Programación Paralela con R y Rstudio.
Una Introducción

Instructor Rina Surós

Septiembre 2022

Facultad de Ingeniería. UDELAR

Montevideo

Ejemplo: generación de Normales Estándar

Este ejemplo sencillo nos muestra cargar la información a los nodos de procesamiento. Usaremos

- detectCores(logical = FALSE). Para detecta los nodos físicos de nuestra máquina
- makeCluster(ncores). Crear un cluster (**cl**) de n nodos
- clusterApply(cl, x = 1:ncores, fun = rnorm) paraleliza los cálculos sobre el cluster **cl** que indiquemos, el vector **x** con los índices que se usarán y carga la función "**fun**" que le indiquemos cada nodo del cluster.
- Usaremos la función rnorm(x,a,b) para generar la cantidad indicada por el parámetro **x** de valores aleatorios con distribución normal.

  En nuestro caso ncores = 4, entonces hacemos variar **x** entre 1 y 4. Aunque tenemos 8 nodos de procesamiento solo usaremos 4.

  El trabajo de generar cada muestra de normales se reparte entre los nodos. Cada nodo genera una muestra del tamaño indicado.

Veamos el script clase_01.R las dos primeras etapas

- **Ejemplo 1**: detectar nodos de cálculo
- **Ejemplo 2**: generar normales No esténdar

Ejemplo 3: Leer datos de un archivo y hacer cálculos con ellos

Los datos contenidos en el archivo train_data.csv, corresponden a variables relacionadas con el uso de tarjeta de credito por clientes y fechas, comprenden un total de 190 variables y más de 5millones de filas. El archivo tiene un tamaño de 15,2 Gb.

Es imposible leer 5 millones de filas de datos con mi computadores, se come la memoria RAM rápidamente.

Los datos contenidos en el archivo **train_data.csv** corresponden a variables relacionadas con el uso de tarjeta de credito por clientes y fechas, comprenden un total de 190 variables y más de 5 millones de filas. El archivo tiene un tamaño de 15,2 Gb.

- No conocemos los nombres (datos anonimizados)
- No todos son numéricos

Se usa en este ejemplo un extracto seleccionado de solo 10 mil filas del total contenido en el archivo **minitrain.csv**.

Procedamos a leer datos y cargarlos en los nodos. Como el archivo **train_data.csv** es muy grande, tenemos que leerlo **por trozos,** usando:

leer_trozo(archivo,parte,longitud)

La lectura genera un mensaje de error ya que no consigue esa función.

Error in checkForRemoteErrors(val) :
4 nodes produced errors; first error: could not find function "leer_trozo”

Es porque no le hemos cargado a los nodos esa función.Procedemos a cargarla

Con clusterExport(cl,c('leer_trozo’)) enviamos la función al cluster

- Ejecutamos de nuevo y aparece un nuevo mensaje de error:

  Error in checkForRemoteErrors(val) :
  4 nodes produced errors; first error: could not find function "read_csv”

- No consigue ahora la función read_csv porque no la hemos cargado en los nodos.

Con clusterExport(cl,c('leer_trozo','read_csv’)) procedemos a exportar la función read_csv a los nodos. Con esto leemos los datos en paralelo, cada nodo lee los trozos que le corresponden en paralelo con el resto de los nodos y hace los respectivos cálculos. Por simplicidad solo sumamos y multiplicamos dos columnas y medimos los tiempos.

- Construimos la función que será ejecutada

  function(archivo,parte,longitud)

Una vez hechos estos cambios logramos ejecutar el código. Veamos la ejecución en la plataforma

Ejemplo 4: Análisis de desempeño de lectura en secuencial vs. paralelo

Leeremos por trozos desde el archivo train_data.csv, **3 millones de datos con 190** columnas de forma:

- Lectura secuencial
- Lectura en paralelo

Comparamos el desempeño.

Intentamos leer directamente los 3 millones de datos no fue posible para ninguno de los equipos que teníamos a disposición. No lo mostraré, tarda en bloquearse.

Los trozos serán, en ambos casos, de longitud 100.000 por lo que se leerán 30 trozos para completar los 30 millones de datos.

Se usará el comando **Sys.time** al inicio y al final dela tarea para medir cuanto tiempo consume cada metodología.

Resultados

Comparación

```
tiempos = c(tf,vf)
names(tiempos) = c('Secuencial','Paralelo')

cociente = as.numeric(tf)/as.numeric(vf)

tiempos
```

```
## Time differences in mins
## Secuencial   Paralelo
##   4.901903   1.996182
```

```
cociente
```

```
## [1] 2.455639
```

Datos de una corrida anterior

Muestra que el paralelo tardó menos de la mitad del tiempo para todas las lecturas. Es bastante más rápido

La aceleración es de 2.4 veces más rápido

En los anexos tienen un pdf con esta explicación detallada: clase_01_mark.pdf y también el clase_01_mark

Código_02.R

Hagamos códigos más complejos con el mismo ejemplo de leer por trozos. Esto es porque más adelante calcularemos una covarianza, preparamos así los pasos preliminares.

En el código anterior hicimos cálculos elementales con 2 columnas. Ahora necesitamos hacer cuentas con cualquier par de columnas de la matriz.

Calcularemos la **media de cada columna** que es la suma de sus valores dividido entre el número de sumandos (valores en la columna)

- Son dos tareas simples: Leemos por trozos la matriz (por fila), a la vez que contabilizamos cuantos datos hay y los vamos acumulando (sumando). Con esto podremos hacer el promedio de cada columna.
- Las columnas son desiguales, algunas tienen muy pocos datos y muchos NA. Debo hacer los cálculos únicamente con datos numéricos.
- **La lectura se hace por paquetes de filas, pero a mi me interesan las columnas. Entonces cada vez que leo un paquete de fila hago la suma de un nuevo dato por columna y sumo el número de datos en ese paquete.**

Veamos el script código_02.R sobre la plataforma

Observación:

- En secuencial uso un for
- En paralelo uso clusterApply()

Escalabilidad

Es la evolución de la aceleración (speedup) de un código a medida que aumentamos el número de de nodos de procesamientos para ejecutarlo

Vimos que una vez medido el

T_1 = Tiempo con **1** nodo. Lo llamamos tiempo secuencial

T_p = Tiempo con **p** nodos. Lo llamamos tiempo paralelo

$A_p = T_1 / T_{p.}$ Es la aceleración que se experimenta el código secuencial al reprogramarse como código paralelo usando **p** nodos de cálculo

Ya vimos que con estos parámetros podemos estudiar el rendimiento del código.

La **Ley de Amdahl** formaliza el concepto de escalabilidad, demostrando que el paralelismo siempre tiene un límite. La aceleración tiene un límite

Limitantes de la Paralelización

Lay de Amdahl

Ningún código es 100 % paralelizable

Consideraciones de la Ley de Amdahl

Consideremos un sistema con **n** nodos de procesamiento. Sean

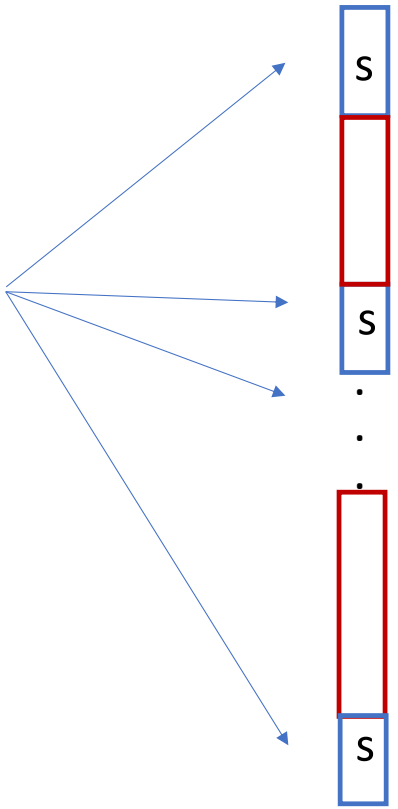
n: numero de nodos a usar
k: fracción de tiempo de código que **es** paralelizable
s: fracción de tiempo del código que es secuencial (no paralelizable)

s es la parte del código inherentemente secuencial. Es decir, s no puede ser paralelizado aunque tengamos procesadores disponibles

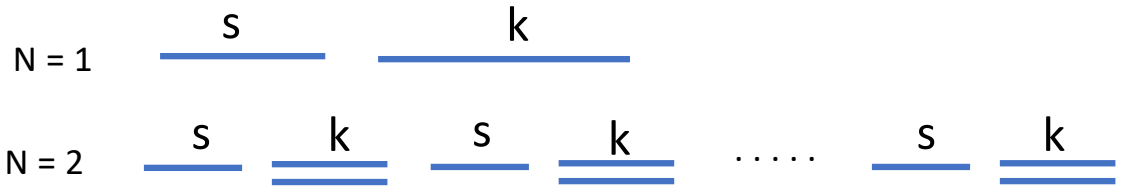
Consideraciones de la Ley de Amdahl

La ejecución de un código paralelo utilizando **n** nodos, tiene las siguientes características:

- El código contiene partes inherentemente secuenciales (**s**) que frenan la posibilidad de acelerar el proceso
- Las partes paralelizables son las que pueden ser ejecutadas con n > 1 nodos de procesamiento, reduciendo el tiempo de ejecución. Veamos un gráfico de ejecución en paralelo para 1 y 2 nodos de procesamiento



n = 1 y n = 2



$$T_T = \Sigma T_{1j} + \Sigma T_{ni}$$

$$T(n) = T_{\text{parte secuencial}} + (T_{\text{parte paralela}} / \text{Número de procesadores}) \qquad (1)$$

Supongamos, sin pérdida de generalidad que

para $n = 1$ el $T(1) = 1$ una unidad de tiempo

$$T(1) = T_{\text{parte secuencial}} + T_{\text{parte paralela}} / 1 = T_{\text{parte secuencial}} + T_{\text{parte paralela}} = s + k$$

$$T(1) = s + k$$

- Para n procesadores en la fórmula 1 se obtiene

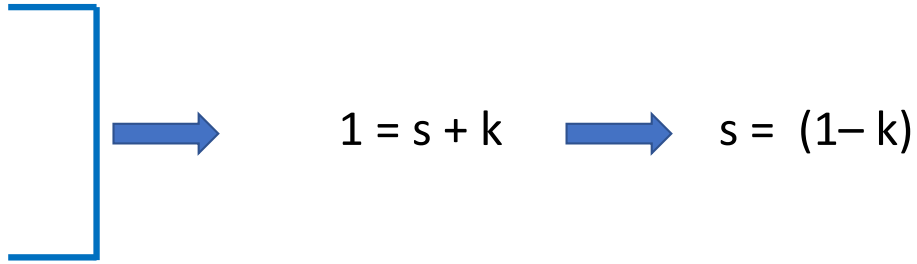
$$T(n) = T_{\text{parte secuencial}} + (T_{\text{parte paralela}}) / n$$

Entonces tenemos que

$T(1) = 1$

y

$T(1) = s + k$



$1 = s + k$ → $s = (1 - k)$

Entonces, re-escribamos T(n) como.

$$T(n) = (1-k) + k/n$$

Parte inherentemente secuencial

Parte paralelizable

Sabemos que

$$A_p = T_1 / T_n.$$

Tenemos que

$T(n) = (1-k) + k/n$

$A(n) = T(1) / T(n). = 1 / (1 - k) + k/n$

$T(1) = 1$

$$\text{Aceleración teórica} = 1 / (1 - k) + k/n$$

Teóricamente el tiempo total del código paralelo es el tiempo de la parte secuencial T_1 más el tiempo que tarda en ejecutarse la parte paralelizable

$$T_t = (1-k) + k/A_n$$

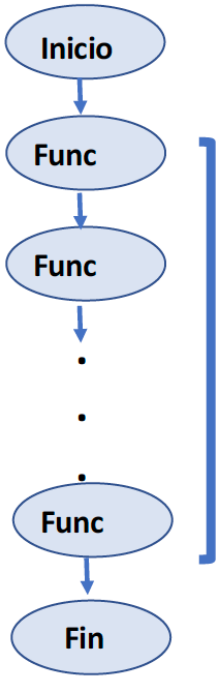
- Aunque tenga infinitos nodos de procesamiento solo podré acelerar la parte del código que es paralelizable

El paralelismo tiene un límite que viene dado por

- Aquella parte que es inherentemente secuencial y por lo tanto no podemos acelerarlo
- El límite de la aceleración del comportamiento del tiempo en la parte paralela

Hilos de Ejecución

Hilos de Ejecución: *thread*



Definimos Hilo de Ejecución (thread control block) como la unidad básica de utilización del CPU. Tiene un ID y un conjunto de registros (código).

En un código secuencial hay un solo hilo de ejecución y un solo procesador, entonces el hilo es la secuencia de instrucciones que el procesador recibe para su ejecución.

El hilo de ejecución controla el flujo dentro de un programa utilizando un **calendario.**

Permite continuar la ejecución mientras se hacen conexiones a memoria, a red o a disco. No se detiene el programa cuando se hacen los procesos de comunicación. Se llama **Concurrencia.**

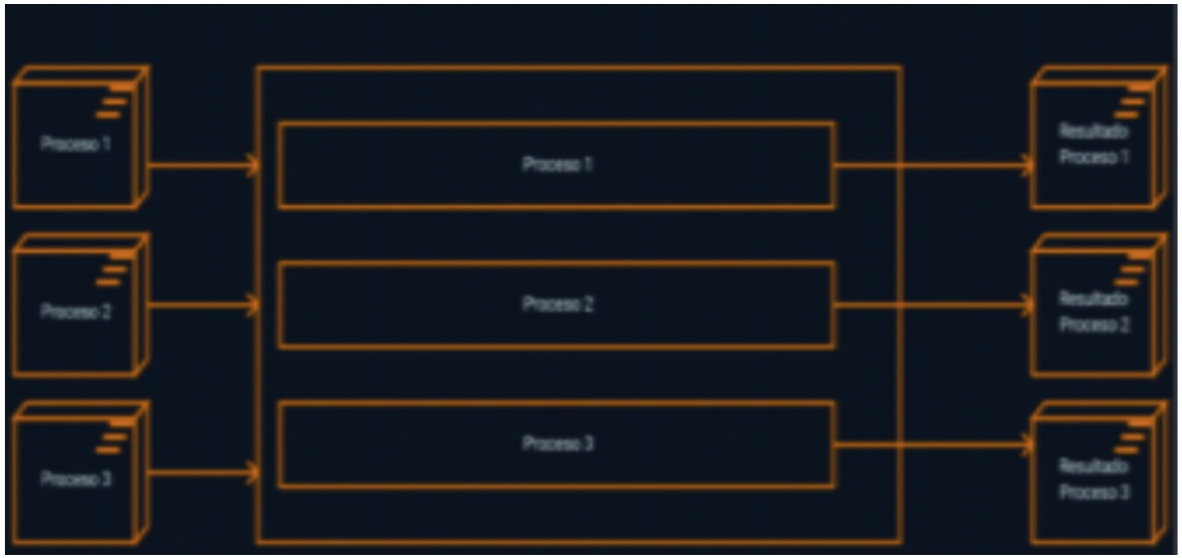
Hilos de Ejecución Concurrente

Multiprogramación: Cuando el código de ejecuta en un único CPU

La **Concurrencia s**oporta dos o más acciones en progreso. Los procesos que se ejecutan de manera independiente. Suceden muchas cosas al tiempo

Hilos de Ejecución Paralela

Paralelismo: Cuando el código de ejecuta en múltiples CPU
El Paralelismo **s**oporta múltiples acciones en progreso. Los procesos que se ejecutan de manera independiente sobre diferentes procesadores y en cada procesador hay concurrencia. Suceden muchas cosas al tiempo y en paralelo



Hilos de Ejecución
Concurrencia vs Paralelismo

- La concurrencia es inherente al programa
- El paralelismo es inherente a la máquina
- ClusterApply automatiza la creación de hilos de ejecución

  makeCluster(n.cores)
- También podemos planificar nuestros hilos con herramientas como:

  Rdsm: Threads Environment for R.

Hilos de Ejecución: *thread*

- Un proceso es un hilo de longitud 1.

plan("sequential")

- Un hilo de ejecución (Thread), es una secuencia de dos o mas procesos cuya ejecución se planifica a futuro.

plan("multicore")

- La secuencia de múltiples procesos están apilados esperando por un recurso disponible.

Hilos de Ejecución: *thread*

Según la capacidad de los recursos, se pueden crear múltiples tipos de hilo.

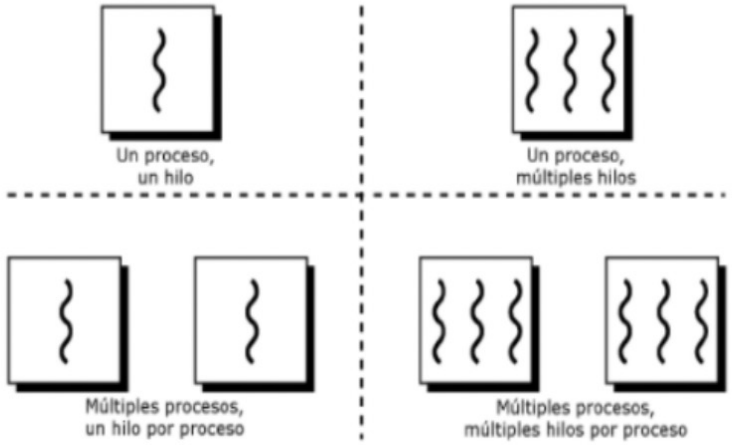
La creación de hilos mejora la calidad del código

En R existen comando como

 # Creacion del cluster

cl <- makeCluster(n.cores)

Que nos permiten crear hilos



Carga Balanceda

Carga Balanceada

La forma de asignar el trabajo a los hilos puede generar que se asigne más trabajo a unos que a otros, haciendo que algunos hilos queden improductivos mientras aún queda trabajo por hacer.

Ejemplo: una matriz con mucho ceros y una operación matricial algebraica.

En una operación por filas, si distribuimos esas filas entre procesadores distintos, las operaciones de las filas con menor cantidad de posiciones cero serán más pesadas que aquellas que tienen muchos ceros.

La librería “parallel” hace automáticamente el balance de las cargas con

clusterApplyLB(cl,x, …..,a,b)

Carga Balanceada

Bench: herramientas para comparar y analizar con precisión los tiempos de ejecución del sistema

Y usaremos la función

```
clusterApplyLB(cl,x, …..,a,b)
```

Que hace énfasis en balancear las cargas en los procesadores

La familia **clusterXXXX** es muy extensa. Proveen diversas formas de paralelizar usando un cluster. Algunas son muy especificas, se puede indicar la de conexión remota a un servidor determinado. Otras funcionan solo para unix o Windows, etc.

- clusterCall(cl = NULL, fun, ...)
- clusterApply(cl = NULL, x, fun, ...)
- clusterApplyLB(cl = NULL, x, fun, ...)
- clusterEvalQ(cl = NULL, expr)
- clusterExport(cl = NULL, varlist, envir = .GlobalEnv)
- clusterMap(cl = NULL, fun, ..., MoreArgs = NULL, RECYCLE = TRUE, SIMPLIFY = FALSE, USE.NAMES = TRUE, .scheduling = c("static", "dynamic"))
- clusterSplit(cl = NULL, seq)
- parLapply(cl = NULL, X, fun, ..., chunk.size = NULL)

- parSapply(cl = NULL, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE, chunk.size = NULL)
- parApply(cl = NULL, X, MARGIN, FUN, ..., chunk.size = NULL) parRapply(cl = NULL, x, FUN, ..., chunk.size = NULL)
- parCapply(cl = NULL, x, FUN, ..., chunk.size = NULL) parLapplyLB(cl = NULL, X, fun, ..., chunk.size = NULL)
- parSapplyLB(cl = NULL, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE, chunk.size = NULL)

Vamos al script