R avancé

masedki.github.io

Organisation

Objectifs de la formation

L'objectif du cours est d'apprendre à

- écrire un code *efficace*
- mener une étude de simulation *reproductible*
- diffuser ses codes

On ne parlera pas des aspects suivants:

- visualisation
- Rshiny
- Rmarkdown

Objectifs de la formation

Le plan de la formation est

- Profilage du code et de la mémoire
- Gestion d'un projet pour une simulation
- Parallélisation de code (Forking and Sockets)
- **Optionnel** : appel d'un code C/C++ depuis R
- **Optionnel** : utilisation de la parallélisation et d'un code C/C++ par le même programme R
- **Optionnel** : polymorphisme sous R et C++

Utilisation du serveur

Pour ce cours, il faut travailler sur le serveur oxygene3. Vous pouvez vous connecter à partir de l'url suivante:

- http://cesp-oxygene3.vjf.inserm.fr:8787/

Nous avons 10 comptes pour la formation

- ID : et1 à et10
- mp : user$432

Ces comptes ne seront valables que durant les 2 jours de formation.

Profilage de code et de mémoire

Développer un code R efficace

Étape pour développer un code

1. Écrire un script basique qui fonctionne
   - utiliser des choses simples comme des boucles
   - créer plusieurs (petites) fonctions
   - utiliser des règles de structure du code
2. Vérifier que votre programme gère les exceptions
3. Amélioration du code

Développer un code R efficace

Étape pour développer un code

1. Écrire un script basique qui fonctionne
   - utiliser des choses simples comme des boucles
   - créer plusieurs (petites) fonctions
   - utiliser des règles de structure du code
2. Vérifier que votre programme gère les exceptions
3. Amélioration du code

Pour réduire le temps de calcul, vous devez détecter les parties de votre programme qui sont les plus longues. Pour cela, on utilise des outils d'analyse de performance.

Développer un code R efficace

Étape pour développer un code

1. Écrire un script basique qui fonctionne
   - utiliser des choses simples comme des boucles
   - créer plusieurs (petites) fonctions
   - utiliser des règles de structure du code
2. Vérifier que votre programme gère les exceptions
3. Amélioration du code

Pour réduire le temps de calcul, vous devez détecter les parties de votre programme qui sont les plus longues. Pour cela, on utilise des outils d'analyse de performance.

Une amélioration du code alterne entre

- détection des parties du programme à optimiser
- utilisation de méthodes de réduction du temps (ou de la mémoire)

Règles de bonnes pratiques

- On évite l'utilisation des points et des accents
- On utilise des noms explicites
- Un nom de classe commence par une majuscule
- Un nom de variable ou de fonction commence par une minuscule
- Un nom de variables termine par _type
- Le code est indenté (CTRL + A, CTRL + I)

Exemple

Ne pas utiliser

```
MaRégression <- function(x,y){
estimateurs <- solve(t(x)%*%x)%*%t(x)%*%y
residus <- y - x %*% estimateurs
list(es=estimateurs, re=residus)
}
```

mais plutôt

```
maRegression <- function(x_mat, y_num){
  thetahat_num <-
    solve(t(x_mat) %*% x_mat) %*% t(x_mat) %*% y_num
  thetahat_num <- as.numeric(thetahat_num)
  residus_num <- y_num - x_mat %*% thetahat_num
  list(thetahat = thetahat_num,
       residus = residus_num)
}
```

Utilisation d'un projet

Il est important de séparer votre programme en plusieurs scripts (donc plusieurs fichiers). Il faut (au moins):

- un fichier où sont définies les fonctions
- un fichier qui teste les fonctions
- un fichier qui génère et/ou importe les données, utilise les fonctions et sauvegarde les résultats
- un fichier qui lit les résultats et s'occupe de leur analyse.

Il est donc plus simple d'utiliser un Projet R, qui se crée par:

`File > New Projet > New Directory`

Utilisation d'un projet

On considère l'exemple suivant.

On génère n observations iid avec

$$X \sim \mathcal{N}(0, 1)$$

et

$$Y = 10X + \varepsilon$$

où

$$\varepsilon \mid X = x \sim \mathcal{N}(0, 1).$$

On crée une fonction qui génère l'échantillon sous forme d'un data.frame et une fonction qui visualise le nuage de points en ajoutant la droite de régression obtenue en minimisant la perte quadratique. L'expérience doit être reproductible.

Utilisation d'un projet

Voici une façon de code ce problème pour la génération des données

```
robs <- function(n_int, beta_num = 10){
  x_num <- eps_num <- y_num <- rep(0, n_int)
  for (i in 1:n_int){
    x_num[i] <- rnorm(1)
    eps_num[i] <- rnorm(1)
    y_num[i] <- x_num[i] * beta_num + eps_num[i]
  }
  dat <- data.frame(
    x = x_num,
    y = y_num
  )
}
```

Utilisation d'un projet

Voici une façon de code ce problème pour la visualisation des données

```
visu <- function(ech_df){
  plot(y ~ x, data = ech_df)
  res.mco <- lm(y ~ x, data = ech_df)
  abline(res.mco, col = "red")
}
```

Profilage de code

Pour effectuer le profilage du code, on peut utiliser le package `profvis`.

```
p <- profvis({
  set.seed(123)
  obs_df <- robs(10^3)
  visu(obs_df)
})
p
htmlwidgets::saveWidget(p, "~/enseignements/profile.html")
```

Reprendre l'exemple précédent avec $n = 10^2$, $n = 10^3$ et $n = 10^4$.

Améliorations d'un code R

Voici quelques pratiques qui ralentissent un code R

Améliorations d'un code R

Voici quelques pratiques qui ralentissent un code R

- utilisation des boucles
- changement dans la dimension des objets
- modification d'une valeur dans un data.frame

Améliorations d'un code R

Voici quelques façon de réduire le temps d'exécution d'un code R

1. Faire seulement ce qui est nécessaire.
2. Utiliser des fonctions optimisées.
3. Utiliser le calcul vectoriel.
4. Éviter les allocations/manipulations de mémoire inutiles.
5. Utiliser la compilation bytecode.
6. Utiliser le calcul parallèle.
7. Recoder des parties en C/C++.

Améliorations d'un code R

Notre script fait-il seulement ce qui est nécessaire?

Améliorations d'un code R

Notre script fait-il seulement ce qui est nécessaire?

Non, la fonction `lm` fait plein de choses qui ne nous servent pas, car on veut un estimateur des paramètres (et c'est tout).

On peut donc proposer une alternative.

On suit les étapes suivantes:

- on développe une nouvelle fonction
- on vérifie qu'elle produit le même résultat (`all.equal`)
- on fait le profilage du code

Améliorations d'un code R

Voici une proposition

```
visu2 <- function(ech_df){
  plot(y ~ x, data = ech_df)
  coeff_num <- .lm.fit(cbind(1, ech_df$x),
                       ech_df$y)$coefficients
  abline(a = coeff_num[1],
         b = coeff_num[2], col = "red")
}
```

On effectue donc le profilage du nouveau code

```
p2 <- profvis({
  set.seed(123)
  obs_df <- robs(10^4)
  visu2(obs_df)
})
p2
```

Améliorations d'un code R

On souhaite maintenant optimiser la partie de génération des données (à titre d'exemple car dans une vraie optique d'optimisation cette partie est négligeable).

On considère trois cas

- le cas actuel: boucle sur objets pré-définis
- boucle sur objets de taille croissante
- utilisation du calcul vectoriel

Améliorations d'un code R

Voici la version avec objets de taille croissante

```
robsSeq <- function(n_int, beta_num = 10){
  x_num <- eps_num <- y_num <- NULL
  for (i in 1:n_int){
    x_num <- c(x_num, rnorm(1))
    eps_num <- c(eps_num, rnorm(1))
    y_num <- c(y_num, x_num[i] * beta_num + eps_num[i])
  }
  dat <- data.frame(
    x = x_num,
    y = y_num
  )
}
```

Améliorations d'un code R

On vérifie que les fonctions `robs` et `robsSeq` produisent les résultats souhaités puis on analyse la fonction `robsSeq`.

Améliorations d'un code R

On vérifie que les fonctions `robs` et `robsSeq` produisent les résultats souhaités puis on analyse la fonction `robsSeq`.

Pour comparer ces fonctions, il faudrait pouvoir les lancer plusieurs fois et comparer la distribution de leur temps de calcul. On peut faire ça avec le package `bench` et le script

```
mbm <- mark(loops = robs(10^3), seq = robsSeq(10^3))
```

Améliorations d'un code R

On vérifie que les fonctions `robs` et `robsSeq` produisent les résultats souhaités puis on analyse la fonction `robsSeq`.

Pour comparer ces fonctions, il faudrait pouvoir les lancer plusieurs fois et comparer la distribution de leur temps de calcul. On peut faire ça avec le package `bench` et le script

```
mbm <- mark(loops = robs(10^3), seq = robsSeq(10^3))
```

. . . mais dans ce cas, la sortie n'est plus la même. . .

Améliorations d'un code R

Il faut utiliser le script

```
mbm <- mark(loops = {
  set.seed(123)
  robs(10^3)
  },
  seq = {
    set.seed(123)
    robsSeq(10^3)
    })
summary(mbm)
plot(mbm)
```

Améliorations d'un code R

Voici la version avec vectorisation

```
robsVec <- function(n_int, beta_num = 10){
  x_num <- rnorm(n_int)
  dat <- data.frame(
    x = x_num,
    y = x_num * beta_num + rnorm(n_int)
  )
}
```

On compare maintenant les approches

Améliorations d'un code R

Il faut utiliser le script

```
mbm <- mark(loops = {
  set.seed(123)
  robs(10^3)
},
seq = {
  set.seed(123)
  robsSeq(10^3)
},
vec = {
  set.seed(123)
  robsVec(10^3)
},
check=FALSE
)
plot(mbm)
summary(mbm)
```

Profilage du code avancé

Profilage du code

Le profilage peut se faire à un niveau macroscopique pour détecter les parties les plus longues d'un programme (`profvis`).

Le profilage peut se faire à un niveau microscopique pour comparer différentes façons d'effectuer une tâche simple (`mark`).

Les résultats (temps de calculs) peuvent être différents en fonction de la taille d'échantillon (ce n'est pas toujours la même méthode qui est optimale pour différentes tailles d'échantillon). On peut donc effectuer une comparaison dans différents cadres.

Profilage du code

Si on a plusieurs situations à investiguer alors il faut faire

- un (ou plusieurs) fichier(s) où les fonctions sont définies
- un fichier faisant le microbenchmark pour les différentes situations

Le second fichier respecte les étapes suivantes

1. nettoyage de la mémoire et importation des fonctions (`source`)
2. definition des conditions de test
3. tests (`mark`)
4. visualisation des resultats (peut être fait dans un autre fichier)

Profilage du code

On considère deux approches permettant de centrer une matrice en colonnes (voir exercice 1 TP 1).

On souhaite comparer l'approche basée sur `vapply` de celle basée sur `lapply` pour $n = 10^2$, 10^3, 10^4, 10^5 et 10^6.

Gestion d'une simulation

Gestion d'une simulation

Une simulation doit être faite avec les fichiers suivants:

- un (ou plusieurs) fichier(s) où les fonctions sont définies
- un fichier faisant la simulation (c'est ici qu'on fixe la graine)
- un fichier analysant les résultats (il n'y a plus de simulation ici)

Gestion d'une simulation

Voici quelques commandes utiles:

- `rm(list=ls())`: pour vider l'environement
- `source()`: pour charger un script (surtout pour un ensemble de fonctions)
- `set.seed()`: pour fixer la graine (gestion de l'aléatoire)
- `save()`/`load()`: pour sauvegarder/charger un ou plusieurs objets R à partir d'un fichier .rda ou .Rdata.
- `pdf()`/`png()`: pour sauvegarder un graphique. Cette fonction doit être suivie de `dev.off()`.

Gestion d'une simulation

On souhaite illustrer la loi de grands nombres.

Pour cela, on considére un échantillon de taille n issue d'une loi de Bernoulli de paramètre p et on calcule la moyenne empirique.

Gestion d'une analyse de données

Gestion d'une analyse de données

Une analyse de données doit être faite avec les fichiers suivants:

- un fichier faisant le traitement des données et sauvegardant les données "nettoyées"
- un fichier faisant les statistiques descriptives
- un fichier faisant les études plus avancées

On illustre cela sur les données de qualité d'air en conservant uniquement les observations sans valeurs manquantes et les variables "Ozone", "Temp", "Month".

Parallélisation :Généralités

Parallélisation

Une façon importe de réduire le temps de calcul est la parallélisation de code.

Parallélisation

Une façon importe de réduire le temps de calcul est la parallélisation de code.

Dans sa version la plus simple, cela s'organise comme:

- Séparer un long programme en plusieurs petits blocs de calculs **indépendants**
- Effectuer ces blocs de calculs sur plusieurs CPU en même temps (donc en parallèle)

Parallélisation

Une façon importe de réduire le temps de calcul est la parallélisation de code.

Dans sa version la plus simple, cela s'organise comme:

- Séparer un long programme en plusieurs petits blocs de calculs **indépendants**
- Effectuer ces blocs de calculs sur plusieurs CPU en même temps (donc en parallèle)
- Regrouper les résultats quand **tous** les calculs sont terminés

Parallélisation

Une façon importe de réduire le temps de calcul est la parallélisation de code.

Dans sa version la plus simple, cela s'organise comme:

- Séparer un long programme en plusieurs petits blocs de calculs **indépendants**
- Effectuer ces blocs de calculs sur plusieurs CPU en même temps (donc en parallèle)
- Regrouper les résultats quand **tous** les calculs sont terminés
- Le point central est que les différents calculs (**chunks**) sont indépendants et ne communiquent pas entre eux.

Parallélisation

Une façon importe de réduire le temps de calcul est la parallélisation de code.

Dans sa version la plus simple, cela s'organise comme:

- Séparer un long programme en plusieurs petits blocs de calculs **indépendants**
- Effectuer ces blocs de calculs sur plusieurs CPU en même temps (donc en parallèle)
- Regrouper les résultats quand **tous** les calculs sont terminés
- Le point central est que les différents calculs (**chunks**) sont indépendants et ne communiquent pas entre eux.

Les différents CPU utilisés peuvent être sur le même ordinateur ou sur un (ou plusieurs) serveurs.

Exemple: évaluer la même fonction sur différents jeux de données simulés dans des calculs bootstrap (ou pour une simulation).

Terminologie

- Un *CPU* (central processing unit) est l'unité de calcul principale d'un système. Des fois le terme CPU fait référence au processeur, des fois un a cœur du processeur. Aujourd'hui, les systèmes ont souvent (au moins) un processeur avec plusieurs cœurs.
- Un *cluster* est un ensemble de machines qui sont toutes connectées et partage les ressources (comme un ordinateur géant).

Terminologie

- Un *CPU* (central processing unit) est l'unité de calcul principale d'un système. Des fois le terme CPU fait référence au processeur, des fois un a cœur du processeur. Aujourd'hui, les systèmes ont souvent (au moins) un processeur avec plusieurs cœurs.
- Un *cluster* est un ensemble de machines qui sont toutes connectées et partage les ressources (comme un ordinateur géant).
- Le *master* est le système qui tourne sur un cœur et qui contrôle le cluster (votre session)

Terminologie

- Un *CPU* (central processing unit) est l'unité de calcul principale d'un système. Des fois le terme CPU fait référence au processeur, des fois un a cœur du processeur. Aujourd'hui, les systèmes ont souvent (au moins) un processeur avec plusieurs cœurs.
- Un *cluster* est un ensemble de machines qui sont toutes connectées et partage les ressources (comme un ordinateur géant).
- Le *master* est le système qui tourne sur un cœur et qui contrôle le cluster (votre session)
- Un *slave* ou *worker* est une machine qui effectue des calculs et répond aux demandes du *master*. Un *slave* tourne sur un cœur.

Parallélisation sous R

Plusieurs packages permettent de faire du calcul parallèle sous R: https://cran.r-project.org/web/views/HighPerformanceComputing.html.

Les packages les plus importants sont:

- Rmpi (low-level)
- parallel (R-level)
- snowFT (R-level)
- foreach/doMC (R-level)

Parallélisation sous R

Plusieurs packages permettent de faire du calcul parallèle sous R: https://cran.r-project.org/web/views/HighPerformanceComputing.html.

Les packages les plus importants sont:

- Rmpi (low-level)
- parallel (R-level)
- snowFT (R-level)
- foreach/doMC (R-level)

Nous allons nous concentrer sur le package **parallel**.

Ce package est dans la version de base de R.

Le package parallel

- Il fait partie du R-core de base depuis la version R 2.14.0 (31-Oct-2011).

Le package parallel

- Il fait partie du R-core de base depuis la version R 2.14.0 (31-Oct-2011).

- Il se base sur les package packages *multicore* (Urbanek, 2009–2014) et *snow* (Tierney, Rossini, Sevcikova, 2003–present) et les généralise.

Le package parallel

- Il fait partie du R-core de base depuis la version R 2.14.0 (31-Oct-2011).
- Il se base sur les package packages *multicore* (Urbanek, 2009–2014) et *snow* (Tierney, Rossini, Sevcikova, 2003–present) et les généralise.
- Il gère la génération de nombres aléatoires.

Le package parallel

- Il fait partie du R-core de base depuis la version R 2.14.0 (31-Oct-2011).
- Il se base sur les package packages *multicore* (Urbanek, 2009–2014) et *snow* (Tierney, Rossini, Sevcikova, 2003–present) et les généralise.
- Il gère la génération de nombres aléatoires.
- Il permet de mobiliser un très grand nombre de CPU en même temps à condition que les calculs des différents CPU soient *indépendants*.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.
3. Découper le long programme en M *chunks* de taille équivalente et envoyer ces *chunks* (et les fonctions associées) aux *slaves*.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.
3. Découper le long programme en M *chunks* de taille équivalente et envoyer ces *chunks* (et les fonctions associées) aux *slaves*.
4. Attendre que **tous** les *slaves* finissent les calculs et récupérer les résultats.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.
3. Découper le long programme en M *chunks* de taille équivalente et envoyer ces *chunks* (et les fonctions associées) aux *slaves*.
4. Attendre que **tous** les *slaves* finissent les calculs et récupérer les résultats.
5. Répéter les étapes (2-4) pour d'autres tâches.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.
3. Découper le long programme en M *chunks* de taille équivalente et envoyer ces *chunks* (et les fonctions associées) aux *slaves*.
4. Attendre que **tous** les *slaves* finissent les calculs et récupérer les résultats.
5. Répéter les étapes (2-4) pour d'autres tâches.
6. Fermer les processus des *slaves*.

Le package parallel

Le modèle de calcul est

1. Démarrer M *slaves* et faire les initialisations nécessaires.
2. Envoyer toutes les données nécessaires pour chaque tâche effectuée par chaque slaves.
3. Découper le long programme en M *chunks* de taille équivalente et envoyer ces *chunks* (et les fonctions associées) aux *slaves*.
4. Attendre que **tous** les *slaves* finissent les calculs et récupérer les résultats.
5. Répéter les étapes (2-4) pour d'autres tâches.
6. Fermer les processus des *slaves*.

Les fonctions de bases sont:

- parApply (version parallel de apply).
- parLapply (version parallel de lapply).
- parSapply (version parallel de sapply).

La méthode sockets

Le package parallel sous windows: méthode sockets

Chargement du package

```
require(parallel)
```

```
Loading required package: parallel
```

Détection du nombre de coeurs

```
detectCores()
```

```
[1] 8
```

Pour cette méthode, il faut créer une connexion entre le R du master et le cœur de chaque slave. Pour cela, on utilise la fonction **makeCluster** (ou **makePSOCKcluster**, ou **makeForkCluster**)

```
coeurs <- detectCores()
grappe <- makeCluster(coeurs - 1)
grappe
```

```
socket cluster with 7 nodes on host 'localhost'
```

Le package parallel sous windows: méthode sockets

La première fois, on vous demandera sûrement de valider la procédure.

Le package parallel sous windows: méthode sockets

La première fois, on vous demandera sûrement de valider la procédure.

Ici j'utilise 7 de mes 8 cœurs (le dernier reste disponible pour mes emails, rédiger mon document tex. . .)

Le package parallel sous windows: méthode sockets

La première fois, on vous demandera sûrement de valider la procédure.

Ici j'utilise 7 de mes 8 cœurs (le dernier reste disponible pour mes emails, rédiger mon document tex. . .)

Comparons les fonctions **myMeanCol.single**, qui utilise **sapply**, avec sa version parallélisée basée sur **parSapply**.

Le package parallel sous windows: méthode sockets

La première fois, on vous demandera sûrement de valider la procédure.

Ici j'utilise 7 de mes 8 cœurs (le dernier reste disponible pour mes emails, rédiger mon document tex. . .)

Comparons les fonctions **myMeanCol.single**, qui utilise **sapply**, avec sa version parallélisée basée sur **parSapply**.

Le package parallel sous windows: méthode sockets

```
require(parallel)
myMeanCol.single <- function(data)
  sapply(data[, names(data) != "ID"], mean)

myMeanCol.multi <- function(grappe, data)
  parSapply(grappe, data[, names(data) != "ID"], mean)
```

Le package parallel sous windows: méthode sockets

```
require(bench)
```

```
Loading required package: bench
```

```
require(parallel)
coeurs <- detectCores()
grappe <- makeCluster(coeurs - 1)
grappe
```

```
socket cluster with 7 nodes on host 'localhost'
```

```
require(parallel)
set.seed(123)
n <- 10**3
d <- 1500
data <- cbind.data.frame(ID=paste("g", 1:n, sep = "."),
                         matrix(rnorm(n * d), n, d))

res.single <- myMeanCol.single(data)
res.multi <- myMeanCol.multi(grappe, data)
all.equal(res.single, res.multi)
```

```
[1] TRUE
```

```
res_micro <- mark(sapply   = myMeanCol.single(data),
                  parSapply  = myMeanCol.multi(grappe, data))
plot(res_micro)
```

Le package parallel sous windows: méthode sockets

On constate que le calcul parallèle ne réduit pas toujours le temps de calcul.

Le package parallel sous windows: méthode sockets

On constate que le calcul parallèle ne réduit pas toujours le temps de calcul.

Une fois les calculs terminés, il faut impérativement fermer les connexions avec la fonction *stopCluster*.

```
stopCluster(grappe)
```

Le package parallel sous windows: méthode sockets

Il faut faire attention à exporter les fonctions que l'on utilise

Le package parallel sous windows: méthode sockets

Il faut faire attention à exporter les fonctions que l'on utilise

```
grappe <- makeCluster(7)
w <- runif(100)
y <- replicate(10, runif(100), simplify = FALSE)
monCalc1 <- function(a, w) mean(a*w)
monCalc2 <- function(a, w) sd(a*w)
mesStat <- function(a, w) c(monCalc1(a,w), monCalc2(a,w))
parLapply(grappe, y, mesStat, w=w)
```

```
Error in checkForRemoteErrors(val) :   7 nodes produced errors;
 first error: impossible to find the function monProd1
```

Ainsi toutes les fonctions utilisées dans le master doivent être exportées sur chaque slave.

Le package parallel sous windows: méthode sockets

```
require(parallel)
grappe <- makeCluster(3)
w <- runif(100)
y <- replicate(3, runif(100), simplify = FALSE)
monCalc1 <- function(a, w) mean(a*w)
monCalc2 <- function(a, w) sd(a*w)
mesStat <- function(a, w) c(monCalc1(a,w), monCalc2(a,w))
clusterExport(grappe, varlist = c("monCalc1", "monCalc2"))
parLapply(cl = grappe, y, mesStat, w=w)
```

```
[[1]]
[1] 0.2491966 0.2325444

[[2]]
[1] 0.2516307 0.2179957

[[3]]
[1] 0.2396276 0.2066993
```

Le package parallel sous windows: méthode sockets

On doit libérer les ressources par la fonction *stopCluster*.

Le package parallel sous windows: méthode sockets

On doit libérer les ressources par la fonction *stopCluster*.

Cependant, si le calcul parallèle est dans une fonction, il est possible que la fonction soit arrêtée par une erreur et donc que la fonction *stopCluster* ne soit pas exécutée.

Le package parallel sous windows: méthode sockets

On doit libérer les ressources par la fonction *stopCluster*.

Cependant, si le calcul parallèle est dans une fonction, il est possible que la fonction soit arrêtée par une erreur et donc que la fonction *stopCluster* ne soit pas exécutée.

La fonction *on.exit* permet de réduire ce risque.

```
simulate = function(cores) {
  cl = makeCluster(cores)
  on.exit(stopCluster(cl))
  ## Do something
}
```

Le package parallel sous windows: méthode sockets

Profilage micro

```
rm(list=ls())
require(VarSelLCM)
require(parallel)
require(bench)
data("heart")
grappe <- makeCluster(4)
clusterExport(grappe, varlist = c("heart", "VarSelCluster"))
res=mark(test1=lapply(1:4, VarSelCluster, x=heart[,-13]),
         test2=parLapply(grappe,
                         1:4,
                         function(g) VarSelCluster(heart[,-13], g)),
         check = F)
stopCluster(grappe)
plot(res)
```

La méthode forking

Le package parallel sous linux: méthode forking

La méthode forking est plus simple. . . mais ne fonctionne pas sous windows.

Le package parallel sous linux: méthode forking

La méthode forking est plus simple. . . mais ne fonctionne pas sous windows.

Vous pouvez utiliser la fonction **mclapply** et spécifier le nombre de cœurs par l'option **mc.cores**.

Le package parallel sous linux: méthode forking

La méthode forking est plus simple. . . mais ne fonctionne pas sous windows.

Vous pouvez utiliser la fonction **mclapply** et spécifier le nombre de cœurs par l'option **mc.cores**.

```
require(parallel)
w <- runif(100)
y <- replicate(3, runif(100), simplify = FALSE)
monCalc1 <- function(a, w) mean(a*w)
monCalc2 <- function(a, w) sd(a*w)
mesStat <- function(a, w) c(monCalc1(a,w), monCalc2(a,w))
mclapply(y, mesStat, w=w, mc.cores = 3)
```

Le package parallel sous linux: méthode forking

Profilage micro

```
rm(list=ls())
require(parallel)
require(VarSelLCM)
data("heart")
T1 <- Sys.time()
test1 <- lapply(1:4, VarSelCluster, x=heart[,-13])
T2 <- Sys.time()
test2 <- mclapply(1:4,
                  function(g) VarSelCluster(heart[,-13], g),
                  mc.cores = 4)
T3 <- Sys.time()
difftime(T2, T1)
difftime(T3, T2)
```

Résumé

Méthodes de parallélisation

Il y a deux façons de faire de la parallélisation: **sockets** ou **forking**.

Ces fonctions sont un peu différentes

- La méthode sockets lance une nouvelle version de R sur chaque cœur. Techniquement, cette connexion est faite par réseautage (la même chose que si connectiez à un serveur à distance), mais la connexion se passe sur votre ordinateur. C'est pour cela que vous pouvez avoir des messages de votre ordinateur demandant d'autoriser R à faire des connexions internes.
- La méthode forking copie l'entière version de R du master sur les autres slaves.
- Il y a des avantages et inconvénients aux deux méthodes:

Méthodes de parallélisation

Sockets

- Pour: fonctionne sur tous les systèmes (windows y compris).
- Pour: un processus par nœud et donc on ne peut pas avoir de contamination.
- Contre: chaque processus est unique donc plus lent.
- Contre: des choses comme le chargement des packages doivent être faites sur chaque slave séparément. Les variables définies sur le master n'existe pas sur les slaves. Il faut donc les exporter explicitement.
- Contre: Plus compliqué à implémenter.

Méthodes de parallélisation

Forking

- Contre: ne marche que sur les systèmes POSIX (Mac, Linux, Unix, BSD).
- Contre: comme les processus sont dupliqués, il faut faire très attention à la génération des nombres aléatoires.
- Contre: si un slave produit une erreur, on ne peut rien récupérer.
- Pour: plus rapide que sockets.
- Pour: copie tout l'espace de travail sur chaque slave.
- Pour: très simple à implémenter.

Je recommande d'utiliser le forking si vous n'êtes pas sous windows (surtout si vous utilisez du code C++).

Infos

SSH

Pour se connecter au serveur en ssh

- sous linux: dans un terminal ssh login@clust-n2.domensai.ecole (attention ne marche que depuis l'école)
- sous windowns: il faut utiliser Putty et mettre clust-n2.domensai.ecole dans le Host Name.

SSH

Pour se connecter au serveur en ssh

- sous linux: dans un terminal ssh login@clust-n2.domensai.ecole (attention ne marche que depuis l'école)
- sous windowns: il faut utiliser Putty et mettre clust-n2.domensai.ecole dans le Host Name.

Pour voir les processus en cours: htop ou top

Pour arrêter un de vos processus : top puis k puis ID process Pour arrêter tous vos processus : `pkill -u login`

Pour lancer un script: `Rscript myscript.R &`

Pour lancer détacher un processus: `screen -S nom` puis lancer le script puis CTRL + A + D.