrimeNumbers <- function(n) {
  n <- as.integer(n)
  if(n > 1e6) stop("n too large")
  primes <- rep(TRUE, n)
  primes[1] <- FALSE
  last.prime <- 2L
  for(i in last.prime:floor(sqrt(n)))
  {
    primes[seq.int(2L*last.prime, n, last.prime)] <- FALSE
    last.prime <- last.prime + min(which(primes[(last.prime+1):n]))
  }
  which(primes)
}
```

Use the above function in a traditional for loop

```
#With for loop
index <- 10:10000
result <- c()
time.forloop = system.time(for (i in index) {result[[i]] <- getPrimeNumbers(i)})[3]
print(time.forloop)
```

```
## elapsed
##  37.747
```

Use it in a lapply manner

```
time.lapply = system.time(lapply(index, getPrimeNumbers))[3]
print(time.lapply)
```

```
## elapsed
##  36.417
```

Use a parLapply wrapper

```
library(doParallel)
Nclust = detectCores(logical = FALSE) -1
cl = makeCluster(Nclust)
registerDoParallel(cl)
#ParLapply
time.parl = system.time(result <- parLapply(cl, index, getPrimeNumbers))[3]
print(time.parl)
```

```
## elapsed
##  20.505
```

Using foreach and dopar

```
#Foreach
time.foreach = system.time(result <- foreach(i=index) %dopar% getPrimeNumbers(i))[3]
stopCluster(cl)
print(time.foreach)
```

```
## elapsed
##  65.407
```

Stoping the cluster In the second line of the last code chunk, we can see the instruction **stopCluster**, this instruction allows us to shutdown the cluster that we created via the instruction makecluster and to free the core of our computer and let them be used by other processes.

Another important aspect of data science is to be able to handle high dimensional data in an effective way. Libraries such as torch *Falbel and Luraschi (2021)* gives the opportunity to do computation on tensors. Tensors are multidimensional data structures that are easy to manipulate and highly used in neural networks. There is multiple tutorials available on their r documentation and it contains numerous functions helping to build neural networks. The main advantage of torch is that it can connect to a GPU which is a powerful calculating tool.

GPU

We talked briefly about CPU in previous section as a calculator for the computer. The GPU acts in a similar way but is optimized for parallel computation and is the main weapon of choice for modern machine learning model. As we said before, the cpu can have multiple cores (in average 2 to 8) and processes can be parallelized through all those cores. Recent GPUs have in general multiple thousands of cores, however those cores can only do very simple tasks and are not useful to do branching tasks (Task using conditions or recursions statements). The specialty of GPUs is then to do tasks like addition and substraction that we call floating point operations (flops). Neural networks relies heavily on matrix multiplications which are composed of multiplications and additions.

GPUs are very expensive tools that cost in the range of $500 to $300,000 and are usually hard to access. Most of laptops in the world do not necessarily have access to gpu, but it is something we wanted to showcase in our project. Fortunately, virtual machines available on Google Colab platform gives access to powerful GPUs such as the NVIDIA V100 to everyday programmers. For this reason, we decided to write a small r jupyter notebook that can be run in the Google Colab environment and that is allowing us to install all the packages we need in a stable environment before knitting our document.

Next sections will focus on concrete example of parallel computation in data science projects.

Application TO Monte Carlo Simulation

Simulating data in parallel using MC Simulation

In this section, we will begin with building Monte Carlo simulations in parallel. Monte Carlo simulation is a statistical method which relies on repeatedly running in a loop by selecting random scenarios for each run. MC simulations can be used for various scenarios. We will use it for simulating 100 random values for two variables a and b from a finite sample with a cap of 10000 draws. For each 100 values generated in a cluster, we will use foreach argument ‘combine’ to combine the results of all the clusters generations for each variable. For example, if the no of detected cores in the machine running the below simulation is 8, then the no of generated random values for a and b would be 8 for each. The manner in which these combinations occur is by estimating the random deviates order for each simulation of random values to be generated before generating the values.

We simulate the same random values for variables a and b using two functions foreach and mclapply. As we proceed, we will observe more about the values generated in parallel before combining them to get 100 random values for each variable. We will next compare the time taken to compute the simulation in total to distinguish between the two functions.

Before simulating in parallel, lets first try running the monte carlo simulation in sequential order to check how fast or slow is the code really running on a single machine.

Running in sequence using foreach

```
set.seed(2021)
N=100
NofDraw=10000

#combine argument combines/sums up piece wise each element of the list or each element * 8
 (no of cores) = results after combining in final "vec"
sortedMCsimlis_seq <- function(x){
  dat <- data.frame(a=sample(1:1000, x, replace=T),
                    b=sample(1:10, x, replace=T ))

  vec <- foreach(ND=rep(ceiling(NofDraw/Nclust), Nclust),.combine='+') %do% {
    z <- integer(x)
    for(i in 1:ND) {
      set.seed(78)
      pos <- sort.list(rbeta(x, shape1=dat$a, shape2=dat$b) ) #estimating random deviates or
der for random values to be generated
      z[pos] <- z[pos] + 1:x #random values generated
    }
    z
  }

  sortedlis <- sort.list(-vec)
  rand <- dat[sortedlis,]
  return(cbind(rand$a,rand$b))
}

randseq <- sortedMCsimlis_seq(N)
summary(randseq)
```

```
##        V1              V2
##  Min.   : 67.0   Min.   : 1.00
##  1st Qu.:334.2   1st Qu.: 3.00
##  Median :507.0   Median : 5.00
##  Mean   :503.6   Mean   : 5.40
##  3rd Qu.:710.8   3rd Qu.: 7.25
##  Max.   :990.0   Max.   :10.00
```

```
#CPU TIME ESTIMATES
systime_mc_FE_seq <- system.time(sortedMCsimlis_seq(N))
systime_mc_FE_seq[3] #CPU time of foreach package
```

```
## elapsed
##   0.637
```

Running in parallel using foreach

```
set.seed(2021)
N=100
NofDraw=10000
Nclust <- detectCores()-1
cl <- makeCluster(Nclust)
clusterSetRNGStream(cl,iseed = set.seed(234))
registerDoParallel(cl)

#combine argument combines/sums up piece wise each element of the list or each element * 8
 (no of cores) = results after combining in final "vec"
sortedMCsimlis <- function(x){
  dat <- data.frame(a=sample(1:1000, x, replace=T),
                    b=sample(1:10, x, replace=T))

  vec <- foreach(ND=rep(ceiling(NofDraw/Nclust), Nclust),.combine='+') %dopar% {
    z <- integer(x)
    for(i in 1:ND) {
      set.seed(78)
      pos <- sort.list(rbeta(x, shape1=dat$a, shape2=dat$b) ) #estimating random deviates or
der for random values to be generated
      z[pos] <- z[pos] + 1:x #random values generated
    }
    z
  }

  sortedlis <- sort.list(-vec)
  rand <- dat[sortedlis,]
  return(cbind(rand$a,rand$b))
}

rand1 <- sortedMCsimlis(N)
summary(rand1)
```

```
##        V1              V2
##  Min.   : 67.0   Min.   : 1.00
##  1st Qu.:334.2   1st Qu.: 3.00
##  Median :507.0   Median : 5.00
##  Mean   :503.6   Mean   : 5.40
##  3rd Qu.:710.8   3rd Qu.: 7.25
##  Max.   :990.0   Max.   :10.00
```

```
#CPU TIME ESTIMATES
systime_mc_FE <- system.time(sortedMCsimlis(N))
systime_mc_FE[3] #CPU time of foreach package
```

```
## elapsed
##   0.333
```

```
stopCluster(cl)
```

Running in parallel using parSapply

```
set.seed(2021)
N=100
NofDraw=10000
Nclust <- detectCores()
cl <- makeCluster(Nclust)
clusterSetRNGStream(cl,iseed = set.seed(234))
ND=rep(ceiling(NofDraw/Nclust), Nclust)
registerDoParallel(cl )
dat <- data.frame(a=sample(1:1000, 100, replace=T),
                  b=sample(1:10, 100, replace=T))

sortedMCsimlis_3 <- function(ND){
  z <- integer(100)
  dat <- data.frame(a=sample(1:1000, 100, replace=T),
                    b=sample(1:10, 100, replace=T))

  for(i in 1:ND) {
    set.seed(78)
    pos <- sort.list(rbeta(100, shape1=dat$a, shape2=dat$b) ) #estimating quantile function
 order for random values to be generated
    z[pos] <- z[pos] + 1:100 #random values generated
  }
  z

}
comb=parSapply(cl,100,sortedMCsimlis_3)
sortedlis <- sort.list(-comb)
rand <- dat[sortedlis,]
# return(cbind(sort(rand$a),sort(rand$b)))
rand3 = cbind(rand$a,rand$b)
summary(rand3)
```

```
##        V1              V2
##  Min.   : 67.0   Min.   : 1.00
##  1st Qu.:334.2   1st Qu.: 3.00
##  Median :507.0   Median : 5.00
##  Mean   :503.6   Mean   : 5.40
##  3rd Qu.:710.8   3rd Qu.: 7.25
##  Max.   :990.0   Max.   :10.00
```

```
#CPU TIME ESTIMATES
systime_mc_PSA <- system.time(parSapply(cl,100,sortedMCsimlis_3))
systime_mc_PSA[3] #CPU time of foreach package
```

```
## elapsed
##   0.062
```

```
stopCluster(cl)
```

Let’s compare if randomly generated values are the same for our satisfaction :

```
#comparing between the two simulated data
#first sorting
x1=sort(randseq)
x2=sort(rand1)
x4=sort(rand3)
#as the below function does element wise comparison,using the sorted vectors of simulated da
ta
sum(diff(x1==x2))
```

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

```
sum(diff(x2==x4))
```

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

```
#as the sum of difference between the sorted vectors is same, we know that simulated data is
same for all 3 functions: foreach and parSapply
```

Now that we have proven that the simulated data is the same using all 3 functions, lets plot the time they take to compute in parallel.

```
# Matrix that will hold all the timing data
time_matmc = c()
time_matmc = rbind(time_matmc, c('MC sim with foreach seq', systime_mc_FE_seq[3]))
time_matmc = rbind(time_matmc, c('MC sim with foreach', systime_mc_FE[3]))
time_matmc = rbind(time_matmc, c('MC sim with parSapply', systime_mc_PSA[3]))
max(time_matmc[,2])
```

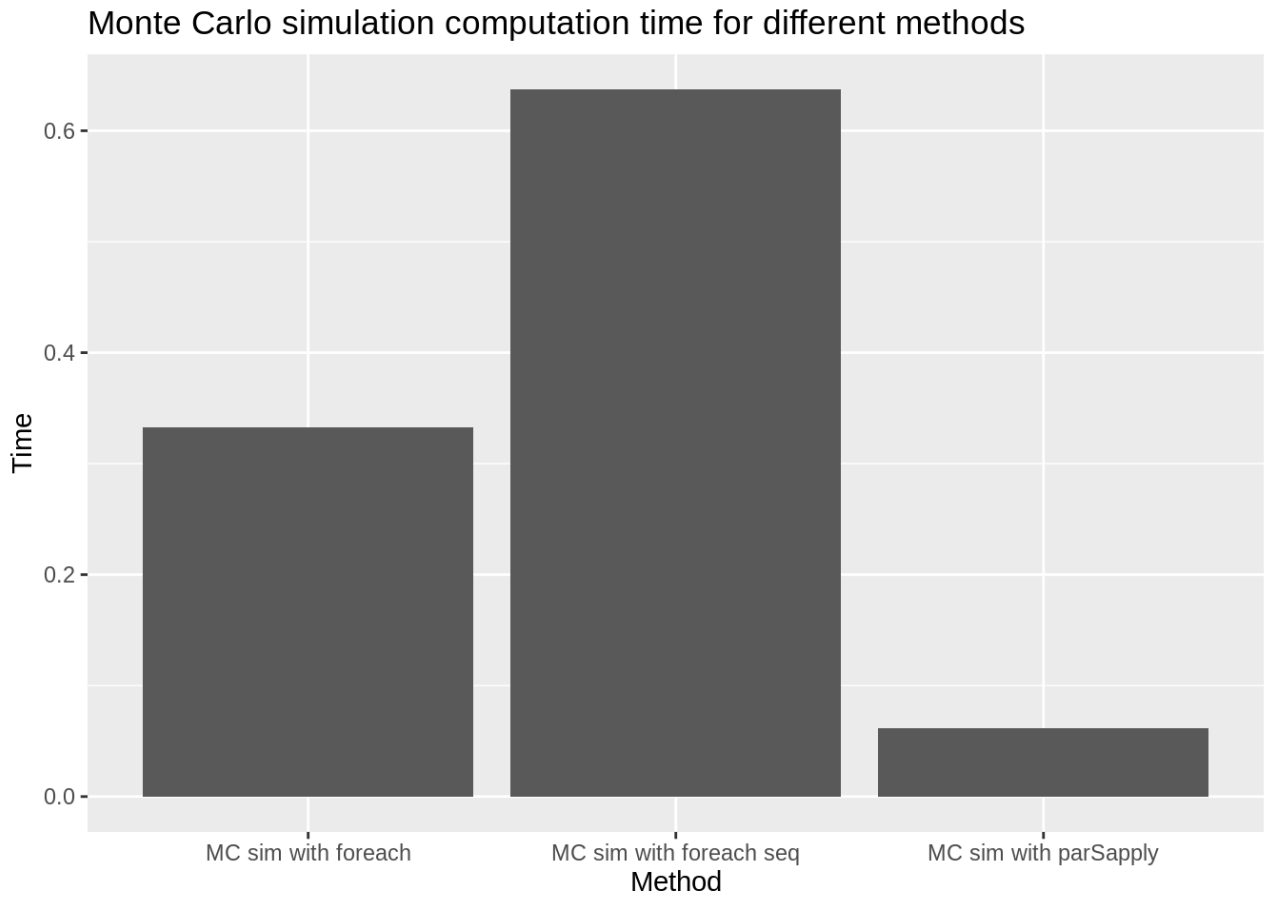
```
## [1] "0.637000000000626"
```

```
tm_df = as.data.frame(time_matmc)
colnames(tm_df) = c('Method','Time')
tm_df$Time = as.numeric(as.character(tm_df$Time))
kable_classic_2(kable(tm_df, caption='Result Summary'), full_width=F)
```

Result Summary

Method	Time
MC sim with foreach seq	0.637
MC sim with foreach	0.333
MC sim with parSapply	0.062

```
ggplot(data=tm_df, aes(x=Method, y=Time)) + geom_col() + ggtitle('Monte Carlo simulation com
putation time for different methods')
```



Here we can see that the three parallel methods really improved the processing duration of the computation of our MonteCarlo simultation. We can also see that the fastest method for our simulation is actually the parSapply.

Dataset Exploration

In this part of the project, we will use the Dota2 Games Results DataSet from the UCI Machine Learning Repository *Dua and Graff (2017)*. Dota2 (https://www.dota2.com/ (https://www.dota2.com/)) is an online video game that is played by millions of users everyday. A game of Dota2 involves 2 teams of 5 players facing each other. During a match, each player must chose exactly one hero and each hero cannot be selected multiple times.

The chosen dataset shows information related to 102,944 Dota2 games. Each row represents the final outcome of a game from the point of view of the blue team. The columns at index 5 and up all represent if one hero was picked and if yes by which team. A value of 1 in the appropriate column means it was pick by the blue team and -1 means it was picked by the red team. The dataset also contains information about which team won the match and some other in-game statistics. However, for the first part of this study we will solely focus on the hero selection information. When the data was extracted, there was a total of 113 available champions in the game and the training set recoreded 92,649 games. Therefore, our first data set will contain 113 columns and have 92,649 rows.

Importation of Dota2 Dataset

```
nb_heroes = 113
#Renaming the heroes' variables
colnames_hero = paste0('H',1:nb_heroes)
colnames_df = c('Win', 'Cluster', 'GameMode', 'GameType',  colnames_hero)
data_train = read.csv('dota2Train.csv', col.names = colnames_df)

# Drop First columns containing other game statistics
# data_heroes = data_train[1:10,-(1:4)]
data_heroes = data_train[,-(1:4)]

# Matrix that will hold all the timing data
time_mat = c()
```

Dataset Overview and description

```
data_heroes[1:10,1:10]
```

```
##    H1 H2 H3 H4 H5 H6 H7 H8 H9 H10
## 1   0  0  0  1  0 -1  0  0  0   0
## 2   0  0  0  1  0 -1  0  0  0   0
## 3   0  0  0  0  0  0 -1  0  0   0
## 4   0  0  0  0  0 -1  0  0 -1   0
## 5   0  1  0  0  0  0  0  0  0   0
## 6   0  0  0  0  0  0  0  0  0   0
## 7   0 -1  0  0  0  0  0  0  0   0
## 8   0  0  1  0  0  0  0 -1  0   0
## 9   0  0  0  1  0  0  0 -1  0   0
## 10  0  1  0  0  1  1  0 -1  0   0
```

To interpret a little bit the dataset, we show in this extract a sample of 10 games for the first 10 heroes. We can see for instance that in game 1 and 2, the hero #4 was chosen in the blue team while the hero #6 was chosen in the red team. In the 10th game we also see that the hero #2, #5 and #6 were chosen in the same team (blue) while the hero #8 was chosen in the red team.

Most popular hero picks

The first task that we will try to tackle with parallel computing is a very simple task of exploratory data analysis. We will try to find the heroes that were picked the most often in a same team. Even if this task is simple, the considerable size of the dataset would be a great opportunity to showcase parallel computation.

Finding the duo of heroes that was picked the most often in the same team across all games.

To find the duo that was played the most often through all games, the first intuition of any analyst could be to compare each champion columns with each other and to count the number of times where the sum of those two columns' elements was equal to either 2 or -2. We want to return the pair of herores that are picked together the most often. Finally, we can quickly understand that we cannot compare a column with itself and that the comparison of the column i with j will give the same results as j with i, which means that we can only look at the upper triangular matrix of the comparison results:

$\underset{i \in H, j \in H}{\arg\max} \sum_{g=1}^{G} [|x_{i,g} + x_{j,g}| = 2], \forall j > i$ Where H is the set of heroes (here 113) and G is the number of games analysed (here 92 649).

Simple double nested for loops integration of the algorithm to find the most popular duo

```
compare_heroes = function(data_heroes){
  nb_heroes = ncol(data_heroes)
  mat_res = c()
  for (i in 1:(nb_heroes-1)){
    for (j in (i+1):nb_heroes){
      hero_presence = data_heroes[, i] + data_heroes[, j]
      hero_same_team = sum(hero_presence %in% c(-2, 2))
      mat_res = rbind(mat_res, c(i, j, hero_same_team))
    }
  }
  return(mat_res)
}

time_func = system.time(hero_comps_two <- compare_heroes(data_heroes))
print(time_func[3])
```

```
## elapsed
##  15.192
```

We can see that this operation is very fast (under 20 seconds) even though without parallel computation. This method will not allow us to appreciate all the power of R parallel libraries, therefore we will use the same operation but this this we will try to find the trio of heroes that was played the most (comparing 3 columns at a time instead of 2). This will allow us to have a better grasp of the magnitude of speed gain obtained by parallel computation.

Finding the trio of heroes that was picked the most often across all games.

This time we need to compare all the different hero columns with each other 3 times for each games. This also means that we will only count the number of time that the sum of the selected columns is either -3 or 3.

$$\underset{i \in H, j \in H, k \in H}{\arg\max} \sum_{g}^{G} [|x_{i,g} + x_{j,g} + x_{k,g}| = 3], \forall k > j > i$$

Where H is the number of heroes (here 113) and G is the number of games analyzed (92,649).

Simple triple nested for loops integration of the algorithm to find the most popular trio

```
compare_heroes = function(data_heroes){
  nb_heroes = ncol(data_heroes)
  mat_res = c()
  for (i in 1:(nb_heroes-2)){
    for (j in (i+1):(nb_heroes-1)){
      for (k in (j+1):nb_heroes){
        # If the N evaluated champions cols are picked, the sum of a given row will either b
e N or -N
        hero_same_team = sum((data_heroes[, i] + data_heroes[, j] + data_heroes[, k]) %in% c
(-3, 3))
        mat_res = rbind(mat_res, c(i, j, k, hero_same_team))
      }
    }
  }
  return(mat_res)
}

time_func = system.time(final_matrix <- compare_heroes(data_heroes))
time_1 = time_func[3]
print(time_1)
```

```
## elapsed
##   897.3
```

```
time_mat = rbind(time_mat, c('Hero trios sequential', time_1))
max(final_matrix[, 4])
```

```
## [1] 502
```

Now, we can see how our simple implementation can be long to run using simple nested R for loops. It took more than 8 minutes to run the whole process, now let see if we can decrease that number using parallel computing.

Using the doParallel Package to find the most played hero trio

The doParallel package allows us to use a very simple approach to transform a normal R for loop into a vectorized process that will run on each core of a CPU. To use the package, we only need to create a foreach loop and add a %dopar% keyword. As mentioned, it is also important that we create a cluster of workers before launching the parallel computations.

```
# Cluster creation
cores=detectCores()
print(cores)
```

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

```
cl = makeCluster(cores[1]-1)
registerDoParallel(cl)

compare_heroes_par = function(data_heroes){
  nb_heroes = ncol(data_heroes)
  res_mat = foreach(i=1:(nb_heroes-2), .combine=rbind) %dopar% {
    mat_temp = c()
    for(j in (i+1):(nb_heroes-1)) {
      for(k in (j+1):nb_heroes) {
        hero_same_team = sum((data_heroes[, i] + data_heroes[, j] + data_heroes[, k]) %in% c
(-3, 3))
        mat_temp = rbind(mat_temp, c(i, j, k, hero_same_team))
      }
    }
    mat_temp
  }
  return(res_mat)
}

time_func = system.time(final_matrix_par <- compare_heroes_par(data_heroes))
time_2 = time_func[3]
print(time_2)
```

```
## elapsed
##  308.21
```

```
time_mat = rbind(time_mat, c('Hero trios dopar', time_2))
max(final_matrix_par[, 4])
```

```
## [1] 502
```

```
# Stopping the cluster after the calculations ended
stopCluster(cl)
```

Impressively, we can see that the time of computing almost linearly decrease with our number of cores. The time went from 897.3 secs to 308.21 which is a gain of 2.9 speed while using 4 cpu cores.

Using the mclapply function of the parallel library

The mclapply function from the parallel library is very simple to use also. In cases where programmers are already using R apply functions, it is very interesting to use this approach since it would mean little code changes for possibly important computing time reductions. In our case, since we were already using nested for loops, it was necessary to change the code and create a sub function that we could parallelize.

```
# In our case to be able to use the same loops as before, but in an apply
# we need to create a function to pass the inner loops inside the apply method
compare_heroes_for_i = function(i){
  mat_temp = c()
  for(j in (i+1):(nb_heroes-1)) {
    for(k in (j+1):nb_heroes) {
      hero_same_team = sum((data_heroes[, i] + data_heroes[, j] + data_heroes[, k]) %in% c(-
3, 3))
      mat_temp = rbind(mat_temp, c(i, j, k, hero_same_team))
    }
  }
  return(mat_temp)
}

# Apply the compare_heroes_for_i to each i using the data set
compare_heroes_apply = function(data_heroes){
  nb_heroes = ncol(data_heroes)
  i=1:(nb_heroes-2)
  mclapply(i, compare_heroes_for_i, mc.cores = cores - 1)
}

# Calculating the processing time
time_func = system.time(final_matrix_apply <- compare_heroes_apply(data_heroes))
time_3 = time_func[3]
print(time_3)
```

```
## elapsed
## 326.345
```

```
time_mat = rbind(time_mat, c('Hero trios mclapply', time_3))

final_matrix_apply = do.call(rbind, final_matrix_apply)
max(final_matrix_apply[, 4])
```

```
## [1] 502
```

The results show that the mclapply gets results that are very close to the one of doparallel in term of speed (326.345 s vs 308.21 s respectively). Since both packages offer very similar results, it is important to note that a user could indeed choose either of the two packages according to the user preferences (using apply or loops).

Optimizing the code to perform matrix operations instead of R for loops

The usage of for loops and even more “R for loops” is almost always a bad sign of optimization. Since R is an interpreted language, we will always loose performance using pure R code if we compare it to any optimized R package that was compiled in C. It is not always easy to visualize, but when it is possible, a problem should almost always be transformed into a matrix/tensor form.

Matrix operations can be done very efficiently on computer systems nowadays, since they have been subject to optimization in low level languages for decades through low level packages such as LaPack, BLAS and MLK. Matrix operations are what constitute the majority of modern computer floating point operations and they also are at the center of the majority of the predictive models and are the building blocks of neural networks.

Dataset transformation

In our case, we can decompose the dataset in a manner that will allow us to do some tensor operations. Since, we need a final results that has 3 dimensions of size 113 (our number of heroes), we can start by transforming the dataset to allow us to directly use sumproducts between three chosen columns. To achieve that we will double the initial size of our initial data set to obtain only 1 and 0 in all the dataset but having two times more rows.

```
heroes_mat = as.matrix(data_heroes)
heroes_mat_all = rbind(heroes_mat > 0, heroes_mat < 0)
class(heroes_mat_all) = 'numeric'
heroes_mat_all[1:10,1:10]
```

```
##       H1 H2 H3 H4 H5 H6 H7 H8 H9 H10
##  [1,]  0  0  0  1  0  0  0  0  0   0
##  [2,]  0  0  0  1  0  0  0  0  0   0
##  [3,]  0  0  0  0  0  0  0  0  0   0
##  [4,]  0  0  0  0  0  0  0  0  0   0
##  [5,]  0  1  0  0  0  0  0  0  0   0
##  [6,]  0  0  0  0  0  0  0  0  0   0
##  [7,]  0  0  0  0  0  0  0  0  0   0
##  [8,]  0  0  1  0  0  0  0  0  0   0
##  [9,]  0  0  0  1  0  0  0  0  0   0
## [10,]  0  1  0  0  1  1  0  0  0   0
```

Changing the optimization function

Now that we only have 0 or 1s, if we want to see how many times 3 selected heroes were chosen during one game, we can calculate the sum of the element wise product of the three vector. $\underset{i \in H, j \in H, k \in H}{\arg\max} \sum_{g}^{G} x_{i,g} \cdot x_{j,g} \cdot x_{k,g}, \forall k > j > i$

Where H is the number of heroes (here 113) and G is **2** times the number of games analyzed (here 1.85298^{5}).

Reusing the dopar package algorithm on the transformed dataset

```
cores=detectCores()
cl = makeCluster(cores[1]-1)
registerDoParallel(cl)

compare_heroes_par = function(data_heroes){
  nb_heroes = ncol(data_heroes)
  res_mat = foreach(i=1:(nb_heroes-2), .combine=rbind) %dopar% {
    mat_temp = c()
    for(j in (i+1):(nb_heroes-1)) {
      for(k in (j+1):nb_heroes) {
        hero_same_team = sum((data_heroes[, i] * data_heroes[, j] * data_heroes[, k]))
        mat_temp = rbind(mat_temp, c(i, j, k, hero_same_team))
      }
    }
    mat_temp
  }
  return(res_mat)
}

time_func = system.time(final_matrix_par <- compare_heroes_par(heroes_mat_all))
time_4 = time_func[3]
print(time_4)
```

```
## elapsed
## 322.316
```

```
time_mat = rbind(time_mat, c('Hero trios dopar 0-1', time_4))
max(final_matrix_par[, 4])
```

```
## [1] 502
```

```
stopCluster(cl)
```

We can see that the results did not really moved in term of speed for the dopar function using the new dataset and even that the operations took more time to execute with that. This is due to the fact that we got one less comparison to do, but that we have 2 times more iteration to do (again loops in R). Therefore we could ask ourselves, why did we create that dataset?

Using einsum to do matrix operations

Since we need to calculate that vector sumproduct in 3 dimensions, there is no very trivial way to produce those tensor operations directly from R. However, we can use a simple operation called *einsum* that is available in almost all matrix/tensor libraries like Tensorflow or Torch (or numpy in python). Our favorite explanation of that function comes from the documentation of Tensorflow (https://www.tensorflow.org/api_docs/python/tf/einsum (https://www.tensorflow.org/api_docs/python/tf/einsum)).

Take $C_{i,k} = \sum_j A_{i,j} B_{j,k}$ or $C[i, k] = sum_j A[i, j] * B[j, k]$

1. remove variable names, brackets and commas, (ik = sum_j ij * jk)
2. replace "*" with "," (ik = sum_j ij , jk)
3. drop summation signs, and (ik = ij, jk)
4. move the output to the right, while replacing "=" with "->." (ij,jk->ik)

Using Tensorflow's explanation for our data set we can apply the same exact steps to our data. We know that we want to find the $\arg\max$ of the following:

$$C_{i,j,k} = \sum_{g}^{G} x_{g,i} \cdot x_{g,j} \cdot x_{g,k} \text{ or } C[i,j,k] = sum_g A[g,i] * A[g,j] * A[g,k]$$

1. (ijk = sum_g gi * gj * gk)
2. (ijk = sum_g gi, gj, gk)
3. (ijk = gi, gj, gk)
4. (gi, gj, gk -> ijk)

The einsum function that we will need to use is then "gi, gj, gk -> ijk"

Using the torch library to calculate the einsum

To accomplish this task, we will use the torch library to perform our tensor operations. Torch is a *Tensors and Neural Networks with 'GPU' Acceleration* library available on the CRAN repository (and for python) that is soaring in popularity in recent years. Even if we can use a GPU, let's see what is the result of running this algorithm on CPU in torch using the power of matrix/tensor computing library. If you kept attention, you will understand that we will do more than the double the amount of operations needed, since we will compute all the possibility of columns comparison. Then, the question is will we still get better results using torch on CPU?

```
#Function take the original dataframe and the device we will use (cpu or gpu)
# Outputs the value and the position of the most popular trio
compare_heroes_torch = function(data_heroes, device_big_ops){
  # convert dataframe to tensor
  heroes_mat_all = torch_tensor(data_heroes, device = 'cpu', dtype=torch_float32())
  tensor_heroes = heroes_mat_all$to(device=device_big_ops)

  #Execute the einsum
  a = torch_einsum('gi,gj,gk->ijk', list(tensor_heroes, tensor_heroes, tensor_heroes))

  #Find position of the argmax while reshaping the result tensor
  shape_tot = torch_flatten(a)$shape
  pos = torch_arange(1, shape_tot+1, device = device_big_ops)
  i = torch_floor_divide(pos, (nb_heroes*nb_heroes))
  j = torch_floor_divide(pos-i*nb_heroes*nb_heroes, nb_heroes)
  k = pos - i * nb_heroes * nb_heroes - j * nb_heroes
  pos = torch_vstack(c(pos, i+1 , j+1, k, torch_flatten(a)))

  # Take only the upper pyramidal part of the tensor to avoid repetitions
  pos = pos[,pos[2,]<pos[3,]]
  pos = pos[,pos[3,]<pos[4,]]

  pos_max = torch_argmax(pos[5,])
  max_val = pos[,pos_max]
  return(max_val)
}

#Time elapsed calculations
time_func = system.time(final_matrix_apply <- compare_heroes_torch(heroes_mat_all, "cpu"))
time_5 = time_func[3]
print(time_5)
```

```
## elapsed
##   8.927
```

```
print(final_matrix_apply)
```

```
## torch_tensor
##  103725
##       9
##      14
##     104
##     502
## [ CPUFloatType{5} ]
```

```
time_mat = rbind(time_mat, c('Hero trios Torch CPU', time_5))
```

We can see how using a parallelized tensor operation library can optimize the calculation speed of our algorithm already. We see a factor of speed increase of 36.1 vs the parallelized for loops approach. One of the important things to note here is that, even if an algorithm is parallelized it does not mean by all means that it is gonna be faster than a non parallelized but optimized algorithm. In our case, since torch is by default very optimized for tensor computations and is alreasy parallelized by default on cpu. We can a get phenomenal performance boost. In the results of the last output (in final_matrix apply) we saw the trio that appeared the most often in the resulting tensor at position 2, 3 and 4 respectively. In the last position of this tensor we also see the number of occurence this trio was taken in all the games by the same team.

Running einsum on torch and on GPU

Nowadays, the majority of the people who specifically want to parallelize vector/matrix/tensor operations will use Graphical Computing Unit (GPU). This is due to the fact that GPU possess thousands of simple cores that can only due specific instructions related to tensor floating points operations. We can easily use torch with a GPU, the only step that needs to be done is to transfer the data that need to be computed on GPU to the GPU memory. The interfacing between the cpu and the gpu memory is the slow part in the process, but once the data is on the GPU, the speedup can be very considerable.

```
time_func = system.time(final_matrix_apply <- compare_heroes_torch(heroes_mat_all, device_bi
g_ops))
time_6 = time_func[3]
print(time_6)
```

```
## elapsed
##   0.366
```

```
print(final_matrix_apply)
```

```
## torch_tensor
##  103725
##       9
##      14
##     104
##     502
## [ CUDAFloatType{5} ]
```

```
time_mat = rbind(time_mat, c('Hero trios Torch GPU', time_func[3]))
```

We can see how using a GPU while performing tensor operations can optimize the calculation speed of out algorithm. We see a factor of speed increase of 24.4 vs the CPU torch method and compared which is very significant. We can see in final_matrix_apply again that the same results giving information about which trio was picked the most often is identical, which proves that our algorithm worked well.

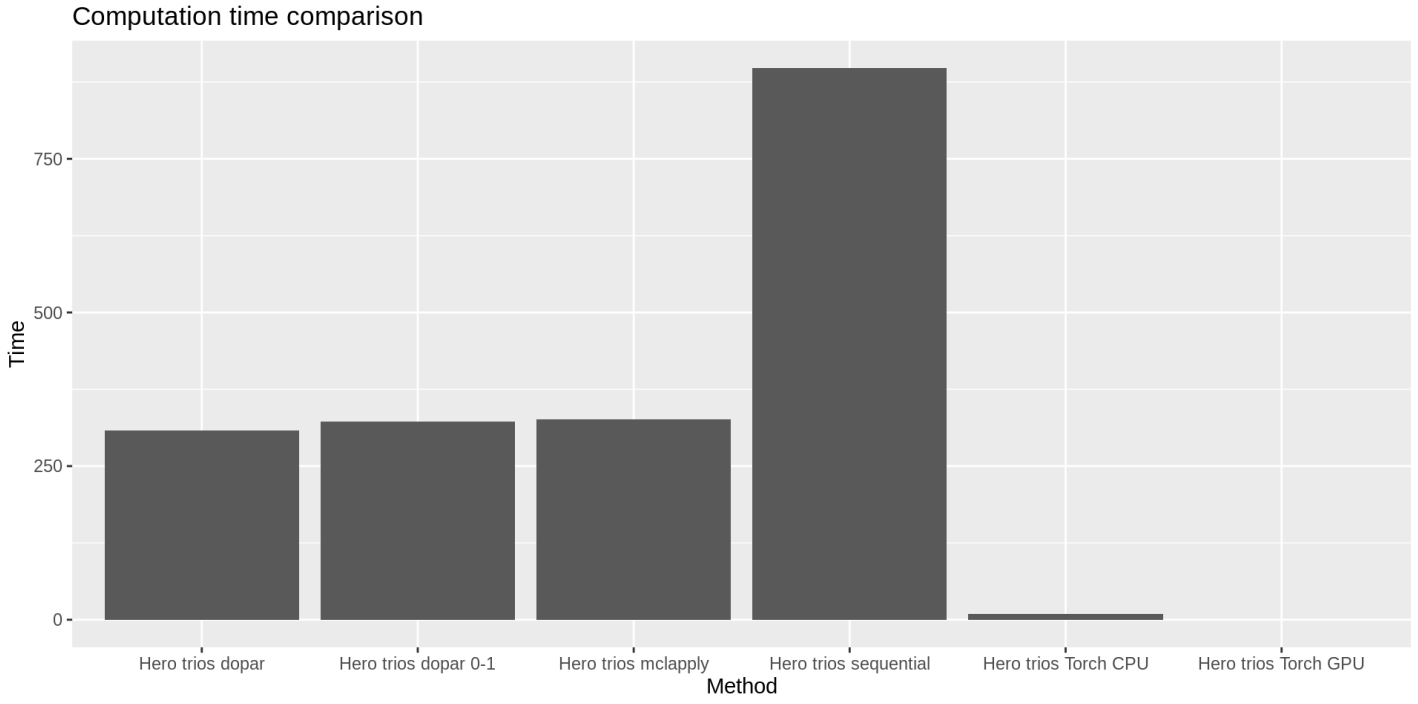
Results and Comparisons

```
tm_df = as.data.frame(time_mat)
colnames(tm_df) = c('Method', 'Time')
tm_df$Time = as.numeric(as.character(tm_df$Time))
kable_classic_2(kable(tm_df, caption='Result Summary'), full_width=F)
```

Result Summary

Method	Time
Hero trios sequential	897.300
Hero trios dopar	308.210
Hero trios mclapply	326.345
Hero trios dopar 0-1	322.316
Hero trios Torch CPU	8.927
Hero trios Torch GPU	0.366

```
ggplot(data=tm_df, aes(x=Method, y=Time)) +  geom_col() +  ggtitle("Computation time compari
son")
```



We can see a huge difference between the sequential and all the parallel methods. We can also see that the transformation of the problem into tensor computation had a very significant impact on the processing speed. Finally, we can see that for that type of computation, using a GPU provides the best results.

Parallel computing for random forests

One of the very popular data science models that can be improve with parallel computing are the ensemble methods. Since models like random forests are based on the aggregation of multiple trees and since each of those models do not dependent on each other during their creation, process like random forests can greatly be improved by parallel computing.

We will then assess the performance of multiple R packages that allow us to create random forests models. We will still use the Dota2 dataset that we used previously and add the information about game types and game categories to the hero information. In each random forests, we will try to predict in each game if if the blue team would have win the game. We will start by using simple random forests of 20 trees to give us an idea of their performance and verify their results. Nowadays, predicting the outcome of a game could be very useful. It can be useful when you are picking your hero if you want to know which one would increase your chance of winning. It can also be useful when beting on esports (online gaming), since this kind practice is more and more common.

Importation & Transformation of the Dota2 Dataset

```
nb_heroes = 113
colnames_hero = paste0('H',1:nb_heroes)
colnames_df = c('Win', 'Cluster', 'GameMode', 'GameType',  colnames_hero)

# Training dataset importation
data_train = read.csv('dota2Train.csv', col.names = colnames_df)
# We remove the cluster information from the dataset because we don't think this variable ca
rry a lot of meaning
data_train = data_train[,-2]
# Transformation of -1, 1 to 0 and 1
data_train$Win = (data_train$Win + 1) / 2
data_train[] = lapply(data_train[], as.factor)
data_train_x = data_train[,-1]
data_train_y = data_train$Win

# Test dataset importation
data_test = read.csv('dota2Test.csv', col.names = colnames_df)
data_test = data_test[,-2]
data_test$Win = (data_test$Win + 1) / 2
data_test[] = lapply(data_test[], as.factor)
data_test_x = data_test[,-1]
data_test_y = data_test$Win

res_mat = c()
```

Simple randomForest model

This first model that we will use a baseline will be a simple randomForest using only one CPU core and building the forest sequentially.

```
# Sequential Random Forest
rf_seq = function(ntree, data_train_x, data_train_y, data_test_x, data_test_y){
  rf_name = 'randForest_seq'
  time_func = system.time(rf_seq_fit <- randomForest(x=data_train_x,
                                                     y=data_train_y, ntree=ntree))
  # Time and results
  time_elapse = round(time_func[3], 2)
  # Acc calculation
  preds = predict(rf_seq_fit, newdata=data_test_x, type='response')
  acc = round(mean(preds == data_test_y)*100, 2)
  return(c(rf_name, ntree, time_elapse, acc))
}

res_rf = rf_seq(20, data_train_x, data_train_y, data_test_x, data_test_y)
res_mat = rbind(res_mat, res_rf)

# Creation of a function to display the random forest results
display_rf_df = function(res_rf_mat){
  res_rf_df = as.data.frame(t(res_rf_mat))
  colnames(res_rf_df) = c('model name', 'nb tree', 'calc time', 'accuracy')
  kable_classic_2(kable(res_rf_df, caption='Result Summary'), full_width=F)
}

display_rf_df(res_rf)
```

Result Summary

model name	nb tree	calc time	accuracy
randForest_seq	20	33.42	55.25

Using the randomForest in conjunction with the doParallel package

Here we leverage the doParallel package to be able to add parallelizitation to the random forest tasks. Here the random forests are built in parallel in each node of the cluster and then the foreach method combine them using the randomForest combine method which allow to combine all the ensemble models together.

```
# Create a cluster using only 4 cores
nb_cluster = 4
cl = makeCluster(nb_cluster)
# Set seed on the cluster
clusterSetRNGStream(cl,iseed = set.seed(234))
registerDoParallel(cl)


rf_par = function(ntree, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y){
  rf_name = 'randForest_par'
  # Parallel random forest nested function
  # For now we will assume that the nb of trees is always a multiple of the total nb of clus
ters
  nested_par_rf = function(ntree, nb_cluster, data_train_x, data_train_y){
    nb_batch = floor(ntree/nb_cluster)
    # In the combine portion we let the random forest packeage reassemble the different
    # forests together after each task has finish running
    rf = foreach(ntree=rep(nb_batch, nb_cluster), .combine=combine,
                 .packages='randomForest', .multicombine = TRUE)  %dopar%
      randomForest(x=data_train_x, y=data_train_y, ntree=ntree)
    return(rf)
  }
  # Time elapsed
  time_func = system.time(rf_par_fit <- nested_par_rf(ntree, nb_cluster, data_train_x, data_
train_y))
  time_elapse = round(time_func[3],2)
  # Acc calculation
  preds = predict(rf_par_fit, newdata=data_test_x, type='response')
  acc = round(mean(preds == data_test_y)*100, 2)
  return(c(rf_name, ntree, time_elapse, acc))
}

res_rf = rf_par(20, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
res_mat = rbind(res_mat, res_rf)
display_rf_df(res_rf)
```

Result Summary

model name	nb tree	calc time	accuracy
randForest_par	20	13.23	54.82

Using the randomForestSRC package

The randomForestSRC package is a package that was introduce to us during the semester. It is very interesting to know that this package is actually using parallel computing by default under the hood. It is using OpenMP shared-memory parallel programming which can significantly increase random forest computation speed.

```
rf_src = function(ntree, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y){
  # Limiting the number of cpu cores for randomForestSRC
  options(rf.cores = nb_cluster)
  rf_name = 'randForestSRC'
  time_func = system.time(rf_src_fit <- rfsrc(Win ~ ., data = data_train, ntree = ntree))
  # Time and results
  time_elapse = round(time_func[3], 2)
  # Acc calculation
  preds = predict(rf_src_fit, newdata=data_test_x, type='response')
  acc = round(mean(preds$class == data_test_y)*100, 2)
  return(c(rf_name, ntree, time_elapse, acc))
}
res_rf = rf_src(20, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
res_mat = rbind(res_mat, res_rf)
display_rf_df(res_rf)
```

Result Summary

model name	nb tree	calc time	accuracy
randForestSRC	20	34.57	55.87

Ranger library

The ranger library Wright and Ziegler (2015) is a powerful library that uses a fully optimised C++ library and that is natively parallelized. In the initial paper of the package, ranger was defined as the *fastest and most memory efficient implementation of randomforests to analyze data on the scale of a genome-wide association study*. It is promising a very strong scaling for high number of features, trees and splits which might prove useful for our dota2 dataset.

```
# By default ranger use the same seed as R so we don't need to change anything here
rf_rngr = function(ntree, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y){
  rf_name = 'RF_Ranger'
  time_func = system.time(rf_ranger_fit <- ranger(Win ~ ., data = data_train, num.trees = nt
ree, num.threads = nb_cluster))
  # Time and results
  time_elapse = round(time_func[3], 2)
  # Acc calculation
  preds = predict(rf_ranger_fit, data=data_test_x, type='response')$predictions
  acc = round(mean(preds == data_test_y)*100, 2)
  return(c(rf_name, ntree, time_elapse, acc))
}
res_rf = rf_rngr(20, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
res_mat = rbind(res_mat, res_rf)
display_rf_df(res_rf)
```

Result Summary

model name	nb tree	calc time	accuracy
RF_Ranger	20	10.39	56.09

Effect of the number of trees on the package execution time

We are going to look at the speed of execution of the above mentioned package while changing the number of trees for each random forest. This will allow us to assess if some package seems to perform better when the number of trees and the volume of computation increase.

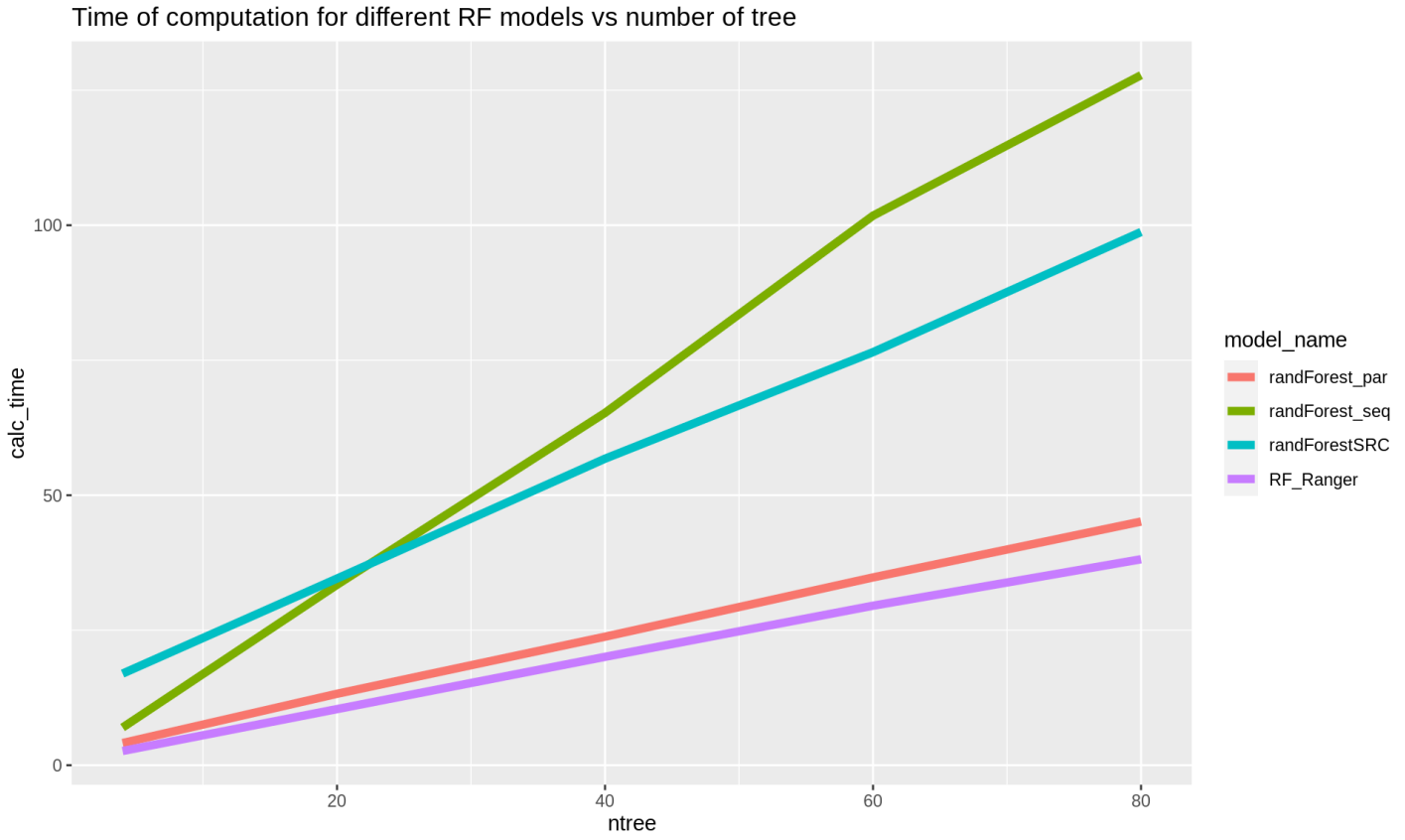
```
# Set of all the different number of trees per forest
tree_set = c(4, 40, 60, 80)
for (i in tree_set){
  res_rf = rf_seq(i, data_train_x, data_train_y, data_test_x, data_test_y)
  res_mat = rbind(res_mat, res_rf)
  res_rf = rf_par(i, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
  res_mat = rbind(res_mat, res_rf)
  res_rf = rf_src(i, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
  res_mat = rbind(res_mat, res_rf)
  res_rf = rf_rngr(i, nb_cluster, data_train_x, data_train_y, data_test_x, data_test_y)
  res_mat = rbind(res_mat, res_rf)
}
```

```
## Growing trees.. Progress: 86%. Estimated remaining time: 4 seconds.
```

```
res_rf_df = as.data.frame(res_mat)
colnames(res_rf_df) = c('model_name', 'ntree', 'calc_time', 'accuracy')
res_rf_df$calc_time = as.numeric(as.character(res_rf_df$calc_time))
res_rf_df$ntree = as.numeric(as.character(res_rf_df$ntree))

speed_up = round(res_rf_df[res_rf_df$model_name == 'randForest_seq' & res_rf_df$ntree == 100
, 3]
          / res_rf_df[res_rf_df$model_name == 'RF_Ranger' & res_rf_df$ntree == 100, 3] ,1)

ggplot(res_rf_df, aes(x=ntree, y=calc_time, group=model_name, colour=model_name)) +
            geom_line(size=2) +  ggtitle("Time of computation for different RF models vs num
ber of tree")
```



In this figure we can clearly see that the slope of the different packages are very different from each other. We can see that the package ranger seems to be the best in all cases and that it is resilient to the augmentation of number of trees. All in all, it is interesting to see that the relation ship between the number of trees and the calculation time is linear and that the best package to do a random forest in term of execution speed seems to be the ranger package followed by the simple doPar method coupled with the random forest package. In the end, it is very impressive to see the performance delivered by the ranger package, since it increase the speed of the random forest creation by fold. In some cases, we saw that the randomForestSRC gave slower performance (even slower than the sequential methods) when the number of trees where very low. However, when increasing the number of trees, it becomes more efficient than the sequential method.

Conclusion

Throughout this report we hope you have apprehended the powerful tool that is parallel computation. This subject does not demand a background in statistics, nor does it demand complex equation to solve problems. It is however a tool that will let you leverage the data you own in a more efficient way. Indeed, we started by showing you intuitive examples with simulated data proving how easy it is to implement to a program. We then tackle the problem of data exploration by trying to find the most used trios and duos of the Dota 2 games with computationally intensive algorithm. This section was a good example of finding optimal way to handle big dataset and showed different matrix manipulation approach to optimize efficiency. Finally, the last example showed a concrete example of parallel task in machine learning environment. By comparing a sequential random forest with parallel random forest, we showed that using the ranger library could lead to a significant speed up of the training phase.

There 2 are important lessons to hold from this work. 1. Before starting a project and going with parallelization, one must have a good idea of how to streamline and simplify a problem. By using the right libraries and the right operations, plenty of minutes could be save. 2. Parallelization is not meant for small problem. Overhead time and compilation could eat the gain made from parallelizing task.

To conclude, we briefly talked about the usage of GPU with the help of Google's virtual machine. This new trend of shared resources is allowing people from all over the globe to access top of the line equipment and is a big opportunity for everyone interested in machine learning. It will become more and more crucial to understand how to use these platforms as the size of data keeps growing.

In this work, we showed you multiple parallel computation packages. However, there is a lot more packages available in CRAN to do parallel computing like TensorFlow for tensor computation.

REFERENCES

Bouknight, W. J., S. Denenberg, D. McIntyre, J. M. Randall, A. Sameh, and D. Slotnick. 1972. "The Illiac IV System."

Corporation, Microsoft, and Steve Weston. 2020. "doParallel: Foreach Parallel Adaptor for the 'Parallel' Package." https://cran.r-project.org/package=doParallel (https://cran.r-project.org/package=doParallel).

Dua, Dheeru, and Casey Graff. 2017. "UCI Machine Learning Repository." University of California, Irvine, School of Information; Computer Sciences. http://archive.ics.uci.edu/ml (http://archive.ics.uci.edu/ml).

Falbel, Daniel, and Javier Luraschi. 2021. "Torch: Tensors and Neural Networks with 'GPU' Acceleration." https://cran.r-project.org/package=torch (https://cran.r-project.org/package=torch).

Kodama, Y., T. Odajima, Eishi Arima, and M. Sato. 2020. "Evaluation of Power Management Control on the Supercomputer Fugaku." *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 484–93.

Microsoft, and Steve Weston. 2020. "Foreach: Provides Foreach Looping Construct." https://cran.r-project.org/package=foreach (https://cran.r-project.org/package=foreach).

Moore, Gordon. 2006. "Cramming More Components onto Integrated Circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, Pp.114 Ff." *Solid-State Circuits Newsletter, IEEE* 11 (October): 33–35. https://doi.org/10.1109/N-SSC.2006.4785860 (https://doi.org/10.1109/N-SSC.2006.4785860).

Roser, Max, and Hannah Ritchie. 2013. "Technological Progress." *Our World in Data*.

Team, R Core. 2020. "R: A Language and Environment for Statistical Computing." Vienna, Austria: R Foundation for Statistical Computing. https://www.rdocumentation.org/packages/parallel/versions/3.6.2 (https://www.rdocumentation.org/packages/parallel/versions/3.6.2).

Turing, A. M. 1937. "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* s2-42 (1): 230–65. https://doi.org/https://doi.org/10.1112/plms/s2-42.1.230 (https://doi.org/10.1112/plms/s2-42.1.230).

Wright, Marvin N., and Andreas Ziegler. 2015. "Ranger: A Fast Implementation of Random Forests for High Dimensional Data in c++ and r." https://doi.org/10.18637/jss.v077.i01 (https://doi.org/10.18637/jss.v077.i01).