TP Calcul Parallel - Code Rmarkdown

Table des matières

---

1 Objectifs et Importance de la progammation parallèle

De manière générale, l'exécution d'une opération sur un ordinateur se fait suivant le principe du calcul séquentiel qui consiste en l'exécution de l'opération à travers des étapes successives, où chaque étape ne se déclenche que lorsque l'étape précédente est terminée, y compris lorsque les deux étapes sont indépendantes a priori sur une seule ressource. L'implémentation d'un programme sur R obéit par défaut au principe du calcul séquentielle. Ceci étant, à mesure que les opérations effectuées portent sur des jeux de données relativement grands, ce principe de calcul révèle un certain nombre de limites :

— Il est trop coûteux en temps de calcul, en mémoire ;
— Les volumes de données à traiter sont trop importants, trop longs à écrire ;
— Les performances sont moins bonnes que sur des machines plus vieilles,...

Lorsque l'exécution d'un programme sur R s'avère lent aux besoins de ses utilisateurs, son temps d'exécution doit être optimisé. Il existe plusieurs stratégies pour arriver à cette optimisation : il est recommandé utiliser des fonctions déjà optimisées et disponibles publiquement ; exploiter les calculs vectoriels et matriciels, qui sont plus rapides que des boucles en R ; éviter les allocations mémoire inutiles, notamment les objets de taille croissante et les modifications répétées d'éléments dans un data frame. Malgré ces méthodes d'optimisation, la programmation peut toujours être aussi lente. Une solution appropriée sur R pour optimiser le temps d'exécution est alors le calcul en parallèle.

Ainsi, l'objectif du calcul en parallèle est d'effectuer plus rapidement un calcul informatique en exploitant simultanément plusieurs unités de calcul.

2 Principe général

1. Briser un calcul informatique en blocs de calcul indépendants ;
2. Exécuter simultanément (en parallèle) les blocs de calcul sur plusieurs unités de calcul ;
3. Rassembler les résultats et les retourner.

Paralléliser un problèmeconsiste à décomposer ce problème en plusieurs sous problèmes à résoudre simultanément à travers différentes ressources, pour ressortir la solution du problème initial, dans un délai optimal. Ainsi, le principe du **calcul parallèle** est d'effectuer simultanément une même tâche ou exécuter un même programme de manière parallèle. Cela est aussi possible à travers différentes machines connectées par un réseau où chacun d'eux reçoit une tache à exécuter. Sur R, L'utilité du calcul en parallèle réside dans le faite qu'il permet d'effectuer plus rapidement et de manière asynchrone l'exécution de programme sur des bases de données volumineuses en exploitant simultanément plusieurs unités de calcul d'un ordinateur appelées cœurs.

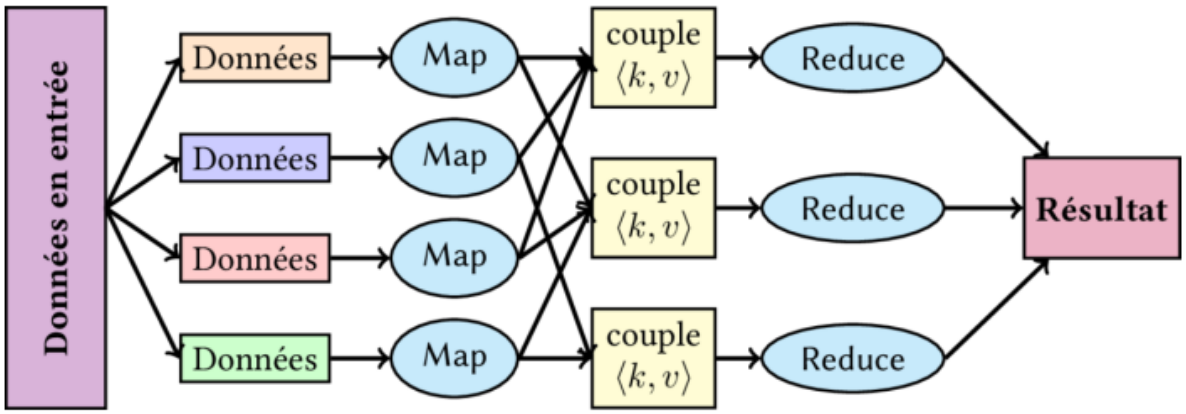


Figure 1 – Modèle de programmation MapReduce

3 Notions de base

1. **Processeur ou CPU :** Son rôle est de lire et d'exécuter les instructionsprovenant d'un programme.
   - Les processeurs sont de nos jours la plupart du temps **divisés en plus d'une unité de calcul**, nommée coeur (en anglais core). Il s'agit alors de **processeurs multi-coeurs**. Ce type de matériel permet de faire du calcul en parallèle sur une seule machine, en exploitant plus d'un coeur de la machine.
   - **Threads :** Les coeurs exécutent ce que l'on appelle des **fils d'exécution** (en anglais **threads**). Un fil d'exécution est une petite séquence d'instructions en langage machine.
   - **Processus monothread** : Lorsque les fils d'exécution sont exécutés séquentiellement par un coeur soit un après l'autre.
   - **Processus multithreads :** Il existe cependant une technologie permettant à un seul coeur physique d'exécuter plus d'un fil d'exécution simultanément. On dit alors que le coeur physique est séparé en **coeurs logiques**. On parle alors d'un coeur multithread.
2. Hadoop : Hadoop est un framework open source largement utilisé pour le traitement et le stockage distribués de données volumineuses (big data). Il fournit une plate-forme pour le traitement parallèle de grandes quantités de données en les répartissant sur un cluster de serveurs.

4 Etapes du calcul parallèle

1. Démarrer m processus "travailleurs" (i.e.cœurs de calcul) et les initialiser ;
2. Envoyer les fonctions et données nécessaires pour chaque tache aux travailleurs ;
3. Séparer les taches en m opérations d'envergure similaire et les envoyer aux travailleurs ;
4. Attendre que tous les travailleurs aient terminer leurs calculs et obtenir leurs résultats ;
5. Rassembler les résultats des différents travailleurs ;
6. Arrêter les processus travailleurs

Le package parallel permet de démarrer et d'arrêter un "cluster" de plusieurs processus travailleur (étape 1). En plus du package `parallel`, on va donc utiliser le package `doParallel` qui permet de gérer les processus travailleurs et la communication (étapes 1) et l'articulation avec le package `foreach`qui permet lui de gérer le dialogue avec les travailleurs (envois, réception et rassemblement des résultats - étapes 2, 3, 4 et 5).

5 Programmation parallèle avec R

- Package **Parallel :** inclus dans la distribution de base de R : Il se base sur l'utilisation de fonctions de la famille des apply.
- Package doParallel et Foreach :
- Le package rmr2 (MapReduce)
- Etc.

6 Quelques fonctions importantes

- Du package Parallel
  - **Detectcores()** : permet de détecter le nombre cœurs de la machine.
  - **Makecluster() :**
  - **Stopcluster() :** est utilisé pour arrêter et libérer les différents workers.
  - La famille des fonctions **Applay,** adaptées au calcul parallèle sous R permet d'exécuter simultanément les opérations sur les différents blocs.
    - parApplay() permet d'effectuer des calculs en parallèle sur une matrice ou un tableau en utilisant un cluster de travailleurs

    - parLapply() permet d'appliquer une fonction à chaque élément d'une liste en utilisant un cluster de travailleurs pour exécuter les calculs en parallèle.
    - parSapplay() permet d'appliquer une fonction de manière parallèle à des éléments d'une liste.Elle prend en argument le jeu de donné et la fonction et retourne un vecteur ou une matrice.
  - clusterEvalQ() : Elle est utilisée pour évaluer une expression sur tous les nœuds d'un cluster parallèle. Elle est utile lorsque vous avez besoin d'exécuter une expression ou de charger des bibliothèques spécifiques sur chaque nœud du cluster avant d'exécuter des tâches parallèles.
- Du package Doparallel et Foreach
  - Foreach : Il constitue une alternative aux fonctions applay utilisé dans le package parallèle. Il fournit une approche simplifiée pour effectuer des boucles parallèles en R, en permettant d'exploiter efficacement les ressources de calcul disponibles sur un système. Il s'appuie généralement sur d'autres packages parallèles, tels que "doParallel" ou "doSNOW", pour exécuter les boucles en parallèle.
  - **%dopar%** est un opérateur spécifique du package "foreach" en R, qui permet d'effectuer des itérations parallèles sur des objets itérables tels que des vecteurs, des listes ou des data frames.
  - registerDoParallel() fait le même que Makecluster
- Package rmr2 (Disponible seulement pour les versions antérieures de R)
  - mapreduce() : La fonction **mapreduce()** est une fonction clé du package R "rmr2" (ou "RHadoop") qui fournit une interface pour exécuter des calculs distribués sur des systèmes de fichiers distribués tels que Hadoop.
  - keyval() : elle est utilisée pour définir des paires clé-valeur qui serviront de données d'entrée pour les opérations de MapReduce. Cette fonction prend deux arguments : une clé et une valeur, et retourne une structure de données représentant une paire clé-valeur.

7 Préliminaire

7.1 garbage collector

```
#vider la mémoire
rm(list=ls())
```

```
#lancer le garbage collector
gc()
```

```
##          used (Mb) gc trigger (Mb) max used (Mb)
## Ncells 520845 27.9    1166272 62.3   643711 34.4
## Vcells 911292  7.0    8388608 64.0  1648775 12.6
```

Le garbage collector permet de gérer automatiquement la mémoire allouée aux objets.

Lorsqu'un programme s'exécute, il alloue de la mémoire pour créer des objets et stocker des données. Cependant, il arrive souvent que certains objets ne soient plus utilisés par le programme, ce qui crée des "déchets" ou des "objets morts" en mémoire. Si ces objets morts ne sont pas libérés, ils peuvent occuper de l'espace précieux en mémoire et entraîner des problèmes tels que des fuites de mémoire.

7.2 Packages

```
#information sur les versions
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19045)
##
## Matrix products: default
```

```
##
## locale:
## [1] LC_COLLATE=French_France.1252  LC_CTYPE=French_France.1252
## [3] LC_MONETARY=French_France.1252 LC_NUMERIC=C
## [5] LC_TIME=French_France.1252
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
## [1] snow_0.4-4        doParallel_1.0.17 iterators_1.0.14  foreach_1.5.2
## [5] tictoc_1.2
##
## loaded via a namespace (and not attached):
##  [1] codetools_0.2-18 digest_0.6.29    lifecycle_1.0.3  magrittr_2.0.3
##  [5] evaluate_0.21    rlang_1.1.0      stringi_1.7.12   cli_3.1.1
##  [9] rstudioapi_0.14  vctrs_0.6.1      rmarkdown_2.22   tools_4.1.2
## [13] stringr_1.5.0    glue_1.6.2       xfun_0.39        yaml_2.2.2
## [17] fastmap_1.1.0    compiler_4.1.2   htmltools_0.5.2  knitr_1.37
```

```
#help(package='parallel')
```

8 Définition de la fonction pour calculer le Min

```
mon_min <- function(v) {
  #copie locale
  temp <- v
  #longueur du vecteur
  n <- length(temp)
  #tri par selection si (n > 1)
  if (n > 1) {
    #recherche des minimums successifs
    for (i in 1:(n - 1)) {
      i_mini <- i
      for (j in (i + 1):n) {
        if (temp[j] < temp[i_mini]) {
          i_mini <- j
        }
      }
      #Echanger
      if (i_mini != i) {
        tempo <- temp[i]
        temp[i] <- temp[i_mini]
        temp[i_mini] <- tempo
      }
    }
  }
  #la plus petite valeur est le min.
  return(temp[1])
}
```

9 Application de la programmation parallèle pour déterminer le Min

```
# Génération d'un vecteur de données
n <- 10
a <- runif(n)
a
```

```
##  [1] 0.1268973 0.2190160 0.4429761 0.1406081 0.5884293 0.9038375 0.4424940
##  [8] 0.9737582 0.7543901 0.4391870
```

9.1 Calcul direct (sans paralleliser)

```
#appel de la fonction sur la totalité du vecteur
tic()
print(paste('Min direct =',mon_min(a)))
```

```
## [1] "Min direct = 0.126897253561765"
```

```
print('>> Temps de calcul - fonction mon_min direct')
```

```
## [1] ">> Temps de calcul - fonction mon_min direct"
```

```
toc()
```

```
## 0.2 sec elapsed
```

9.2 Calcul en utilisant la programmation parallèle

```
#affichage nombre de coeurs dispo
print(parallel::detectCores())
```

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

```
#nombre de blocs des donnees = nombre de cores
k <- 4
#partition en blocs des donn?es
blocs <- split(a,1+(1:n)%%k)
print(blocs)
```

```
## $`1`
## [1] 0.1406081 0.9737582
##
## $`2`
## [1] 0.1268973 0.5884293 0.7543901
##
## $`3`
## [1] 0.2190160 0.9038375 0.4391870
##
## $`4`
## [1] 0.4429761 0.4424940
```

```
#appel de la fonction sur la totalité du vecteur
tic()
print(paste('Min direct =',mon_min(a)))
```

```
## [1] "Min direct = 0.126897253561765"
```

```
print('>> Temps de calcul - fonction mon_min direct')
```

```
## [1] ">> Temps de calcul - fonction mon_min direct"
```

```
toc()
```

```
## 0.02 sec elapsed
```

```
#pour mesurer le processus global de **parallel**
tic()
#Demarrage des moteurs (workers)
clust <- parallel::makeCluster(4)
#lancement des min en parallele
res <- parallel::parSapply(clust,blocs,FUN = mon_min)
#résultats intermédiaires
print(res)
```

```
##         1         2         3         4
## 0.1406081 0.1268973 0.2190160 0.4424940
```

```
#fonction de consolidation
print(paste('Min parallel =',mon_min(res)))
```

```
## [1] "Min parallel = 0.126897253561765"
```

```
#Eteindre les moteurs
parallel::stopCluster(clust)
#affichage temps de calcul
print('>> Temps de calcul total avec parSapply min par bloc')
```

```
## [1] ">> Temps de calcul total avec parSapply min par bloc"
```

```
#temps de calcul
toc()
```

```
## 3.31 sec elapsed
```

9.2.1 Calcul de la moyenne

```
#appel de la fonction sur la totalité du vecteur
tic()
print(paste('Moyenne direct =',mean(a)))
```

```
## [1] "Moyenne direct = 0.503159347549081"
```

```
print('>> Temps de calcul - fonction moyenne direct')
```

```
## [1] ">> Temps de calcul - fonction moyenne direct"
```

```
toc()
```

```
## 0.02 sec elapsed
```

```
#pour mesurer le processus global de **parallel**
tic()
#Demarrage des moteurs (workers)
clust <- parallel::makeCluster(4)
#lancement des min en parallele
res <- parallel::parSapply(clust,blocs,FUN = mean)
poids<-parallel::parSapply(clust,blocs,FUN = length)
```

```r
#résultats intermédiaires
print(res)
```

```
##         1         2         3         4
## 0.5571831 0.4899055 0.5206802 0.4427351
```

```r
#fonction de consolidation
moy<-weighted.mean(res,poids)
print(paste('Moyenne parallel =',moy))
```

```
## [1] "Moyenne parallel = 0.503159347549081"
```

```r
#Eteindre les moteurs
parallel::stopCluster(clust)
#affichage temps de calcul
print('>> Temps de calcul total avec parSapply moy par bloc)')
```

```
## [1] ">> Temps de calcul total avec parSapply moy par bloc)"
```

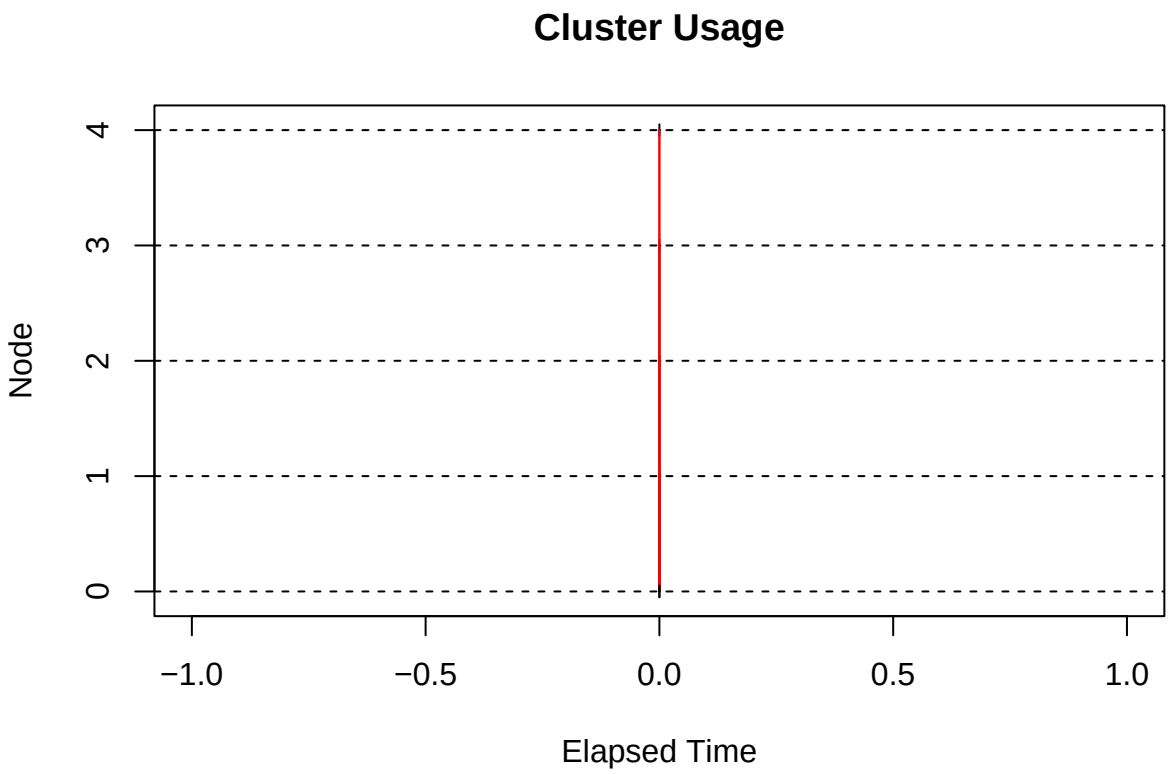
```r
#temps de calcul
toc()
```

```
## 4.48 sec elapsed
```

Rapport sur l'usage des coeurs

```r
# Rapport sur l'usage des coeurs
cl <- snow::makeCluster(k)
ctime1 <- snow.time(clusterApply(cl,blocs,fun=mean))

plot(ctime1)
```



10 Avec les packages Doparallel et Foreach

```
#nombre de cores à exploiter
#k <- 4
tic()
#partition en blocs des donnees
blocs <- split(a,1+(1:n)%%k)
#print(blocs)

#configurer les cores
doParallel::registerDoParallel(k)

#itérer sur les blocs
res <- foreach::foreach(b = blocs, .combine = c) %dopar% {
  return(mon_min(b))
}

#résultats intermédiaires
#print(res)

#minimum global
print(paste('Min foreach/dopar =',mon_min(res)))
```

```
## [1] "Min foreach/dopar = 0.126897253561765"
```

```
#stopper les cores
doParallel::stopImplicitCluster()

#affichage temps de calcul
print('>> Temps de calcul total avec foreach/dopar (split + min par bloc)')
```

```
## [1] ">> Temps de calcul total avec foreach/dopar (split + min par bloc)"
```

```
#temps de calcul
toc()
```

```
## 3.22 sec elapsed
```

11 La regression en parallèle

Données : Nous utilisons les données mtcars. Nous cherchons à expliquer la consommation (mpg) en fonction des autres variables.

```
data<-(data(mtcars))
#View(mtcars)
```

```
alea <- runif(nrow(mtcars))
cle <- ifelse(alea < 0.5, 1, 2)
blocs <- split(mtcars,cle)
print(blocs)
```

```
## $`1`
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
##
## $`2`
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
```

```
## Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
#reduce
reduce_lm <- function(D){
 #nombre de lignes
 n <- nrow(D)
 #récupération de la cible
 y <- D$mpg
 #prédictives
 X <- as.matrix(D[,-1])
 #rajouter la constante en première colonne
 X <- cbind(rep(1,n),X)
 #calcul de X'X
 XtX <- t(X) %*% X
 #calcul de X'y
 Xty <- t(X) %*% y
 #former une structure de liste
 res <- list(XtX = XtX, Xty = Xty)
 #renvoyer le tout
 return(res)
}
```

```
#Demarrage des moteurs (workers)
clust <- parallel::makeCluster(4)
#lancement des min en parallele
res <- parallel::parSapply(clust,blocs,FUN = reduce_lm)

#résultats intermédiaires
print(res)
```

```
##     1           2
## XtX numeric,121 numeric,121
## Xty numeric,11  numeric,11
```

```
#fonction de consolidation
#consolidation
#X'X
MXtX <- matrix(0,nrow=ncol(mtcars),ncol=ncol(mtcars))
for (i in seq(1,length(res)-1,2)){
 MXtX <- MXtX + res[[i]]
}
print(MXtX)
```

```
##                    cyl       disp         hp       drat         wt       qsec
##        32.000   198.000    7383.10    4694.00   115.0900   102.9520    571.160
## cyl   198.000  1324.000   51872.40   32204.00   691.4000   679.4040   3475.560
## disp 7383.100 51872.400 2179627.47 1291364.40 25094.7960 27091.4888 128801.504
## hp   4694.000 32204.000 1291364.40  834278.00 16372.2800 16471.7440  81092.160
## drat  115.090   691.400   25094.80   16372.28   422.7907   358.7190   2056.914
```

```
## wt    102.952   679.404   27091.49   16471.74   358.7190   360.9011   1828.095
## qsec  571.160  3475.560  128801.50   81092.16  2056.9140  1828.0946  10293.480
## vs     14.000    64.000    1854.40    1279.00    54.0300    36.5580    270.670
## am     13.000    66.000    1865.90    1649.00    52.6500    31.3430    225.680
## gear  118.000   710.000   25650.30   17112.00   432.9500   366.5820   2097.460
## carb   90.000   604.000   23216.10   15776.00   321.2600   310.5020   1547.670
##            vs       am      gear      carb
##        14.000   13.000   118.000    90.000
## cyl    64.000   66.000   710.000   604.000
## disp 1854.400 1865.900 25650.300 23216.100
## hp   1279.000 1649.000 17112.000 15776.000
## drat   54.030   52.650   432.950   321.260
## wt     36.558   31.343   366.582   310.502
## qsec  270.670  225.680  2097.460  1547.670
## vs     14.000    7.000    54.000    25.000
## am      7.000   13.000    57.000    38.000
## gear   54.000   57.000   452.000   342.000
## carb   25.000   38.000   342.000   334.000
```

```
#X'y
MXty <- matrix(0,nrow=ncol(mtcars),ncol=1)
for (i in seq(2,length(res),2)){
 MXty <- MXty + res[[i]]
}
print(MXty)
```

```
##            [,1]
##         642.900
## cyl    3693.600
## disp 128705.080
## hp    84362.700
## drat   2380.277
## wt     1909.753
## qsec  11614.745
## vs      343.800
## am      317.100
## gear   2436.900
## carb   1641.900
```

```
#Eteindre les moteurs
parallel::stopCluster(clust)
```

Estimation des paramètres de la régression. Les estimateurs â sont produits à l'aide de procédure solve() de R.

```
#coefficients de la régression
a.chapeau <- solve(MXtX,MXty)
print(a.chapeau)
```

```
##             [,1]
##      12.30337416
## cyl  -0.11144048
## disp  0.01333524
## hp   -0.02148212
## drat  0.78711097
## wt   -3.71530393
```

```
## qsec  0.82104075
## vs    0.31776281
## am    2.52022689
## gear  0.65541302
## carb -0.19941925
```

Vérification - Procédure lm() de R. A titre de vérification, nous avons effectué la régression à l'aide de la procédure lm() de R.

```
print(summary(lm(mpg~.,data=mtcars)))
```

```
##
## Call:
## lm(formula = mpg ~ ., data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4506 -1.6044 -0.1196  1.2193  4.6271
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 12.30337   18.71788   0.657   0.5181
## cyl         -0.11144    1.04502  -0.107   0.9161
## disp         0.01334    0.01786   0.747   0.4635
## hp          -0.02148    0.02177  -0.987   0.3350
## drat         0.78711    1.63537   0.481   0.6353
## wt          -3.71530    1.89441  -1.961   0.0633 .
## qsec         0.82104    0.73084   1.123   0.2739
## vs           0.31776    2.10451   0.151   0.8814
## am           2.52023    2.05665   1.225   0.2340
## gear         0.65541    1.49326   0.439   0.6652
## carb        -0.19942    0.82875  -0.241   0.8122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.65 on 21 degrees of freedom
## Multiple R-squared:  0.869,  Adjusted R-squared:  0.8066
## F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

12 MAP REDUCE