Calcul en parallèle sur CPU avec R

en exploitant un ordinateur multi-coeurs, la grappe de serveurs de calcul du Département de mathématiques et de statistique ou une instance Amazon EC2

Sophie Baillargeon, Université Laval

23 novembre 2017 (mise à jour mineure le 18 novembre 2019)

Table des matières

1 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 (Baillargeon, 2017b). 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. Si ce programme est la définition d'une fonction destinée à être partagée, alors reprogrammer en C ou C++ les bouts les plus lents du corps de la fonction est une bonne solution. Par contre, 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 (Matloff, 2011 ; McCallum et Weston, 2011).

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 :

1. briser un calcul informatique en blocs de calcul indépendants,
2. exécuter simultanément, soit en parallèle, les blocs de calcul sur plusieurs unités de calcul,
3. 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 (*Central Processit Unit*) et/ou sur GPU (*Graphical Processing Unit*), aussi nommé accélérateur. 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. En fait, 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. Seul l'utilisation de CPU sera illustrée ici. Ainsi, dans le reste de ce document, le terme calcul en parallèle réfère toujours au calcul en parallèle sur CPU.

Ce document est un tutoriel qui explique, à l'aide d'exemples, comment effectuer du calcul en parallèle (sur CPU) avec R. Premièrement, de la terminologie et quelques concepts de base sont décrits à la section 2. La section 3 présente les problèmes utilisés dans les exemples. La programmation parallèle en R est ensuite introduite à la section 4, dans le contexte le plus simple de calculs en parallèle, soit celui de l'exploitation de plusieurs unités de calcul sur un seul ordinateur local. Les sections suivantes montrent comment procéder pour exploiter des ressources plus complexes qu'une seule machine, soit la grappe de serveurs du Département de mathématiques et de statistique (section 5), puis la plate-forme publique de *cloud computing* Amazon EC2 (section 6).

2 Terminologie et notions de base

La documentation sur le calcul en parallèle fait intervenir certains termes techniques qu'il est bon de d'abord s'assurer de bien comprendre. Voici un survol de la terminologie et de concepts de base concernant le calcul en parallèle.

2.1 Positionnement du calcul en parallèle comme domaine scientifique

Le calcul en parallèle est souvent vu comme un sous-domaine du **Calcul Informatique de Pointe** (CIP), en anglais *Advanced Research Computing* (ARC), qui pourrait être décomposé ainsi (St-Onge, 2017) :

- **Calcul de haute performance**, en anglais *High Performance Computing* (HPC) :
  - Profilage de code,
  - **Calcul en parallèle sur CPU**,
  - Calcul en parallèle sur GPU (sur accélérateurs) ;
- Gestion de données et partage de données, potentiellement massives ;
- Soumission de tâches de calcul ;
- etc.

Le calcul informatique de pointe est un domaine scientifique en lui même. Il apporte des outils pour résoudre des problèmes dans de divers autres domaines.

2.2 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.

L'annexe A présente une vue d'ensemble des composantes matérielles usuelles d'un ordinateur personnel. 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 que le disque, mais plus rapide d'accès que ce dernier,
- mémoire cache (intégrée au processeur) : stockage temporaire, petite capacité, très rapide d'accès.

Lorsqu'un programme est exécuté par un processeur, ce dernier a besoin de stocker temporairement des données en mémoire. Il utilise d'abord sa cache, puis la RAM si la cache est insuffisante. Cependant, si le programme à exécuter requiert plus d'espace de stockage que la capacité de la RAM, voici ce qui peut arriver :

- l'exécution du programme est arrêtée et une erreur est retournée ;
- le programme utilise de l'espace de stockage sur le disque, ce qui ralentit l'exécution du programme ;
- le programme « plante » (ne répond plus).

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.

Le calcul en parallèle sur une grappe de serveurs permet souvent d'avoir accès à plus de RAM que sur un ordinateur personnel.

Processeur ou CPU

Le rôle du processeur ou CPU (pour *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.

La terminologie est un peu embêtante ici, car certains utilisent le terme processeur pour parler d'une unité de calcul (un coeur), alors que d'autres utilisent le terme processeur pour désigner la puce informatique qui comporte la ou les unités de calcul et tout ce qui l'accompagne. Ici, 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*. Il faut savoir cependant qu'un coeur physique séparé en disons 2 coeurs logiques n'est pas aussi rapide que 2 coeurs physiques. Ce détail technique expliquera plus loin certaines différences de performance observées entre des ordinateurs personnels.

Notons qu'un ordinateur peut comporter plus d'un processeur. Un ordinateur personnel possédant plus d'un processeur est rare, mais il est courant pour un noeud dans une grappe de serveurs de calcul de posséder plus d'un processeur.

Processeur graphique ou GPU

Un processeur graphique ou GPU (pour *Graphical Processing Unit*) est un « processeur massivement parallèle, disposant de sa propre mémoire assurant les fonctions de calcul de l'affichage » (Gouarin et al. 2012). 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 de calcul. Source : http://calcul.math.cnrs.fr/Documents/Ecoles/LEM2I/Mod3/paral.pdf

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). Pour un utilisateur de R n'ayant pas de formation en informatique, apprendre une telle technologie peut représenter un défi. Heureusement, certains packages R ont été développés pour éviter d'avoir à utiliser directement la technologie

en question (Eddelbuettel, 2017). Ils proposent des fonctions R usuelles faisant office d'interface entre R et l'outil pour communiquer avec le GPU (Matloff, 2015).

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. Il est fort probable qu'ils deviennent plus populaires et plus nombreux dans un avenir rapproché.

2.3 Grappe de serveurs

Une grappe de serveurs (en anglais *computer cluster*) est un ensemble d'ordinateurs interconnectés. Ces ordinateurs constituent les noeuds (en anglais *nodes*) de la grappe. Les mots « serveurs », « noeuds » ou même « machines » désignent tous, dans ce contexte, un ordinateur. Le calcul en parallèle exploitant plusieurs noeuds d'une grappe de serveurs est parfois nommé « calcul distribué ». Aussi, une grappe de serveurs est souvent appelée « grappe de calcul ». L'expression « grappe de serveurs » est préférée ici, car elle met davantage en évidence le fait que cette grappe est physique et elle comporte plusieurs ordinateurs.

Plus tard dans ce tutoriel, nous créerons des « grappes de processus R », qui sont aussi parfois appelées « grappes de calcul ». Cependant, contrairement aux grappes de serveurs, les grappes de processus R sont virtuelles plutôt que physiques, et peuvent résider sur un seul ordinateur, comme sur plusieurs. Les noeuds d'une grappe de processus R sont des processus R et non des ordinateurs. Nous y reviendrons. Pour l'instant, nous parlons de grappes de serveurs physiques.

Comparaison d'un ordinateur dans une grappe de serveurs à un ordinateur personnel

Voici quelques caractéristiques d'un ordinateur dans une grappe de serveurs, comparativement à un ordinateur personnel (PC) :

- possède souvent plus d'un processeur, qui sont typiquement chacun divisés en plus d'un coeur ;
- a souvent une RAM de plus grande capacité qu'un PC ;
- ne possède pas toujours de stockage permanent (disque dur ou SSD) ;
- les seuls périphériques vraiment nécessaires à un serveur dans une grappe sont des ports réseau (pas besoin de connexion à un clavier, une souris, un écran, etc.).

Exemple d'architectures de grappes de serveurs

Il y a une multitude d'architectures possibles pour une grappe de serveurs. La figure 2 en présente un exemple.

Cette architecture a les caractéristiques suivantes :

- RAM locale à chaque noeud,
- système de fichier partagé (pas de disque pour stockage permanent dans les noeuds de calcul, disque à part mais tous les noeuds y sont connectés).

Les communications dans une grappe de serveurs sont typiquement effectuées comme suit :

- à l'intérieur d'un serveur : par un bus local,
- entre les serveurs : par un réseau (ex. Ethernet ou InfiniBand).

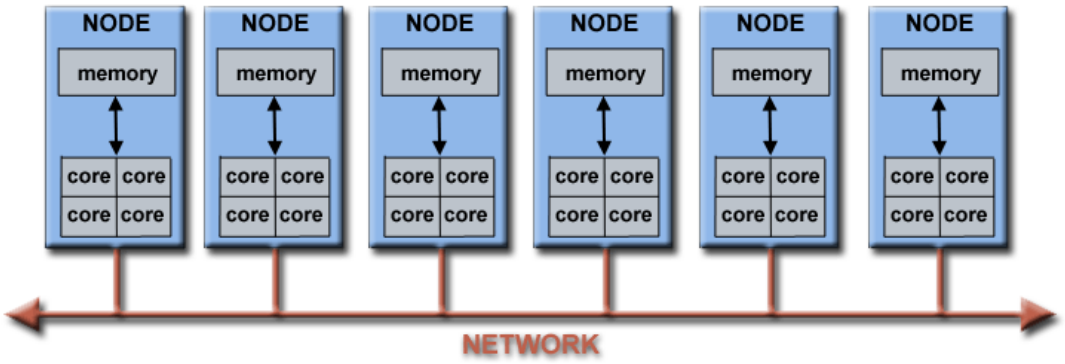

FIGURE 2 – Exemple d'architecture d'une grappe de serveurs. Source : https://computing.llnl.gov/tutorials/parallel_comp

2.4 Facteurs influençant la vitesse d'un calcul en parallèle

Les facteurs suivants ont un impact sur la vitesse d'un calcul en parallèle :

- la nature et la complexité du calcul demandé,
- la puissance des coeurs de calcul utilisés,
- le nombre de coeurs exploités,
- la vitesse des communications.

On appelle temps inactif (en anglais *overhead*) le temps consacré aux communications qui ne contribuent pas directement à la progression des tâches, mais qui a un impact sur le temps total de calcul.

En principe, plus on utilise de coeurs dans un calcul en parallèle, plus le temps total de calcul devrait être petit. Il arrive cependant que l'utilisation de plus de coeurs cause un ralentissement du calcul plutôt qu'une accélération. Cela est typiquement dû à plus de temps consacré aux communications. Un calcul distribué sur une grappe de serveurs risque plus de souffrir de temps inactif qu'un calcul en parallèle sur une seule machine, car les communications inter serveurs via un réseau sont typiquement plus lentes que les communications locales sur une machine.

2.5 Indépendance entre les tâches effectuées en parallèle

Ce document illustre comment mettre en oeuvre des calculs en parallèle uniquement dans le cas de tâches à effectuer simultanément qui sont complètement indépendantes. Voici quelques exemples de problématiques de nature statistique pouvant être résolues avec les outils et procédures décrits ici :

- Ajustement de différents modèles sur le même jeu de données, en changeant soit le type de modèle, soit la valeur des paramètres de réglage du modèle, soit les variables incluses dans le modèles. Le premier problème traité dans les exemples de ce tutoriel est de ce type.
- Prédiction de valeurs en différents points, mais à partir du même modèle et des mêmes données, comme en régression ou classification par la méthode des k plus proches voisins, ou en interpolation. Le deuxième problème traité dans les exemples de ce tutoriel est de ce type.
- Simulation Monte Carlo. Il s'agit d'effectuer à plusieurs reprise une simulation d'observations, toujours suivant la même distribution, qui elle dépend d'un modèle et d'un jeu de données d'origine. Les simulations sont suivies d'un calcul sur les observations simulées. Pour finir, les résultats sont agrégés, souvent simplement en calculant une moyenne.
- Bootstrap ou validation croisée. Ces deux techniques requièrent des calculs répétés sur différents échantillons tirés avec ou sans remise à partir d'une même base de données.

Certains problèmes de parallélisme implique la division d'un ensemble de données en sous-ensembles disjoints. Ces sous-ensembles sont répartis entre des coeurs de calcul et le même calcul est réalisé sur tous les sous-ensembles simultanément. Ce type de parallélisme est très utile lorsque les données à traiter sont tellement

volumineuses que les observations doivent être distribuées sur plusieurs noeuds dans une grappe de serveurs. Par exemple, supposons que nous souhaitions calculer la moyenne des valeurs observées d'une variable à partir de données distribuées. Nous pourrions d'abord calculer une somme pour chacun des sous-ensembles disjoints d'observations stockés sur différents noeuds. Ces sommes étant indépendantes, elles pourraient être calculées en parallèle. Ensuite, il ne resterait plus qu'à additionner toutes ces sommes et diviser le tout par le nombre total d'observations afin de trouver la moyenne globale.

Cette solution pourrait être implémentée à l'aide du modèle de programmation *MapReduce*. Ce modèle est représenté dans la figure 3.

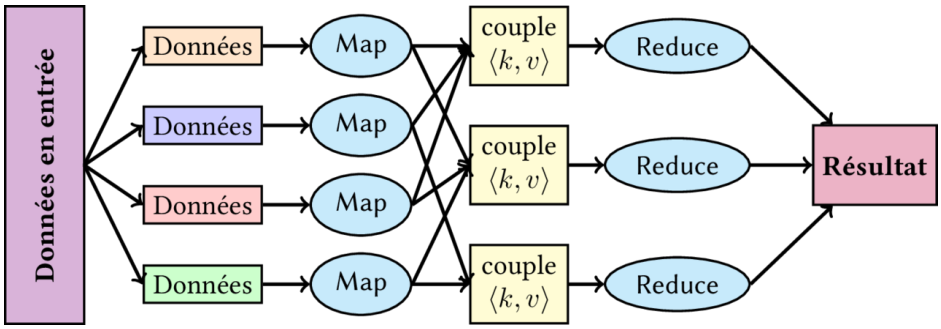

FIGURE 3 – Schéma de fonctionnement du MapReduce. Source : https://fr.wikipedia.org/wiki/MapReduce

Les deux composantes principales de ce modèle, soit les étapes *map* et *reduce*, peuvent être définies ainsi :

- *map* : appliquer un même traitement sur différents sous-ensembles de données,
- *reduce* : mettre en commun, c'est-à-dire agréger, les résultats obtenus sur les sous-ensembles.

La plateforme logicielle Hadoop de Apache est une des implémentations les plus connue du modèle MapReduce.

Depuis sa création, le modèle MapReduce a été revu et amélioré. De nos jours, un des outils les plus populaires pour résoudre un problème avec répartition de données entre plusieurs coeurs de calcul est Spark, aussi de Apache. Les ensembles de données distribués résilients (RDD pour *resilient distributed datasets*) de Spark permettent d'atteindre de meilleures performance que le modèle MapReduce.

Les problèmes de parallélisme nécessitant la répartition de données entre des coeurs de calcul ne sont pas approfondis dans ce document. Des problèmes de parallélisme nécessitant des communications entre les coeurs en cours de calcul, ne sont pas abordés non plus.

3 Présentation des problèmes utilisés dans les exemples

Dans ce tutoriel, deux problèmes seront traités dans les exemples. Le premier servira principalement à illustrer comment réaliser du calcul R en parallèle. Le second sera utilisé pour comparer des temps d'exécution de calcul en parallèle en utilisant différentes ressources de calcul.

Les deux problèmes utilisés ici peuvent être qualifiés de *embarrassingly parallel*. Cela signifie qu'ils se divisent facilement en tâches indépendantes pouvant être exécutées simultanément. Aucune tâche n'a besoin du résultat d'une autre tâche pour être exécutée.

3.1 Sélection de variables par recherche exhaustive

Supposons que nous cherchons à sélectionner la meilleure combinaison de variables à inclure dans un modèle linéaire (potentiellement généralisé). Il s'agit d'un problème commun en statistique. Pour le résoudre, nous souhaitons réaliser une recherche exhaustive de la meilleure solution parmi toutes les solutions possibles. Il s'agit donc d'un problème discret d'optimisation, que nous allons solutionner par « force brute ».

Ici, nous n'allons pas permettre l'intégration d'interactions entre variables dans le modèle. Nous nous limitons aux effets simples des variables. Le nombre de solutions possibles à notre problème, donc le nombre de modèles à ajuster au cours de la recherche exhaustive, est donc $2^p - 1$ où p est le nombre de variables explicatives potentielles. Le -1 est nécessaire, car nous voulons que le modèle comporte au moins une variable explicative. Avec l'accroissement de p, le nombre de modèles à ajuster devient rapidement grand. L'exécution de cette recherche exhaustive sur un seul coeur de calcul peut donc être, dans certains cas, très longue.

Cet exemple a été choisi, car il devrait être simple à comprendre pour un statisticien. Aussi il ne devrait pas être trop compliqué d'adapter le code fourni ici à tout autre problématique similaire d'optimisation par recherche exhaustive en statistique.

Il est facile d'identifier les tâches indépendantes pouvant être lancées en parallèle. Il s'agit de l'ajustement des modèles utilisant différentes combinaisons de variables explicatives, suivi de l'évaluation du critère à optimiser. La seule chose qui change, d'une tâche à l'autre, est la combinaison de variables explicatives incluses dans le modèle.

Il existe déjà des outils en R pour réaliser une optimisation dans le but de sélectionner des variables à inclure dans un modèle. La page web suivante fait un bon tour d'horizon de ce qui existe : All subset regression with leaps, bestglm, glmulti, and meifly. Nous allons utiliser le même exemple que celui utilisé sur cette page web, qui provient en fait de Hosmer, Lemeshow et Sturdivant (2013).

Les données traitées sont tirées d'une étude pour identifier des facteurs de risque de donner naissance à un bébé de faible poids. Ces données sont incluses dans le package R `aplore3`.

```
# À faire initialement : installation du package
# install.packages("aplore3")
```

```
library(aplore3)
```

```
## Warning: package 'aplore3' was built under R version 3.6.1
```

La variable réponse, `low`, est binaire. Ses niveaux sont `">= 2500 g"` et `"< 2500 g"` (bébé de faible poids). Nous cherchons la meilleur combinaison de variables à inclure dans un modèle de régression logistique qui prédit cette variable réponse. Il y a 8 variables explicatives potentielles : `age`, `lwt`, `race`, `smoke`, `ptl`, `ht`, `ui` et `ftv`. Elles sont décrites dans la fiche d'aide du jeu de données.

```
help(lowbwt)
```

Le modèle complet, avec toutes les variables explicatives, est le suivant.

```r
rl <- glm(low ~ age + lwt + race + smoke + ptl + ht + ui + ftv,
          data = lowbwt, family = binomial)
summary(rl)
```

```
##
## Call:
## glm(formula = low ~ age + lwt + race + smoke + ptl + ht + ui +
##     ftv, family = binomial, data = lowbwt)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.7722  -0.8040  -0.5045   0.8746   2.2231
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   1.03635    1.26647   0.818  0.41319
## age          -0.04079    0.03922  -1.040  0.29832
## lwt          -0.01636    0.00721  -2.270  0.02323 *
## raceBlack     1.12251    0.54311   2.067  0.03875 *
## raceOther     0.69388    0.46950   1.478  0.13943
## smokeYes      0.75024    0.43167   1.738  0.08221 .
## ptlOne        1.71542    0.54301   3.159  0.00158 **
## ptlTwo, etc. -0.02002    0.96939  -0.021  0.98352
## htYes         1.90929    0.72963   2.617  0.00888 **
## uiYes         0.75203    0.47273   1.591  0.11165
## ftvOne       -0.48603    0.48814  -0.996  0.31941
## ftvTwo, etc.  0.11418    0.46233   0.247  0.80494
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 234.67  on 188  degrees of freedom
## Residual deviance: 192.54  on 177  degrees of freedom
## AIC: 216.54
##
## Number of Fisher Scoring iterations: 4
```

Voyons maintenant comment nous pourrions effectuer la recherche exhaustive du meilleur modèle sans exploiter le calcul en parallèle. Premièrement, nous devons identifier tous les modèles possibles. Les instructions suivantes créent une liste contenant $2^8 - 1 = 255$ formules R, soit une par modèle possible.

```r
# Matrice d'indicatrices des variables à inclure par modèle (une ligne = un modèle)
matIndic <- do.call(expand.grid, args = rep(list(c(FALSE, TRUE)), times = 8))
matIndic <- as.matrix(matIndic[-1, ])
colnames(matIndic) <- c("age", "lwt", "race", "smoke", "ptl", "ht", "ui", "ftv")
# Création de la liste
formules <- vector(mode = "list", length = nrow(matIndic))
for (i in 1:nrow(matIndic)){
  vars <- colnames(matIndic)[matIndic[i,]]
  formules[[i]] <- as.formula(paste("low ~", paste(vars, collapse = " + ")))
}
```

Voici de quoi ont l'air les 4 derniers éléments de cette liste.

```
tail(formules, n = 4)
```

```
## [[1]]
## low ~ race + smoke + ptl + ht + ui + ftv
##
## [[2]]
## low ~ age + race + smoke + ptl + ht + ui + ftv
##
## [[3]]
## low ~ lwt + race + smoke + ptl + ht + ui + ftv
##
## [[4]]
## low ~ age + lwt + race + smoke + ptl + ht + ui + ftv
```

Ensuite, nous allons exécuter une boucle qui ajuste tous les modèles et conserve la valeur du critère à optimiser pour chacun d'eux. Utilisons ici le critère du AIC retourné par la fonction `AIC` du package `stats` de R (dans l'installation de base).

```
# Initialisation d'un vecteur pour contenir les valeurs de AIC pour chaque modèle
AICs <- rep(NA, times = length(formules))
# Boucle sur toutes les formules, donc tous les modèles
for (i in 1:length(formules)){
  # Ajustement du modèle
  rli <- glm(formules[[i]], data = lowbwt, family = binomial)
  # Calcul du critère AIC pour ce modèle
  AICs[i] <- AIC(rli)
}
```

Le nombre de variables explicatives potentielles est ici plutôt petit. Cette boucle n'est donc pas longue à rouler. Nous allons tout de même voir plus loin comment exploiter plus d'un coeur de calcul pour exécuter simultanément différentes itérations de cette boucle. Il s'agit d'un exemple jouet pour comprendre comment réaliser du calcul en parallèle en R.

Pour clore la présentation de l'exemple, voyons quel est le résultat de l'optimisation effectuée. Identifions le meilleur modèle, soit celui optimisant le AIC. La définition du AIC utilisée par la fonction `AIC` est telle que plus le AIC est petit, meilleur est le modèle. Nous cherchons donc un minimum.

```
formules[which(AICs == min(AICs))]
```

```
## [[1]]
## low ~ lwt + race + smoke + ptl + ht + ui
```

Le modèle avec le plus petit AIC est donc celui contenant les variables `lwt`, `race`, `smoke`, `ptl`, `ht` et `ui`.

3.2 Estimation de densité par noyau

Nous allons également utiliser un autre exemple pour comparer des performances de calcul en parallèle sur différentes machines et différentes grappes de calcul. Cet exemple est celui utilisé dans Baillargeon (2017b).

Il s'agit d'un exemple, tiré de Peng et de Leeuw (2002), d'écriture d'une fonction ayant pour but d'estimer la densité de probabilité d'une variable aléatoire par la méthode du noyau (en anglais *kernel density estimation*) à partir d'observations de la variable aléatoire.

Il existe en fait déjà une fonction dans le package `stats` pour faire de l'estimation de densité par noyau. Il s'agit de la fonction `density`. Nous allons écrire une version moins puissante de la fonction `density`. L'estimation de densité par noyau se fait par la formule suivante :

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

où

- x est le point en lequel nous souhaitons obtenir une estimation,
- x_i pour $i = 1, 2, \ldots, n$ sont les observations,
- K est une fonction noyau (en anglais *kernel*) à définir et
- h est un paramètre de lissage (parfois appelée fenêtre) : plus la valeur de h est grande, plus la courbe obtenue est lisse.

La fonction `density` permet l'utilisation de plusieurs fonctions noyau via l'argument `kernel`. Nous allons plutôt nous restreindre au noyau gaussien, qui est en fait la fonction de densité d'une distribution normale standard. Nous allons donc utiliser la fonction `dnorm` pour évaluer la fonction K dans la formule ci-dessus.

Voici la première fonction proposée.

```
#' Estimation de densité par noyau gaussien
#'
#' Version 1 : utilisation de 2 boucles imbriquées
#'
#' @param x vecteur numérique contenant les observations
#' @param xpts vecteur numérique contenant les points en lesquels on désire
#'             effectuer l'estimation de la densité
#' @param h nombre réel > 0 : la valeur du paramètre de lissage
#'
#' @return vecteur numérique contenant la densité estimée en tous les points de xpts
#'
ksmooth <- function(x, xpts, h)
{
  dens <- double(length(xpts))
  n <- length(x)
  for(i in 1:length(xpts)) {
    ksum <- 0
    for(j in 1:length(x)) {
      d <- xpts[i] - x[j]
      ksum <- ksum + dnorm(d / h)
    }
    dens[i] <- ksum / (n * h)
  }
  dens
}
```

Dans la fonction `ksmooth`, le premier argument, nommé `x`, n'est pas équivalent au x de la formule. Le x de la formule représente un point en lequel nous voulons faire l'estimation. Son équivalent dans la fonction `ksmooth`

est donc un élément du vecteur `xpts`. La fonction `ksmooth` travaille de façon vectorielle. Elle peut réaliser l'estimation en plusieurs points en un seul appel de fonction. Ce sont les x_i de la formule que l'on retrouve dans le vecteur `x`. Dans la boucle, ce vecteur `x` est parcouru en utilisant l'indice `j`. Alors, en fait, `x[j]` dans le corps de la fonction représente un x_i dans la formule. Le code aurait pu coller davantage à la notation dans la formule pour être encore plus clair, mais il a été choisi de le conserver tel qu'il a été proposé dans Peng et de Leeuw (2002).

Nous allons éventuellement écrire une version de cette fonction qui utilise du calcul en parallèle.

Voici un exemple d'utilisation de cette fonction.

```
# Simulation d'observations normales standard
x <- rnorm(10000)

# Points en lesquels nous souhaitons estimer la densité
xpts <- seq(from = -4, to = 4, by = 0.25)

# Estimation de densité avec density
dens <- density(x, from = -4, to = 4, n = 161, kernel = "gaussian", bw = 1)

# Estimation de densité avec ksmooth
densk <- ksmooth(x = x, xpts = xpts, h = 1)

# Représentation graphique des résultats
par(mar = c(4, 4, 1, 1) + 0.1)
hist(x, freq = FALSE, ylab = "densité", cex.main = 0.9, main = "")
lines(dens, col = "red", lwd = 2)
lines(x = xpts, y = densk)
legend(x = "topright", bty = "n", lty = 1, lwd = c(2,1),
       col = c("red", "black"), legend = c("density", "ksmooth"))
par(mar = c(5, 4, 4, 2) + 0.1)
```

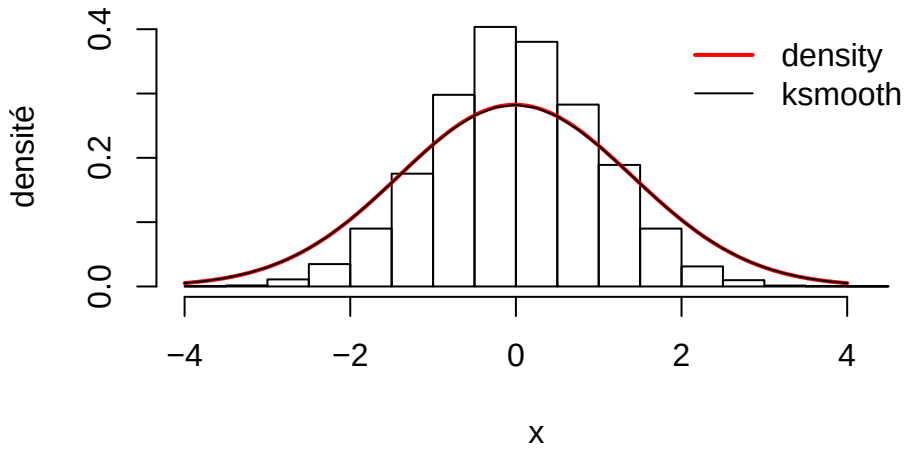

FIGURE 4 – Densité empirique d'observations simulées selon une loi normale standard

Un histogramme est aussi une méthode d'estimation de densité. Dans la figure 4, deux courbes de densité estimée par la méthode du noyau ont été superposées à un histogramme. La première courbe à été obtenue avec la fonction `density`, la deuxième avec notre fonction `ksmooth`. Les deux courbes se chevauchent parfaitement. Notre fonction effectue donc le même calcul que la fonction `density` avec fonction de noyau gaussien.

4 Programmation parallèle en R sur un ordinateur multi-coeurs

Les outils pour effectuer des calculs en parallèle sur CPU en R ont beaucoup évolué au cours des dernières années. Plusieurs packages ont été développés (Eddelbuettel, 2017 ; Mahdi, 2014). Depuis la version 2.14.0 de R, un de ces packages a été inclus dans la distribution de base de R. Il est donc déjà installé sur tout ordinateur ayant une installation de base de R (mais il n'est pas chargé par défaut lors du démarrage d'une nouvelle session R). Il est maintenu par le R core team, ce qui est habituellement un gage de qualité. Ce package se nomme `parallel`. Il incorpore des versions légèrement modifiées des packages `multicore` (maintenant archivé) et `snow` (encore disponible sur le CRAN), qui font partie des premiers packages de calcul parallèle en R à avoir été développés. Nous allons voir comment utiliser le package `parallel`.

Le lancement de tâches en parallèle avec le package `parallel` se base sur l'utilisation de fonctions de la famille des `apply`. Une alternative à ce type de fonction sont les outils d'écriture de boucles proposés dans le package `foreach`. Le package `doParallel` permet le lancement de calculs en parallèle avec `foreach`. Il sera rapidement présenté.

Pendant longtemps, le package le plus utilisé pour le calcul en parallèle dans la communauté R était `snowfall`, soit une version améliorée du package `snow`. En 2012, Martin Leclerc, alors étudiant à l'Université Laval au doctorat en mathématiques concentration statistique, avait donné un séminaire dans lequel il traitait de l'utilisation de ce package pour lancer des calculs en parallèle sur la grappe de calcul du Département de mathématiques et de statistique. Avec l'arrivée du package `parallel`, le package `snowfall` semble perdre en popularité. Nous n'allons pas voir ici comment l'utiliser.

Notons que certains packages permettent l'envoi de tâches en parallèle simplement par l'intermédiaire d'un argument de certaines de leurs fonctions. C'est le cas par exemple des fonctions de la famille des `ply` (alternative aux fonctions de la famille des `apply`) offertes dans le package `plyr`. Ces fonctions requièrent en fait l'initialisation préalable d'une grappe de processus R à l'aide de fonctions d'un package pour le calcul en parallèle en R tel que `parallel`.

Étapes de travail

Le lancement de calculs en parallèle en R comporte typiquement trois grandes étapes :

1) Initialiser une grappe de processus R.
2) Lancer le calcul sur la grappe en utilisant des fonctions spécifiques au lancement de calculs en parallèle. Ces fonctions s'occupent de diviser et soumettre les tâches simultanément sur les différents processus de la grappe.
3) Fermer la grappe.

Dans les sous-sections suivantes, nous allons voir des exemples de réalisation de ces étapes pour résoudre le problème de sélection de variables par recherche exhaustive avec le package `parallel`, puis avec le package `doParallel`.

À cette étape de l'apprentissage du fonctionnement du calcul en parallèle en R, nous allons exploiter les coeurs de calcul d'un seul ordinateur personnel. Plus loin, nous verrons comment exploiter une grappe de serveurs ou des ressources infonuagiques (en anglais *cloud computing*).

4.1 Utilisation du package `parallel`

Nous allons d'abord voir comment lancer des ajustements de modèles en parallèle pour réaliser la sélection de variable par recherche exhaustive décrite à la section 3.1. Ce que le package `parallel` (R Core Team, 2017) offre pour lancer des tâches non identiques en parallèle sont des versions adaptées des fonctions `apply`, `lapply` ou `sapply`. La syntaxe de la boucle `for` exécutée précédemment ne se traduit pas directement en lancement de calculs parallèles avec des fonctions du package `parallel`. Il faut d'abord transformer la boucle en appel à une fonction de la famille des `apply`. Il est utile à ce point de bien comprendre le fonctionnement des fonctions de la famille des `apply` (Baillargeon, 2017a).

Tout d'abord, voici une copie de la boucle qui nous intéresse.

```
AICs <- rep(NA, times = length(formules))
for (i in 1:length(formules)){
  rli <- glm(formules[[i]], data = lowbwt, family = binomial)
  AICs[i] <- AIC(rli)
}
```

Cette boucle `for` itère sur `i` allant de 1 à 255, soit le nombre de modèles à évaluer. Lors d'une itération, la formule en position `i` dans la liste `formules` est donnée en entrée à la fonction `glm`. Un modèle est ainsi ajusté. Ensuite, le AIC de ce modèle est évalué et le résultat est stocké en position `i` dans le vecteur `AICs`. Voici maintenant une version `apply` de cette boucle.

```
AICs_apply <- sapply(
  X = formules,
  FUN = function(x) {
    AIC(glm(formula = x, data = lowbwt, family = binomial))
  }
)
```

Cet appel à `sapply` itère sur les éléments de l'objet fourni comme argument `X` (premier argument). Elle prend ensuite comme argument une fonction, qui détermine le calcul à faire sur chaque élément. Ici, nous avons créé une fonction anonyme qui effectue tous les calculs désirés, soit l'ajustement du modèle avec une certaine formule et l'évaluation du AIC (les appels aux fonction `glm` et `AIC` ont été combinés en une seule instruction).

Lors de l'utilisation d'une fonction de la famille des `apply`, le programmeur n'a pas besoin de se soucier du stockage du résultat comme il doit le faire lors de l'utilisation d'une boucle. La fonction de la famille des `apply` s'en occupe. Ici, le `sapply` retourne les AIC bout-à-bout dans un vecteur, tout comme le fait la boucle `for`. En bout de ligne, la boucle `for` et l'appel à `sapply` retournent exactement le même résultat.

```
all.equal(AICs_apply, AICs)
```

```
## [1] TRUE
```

Nous aurions pu utiliser la fonction `lapply` plutôt que `sapply`. Elle aussi effectue une boucle sur les éléments d'un vecteur ou d'une liste. Cependant, `lapply` retourne le résultat sous forme de liste (d'où le `l` dans son nom). Ici, le résultat est une seule valeur numérique et nous souhaitions stocker le résultat pour chaque modèle ajusté dans un vecteur. La fonction `sapply` effectue la transformation du format des résultats d'une liste vers un vecteur pour nous.

Avec un peu de pratique, il est plutôt simple de transformer une boucle `for` en un appel à la fonction `lapply` (ou `sapply`). La fonction `lapply` peut itérer sur les éléments de n'importe quel vecteur ou liste, incluant un vecteur contenant les entiers 1 à n. La fonction appliquée sur ces éléments peut quant à elle effectuer n'importe quel calcul ! Il suffit que son premier argument représente un élément de l'objet donné comme argument `X` à `lapply`.

La fonction fournie à l'argument `FUN` lors de l'appel à `lapply` peut comporter plus d'un argument. Les arguments autres que le premier peuvent être donnés en entrée dans l'appel à la fonction `lapply`, grâce à son argument spécial `...`. Dans notre exemple, nous pourrions ajouter un deuxième argument à la fonction anonyme, servant à passer en entrée le jeu de données (argument `data` ci-dessous). Dans l'appel au `sapply`, nous pourrions alors passer une valeur à cet argument comme suit.

```
AICs_apply <- sapply(
  X = formules,
  FUN = function(x, data) {
    AIC(glm(formula = x, data = data, family = binomial))
  },
  data = lowbwt
)
```

Il sera facile de transformer cet appel à la fonction `sapply` en un lancement de calculs en parallèle avec la fonction `parSapply` du package `parallel`.

Remarque : Si nous devions répéter exactement un même calcul plusieurs fois (par exemple si nous effectuions une simulation Monte Carlo), nous pourrions utiliser la fonction `clusterEvalQ` du package `parallel` pour lancer des calculs identiques simultanément dans tous les processus R parallèles de la grappe initialisée. Nous ne pouvions pas utiliser cette fonction ici, car les calculs faits en parallèle sont différents (différents modèles ajustés).

1) Initialisation de la grappe de processus R

Il faut d'abord s'assurer que le package `parallel` est chargé.

```
library(parallel)
```

Nous pouvons ensuite vérifier combien de coeurs de calcul comporte l'ordinateur sur lequel nous travaillons. Nous aurons besoin de cette information.

```
ncores <- detectCores()
ncores
```

```
## [1] 4
```

S'agit-il de coeurs logiques ou physiques ?

```
detectCores(logical = FALSE)
```

```
## [1] 2
```

Dans mon cas, l'ordinateur sur lequel je travaille comporte un total de 4 coeurs logiques répartis en 2 coeurs physiques. Mais ce qui compte pour la prochaine étape, c'est le nombre de coeurs logiques.

Nous sommes maintenant prêt à initialiser une grappe de processus R avec la fonction `makeCluster`. Cela signifie de créer un ensemble de copies de processus R avec lesquels nous communiquerons à l'aide de connecteurs (en anglais *sockets*). Ces processus seront démarrés, autant que possible, sur différents coeurs de calcul. Les fonctions du package `parallel` automatisent les communications entre notre session R principale et les processus R de la grappe.

Il faut spécifier à la fonction `makeCluster` sur quels ordinateurs les processus R doivent être démarrées et combien de processus R démarrer sur chaque ordinateur. Lorsque nous souhaitons travailler localement, en exploitant seulement les coeurs du PC que nous utilisons, il suffit de fournir à `makeCluster` le nombre de processus R à démarrer. Ce nombre peut être n'importe quel entier positif, mais si notre but est d'accélérer des temps d'exécution, il est inutile de démarrer plus de processus que le nombre de coeurs de calcul sur l'ordinateur. Il peut même être utile de laisser un coeur libre pour que les autres processus roulant sur l'ordinateur ne soient pas trop ralentis. C'est ce que nous allons faire ici.

```
grappe <- makeCluster(ncores - 1)
```

Remarque : Sous Windows, la première fois que j'ai soumis cette commande, une autorisation m'a été demandée.

2) Soumission des instructions de calcul parallèle

Nous allons maintenant transformer l'appel à la fonction `sapply` par un appel à la version parallèle de cette fonction offerte par le package `parallel`, nommée `parSapply`. Il suffit de fournir un objet de plus en entrée à la fonction, soit une valeur à l'argument `cl`. Cet objet doit avoir été créé par la fonction `makeCluster` (ou une fonction équivalente). Si nous utilisons notre première version de l'appel au `parSapply`, voici ce que nous obtenons.

```
AICs_par <- parSapply(
  cl = grappe,
  X = formules,
  FUN = function(x) {
    AIC(glm(formula = x, data = lowbwt, family = binomial))
  }
)
```

```
## Error in checkForRemoteErrors(val) :
##   3 nodes produced errors; first error: object 'lowbwt' not found
```

L'exécution de cette instruction retourne une erreur parce que l'objet R `lowbwt` (le jeu de données) n'est pas présent dans l'environnement de travail des processus R de la grappe de calcul. Il faut soit ajouter un appel à la fonction `clusterExport` pour l'exporter vers chacun des processus :

```
clusterExport(cl = grappe, varlist = "lowbwt")
AICs_par <- parSapply(
  cl = grappe,
  X = formules,
  FUN = function(x) {
    AIC(glm(formula = x, data = lowbwt, family = binomial))
  }
)
```

soit utiliser la deuxième version de l'appel à la fonction `parSapply` que nous avons écrite, c'est-à-dire la version avec une fonction anonyme prenant le jeu de donnée en argument :

```
AICs_par <- parSapply(
  cl = grappe,
  X = formules,
  FUN = function(x, data) {
    AIC(glm(formula = x, data = data, family = binomial))
  },
  data = lowbwt
)
```

Est-ce que ce que nous venons de faire a fonctionné ? Voyons d'abord si nous avons obtenu les mêmes résultats.

```
all.equal(AICs_par, AICs)
```

```
## [1] TRUE
```

Oui, nous obtenons les mêmes résultats. Comparons maintenant le temps d'exécution en lançant les calculs séquentiellement versus en les lançant en parallèle à l'aide de la fonction R `system.time` :

```
# Calcul séquentiel
system.time(
  AICs_apply <- sapply(
    X = formules,
    FUN = function(x, data) {
      AIC(glm(formula = x, data = data, family = binomial))
    },
    data = lowbwt
  )
)
```

```
##    user  system elapsed
##    0.87    0.00    0.95
```

```r
# Calcul parallèle
system.time(
  AICs_par <- parSapply(
    cl = grappe,
    X = formules,
    FUN = function(x, data) {
      AIC(glm(formula = x, data = data, family = binomial))
    },
    data = lowbwt
  )
)
```

```
##    user  system elapsed
##    0.00    0.00    0.38
```

Les calculs ont été environ 2 fois plus rapides en parallèle. Ici, nous avons exploité seulement 3 coeurs logiques de calcul, alors ce résultat est sensé. Nous allons revenir plus loin sur la comparaison de temps d'exécution pour des calculs séquentiels versus parallèles. Pour l'instant, remarquons seulement que pour le calcul séquentiel le temps `elapsed` est égal ou pratiquement égal à la somme des temps `user` et `system`, mais pas pour le calcul en parallèle. Pourquoi ? Parce qu'avec les calculs en parallèle, presque rien n'est effectué dans la session R à partir de laquelle les calculs sont lancés, à part des communications avec les processus R de la grappe initialisée avec `makeCluster`. Les calculs sont plutôt exécutés dans les processus R de la grappe.

Remarque : Si nous avions eu besoin de charger un package dans les processus R de la grappe de calcul, nous aurions dû utiliser la fonction `clusterEvalQ` comme dans cet exemple :

```r
clusterEvalQ(cl = grappe, expr = library(dplyr))
```

3) Fermeture de la grappe de processus R

Lorsque nous avons terminé nos calculs en parallèle, il est recommandé de fermer la grappe de processus R. En arrêtant ces processus parallèles, la mémoire qu'ils utilisent est libérée. La fonction `stopCluster` ferme une grappe de calcul initialisée avec `makeCluster`.

```r
stopCluster(grappe)
```

4.2 Utilisation du package `doParallel`

Il existe un certain nombre d'alternatives au package `parallel`. L'une d'entre elles est rapidement présentée ici : le package `doParallel` (Microsoft Corporation et Weston, 2017). En fait, le package `doParallel` dépend du package `parallel`, mais il propose une syntaxe alternative pour le lancement de calculs en parallèle, non basée sur l'appel de fonctions de type `apply`. Le package `doParallel` se base plutôt sur des appels à la fonction `foreach`. Sans entrer dans les détails, voici un exemple de code qui solutionne le problème de sélection de variables par recherche exhaustive décrit à la section 3.1 en utilisant `doParallel`.

```r
library(doParallel)

lowbwt <- lowbwt  # pour avoir une copie de lowbwt dans l'environnement de travail
grappe <- makeCluster(ncores - 1)
registerDoParallel(grappe)

AICs_for <- foreach(x = formules, .combine = c)  %dopar% {
  AIC(glm(formula = x, data = lowbwt, family = binomial))
}

stopCluster(grappe)
```

Mentionnons d'abord qui si nous avions utilisé `%do%` au lieu de `%dopar%` dans le code précédent, nous aurions effectué le même calcul, mais séquentiellement sur un seul coeur de calcul plutôt qu'en parallèle sur plusieurs coeurs. Dans le code précédent, la fonction `registerDoParallel` copie l'environnement de travail de notre session R courante dans les processus R parallèles ouverts par `makeCluster`. Il faut donc s'assurer que le jeu de données, c'est-à-dire l'objet `lowbwt`, soit dans notre environnement de travail avant d'appeler `registerDoParallel` (l'objet `lowbwt` d'origine se trouve plutôt dans l'environnement du package `aplore3`). Ainsi, l'appel à l'opérateur `%dopar%`, qui lance les calculs en parallèle, ne génère pas l'erreur que `lowbwt` est introuvable. L'objet se trouve bien dans l'environnement de travail des processus parallèles.

Notons que si nous modifions un objet de l'environnement de travail de notre session R courante après l'appel à la fonction `registerDoParallel`, il est seulement modifié dans notre session courante, pas dans les processus de la grappe.

Cette présentation de l'utilisation du package `doParallel` est très brève. Elle vise seulement à mettre en évidence l'existence d'alternatives ou de compléments à `parallel`. Le reste de ce document exploitera uniquement le package `parallel`.

4.3 Comparaison de temps d'exécution avec différents ordinateurs personnels

Afin de mieux comprendre les gains en temps d'exécution que peut apporter le calcul en parallèle, quelques expérimentations ont été réalisées sur quatre ordinateurs personnels différents. La problématique d'estimation de densité par noyau décrite à la section 3.2 a été utilisée pour ces expérimentations. La procédure d'expérimentation et les résultats obtenus sont présentés ici.

Caractéristiques des ordinateurs utilisés

Voici d'abord les caractéristiques des quatre ordinateurs utilisés.

Ordinateur personnel 1

- Type : portable
- Système d'exploitation : Windows 7, 64 bits
- Mémoire (RAM) : 8 GB
- Processeur : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz 2.50 GHz
  - Caractéristiques de ce processeur : 2 coeurs physiques possédant chacun 2 coeurs logiques

Ordinateur personnel 2

- Type : portable
- Système d'exploitation : Windows 10, 64 bits
- Mémoire (RAM) : 12 GB
- Processeur : Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
  - Caractéristiques de ce processeur : 2 coeurs physiques possédant chacun 2 coeurs logiques

Cet ordinateur a un processeur plus puissant que le premier (7^e génération de processeur Intel(R) Core(TM) i7 au lieu d'un processeur Intel(R) Core(TM) de génération précédente et fréquence de base de 2.70 GHz au lieu de 2.50 GHz). Il a aussi une mémoire de plus grande capacité que l'ordinateur 1. Cependant, étant donné que qu'il n'y a pas d'enjeux d'utilisation de mémoire dans la problématique utilisée pour les expérimentations, cet aspect ne devrait donc pas avoir d'impact sur les résultats.

Ordinateur personnel 3

- Type : PC de bureau
- Système d'exploitation : Windows 10, 64 bits
- Mémoire (RAM) : 12 GB
- Processeur : Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz 3.30 GHz
  - Caractéristiques de ce processeur : 4 coeurs physiques

L'ordinateur comporte 4 coeurs, tous physiques ceux-là (pas de *multithreading* dans ce processeur). Le processeur de cet ordinateur est plus puissant que celui du premier ordinateur, mais moins puissant que celui du deuxième, malgré une fréquence de base plus grande (3.30 versus 2.70), car il s'agit d'un processeur d'ancienne génération.

Ordinateur personnel 4

- Type : PC de bureau
- Système d'exploitation : Windows 10, 64 bits
- Mémoire (RAM) : 16 GB
- Processeur : Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz 4.00 GHz
  - Caractéristiques de ce processeur : 4 coeurs physiques possédant chacun 2 coeurs logiques

C'est l'ordinateur le plus puissant des trois. Son processeur a la plus haute fréquence (4.00 GHz) et il comporte plus de coeurs (8 coeurs logiques au total).

Programme R

Pour mener ces expérimentations, il a tout d'abord fallu écrire une version de la fonction `ksmooth` qui exploite le calcul vectoriel pour se débarrasser de la boucle sur les éléments de `x`, ainsi qu'un appel à la fonction `sapply` pour remplacer la boucle sur les éléments de `xpts`.

La fonction d'origine :

```
#' Estimation de densité par noyau gaussien, version 1
ksmooth <- function(x, xpts, h)
{
  dens <- double(length(xpts))
  n <- length(x)
  for(i in 1:length(xpts)) {
    ksum <- 0
    for(j in 1:length(x)) {
      d <- xpts[i] - x[j]
      ksum <- ksum + dnorm(d / h)
    }
    dens[i] <- ksum / (n * h)
  }
  dens
}
```

a été modifiée comme suit :

```
#' Estimation de densité par noyau gaussien avec calcul séquentiel
ksmooth_apply <- function(x, xpts, h)
{
  n <- length(x)
  sapply(
    X = xpts,
    FUN = function(xpts_i) {
      sum(dnorm((xpts_i - x) / h)) / (n * h)
    }
  )
}
```

Ce sont les calculs en différents points (éléments de `xpts`) qui ont été effectués en parallèle. Il a suffi de remplacer l'appel à `sapply` par un appel à `parSapply`. Pour ce faire, il a fallu ajouter un argument à la fonction `ksmooth_apply` pour fournir en entrée un objet créé par la fonction `makeCluster`.

```
#' Estimation de densité par noyau gaussien avec calcul en parallèle
ksmooth_par <- function(grappe, x, xpts, h)
{
  n <- length(x)
  parSapply(
    cl = grappe,
    X = xpts,
    FUN = function(xpts_i) {
      sum(dnorm((xpts_i - x) / h)) / (n * h)
    }
  )
}
```

Un vecteur d'observations assez grand pour qu'une évaluation en un point soit encore rapide, mais pas instantanée, a été simulé.

```
set.seed(827)
x <- rnorm(1000000)
```

Les valeurs des points en lesquels une estimation de densité par noyau a été réalisé forment une séquence de nombres entre -4 et 4 inclusivement, séparés par des bonds de valeur 0.05.

```
xpts <- seq(from = -4, to = 4, length.out = 161)
```

La longueur du vecteur `xpts` détermine le nombre d'itérations à répartir entre les différents processus R parallèles. Dans ces expérimentations, 161 itérations ont été réparties, ce qui n'est pas une grande quantité.

La grappe de processus R a été initialisée comme suit.

```
grappe <- makeCluster(ncores - 1)
```

Le but de ces expérimentations était de comparer les temps d'exécution entre le calcul séquentiel sur un seul coeur de calcul et le calcul en parallèle sur plus d'un coeur de calcul, et ce, sur des machines possédant différents types de processeur. La fonction `ksmooth_apply`, puis la fonction `ksmooth_par`, ont été appelées. Cependant, une comparaison basée sur un seul appel à chacune des fonctions aurait pu être trompeuse. Il aurait été possible que, par hasard, un autre programme roule en même temps et ralentisse des calculs. Cent appels à chacune des fonctions ont donc été exécutés, puis les temps médians d'exécution ont été comparés. Le package `microbenchmark` a été utilisé pour faciliter la mise en oeuvre de ces comparaisons. Voici comment procéder.

```r
library(microbenchmark)

microbenchmark(unit = "s",
  dens_apply = ksmooth_apply(x = x, xpts = xpts, h = 1),
    dens_par = ksmooth_par(grappe = grappe, x = x, xpts = xpts, h = 1)
)
```

```
## Unit: seconds
##        expr       min        lq      mean    median        uq       max neval
##  dens_apply 13.059511 13.110206 13.230693 13.172241 13.283097 14.890084   100
##    dens_par  5.605728  5.697177  5.788225  5.736612  5.800708  7.200513   100
```

Finalement, la grappe de processus R a été arrêtée.

```r
stopCluster(grappe)
```

Le temps médian du calcul utilisé dans ces expérimentations, lorsque réalisé séquentiellement (ligne `dens_apply` de la sortie de `microbenchmark`), est seulement de 13 secondes sur l'ordinateur personnel 2. En pratique, nous n'aurions pas intérêt à effectuer ce calcul en parallèle vu sa rapidité. Un calcul aussi rapide a été choisi pour ces expérimentations parce qu'il devait être lancé un très grand nombre de fois (100 répétitions dans plusieurs contextes différents, présentés dans la présente section, mais aussi dans d'autres sections plus loin).

Résultats

Tout d'abord, le tableau 1 résume les caractéristiques des quatre ordinateurs personnels utilisés. Ces caractéristiques permettront d'expliquer les résultats obtenus.

Table 1: Caractéristiques des quatre ordinateurs personnels utilisés

Caractéristique	PC 1	PC 2	PC 3	PC 4
Processeur (tous Intel(R) Core(TM))	i5-2520M	i7-7500U	i5-2500K	i7-4790K
Fréquence de base du processeur (GHz)	2.50	2.70	3.30	4.00
Nombre de coeurs physiques	2	2	4	4
Nombre de coeurs logiques	4	4	4	8

Différents processeurs

Le tableau 2 permet de comparer les résultats obtenus en terme de temps de calcul entre les différents ordinateurs. Les grappes de calcul parallèle employées comportaient toujours un coeur de moins que le nombre total de coeurs (potentiellement logiques) de l'ordinateur.

Table 2: Temps de calcul médians, en secondes, sur les quatre ordinateurs

	PC 1	PC 2	PC 3	PC 4
calcul séquentiel	17.062	13.172	14.171	11.546
calcul parallèle	10.588	5.737	5.111	2.251
facteur d'accélération	1.61	2.30	2.77	5.13
nombre de processus R parallèles dans la grappe	3	3	3	7

Les temps d'exécution pour le calcul séquentiel varient d'un ordinateur à l'autre. En raison de la puissance des processeurs qu'ils possèdent, le PC 1 est le plus lent et le PC 4 le plus rapide. Le PC 3 est plus lent que le PC 2 même si la fréquence de sont processeur est plus grande (3.30 versus 2.70), car il possède un processeur d'ancienne génération, donc moins puissant.

Le calcul en parallèle permet une réduction du temps d'exécution sur toutes les machines.

Le facteur d'accélération du calcul parallèle versus séquentiel est plus grand pour le PC 2 que pour le PC 1 en raison de l'amélioration de la technologie de *multithreading* entre les deux processeurs.

Cependant, les 3 processus parallèles exploités sur les PC 1 et 2 ne permettent pas d'atteindre un aussi grand facteur d'accélération que les 3 processus exploités sur le PC 3. Ce résultat est dû au fait que les PC 1 et 2 possèdent en réalité seulement 2 coeurs physiques, chacun divisés en 2 coeurs logiques, alors que les coeurs du PC 3 sont tous physiques.

Les calculs lancés sur une grappe de 3 processus parallèles sur le PC 3 permettent presque d'atteindre un facteur d'accélération de 3. On n'atteint pas tout à fait ce gain intuitivement attendu parce que le calcul en parallèle requière du temps de communication entre les noeuds de la grappe de calcul en plus du temps réellement utilisé pour effectuer les calculs.

Le plus grand facteur d'accélération est sans surprise observé sur le PC 4, qui possède le plus grand nombre de coeurs et sur lequel une grappe de 7 processus à été utilisée.

Différents nombres de processus R parallèles dans la grappe de calcul

La figure 5 montre l'effet du nombre de processus R inclus dans la grappe de calcul sur le temps de calcul en parallèle sur le PC 2, qui possède au total 4 coeurs logiques (2 coeurs physiques, chacun divisés en 2 coeurs logiques).

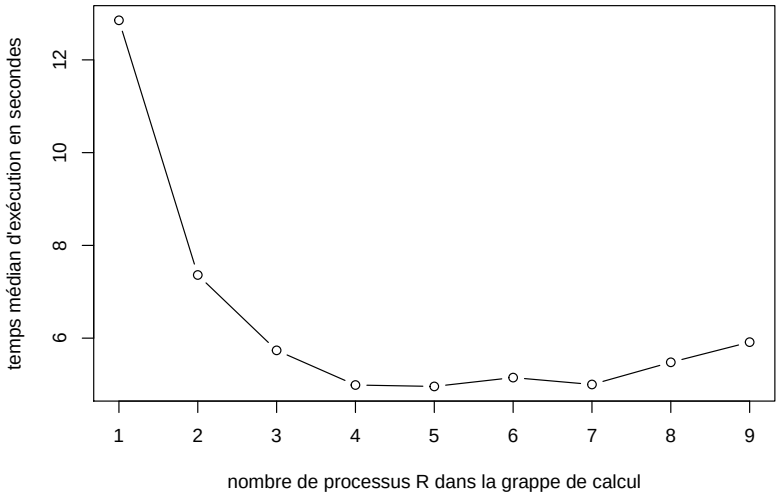

FIGURE 5 – Temps de calcul médians, en secondes, sur le PC 2, en fonction du nombre de processus R parallèles exploités

Étant donné que le PC 2 comporte deux coeurs physiques, le gain en temps en passant de l'utilisation de 1 seul processus à l'utilisation de 2 processus en parallèle est important. Le temps d'exécution est presque réduit de moitié (7.36 secondes versus 12.85 secondes). Utiliser 3 ou 4 processus R parallèles continue d'apporter des gains en temps. Cependant, utiliser plus de 4 processus R parallèles sur cette machine à 4 coeurs logiques ne permet pas d'accélérer davantage les calculs comparativement à l'utilisation de 4 processus parallèles. Même qu'à partir de 8 processus R en parallèle, les calculs sont ralentis par rapport à ceux effectués avec la grappe optimale pour ce PC à 4 processus parallèles.

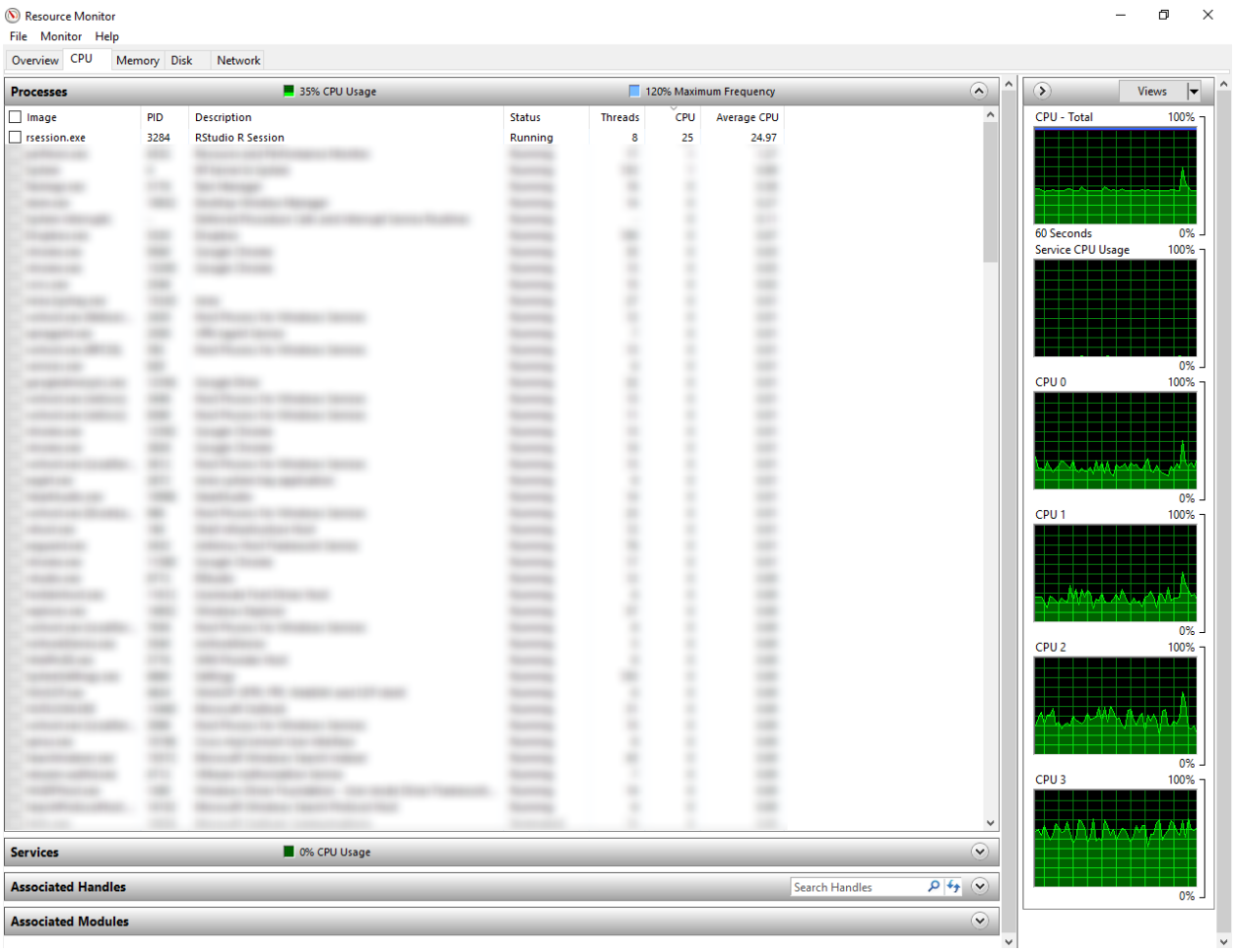

FIGURE 6 – Utilisation du processeur pendant des calculs séquentiels

Il semble donc que l'exploitation de grappes de calcul contenant plus de processus parallèles que le nombre de coeurs logiques de l'ordinateur employé n'est pas avantageuse.

4.4 Monitorage de l'utilisation du processeur pendant des calculs R séquentiels et parallèles

Pour comprendre encore mieux comment le processeur d'un ordinateur est exploité pendant des calculs R, cette section présente des copies d'écran du monitorage des ressources pendant qu'étaient effectués des calculs séquentiels et parallèles sur l'ordinateur personnel 2.

Sous Windows, le moniteur de ressources peut s'ouvrir ainsi :

- appuyer sur la touche de logo Windows + la touche R (raccourci clavier Windows + R) > ouvre la boîte de dialogue « Exécuter » ;
- taper `Resmon.exe` dans la boîte de dialogue « Exécuter » et appuyer sur la touche Enter.

La figure 6 présente l'utilisation des 4 coeurs (moniteurs nommés CPU0 à CPU3) du PC pendant des calculs séquentiels. Tous ces calculs sont effectués à partir d'une seule session R. Il s'agit du processus nommé `rsesssion.exe`, qui est associé à une utilisation moyenne de 24.97% de la capacité de calcul totale du PC (moniteur nommé CPU - Total). Cela revient donc à l'utilisation d'un seul coeur. En réalité, les calculs ont été répartis entre les 4 coeurs de calcul du PC, mais sans dépasser la capacité de calcul d'un seul coeur.

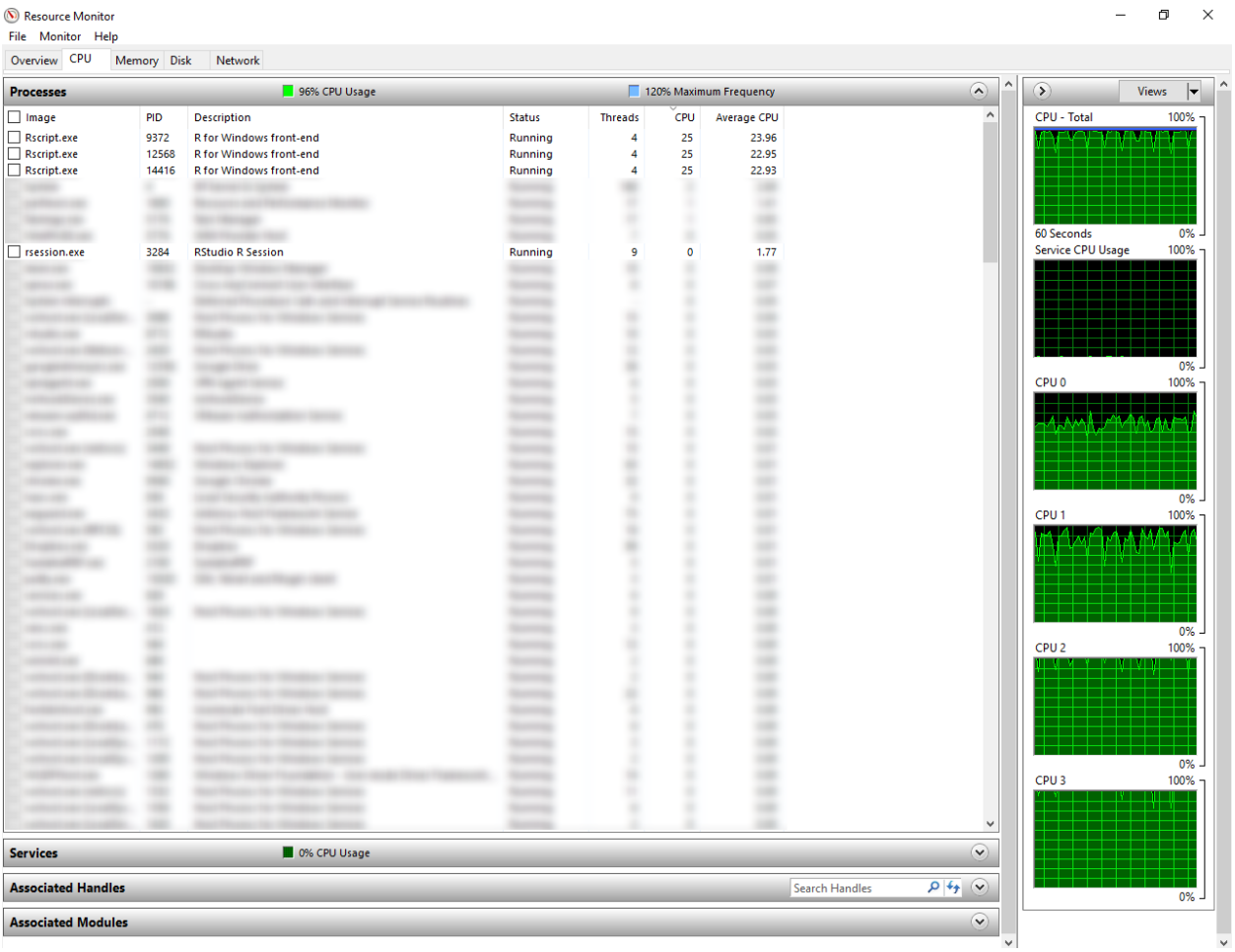

FIGURE 7 – Utilisation du processeur pendant des calculs sur une grappe de 3 processus R parallèles

Notons que, globalement, 35% du CPU était utilisé dans la figure 6. C'est donc dire qu'environ 10% (35% - 25%) du CPU était utilisé par les autres processus actifs sur le PC pendant les calculs.

Pour les calculs parallèles, j'ai demandé à la fonction `makeCluster` de créer une grappe de processus R. Le moniteur de ressources a été ouvert pendant un calcul sur une grappe de 3 processus R. La figure 7 présente donc, en plus du processus pour la session R à partir de laquelle les calculs ont été lancés (`rsesssion.exe`), trois autres processus, nommés `Rscript.exe`. Ceux-ci ont utilisé en moyenne (*Average CPU* dans la figure) 23.96%, 22.95% et 22.93% de la capacité de calcul totale du PC, pour un total d'environ 70%. S'ajoute à cette quantité environ 2% du CPU utilisé par `rsesssion.exe`. Ainsi, ces calculs ont presque utilisé au total les trois quarts de la capacité totale de calcul du PC, ce qui se rapproche de l'exploitation de 3 coeurs sur 4.

5 Calcul R en parallèle sur la grappe de serveurs de calcul du DMS

Maintenant que nous savons comment lancer des calculs en parallèle localement, soit sur la machine à partir de laquelle nous travaillons, la prochaine étape est d'apprendre à lancer des calculs en parallèle sur d'autres machines, par connexion à distance. Nous verrons comment faire ça dans cette section, en utilisant la grappe de serveurs de calcul du DMS (Département de mathématiques et de statistique de l'Université Laval). Il faut notamment apprendre à :

- adapter les instructions pour l'initialisation de la grappe de processus R parallèles ;
- se connecter à distance aux serveurs de la grappe de calcul du DMS ;
- transférer des fichiers entre notre ordinateur et la grappe de calcul du DMS ;
- soumettre du code R en ligne de commande (pas obligatoire, mais plus pratique).

Les ordinateurs du réseau informatique du DMS sont sous un système d'exploitation Linux. Même avec peu d'expérience sous Linux, il n'est pas si compliqué de suivre les instructions décrites ici. Il suffit d'être minimalement à l'aise avec le terminal et de connaître quelques commandes Linux de base. Le web regorge de bons tutoriels Linux pour ceux qui désirent se familiariser davantage avec ce système d'exploitation.

Notons que nous aurions aussi pu parler ici de l'utilisation des ressources offertes par Calcul Québec. Il s'agit de supercalculateurs pour le calcul en parallèle utilisables par tout chercheur (et ses étudiants) admissible aux subventions provenant des conseils de recherche canadiens, à la condition d'avoir obtenu des accès aux ressources. Ce sujet ne sera pas abordé dans ce document, mais Calcul Québec offre beaucoup d'information à ce sujet, notamment par son wiki et par le biais de formations.

5.1 Caractéristiques des serveurs de la grappe de calcul du DMS

En 2008, le Département de mathématiques et de statistique s'est doté d'une grappe de serveurs de calcul. Les noeuds de cette grappe sont nommés « `dms1` » à « `dms12` ». Les machines `dms11` et `dms12` sont un peu plus récentes. L'accès à ces serveurs peut uniquement se faire à partir du l'ordinateur nommé « `maitre` », qui est lui accessible à distance.

TABLE 3: Caractéristiques des noeuds de la grappe de calcul du DMS et de l'ordinateur `maitre`. Source : http://archimede.mat.ulaval.ca/dokuwiki/doku.php?id=dms:cluster

Nom	Processeur Intel(R)	Fréquence (GHz)	Mémoire (Go)	Coeurs physiques	Coeurs logiques
`dms1`	Xeon(R) E5450	3.00	16	8	8
`dms2`	Xeon(R) E5450	3.00	16	8	8
`dms3`	Xeon(R) E5450	3.00	8	8	8
`dms4`	Xeon(R) E5450	3.00	8	8	8
`dms5`	Xeon(R) E5450	3.00	8	8	8
`dms6`	Xeon(R) E5450	3.00	8	8	8
`dms7`	Xeon(R) E5450	3.00	8	8	8
`dms8`	Xeon(R) E5450	3.00	8	8	8
`dms9`	Xeon(R) E5450	3.00	8	8	8
`dms10`	Xeon(R) E5450	3.00	8	8	8
`dms11`	Xeon(R) X5570	2.93	96	8	16
`dms12`	Xeon(R) X5680	3.33	192	12	24
`maitre`	Xeon(R) E5410	2.33	8	4	4

Notons que l'ordinateur `maitre` est moins puissant que les autres et il ne doit pas être surchargé. Il pourrait potentiellement y avoir beaucoup de gens qui y sont simultanément connectés pour lancer des tâches. Il est préférable d'éviter de faire du calcul directement sur cette machine.

5.2 Préparation à l'utilisation de la grappe de calcul du DMS

Pour réaliser des calculs R en parallèle sur la grappe de calcul du DMS, il faut d'abord rencontrer les conditions techniques suivantes.

i. Avoir accès au Réseau de données de l'Université Laval (RESUL)

Sur le campus de l'Université Laval, il est simple d'avoir accès au RESUL. Dans les laboratoires informatiques (et la plupart des bureaux), les ordinateurs sont connectés par câble au réseau. La connexion sans fil **eduroam** donne aussi accès aux ressources internes de l'Université Laval.

Hors campus, il est possible de se connecter à distance au RESUL à l'aide d'un VPN. Des informations à ce sujet peuvent être trouvées sur la page web https://www.dti.ulaval.ca/connexion-au-reseau-de-lul/reseau-distance. Il faut avoir un accès à distance par logiciel (Cisco AnyConnect Secure Mobility Client).

ii. Avoir accès à `maitre`

Pour utiliser la grappe de calcul du DSM, il faut aussi avoir :

a. un compte sur le réseau informatique du DMS ;
b. l'accès à la machine `maitre`.

Tous les étudiants et employés du DMS ont automatiquement un compte sur le réseau informatique du DMS. Cependant, pour avoir accès à la machine à `maitre`, il faut en faire la demande à l'administrateur du réseau, Michel Lapointe (michel.lapointe@mat.ulaval.ca).

iii. Avoir un outil pour lancer des protocoles `ssh`

SSH est un protocole de communication sécurisé permettant de se connecter à distance à un ordinateur sur système d'exploitation Linux/UNIX. Ce protocole sera utilisé pour établir une connexion à distance à la machine `maitre` du réseau informatique du DMS.

La plupart des ordinateurs sous Linux/UNIX ou Mac OS comporte un client `ssh` par défaut. Cependant, sous Windows, il faut installer un outil pour lancer des protocoles SSH, tel que **PuTTY**. L'utilisation de cet outil est illustré ici. PuTTY est téléchargeable à partir du site web http://www.putty.org/ (télécharger l'installateur MSI ou directement l'exécutable `putty.exe`).

Lors d'une première connexion à un hôte, peu importe le système d'exploitation, il faut accepter de poursuivre la connexion même s'il est impossible d'établir l'authenticité de l'hôte. Cette autorisation a pour effet d'ajouter cet hôte à la liste des hôtes reconnus. Ainsi, lors d'une prochaine connexion au même hôte, aucune autorisation n'est demandée.

Voici comment se connecter à `maitre` selon le système d'exploitation de l'ordinateur à partir duquel la connexion est établie. Le nom public complet de la machine `maitre` est `maitre.mat.ulaval.ca`.

Connexion à `maitre` à partir d'un terminal Linux/UNIX ou Mac OS :

- Lancer la commande suivante (avec le bon nom d'usager au lieu de `sbaillar`).

```
ssh maitre.mat.ulaval.ca -l sbaillar
```

Connexion à `maitre` à partir de Windows avec PuTTY :

- Démarrer PuTTY (double cliquer sur `putty.exe`) ;
- entrer « `maitre.mat.ulaval.ca` » dans le champ « Host Name » (voir figure 8) ;
- s'assurer que SSH est sélectionné comme type de connexion (port 22 OK) ;
- cliquer sur « Open ».

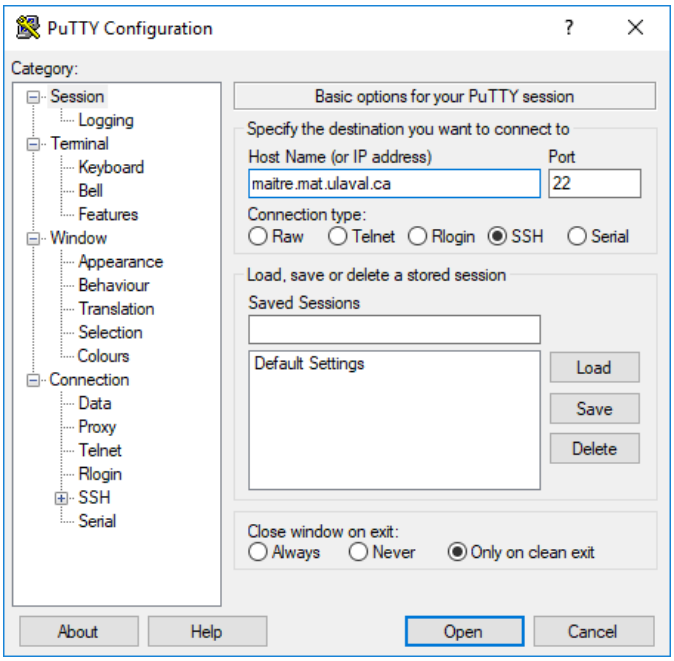

FIGURE 8 – Fenêtre PuTTY lors de la connexion à maitre

Il faut fournir :

- son nom d'usager sur le réseau du DMS, s'il n'a pas déjà été fourni,
- son mot de passe sur le réseau du DMS.

```
login as: sbaillar
sbaillar@maitre.mat.ulaval.ca's password:
Last login: Thu Jul 27 10:09:05 2017 from 132.203.87.234
Thu Jul 27 10:22:19 EDT 2017
/users/sbaillar
maitre
```

iv. Pouvoir se connecter aux noeuds `dms1` à `dms12` à partir de `maitre` sans mot de passe

Afin de pouvoir se connecter aux noeuds `dms1` à `dms12` à partir de `maitre` sans mot de passe, il faut réaliser les étapes suivantes à partir de `maitre`.

a. Lancer la commande suivante : `ssh-keygen`

```
maitre$ ssh-keygen
```

La commande crée, dans le répertoire `/users/sbaillar/.ssh` (avec le bon nom d'usager au lieu de `sbaillar`), les fichiers `id_rsa` et `id_rsa.pub`, qui contiennent des clés (la première privée, la deuxième publique).

b. Copier le contenu du fichier `id_rsa.pub` dans un fichier nommé `authorized_keys`. Pour ce faire, il faut d'abord se positionner dans le répertoire `/users/sbaillar/.ssh` avec la commande `cd`.

```
maitre$ cd .ssh
```

Ensuite, si le fichier `authorized_keys` n'existe pas encore dans le répertoire `/users/sbaillar/.ssh`, il faut utiliser la commande `cat` comme suit :

```
maitre$ cat id_rsa.pub > authorized_keys
```

si le fichier existe déjà, il faut plutôt utiliser la commande `cat` suivante :

```
maitre$ cat id_rsa.pub >> authorized_keys
```

Notons que, dans les commandes précédentes, `maitre$` représente l'invite de commandes du terminal lorsque nous sommes connectés sur la machine portant ce nom. Cette séquence de caractères ne fait pas vraiment partie de la commande, mais elle a été incluse ici pour indiquer clairement que la commande doit être soumise à partir de la machine `maitre`.

v. Avoir les autorisations pour se connecter aux noeuds dms1 à dms12 à partir de `maitre`

Cette étape est en fait facultative, mais elle facilite le travail par la suite. Il faut, soit préalablement, soit en cours de calcul, obtenir les autorisations pour se connecter aux noeuds `dms1` à `dms12` à partir de `maitre`. Il s'agit du même type d'autorisation que celui mentionné à la section 5.2-iii, soit celle à fournir lors d'une première connexion SSH à un hôte.

Pour fournir préalablement toutes ces autorisations, il suffit de se connecter une première fois à tous les noeuds `dms` que nous souhaitons utiliser, à partir de `maitre`, afin de fournir des autorisations. Voici les étapes à effectuer pour les 12 noeuds :

a. Lancer la commande : `ssh dms12`
(où `dms12` doit être remplacé par le nom de noeud pour l'itération en cours).

```
maitre$ ssh dms12
```

Si une erreur de type « `Host key verification failed` » survient, c'est que la clé du noeud hôte a changé. Pour l'effacer, lancer la commande : `ssh-keygen -R dms12`

```
maitre$ ssh-keygen -R dms12
```

puis lancer de nouveau la commande `ssh` de connexion au noeud.

b. Fournir l'autorisation en répondant `yes` à une question de ce type :

```
The authenticity of host 'dms12 (10.1.1.21)' can't be established.
RSA key fingerprint is a5:9b:8d:bd:35:89:9e:d7:f0:f7:4c:77:eb:bf:f8:32.
Are you sure you want to continue connecting (yes/no)?
```

c. Se déconnecter du noeud avec le commande `exit`

```
dms12$ exit
```

vi. Avoir préalablement installé tous les packages R requis

Si les calculs à effectuer requièrent des fonctions ou des données provenant de packages non inclus dans la distribution de base de R, il faut préalablement installer ces packages. Cette étape doit être réalisée une seule fois, sur `maitre` ou n'importe quel noeud de la grappe. Le choix de la machine n'importe pas, car les packages seront installés dans le compte de l'utilisateur sur le réseau informatique du DMS. Tous les ordinateurs de la grappe de calcul ont accès à ce compte (système de fichiers partagé).

Pour télécharger un package à partir du CRAN, puis l'installer, le plus simple est d'ouvrir une session R avec la commande `R`.

```
maitre$ R
```

Dans l'exemple ci-dessus, la commande est lancée à partir de la machine `maitre`. La commande `R`, lancé dans un terminal Linux, ouvre une session R directement dans le terminal, en utilisant le même répertoire courant en R que dans le terminal Linux au moment de lancer la commande.

Ensuite, dans la session R, il faut appeler la fonction `install.packages`. S'il s'agit de la toute première fois qu'on tente d'installer un package sur le réseau informatique du DMS, la commande suivante est appropriée.

```r
install.packages("aplore3")
```

La soumission de cette commande générera le message suivant :

```
Installing package into ‘/usr/lib64/R’
(as ‘lib’ is unspecified)
Warning in install.packages("aplore3") :
  'lib = "/usr/lib64/R"' is not writable
Would you like to use a personal library instead?  (y/n)
```

Lorsque nous ne possédons pas encore de répertoire personnel pour les packages R dans notre compte, nous pouvons répondre `y` (pour `yes`) à cette question et laisser R procéder à la création d'un tel répertoire. Ce répertoire devrait être automatiquement ajouté à la liste des chemins de recherche pour les packages, ce que nous pouvons vérifier avec la commande suivante.

```
.libPaths()
```

Si nous possédons déjà un répertoire personnel pour les packages R dans notre compte, qui se trouve déjà dans la liste des chemins de recherche pour les packages, il faut le spécifier dans l'appel à la fonction `install.packages` comme suit.

```
install.packages("aplore3", lib = "~/Rlib_x86_64")
```

Dans le chemin `"~/Rlib_x86_64"` de l'exemple précédent, le tilde (`~`) représente le répertoire personnel de base de l'utilisateur (en anglais le *home*), par exemple `/users/sbaillar`.

Les deux appels à `install.packages` précédents pousseront R à nous demander de sélectionner interactivement un miroir du CRAN. Une fois l'installation complété avec succès, la session R peut être fermée comme suit.

```
q()
```

Si le package à installer ne provient pas du CRAN, mais que nous possédons plutôt son fichier source .tar.gz quelque part dans notre compte, l'appel à la fonction `install.packages` doit contenir l'argument `repos = NULL`. Aussi, la valeur fournie au premier argument doit maintenant être le chemin d'accès au fichier source, comme dans l'exemple suivant.

```
install.packages("~/aplore3_0.9.tar.gz", lib = "~/Rlib_x86_64", repos = NULL)
```

5.3 Lancement de calculs R parallèles sur la grappe de serveurs du DMS

Une fois les étapes préalables complétées, nous sommes prêts à lancer des calculs R en parallèle sur la grappe de calcul du DMS. Il est possible de lancer les calculs soit dans une session R interactive, soit en ligne de commande. Le grand avantage de la ligne de commande est la possibilité d'exécuter la commande en arrière-plan, grâce à un signe `&` placé à la fin de la commande. Ainsi, si la connexion à l'ordinateur à partir duquel les calculs sont lancés est interrompue pendant les calculs, ceux-ci ne sont pas arrêtés. Ils le sont cependant si une telle situation survient lorsque les calculs ont été lancés interactivement à partir d'une session R ou en ligne de commande sans utiliser le `&`. Étant donné que les connexions interrompues ne sont pas rares, il est vraiment pratique, voire même essentiel pour les longs calculs, de lancer les commandes de calcul en arrière-plan. C'est ce qui sera illustré ici.

Le lancement de calculs R parallèles sur la grappe de serveurs du DMS comporte plusieurs étapes. Il faut :

1. être connecté à `maitre` ;
2. vérifier l'état des noeuds de la grappe de serveurs du DMS afin de choisir quels noeuds exploiter ;
3. rassembler les instructions pour lancer les calculs en un seul fichier, nommé programme ou script R :
   - ce script doit correctement initialiser la grappe de processus R pour les calculs en parallèle ;
4. transférer une copie du script, et de tous les fichiers dont il dépend, de l'ordinateur local vers le système de fichiers du réseau Linux du DMS ;
5. soumettre le script en ligne de commande dans le terminal ;
6. transférer les fichiers de résultats du système de fichiers du réseau du DMS vers l'ordinateur local ;
7. terminer la connexion à `maitre`.

Voici de l'information détaillée concernant ces étapes.

1) Connexion à `maitre`

a. Se connecter au réseau de l'Université Laval.
b. Se connecter à `maitre` (procédure décrite à la section 5.2-iii).

2) Vérification de l'état des noeuds de la grappe de serveurs du DMS

Avant de lancer un calcul sur un des noeuds de la grappe, il faut vérifier s'il est déjà utilisé ou non. Lancer un calcul sur des coeurs déjà en train d'être exploités a pour effet de ralentir les processus déjà en cours d'exécution sur ces coeurs et la réalisation du calcul lancé. La commande `qstat -f`, soumise à partir de `maitre`, permet d'obtenir de l'information sur l'utilisation des noeuds.

```
maitre$ qstat -f
```

Exemple de sortie obtenue :

```
queuename                      qtype resv/used/tot. load_avg arch          states
---------------------------------------------------------------------------------
all.q@dms1                     BIP   0/0/8          8.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms10                    BIP   0/0/8          0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms11                    BIP   0/0/16         0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms12                    BIP   0/0/24         24.08    lx-amd64
---------------------------------------------------------------------------------
all.q@dms2                     BIP   0/0/8          8.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms3                     BIP   0/0/8          8.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms4                     BIP   0/0/8          8.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms5                     BIP   0/0/8          0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms6                     BIP   0/0/8          0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms7                     BIP   0/0/8          0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms8                     BIP   0/0/8          0.01     lx-amd64
---------------------------------------------------------------------------------
all.q@dms9                     BIP   0/0/8          0.01     lx-amd64
```

Intéressons-nous à la variable `load_avg` dans ce tableau. Le *load average* désigne, sous les systèmes Linux/UNIX, une moyenne de la charge système, soit une mesure de la quantité de travail effectuée par le système, pendant une certaine période de temps. La sortie précédente nous dit que les noeuds `dms1` à `dms4` et `dms12` sont en train d'être utilisés à leur pleine capacité (charge système environ égale au nombre total de coeurs), alors que la charge système des autres noeuds est à peu près nulle. La dernière colonne du tableau dans la sortie pourrait nous indiquer que l'état de certain noeud est problématique, ce qui n'est pas le cas ici.

Pour en savoir plus sur l'utilisation d'un noeud en particulier, il est possible de s'y connecter par une commande `ssh`, puis d'afficher les processus en cours sur la machine avec la commande `top` (pour retrouver l'invite de commandes dans le terminal après un `top`, tapez `q`). Pour demander à la commande `top` d'afficher uniquement les processus d'un utilisateur en particulier, il faut ajouter l'option `-u` à la commande `top`, comme dans l'exemple suivant : `top -u sbaillar`. La commande `lscpu` retourne quant à elle des spécifications de la machine, dont le nombre de coeurs de calcul, qui peut aussi être obtenu avec la commande `nproc`.

À partir de ces informations, nous pouvons choisir quels noeuds utiliser pour nos calculs en parallèle. Tel que mentionné précédemment, si un des noeuds est déjà utilisé, il est préférable de ne pas lui envoyer de calculs, car il sera plus lent qu'un noeud libre. Aussi, il faut s'assurer que l'état des noeuds utilisés n'est pas problématique. Ici, nous pourrions envisager utiliser les noeuds `dms5` à `dms11`, mais nous nous contenterons d'exploiter `dms9`, `dms10` et `dms11`.

3) Préparation du script R de calcul

Reprenons la problématique présentée à la section 3.1. Rassemblons les instructions requises pour le lancement des calculs en parallèle en un seul script, le suivant.

```r
# Chargement du package parallel et du package contenant les données
library(parallel)
library(aplore3)

# Création de la liste de formules pour les modèles à ajuster
matIndic <- do.call(expand.grid, args = rep(list(c(FALSE, TRUE)), times = 8))
matIndic <- as.matrix(matIndic[-1, ])
colnames(matIndic) <- c("age", "lwt", "race", "smoke", "ptl", "ht", "ui", "ftv")
formules <- vector(mode = "list", length = nrow(matIndic))
for (i in 1:nrow(matIndic)){
  vars <- colnames(matIndic)[matIndic[i,]]
  formules[[i]] <- as.formula(paste("low ~", paste(vars, collapse = " + ")))
}

# Initialisation de la grappe de processus R
spec <- c(rep(paste0("dms", 9:10), each = 8), rep("dms11", 16))
grappe <- makeCluster(spec)

# Calculs
AICs_par <- parSapply(
  cl = grappe,
  X = formules,
  FUN = function(x, data) {
    AIC(glm(formula = x, data = data, family = binomial))
  },
  data = lowbwt
)
names(AICs_par) <- as.character(unlist(formules))
print(sort(AICs_par))
```

```
# Fermeture de la grappe de processus R
stopCluster(grappe)
```

Adaptation de l'appel à la fonction `makeCluster`

Seules quelques instructions sont nouvelles ou différentes dans ce script, par rapport à ce qui avait été utilisé à la section 4.1. Il s'agit d'abord des instructions pour l'initialisation de la grappe de processus R. Comme lors de l'initialisation d'une grappe de calcul locale, il faut utiliser la fonction `makeCluster`. Cependant, il ne suffit plus de donner en argument le nombre de processus parallèles à exploiter. Il faut maintenant identifier les ordinateurs hôte sur lesquels faire rouler les processus de la grappe. Cette information doit être fournie par l'argument `spec`.

Cependant, notons d'abord que nous utilisons toujours le type de grappe de calcul par défaut avec le package `parallel`, soit le type `PSOCK`. Cet argument n'a donc pas besoin d'être fourni en entrée lors de l'appel à la fonction `makeCluster`. Selon la fiche d'aide de cette fonction du package `parallel`, ce type de grappe est décrit ainsi :

> *It runs Rscript on the specified host(s) to set up a worker process which listens on a socket for expressions to evaluate, and returns the results (as serialized objects).*

Appelons ce type de grappe « grappe par connecteurs » (en anglais *socket cluster*).

La fiche d'aide de la fonction `makeCluster` du package `parallel` n'est pas très bavarde à propos du format que doit avoir la valeur fournie à l'argument `spec`. Cependant, cette fonction est en fait une version améliorée de la même fonction dans le package `snow`. La fiche d'aide de la fonction `makeCluster` du package `snow` renvoie à la page web http://homepage.divms.uiowa.edu/~luke/R/cluster/cluster.html pour plus d'informations. On y apprend que pour une grappe par connecteurs, la valeur fournie à l'argument `spec` peut être un vecteur contenant autant d'éléments que de processus désirés dans la grappe. Chaque élément doit être une chaîne de caractère contenant le nom de l'ordinateur hôte sur lequel faire rouler le processus. Ce nom pourrait aussi être remplacé par l'adresse IP de l'ordinateur hôte.

Dans le script précédent, la valeur fournie à l'argument `spec` lors de l'appel à la fonction `makeCluster` est la suivante :

```
spec <- c(rep(paste0("dms", 9:10), each = 8), rep("dms11", 16))
spec
```

```
##  [1] "dms9"  "dms9"  "dms9"  "dms9"  "dms9"  "dms9"  "dms9"  "dms9"
##  [9] "dms10" "dms10" "dms10" "dms10" "dms10" "dms10" "dms10" "dms10"
## [17] "dms11" "dms11" "dms11" "dms11" "dms11" "dms11" "dms11" "dms11"
## [25] "dms11" "dms11" "dms11" "dms11" "dms11" "dms11" "dms11" "dms11"
```

Il s'agit d'un vecteur de longueur 32. La grappe créée comptera donc 32 processus, dont 8 sur l'hôte `dms9`, 8 autres sur l'hôte `dms10` et 16 sur l'hôte `dms11`. Ces hôtes avaient été sélectionnés à l'étape précédente.

Instructions pour l'impression des résultats

Les autres instructions nouvelles dans le script sont les suivantes (à la fin de la section *Calculs*).

```
names(AICs_par) <- as.character(unlist(formules))
print(sort(AICs_par))
```

Elles permettent d'imprimer intelligemment les résultats, soit en présentant les valeurs de AIC obtenues en ordre croissant et accompagnées des formules auxquelles elles sont associées.

4) Transfert du script R et des fichiers dont il dépend

Supposons que le script R créé à l'étape 1 soit dans le fichier `SelectionVariables.R`, sur l'ordinateur à partir duquel nous travaillons. Il faut transférer ce fichier dans notre compte sur le réseau informatique du DMS. Si ce script dépendait d'autres fichiers, par exemple de fichiers de données, il faudrait aussi les transférer. Ici, ce n'est pas le cas.

Voyons maintenant comment transférer un fichier vers la machine `maitre` du réseau Linux du DMS. Rappelons que le fichier transféré sera accessible à partir de toutes les machines de la grappe de serveurs du DMS.

La procédure dépend encore une fois du système d'exploitation de l'ordinateur local. Les ordinateurs sous Linux/UNIX ou Mac OS comportent presque toujours un client `scp`, pour le transfert sécurisé de fichiers (utilise un protocole SSH). Cependant, sous Windows, il faut encore une fois installer un outil. Le programme **PuTTY**, mentionné précédemment, permet de réaliser le transfert, mais il n'est pas le plus convivial des outils pour cette tâche. Il est plus simple d'utiliser le logiciel **WinSCP**, téléchargeable à partir du site web https://winscp.net/eng/download.php (conseil : télécharger « *Installation package* »).

Rappelons qu'il est normal qu'une autorisation soit demandée lors d'une première connexion à un hôte.

Transfert à partir d'un terminal Linux/Unix ou Mac OS :

- Lancer une commande `scp` telle la suivante, à partir du répertoire contenant localement le fichier à transférer (avec le bon nom d'usager au lieu de `sbaillar` et le répertoire d'accueil désiré).

```
scp SelectionVariables.R sbaillar@maitre.mat.ulaval.ca:~/R
```

Transfert à partir de Windows avec WinSCP :

- Démarrer WinSCP ;
- s'assurer que SFTP est sélectionné comme « File protocole » ;
- entrer (voir figure 9) :
  - « `maitre.mat.ulaval.ca` » dans le champ « Host name »,
  - 22 dans le champ « Port number »
  - le bon nom d'usager dans le champ « User name »,
  - le mot de passe de cet usager dans le champ « Password » ;

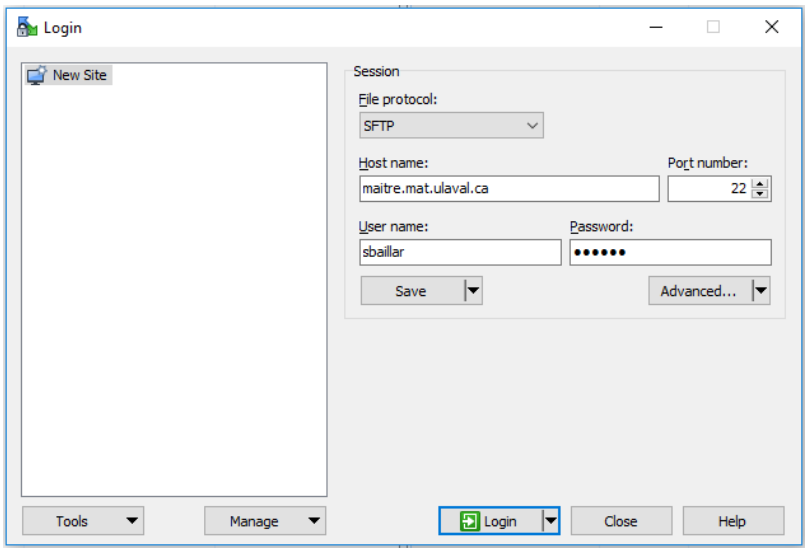

FIGURE 9 – Fenêtre de connexion WinSCP

- cliquer sur « Login » ;

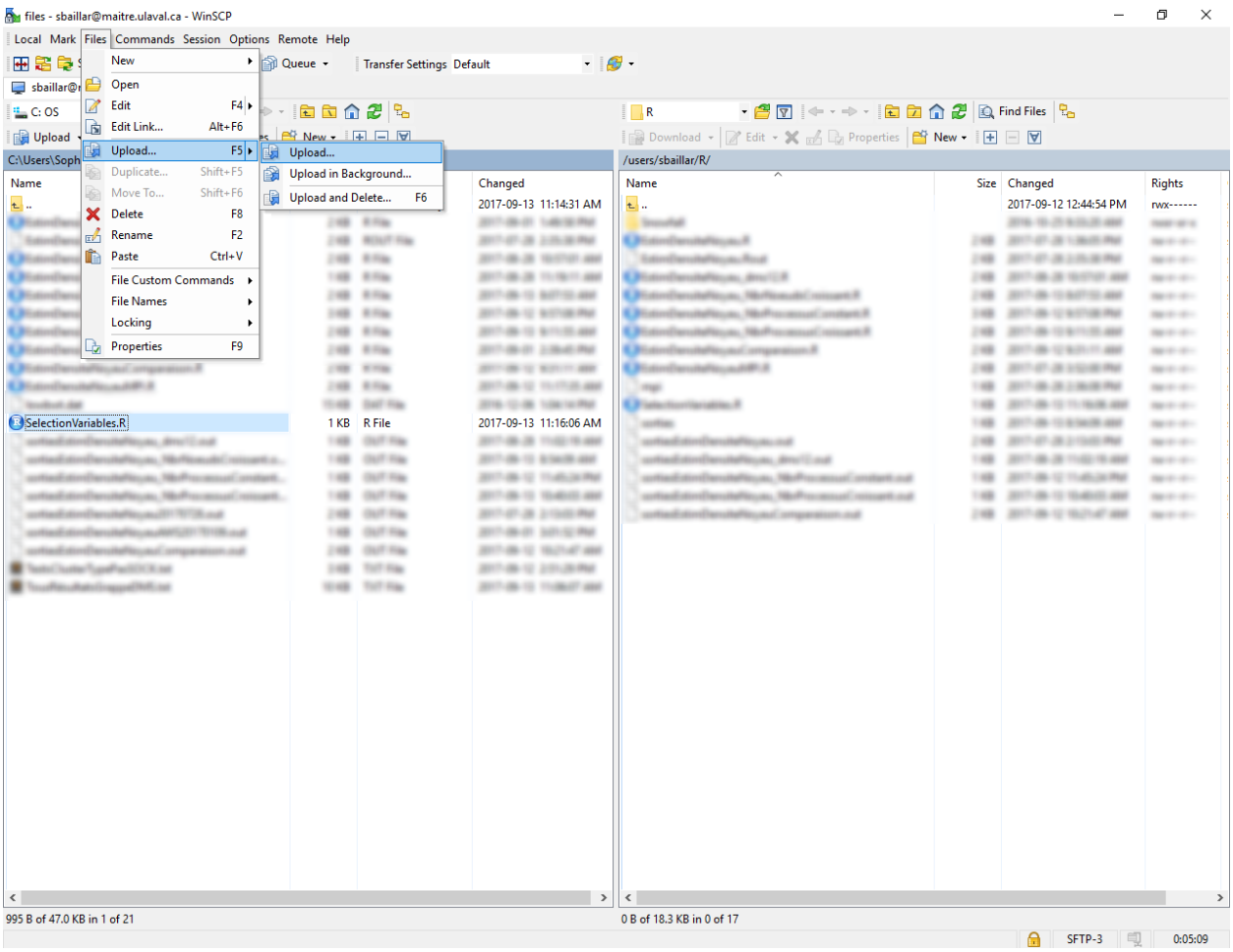

FIGURE 10 – Fenêtre WinSCP lorsque connecté

- se positionner (voir figure 10) :
  - localement (sous-fenêtre de gauche) dans le répertoire contenant le fichier à transférer,
  - à distance (sous-fenêtre de droite) dans le répertoire où nous souhaitons transférer le fichier ;
- sélectionner le fichier à transférer ;
- aller dans le menu « File > Upload… » et sélectionner « Upload… » (voir figure 10), ou cliquer sur le bouton « Upload », ou appuyer sur F5, etc. ;
- cliquer sur « OK ».

5) Soumission du script R de calcul

Il est conseillé de lancer les calculs à partir d'un des noeuds que nous souhaitons exploiter plutôt qu'à partir de `maitre`, car `maitre` est le point d'accès à la grappe de serveurs et il ne doit pas être surchargé. De plus, il est moins rapide que les noeuds de la grappe. Alors connectons-nous d'abord à un des noeuds présélectionné, par une commande `ssh` telle la suivante :

```
maitre$ ssh dms11
```

Pour soumettre le script, il est plus simple de d'abord se positionner dans le répertoire contenant le fichier du script, avec la commande `cd`.

```
dms11$ cd R
```

Ensuite, le script est soumis, de préférence avec la commande `Rscript`. Il y a quelques années, cette commande n'existait pas. On soumettait plutôt des scripts R en ligne de commande avec `R CMD BATCH`. De nos jours, la commande `Rscript` est préférée à `R CMD BATCH`. Voici un exemple de commande pour soumettre le script `SelectionVariables.R`.

```
dms11$ Rscript --vanilla SelectionVariables.R > sortiesSelectionVariables.out &
```

Dans cette commande, l'option `--vanilla` est facultative, mais elle permet, entre autres, de s'assurer de travailler à partir d'un environnement courant de session R initialement vide. Ensuite, le nom du fichier contenant le script R est mentionné. L'option `> sortiesSelectionVariables.out` permet de rediriger les sorties dans un fichier externe (ici `sortiesSelectionVariables.out`) plutôt que de les afficher dans le terminal. Finalement, le signe `&` à la toute fin permet d'exécuter la commande en arrière-plan, tel qu'expliqué au début de cette section.

6) Transfert de fichiers de sorties et résultats au besoin

Lorsque les calculs sont terminés, nous souhaitons souvent transférer les fichiers produits en sortie du réseau informatique du DMS vers notre ordinateur local. Il suffit de procéder comme à la section 5.3-4, mais en effectuant un transfert en direction inverse, soit de `maitre` vers notre ordinateur local.

Lorsque l'ordinateur local est sous Linux/UNIX ou Mac OS, il suffit d'utiliser une commande `scp` (lancée à partir de l'ordinateur local) telle que celle-ci.

```
scp sbaillar@maitre.mat.ulaval.ca:~/R/sortiesSelectionVariables.out ~
```

Dans cette commande, le premier argument doit identifier le fichier à transférer (celui sur l'hôte distant) et le deuxième argument est l'emplacement sur l'ordinateur local où l'on souhaite accueillir le fichier (ci-dessus seulement `~`).

Pour un ordinateur local sous Windows, l'utilisation de `WinSCP` pour effectuer le transfert est très similaire à la procédure décrite au point 4).

7) Terminaison de la connexion à `maitre`

La connexion à `dms11`, puis à `maitre` peut maintenant être terminée avec la commande `exit`.

```
dms11$ exit
```

```
maitre$ exit
```

5.4 Comparaison de temps d'exécution avec différentes grappes de processus R déployées sur la grappe de serveurs du DMS

Des expérimentations ont été effectuées pour comprendre les gains en temps d'exécution que l'utilisation de la grappe de serveurs de calcul du DMS peut permettre d'obtenir. Comme à la section 4.3, la problématique d'estimation de densité par noyau décrite à la section section 3.2 a été utilisée pour ces expérimentations. La procédure d'expérimentation et les résultats obtenus sont présentés ici. Le script R utilisé pour lancer les expérimentations se trouve à l'annexe B.

Cas 1) Différents nombres de processus R, sur un seul serveur

La figure 11 présente des temps de calcul médians obtenus localement sur le serveur `dms11`, avec des grappes contenant un nombre croissant de processus R. Il s'agit en fait de l'équivalent de la figure 5, mais en exploitant un ordinateur plus puissant pour les calculs que l'ordinateur personnel 2 utilisé à la section 4.3. Le serveur `dms11` possède 16 coeurs logiques, mais seulement 8 coeurs physiques. En raison des observations faites à la section 4.3, je n'ai pas tenté d'utiliser une grappe à plus de 16 processus parallèles sur cet ordinateur.

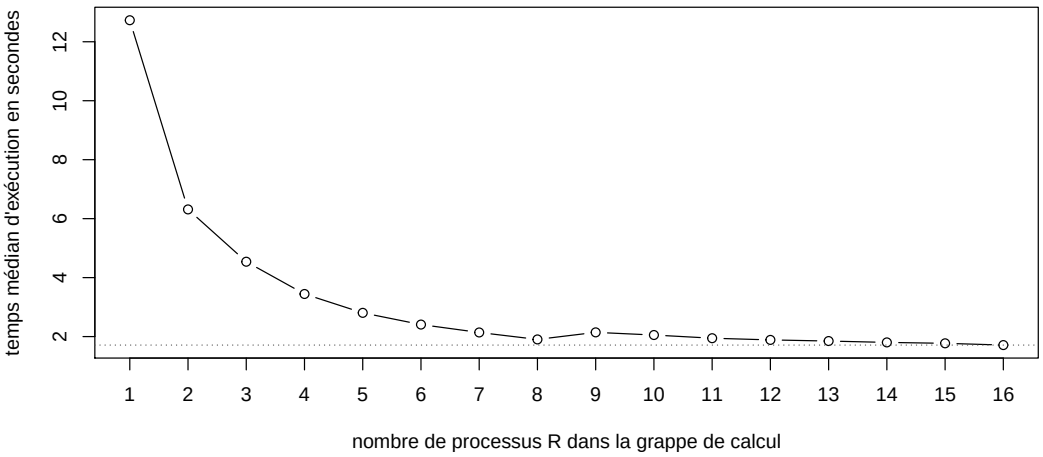

FIGURE 11 – Temps de calcul médians, en secondes, sur dms11, en fonction du nombre de processus R parallèles exploités

Le calcul effectué ici prend 12.62 secondes à rouler séquentiellement. Ce temps est équivalent au temps du calcul avec des outils de calcul parallèle, mais sur une grappe d'un seul processus R (12.73 secondes, premier point de la figure 11). Le temps de calcul diminue de façon progressive en passant de 1 à 8 processus R parallèles, soit le nombre de coeurs physiques du serveur. Augmenter le nombre de processus parallèles au-delà de 8 ne permet pas de réduire de façon notable le temps de calcul.

Cas 2) Nombre fixe de processus R, distribués sur différents nombres de serveurs

Les résultats précédents ont été obtenus sur une seule machine. La vraie nouveauté ici est l'utilisation simultanée de plusieurs ordinateurs. Le tableau 4 présente des temps d'exécution pour des calculs effectués sur une grappe de 8 processus R parallèles. Cependant, ces processus sont distribués sur un nombre croissant de serveurs, en partant d'un seul en allant jusqu'à 8 serveurs (1 processus par serveur dans ce cas).

TABLE 4: Temps de calcul médians, en secondes, en fonction du nombre de serveurs exploités pour une grappe de 8 processus R

nombre de serveurs	1	2	3	4	5	6	7	8
temps	2.69846	2.61850	2.59964	2.64480	2.61652	2.61140	2.61451	2.66634

Ces temps sont tous similaires (autour de 2.65 secondes). Remarquons que le serveur `dms11` était plus rapide avec une grappe de 8 processus R (1.91 secondes). Ici, ce sont les serveurs `dms1` à `dms10`, à l'exception des serveurs `dms4` et `dms8`, qui sont exploités. Les serveurs exploités étaient tous libres lorsque les calculs ont été lancés. Ces serveurs ont tous les mêmes caractéristiques et sont plus vieux que `dms11`. Ainsi, il semble que le fait de distribuer les processus de la grappe de calcul sur plusieurs serveurs ne ralentit pas les calculs. Les communications entre serveurs sont donc rapides.

Lacement des calcul à partir d'un serveur non utilisé par la grappe de processus R

Les résultats présentés dans le tableau 4 ont été obtenus en initialisant les grappes de calcul à partir du serveur `dms1`. Toutes ces grappes ont au moins un processus roulant sur ce serveur. Est-ce qu'initialiser une grappe de calcul à partir d'un serveur non exploité dans la grappe induit une certaine perte de temps en communication ? J'ai mesuré le temps de calcul sur une grappe de 8 processus R parallèles, tous sur le serveur `dms2`, mais en faisant rouler le script d'initialisation des calculs sur `dms1`. J'ai obtenu un temps médian pour 100 exécutions de 3.00 secondes. C'est un peu plus long que les 2.70 secondes obtenues avec une grappe similaire de 8 processus parallèles tous déployés sur un serveur de même capacité que `dms2`, soit `dms1`, mais en initialisant la grappe sur ce même serveur.

Cas 3) Autant de processus R que de coeurs de calcul, sur différents nombres de serveurs

Qu'arrive-t-il maintenant avec le temps de calcul si on augmente le nombre de serveurs utilisés, mais cette fois en exploitant toujours au maximum les coeurs de calcul des serveurs ?

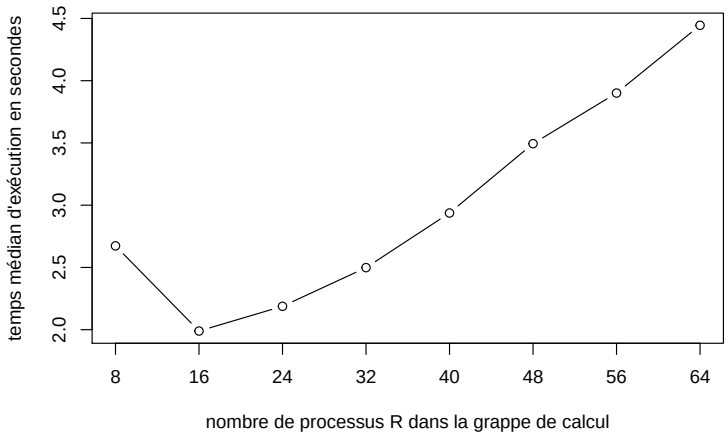

FIGURE 12 – Temps de calcul médians, en secondes, en fonction du nombre de processus R parallèles exploités, en distribuant ces processus par groupes de 8 sur différents serveurs à 8 coeurs

La figure 12 présente les temps de calcul d'une grappe de 8 processus sur 1 serveur, à une grappe de 16 processus sur 2 serveurs, jusqu'à une grappe de 64 processus sur 8 serveurs, en se limitant toujours à 8 processus par serveur (sur des serveurs similaires à 8 coeurs physiques). Le temps médian le plus petit est obtenu avec une grappe de 16 processus sur 2 serveurs. En utilisant une grappe plus grosse, le calcul se met à être plus lent. Ce résultat est probablement explicable par plus de temps inactif. Rappelons que le temps inactif se définit par du temps consacré aux communications, qui ne contribue pas directement à la progression des tâches, mais qui a un impact sur le temps total de calcul. Il peut survenir lorsque le réseau de communication rencontre des problèmes de congestion (en anglais *bottleneck*). Par exemple, si trop de processus cherchent à communiquer leurs résultats en même temps à la session R à partir de laquelle les calculs ont été initialisés, certains processus peuvent se retrouver à perdre beaucoup de temps à attendre leur tour pour retourner leurs résultats. Ainsi, même si en théorie plus on exploite de coeurs dans un calcul en parallèle, plus le temps total de calcul devrait être petit, il arrive que l'utilisation de plus de coeurs cause un ralentissement du calcul plutôt qu'une accélération.

Ce phénomène est observé ici parce que le calcul réalisé dans les expérimentations ne prend que quelques secondes à rouler. Le temps inactif est aussi de l'ordre de secondes et devient à un certain point plus grand que le temps réellement consacré à faire progresser le calcul. Si le calcul à faire était plus long et nécessitait réellement le recours au calcul en parallèle, par exemple si son temps d'exécution était de l'ordre de quelques heures, le temps inactif prendrait un temps négligeable en comparaison. Dans ce cas, il est probable que l'augmentation du nombre de processus dans la grappe de calcul continuerait de faire diminuer le temps global de calcul au delà de 16 processus.

6 Calcul R en parallèle avec Amazon EC2

Selon Wikipedia, le *cloud computing* se définit ainsi :

> Le cloud computing, ou l'informatique en nuage ou nuagique ou encore l'infonuagique (au Canada francophone), est l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'intermédiaire d'un réseau, généralement internet.

Selon cette définition, l'utilisation à distance de la grappe de serveurs du DMS est donc une forme de cloud computing. Cependant, l'accès à ces serveurs n'est pas public. Il est réservé aux employés et étudiants du Département de mathématiques et de statistique.

Plusieurs compagnies proposent des plate-forme cloud d'accès public, mais payant. Cette section présente comment utiliser la plate-forme Amazon Web Services pour faire du calcul R en parallèle. Nous aurions aussi pu utiliser Google Cloud Platform, Microsoft Azure ou une autre des nombreuses plate-formes existantes.

Amazon Web Services, ou AWS, est l'une des plate-formes cloud les plus anciennes et connues. AWS offre plusieurs produits, dont certains pour le calcul, d'autres pour le stockage de données et bien plus. Nous nous intéressons ici à Amazon Elastic Compute Cloud (EC2), qui permet de démarrer des machines virtuelles (en d'autres mots des serveurs virtuels) dans le cloud et de, notamment, s'y connecter à distance pour y lancer des calculs. Ces machines virtuelles Amazon EC2 sont appelées instances.

Les sous-sections suivantes expliquent comment démarrer et travailler avec des instances Linux Amazon EC2. Il y aura quelques similarités avec l'utilisation de la grappe de serveurs du DMS en raison du choix d'utiliser Linux. Amazon EC2 offre aussi des instances Windows. Cependant, pour le lancement de calculs scientifiques, il est plus courant et pratique d'utiliser Linux. Dans le présent document, une instance Amazon EC2 réfère toujours à une machine virtuelle Linux.

La première étape à compléter pour lancer des calculs sur Amazon EC2 est la création et la configuration d'un compte AWS. Ensuite, il est avantageux de créer un modèle de machine virtuelle contenant tous les programmes requis pour nos calculs. Ce modèle de machine virtuelle est appelé AMI, pour *Amazon Machine Images*, dans la terminologie d'Amazon. Finalement, des calculs peuvent être lancés sur une ou plusieurs instances de l'AMI créée préalablement.

Dans cette section, les exemples inclus pour illustrer la procédure se distinguent du reste du texte en étant encadrés. Ils présentent comment j'ai procédé à l'automne 2017 pour faire du calcul R en parallèle sur Amazon EC2. Pour reproduire ces exemples, vous devez adapter les noms employés à votre situation.

6.1 Configuration d'un compte AWS

Afin de pouvoir utiliser Amazon EC2, il faut d'abord posséder un compte AWS. Aussi, il est conseillé de configurer préalablement ce compte afin de faciliter certaines étapes futures. Ces configurations préalables ne sont pas obligatoires, mais recommandées par Amazon. Effectuer ces configurations est une étape technique que nous avons heureusement besoin de réaliser une seule fois.

Les étapes proposées ici sont celles proposées dans la documentation officielle d'Amazon EC2 :

> Configuration avec Amazon EC2 :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

Je vous invite à suivre les étapes présentées sur cette page. Cette sous-section est un complément à cette documentation, contenant des exemples et explications supplémentaires.

Notons, avant de commencer, que la traduction française de la documentation de AWS n'est pas toujours très claire. Il faut dire qu'il y a beaucoup de termes techniques, potentiellement nouveaux, à apprivoiser. En cas de doute sur la traduction, il vaut mieux lire la documentation en anglais. Sur tout le site de AWS, comme dans la console AWS que nous utiliserons bientôt, passer d'une langue à l'autre est très simple. Il suffit de sélectionner la langue désirée dans un menu dans le haut ou le bas de la page.

Notons aussi qu'on trouve sur le web une grande quantité de tutoriels maison concernant l'utilisation d'Amazon EC2. Certains de ces tutoriels ne mentionnent pas ces étapes, ou mentionnent des étapes différentes. Le principal danger avec ces tutoriels est qu'ils ne soient pas à jour (tout comme ce tutoriel le deviendra dans quelques temps). AWS a évolué au fil des ans et continue d'évoluer. La documentation officielle est, en principe, toujours à jour. Alors il vaut mieux s'y fier que de se fier à un blog datant de plus de 5 ans.

Voici donc des informations complémentaires à la documentation d'Amazon. Il est essentiel de lire la documentation d'Amazon « Configuration avec Amazon EC2 » en même temps que cette sous-section.

1) Création d'un compte AWS

Tout d'abord, il faut s'inscrire à AWS. Pour ce faire, rendez-vous sur la page web https://aws.amazon.com/fr/, puis cliquez sur le bouton en haut à droite « S'inscrire » ou sur le bouton « Créez un compte gratuit » dans l'encadré. Vous serez redirigé vers une page web sécurisée. Il suffit maintenant de suivre les étapes indiquées. La page web suivante détaille ces étapes :

> Comment créer et activer un nouveau compte Amazon Web Services ? : https://aws.amazon.com/fr/premiumsupport/knowledge-center/create-and-activate-aws-account/

Il faut fournir :

- un numéro de carte de crédit,
- un numéro de téléphone auquel il est possible de vous rejoindre dans les minutes suivantes.

L'ouverture d'un compte est gratuite, c'est l'utilisation des produits et services AWS qui ne l'est pas. Il faut néanmoins fournir tout de suite un numéro de carte de crédit.

Lors de l'ouverture d'un compte AWS, certains produits sont offerts gratuitement pendant 12 mois. C'est le cas de certaines instances Amazon EC2 qui ne possèdent qu'un seul coeur de calcul. Nous aurons besoin d'instances à plus de coeurs pour effectuer du calcul en parallèle. L'utilisation d'instances intéressantes pour le calcul en parallèle n'est pas très dispendieuse. Des exemples de coûts seront présentés plus loin.

Mon inscription à AWS s'est faite avec l'adresse courriel `sophie.baillargeon@mat.ulaval.ca` et mon identifiant de compte, dont j'aurai besoin à l'étape suivante, est 341019361899.

2) Création d'un utilisateur IAM

Plusieurs utilisateurs peuvent être associés à un même compte AWS. Ils sont nommés utilisateurs IAM, pour *Identity and Access Management*. Ces utilisateurs peuvent avoir différentes autorisations d'accès. Même si vous planifiez être le seul utilisateur de votre compte, je vous propose de suivre les instructions du tutoriel « Configuration avec Amazon EC2 » dans la section « Créer un utilisateur IAM » et de vous créer un utilisateur IAM. C'est ce qu'Amazon recommande de faire.

En suivant ces instructions, j'ai créé un utilisateur portant le nom `sophie`, que j'ai ajouté à un nouveau groupe nommé `admin`, qui utilise la « stratégie » `AdministratorAccess`.

Nous créerons à l'étape 5 un « groupe de sécurité ». Il ne s'agit pas de la même chose que le groupe `admin` créé ici. La terminologie AWS est parfois embêtante au début, mais avec le temps on finit par s'y retrouver.

Ensuite, je me suis déconnectée de mon compte AWS et je me suis reconnectée sur l'URL https://341019361899.signin.aws.amazon.com/console/ avec l'utilisateur IAM que je venais de créer.

La documentation suivante peut vous apporter, au besoin, plus d'informations au sujet des utilisateurs IAM :

> Informations d'identification de sécurité AWS :
> http://docs.aws.amazon.com/fr_fr/general/latest/gr/aws-security-credentials.html.

3) Création d'une paire de clés

Après avoir démarré des instances Amazon EC2, nous voudrons nous y connecter à distance, tout comme nous l'avons fait à la section 5 avec la machine `maitre` du réseau informatique du DMS. Cependant, au lieu d'utiliser des mots de passe pour sécuriser les connexions à distance sur ses machines virtuelles, Amazon utilise des paires de clés. Une des clés de la paire doit se retrouver sur l'ordinateur à partir duquel nous nous connectons. Notons que nous avions aussi utilisé des paires de clé à la section 5.2-iv pour arriver à nous connecter sans mot de passe aux serveurs `dms1` à `dms12` à partir de `maitre` sur le réseau informatique du DMS.

Il faut maintenant suivre les instructions du tutoriel « Configuration avec Amazon EC2 » dans la section « Créer une paire de clés ».

> J'ai suivi ces instructions pour me créer une paire de clé. J'ai conservé la région qui m'était proposée par défaut, soit « USA Ouest (Oregon) », même s'il ne s'agit pas de la région la plus près de chez moi, car elle est associée à de bas tarifs. Le choix de la région a plus d'impact, par exemple, pour ceux qui hébergent des sites web sur des instances Amazon EC2 que pour ceux qui y effectuent du calcul scientifique. Il aurait tout de même été peut-être préférable que je sélectionne une région plus près de moi.
>
> J'ai nommé ma paire de clé `sophie-key-pair-uswest2` et enregistré le fichier de clé privée, nommé `sophie-key-pair-uswest2.perm`, dans le répertoire `C:\Users\Sophie\.ssh\`.

Cet emplacement a été choisi, car c'est celui suggéré dans le tutoriel suivant d'Amazon.

> Lancement d'une machine virtuelle Linux :
> https://aws.amazon.com/fr/getting-started/tutorials/launch-a-virtual-machine/

Attention, dans le présent document nous ne suivons pas ce tutoriel, mais libre à vous de le compléter pour vous familiariser avec Amazon EC2. Ici, nous utilisons des connexions à distance mieux sécurisées que celles employées dans ce tutoriel.

Préparation du fichier de clé privée sur Linux/Unix ou Mac OS

Lorsque notre ordinateur local est sous Linux/UNIX ou Mac OS, Amazon suggère d'enregistrer le fichier de clé privée dans le répertoire `~/.ssh/` (rappelons que `~` représente le répertoire personnel de base de l'utilisateur).

Il est aussi recommandé de s'assurer que le fichier de clé privée n'est pas visible de façon publique grâce à une commande `chmod` telle la suivante.

```
chmod 400 ~/.ssh/sophie-key-pair-uswest2.perm
```

Préparation du fichier de clé privée sur Windows

Lorsque notre ordinateur local est sous Windows, Amazon suggère aussi d'enregistrer le fichier de clé privée dans un répertoire nommé `.ssh` positionné dans le répertoire personnel de base de l'utilisateur.

Pour ceux qui souhaitent éventuellement utiliser PuTTY pour se connecter à distance à une instance Amazon EC2, il faut aussi suivre les instructions de la sous-section intitulée « Pour préparer la connexion à une instance Linux depuis Windows avec PuTTY » du tutoriel « Configuration avec Amazon EC2 ». Elles servent à convertir le fichier de clé privé de format `.pem` en un fichier `.ppk`, dont PuTTY a besoin.

Puisque j'ai suivi ces instructions, mon répertoire `C:\Users\Sophie\.ssh\` contient le fichier `sophie-key-pair-uswest2.ppk` en plus du fichier `sophie-key-pair-uswest2.perm`.

4) Création d'un Virtual Private Cloud (VPC)

Les utilisateurs d'Amazon EC2 qui désirent avoir un meilleur contrôle sur les adresses IP de leurs instances peuvent utiliser les Amazon *Virtual Private Cloud* (VPC). Ce n'est pas notre cas, donc nous pouvons sauter l'étape 4. Nous utiliserons le VPC qu'Amazon nous fournit par défaut.

5) Création d'un groupe de sécurité

Un groupe de sécurité permet de contrôler qui peut se connecter à distance à une instance Amazon EC2. Si la connexion est ouverte à tous, ce qui est le cas par défaut, il suffit de posséder le nom d'hôte public de l'instance (nous y reviendrons) et la clé privée afin d'arriver à établir la connexion avec l'instance. Les groupes de sécurité offrent une couche de protection supplémentaire. Après tout, l'utilisation de nos instances nous sera facturé peu importe d'où on s'y connecte. Il est donc bon de s'assurer que seules les personnes autorisées arrivent à se connecter. Pour ceux qui stockent des données privées sur des instances, cette couche de sécurité supplémentaire est essentielle.

Un groupe de sécurité permet donc de restreindre l'accès à une instance à une seule ou à un petit groupe d'adresses IP publiques. Une adresse IP (pour *Internet Protocol*) est un numéro d'identification attribué à tous les ordinateurs connectés à un réseau informatique utilisant l'*Internet Protocol*. Par exemple, tous les appareils connectés à mon wifi (pour *wireless local area networking*) domestique ont la même adresse IP publique sur le web. Les appareils connectés au wifi de mon voisin ont une autre adresse IP. Cependant, lorsque je me connecte à distance au réseau de l'Université Laval via un VPN, l'adresse IP de l'ordinateur que j'utilise change pour en prendre une associée au réseau de l'université.

Le tutoriel « Configuration avec Amazon EC2 » dans la section « Créer un groupe de sécurité » explique comment créer un groupe de sécurité.

J'ai créé un groupe de sécurité nommé `sophie-SG-uswest2`, avec description « SG pour IAM sophie, region uswest2 » et VPC par défaut. Dans un premier temps, j'ai mis dans le groupe seulement une règle entrante SSH avec mon IP détecté automatiquement. Lorsque j'ai créé ce groupe, j'étais chez moi et mon ordinateur portable n'était pas connecté au réseau de l'Université Laval via un VPN. C'est donc l'adresse IP associée à mon wifi domestique qui a été incluse dans le groupe. Étant donné que je travaille parfois de la maison et parfois de l'université, j'ai ensuite ajouté au groupe `sophie-SG-uswest2` une règle entrante SSH avec mon IP lorsque je travaille du Département de mathématiques et de statistique.

De façon général, lors de la connexion à une instance, il faut simplement s'assurer que l'adresse IP associée à l'ordinateur utilisé est acceptée par le groupe de sécurité de l'instance. Il faut donc que le groupe que vous créez contienne des règles entrantes pour tous les réseaux à partir desquels vous êtes susceptibles de vouloir vous connecter à une instance Amazon EC2.

Le groupe de sécurité utilisé par défaut sur Amazon EC2, dont le nom débute par `launch-wizard`, est en fait ouvert à toutes les adresses IP. Alors si la couche de sécurité supplémentaire apportée par les groupes de sécurité vous importe peu, vous pouvez toujours sauter la présente étape et utiliser le groupe de sécurité `launch-wizard`.

6.2 Informations concernant les produits offerts par Amazon EC2

Amazon EC2 permet donc de démarrer des machines virtuelles dans la cloud et de les utiliser à distance. Ces machines virtuelles, nommées instances, peuvent être de différents types. C'est le type de l'instance qui détermine les caractéristiques de celle-ci, telle que :

- le type de processeur,
- le nombre de coeur de calculs,
- l'espace mémoire,
- le type de stockage permanent,
- etc.

Le type de l'instance détermine aussi sont coût d'utilisation à la demande. Typiquement, plus une instance est puissante ou contient de l'espace de stockage, plus elle est dispendieuse à utiliser.

Types d'instance Amazon EC2

Amazon EC2 offre des types d'instances à plusieurs coeurs de calcul. La page web suivante propose une description des types d'instances offerts par Amazon.

> Types d'instances Amazon EC2 : https://aws.amazon.com/fr/ec2/instance-types/

Les coûts d'utilisation à la demande de ces types d'instances, selon la région, sont présentés sur la page web suivante.

> Tarification à la demande Amazon EC2 : https://aws.amazon.com/fr/ec2/pricing/on-demand/

Certains types sont optimisés pour le calcul. Par exemple, le type `c4.8xlarge` comporte 36 coeurs de calcul virtuels (en anglais *virtual CPU*). Il ne faut pas s'attendre à autant de performance d'un coeur virtuel que d'un coeur physique. Un coeur virtuel est probablement comparable à un coeur logique en ce qui concerne les performances qu'il permet d'atteindre.

Facturation

Pour voir les informations relatives à la facturation de l'utilisation de ses instances Amazon EC2, il faut se connecter à son compte AWS avec son nom d'utilisateur de compte global AWS et non avec un nom d'utilisateur IAM. Pour arriver à faire ça à partir d'une URL similaire à https://341019361899.signin.aws.amazon.com/console/, il faut cliquer sur le lien en bas du bouton « Connexion » intitulé « Identifiez-vous à l'aide de vos informations de connexion au compte racine ». Les informations se trouvent dans le service « Facturation »

Il existe quatre méthode de tarification Amazon EC2, décrites sur le page web suivante.

> Tarification Amazon EC2 : https://aws.amazon.com/fr/ec2/pricing/

Nous exploiterons ici la tarification à la demande. Depuis le 2 octobre 2017, l'utilisation d'instances avec cette méthode de tarification est facturé à la seconde, avec une facturation minimale de 60 secondes. Les frais sont chargés sur la carte de crédit dont le numéro à été fourni à l'ouverture du compte AWS.

6.3 Configuration d'une AMI pour des calculs en R

Une fois les configurations de notre compte AWS complétées, nous avons tout en main pour démarrer des machines virtuelles Amazon EC2. Toute machine virtuelle Amazon EC2 est en fait une instance d'une AMI. Comme mentionné en introduction, l'acronyme AMI signifie *Amazon Machine Image*. Une AMI est en quelque sorte un modèle de machine virtuelle, contenant un système d'exploitation et certaines applications. Nous avons besoin d'une AMI Linux sur laquelle sont installés le logiciel R ainsi que tous les packages R requis pour nos calculs.

Amazon propose plusieurs AMI, certaines sur Linux et d'autres sur Windows. Les AMI contiennent déjà quelques applications, mais aucune ne contient le logiciel R.

Il est possible de créer nos propres AMI et de les partager avec les autres. On retrouve dans la console AWS des AMI créées par d'autres utilisateurs d'Amazon EC2 et rendues disponibles à tous. Certaines de ces AMI possèdent R. C'est le cas, par exemple, des AMI créée par Louis Aslett et présentées sur la page web http://www.louisaslett.com/RStudio_AMI/. Ces AMI sont souvent citées par les utilisateurs de R sur Amazon EC2. Par contre, dans notre cas, elles ne répondent pas tout à fait à nos besoins. Elles sont conçues dans le but d'une utilisation de R via un serveur RStudio. De plus, les packages R dont nous avons besoin n'y sont peut-être pas tous installés.

Ici, nous allons plutôt voir comment créer notre propre AMI, qui répondra parfaitement à nos besoins. Pour ce faire, il faut lancer une instance d'une AMI de base, y faire toutes les installations requises, puis créer une image de notre instance. Nous suivrons en partie les instructions présentées dans la documentation suivante :

> Démarrage sur les instances Linux Amazon EC2 :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/EC2_GetStarted.html

i. Prérequis : être connecté à son compte AWS

Avant de commencer, assurez-vous d'être connecté à votre compte AWS.

> De mon côté, je me suis connectée dans un navigateur web à partir de l'URL https://341019361899.signin.aws.amazon.com/console/ en utilisant mon nom d'utilisateur IAM `sophie` et le mot de passe associé.

Cette connexion ouvre la console AWS. Dans celle-ci, il faut d'abord se rendre dans le service EC2 (*EC2 Management Console*).

1) Lancement d'une instance

> À cette étape initiale, je n'ai pas lancé une instance de la même AMI que celle mentionnée dans le tutoriel « Démarrage sur les instances Linux Amazon EC2 » à la section « Étape 1 : Lancement d'une instance ». J'ai plutôt sélectionné l'AMI « Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-6e1a0117 », parce qu'Ubuntu est le système d'exploitation Linux avec lequel je suis la plus familière.

Il est inutile de sélectionner un type d'instance puissant pour l'instant. À ce stade-ci, nous ne sommes pas encore prêts à faire des calculs. Nous allons seulement effectuer des installations de logiciels. Alors choisissions le type `t2.micro`, gratuit pour ceux qui ont ouvert leur compte AWS il y a moins de 12 mois.

> Finalement, j'ai sélectionné :
>
> - le groupe de sécurité `sophie-SG-uswest2` et
> - la paire de clé `sophie-key-pair-uswest2`, que j'avais tous deux créés précédemment.

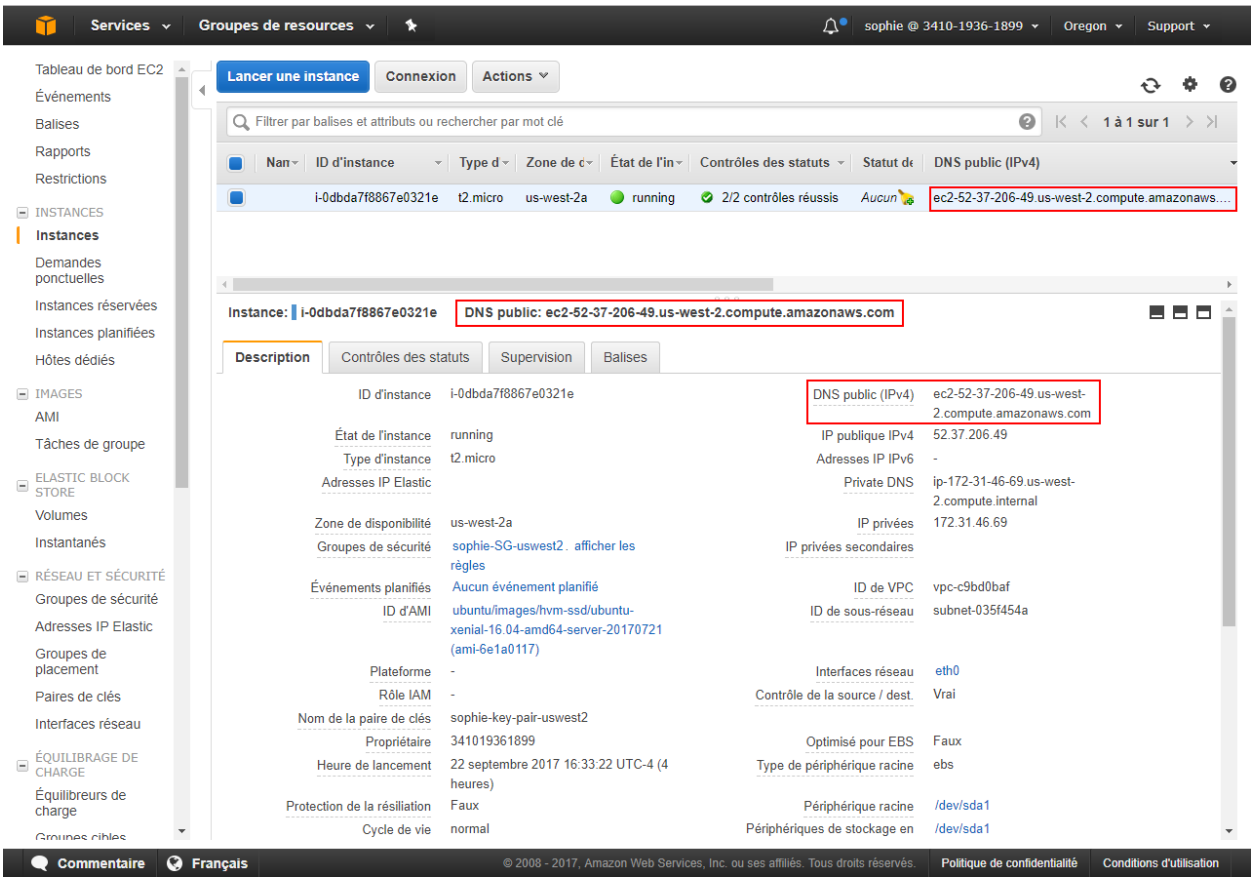

FIGURE 13 – Exemple de repérage du DNS public (rectangle rouge) dans l'onglet « Instances » de la console AWS EC2

Démarrer une instance prend toujours quelques minutes. En cliquant sur « Afficher les instances », nous sommes redirigés vers l'onglet « Instances » de la console AWS EC2. Lorsque le démarrage de l'instance est complété, le champ « Contrôle des statuts » prend la valeur « 2/2 contrôles réussis ».

2) Connexion à l'instance à distance

Le tutoriel « Démarrage sur les instances Linux Amazon EC2 » suggère de se connecter à l'instance à partir d'un navigateur web. Cependant, Java doit être installé et activé dans le navigateur utilisé. Certain navigateurs, par exemple Chrome, ne prennent plus en charge NPAPI, la technologie requise pour les applets Java. Ici, nous allons donc plutôt nous connecter à distance à l'instance à l'aide d'un protocole SSH, comme nous l'avons fait à la section 5 pour nous connecter à un ordinateur du réseau informatique du DMS.

La procédure à suivre dépend encore une fois du système d'exploitation de l'ordinateur à partir duquel nous souhaitons établir la connexion. Cependant, peu importe le système d'exploitation, nous avons besoin de connaître le nom d'hôte public de l'instance. Ce nom est le DNS public de l'instance, qui peut être trouvé dans la console AWS EC2, sous l'onglet « Instances » (voir figure 13).

Le DNS public de l'instance que j'ai démarrée est :
`ec2-54-69-150-24.us-west-2.compute.amazonaws.com`.

Nous devrons utiliser un nom d'hôte précédé d'un nom d'utilisateur, du format `nom_utilisateur@dns_public`. Ce nom d'utilisateur n'est pas votre nom d'utilisateur IAM. Il dépend plutôt de l'AMI choisi. Pour une instance d'une AMI Ubuntu, il faut utiliser le nom d'utilisateur `ubuntu`. La section « Lancement d'une session PuTTY » de la page de documentation « Connexion à votre instance Linux à partir de Windows à l'aide de PuTTY » contient une liste des noms d'utilisateur à employer en fonction de la sorte d'AMI utilisé.

> Ainsi, lors d'une connexion SSH, le nom d'hôte complet à utiliser dans mon cas est : `ubuntu@ec2-52-37-206-49.us-west-2.compute.amazonaws.com`.

Le tutoriel d'Amazon Lancement d'une machine virtuelle Linux explique comment procéder pour établir la connexion à une instance selon le système d'exploitation. Cependant, dans ce tutoriel, l'adresse IP de l'instance est utilisée au lieu de son DNS public. Les deux façons de faire sont possibles.

Rappelons que peu importe le système d'exploitation, il est normal qu'une autorisation soit demandée lors d'une première connexion à un hôte.

Connexion à une instance Amazon EC2 à partir d'un terminal Linux/Unix ou Mac OS :

La page web suivante de la documentation d'Amazon explique comment procéder pour se connecter à une instance Amazon EC2 à partir d'un terminal Linux/UNIX ou Mac OS.

> Connexion à votre instance Linux à l'aide de SSH :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

En résumé, si l'étape préalable de rendre le fichier de clé privée non visible publiquement a bien été effectuée (voir section 6.1-3), il suffit de soumettre une commande SSH. Cette commande doit inclure, en plus du nom d'hôte complet, le nom du fichier contenant le clé privée sur l'ordinateur local.

> Voici le commande SSH que j'aurais employée si je travaillais sous Linux/UNIX ou Mac OS.
>
> ```
> ssh -i "sophie-key-pair-uswest2.pem" \
> ubuntu@ec2-52-37-206-49.us-west-2.compute.amazonaws.com
> ```
>
> Cette commande aurait dû être lancée à partir du répertoire courant contenant le fichier de clé privée `sophie-key-pair-uswest2.pem`, ou être modifiée pour spécifier le chemin d'accès complet au fichier de clé privée.

Connexion à une instance Amazon EC2 à partir de Windows avec PuTTY :

Comme à la section 5.2-iii, nous allons utiliser PuTTY pour établir une connexion à distance à partir de Windows, cette fois avec une instance Amazon EC2 plutôt qu'avec un serveur du réseau informatique du DMS. La documentation suivante indique comment se connecter à une instance Amazon EC2 à partir de Windows avec PuTTY.

> Connexion à votre instance Linux à partir de Windows à l'aide de PuTTY :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/putty.html

La section « Lancement d'une session PuTTY » contient les étapes à suivre (l'étape 1 peut être sautée). La procédure de connexion à une instance Amazon EC2 avec PuTTY est un peu plus compliquée que la procédure décrite à la section 5.2-iii du présent tutoriel. C'est l'utilisation d'une paire de clé au lieu d'un mot de passe qui complique un peu les choses. Mais en suivant bien les étapes décrite dans la documentation d'Amazon, ça fonctionne.

Dans le champ « Host Name » de la fenêtre PuTTY, j'ai indiqué :
`ubuntu@ec2-54-69-150-24.us-west-2.compute.amazonaws.com`.

La première fois qu'une connexion à une instance Amazon EC2 est établie avec une paire de clé, il est utile de sauvegarder la session PuTTY avant de cliquer sur « Open ». Ainsi, il ne sera pas nécessaire de localiser de nouveau cette clé privée à chaque nouvelle connexion.

Une fois connecter à notre instance, nous pouvons interagir avec elle.

Remarque : Un autre outil proposé par Amazon pour lancer des protocoles SSH sous Windows est Git Bash. L'utilisation de cet outil est mentionnée dans le tutoriel Lancement d'une machine virtuelle Linux.

3) Installation de R et des packages nécessaires

Nous avons besoin que le logiciel R soit installé sur notre instance. Sur une instance d'une AMI Ubuntu, nous pouvons procéder à l'installation de R en suivant les instructions décrites sur cette page web :

http://basicgroundwork.blogspot.ca/2015/05/install-r-in-ubuntu-1404.html

La première étape est d'ajouter un des miroirs du CRAN à la liste des dépôts informatiques depuis lesquels notre instance peut acquérir des logiciels. Cette liste se trouve dans le fichier `/etc/apt/sources.list`. Nous allons y ajouter le miroir `http://cran.rstudio.com/bin/linux/ubuntu`, mais un autre miroir du CRAN aurait aussi fait l'affaire. Pour effectuer cet ajout, le blog cité plus haut suggère d'ouvrir le fichier avec `vi`, de l'éditer pour ajouter la ligne : `deb http://cran.rstudio.com/bin/linux/ubuntu trusty/`, puis d'enregistrer et fermer le fichier. Encore plus simplement, il est possible de réaliser cette édition en une seule commande soumise dans le terminal, la suivante.

```
printf 'deb http://cran.rstudio.com/bin/linux/ubuntu trusty/\n'| sudo tee -a \
/etc/apt/sources.list
```

Il faut ensuite mettre à jour la liste des fichiers disponibles dans les dépôts énumérés dans le fichier `/etc/apt/sources.list` avec la commande suivante.

```
sudo apt-get update
```

Si l'erreur suivante est générée :

```
W: GPG error: http://cran.rstudio.com/bin/linux/ubuntu trusty/ Release: The following
signatures couldn't be verified because the public key is not available: NO_PUBKEY
51716619E084DAB9
```

alors les commandes suivantes devraient régler le problème (en utilisant le même numéro de clé publique que dans l'erreur).

```
gpg --keyserver hkp://keyserver.ubuntu.com:80 --recv-key 51716619E084DAB9
gpg -a --export 51716619E084DAB9| sudo apt-key add -
sudo apt-get update
```

Finalement, nous sommes en mesure de procéder à l'installation de R avec la commande suivante.

```
sudo apt-get install r-base
```

Maintenant, nous pouvons effectuer l'installation des packages R non inclus dans l'installation de base dont nous avons besoin. Pour ce faire, le plus simple est d'ouvrir une session R avec la commande R.

```
R
```

Ensuite, nous pouvons utiliser la fonction `install.packages()` avec la liste des packages à installer, comme dans cet exemple de commande.

```
install.packages(c("aplore3", "microbenchmark"))
```

Nous devons accepter la création d'une librairie personnelle en répondant `y` (pour *yes*) aux questions suivantes :

```
Installing packages into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
Warning in install.packages(c("aplore3", "microbenchmark")) :
  'lib = "/usr/local/lib/R/site-library"' is not writable
Would you like to use a personal library instead?  (y/n)
```

```
Would you like to create a personal library
~/R/x86_64-pc-linux-gnu-library/3.4
to install packages into?  (y/n)
```

puis sélectionner un miroir du CRAN. Les installations durent parfois quelques minutes. La session R peut ensuite être fermée avec la commande `q()`.

Si jamais d'autres installations sont requises pour les calculs, c'est le temps de les effectuer. Lorsque notre instance est apte à rouler nos calculs, nous allons en faire une image.

4) Création d'une image de l'instance

Dans l'onglet « Instances » de la console AWS EC2 (comme dans la figure 13), il est simple de créer l'image d'une instance. Il suffit de la sélectionner, puis d'aller dans le menu « Actions > Image > Créer l'image ». Il faut fournir un nom et une description pour l'image. Nous pouvons aussi choisir la taille du stockage qui sera associé aux instances de l'image (par défaut 8 gibioctets).

> J'ai donné le nom `ubuntu-R` à mon image. Son identifiant est `ami-df8471a7`.

Pour en apprendre davantage sur les AMI, la documentation suivante est utile.

> Amazon Machine Images (AMI) :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/AMIs.html

5) Terminaison de la connexion et résiliation de l'instance

Nous pouvons maintenant mettre fin à notre connexion à distance à l'instance à l'aide de la commande `exit` dans le terminal. Encore plus important, l'instance utilisée pour créer notre AMI devrait être résiliée. Nous démarrerons d'autres instances de notre AMI, en utilisant des types d'instances à plus de coeurs de calculs que le type `t2.micro`.

Pour résilier une instance dans l'onglet « Instances » de la console AWS EC2 (comme dans la figure 13), il faut la sélectionner, puis aller dans le menu « Actions > État de l'instance > Résilier » (en anglais *Terminate*).

La résiliation d'une instance y met fin complètement (la détruit). Il est aussi possible d'arrêter une instance de façon temporaire afin de pouvoir la redémarrer plus tard. Cette action se nomme « Arrêter » (en anglais *Stop*). Une de ces deux actions est nécessaire pour mettre fin à la facturation de l'utilisation de l'instance. Tant qu'une instance roule (état *running*), même si aucun processus de calcul n'est exécuté par celle-ci, elle est facturée.

6.4 Lancement de calculs R parallèles sur une instance Amazon EC2

Étant donné le grand nombre de coeurs de calcul de certains types d'instance Amazon EC2, il est possible d'accélérer des calculs de façon appréciable en utilisant une seule instance Amazon EC2. Les étapes à suivre pour ce faire correspondent à celles présentées à la section 5.3 pour le lancement de calculs R parallèles sur la grappe de serveurs du DMS, à quelques différences mineures près.

Tout d'abord, la préparation du script de calcul peut être effectuée en premier, puisque que nous savons d'avance que le seul hôte de la grappe de processus R sera l'instance à laquelle nous serons connectés. En préparant le script avant de démarrer l'instance, nous pourrons minimiser le temps d'ouverture de l'instance, donc minimiser son coût d'utilisation.

Aussi, nous supposons être dans une situation où un seul utilisateur est en mesure de se connecter à une instance grâce aux mesures de sécurité prises. Cet utilisateur connaît donc l'état de l'instance puisqu'il le contrôle. Dans ce cas, il est inutile de vérifier l'état de l'instance. Il s'agit donc d'une étape de moins à effectuer. Par contre, il faut ajouter une nouvelle étape, à réaliser avant la connexion à une instance : le lancement de l'instance.

Toutes les étapes à réaliser sont détaillées ci-dessous. Pour minimiser les coûts de l'exemple réalisé ici, nous utiliserons un type d'instance optimisé pour le calcul, mais contenant seulement deux coeurs virtuels et coûtant autour de 0.10 $US par heure selon la région : le type `c4.large`. Nous aurions aussi pu choisir le type `c5.large`, aussi à deux coeurs virtuels et coûtant un peu moins cher d'utilisation (autour de 0.085 $US par heure). Cependant, ce type n'était pas encore disponible au moment de réaliser ces manipulations. Il est maintenant disponible (novembre 2017) dans certaines régions aux États-Unis, notamment dans la région « USA Ouest (Oregon) ».

i. Prérequis : être connecté à son compte AWS

Comme à la section 6.3, il faut d'abord être connecté à son compte AWS.

> Je me suis connectée dans un navigateur web à partir de l'URL https://341019361899.signin.aws.amazon.com/console/ en utilisant mon nom d'utilisateur IAM 'sophie' et le mot de passe associé.

Dans la console AWS, il faut se rendre dans le service EC2.

1) Préparation du script R de calcul

Encore une fois, nous travaillerons ici avec la problématique présentée à la section 3.1. Le script R, qui est nommé ici `SelectionVariablesAWS.R`, est le même que le script `SelectionVariables.R` de la section 5.3-3, à l'exception de deux lignes. Le bout suivant :

```
# Initialisation de la grappe de processus R
spec <- c(rep(paste0("dms", 9:10), each = 8), rep("dms11", 16))
grappe <- makeCluster(spec)
```

est remplacé par celui-ci :

```
# Initialisation de la grappe de processus R
grappe <- makeCluster(2)
```

Ainsi, seul l'appel à `makeCluster` diffère entre les deux scripts. Puisque nous exploiterons ici seulement l'instance à laquelle nous serons connectés, nous appelons la fonction `makeCluster` en lui donnant en entrée un seul argument : le nombre de processus R à inclure dans la grappe de calcul. Puisque nous démarrerons une instance de type `c4.large` à deux coeurs de calculs virtuels, nous demandons 2 processus R parallèles dans la grappe de calcul.

2) Lancement d'une instance de notre AMI pour des calculs en R

Dans la console Amazon EC2, allons dans l'onglet `AMI` à partir du menu à gauche. Sélectionnons l'AMI créée précédemment (dans mon cas `ami-df8471a7`). Ensuite, dans le menu « Actions » sélectionnons « Lancer ».

Notons aussi que nous aurions pu utiliser le bouton « Lancer une instance » du Tableau de bord EC2, puis sélectionner la bonne AMI.

Il est temps de sélectionner un type d'instance. Nous avons choisi d'utiliser une instance de type `c4.large`. Sélectionnons ce type, puis cliquons sur « Vérifier et lancer ».

Maintenant, cliquons sur le lien « Modifier les groupes de sécurité » à droite. Il faut cocher « Sélectionner un groupe de sécurité existant » et sélectionner le groupe de sécurité créé lors des configurations (dans mon cas `sophie-SG-uswest2`). Cliquons ensuite sur « Vérifier et lancer », puis sur « Lancer ».

Une fenêtre s'ouvre alors pour la sélection d'une paire de clé. Choisissions une paire de clé existante, soit celle créée à la section 6.1-3 (dans mon cas `sophie-key-pair-uswest2`). Cochons la case dans la bas pour attester que nous avons accès au fichier de clé privée de la paire sélectionnée. Finalement, cliquons sur « Lancer des instances ».

3) Connexion à l'instance à distance

Repérons d'abord le DNS public de l'instance que nous venons de démarrer (comme dans la figure 13).

> Dans mon cas, il s'agit de `ec2-35-164-150-87.us-west-2.compute.amazonaws.com`.

Pour se connecter à l'instance, il faut suivre la procédure décrite à la section 6.3-2.

Si nous sommes sous Windows et que nous avons sauvegardé la session PuTTY lors d'une connexion précédente à une instance Amazon EC2 avec notre fichier de clé privée, il suffit de :

- sélectionner la bonne session dans le champ « Saved session » ;
- cliquer sur le bouton « load » ;
- modifier le nom de l'hôte pour celui de la nouvelle instance ;
- cliquer sur « Open ».

4) Transfert du script R et des fichiers dont il dépend

Le transfert du script R, et des fichiers dont il dépend s'il y a lieu, vers une instance Amazon EC2 se réalise à peu près de la même façon qu'à la section 5.3-4 (transfert vers un ordinateur du réseau informatique du DMS). Encore une fois, les distinctions entre les deux approches proviennent de l'utilisation d'une paire de clé.

Transfert vers une instance Amazon EC2 à partir d'un terminal Linux/Unix ou Mac OS :

La procédure à suivre est documentée ici :

> Section Transfert de fichiers des instances Linux à partir de Linux à l'aide de SCP :
> http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

La commande `scp` doit identifier le fichier de clé privée en plus du fichier à transférer (provenant de l'ordinateur local) et de l'emplacement d'accueil pour le fichier sur l'hôte distant, comme dans l'exemple suivant.

```
scp -i ~/.ssh/sophie-key-pair-uswest2.pem ~/SelectionVariablesAWS.R \
ec2-user@ec2-198-51-100-1.compute-1.amazonaws.com:~
```

Transfert vers une instance Amazon EC2 à partir de Windows avec WinSCP :

La procédure à suivre en utilisant WinSCP est documentée ici :

> Section Transfert de fichiers vers votre instance Linux à l'aide de WinSCP :
> http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html

5) Soumission du script R de calcul

La soumissions du script R de calcul s'effectue exactement de la même façon que sur une ordinateur du réseau informatique du DMS (section 5.3-5). Il faut utiliser une commande `Rscript` telle que la suivante.

```
Rscript --vanilla SelectionVariablesAWS.R > sortiesSelectVariablesAWS.out &
```

6) Transfert de fichiers de sorties et résultats au besoin

Lorsque les calculs sont terminés, nous pouvons ramener les fichiers de sorties et résultats de notre instance vers notre ordinateur local. Il est important d'effectuer ce transfert avant de résilier l'instance, car les fichiers seront détruit lors de la résiliation.

Le transfert s'effectue à partir de l'ordinateur local, comme à la section 6.4-4, mais en direction inverse. Par exemple, pour un ordinateur local sous Linux/UNIX ou Mac OS, la commande `scp` serait similaire à la suivante :

```
scp -i ~/.ssh/sophie-key-pair-uswest2.pem \
ec2-user@ec2-198-51-100-1.compute-1.amazonaws.com:~/sortiesSelectVariablesAWS.out ~
```

7) Terminaison de la connexion et résiliation ou arrêt de l'instance

Si nos calculs sont terminés, il est inutile de laisser rouler les instances et de continuer de payer. Nous allons donc d'abord nous déconnecter de l'instance avec la commande `exit` dans le terminal. Puis, dans l'onglet « Instances » de la console AWS EC2, nous allons sélectionner l'instance, puis aller dans le menu « Actions > État de l'instance » et sélectionner :

- « Résilier » pour y mettre fin complètement ou
- « Arrêter » si nous souhaitons nous y reconnecter plus tard, ou ne pas détruire les fichiers qu'elle contient.

Est-ce possible d'utiliser plus d'une instance à la fois ?

Il est tout à fait possible de se créer une grappe de serveurs de calcul constituée de plusieurs instances Amazon EC2 pour effectuer du calcul R en parallèle. Par contre, il y a des étapes techniques à réaliser pour y arriver. Cette possibilité n'est pas documenté ici par manque de temps.

On retrouve sur le web quelques sources d'information sur le sujet, notamment Nicholson (2014 et 2015) et Mount (2016). Cependant, les bons tutoriels ne semblent pas très nombreux.

Quelques défis dans la mise en place d'une grappe d'instances Amazon EC2 sont :

- l'automatisation du démarrage, de l'identification, puis de la résiliation ou l'arrêt des instances, potentiellement nombreuses ;
- la réalisation d'étapes préalables pour permettre la communication entre l'instance maîtresse, soit celle à partir de laquelle les calculs seraient lancés, et les autres instances (paires de clés à obtenir un peu comme nous l'avons fait dans la section 5.2-iv).

Un outil facilitant la réalisation de ces étapes est l'interface de ligne de commande AWS (en anglais AWS *Command Line Interface* ou CLI). Cet outil est déjà installé sur les AMI Amazon Linux, mais pas sur les AMI Ubuntu (type d'AMI utilisé à la section 6.3). Ainsi, il semblerait avantageux, lors de la création d'une

grappe d'instances Amazon EC2, d'utiliser une AMI Amazon Linux, sur laquelle R et les packages requis par nos calculs seraient installés.

Exploiter plusieurs instances permettrait de multiplier le nombre de coeur de calculs utilisés en parallèle, donc potentiellement d'obtenir des accélérations de temps d'exécution vraiment grandes. Bien sûr, chaque instance de la grappe serait facturée individuellement. Il faudrait faire quelques recherches ou expérimentations pour cerner la configuration qui permettrait d'atteindre le meilleur compromis entre des performances maximisées, mais des coûts minimisés. Serait-il plus avantageux d'utiliser

- une grappe formée de plusieurs instances comportant peu de coeurs de calcul, ou plutôt
- une grappe formée de quelques instances comportant plusieurs coeurs de calcul ?

6.5 Comparaison de temps d'exécution avec différentes grappes de processus R déployées sur une intance Amazon EC2

Comme aux sections 4.3 et 5.4, la problématique d'estimation de densité par noyau décrite à la section section 3.2 a été exploitées pour effectuer des expérimentations de calcul avec une instance Amazon EC2. Les scripts R utilisés pour lancer les expérimentations sont similaires au script de l'annexe B. Seule la définition de l'objet `specs` diffère. La valeur qui a été assignée à cet objet est spécifiée dans le tableau 5.

Les expérimentations menées avec des instances Amazon EC2 ont permis de tester l'utilisation de différents nombres de processus R, mais toujours sur une seule instance à la fois. Trois types d'instances ont été testées. Elles sont décrites dans le tableau 5, où vCPU signifie « nombre de coeurs de calcul virtuels », le coût horaire est celui d'une utilisation à la demande dans le région « USA Ouest (Oregon) » et `specs` est la valeur attribuée à l'objet `specs` dans le script R de calcul (annexe B modifiée).

TABLE 5: Caractéristiques des trois types d'instances utilisées

Type d'instance	vCPU	Coût horaire	`specs`
`c4.large`	2	0.10 $US	`2:4`
`c4.8xlarge`	36	1.59 $US	`36`
`m4.16xlarge`	64	3.20 $US	`64`

Le tableau 6 présente les temps d'exécution obtenus pour les calculs séquentiels (1 processus R) et parallèles. Sur une instance de type `c4.large`, à deux coeurs virtuels, passer d'un calcul séquentiel à un calcul sur 2 processus R parallèles accélère le temps de calcul par un facteur de 1.57. Cependant, utiliser une grappe de calcul à plus de 2 processus R ralentit un peu les calculs comparativement à l'utilisation de seulement 2 processus. Ce résultat concorde avec celui présenté dans la figure 5. Il semble indiquer que le nombre optimal de processus R à inclure dans une grappe de calcul locale est égal au nombre total de coeurs de calcul (potentiellement logiques ou virtuels) de la machine exploitée.

L'instance de type `c4.8xlarge` est la plus rapide, parmi celle testées, pour le calcul séquentiel. Exploiter les 36 coeurs virtuels de l'instance avec 36 processus R parallèles accélère le temps de calcul par un facteur de 10.42. Ce facteur est loin d'être égal au nombre de coeurs virtuels. Comme il avait été soulevé précédemment, les coeurs virtuels des instances Amazon EC2 ne permettent pas des accélérations aussi importantes que des coeurs physiques.

L'instance de type `m4.16xlarge` a un facteur d'accélération du calcul parallèle similaire (10.35) à celui de l'instance de type `c4.8xlarge`, malgré l'exploitation d'un plus grand nombre de coeurs virtuels (64). Nous pourrions donc en conclure que l'utilisation d'instances d'un type dit optimisé pour le calcul (type dont le nom débute par `c`) est avantageux pour les calculs en parallèle.

TABLE 6: Temps de calcul médians, en secondes, en fonction du type d'instance Amazon EC2 utilisé et du nombre processus R parallèles dans la grappe de calcul

Type d'instance	Nombre de processus R	Temps
`c4.large`	calculs séquentiels : 1	11.65959
`c4.large`	2	7.436513
`c4.large`	3	7.731512
`c4.large`	4	7.667299
`c4.8xlarge`	calculs séquentiels : 1	11.14296
`c4.8xlarge`	36	1.06918
`m4.16xlarge`	calculs séquentiels : 1	12.036750
`m4.16xlarge`	64	1.162605

Notons que quelques semaines après avoir mené ces expérimentations, Amazon a rendu disponible, dans certaines régions, de nouveaux types d'instances optimisées pour le calcul : la série `c5`. Cette série comprend le type `c5.18.xlarge` à 72 coeurs virtuels au coût de 3.06 $US l'heure dans le région « USA Ouest (Oregon) ». Il est probable que ce type d'instance permettrait d'obtenir des accélérations entre plus importantes que celles obtenues ici.

7 Discussion et conclusion

Le calcul R en parallèle sur CPU a été approfondi dans ce document. Après avoir présenté quelques notions de base concernant le calcul de haute performance, l'écriture d'un programme R lançant des calculs en parallèle a été illustré. Le principal package R exploité ici pour réaliser de la programmation parallèle fut `parallel`. De plus, nous avons vu comment déployer une grappe de processus R sur

- un ordinateur multi-coeurs,
- la grappe de serveurs de calcul du DMS (Département de mathématiques et de statistique),
- une instance Amazon Elastic Compute Cloud (EC2).

Des expérimentations ont été menées afin de mieux comprendre les gains en temps d'exécution que peut apporter le calcul parallèle en R. Le tableau 7 rassemble certains des meilleurs résultats présentés dans les sections 4.3 (ordinateur personnel), 5.4 (grappe de serveurs du DMS) et 6.5 (instance Amazon EC2). Toutes les expérimentations effectuaient le même calcul d'estimation de densité par noyau (section 3.2). Le nombre de coeurs de calcul exploités, leurs types (physique, logique ou virtuel) ainsi que leurs capacités de calcul (qui dépendent du processeur) ont une influence sur les gains en temps d'exécution obtenus. En général, plus on exploite de coeurs, plus le calcul est rapide. Aussi, les coeurs logiques et virtuels (terminologie d'Amazon EC2) n'atteignent pas les mêmes performances que les coeurs physiques. Finalement, les différents processeurs sur le marché offrent des performances variables.

Les préparatifs techniques pour réaliser les expérimentations rapportées dans le tableau 7 vont des plus simples au plus complexes en allant de gauche à droite dans le tableau. Cependant, une fois tout configuré pour une première fois, le lancement des calculs a un niveau de complexité plutôt similaire d'un contexte à l'autre.

TABLE 7: Résumé de certains des meilleurs temps de calcul médians, en secondes, obtenus au cours des expérimentations

	PC 4	`dms11`	`dms1` + `dms2`	`c4.8xlarge`
calcul séquentiel	11.546	12.618	16.628	11.143
calcul parallèle	2.251	1.712	1.989	1.069
facteur d'accélération	5.13	7.37	8.36	10.42
nombre de coeurs exploités	7 sur 8	16 sur 16	2 x 8 sur 8	36 sur 36
types de coeurs exploités	logiques	logiques	physiques	virtuels
processeur(s)	Core(TM) i7-4790K	Xeon(R) X5570	Xeon(R) E5450	Xeon(R) E5-2660
(tous Intel(R))	4.00 GHz	2.93 GHz	3.00 GHz	2.60 GHz
connexion	locale	à distance	à distance	à distance

Travailler localement sur une seule machine est le contexte le plus simple. Cependant, nous n'avons pas tous la chance d'avoir physiquement accès à un ordinateur puissant. Et lancer des calculs en parallèle sur un ordinateur le ralentit. Si l'ordinateur roulant les calculs est aussi celui sur lequel nous souhaitons continuer de travailler, le ralentissement peut être problématique.

Il est souvent avantageux de lancer des calculs longs à distance, sur un (ou des) machine(s) autre(s) que celle que nous utilisons pour nos autres tâches informatiques. La connexion à distance à une machine ajoute des étapes au processus, mais n'est pas bien compliquée une fois initié. Encore faut-il avoir accès à une bonne machine. Le Département de mathématiques et statistique rend disponible gratuitement des machines pour ses membres. Bien sûr, nous devons partager ces machines. Il peut donc arriver que peu ou pas de machines soient libres. Cependant, selon mon expérience, une telle situation n'est pas très fréquente.

Moyennant de faibles coûts, Amazon EC2 offre aussi l'accès à de bonnes machines virtuelles. Payer pour ce service peut être avantageux monétairement s'il nous évite l'achat d'ordinateurs de calcul puissants mais dispendieux. Utiliser des instances Amazon EC2 requière quelques étapes techniques préalables, dont l'ouverture d'un compte AWS (pour *Amazon Web Service*) et des préparatifs pour sécuriser la connexion à distance aux instances. Il y a aussi de la terminologie à débroussailler au début, par exemple les utilisateurs IAM (pour *Identity and Access Management*), les groupes de sécurité, les AMI (pour *Amazon Machine Image*), les types d'instances (`t2.micro`, `c4.8xlarge`, et autres), etc. La compréhension de tous des concepts demande un peu de temps, mais n'est pas si ardu en fin de compte.

Afin de multiplier le nombre de coeurs utilisés, il faut exploiter une grappe de serveurs. Ce document a présenté comment lancer des calculs R en parallèle sur plusieurs machines de la grappe de serveurs du DMS en faisant rouler un seul script. La création et l'utilisation d'une grappe d'instances Amazon EC2 n'ont pas été approfondies ici, mais des pistes ont été fournies pour arriver à le faire.

Bien sûr, ce document n'a pas traité de toutes les ressources disponibles pour réaliser du calcul R en parallèle. Calcul Québec gère des supercalculateurs utilisables par tout chercheur admissible aux subventions provenant des conseils de recherche canadiens. Aussi, beaucoup d'autres plate-formes de cloud computing autres que Amazon EC2 existent, notamment Google Cloud Platform et Microsoft Azure.

Si ce document avait une suite, il serait intéressant que celle-ci traite de calcul R en parallèle sur GPU. Dans certains domaines, notamment en apprentissage profond (en anglais *deep learning*), le calcul sur GPU est la norme. Il permet l'exploitation d'un plus grand nombre de coeurs de calcul par machine, et en conséquence l'atteinte de temps de calcul significativement plus rapides, pour un même nombre de machines, que ceux obtenus avec le calcul en parallèle sur CPU. Calcul Québec possède des supercalculateurs pour le calcul sur GPU et Amazon EC2 offre aussi des types d'instances avec de bons GPU de calcul. Il ne reste plus qu'à souhaiter que le communauté R rende disponible plus de code et de documentation pour faciliter la réalisation de calcul R sur GPU.

Bibliographie

Baillargeon, S. (2017). ***STT-4230/STT-6230 R pour scientifique***. Université Laval. [Notes de cours].

a) Calculs de base en R : Fonctions de la famille des apply. URL : http://archimede.mat.ulaval.ca/dokuwiki/doku.php?id=r:calculs:calculsbase#fonctions-de-la-famille-des-apply

b) Optimisation de temps d'exécution en R. URL : http://archimede.mat.ulaval.ca/dokuwiki/doku.php?id=r:amelioration:optimisation

Eddelbuettel, D. (2017). ***High-Performance and Parallel Computing with R***. [CRAN Task View], version 2017-11-10. URL : https://CRAN.R-project.org/view=HighPerformanceComputing

Gouarin, L., Louvet, V. et Series, L. (2012, avril). ***Introduction au calcul parallèle***. Formation LIEM2I. Groupe Calcul CNRS. [Présentation]. URL : http://calcul.math.cnrs.fr/Documents/Ecoles/LEM2I/Mod3/paral.pdf

Hosmer, D.W., Lemeshow, S. et Sturdivant, R.X. (2013). ***Applied Logistic Regression***, Third Edition. John Wiley & Sons Inc.

Mahdi, E. (2014). **A Survey of R Software for Parallel Computing**. *American Journal of Applied Mathematics and Statistics*, **2**(4), p. 224–230. URL : http://pubs.sciepub.com/ajams/2/4/9

Matloff, N. (2011). ***Programming on parallel machines***. University of California, Davis.

Matloff, N. (2015, 23 janvier). ***Parallel Programming with GPUs and R***. [Billet de blogue]. URL : https://matloff.wordpress.com/2015/01/23/gpu-tutorial-with-r-interfacing/

McCallum, Q. E. et Weston, S. (2011). ***Parallel R : Data Analysis in the Distributed World***. O'Reilly.

Microsoft Corporation et Steve Weston (2017). ***`doParallel` : Foreach Parallel Adaptor for the `parallel` Package***. [R package], version 1.0.11. URL : https://CRAN.R-project.org/package=doParallel

- [Vignette]. URL : https://cran.r-project.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf

Mount, J. (2016, 22 janvier). ***Example using R on Amazon ec2***. [Dépôt Github]. URL : https://github.com/JohnMount/ec2R

Nicholson, W. (2014, 19 août). ***Creating a virtual cluster in R using Amazon ec2***. [Billet de blogue]. URL : https://wbnicholson.wordpress.com/2014/08/19/creating-a-virtual-cluster-in-r-using-amazon-ec2/

Nicholson, W. (2015). ***Creating an on-demand cluster with Amazon's elastic compute cloud***. Cornell University. [Tutoriel]. URL : http://stat.cornell.edu/sites/default/files/EC2Instructions.pdf

Peng, R. D., et de Leeuw, J. (2002). ***An Introduction to the .C Interface to R.*** UCLA : Academic Technology Services, Statistical Consulting Group. URL : http://www.biostat.jhsph.edu/~rpeng/docs/interface.pdf

R Core Team (2017). ***R : A language and environment for statistical computing***. R Foundation for Statistical Computing, Vienna, Austria. [Logiciel]. URL : https://www.R-project.org/

- [Vignette] du package `parallel` : https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf

RPubs : ***All subset regression with `leaps`, `bestglm`, `glmulti`, and `meifly`***. [Page web]. URL : https://rstudio-pubs-static.s3.amazonaws.com/2897_9220b21cfc0c43a396ff9abf122bb351.html

St-Onge, P.-L. (2017, 28 septembre). ***Introduction to Advanced Research Computing (ARC)***. Calcul Québec. [Présentation]. URL : http://www.hpc.mcgill.ca/downloads/intro_arc/20170928%20-%20McGill%20-%20Introduction%20to%20ARC.pdf

Documentation d'Amazon AWS contenant des instructions suivies dans le présent tutoriel :

Configuration avec Amazon EC2 :
http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

Comment créer et activer un nouveau compte Amazon Web Services ? :
https://aws.amazon.com/fr/premiumsupport/knowledge-center/create-and-activate-aws-account/

Lancement d'une machine virtuelle Linux :
https://aws.amazon.com/fr/getting-started/tutorials/launch-a-virtual-machine/

Démarrage sur les instances Linux Amazon EC2 :
http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/EC2_GetStarted.html

Connexion à votre instance Linux à l'aide de SSH :
http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html

Connexion à votre instance Linux à partir de Windows à l'aide de PuTTY :
http://docs.aws.amazon.com/fr_fr/AWSEC2/latest/UserGuide/putty.html

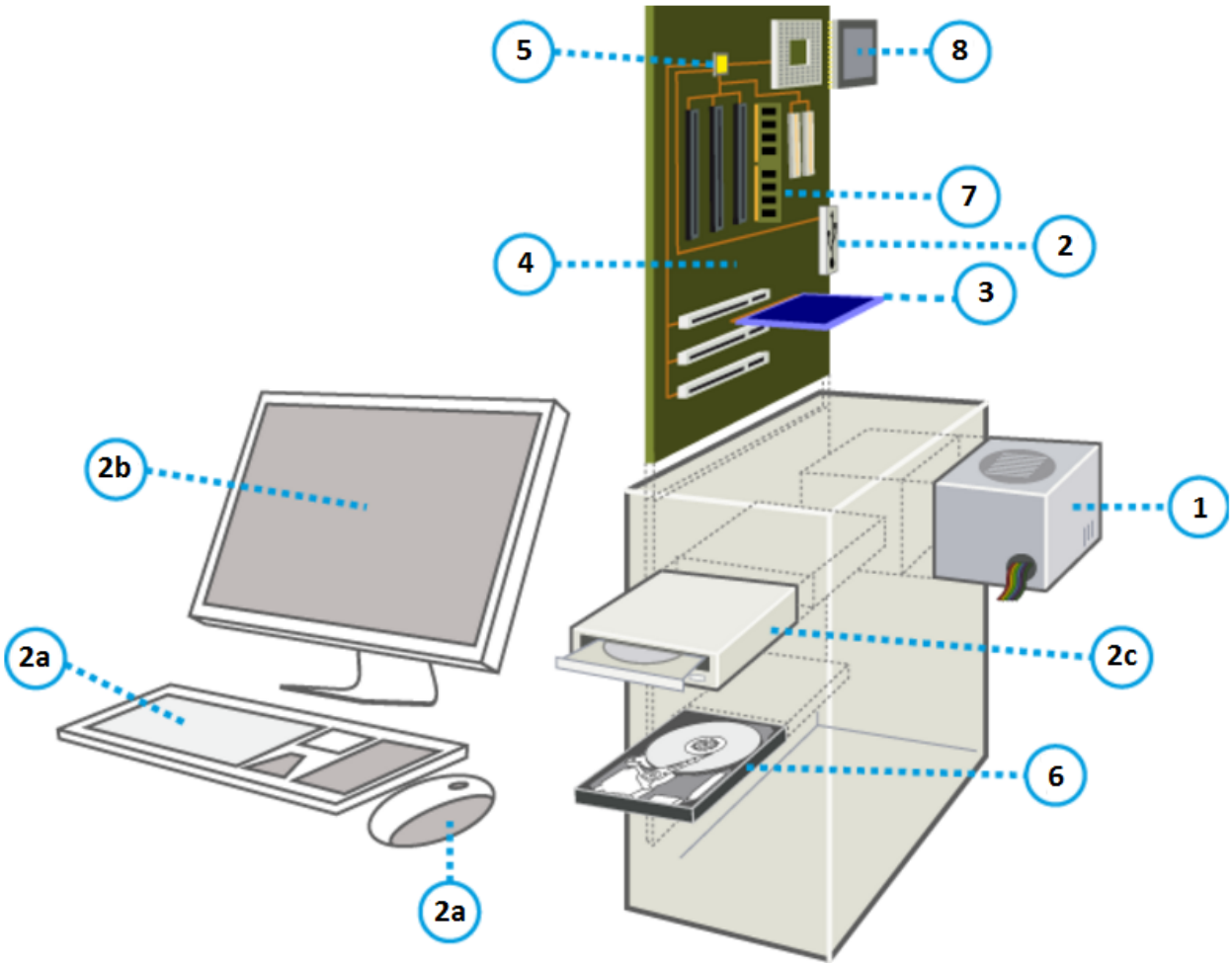

FIGURE 14 – Composantes matérielles usuelles d'un ordinateur personnel (de bureau comme ici ou même portable). Source : http://www.imedias.pro/cours-en-ligne/informatique/ordinateur/composants-ordinateur/

Annexe A : Composantes matérielles d'un ordinateur personnel

1. des composantes en charge de l'apport électrique et du refroidissement (bloc d'alimentation électrique, batterie, ventilateur, dissipateur de chaleur) ;
2. des ports pour périphériques tels que :
   - a) entrée : clavier, souris, etc. ;
   - b) sortie : écran, imprimante, etc ;
   - c) entrée-sortie : du stockage amovible, par exemple un lecteur CD/DVD, une clé USB, etc. ;
3. des cartes d'extension :
   - carte graphique ou vidéo (contenant un GPU pour *Graphical Processing Unit*),
   - carte de son (possiblement en périphérique),
   - carte réseau (possiblement en périphérique) ;
4. une carte mère pour accueillir et relier les composantes ;
5. des bus pour la communication entre les composantes ;
6. de l'espace de stockage permanent : disque dur, SSD (*Solid State drive*) ;
7. de l'espace de stockage temporaire : mémoire vive (RAM pour *Random Access Memory*) ;
8. un processeur (CPU pour *Central Processing Unit*).

Annexe B : Script R pour les expérimentations sur la grappe de serveurs du DMS

Voici le script R utilisé pour lancer les expérimentations sur la grappe de serveurs du DMS. Au moment où ces expérimentations ont été effectuées, les serveurs `dms4`, `dms8` et `dms12` étaient occupés. Ainsi, seulement les 9 autres serveurs de la grappe, qui étaient totalement libres, ont été exploités. Dans ce code, l'objet `specs` est une liste de toutes les grappes de calcul à tester. Le calcul est lancé 100 fois sur chacune des grappes de `specs`.

```
# À soumettre préalablement
# install.packages("microbenchmark", lib = "~/Rlib_x86_64")

library(parallel)
library(microbenchmark)

# --- Préparation ---

# Estimation de densité par noyau gaussien avec calcul séquentiel
ksmooth_apply <- function(x, xpts, h)
{
  n <- length(x)
  sapply(xpts, function(xpts_i) { sum(dnorm((xpts_i - x)/h)) / (n * h) })
}

# Estimation de densité par noyau gaussien avec calcul en parallèle
ksmooth_par <- function(grappe, x, xpts, h)
{
  n <- length(x)
  parSapply(grappe, xpts, function(xpts_i) { sum(dnorm((xpts_i - x)/h)) / (n * h) })
}

# Simulation d'observations
set.seed(827)
x <- rnorm(1000000)
# Détermination des points en lesquels effectuer une estimation
xpts <- seq(from = -4, to = 4, length.out = 161)

# --- Calculs ---

# Calculs séquentiels

out <- microbenchmark(dens_apply = ksmooth_apply(x = x, xpts = xpts, h = 1))
cat("Calculs séquentiels", median(out$time)/1000000000, "\n")

# Calculs parallèles

# Spécifications pour les différentes grappes de calculs testées
specs <- list(

  # Nombre de processus croissants, sur un seul noeud
  "dms11x1" = rep("dms11", times = 1),
  "dms11x2" = rep("dms11", times = 2),
  "dms11x3" = rep("dms11", times = 3),
  "dms11x4" = rep("dms11", times = 4),
```

```
  "dms11x5" = rep("dms11", times = 5),
  "dms11x6" = rep("dms11", times = 6),
  "dms11x7" = rep("dms11", times = 7),
  "dms11x8" = rep("dms11", times = 8),
  "dms11x9" = rep("dms11", times = 9),
  "dms11x10" = rep("dms11", times = 10),
  "dms11x11" = rep("dms11", times = 11),
  "dms11x12" = rep("dms11", times = 12),
  "dms11x13" = rep("dms11", times = 13),
  "dms11x14" = rep("dms11", times = 14),
  "dms11x15" = rep("dms11", times = 15),
  "dms11x16" = rep("dms11", times = 16),

  # Nombre de processus constant (8), mais nombre de noeuds croissant
  "dms1x8" = rep("dms1", times = 8),
  "dms1x4+dms2x4" = rep(paste0("dms", 1:2), each = 4),
  "dms1x3+dms2x3+dms3x2" = c(rep(paste0("dms", 1:2), each = 3), rep("dms3", times = 2)),
  "dms1x2-dms3x2+dms5x2" = rep(paste0("dms", c(1:3, 5)), each = 2),
  "dms1x2-dms3x2+dms5x1+dms6x1" = c(rep(paste0("dms", 1:3), each = 2), paste0("dms", 5:6)),
  "dms1x2+dms2x2+dms3x1+dms5x1-dms7x1" =
    c(rep(paste0("dms", 1:2), each = 2), paste0("dms", c(3,5:7))),
  "dms1x2+dms2x1+dms3x1+dms5x1-dms7x1+dms9x1" =
    c(rep("dms1", times = 2), paste0("dms", c(2:3, 5:7, 9))),
  "dms1x1-dms3x1+dms5x1-dms7x1+dms9x1+dms10x1" = paste0("dms", c(1:3, 5:7, 9, 10)),

  # Nombre de processus et nombre de noeuds croissants
  "dms1x8" = rep("dms1", times = 8),
  "dms1x8+dms2x8" = rep(paste0("dms", 1:2), each = 8),
  "dms1x8-dms3x8" = rep(paste0("dms", 1:3), each = 8),
  "dms1x8-dms3x8+dms5x8" = rep(paste0("dms", c(1:3, 5)), each = 8),
  "dms1x8-dms3x8+dms5x8+dms6x8" = rep(paste0("dms", c(1:3, 5, 6)), each = 8),
  "dms1x8-dms3x8+dms5x8-dms7x8" = rep(paste0("dms", c(1:3, 5:7)), each = 8),
  "dms1x8-dms3x8+dms5x8-dms7x8+dms9x8" = rep(paste0("dms", c(1:3, 5:7, 9)), each = 8),
  "dms1x8-dms3x8+dms5x8-dms7x8+dms9x8+dms10x8" =
    rep(paste0("dms", c(1:3, 5:7, 9, 10)), each = 8)
)

# Boucle sur toutes les grappes définies dans specs
for (i in 1:length(specs)){
  # Initialisation de la grappe de sessions R
  spec <- specs[[i]]
  grappe <- makeCluster(type='PSOCK', spec=spec)
  # Lancement des calculs en parallèle
  out <- microbenchmark(dens_par = ksmooth_par(grappe = grappe, x = x, xpts = xpts, h = 1))
  # Impression du résultat
  cat(names(specs)[i], median(out$time)/1000000000, "\n")
  # Fermeture de la grappe
  stopCluster(grappe)
}
```