Calcul parallele sur R

Fogwoung Djoufack Sarah-Laure & Aissatou Sega Diallo

2025-03-25

INTRODUCTION

Dans le contexte de l'analyse de grandes bases de données et des statistiques, le calcul parallèle apparaît comme une solution incontournable pour accélérer les traitements. Il s'agit de diviser une tâche complexe en plusieurs petites tâches qui s'exécutent simultanément. Cela permet de réduire drastiquement le temps de calcul, surtout lorsqu'on manipule des millions de lignes ou qu'on réalise de nombreuses simulations.

Qu'est-ce que le calcul parallèle ?

Le **calcul parallèle** consiste à exécuter plusieurs tâches en même temps en utilisant plusieurs ressources informatiques(appelées coeurs ou threads). En pratique, au lieu de réaliser toutes les opérations une par une (calcul séquentiel), on les répartit entre plusieurs processeurs ou coeurs pour exécuter plusieurs tâches en même temps.

Dans un calcul parallèle, un grand problème est **divisé** en plusieurs sous-problèmes plus petits, et ces sous-problèmes sont **résolus simultanément**, ce qui permet de gagner du temps.

Différence entre calcul séquentiel et calcul parallèle

Calcul Séquentiel

Principe :
Chaque tâche est exécutée l'une après l'autre.
Exemple :
Si vous devez traiter 100 itérations d'un calcul sur une grande base de données, chaque itération attend la fin de la précédente.

Avantages :
- Simplicité d'implémentation.
- Gestion de la mémoire souvent plus directe.

Limitations :
- Lorsque le volume de données est important, le temps d'exécution peut devenir prohibitif.

Calcul Parallèle

Principe :
Le travail est divisé entre plusieurs coeurs du processeur.
Exemple :
Si votre ordinateur dispose de 4 coeurs, vous pouvez distribuer les 100 itérations en affectant environ 25 itérations à chaque coeur pour qu'elles s'exécutent simultanément.

Avantages :
- Réduction significative du temps de calcul grâce à l'exécution simultanée des tâches.

Limitations :
- La mise en place du parallélisme demande une gestion de la répartition des tâches et la collecte des résultats, ce qui est généralement pris en charge par le "programme maître".

Les différents types de parallélisme

Dans cette partie, nous explorons les types de parallélisme que l'on peut utiliser pour diviser une tâche complexe en plusieurs sous-tâches exécutées simultanément.

Parallélisme de données

Le parallélisme de données consiste à appliquer la même opération à plusieurs morceaux de données différents en même temps. En d'autres termes, l'objectif est de diviser les données en petites parties, de sorte que chaque partie de données soit traitée de façon indépendante et simultanée par un ou plusieurs processeurs (cœurs). Ce type de parallélisme est particulièrement efficace lorsque la tâche à réaliser peut être appliquée de manière uniforme à de grandes quantités de données.

Exemple: Calculer la moyenne de revenu pour chaque région dans une grande base de données où chaque région est traitée en parallèle.

Parallélisme de tâches

Le parallélisme de tâches consiste à exécuter différentes opérations en même temps. Contrairement au parallélisme de données où la même opération est effectuée sur différentes parties des données, ici, les tâches elles-mêmes sont différentes, mais elles peuvent être exécutées en parallèle sans se gêner les unes les autres.

**Exemple*: Une tâche fait le nettoyage des données, une autre effectue une analyse statistique, et une autre génère des graphiques. Ces trois tâches sont effectuées en parallèle.

Parallélisme distribué et parallélisme partagé

Il existe deux principales approches pour utiliser plusieurs processeurs afin de faire du calcul parallèle : le parallélisme distribué et le parallélisme partagé. Ces approches déterminent comment les ressources sont partagées entre plusieurs processeurs, qu'ils soient dans un même ordinateur ou répartis sur plusieurs machines.

Le *parallélisme partagé* est basé sur l'idée que plusieurs cœurs d'un même ordinateur (ou plusieurs threads dans un même processus) partagent la même mémoire. Les différents processeurs ou cœurs d'un même ordinateur peuvent accéder aux mêmes données en temps réel. Ils peuvent partager des ressources comme la mémoire, les variables et les résultats intermédiaires.

D'autre part, le *parallélisme distribué* va plus loin en répartissant le calcul sur plusieurs machines connectées par un réseau. Chaque machine travaille sur une portion des données et communique avec les autres machines pour échanger des informations ou combiner les résultats. Dans ce cas, chaque machine peut avoir sa propre mémoire et ses propres ressources, et elles doivent s'échanger des données via un réseau (comme Internet ou un réseau local).

Concepts Techniques

1. Coeurs du Processeur

Définition :
Un **coeur** est une unité physique de traitement au sein d'un processeur.
- **Exemple :**
Un processeur *quad-core* possède 4 coeurs physiques, chacun pouvant exécuter des instructions indépendamment des autres.

2. Threads (ou Processeurs Logiques)

Définition :
Un **thread** est une séquence d'instructions que le processeur peut exécuter.
- **Hyper-threading :**
C'est une technologie qui permet à un seul coeur de traiter plusieurs threads en simultané.
- **Exemple :**
Un processeur quad-core doté d'hyper-threading peut gérer 8 threads, c'est-à-dire qu'il peut théoriquement exécuter 8 tâches en même temps.

3. Lien entre Coeurs et Threads

- **Physiquement :**
  Vous disposez d'un nombre fixe de coeurs physiques (par exemple, 4 coeurs sur un processeur quad-core).

- **Logiquement :**
  Chaque coeur peut être "dédoublé" en plusieurs threads grâce à l'hyper-threading. Ainsi, même avec 4 coeurs, vous pouvez avoir plus d'unités d'exécution simultanée (par exemple, 8 threads).

- **En pratique pour le calcul parallèle :**
  On parle souvent de "coeurs" pour simplifier, mais ce sont en réalité les **threads** (unités logiques) qui exécutent les tâches. Le nombre de threads disponibles détermine combien de tâches peuvent être réellement exécutées en parallèle.

4. Utilisation en R

Pour connaître le nombre de coeurs ou de threads disponibles sur votre machine, vous pouvez utiliser la fonction `detectCores()` du package **parallel** :

```
# Ce package n'a pas besoin d'être téléchargé au préalable,
# il est directement disponible lorsque R est installé
library(parallel)

# Nombre de coeurs physiques
nb_coeurs_physiques <- detectCores(logical = FALSE)
print(nb_coeurs_physiques)
```

```
## [1] 10
```

```
# Be careful, ce n'est pas parce que l'ordi est intel core i5
#par exemple qu'il a automatiquement 5 coeurs

# Nombre de threads (unités logiques)
nb_threads <- detectCores(logical = TRUE)
print(nb_threads)
```

```
## [1] 12
```

```
## Mon ordinateur a 10 coeurs et 12 thread et donc
#il y a 2 coeurs qui grace au hyper-threading gerent 2 threads chacun
#et les 8 autres threads gerent 1 thread chacun.
# Donc pour le calcul parallele, R va utiliser le
#nombre de threads=12 pour optimiser le calcul parallèle
```

5. Programme Maître et son rôle dans le calcul parallèle

Le programme maître est essentiel pour coordonner le calcul parallèle en distribuant les tâches et en recueillant les résultats. Il est chargé de trois responsabilités principales :

- **Répartition des tâches:**

Le programme maître divise une tâche globale en plusieurs sous-tâches qui peuvent être exécutées simultanément par différents threads ou cœurs de processeur. L'objectif est de distribuer la charge de travail de manière équilibrée pour maximiser l'efficacité de l'exécution parallèle. Ainsi, chaque thread travaille sur une portion de la tâche, ce qui accélère l'exécution du programme.

- **Lancement en parallèle:**

Une fois que les sous-tâches sont réparties, le programme maître envoie chaque sous-tâche aux threads disponibles pour exécution simultanée. Ces threads peuvent être répartis sur les différents cœurs de processeur ou machines.

- **Collecte des résultats:**
  Après que chaque thread a terminé sa partie du travail, le programme maître récupère les résultats produits. Souvent, cela se fait à l'aide d'une fonction comme do.call() en R, qui permet de combiner ou de regrouper les résultats des différentes tâches en un seul ensemble de données final.

D'autres part, nous distinguons les sous-programmes (ou workers).

6. Sous-programmes (ou workers) et leur rôle dans le calcul parallèle

Ce sont les processus secondaires qui reçoivent une partie du travail et exécutent la tâche qui leur est assignée. Une fois leur travail terminé, ils renvoient les résultats au programme maître qui les agrège pour obtenir le résultat final.

Importance du calcul parallèle

Dans le domaine de la science des données, de la statistique et même du machine learning , le calcul parallèle est essentiel car il permet de :

1. **Accélérer le traitement des données** : En divisant le travail, on le fait plus rapidement. Par exemple, pour calculer des moyennes ou d'autres statistiques sur une grande base de données, l'exécution en parallèle permet de distribuer le travail et de gagner du temps.

2. **Optimiser l'utilisation des ressources** : Plusieurs processeurs ou machines travaillent ensemble, ce qui permet de mieux exploiter la puissance de calcul disponible.

3. **Gérer les grandes quantités de données** : Pour des bases contenant des millions voire des dizaines de millions de lignes, le calcul séquentiel serait trop lent pour fournir des résultats en temps utile.

Pour mieux comprendre celà, nous allons generer un jeu de données contenant 20 000 lignes représentant des revenus dans différentes régions du Sénégal puis comparer les temps d'exécution des calculs réalisés en mode séquentiel et en mode parallèle.

- *Generons les données*

On va simuler des données comprenant des identifiants (id), des sexes (Homme/Femme), des régions (par exemple, 5 régions au Sénégal), et des revenus qui suivent une distribution normale.

```
set.seed(123)
library(dplyr)
```

```
##
## Attachement du package : 'dplyr'

## Les objets suivants sont masqués depuis 'package:stats':
##
##     filter, lag

## Les objets suivants sont masqués depuis 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
# Création d'une base plus volumineuse (10 millions de lignes)
df <- data.frame(
  id = 1:1e7, # 10 fois plus de données
  sexe = sample(c("Homme", "Femme"), 1e7, replace = TRUE),
  region = sample(c("Dakar", "Thiès", "Saint-Louis", "Ziguinchor", "Kaolack"),
                  1e7, replace = TRUE),
  revenu = rnorm(1e7, mean = 250000, sd = 50000)
)
```

- **Methode séquentielle** Dans cette méthode, nous allons calculer la moyenne des revenus par région de manière séquentielle.

```r
system.time({
  result_seq <- df %>%
    group_by(region) %>%
    summarise(
      moyenne = mean(revenu),
      ecart_type = sd(revenu), # Ajout d'un calcul supplémentaire
      count = n()
    )
})
```

```
## utilisateur     système      écoulé
##        0.07        0.02        0.23
```

En calcul parallèle

Dans cette méthode, nous allons répartir le travail sur 12 threads disponibles pour exécuter la même opération en parallèle.

```r
########## INSTALLATION DES PACKAGES ##########
# Exécuter cette partie une seule fois
# install.packages(c("dplyr", "doParallel", "foreach", "ggplot2"))

#library(doParallel)
#library(foreach)

# Détection automatique des coeurs
#n_cores <- detectCores() - 1 # Garde un coeur libre
#cl <- makeCluster(n_cores)
#registerDoParallel(cl)

#system.time({
#  result_par <- foreach(
#    region_split = split(df, df$region),
#    .combine = bind_rows,
#    .packages = "dplyr"
#  ) %dopar% {
 #   region_split %>%
#      summarise(
#        region = first(region),
#        moyenne = mean(revenu),
#        ecart_type = sd(revenu),
#        count = n()
#      )
#  }
```

```
#})

#stopCluster(cl)

# Probleme to be solve, ca semble prendre plus de temps bizzarement
```

Outils et packages pour le calcul parallèle en R

Dans cette section, nous allons explorer les outils et packages qui permettent de mettre en œuvre le calcul parallèle en R pour les deux grandes méthodes de parallélisation : parallélisation locale et parallélisation distribuée.

1. Calcul Parallèle en Local (Sur un Seul Ordinateur)

1.1 Le Package `parallel`

- **Description :**
  Le package `parallel` est intégré à R et fournit des fonctions de base pour la parallélisation, telles que :
  - `mclapply()`, `mcmapply()`: fonctionnent sous Linux/Mac pour lancer des tâches en parallèle.
  - `makeCluster()`, `clusterApply()`, `parLapply()`, etc. : ces fonctions permettent de créer un cluster de workers, c'est-à-dire d'ouvrir plusieurs sessions R en parallèle (fonctionne aussi sur Windows).
- **Exemple de Code :**

```
  library(parallel)

  # Définir le nombre de cores à utiliser
  n_cores <- detectCores(logical = TRUE)  # par exemple, 12 sur ma machine

  # Exemple avec mclapply (fonctionne sur Linux/Mac)
#  resultats <- mclapply(1:10, function(x) { sum(rnorm(1e6)) }, mc.cores = n_cores)
#  print(resultats)

  # Exemple avec clusterApply

# Créer un cluster de 4 workers pour exécuter des tâches en parallèle
cl <- makeCluster(4)
```

```r
# Appliquer une fonction à chaque élément du vecteur 1:10
# en répartissant les tâches sur les 4 workers.
# Pour chaque élément (de 1 à 10), la fonction
# génère 1 million de nombres aléatoires (rnorm(1e6))
# et calcule leur somme (sum(rnorm(1e6))).
resultats2 <- clusterApply(cl, 1:10, function(x) { sum(rnorm(1e6)) })

# Libérer le cluster en fermant les 4 workers,
# afin de libérer les ressources système utilisées
stopCluster(cl)

# Afficher la liste des 10 résultats obtenus,
# chaque élément correspondant à la somme calculée par la fonction
print(resultats2)
```

```
## [[1]]
## [1] -205.3716
##
## [[2]]
## [1] 261.3109
##
## [[3]]
## [1] -810.0334
##
## [[4]]
## [1] 633.0589
##
## [[5]]
## [1] -128.6271
##
## [[6]]
## [1] 1279.774
##
## [[7]]
## [1] -1323.693
##
## [[8]]
## [1] 580.787
##
## [[9]]
## [1] 1049.733
##
## [[10]]
## [1] 919.6673
```

JUST FOR ME: Je me posais la question de pourquoi là j'ai 12 threads dans mon ordi et là je suis entrain de traiter 10 vecteurs d'éleme,ts alors pourquoi ne pas utiliser 10threads alors que j'en ai la possibilité. Mais enfaite, certe là en utilisant 10threads et donc 10workers, ca reduirait le temps de traitement mais parfois askip c'est mieux de choisir un nombre plus faible pour éviter de saturer le systeme ou pour des raisons de gestion de la mémoire.

MAIS COMPARONS LE TEMPS POUR VOIR

```
library(parallel)

# Comparaison avec 4 workers (threads)
cl4 <- makeCluster(4)  # Créer un cluster avec 4 workers
time_4 <- system.time({
  # Appliquer la fonction à chaque élément de 1:10 sur le cluster de 4 workers
  result_4 <- clusterApply(cl4, 1:10, function(x) { sum(rnorm(1e6)) })
})
stopCluster(cl4)  # Libérer le cluster
print("Temps avec 4 workers :")
```

```
## [1] "Temps avec 4 workers :"
```

```
print(time_4)
```

```
## utilisateur     système      écoulé
##        0.00        0.00        0.18
```

```
# Comparaison avec 10 workers (threads)
cl10 <- makeCluster(10)  # Créer un cluster avec 10 workers
time_10 <- system.time({
  # Appliquer la fonction à chaque élément de 1:10 sur le cluster de 10 workers
  result_10 <- clusterApply(cl10, 1:10, function(x) { sum(rnorm(1e6)) })
})
stopCluster(cl10)  # Libérer le cluster
print("Temps avec 10 workers :")
```

```
## [1] "Temps avec 10 workers :"
```

```
print(time_10)
```

```
## utilisateur     système      écoulé
##         0.0         0.0         0.1
```

On voit que le temps avec 4 workers est de 0,16 secondes et le temps avec les 10 workers est de 0,08 workers. Mais quand on ouvre le gestionnaire de taches pour voir la mémoire utilisée par chacune et qu'on somme l'espace mémmoir utilisée par les 4 workers d'une part et puis l'espace memoir pour les 10 workers on voit bien cette diffrence de taille de memoire utilisée. (Aussi pour voir cet espace memoire utilisé,on lance le programme puis on peut aller dans le gestionnaire de taches puis aller sur details puis rechercher R script exe qui montre l'espace memoire utilisé pour chacun des workers lancés)

PENSER MAYBE A METTRE UNE CAPTURE D'ECRAN POUR LA COMPARAISON

LA SUITE SUR LES PACKAGES A VENIR, JE VAIS METTRE DANSLE RMD QUAND J'AURAIS BIEN COMPRIS.

POUR APRES, RECHERCHER LES INFOS DE CE QUE SIGNIFIE LES utilisateur et systeme dans les resultats affichés. Comment choisir le nombre idéal de coeurs ? Y a t il meme une facon ?