TP3: Parallélisation de code

L'ensemble des séances se fera sur le serveur auquel vous pouvez vous connecter par l'url suivante

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

Créer un répertoire nommé Ravance. Dans ce répertoire, créer un répertoire `TP3parallelisation`.

Exercise 1 (Échauffement)

1. Créer une fonction qui génère une matrice de taille $n \times d$ composée de réalisations indépendantes de $\mathcal{N}(0, 1)$
2. Générer $N = 20$ matrices aléatoires en utilisant la fonction précédente avec les arguments $n = 10^4$ and $d = 5$ (attention à la reproductibilité de votre expérience).
3. Trouver une façon efficace de calculer la moyenne de chaque colonne, pour chaque échantillon, sans utiliser de parallélisation. Le résultat est stocké dans une matrice de taille $N \times d$.
4. Proposer une alternative qui utilise le calcul parallèle avec la fonction `mclapply`.
5. Vérifier le nombre de CPU disponibles avec la fonction `detectCores`.
6. Effectuer le profilage de code par la fonction `system.time` des approches non-parallélisées et parallélisées. Vous pouvez considérer une parallélisation sur 2, 10 et 20 CPUs.
7. On souhaite appliquer la fonction `summary` sur chaque échantillon. Comparer l'approche non-parallélisée utilisant la fonction `lapply` et l'approche parallélisée utilisant la fonction `mclapply` avec 2, 5 et 10 CPUs. Comparer les approches avec $N = 200$, $n = 10^4$ et $d = 5$. Conclure

Exercise 2 (Profilage de code et calcul parallèle)

Nous disposons de données issues d'une enquête. Pour anonymiser les données, nous avons cachés toutes les informations relatives à l'identité des personnes interrogées, y compris le nom de l'agent réalisant l'enquête (attention, c'est une mauvaise pratique; on préfere savoir le nom de l'agent afin de pouvoir ajuster l'effet de l'agent sur les réponses). Nous souhaitons ajuster un modèle de régression linéaire simple afin d'expliquer le temps d'ancienneté dans l'entreprise en fonction du salaire. Après la collecte des données, nous avons su qu'un des quatre agents avait fait une terrible erreur. Il avait presque tout le temps arrondi le salaire (par exemple \$77K a été enregistré \$80K). Comme nous ne pouvons pas détecter quels salaires ont été arrondis, nous pouvons créer des données artificielles issues du même procédé et ajuster un modèle qui permet de déterminer à quel point le coefficient estimé est biaisé.

```
save <- numeric()
for (i in 1:reps) {
   x <- rnorm(n)
   y <- beta0 + beta1*x + rnorm(n)
   # Add rounding. Since we're dealing with standardized data, round to 2 decimals.
   badinterviewer <- sample(1:n, .25*n, replace = FALSE)
   x[badinterviewer] <- ceiling(x[badinterviewer])
   coef <- lm(y ~ x)$coef[2]
   save <- c(save, coef)
 }
 save <- data.frame(save)
```

1. Effectuer le profilage de code du script précédent avec $reps = 10000$, $n = 100$, $beta0 = 2$ et $beta1 = .7$.

2. On peut clairement voir que l'appel à la fonction `lm` est la partie du code la plus longue. Cette fonction est optimisée mais elle calcule de nombreuses quantités que nous n'utilisons pas ici (on ne souhaite avoir que l'estimateur du coefficient de régression associé à la variable salaire). Nous avons deux options:
   - Écrire votre propre solver qui retourne l'estimateur MCO.
   - Utiliser la fonction interne R, `.lm.fit`, qui prend comme argument uniquement une matrice de design (dont la constante est déjà présente) et une variable réponse.

   Comparer les trois approches.
3. Pouvez-vous améliorer le code général sans parallélisation?
4. Proposer une version parallélisée qui optimise le script. Dans la première version, l'échantillonnage aléatoire peut-être fait sur différents CPUs. Dans une seconde version, tous les échantillonnages aléatoires sont faits avant la parallélisation. Comparer les trois approches optimisées (temps de calculs et reproductibilité).

Exercise 3 (Forking vs. Sockets)

On considére le modèle linéaire

$$Y = \alpha + X^\top \beta + \varepsilon$$

où $X \in \mathbb{R}^2$ est composé de deux réalisations indépendantes d'une loi normale standard et où le bruit ε est aussi une réalisation indépendante d'une loi normale standard. Pour la suite, on considère $\alpha = 2$ et $\beta = (1, 1)^\top$.

1. Créer une fonction qui génère un échantillon à partir du modèle décrit précédemment. La fonction doit retourner un `data.frame` dont les colonnes sont nommées Y, $X1$ et $X2$.
2. Créer une fonction qui divise aléatoirement un échantillon en deux parties: l'échantillon d'apprentissage et l'échantillon test. L'échantillon d'apprentissage est utilisé pour obtenir l'OLS tandis que l'échantillon test est utilisé pour évaluer l'erreur quadratique moyenne (MSE) pour la prédiction. La fonction n'utilise pas de parallélisation.
3. Générer un échantillon de taille $n = 1000$. Avec la fonction `lapply`, effectuer $R = 200$ évaluations du MSE pour la prédiction.
4. Proposer une version parallélisée pour évaluer cet MSE en utilisant la méthode `forking` lorsqu'un seul échantillon est observé.
5. Comparer les deux approches par la fonction `system.time`.
6. L'approche parallélisée effectue des générations aléatoires sur différents CPUs. Proposer une solution pour éviter ce problème. Comparer les performances de votre nouvelle approche.
7. Proposer une version parallélisée, basée sur le `forking`, qui évalue le MSE pour la prédiction lorsque N échantillons sont observés.
8. On souhaite maintenant implémenter une version parallélisée basée sur le `sockets`. On ne s'intéresse que au cas où un seul échantillon est observé. Pour cela, respecter les étapes suivantes
   (a) Démarrer un cluster avec d CPU en utilisant la fonction `makeCluster`.
   (b) Exécuter toutes les exportations nécessaires sur les différents CPU (e.g. exportation des données, chargement de package).
   (c) Utiliser la fonction par(L-S)apply pour remplacer les appels à la fonction (l-s)apply.
   (d) Fermer proprement la connection.

Exercise 4 (Génération aléatoire et calculs parallèle.)

Sous R, il y a principalement deux objets pour gérer les graines de générations aléatoires: `set.seed()` et `.Random.seed`. Pour la majorité des problèmes, il est suffisant d'utiliser la fonction `set.seed()` pour obtenir une expérience reproductible. Cette fonction fournit un entier comme graine. On l'utilise comme suit:

```
set.seed(1991)
runif(2)
runif(2)
set.seed(1991)
runif(2) # exactly the same random numbers as before
```

Le second objet, `.Random.seed`, permet de sauvegarder et de restaurer l'état du générateur de nombres aléatoires (RNG). En réalité `.Random.seed` est un vecteur d'entier. Son premier élément spécifie le type du RNG et le type du générateur gaussien. Par exemple, le premier élément indique que l'on utilise la méthode "L'Ecuyer-CMRG" pour le RNG et l'approche "Box-Muller" pour la génération de variables gaussiennes. Les autres éléments de `.Random.seed` stockent la graine aléatoire actuelle.

On utilise cet objet de façon particulière. Il peut être sauvegardé sans définir explicitement la graine (on ne doit pas nécessairement utiliser la fonction `set.seed()`).

```
seed <- .Random.seed
runif(2)
runif(2)
.Random.seed <- seed
runif(2)
```

L'objet `.Random.seed` est définit dans l'environnement global. Cela peut causer des problèmes si vous définissez l'objet `.Random.seed` à l'intérieur d'une fonction sans tenir compte de l'environnement. Ainsi, changer cet objet dans une fonction par une simple affectation ne changera pas la graine (la valeur sera définie dans l'environnement d'exécution). Cette idée peut être utiliser pour sauvegarder la graine actuelle ou la définir à l'intérieur d'une fonction:

```
reproducible_runif <- function(seed = NULL) {
    if(is.null(seed)) {
        seed <- .Random.seed
    } else {
        assign(x = ".Random.seed", value = seed, envir = .GlobalEnv)
    }
    return(list(x = runif(1), seed = seed))
}
```

Ainsi, cette fonction retournera un nombre aléatoire et l'expérience sera reproductible:

```
r1 <- reproducible_runif()
r1$x
runif(10)
r2 <- reproducible_runif(seed = r1$seed) # use the seed from the initial call
r2$x # exactly the same as for r1
```

Cela devient plus complexe lorsqu'on utilise plusieurs CPU. Avant de donner plus de détails, considérons l'exemple suivant. Nous faisons un appel à la fonction `mclapply` qui génère un nombre aléatoire selon une loi uniforme à chaque itération. Cette fonction prend pour arguments: un vecteur de 10 éléments comme argument `X`, une fonction enveloppe autour de `runif(1)` qui ignore les éléments de X, un nombre de CPUs et aussi l'argument `mc.set.seed = FALSE`. On exécute le script et on remarque quelque chose d'étrange.

```
library(parallel)
rn1 <- unlist(
    mclapply(X = 1:10,
             FUN = function(x) runif(1),
             mc.cores = 2,
             mc.set.seed = FALSE)
)
```

Ce phénomène s'explique simplement: le même environnement de travail est chargé depuis le master sur les slaves. Cela veut dire que `.Random.seed` sera copié depuis le master et donc l'état de RNG sera le même sur chaque slave. Cela implique que la même séquence de nombres aléatoires sera générée sur chaque slave.

Une alternative est de définir différentes graines sur les slaves. On risque d'obtenir des nombres générés qui soient juste décalés (i.e., répétés périodiquement et donc corrélés dans le temps). Pour résoudre ce problème on utilise le RNG "L'Ecuyer-CMRG", qui a une longue période avec une petite graine. Pour définir le RND comme "L'Ecuyer-CMRG", on lance `RNGkind("L'Ecuyer-CMRG")`, et on change l'argument `mc.set.seed` de `mclapply` en `TRUE`:

```
RNGkind("L'Ecuyer-CMRG")

rng1 <- unlist(
    mclapply(X = 1:10,
             FUN = function(x) runif(1),
             mc.cores = 2,
             mc.set.seed = TRUE)
)

rng1
```

Les éléments sont maintenant différents. De plus, "L'Ecuyer-CMRG" utilise nextRNGStream() pour obtenir une graine suivante "non corrélée". Effectuons le même appel

```
rng2 <- unlist(
    mclapply(X = 1:10,
             FUN = function(x) runif(1),
             mc.cores = 2,
             mc.set.seed = TRUE)
)

rng2
identical(rng1, rng2)
```

Le second `rng2` est absolument identique à `rng1`. Cela s'explique par le fait que l'objet `.Random.seed` du master n'est pas affecté par les processus lancés sur les slaves. On obtiendra donc les mêmes nombres tant que l'objet `.Random.seed` ne sera pas changé (e.g. par une appel de type `runif(1)` ou `set.seed()` dans le master).

Note que si `mc.set.seed=TRUE`, mais que RNG est différent de "L'Ecuyer-CMRG", alors utiliser l'argument `set.seed()` ne donne pas lieu à une expérience reproductible.