Programming with Big Data in R: computación paralela con datos distribuidos

Carlos Pérez-González - Área de Estadística e Investigación Operativa (Universidad de La Laguna)

Jornadas "Desmitificando BigData" - ETSII, Junio, 2014

Sobre esta presentación

Download

La presentación es una versión simplificada de la que se encuentra disponible en el proyecto pdbR:

https://github.com/wrathematics/pbd-tutorial/blob/master/pbdr/useR/pbdr.pdf?raw=true

Speaking Serial R with a Parallel Accent

Los ejemplos presentados se pueden ver con más detalle en el manual **pbdDEMO** titulado *Speaking Serial R with a Parallel Accent*

http://goo.gl/HZkRt

La instalación de pbdR

Instrucciones

Las instrucciones para configurar un entorno pbdR están disponibles en:

http://r-pbd.org/install.html

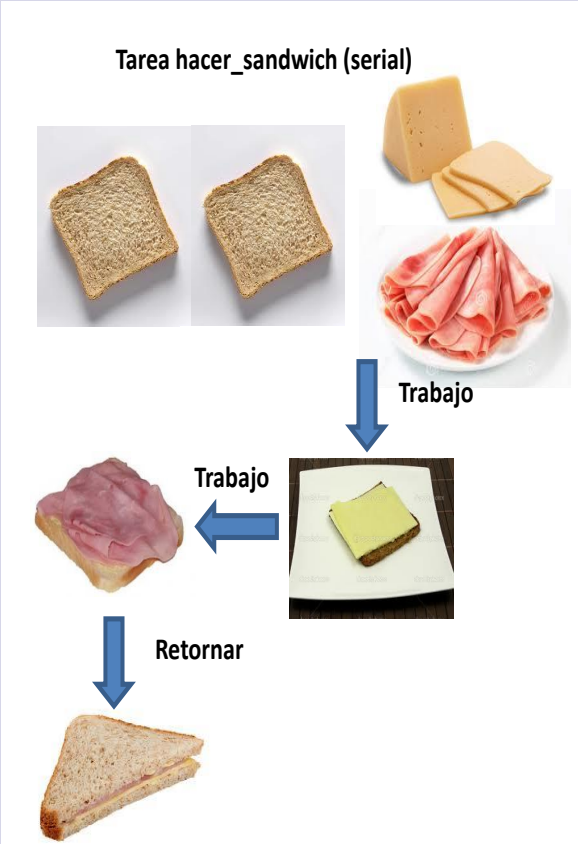
Aquí se explica como instalar R, MPI, y pbdR.

Contenidos

Paralelismo

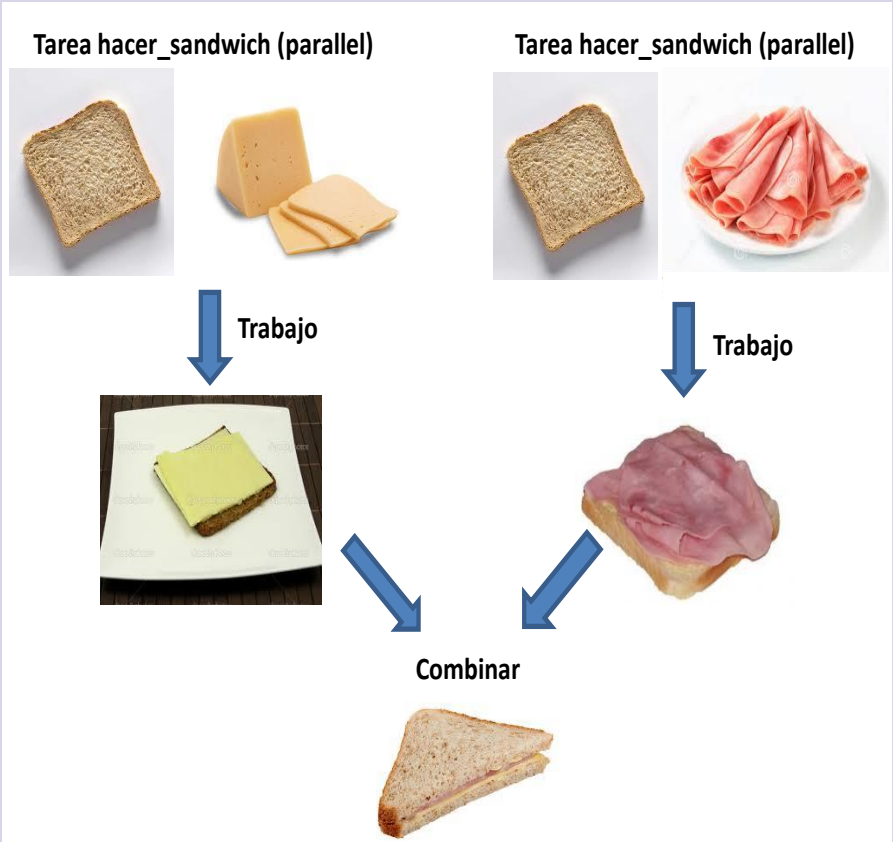
Programación en serie o secuencial

hacer_sandwich_serial.sh

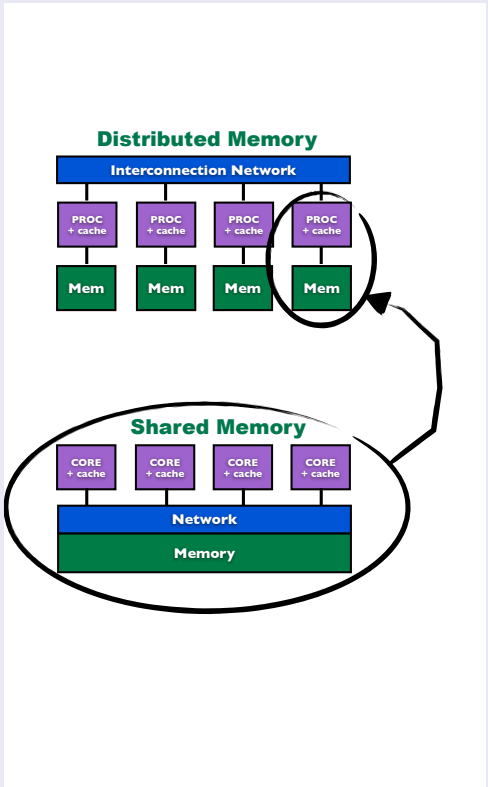


Programación en paralelo

mpirun -np 2 hacer_sandwich_parallel.sh



Sistema distribuido



- En el sistema de memoria compartida se trata de saber qué tareas pueden ser paralelizadas (OpenMP, multihilo). Ver paquete multicore de R.
- En el sistema distribuido el enfoque consiste en saber como asignar los datos a los nodos y cómo funciona la comunicación entre ellos (MPI, Hadoop). Utilizamos paquete pbdMPI de R.
- R proporciona una interfaz a las librerías de cálculo algebraico BLAS/LAPACK en un entorno HPC (computación de altas prestaciones),

Programando con BigData en R (pbdR)

pbdR
Programming with Big Data in R

- Es un conjunto de paquetes de R (libre).
- Permite enlazar el código compilado de alto rendimiento con la alta productividad de scripts en R
- Proporciona una solución analítica con big-data.
- Los métodos tiene una sintaxis muy similar a la de R.

El paradigma pbdR

Los programas **pbdR** son programa R

La diferencia es que:

- Se ejecutan en proceso de lotes o batch mode (la ejecución es no-interactiva).
- La codificación en paralelo utiliza el estilo Un progama/Múltiples datos (Single Program/Multiple Data , SPMD).
- Se centra en la paralelización de los datos (distribución de datos en los nodos).

Ejecución en lotes

- Ejecutar un programa secuencial R en batch:

```
1 Rscript my_serial_script.r
```

o, también,

```
1 R CMD BATCH my_serial_script.r
```

- Ejecutar un programa paralelo en R (con MPI) en batch:

```
1 mpirun -np 2 Rscript my_parallel_script.r
```

Características de Single Program/Multiple Data (SPMD)

- Sólo se escribe un programa, pero se ejecuta en lotes en todos los procesadores.
- Los diferentes procesadores son autónomos; no hay manager.
- Es un modelo de programación bastante extendido para grandes máquinas.

Interfaz de paso de mensajes (Message Passing Interface, MPI)

- La librería *MPI* representa el estándar utilizado en las comunicaciones (de datos e instrucciones) entre distintos nodos/computadores.
- *Implementaciones*: OpenMPI, MPICH2, ...
- Permite el paralelismo (a través de las comunicaciones) en sistemas distribuidos.
- *Comunicador*: Es el canal que permite gestionar las comunicaciones entre procesadores.

Operaciones MPI (1 de 2)

- **Crear un comunicador**: Crear y cerrar canales de comunicación.
  `init()` — inicializa el comunicador
  `finalize()` — cierra el comunicador

- **Rango de canal**: Se puede determinar la posición o rango de un procesador en el canal.
  `comm.rank()` — "¿Qué procesador soy?"
  `comm.size()` — "¿Cuántos procesadores habemos en el canal?"

- **Visualización**: Ver el output en diferentes posiciones del canal.
  `comm.print(x)`
  `comm.cat(x)`

Operaciones MPI (2 de 2)

- **Reducción o reducer**: Por ejemplo, cada procesador tiene un número `x`; los reunimos todos y aplicamos una operación aritmética (suma, resta, max, min, ...).
  `reduce(x, op='sum')` — reduce a un procesador
  `allreduce(x, op='sum')` — reduce en todos los procesadores

- **Recoger o gather**: Por ejemplo, cada procesador tiene un número; creamos un objeto (array) en algún procesador que los contenga a todos.
  `gather(x)` — recoger en uno
  `allgather(x)` — recoger en todos

- **Transmitir o broadcast**: Por ejemplo, un procesador tiene un número `x` y queremos enviarlo a los demás.
  `bcast(x)`

- **Barrera**: Esta es una instrucción que actúa a modo de "computation wall" de forma que ningún procesador puede continuar hasta que todos los demás lo puedan hacer.
  `barrier()`

Ejemplo sencillo 1

Gather: test_gather_parallel.r

```
1  library(pbdMPI, quietly=TRUE)
2  init()
3  comm.set.seed(diff=TRUE)
4
5  n <- sample(1:100, size=10)  # simular una muestra en cada nodo
6
7  comm.print(n, all.rank =T)   # print la muestra de cada nodo
8
9  gt <- gather(n, rank.dest=1) # recoger todas las muestras en el nodo 1
10 comm.print(gt, all.rank =T)
11
12 finalize()
```

mpirun -np 2 Rscript test_gather_parallel.r:

```
1  [1] "# simular una muestra en cada nodo"
2  COMM.RANK = 0
3    [1] 28 52 18 77 83 13 65 43 71 36
4  COMM.RANK = 1
5    [1] 60 72 38  3 59 42 50 58 34 57
6  [1] "# reunir las muestras de todos los nodos"
7  COMM.RANK = 0
8  NULL
9  COMM.RANK = 1
10   [1] 28 52 18 77 83 13 65 43 71 36 60 72 38  3 59 42 50 58 34 57
```

Ejemplo sencillo 2

Reducción: test_reduce_parallel.r

```
1  library(pbdMPI, quietly=TRUE)
2  init()
3  comm.set.seed(diff=TRUE)
4
5  n <- sample(1:100, size=10)  # simular una muestra en cada nodo
6
7  comm.print(n, all.rank =T)   # print la muestra de cada nodo
8
9  sm <- allreduce(n, op="sum") # sumar los elementos corresp. de cada muestra
10 comm.print(sm, all.rank =T)
11
12 finalize()
```

mpirun -np 2 Rscript test_reduce_parallel.r:

```
1  [1] "# simular una muestra en cada nodo"
2  COMM.RANK = 0
3    [1] 28 52 18 77 83 13 65 43 71 36
4  COMM.RANK = 1
5    [1] 60 72 38  3 59 42 50 58 34 57
6  [1] "# sumar los elementos corresp. de cada
       muestra"
7  COMM.RANK = 0
8    [1]  88 124  56  80 142  55 115 101 105  93
9  COMM.RANK = 1
10   [1]  88 124  56  80 142  55 115 101 105  93
```

Ejemplo 1 : Paralelizar varias simulaciones

Código secuencial (Rscript test_simula_serial.r)

```
# simular una muestra en cada nodo
n1 <- sample(1:1e7 , size =10)
n2 <- sample(1:1e7 , size =10)
n3 <- sample(1:1e7 , size =10)
n4 <- sample(1:1e7 , size =10)

print(n1)
print(n2)
print(n3)
print(n4)

# sumar los elementos corresp. de cada muestra
sm <- n1+n2+n3+n4
print(sm)
```

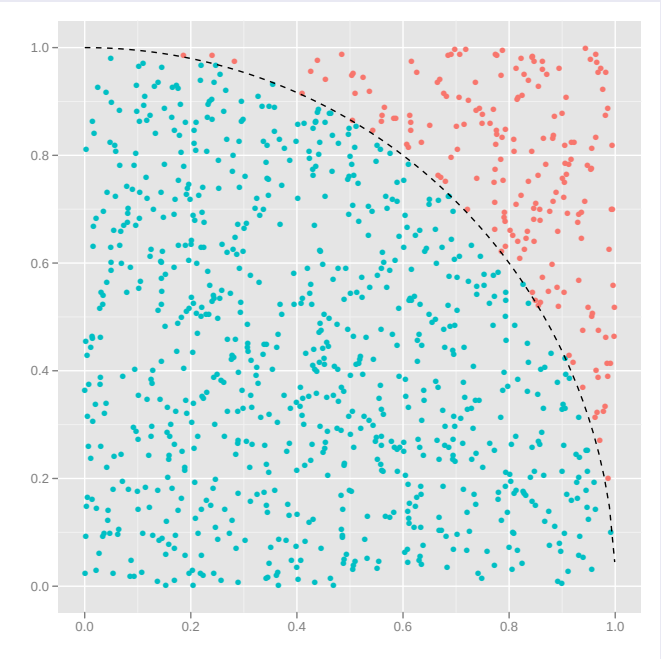
Código paralelo (mpirun -np 4 test_simula_parallel.r)

```
library(pbdMPI, quiet = TRUE)
init()
comm.set.seed(diff=TRUE)

# simular una muestra en cada nodo
n <- sample(1:1e7 , size =10)
comm.print(n, all.rank =T)

# sumar los elementos corresp. de cada muestra
sm <- allreduce(n, op="sum")
comm.print(sm, all.rank =F)

finalize()
```

Ejemplo 2 : Simulación distribuida (problema)

Supongamos que obtenemos una muestra de N observaciones uniformes (x_i, y_i) en el cuadrado unidad $[0,1] \times [0,1]$. Entonces

$$\pi \approx 4 \left(\frac{\#\text{ Dentro círculo}}{\#\text{ Total}} \right) = 4 \left(\frac{\#\text{ Azul}}{\#\text{ Azul} + \#\text{ Rojo}} \right)$$



Ejemplo 2 : Algoritmo de la simulación distribuida (GBD)

1. Sea N el tamaño de la simulación; tomaremos $N = 50,000,000$.
2. Se genera una $N \times 2$ matriz $\mathbf{X} = (\mathbf{x}, \mathbf{y})$ de observaciones uniformes $U(0, 1)$.
3. Se divide la matrix completa $\mathbf{X}$ en 4 sub-conjuntos de filas (matrices locales) que pasamos a cada uno de los nodos.
4. Contar (localmente) el número de filas que satisfacen $x^2 + y^2 \leq 1$
5. Preguntar a todos los nodos cuál es su respuesta; sumarlas todas (reducción).
6. Tomamos este valor y lo multiplicamos por $4/N$
7. Que el procesador en el rango 0 haga print del output.

Ejemplo 2 : Simulación distribuida (código)

Código secuencial (Rscript test_pi_serial.r)

```
1  N <- 50e6
2  X <- matrix(runif(N * 2), ncol=2)
3  r <- sum(rowSums(X^2) <= 1)
4  PI <- 4*r/N print(PI)
```

Código paralelo (mpirun -np 4 test_pi_parallel.r)

```
1  library(pbdMPI, quiet = TRUE) init()
       comm.set.seed(diff=TRUE)
2
3  N.gbd <- 50e6 / comm.size()
4  X.gbd <- matrix(runif(N.gbd * 2), ncol = 2)
5  r.gbd <- sum(rowSums(X.gbd^2) <= 1)
6  r <- allreduce(r.gbd, op='sum')
7  PI <- 4*r/(N.gbd * comm.size())
8  comm.print(PI)
9
10 finalize()
```

Ejemplo 3 : Covarianza muestral

$$cov(\boldsymbol{X}_{N\times p}) = \frac{1}{N-1}(\boldsymbol{X} - \mathbf{1}\mu_{\boldsymbol{x}})^t(\boldsymbol{X} - \mathbf{1}\mu_{\boldsymbol{x}}) = \frac{1}{N-1}\sum_{i=1}^{N}(\boldsymbol{x}_i - \mu_{\boldsymbol{x}})(\boldsymbol{x}_i - \mu_{\boldsymbol{x}})^T$$

Ejemplo 3 : Covarianza muestral (problema)

1. Determinar el número total de filas N.
2. Hallamos el vector de medias de las columnas de la matriz completa.
3. Restamos dicho vector a cada columna de la matriz local (en cada nodo).
4. Hallamos el producto cruzado localmente y reducimos.
5. Se divide por $N - 1$.

Ejemplo 3 : Algoritmo de la covarianza muestral (GBD)

Código secuencial (Rscript test_covariance_serial.r)

```
N <- 50e6

library(MASS)
sigma <- matrix(c(3,1,1,3),2,2)
X<- mvrnorm(n=N, rep(0, 2), sigma)

N <- nrow(X)
mu <- colSums(X)/N
X <- sweep(X, STATS=mu , MARGIN=2)
Cov.X <- crossprod(X)/(N -1)
print(Cov.X)
```

Código paralelo (mpirun -np 4 test_covariance_parallel.r)

```
library (pbdMPI , quiet=TRUE)
init()
comm.set.seed(diff=T)

N.gbd <- 50e6 / comm.size()
library(MASS)
sigma <- matrix(c(3,1,1,3),2,2)
X.gbd<- mvrnorm(n=N.gbd, rep(0, 2), sigma)

N <- allreduce(nrow(X.gbd), op="sum")
mu <- allreduce(colSums(X.gbd) / N, op="sum")
X.gbd <- sweep(X.gbd, STATS=mu, MARGIN=2)
Cov.X <- allreduce(crossprod(X.gbd), op="sum") / (N-1)

comm.print(Cov.X)
finalize()
```

Ejemplo 4 : Regresión lineal

Básicamente, se trata de estimar $\boldsymbol{\beta}$ tal que

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

Cuando $\boldsymbol{X}$ tiene rango completo,

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

Ejemplo 4 : Regresión lineal (problema)

1. Localmente cada nodo se encarga de un vector $\mathbf{x}^T$ formado por varias filas de $\boldsymbol{X}$.
2. Cada nodo se encarga de calcular $\mathbf{A} = \mathbf{x}^T * \mathbf{x}$. Entonces, reducimos sumando las matrices resultantes en cada nodo.
3. Hacemos lo mismo con $\mathbf{B} = \mathbf{x}^T * \boldsymbol{y}$ (cálculo local y reducción en forma de suma).
4. Finalmente, se calcula $\mathbf{A}^{-1} * \mathbf{B}$.

Un ajuste de regresión lineal

Ejemplo 4 : Algoritmo de la regresión lineal (GBD)

Código secuencial (Rscript test_regresion_serial.r)

```
N <- 50e6
p <- 2
# Simulate data
X <- matrix(rnorm(N * p), ncol = p)
beta <- c(1,2)
epsilon <- rnorm(N)
y <- X%*%beta + epsilon

# Estimar beta
tX <- t(X)
A <- tX %*% X
B <- tX %*% y

ols <- solve(A) %*% B
print(ols)
```

Código paralelo (mpirun -np 4 test_regresion_parallel.r)

```
library (pbdMPI , quiet=TRUE)
init()
comm.set.seed(diff=T)

N.gbd <- 50e6 / comm.size()
p <- 2
### Simula
X.gbd <- matrix(rnorm(N.gbd*p), ncol = p)
beta <- c(1,2)
epsilon <- rnorm(N.gbd)
y.gbd <- X.gbd%*%beta + epsilon

# Estimar beta
tX.gbd <- t(X.gbd)
A <- allreduce(tX.gbd %*% X.gbd, op = "sum")
B <- allreduce(tX.gbd %*% y.gbd, op = "sum")

ols <- solve(A) %*% B
comm.print(ols)

finalize()
```