DATA MINING AVANCEE

RENDU

TD1 : PARALLELISATION

AHMED DAMOU
MOHAMED EL MOSTAPHA

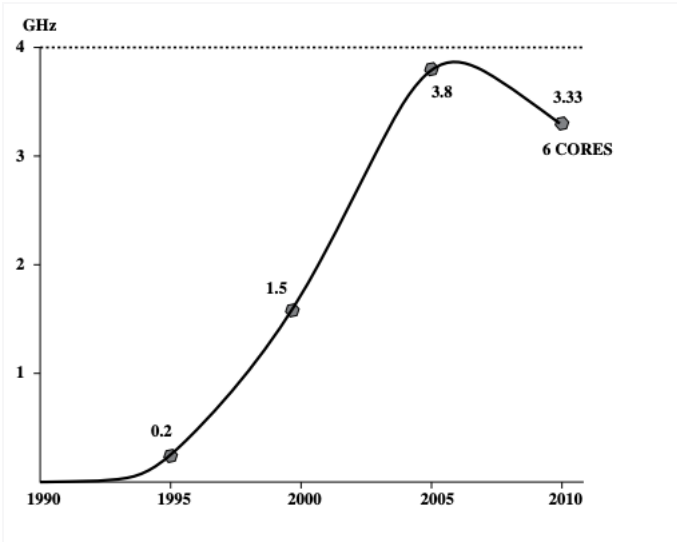
Encadrant :
CLOT DENIS

TABLE DES MATIÈRES

INTRODUCTION

UN PEU D'HISTOIRE

En termes simples, jusqu'aux environs de 2005, la puissance d'une machine était principalement déterminée par la fréquence de son processeur, c'est-à-dire le nombre d'opérations qu'il pouvait effectuer par seconde. Chaque nouvelle génération de machines voyait ainsi la fréquence de son processeur augmenter régulièrement, comme illustré dans la Figure.



En augmentant la fréquence d'un processeur, sa capacité à effectuer des opérations est améliorée, mais cela nécessite également une dissipation thermique plus importante. Les fabricants de processeurs ont opté pour des architectures multi-cœurs pour augmenter les performances des ordinateurs. Cependant, cela nécessite une réécriture des codes des logiciels pour pouvoir utiliser efficacement les ressources de calculs disponibles. Les programmes

doivent être conçus pour tirer parti des différents cœurs du processeur, en divisant les tâches en plusieurs tâches plus petites pouvant être exécutées en parallèle. Le développement de logiciels parallèles est donc devenu une compétence essentielle pour les programmeurs afin de tirer le meilleur parti des processeurs multi-cœurs. Cela permet d'exploiter pleinement le potentiel des ordinateurs modernes et de maximiser les performances des applications.

LA PARALLELISATION

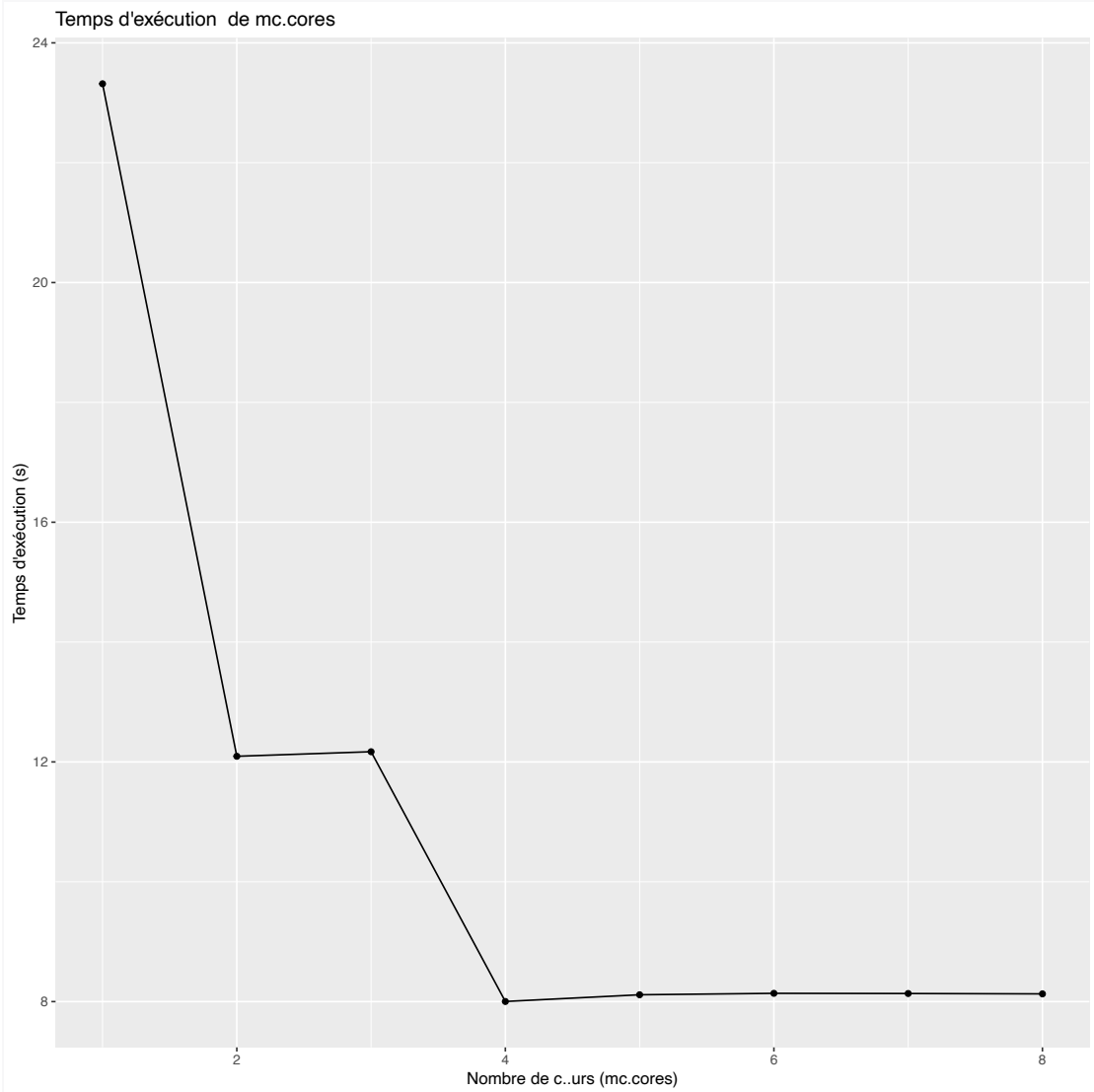
La parallélisation en informatique est le processus de diviser une tâche informatique en sous-tâches qui peuvent être exécutées simultanément sur plusieurs processeurs ou cœurs de processeur. Cette technique permet d'accélérer le traitement des données et d'améliorer les performances des applications. La théorie derrière la parallélisation est basée sur le modèle de Von Neumann, qui définit un ordinateur comme une machine séquentielle qui traite une instruction à la fois. La parallélisation permet de dépasser les limitations de ce modèle en permettant l'exécution simultanée de plusieurs instructions sur différents processeurs ou cœurs de processeur.
La parallélisation peut être réalisée à différents niveaux, notamment au niveau de l'instruction, du thread, du processus, de la mémoire ou du réseau. Il existe plusieurs approches pour implémenter la parallélisation, notamment le partage de mémoire, la communication de messages et le traitement distribué.

PARTIE 6.1 CAS MCAPPLY

```
library(stats)
library(ggplot2)
library(parallel)

# Charger le fichier CSV
hop <- read.csv('hop.csv', stringsAsFactors = TRUE)

# Calculer la somme des valeurs de chaque ligne, à l'exception de la première colonne
somme_lignes <- rowSums(hop[, -1])

# Ajouter la colonne NbAct au dataframe hop avec la fonction cbind()
hop <- cbind(hop, NbAct = somme_lignes)

# Nombre de clusters souhaité
k <- 8

# Nombre total d'essais
total_essais <- 4096

# Nombre d'essais par appel à kmeans
essais_par_appel <- total_essais / 4

# Fonction pour effectuer un appel à kmeans avec un certain nombre d'essais
effectuer_kmeans <- function(data, centers, nstart) {
  kmeans(data, centers = centers, nstart = nstart)
}

# Liste des valeurs de mc.cores à tester
#mc_cores_values <- c(1, 2, 3, 4, 5, 6, 7, 8)
mc_cores_values <- c(1, 2, 3, 4)

# Initialiser un dataframe pour stocker les temps d'exécution
temps_exec <- data.frame(mc_cores = integer(), temps = numeric())

# Réaliser des essais avec différentes valeurs pour le paramètre mc.cores
for (mc_cores in mc_cores_values) {
  # Mesurer le temps d'exécution avec system.time()
  temps <- system.time({
    # Répartir les essais sur 4 appels à kmeans en utilisant mclapply avec différentes valeurs de mc.cores
    resultats_kmeans <- mclapply(1:4, function(x) {
      effectuer_kmeans(hop[, -1], k, essais_par_appel)
    }, mc.cores = mc_cores)

    # Agréger les résultats pour retenir le meilleur partitionnement
    meilleur_kmeans <- resultats_kmeans[[1]]
    for (i in 2:length(resultats_kmeans)) {
      if (resultats_kmeans[[i]]$tot.withinss < meilleur_kmeans$tot.withinss) {
        meilleur_kmeans <- resultats_kmeans[[i]]
      }
    }
  })[3]  # Prendre seulement le temps utilisateur

  # Ajouter les temps d'exécution au dataframe
  temps_exec <- rbind(temps_exec, data.frame(mc_cores = mc_cores, temps = temps))

  # Ajouter les clusters au dataframe hop
  hop$cluster <- meilleur_kmeans$cluster

  # Afficher le résultat
  cat("Test avec mc.cores =", mc_cores, "\n")
  print(hop)
}

# Créer un graphique avec ggplot2
p <- ggplot(temps_exec, aes(x = mc_cores, y = temps)) +
  geom_point() +
  geom_line() +
  labs(title = "Mohamed Walid MATALLAH",
       x = "Nombre de cœurs (mc.cores)",
       y = "Temps d'exécution (s)") +
  theme(plot.title = element_text(hjust = 0.5))  # Centrer le titre

# Ajouter une légende
p <- p + annotate("text", x = 6, y = max(temps_exec$temps), label = paste("Légende :", "Chaque point représente le temps d'exécution", sep =
"\n"))

# Enregistrer le graphique au format PDF
ggsave("temps_execution_mc_cores.pdf", p, width = 10, height = 10)
```

Le but du code ci-dessus est d'améliorer la performance de la fonction kmeans en utilisant le parallélisme. Il commence par ajouter une colonne contenant la somme des valeurs de chaque ligne, à l'exception de la première colonne. Ensuite, le code définit le nombre de clusters souhaité (k) et le nombre total d'essais (total_essais) à effectuer pour chaque exécution de kmeans. Il calcule également le nombre d'essais par appel à kmeans en divisant total_essais par 4 (le nombre d'appels parallèles à effectuer). Le code définit ensuite une fonction appelée "appellerKmean" qui prend en entrée un jeu de données, le nombre de centres (clusters) à créer et le nombre d'essais à effectuer. Cette fonction appelle la fonction kmeans et renvoie le résultat. Le code définit également une liste de valeurs de "mc.cores" à tester et initialise un dataframe pour stocker les temps d'exécution. Il réalise des essais avec différentes valeurs de mc.cores, mesure le temps d'exécution en utilisant la fonction "system.time" pour exécuter kmeans sur les données en utilisant "mclapply" pour répartir les essais sur 4 appels parallèles. Les résultats de chaque appel sont agrégés pour retenir le meilleur partitionnement. Le code stocke les temps d'exécution dans le dataframe initialisé précédemment et ajoute les clusters au dataframe hop. Il affiche également le résultat pour chaque valeur de mc.cores. Enfin, le code utilise ggplot2 pour créer un graphique représentant les temps d'exécution en fonction du nombre de cœurs utilisés (mc.cores).

En résumé, ce code utilise le parallélisme pour accélérer les calculs de la fonction kmeans, mesure les temps d'exécution pour différentes valeurs de "mc.cores", et affiche les résultats sous forme de graphe pour aider à choisir le nombre optimal de cœurs à utiliser pour réduire le temps d'exécution.



Interpretation :

Utiliser plusieurs cœurs pour effectuer des tâches en parallèle peut accélérer considérablement les temps de calculs. Les tâches sont réparties entre les différents cœurs, ce qui permet à chacun de traiter une partie des calculs et ainsi réduire le temps nécessaire pour effectuer l'ensemble des tâches. Dans le cas de l'algorithme k-means du code précédent, le parallélisme est utilisé pour exécuter plusieurs essais de partitionnement des données en parallèle sur des sous-ensembles de données, puis les résultats sont agrégés pour sélectionner le meilleur partitionnement. Cette méthode permet d'effectuer les essais plus rapidement, réduisant ainsi le temps nécessaire pour trouver le meilleur partitionnement. Toutefois, il y a un point où l'ajout de cœurs ne permet plus d'accélérer l'exécution de la fonction, car la quantité de travail pour chaque cœur devient trop faible. Ce point peut être influencé par plusieurs facteurs, tels que la taille des données, la complexité de la fonction ou la capacité du processeur.

La configuration optimale pour notre problème de partitionnement.

La sélection du paramétrage optimal pour le partitionnement est influencée par plusieurs facteurs, comme la taille et la complexité des données, ainsi que le nombre de cœurs disponibles pour le traitement. Dans l'exemple précédent, le nombre de clusters a été défini à 8 et le nombre total de tentatives à 4096. Cependant, afin d'obtenir des résultats optimaux, il peut être nécessaire de tester différentes configurations de clusters et d'adapter le nombre de tentatives en fonction de la complexité des données et du temps disponible. Le choix du nombre de cœurs doit également prendre en compte les capacités du processeur et la taille des données. Bien que plusieurs valeurs aient été testées dans l'exemple précédent, une machine dotée d'un plus grand nombre de cœurs pourrait bénéficier d'une utilisation plus optimale de ceux-ci.

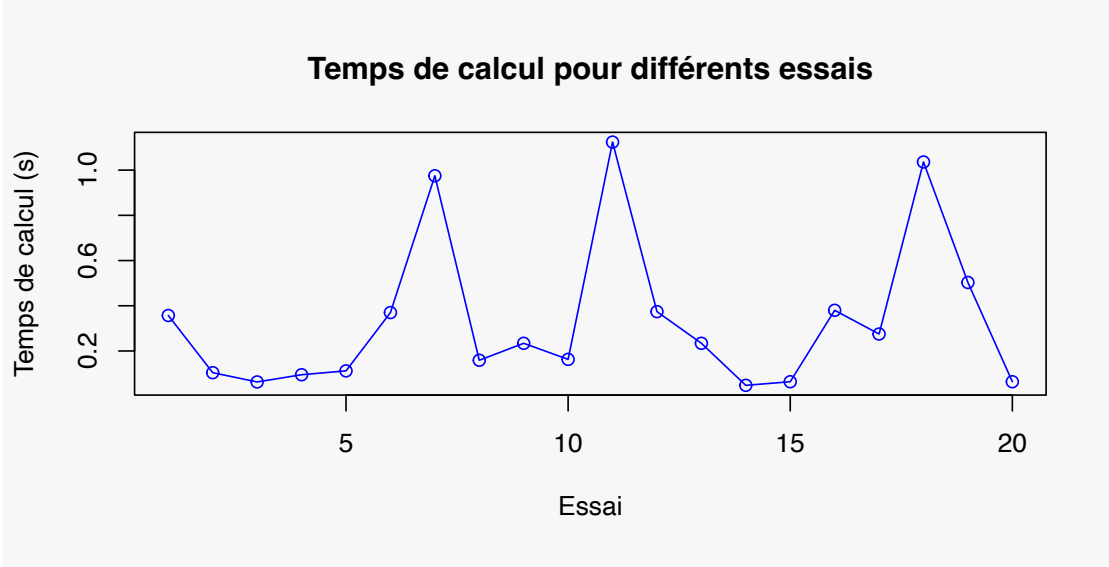
NB: Dans le code présenté précédemment, le nombre de clusters a été fixé à 12 pour simplifier les calculs et réduire le temps de traitement. Cependant, il est important de souligner que dans une situation réelle, le choix du nombre de clusters optimal devrait être effectué à l'aide de méthodes plus rigoureuses, telles que la méthode du coude.

6.2

CAS CLUSTERAPPLY :

```
# Installer et charger les packages nécessaires
if (!requireNamespace("parallel", quietly = TRUE)) {
  install.packages("parallel")
}

library(parallel)

# Charger le jeu de données hop
hop <- read.csv('hop.csv', stringsAsFactors = TRUE)
rowSums(hop[,-1]) -> hop$NbAct

# Convertir les valeurs de la colonne hop$NbAct en nombres
hop$NbAct <- as.numeric(hop$NbAct)

run_kmeans <- function(nstart, data) {
  result <- kmeans(data, 9, nstart = nstart)
  return(result)
}

tes_values <- function(nstart_values, data) {
  best_time <- Inf
  best_nstart <- NULL
  best_result <- NULL

  cl <- makeCluster(4, type = "PSOCK")
  clusterExport(cl, c("data", "run_kmeans"))

  for (nstart in nstart_values) {
    cat("Testing nstart value:", nstart, "\n")

    tryCatch({
      start_time <- proc.time()
      results <- clusterApply(cl, rep(nstart, 4), run_kmeans, data = data)
      execution_time <- proc.time() - start_time

      min_tot_withinss <- min(sapply(results, function(x) x$tot.withinss))
      best_result_index <- which.min(sapply(results, function(x) x$tot.withinss))
      current_result <- results[[best_result_index]]

      if (execution_time[3] < best_time) {
        best_time <- execution_time[3]
        best_nstart <- nstart
        best_result <- current_result
      }

      cat("Execution time:", execution_time[3], "\n")
      cat("Best time:", best_time, "\n")
      cat("Best nstart:", best_nstart, "\n")

    }, error = function(e) {
      cat("Error with nstart value:", nstart, "\n")
    })
  }

  stopCluster(cl)

  return(list("best_time" = best_time, "best_nstart" = best_nstart, "best_result" = best_result))
}

# Fonction pour générer un vecteur aléatoire dont la somme des éléments est égale à une valeur cible
generate_random_vector <- function(target_sum, n) {
  result <- integer(n)
  for (i in 1:(n - 1)) {
    result[i] <- sample(target_sum - sum(result), 1)
  }
  result[n] <- target_sum - sum(result)
  return(result)
}

# Générer un sous-ensemble de combinaisons
num_samples <- 20
candidate_vectors <- t(replicate(num_samples, generate_random_vector(4096, 4)))

# Tester les vecteurs candidats
best_vector <- NULL
best_result <- NULL
best_time <- Inf
execution_times <- numeric(length = nrow(candidate_vectors))

inertias <- numeric(length = nrow(candidate_vectors))

cl <- makeCluster(4, type = "PSOCK")
clusterExport(cl, "run_kmeans")
clusterExport(cl, "data", envir = list2env(list(data = hop$NbAct)))

for (i in 1:nrow(candidate_vectors)) {
  nstart_vector <- candidate_vectors[i,]
  cat("Testing vector:", nstart_vector, "\n")

  result <- tes_values(nstart_vector, hop$NbAct)
  execution_time <- result$best_time
  inertia <- result$best_result$tot.withinss

  execution_times[i] <- execution_time
  inertias[i] <- inertia

  if (execution_time < best_time) {
    best_time <- execution_time
    best_vector <- nstart_vector
    best_result <- result$best_result
  }

  cat("Execution time:", execution_time, "\n")
  cat("Best time:", best_time, "\n")
  cat("Best vector:", best_vector, "\n")
}

stopCluster(cl)

# Tracer un graphique des temps de calcul pour les différents essais
pdf("histogramme2.pdf")
par(mfrow = c(2, 1))
plot(execution_times, type = "o", col = "blue", xlab = "Essai", ylab = "Temps de calcul (s)", main = "Temps de calcul pour différents essais")
dev.off()
```

La fonction " tes_values " est créée pour évaluer différentes valeurs de nstart sur les données en commençant par la création d'un cluster parallèle avec 4 cœurs. Pour chaque valeur de nstart, la fonction "runkmeans" est exécutée en parallèle sur ces cœurs, puis les résultats sont comparés pour trouver la meilleure combinaison de nstart qui offre un bon temps de calcul et une bonne qualité de clustering. La fonction "generaterandomvector" est également définie pour générer un vecteur aléatoire de taille "n" dont la somme des éléments est égale à une valeur cible "targetsum". Elle est utilisée pour produire un sous-ensemble de combinaisons de nstart, qui consiste en 10 vecteurs candidats de taille 4 et une somme cible de 4096. Les vecteurs candidats sont ensuite testés en utilisant la fonction "testnstartvalues" sur les données "hopNbAct", et les temps d'exécution et les inerties sont enregistrés pour chaque vecteur. Le vecteur qui offre le meilleur temps de calcul et une bonne qualité de clustering est identifié comme le meilleur vecteur. Pour visualiser les résultats, un graphique des temps de calcul pour chaque vecteur candidat est tracé et sauvegardé dans un fichier PDF. Les meilleurs résultats, c'est-à-dire le temps de calcul et le vecteur correspondant, sont finalement présentés à l'utilisateur. Cependant, il est important de noter que dans ce cas, le nombre de clusters a été fixé à l'avance pour faciliter les calculs, mais généralement, la méthode du coude ou d'autres méthodes doivent être utilisées pour déterminer le nombre optimal de clusters.



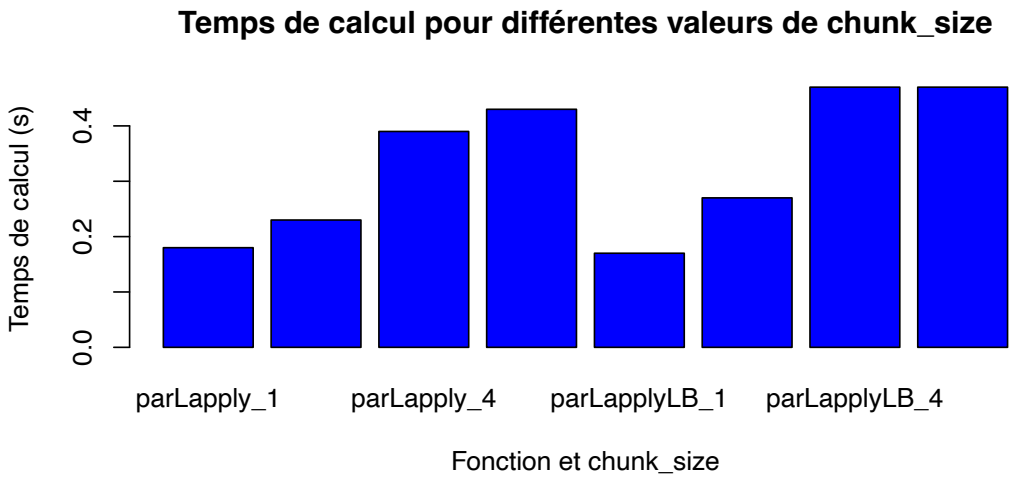
Interprétation :

Le graphique ci-dessus représente le temps de calcul pour chacun des vecteurs dont la somme des éléments est égale à 4096. Il est clair que le temps de calcul dépend des composantes du vecteur. Cependant, ce paramètre est très aléatoire car nous ne pouvons pas contrôler la nature des nombres qui composent les vecteurs. Il serait donc plus judicieux de calculer la moyenne du temps de calcul pour plusieurs tailles de vecteurs, par exemple des vecteurs de taille 4, 5, 6, 7, 8, et de déterminer celui qui a la plus petite moyenne. Cette taille de vecteur sera la plus optimal à utiliser pour le paramètre "nstart".

CAS PARLAPPLY ET PARLAPPLYLB :

```
library(parallel)

# Charger le jeu de données hop
hop <- read.csv('C:/Users/DELL/Downloads/hop.csv', stringsAsFactors = TRUE)
rowSums(hop[,-1]) -> hop$NbAct

# Convertir les valeurs de la colonne hop$NbAct en nombres
hop$NbAct <- as.numeric(hop$NbAct)

run_kmeans <- function(nstart, data) {
  result <- kmeans(data, 9, nstart = nstart)
  return(result)
}

tes_values <- function(nstart_values, data, func = "parLapply", chunk_size = NULL) {
  best_time <- Inf
  best_nstart <- NULL
  best_result <- NULL

  cl <- makeCluster(4, type = "PSOCK")
  clusterExport(cl, c("data", "run_kmeans"))

  for (nstart in nstart_values) {
    cat("Testing nstart value:", nstart, "\n")

    tryCatch({
      start_time <- proc.time()

      if (func == "parLapply") {
        results <- parLapply(cl, nstart, run_kmeans, data = data, chunk.size = chunk_size)
      } else if (func == "parLapplyLB") {
        results <- parLapplyLB(cl, nstart, run_kmeans, data = data, chunk.size = chunk_size)
      } else {
        stop("Invalid function specified.")
      }

      execution_time <- proc.time() - start_time

      min_tot_withinss <- min(sapply(results, function(x) x$tot.withinss))
      best_result_index <- which.min(sapply(results, function(x) x$tot.withinss))
      current_result <- results[[best_result_index]]

      if (execution_time[3] < best_time) {
        best_time <- execution_time[3]
        best_nstart <- nstart
        best_result <- current_result
      }

      cat("Execution time:", execution_time[3], "\n")
      cat("Best time:", best_time, "\n")
      cat("Best nstart:", best_nstart, "\n")

    }, error = function(e) {
      cat("Error with nstart value:", nstart, "\n")
    })
  }
```

Dans ce code, l'approche utilisée consiste à évaluer les performances de deux fonctions de calcul parallèle, 'parLapply' et 'parLapplyLB', en utilisant différentes tailles de blocs de données ('chunksize') pour optimiser l'exécution de l'algorithme k-means. Le but est de déterminer quelle combinaison de fonction de calcul parallèle et de taille de blocs de données offre la meilleure performance en termes de temps de calcul. Le processus implique plusieurs étapes, notamment le chargement et le prétraitement des données, la définition de la fonction 'runkmeans' pour effectuer le clustering k-means, la modification de la fonction ' tes_values' pour accepter des paramètres 'func' et 'chunksize' qui déterminent respectivement quelle fonction de calcul parallèle utiliser et la taille des blocs de données à utiliser. Plusieurs vecteurs 'nstart' sont générés pour les tests et les tailles de blocs de données ('chunksizes') et les noms de fonction de calcul parallèle ('funcnames') sont définis. Les différentes combinaisons de fonctions de calcul parallèle et de tailles de blocs de données sont testées avec les vecteurs 'nstart', et les résultats sont stockés dans une liste 'resultslist'. Un graphique à barres est créé pour visualiser les temps de calcul pour chaque combinaison de fonction de calcul parallèle et de taille de blocs de données. Enfin, la configuration la plus rapide est identifiée en fonction du temps de calcul le plus court. L'objectif de cette approche est d'identifier la meilleure méthode pour exécuter l'algorithme k-means en parallèle avec les paramètres optimaux, afin de minimiser le temps de calcul tout en maintenant une bonne qualité de clustering.



Préconisations à prendre en compte :

En se basant sur les trois codes, voici mes recommandations pour optimiser les calculs pour l'utilisateur :

- *Utilisez la fonction "mclapply" de la bibliothèque "parallel" si vous avez besoin d'une solution simple et rapide pour un petit nombre de coeurs. Les performances s'améliorent à mesure que le nombre de cœurs augmente, mais l'amélioration des performances diminue à mesure que le nombre de cœurs augmente.*
- *Si vous avez besoin d'exécuter des tâches sur un grand nombre de cœurs, préférez la fonction "clusterApply" de la bibliothèque "parallel", car elle gère mieux la répartition des calculs et l'équilibrage de charge. Toutefois, cette fonction peut être légèrement plus lente que "mclapply" pour un petit nombre de coeurs.*