Package R: Introduction à parallel

Adrien JUPITER & Claire Mazzucato

20 Décembre 2020

Contents

Introduction

Lorsqu'un programme R s'avère trop lent pour répondre aux besoins de ses utilisateurs, son temps d'exécution doit être optimisé. Il existe plusieurs stratégies pour arriver à cette optimisation. Il est recommandé, notamment, de:

- 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 l'application de ces bonnes pratiques, il arrive qu'un programme R demeure trop lent. Dans la situation d'un programme R qui doit appeler un très grand nombre de fois une fonction, potentiellement développée par d'autres personnes, une solution appropriée pour optimiser le temps d'exécution est alors le calcul en parallèle.

L'objectif du calcul en parallèle est d'effectuer plus rapidement un calcul informatique en exploitant simultanément plusieurs unités de calcul. Dans sa version la plus simpliste, un calcul en parallèle est effectué en réalisant les étapes suivantes:

- briser un calcul informatique en blocs de calcul indépendants
- exécuter simultanément, soit en parallèle, les blocs de calcul sur plusieurs unités de calcul
- rassembler les résultats et les retourner

La réalisation de calculs en parallèle requière donc l'accès à plusieurs unités de calcul. Celles-ci peuvent être localisées sur CPU et/ou sur GPU. Le calcul en parallèle sur GPU a été prouvé plus rapide que sur CPU pour bon nombre d'applications. Malgré cela, le calcul sur GPU a été, pour l'instant, moins étudié et documenté par la communauté R que le calcul en parallèle sur CPU. En conséquence, il existe moins de packages R pour soutenir un utilisateur de R dans la réalisation de calculs en parallèle sur GPU que sur CPU. Le seul package pour le calcul en parallèle fourni avec la distribution de base de R, donc ayant le sceau de confiance du R core team, permet uniquement de réaliser du calcul R en parallèle sur CPU. Ce package, nommé parallel, est l'outil privilégié dans ce document pour lancer des calculs R en parallèle.

Composantes matérielles d'un ordinateur impliquées dans le calcul en parallèle

Afin de comprendre le calcul informatique de pointe, il est utile de connaître minimalement les composantes d'un ordinateur et son fonctionnement. Les composantes d'un ordinateur sont souvent sous-divisées en 2 grandes classes : les composantes matérielles et logicielles. Nous nous intéressons ici aux composantes matérielles uniquement. Les composantes les plus importantes pour le calcul en parallèle sont décrites dans les sous-sections suivantes. Il s'agit de l'espace de stockage, ainsi que des processeurs, CPU et GPU, sur lesquels sont situés les unités de calcul.

Espace de stockage

Étant donné que des calculs traitent des données et en produisent d'autres, un espace pour stocker celles-ci est nécessaire. Les espaces de stockage les plus couramment retrouvés sur un ordinateur sont les suivants:

- disque : stockage permanent, grande capacité, le moins rapide d'accès (même si les disques SSD, soit les Solid State drive, sont plus rapides que les disques durs classiques)
- mémoire vive (ou RAM pour Random Access Memory) : stockage temporaire, capacité plus petite quele disque, mais plus rapide d'accès que ce dernier
- mémoire cache (intégrée au processeur) : stockage temporaire, petite capacité, très rapide d'accès

Aucune de ces issues n'est réellement souhaitable. Que fait R dans une telle situation ? Généralement, R arrête le programme et génère une erreur, mais il arrive parfois qu'il plante.

Processeur ou CPU

Le rôle du processeur ou CPU (Central Processing Unit) est de lire et d'exécuter les instructions provenant 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. Un processeur désigne une puce informatique pouvant contenir plus d'un coeur. 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. Les fils d'exécution sont exécutés séquentiellement par un coeur, soit un après l'autre. 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.

Processeur graphique ou GPU

Un processeur graphique ou GPU (Graphical Processing Unit) est un « processeur massivement parallèle, disposant de sa propre mémoire assurant les fonctions de calcul de l'affichage ». À l'origine développé uniquement pour l'affichage graphique, et très exploité par les jeux vidéos, plusieurs GPUs sont maintenant conçus de façon à pouvoir y lancer des calculs.

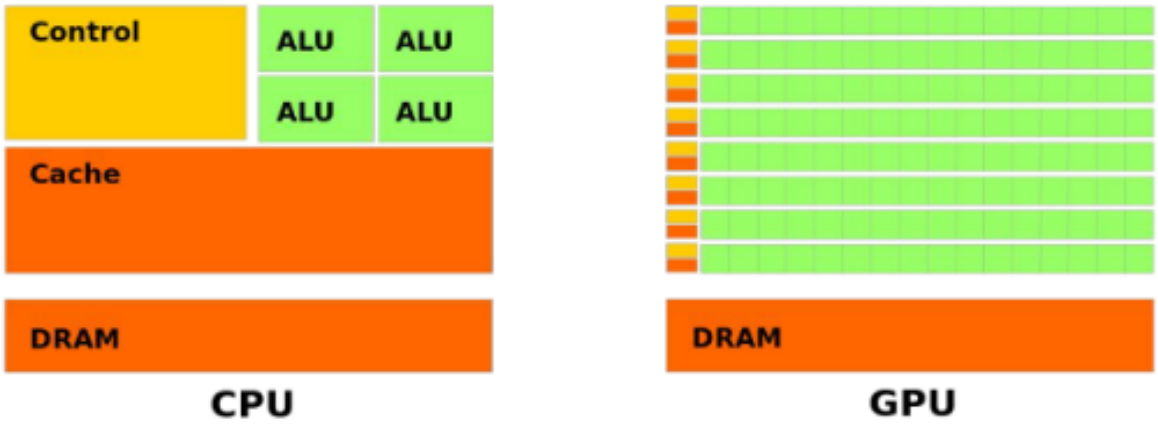

Figure 1: CPU versus GPU, en orange la mémoire, en jaune les unités de contrôle et en vert les unités decalcul

Comme l'illustre la figure 1, un GPU contient un bien plus grand nombre d'unités de calculs qu'un CPU, d'où un potentiel d'accélération accru. Cependant, y lancer des calculs en parallèle requière l'utilisation d'une technologie propre au GPU exploité (par exemple CUDA sur un GPU NVIDIA). Certains packages R ont été développés pour éviter d'avoir à utiliser directement la technologie en question. Ils proposent des fonctions R usuelles faisant office d'interface entre R et l'outil pour communiquer avec le GPU. Ces packages R sont pour l'instant peu nombreux et relativement récents. Ils sont moins matures que ceux servant à lancer des calculs R en parallèle sur CPU. Ils ne seront pas abordés dans ce document, mais il est bien de garder en tête leur existence.

Le package parallel et son utilisation

Le package parallel est inclus dans R depuis sa version 2.14.0 (31 octobre 2011). Il propose un ensemble de fonctionnalités pour introduire du parallélisme dans les codes R: il s'agit également d'attirer le plus grand nombre d'utilisateurs de R et développeurs de packages vers le calcul parallèle compte tenu de l'évolution des architectures matérielles. parallel est dérivé des packages CRAN multicore (2009-) et snow (2003-): il propose des fonctions de substitutions à la plupart des fonctionnalités de ces deux packages, en intégrant par ailleurs la gestion des générateurs de nombres pseudo-aléatoires parallèles. De nombreux codes R intègrent les fonctions de la famille *apply qui sont des versions avantageuses des boucles for (en terme de performance souvent et en terme de lisibilité). Dans cette logique fonctionnelle, la plupart des utilisateurs de parallel se focaliseront sur les versions parallèles de ces fonctions: parApply, parLapply, parSapply, parVapply et autres (dans la suite, on parlera des par*apply). Ces fonctions reprennent les même arguments que les versions séquentielles, mais nécessitent la création d'un pool de processus fils avec la fonction makeCluster, le passage de ce pool en premier argument, puis la suppression de ce pool:

Installation

Le package parallel est par défault installé avec R. Si vous avez un doute vous pouvez lancer la commande suivante afin de s'en assurer:

```
installed.packages()
```

Si le package est manquant, vous pouvez l'installer à partir du CRAN. Le CRAN est le dépôt informatique de packages R géré par le R core team. C'est là que la majorité des packages R sont rendus disponibles publiquement. Pour installer un package à partir d'un des miroirs du CRAN, il suffit d'utiliser la fonction install.packages comme dans cet exemple :

```
install.packages("parallel")
```

Il vous suffit maintenant d'appeler la librairie dans votre code en ajoutant la ligne suivante:

```
library(parallel)
```

A partir de là, vous avez le minimum requis afin de pouvoir écrire du code parallel sous R.

Pour savoir combien de coeurs on dispose sur notre machine, on peut utiliser la fonction detectCores() du package parallel:

```
library("parallel")
detectCores() # nombre de coeurs physiques
```

Syntaxe pour lancer un calcul parallèle

On considère la fonction suivante qui permet de calculer la moyenne d'un échantillon de taille r simulée selon une loi normale de paramètre mean et sd:

```
myfun <- function(r, mean = 0, sd = 1) {
  mean(rnorm(r, mean = mean, sd = sd))
}
```

On souhaite répéter 100 fois cette fonction :

- 25 fois quand r vaut 10
- 25 fois quand r vaut 1000
- 25 fois quand r vaut 100000
- 25 fois quand r vaut 10000000

avec mean=5 et sd=10.

Syntaxe dans le cas non parallèle

Pour répondre à la problématique posée, on va d'abord utiliser la fonction sapply() qui permet de répondre au problème de façon séquentielle. Autrement dit, elle va appliquer la fonction myfun() itération après itération sur les différentes valeurs de r qu'on donne dans le 1er argument de la fonction sapply(). D'abord, on créé le vecteur contenant les valeurs de r et on prépare aussi l'objet qui va contenir les résultats.

```
r_values <- rep(c(10, 1000, 100000, 10000000), each = 25)
resultats <- data.frame(r_values = factor(r_values))
```

On lance la fonction sapply() et on regarde le temps de calcul :

```
system.time(
  resultats$res_non_par <- sapply(r_values, FUN = myfun,
```

```
                mean = 5, sd = 10) # options de la fonction myfun
)
```

```
##    user  system elapsed
##  18.276   1.072  19.350
```

Syntaxe dans le cas parallèle

Pour exécuter la fonction précédente dans le cas parallèle, la syntaxe est la suivante :

```
P <- 4  # définir le nombre de coeurs
cl <- makeCluster(P) # réserve 4 coeurs - début du calcul
system.time(
 res_par <- clusterApply(cl, r_values, fun = myfun, # évalue myfun sur r_values
                  mean = 5, sd = 10)  # options de myfun
)
```

```
##    user  system elapsed
##   0.028   0.004   7.838
```

```
stopCluster(cl) # libère 4 coeurs - fin du calcul
```

La syntaxe est donc pratiquement la même que dans le cas non parallèle. Il suffit simplement d'utiliser la fonction makeCluster() au début pour réserver le nombre de coeurs nécessaires et la fonction stopCluster() à la fin pour libérer les coeurs. De même, c'est la fonction clusterApply() qui permet de répartir la fonction myfun() vers les différents coeurs.

Explication et application des fonctions "apply"

Il existe des versions parallélisées de ces fonctions. Celles-ci sont nommées :

- parLapply()
- parSapply()
- parApply(
- clusterMap()

Ces fonctions font appel à la fonction clusterApply(). Elles semblent en général un peu plus longue en temps de calcul mais permettent de simplifier la syntaxe de sortie (fonction parSapply()) ou d'utiliser des arguments différents en fonction de l'itération (fonction clusterMap()).

Exemple:

Nous avons une liste contenant quatre grands vecteurs. Pour chacun des quatre grands vecteurs, calculons la moyenne. Ci-dessous, le calcul de la moyenne en utilisant une approche classique de lapply:

```
data <- 1:1e9
data_list <- list("1" = data,
                  "2" = data,
                  "3" = data,
                  "4" = data)
time_benchmark <- system.time(
  lapply(data_list, mean)
)
```

Voici le benchmark de ce bout de code:

```
time_benchmark
```

```
##    user  system elapsed
##  17.325   0.000  17.326
```

Ce calcul a pris à peu près 17 secondes. Les quatre calculs effectués ci-dessus sont totalement indépendants. C'est à dire qu'ils ne dépendent pas les uns des autres. La moyenne de l'élément 1 ne dépend en aucun cas de la moyenne de l'élément 2. Par défaut, la plupart des fonctions de R s'exécutent sur un seul coeur de traitement. Avec les processeurs multicœurs de la plupart des systèmes actuels, le potentiel de réduction du temps d'exécution en divisant simplement les tâches sur plusieurs cœurs est très important. Ci-dessous, le même calcul est implémenté en utilisant l'équivalent multi-core de lapply (parLapply) de la bibliothèque parallèle:

```
library(parallel)
cl <- parallel::makeCluster(detectCores())
time_parallel <- system.time(
  parallel::parLapply(cl,
                      data_list,
                      mean)
)# Close cluster
parallel::stopCluster(cl)
```

En ajoutant trois lignes de code, le temps de calcul a été réduit de près de 75%! C'est l'avantage des processeurs multicœurs modernes. Avec un processeur quadricœur et on exécute quatre calculs sur quatre cœurs au lieu d'un, ce qui le rend quatre fois plus rapide.

Autres packages de calcul parallèle

- snowFT : ce package permet de gérer le choix des graines de simulation de façon optimale à l'intérieur de chaque coeur. Nous en présenterons un exemple à la fin de ce chapitre car son utilisation semble prometteuse.
- foreach : ce package permet de faire des boucles de type for en utilisant une syntaxe légèrement différente et ceci peut se faire en calcul parallèle. Ce package est en général couplé avec le package

Un cas d'utilisation, le package Foreach

La fonction foreach fonctionne d'une manière très similaire à une boucle conventionnelle, mais en plus de l'index, elle a besoin d'informations sur la manière de structurer la sortie et sur les bibliothèques qui doivent être accessibles dans la boucle multicœur. Comme notre calcul des moyennes est effectué à l'aide des fonctions de base R, il n'est pas nécessaire de transmettre des packages à foreach.

Exemple:

```
library(doParallel)
library(parallel)
library(foreach)
cl <- parallel::makeCluster(detectCores())
doParallel::registerDoParallel(cl)time_foreach <- system.time({
  r <- foreach::foreach(i = 1:length(data_list),
                        .combine = rbind) %dopar% {
    mean(data_list[[i]])
  }
})
time_foreach[3]
parallel::stopCluster(cl)
```

Génération de nombres pseudo-aléatoires

Dans de nombreux cas, les calculs à paralléliser font intervenir le générateur de nombres pseudo-aléatoires, au travers de nombreuses fonctions comme sample, rnorm, runif,... C'est bien sûr le cas des implémentations du boostrap, des méthodes Monte-Carlo, des méthodes itératives avec point de départ (k-means par exemple, méthodes EM), etc. Techniquement parlant, le générateur réalise le calcul des éléments d'une suite mathématique qui ont des propriétés proche du hasard. Cette suite doit cependant être initialisée à partir d'un graine (ou seed): il s'agit de l'objet .Random.seed (un vecteur d'entier, pour les curieux) que l'on peut initialiser avec la commande set.seed comme suit:

```
set.seed(63)
rnorm(1)
1.324112
set.seed(63) #reproductibilité
rnorm(1)  1.324112
rnorm(1)  -1.873185
```

Dans le cas parallèle, chaque processus, père ou fils, doit proposer des séquences de nombres indépendantes... il faut donc garantir que les graines ne sont pas les même sur chaque processus! Le package parallel propose une solutionrobuste: le générateur dit « L'Ecuyer (1999) » est disponible via la commande RNGkind("L'Ecuyer-CMRG") et il présente les bonnes propriétés pour la génération en parallèle. Pour s'en servir, la procédure sera la suivante en fonction des approches:

- 1. ajouter clusterSetRNGStream(cl) (cette fonction contient un appel à RNGkind("L'Ecuyer-CMRG")) après tout appel à makeCluster;
- 2. laisser l'option mc.set.seed=TRUE (valeur par défaut) lors de l'appel à mclapply ou mcparallel.

Un exemple ci-dessous qui permet d'estimer la variabilité des coefficients d'un modèle linéaire par la méthode bootstrap, en parallèle avec parLapply:

```
n <- 1000
nb.simu <- 1000
x <- rnorm(n,0,1)
y <- rnorm(n,1+2*x,2)
data <- data.frame(x,y)
cl <- makeCluster(4, type="FORK")
clusterSetRNGStream(cl)
bootstrap.coef <- parLapply(cl, 1:nb.simu, fun=function(i) {
  bootstrap.resample <- data[sample(n,n,replace=T),]
  lm(y~x,bootstrap.resample)$coef
  })
stopCluster(cl)
```

Conclusion

Le package parallel de R permet de faire du calcul parallel dans un language de haut niveau. Néaenmoins, il est necessaire d'avoir quelques notions/connaissances en informatique theorique pour la compréhension global de ce package. Compte tenu de ce paramètre, ce document ne couvre pas l'ensemble des fonctionnalité qu'offre de package, il a pour but de faire une petite présentation/introduction sur la parallelisation de code qui est une autre manière programmer et qui n'est pas intuitif au départ. C'est pourquoi, ce doculent présente quelques explications simples sur la notions de parallelisation avec quelques cas d'utilisation sous le langage R avec ce package parallel.

Bibliographie

- Calcul en parallèle sur CPU avec R - Sophie Baillargeon, Université Laval
- Calcul parallèle avec R - Vincent Miel & Violaine Louvet
- Introduction to parallel computing with R - Hana Sevcikova, University of Washington
- Quick Intro to Parallel Computing in R - Matt Jones
- Getting Started With Parallel Programming In R - Jens Moll-Elsborg