

Técnicas de Simulación

05. Paralelización y Bootstrapping

Miguel A. Castellanos

Contents

1	Paralelización	1
1.1	Aclaraciones previas sobre la terminología	2
1.2	Dos cosas importantes a tener en cuenta	3
1.3	Dos maneras de paralelizar: forks y sockets	5
1.4	La función lapply	5
1.5	El paquete <i>Parallel</i>	7
1.6	El paquete doParallel	13
1.7	Un último consejo	14
2	Bootstraping	14
2.1	Tarea 1	15
2.2	Tarea 2	15
2.3	Tarea 3	15
2.4	Tarea 4	17

1 Paralelización

La arquitectura de los procesadores de un ordenador actual es algo bastante complejo; en principio está compuesto por un socket que es la típica “pastilla” que solemos conocer (como se ve en la figura siguiente), pero, a su vez, este socket está compuesto por distintos núcleos o “cores”, cada uno de ellos independiente del resto como se ve en la figura inferior.



Figure 1: imagen de un procesador intel

En eso consiste la informática actual; desde hace unos 15 años los ordenadores no son más rápidos, sino más complejos, introduciendo más y más núcleos que permiten realizar tareas independientes en forma paralela.

Antes, cuando abrías Word, se utilizaba el único núcleo disponible para abrirlo, y si mientras tanto realizabas cualquier otra tarea como cambiar de canción, tenías que esperar a que terminase de abrir Word para que el procesador pudiese cambiar la canción. Ahora, al tener varios núcleos, no hay tiempos de espera, ya que las tareas se pueden realizar a la vez porque son encargadas a núcleos distintos, ofreciendo una eficiencia enorme al trabajar con el sistema operativo (un núcleo se encarga de abrir Word y otro de cambiar de canción). Dicho de otra manera, los procesadores no son más complejos con los años, sino que cada vez hay más núcleos en cada procesador.

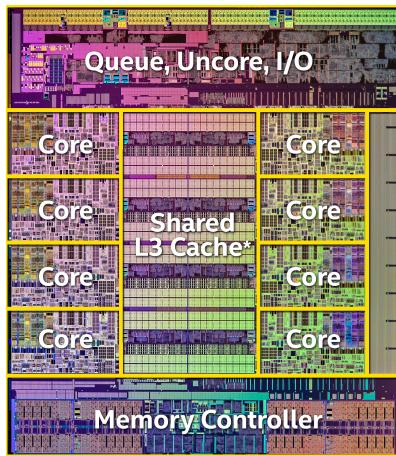


Figure 2: imagen interna de los nucleos de un procesador

Esto no es del todo cierto, por supuesto, es solo una simplificación. El desarrollo de los procesadores en los últimos años está alcanzando hitos muy interesantes como el desarrollo en 5 nanómetros, la arquitectura ARM o el uso de GPUs para procesamiento masivo.

R por defecto solo utiliza un núcleo, así que cuando arrancamos un proceso (un cálculo) hay que esperar a que termine para arrancar el siguiente. Esto no tiene demasiada importancia cuando trabajamos en estadística convencional porque los tiempos de ejecución son pequeños, pero en simulación, bigdata, machine learning, etc., sí es una cuestión muy relevante.

¿Habría alguna forma de utilizar todos esos núcleos que tenemos parados en nuestro procesador para acelerar el trabajo?

Sí, por supuesto, es lo que se llama la **parallelización de procesos** y es una de las ramas más interesantes de la informática (y también de las más complejas). Aquí vamos a ver algunas de las opciones más comúnmente utilizadas en R para parallelizar procesos.

Lo primero que tienes que aprender es a visualizar los núcleos de tu ordenador; simplemente, si trabajas con Windows, teclea **Ctr+Shift+Esc** y podrás ver el gestor de tareas de Windows (como el de la siguiente figura) donde aparece el número de procesadores físicos y virtuales que posee tu máquina.

Como podemos ver en el ejemplo de esta figura disponemos de 4 cores (núcleos) con dos hilos por core, lo que nos da 8 procesos en paralelo como máximo. Si quisieramos monitorizar en tiempo real el trabajo de nuestros núcleos tendríamos que abrir el *Monitor de Recursos* de Windows, simplemente clicando en la parte inferior del gestor de tareas. Para los que trabajen en Linux con Ubuntu pueden utilizar *htop* o el *gnome-monitor* y en mac el *monitor de actividad*.

1.1 Aclaraciones previas sobre la terminología

Llamamos socket al procesador o “pastilla” o CPU que controla el ordenador; este a su vez está compuesto por varios “cores” o núcleos. En la figura anterior, un i7 de intel, podemos distinguir 4 núcleos físicos diferentes. Además, cada uno de estos cores físicos, en los procesadores modernos y de alta gama, puede tener hasta dos

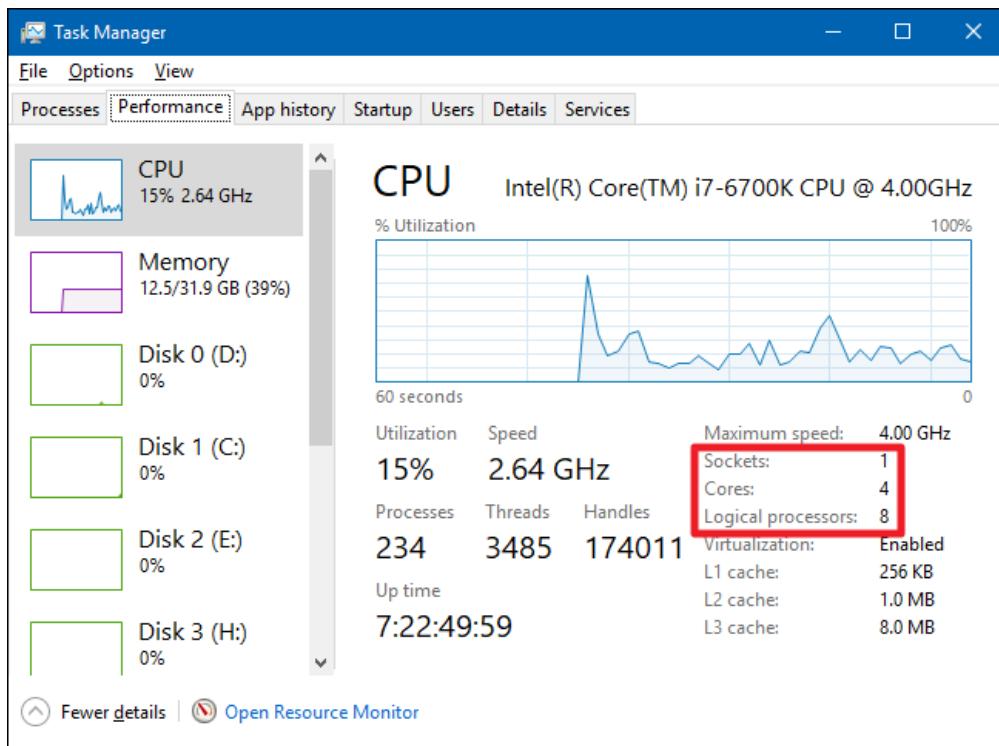


Figure 3: Task Manager de Windows

cores virtuales (esto es más complicado de explicar, se basa en la tecnología [Hyper-Threading](#)) por lo que, en nuestro ejemplo, tendríamos 1 (socket) x 4 (cores físicos) x (2 cores virtuales) = 8 cores virtuales o threads (hilos). Es decir, con ese procesador podríamos “lanzar” 8 procesos simultáneamente, acortando los tiempos de nuestra simulación a 1/8. No todos los procesadores permiten 8 hilos, los más antiguos permiten solo 2 y las versiones más potentes (como la Xeon de intel, o los Ryzen de AMD) llegan hasta los 32 físicos x 2 = 64 cores virtuales. Ahora veremos cuántos hilos podemos tener con nuestro ordenador.

Además, esta situación se puede poner más interesante: nosotros usamos ordenadores que tienen un único procesador (un socket), pero es posible tener ordenadores con múltiples sockets (2, 4, 12 CPUs), que a su vez se organizan en torno a clusters de ordenadores. Cuando el número es muy grande ya se les empieza a llamar **supercomputadores** donde suelen competir USA con China por tener los más potentes. La lista de los 500 mejores ([wikipedia](#)) varía cada poco, y nosotros aportamos el marenostrom, que, a fecha de hoy, es el más potente de Europa ([elpaís.es](#)).

Como anécdota, una forma interesante de paralelizar procesos es tener millones de ordenadores separados por el mundo que se conectan entre sí para hacer algún tipo de cálculo complejo cuando no están dedicados a otras tareas (cuando están en reposo). Esta es la estrategia de computación distribuida que utilizó el SETI para detectar vida extraterrestre y que se ha dejado de utilizar hace muy poco ([seti](#)).

1.2 Dos cosas importantes a tener en cuenta

Como he dicho, la paralelización es una de las ramas más activas e interesantes de la informática y va desde el uso de los cores de nuestra CPU hasta la supercomputación cuántica; pero en todos los niveles hay limitaciones, y es que hay tareas que se pueden parallelizar y otras no, ya que por definición los hilos son independientes (bueno, realmente esto no sería estrictamente cierto para la computación cuántica).

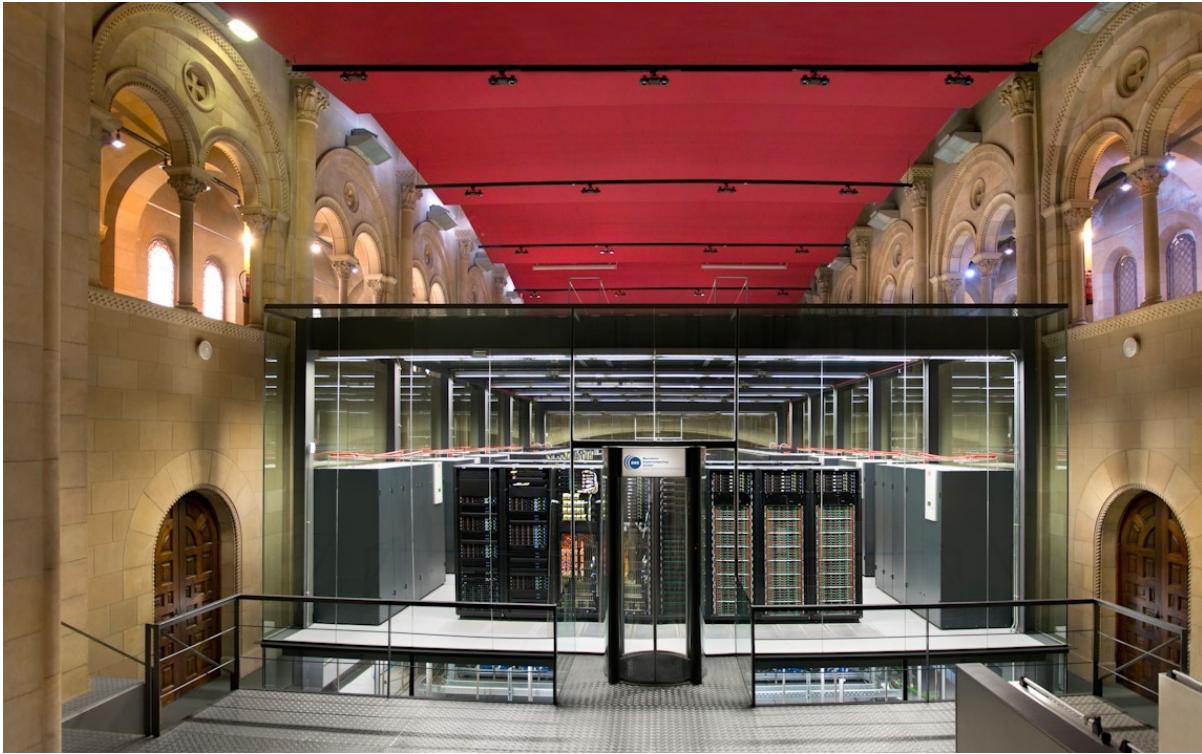


Figure 4: superordenador marenostrom, Barcelona

1.2.1 Independencia de los procesos

Los procesos ejecutados en hilos independientes están aislados, es decir, no se pueden comunicar entre sí. Los procesos son iniciados por algún organizador de procesos, como el sistema operativo, que los va lanzando según vayan terminando los anteriores, pero no hay garantía de cuándo se inician y cuándo se terminan.

Fíjate en la imagen anterior, R controla los 6 cores de un procesador. Hay control para generar los hilos y hay control para recoger los resultados al final, pero no hay control (o al menos este no es trivial) sobre lo que ocurre en medio. Eso significa que en el momento del inicio se lanzarán los 6 primeros procesos, pero cada uno de ellos tardará un tiempo distinto; por ejemplo, si el primero en acabar es el proceso del core4, el proceso 7 se mandará al core4. Dicho de otra manera, no es posible saber qué pasa “en medio” entre el inicio y el final de los procesos.

Eso quiere decir que si un proceso requiere de información de otros procesos para realizar una tarea no es posible paralelizarlo. Por ejemplo, tenemos una tarea en la que hay un sistema temporal y para calcular un valor en t_1 necesito conocer el valor anterior t_0 , pues en ese caso no puede paralelizarse (no puede ir cada ciclo en un proceso distinto), tendrá que correr todo por el mismo core. Ahora bien, si por otro lado quiero correr 1000 sistemas dinámicos con distintos parámetros para conocer el efecto de esos parámetros sobre el resultado, entonces esa sí es una situación perfecta para paralelizar, ya que cada uno de esos procesos (independientes) los puedo mandar a un core distinto.

1.2.2 Costes de la paralelización

Lo segundo importante a tener en cuenta es que la paralelización es ventajosa cuando la tarea es muy demandante, si no, no merece la pena. Cuando paralelizamos, R y el sistema operativo tienen que realizar tareas de creación, gestión y control de los procesos y eso supone un gasto en tiempo y recursos. Si la tarea es sencilla, la penalización por esa gestión es superior al beneficio de usar múltiples cores y no compensa. No

es interesante paralelizar para el típico análisis estadístico de datos, lo es cuando estamos en simulaciones o cálculos masivos.

1.3 Dos maneras de paralelizar: forks y sockets



Existen dos formas de paralelizar procesos en R: los forks y los sockets.

Forks es la manera más sencilla de paralelizar en R, consiste en crear una **copia exacta** (un clon) del kernel de R (con sus librerías, variables creadas, etc.) en cada uno de los núcleos; al terminar el proceso la copia se destruye y solo queda el kernel original. De esta manera cada proceso “sabe” todo lo que sabía el kernel en el momento de iniciar la paralelización.

Sockets lo que hace es **crear un nuevo kernel** de R en cada uno de los núcleos. Es como si hubiésemos abierto 8 sesiones distintas de R. Así, cuando cargo unos datos o importo una librería en R tengo que abrirla a la vez en todos los núcleos.

La primera forma es sin duda la más sencilla, pero no está disponible para Windows, solo para sistemas POSIX (Linux y Mac, por si necesitabais otro motivo para cambiarlos). Cada uno tiene sus ventajas y sus inconvenientes:

1. Sockets funciona en Windows y forks no
2. Forks es más sencillo
3. Forks es más rápido
4. Con sockets tienes que crear las variables implicadas en cada socket, en forks es automático
5. En forks, al ser clones, si no tenemos cuidado pueden producirse interacciones no deseadas con las distribuciones aleatorias. Se puede evitar controlando `set.seed()`.

1.4 La función lapply



Muchas de las funciones paralelizadas de R son variantes de lapply, por eso primero vamos a explicar (o recordar) su uso:

`lapply` es la versión `apply` para listas; acepta un vector o lista y una función y devuelve una lista con los resultados.

```
lapply(lista_o_vector, función, resto_de_argumentos...)  
v <- 1:100 # creamos un vector  
  
# basico  
r <- lapply(v, log) # calculamos el logaritmo de v  
class(r) # devuelve una lista  
  
## [1] "list"
```

Podemos pasarle nuestra propia función `foo`, el **primer argumento de la función siempre es el que recoge cada uno de los elementos del vector** (o lista), en nuestro caso `x`.

```
# nuestra propia funcion foo  
foo <- function(x){  
  x+1  
}  
  
# v se pasa automaticamente como el primer argumento de foo (x)  
r <- lapply(v, foo)
```

Pero podemos pasarle todos los argumentos que necesitemos:

```
# pasando mas argumentos a foo
foo <- function(x, y, z){
  x+y+z
}

r <- lapply(v, foo, y = 2, z=3)
r <- lapply(v, foo, 2, 3)
```

Una forma habitual de escribir el código es ahorrarnos darle nombres a las funciones, compactando todo el código en una única sintaxis.

```
r <- lapply(v, function(x, y=2, z=3){
  x+y+z
})
```

En vez de un vector podemos pasar a lapply una lista, lo que permite introducir elementos más complejos.

```
l <- list(a=1,b=2,c=3)

foo <- function(x, y, z){
  x+y+z
}

r <- lapply(l, foo, y=1, z=1)
```

Fíjate en esta sintaxis:

```
l <- list(
  rnorm(100),
  rnorm(100))

lapply(l, mean)

## [[1]]
## [1] 0.01588558
##
## [[2]]
## [1] -0.01933049
```

Creamos un lista de dos elementos, cada uno de ellos es un vector aleatorio, después calculamos la media con cada vector.

Una función también relevante es *mapply*, es la versión para múltiples listas y vectores de *apply*. Lo que hace es: en el primer ciclo de la función pasa el primer elemento de todas las listas, en el segundo ciclo pasa todos los segundos elementos de todas las listas, y así sucesivamente. Es más fácil entenderlo con un ejemplo. Ojo, porque a diferencia de lapply el primer argumento es la función y luego el resto de argumentos.

```
mapply(mean, 1:3, 1:3)

## [1] 1 2 3
mapply(
  function(x,y){
    x^y
  }, x=c(2,3), y=c(3,4))

## [1] 8 81
```

```
# pasa a FUN el primer valor de todos los argumentos, luego el segundo, etc
# si faltan argumentos los recicla
```

```
mapply(function(x,y){
  x+y
}, 1:5, 10)
```

```
## [1] 11 12 13 14 15
```

En general, **hay que tener bastante cuidado con toda la familia apply**, pero especialmente con mapply porque su sintaxis puede ser liosa y generar errores inesperados.

1.5 El paquete *Parallel*



El primer paquete del que disponemos para paralelizar en R es *parallel* que permite hacer tanto fork como sockets con él. Lo primero que hay que hacer es conocer el número de cores de los que disponemos en nuestra CPU, tanto físicos como lógicos:

```
require(parallel)
```

```
## Loading required package: parallel
detectCores()

## [1] 4
detectCores(logical=F)
```

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

En mi caso tengo 4 cores físicos y como es antiguo no puedo duplicarlos en cores lógicos (no dispongo de Hyper-Threading). Con esos 4 cores voy a trabajar el resto de la lección.

1.5.1 Forks con parallel

Recuerda que si estás utilizando Windows esto no va a funcionar ya que no se puede hacer forks con ese sistema operativo. Aunque no puedas ejecutarlo léelo para entender el resto de la lección.

El uso es muy sencillo utilizando *mcapply*, que es la versión paralelizada de lapply. Fíjate en el código:

Vamos a crear una función llamada meanDist que básicamente calcula la distribución muestral de la media extrayendo k muestras de n elementos.

```
# uso de mcapply: el mismo que lapply pero con mc.cores
# mejor si tenemos abierto un visor de procesos, en win se puede abrir
# con Ctrl+Shift+Esc
# diseñamos una tarea típica de simulación de las que hemos creado en semanas anteriores
k = 1000 # muestras
n = 100 # elementos

meanDist <- function(x, k, n){
  muestras <- matrix(sample(0:10, k*n, replace = T), nrow = k)
  m <- vector(length=k)
  for (i in 1:k) m[i] <- mean(muestras[i,])
  # print(sprintf("Proceso numero: %d", x)) # solo para ver por donde van los procesos
  return (list(media=mean(m), sd=sd(m))) # devuelve una lista
```

```

}

meanDist(NULL, 1000,100) # solo comprobamos que funciona, por eso x = NULL

## $media
## [1] 5.00516
##
## $sd
## [1] 0.3085741

# fíjate que x no interviene en la ejecución, solo k y n, en este caso lo usamos para que lapply ejecuta
# Repetimos 100 veces (1:100) para generar carga de trabajo a los cores

system.time({
  rlist <- lapply(1:100, meanDist, 10000, 100)
})

##      user    system elapsed
## 12.586    0.004   12.597

system.time({
  rlist <- mclapply(1:100, meanDist, 10000, 100, mc.cores=4)
})

##      user    system elapsed
## 15.004    0.304   5.816

```

La versión paralelizada obtiene valores significativamente inferiores a lapply porque se están utilizando los 4 cores de mi CPU. De hecho, si tenías abierto el monitor de recursos habrás visto algo similar a la siguiente gráfica:

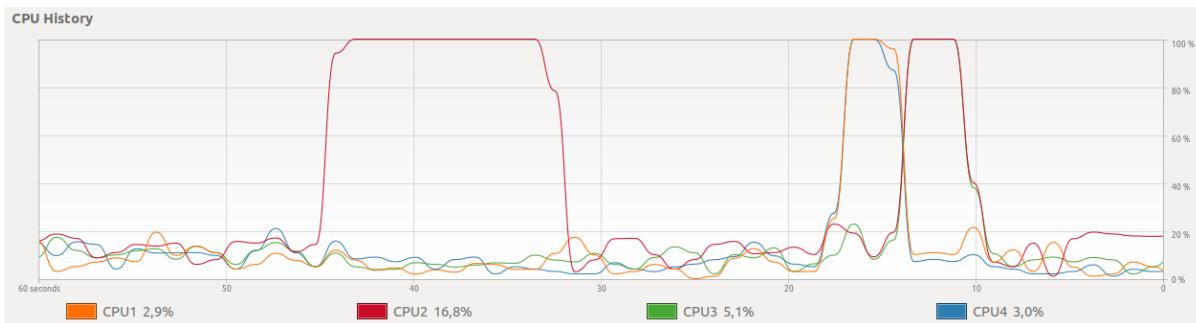


Figure 5: Actividad de los núcleos

En la gráfica se aprecia cómo al mandar el lapply se ha utilizado un único core (el 2, en rojo) que ha consumido unos 12 segundos (entre el 55 y el 33), pero al mandar el mclapply se han lanzado 4 procesos (cores 1, 2, 3 y 4; el 3, el verde, no se ve muy claro pero también ha aumentado su carga de ejecución) tardando significativamente menos todos juntos: 5 segundos. Esta diferencia sería mayor cuanto más demandante fuese la tarea.

Y si quisiésemos dibujar en unos gráficos las medias y las desviaciones típicas encontradas podríamos utilizar el siguiente código:

```
require(ggplot2)
```

```
## Loading required package: ggplot2
```

```

require(gridExtra)

## Loading required package: gridExtra
# extraemos la información de la lista devuelta rlist
media <- sapply(rlist, "[[", "media")
sd <- sapply(rlist, "[[", "sd")

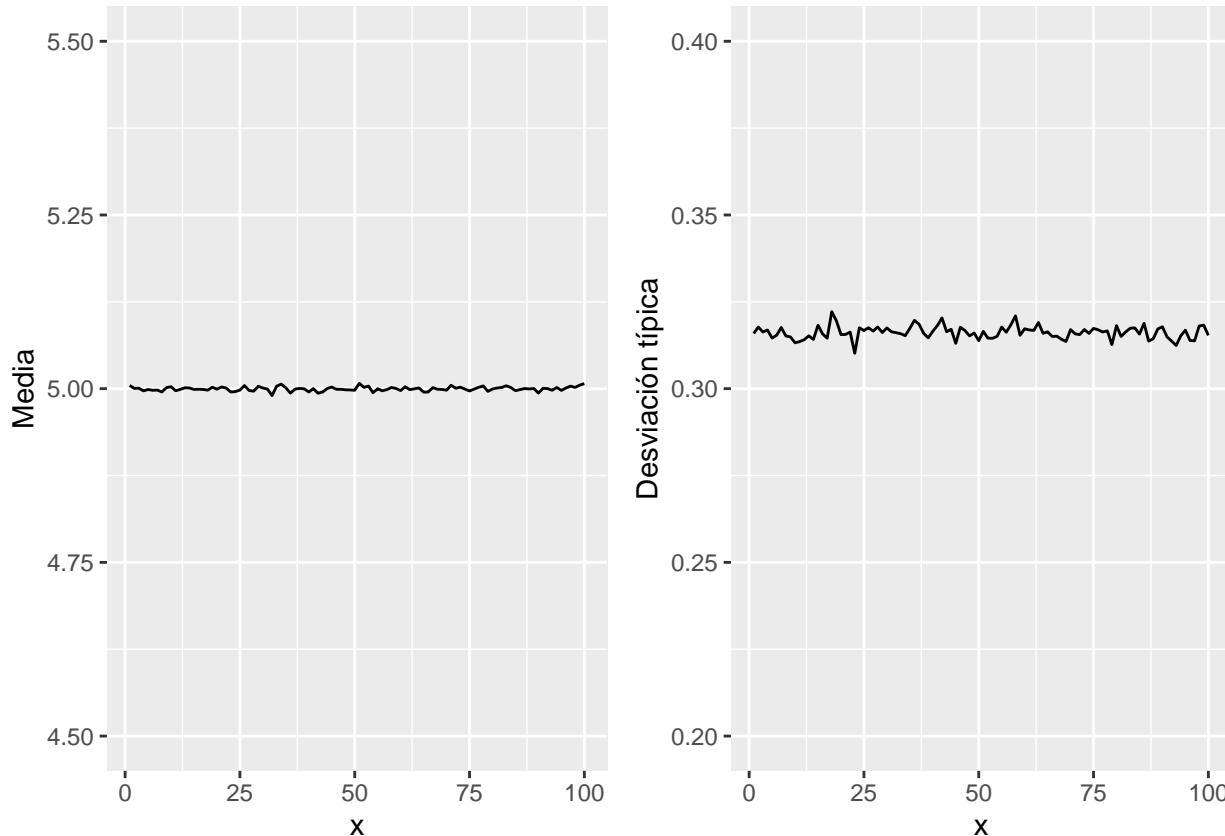
par(mfrow=c(1,2))

datos <- data.frame(x=1:100, y=media)
plot1 <- ggplot(datos, aes(x=x, y=y)) +
  geom_line() +
  ylim(4.5 , 5.5) + ylab("Media")

datos <- data.frame(x=1:100, y=sd)
plot2 <- ggplot(datos, aes(x=x, y=y)) +
  geom_line() + ylim(0.2 , 0.4) + ylab("Desviación típica")

grid.arrange(plot1, plot2, ncol=2)

```



Como podemos interpretar en el gráfico, las medias de las 100 repeticiones (en el eje x, representada con x) de la construcción de la distribución muestral de la media nos da valores similares para cada una de ellas y lo mismo pasa para las desviaciones típicas.

Aunque no se aprecie, arrancar el control de los forks colleva una penalización de tiempo, por eso es interesante paralelizar solo cuando la tarea es altamente demandante de recursos.

El mismo ejemplo podríamos haber puesto usando mcmapply.

1.5.2 Sockets con parallel

Como se ha explicado antes, al crear sockets creamos una nueva instancia de R, completamente independiente de la original, en cada uno de los cores de la CPU.

```
require(parallel)

detectCores()

## [1] 4

detectCores(logical=F)

## [1] 4

cl = makeCluster(4)
```

Con esta sintaxis creamos un cluster de 4 cores, uno por cada uno de los cores de los que dispongo. Así que ahora tengo una sesión de R (la original, a la que llamaremos a partir de ahora **padre**) y 4 sesiones nuevas creadas, a las que llamaremos **hijos**.

Para ejecutar una sentencia en los sockets hay que usar la función clusterEvalQ

```
clusterEvalQ(cl, 2 + 2) # va a ser evaluado en los 4 sockets

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

Hemos obtenido cuatro veces un 4, un resultado por cada uno de las sesiones hijas que hay en cada uno de mis cores.

La dificultad de los sockets reside en comprender que **lo que ocurre en la sesión original (padre) no ocurre también en las sesiones hijas**, por ejemplo fíjate en esta sintaxis:

```
x <- 1

clusterEvalQ(cl, x+1)
```

Nos da un error, porque definimos `x <- 1` en sesión padre, pero no lo hemos definido en las sesiones hijas, pero eso nos dice que no conoce `x`; para que funcione correctamente primero hay que definir `x` en todos los hijos.

```
clusterEvalQ(cl, x <- 1) # define x en los 4 cores

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

```

## 
## [[4]]
## [1] 1
clusterEvalQ(cl, x+1)      # ahora se puede usar

## [[1]]
## [1] 2
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] 2

```

Ahora sí funciona, cada uno de los hijos conoce el valor de x ($x < -1$) y puede calcular $x + 1$. No es complicado, solo hay que tenerlo en cuenta y prestar atención. Un par de consideraciones más:

1. A la inversa también ocurre, si defino algo en los hijos no es conocido por el proceso padre.

```

clusterEvalQ(cl, y <- 10)  # define y en los 4 cores
y + 2 # error porque y no ha sido definida en padre

```

1. Exactamente lo mismo pasa con las librerías, si vas a usar psych necesitas cargarla en todos los hijos

```
clusterEvalQ(cl, require(psych))
```

```

## [[1]]
## [1] TRUE
##
## [[2]]
## [1] TRUE
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] TRUE

```

1. Y para exportar variables o funciones desde padre a los hijos se puede usar **clusterExport()**.

```

# v <- rnorm(100)
# clusterExport(cl, "v")    # ojo fijate que es una cadena "v"
# clusterEvalQ(cl, mean(v)) # todos con el mismo valor, claro
#
# foo <- function(x){
#   sum(x)/length(x)
# }
#
# clusterExport(cl, "foo")
# clusterEvalQ(cl, foo(v))

```

Por último, para ejecutar funciones en cluster se dispone de toda la familia **par*apply** (por ejemplo: parLapply, parMapply, etc.). La sintaxis es similar a las funciones de origen pero son versiones vectorizadas para sockets:

```
parLapply(cluster, lista, funcion, resto_de_argumentos...)
```

Vamos a ver un ejemplo final del uso de sockets con parallel. Vamos a modificar la función *meanDist* para que el primer parámetro que le pasemos le indique el tamaño de la muestra. En vez de tenerlo fijo a n, este valor va ir cambiando desde n=5 a n=100. Es decir, vamos a calcular la media y la sd obtenidas para la distribución muestral de la media, pero variando en cada ciclo el tamaño de la muestra.

```
# Redefinimos meanDist
meanDist2 <- function(n, k){
  muestras <- matrix(sample(0:10, k*n, replace = T), nrow = k)
  m <- vector(length=k)
  for (i in 1:k) m[i] <- mean(muestras[i,])
  # print(sprintf("Proceso numero: %d", x))
  return (list(n=n, media=mean(m), sd=sd(m)))
}

rlist <- parLapply(cl, 5:100, meanDist2, k=10000)

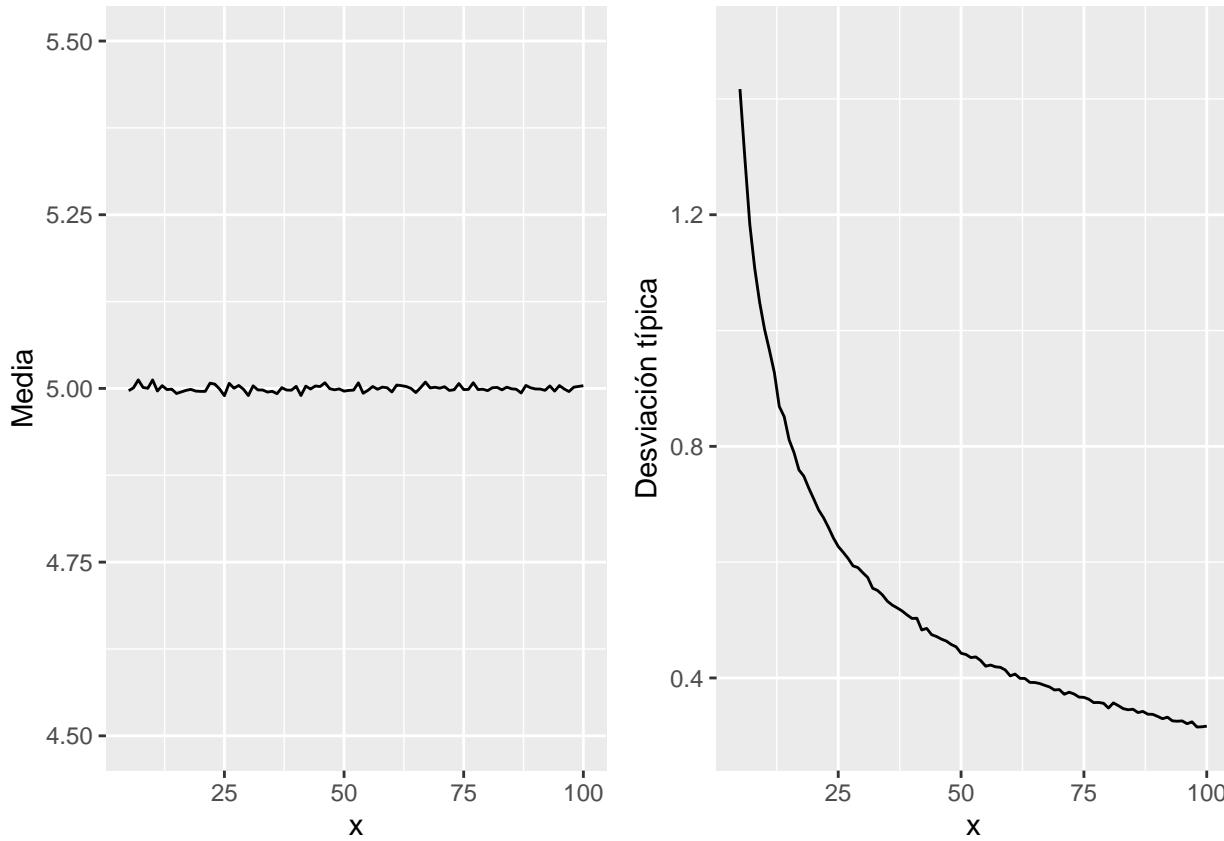
# extraemos la informacion de la lista devuelta rlist y hacemos el dibujo
media <- sapply(rlist, "[[", "media")
sd <- sapply(rlist, "[[", "sd")

par(mfrow=c(1,2))

datos <- data.frame(x=5:100, y=media)
plot1 <- ggplot(datos, aes(x=x, y=y)) +
  geom_line() +
  ylim(4.5 , 5.5) + ylab("Media")

datos <- data.frame(x=5:100, y=sd)
plot2 <- ggplot(datos, aes(x=x, y=y)) +
  geom_line() + ylim(0.3 , 1.5) + ylab("Desviación típica")

grid.arrange(plot1, plot2, ncol=2)
```



Como hemos comprobado en días anteriores, la media de la distribución muestral de la media es siempre la misma (μ) pero la desviación típica disminuye al aumentar el tamaño de la muestra, n : $(\sigma/\sqrt{n - 1})$.

Por último, para terminar y cerrar todas las sesiones hijas de R abiertas en los cores tenemos que cerrar el cluster que hemos creado.

```
stopCluster(cl)
```

1.6 El paquete doParallel

El paquete *doParallel* siempre se usa conjuntamente con el paquete *foreach* y supone el uso de paralelización sin necesidad de recurrir a la familia *apply*, solo utilizando bucles. Tiene la ventaja de que **su uso es el mismo para Linux y para Windows** por lo que no hay que modificar el código si se trabaja con ambos sistemas. En realidad, lo que hace es elegir un sistema de forks o sockets automáticamente dependiendo del sistema operativo, pero para el usuario es directo, no tiene que cambiar el código [stackoverflow](#).

Vamos a repetir la ejecución anterior pero usando *doParallel*; ahora lo vamos a implementar a través de un bucle *foreach* (para cada elemento). Existen dos opciones en esta función, con `%do%` tenemos una versión serial y con `%dopar%` tenemos una versión paralelizada. Al final terminamos cerrando el cluster.

```
library(foreach)
library(doParallel)

## Loading required package: iterators
cl <- makeCluster(4)
registerDoParallel(cl)

system.time({
```

```

k <- foreach (n=5:200) %dopar% {
  meanDist2(n, 10000)
}
## user system elapsed
## 0.116 0.012 11.301
stopCluster(cl)

```

Ambas opciones, parallel y doParallel son perfectas para paralelizar procesos.

1.7 Un último consejo

Cuando se utiliza la paralelización es porque estás diseñando tareas de simulación con alta demanda de recursos y que probablemente se alargarán en el tiempo. En estas situaciones es fácil que se produzca algo que pare la ejecución y la eche a perder: la aparición de NaNs, divisiones por cero, falta de espacio en el disco, apagones de luz, etc. Es interesante escribir el código lo más robusto posible evitando estos problemas, pero aún así es una buena práctica diseñar la ejecución para que se vayan salvando los resultados parciales, de tal forma que no sea necesario volver a empezar desde el principio cuando se produce una ruptura de la ejecución.

2 Bootstrapping

El bootstrap es una técnica estadística desarrollada por Bradley Efron y que está basada en la afirmación de que una distribución teórica puede ser completamente descrita por la extracción de submuestras de una muestra empírica. Por ejemplo, si tenemos una población P , y de ella extraemos una única muestra m , podríamos reconstruir las propiedades de P llevando a cabo submuestreos sucesivos en m . Probablemente a estas alturas del máster el alumno ya conozca sobradamente la técnica de bootstrapping y haya estudiado sus características, tipos, etc., y conozca el paquete `boot`, pero aquí la vamos a programarla desde la lógica para entender su concepto y que luego pueda ser integrada en las simulaciones. La técnica de bootstrap se entiende mejor con un ejemplo en R, vamos a intentar estimar la distribución de nuestro conocido estadístico media:

```

# bootstrap para el estadistico media

N=1000 # tamaño de la poblacion
n=100 # tamaño de la muestra

# creamos un poblacion de N elementos y vemos su media y sd
pob <- rnorm(N, 5, 2)
m.pob = mean(pob)
sd.pob = sd(pob)

# Sobre esta poblacion vamos a extraer una muestra de n elementos
muestra <- sample(pob, n)

# Si el bootstrap funciona vamos a extraer submuestras de muestra, y la distribucion de estas submuestras

# primero hay que definir dos parametros
nBoot = 10000 # veces que repetimos el submuestreo
nsub = 25      # tamaño de las submuestras

res <- vector(length = nBoot)

```

```

for(i in 1:nBoot){
  x <- sample(muestra, nsub, replace=T) # submuestreos
  res[i] <- mean(x) # media de la submuestra
}

sprintf("media poblacion = %.4f, media bootstrap = %.4f", m.pob, mean(res))

## [1] "media poblacion = 4.9136, media bootstrap = 4.9893"
sprintf("sd/sqrt(n) = %.4f, sd bootstrap = %.4f", sd.pob/sqrt(nsub), sd(res))

## [1] "sd/sqrt(n) = 0.3906, sd bootstrap = 0.3959"
# Calculamos el IC por bootstrap al 95% usando percentiles
sprintf("Linf = %.4f, Estimate = %.4f, Lsup = %.4f", quantile(res, 0.025), quantile(res, 0.5), quantile(res, 0.975))

## [1] "Linf = 4.2145, Estimate = 4.9878, Lsup = 4.4794"
# Calculamos el IC por bootstrap al 95% usando el error típico
sprintf("Linf = %.4f, Estimate = %.4f, Lsup = %.4f", mean(res)-1.96*sd(res), mean(res), mean(res)+1.96*sd(res))

## [1] "Linf = 4.2133, Estimate = 4.9893, Lsup = 5.7653"

```

Como se puede ver los valores obtenidos por bootstrap para el estadístico media son muy parecidos a los teóricos que tenemos con la población, y es que se puede demostrar que bootstrap es una buena estimación de las distribuciones poblacionales de los parámetros. Fíjate que hemos establecido nsub en 25 (el tamaño de las submuestras), lo que parece un valor razonable. Este nsub no se puede confundir con n, el tamaño de la muestra que en nuestro caso ha sido n=100. Hay mucha discusión teórica sobre cuánto debe ser nsub para una correcta estimación pero excede los objetivos de esta lección. En el ejemplo anterior hemos utilizado para calcular el intervalo de confianza tanto los *quantiles* (0.025, 0.50 y 0.975) como el error típico; como la discusión de cuándo sería oportuno utilizar una u otra aproximación también excede los contenidos del curso las utilizaremos indistintamente.

2.1 Tarea 1

Construye una versión paralelizada del bootstrap anterior

2.2 Tarea 2

Lleva a cabo un boostrap paralelizado para estimar los intervalos de confianza al 95% para el estadístico varianza.

2.3 Tarea 3

Estudia la relación (ratio) entre el tamaño de la muestra y de las submuestras en la precisión de la estimación por bootstrap con el estadístico media. Usa una versión paralelizada.

La utilidad más común de bootstrap es la de estimar los errores de los coeficientes en los modelos cuando estos no se pueden estimar de forma convencional. Un ejemplo un poco más complejo que el anterior: supongamos que tengo unos datos y con ellos calculo un modelo lineal del tipo $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$ estimado por el procedimiento *lm()* y me gustaría conocer el intervalo de confianza al 95% para cada uno de los tres coeficientes del modelo (β_0, β_1 y β_2). Obviamente al usarse el método de mínimos cuadrados podemos desde el inicio conocer tanto los coeficientes como sus intervalos de confianza, pero vamos a calcularlos usando bootstrap.

```

# Creamos unos datos aleatorios para una regresion con los coeficientes: 3, 5 y 2 respectivamente, mas un error e

set.seed(1)          # por replicabilidad
x1 = rnorm(100, 5, 1)
x2 = rnorm(100, 25, 5)
e = rnorm(100, 0, 3)
y = 3 + 5 * x1 + 2 * x2 + e

datos <- data.frame(y=y, x1 = x1, x2 = x2)

# ajustamos el modelo lm para nuestros datos
mod= lm(y~x1+x2, datos)
mod$coefficients

## (Intercept)      x1      x2
## 3.56141      5.06333     1.96792
# los coeficientes obtenidos son: 3.56, 5.06 y 1.96

# Vamos a encontrar la distribucion de esos coeficientes por bootstrap

nBoot = 10000 # veces que repetimos el submuestreo
nsub = 25      # tamaño de las submuestras

B=array(0,dim=c(nBoot, 3)) # porque son 3 coeficientes a guardar B0, B1 y B2

# hacemos un bootstrap de los coeficientes estimados por lm para cada una de las submuestras

for(i in 1:nBoot){
  df <- datos[sample(1:nrow(datos), nsub, replace=T),] # submuestreo de las filas
  fit <- lm(y~x1+x2, df)          # ajuste del modelo para esa submuestra
  B[i,] <- fit$coefficients      # guardamos los coeficiente
}

# Los valores obtenidos por bootstrap son:

sprintf("B0: mean = %.4f, sd = %.4f", mean(B[,1]), sd(B[,1]))

## [1] "B0: mean = 3.9864, sd = 4.7192"
sprintf("B1: mean = %.4f, sd = %.4f", mean(B[,2]), sd(B[,2]))

## [1] "B1: mean = 5.0025, sd = 0.7617"
sprintf("B2: mean = %.4f, sd = %.4f", mean(B[,3]), sd(B[,3]))

## [1] "B2: mean = 1.9624, sd = 0.1328"
# comparados con los obtenidos por minimos cuadrados
summary(mod)

##
## Call:
## lm(formula = y ~ x1 + x2, data = datos)
##
## Residuals:
##    Min     1Q Median     3Q    Max 
## -10.5   -3.5    0.0    3.5   10.5 

```

```

## -8.8308 -1.3094  0.0061  1.9107  7.9182
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 3.56141   2.44171   1.459   0.148
## x1          5.06333   0.35026  14.456 <2e-16 ***
## x2          1.96792   0.06569  29.959 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.13 on 97 degrees of freedom
## Multiple R-squared:  0.9193, Adjusted R-squared:  0.9177
## F-statistic: 552.8 on 2 and 97 DF,  p-value: < 2.2e-16

```

Los valores obtenidos por bootstrap son razonablemente parecidos a los obtenidos por mínimos cuadrados y bastante similares a los originales que generaron los datos. Obviamente la utilidad de bootstrap no está en estimar lo que ya se sabe, sino en ofrecer una estimación en aquellos procedimientos en los que no es posible conocer la forma de la distribución de los parámetros.

2.4 Tarea 4

En la semana anterior calculamos un modelo estacionario basado en los componentes numérico, verbal y perceptivo. Usando un método de fuerza bruta encontramos la combinación de valores β_{COG} y β_{fp} que hacía mínimo el error cuadrático.

1. Calcula por bootstrap los intervalos de confianza al 95% para β_{COG} y β_{fp} .
2. Crea una versión no paralelizada de tu función de bootstrap y otra sí paralelizada y compara los tiempos de ejecución.