# 1 Objectif

**Programmation parallèle sous R. Utilisation des librairies « parallel » et « doParallel ».**

Les ordinateurs personnels sont de plus en plus performants. Ils sont maintenant pour la plupart dotés de processeurs multi-cœurs. Dans le même temps, les logiciels grands publics de Data Mining, libres ou non, restent souvent monothread. Un seul cœur est sollicité durant les calculs, tandis que les autres restent inactifs. Ca fait un peu gaspillage.

Récemment, nous avons introduit deux variantes multithread de l'analyse discriminante linéaire dans Sipina 3.10[1] et 3.11[2]. Après une analyse succincte de l'algorithme, nous nous sommes concentrés sur la construction de la matrice de variance covariance intra-classes qui semblait constituer le principal goulot d'étranglement du processus. Deux types de décomposition des calculs ont été proposés, que nous avons exploité en nous appuyant sur la programmation multithread sous Delphi (langage de développement de Sipina). Nous avons constaté que l'amélioration des performances par rapport à la version séquentielle est spectaculaire lorsque l'ensemble des cœurs disponibles sur la machine sont pleinement utilisés.

Durant le travail d'analyse qui m'a permis de mettre au point les solutions introduites dans Sipina, j'avais beaucoup étudié les mécanismes de parallélisation disponibles dans d'autres outils de Data Mining. Ils sont plutôt rares en réalité. J'avais quand même noté que des stratégies très perfectionnées sont proposées pour le logiciel R. Il s'agit souvent d'environnements qui permettent d'élaborer des programmes tirant parti des capacités étendues des machines à processeurs multi-cœurs, multiprocesseurs, et même d'un cluster de machines. Je me suis intéressé en particulier au package « **parallel** »[3] qui est lui-même une émanation des packages « snow » et « multicore ». Entendons-nous bien, la librairie ne permet pas d'accélérer miraculeusement une procédure existante. En revanche, elle nous donne l'opportunité d'exploiter efficacement les ressources machines à condition de reprogrammer la procédure en réorganisant judicieusement les calculs[4]. L'idée maîtresse est de pouvoir décomposer le processus en tâches que l'on peut exécuter en parallèle, charge à nous par la suite d'effectuer la consolidation.

Dans ce tutoriel, nous détaillons la parallélisation d'un algorithme de calcul d'une matrice de variance covariance intra-classes sous **R 3.0.0**. Dans un premier temps, nous décrivons un programme séquentiel, mais facilement transformable. Dans un deuxième temps, nous utilisons les outils des packages « parallel » et « doParallel » pour lancer les tâches élémentaires sur les cœurs disponibles. Nous comparerons alors les temps de traitement. Disons-le d'emblée, contrairement aux exemples jouets que l'on peut consulter ici ou là sur le web, le bilan est passablement mitigé quand il s'agit de traiter des bases volumineuses réalistes. Les solutions achoppent sur la gestion des données.

---

[1] http://tutoriels-data-mining.blogspot.fr/2013/05/multithreading-pour-lanalyse.html

[2] http://tutoriels-data-mining.blogspot.fr/2013/06/multithreading-equilibre-pour-la.html

[3] http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf

[4] La démarche n'est pas triviale, des solutions alternatives sont explorées. Ex. http://www.flowlang.net/2012/05/multicore-dilemma-in-big-data-era-is.html

# 2 Calcul de matrice de variance covariance intra-classes

Les observations sont réparties dans K groupes définis par une variable Y prenant ses valeurs dans {1, 2, …, K}. Elles sont décrites par *p* variables X = ($X_1$, $X_2$, …, $X_p$). Nous disposons d'un échantillon de taille *n.* Un individu est noté ω, sa valeur pour la variable Y s'écrit y(ω). Le nombre d'observations appartenant à la classe Y = k est égal à $n_k$. Pour obtenir la matrice de variance covariance intra-classes **S**, nous devons tout d'abord calculer les matrices conditionnelles $\mathbf{W}_k$ relatives à chaque groupe.

$$W_k = \left( \sum_{\omega:y(\omega)=k} [x_i(\omega) - \bar{x}_{i,k}] \times [x_j(\omega) - \bar{x}_{j,k}] \right)_{i,j=1,\cdots,p}$$

Où $\bar{x}_{i,k}$ représente la moyenne de la variable $X_i$ pour les individus de la classe (Y = k).

La matrice S est obtenue avec

$$S = \frac{1}{n-K} \sum_{k=1}^{K} W_k$$

## 2.1 Décomposition des calculs

Il y a plusieurs stratégies possibles. Nous avons choisi d'organiser les calculs en trois étapes. L'objectif est de pouvoir décomposer la construction des $W_k$ – la plus gourmande en temps de traitement – en tâches indépendantes pouvant être lancées en parallèle.

**[1]** La première étape consiste à produire un vecteur de numéros associant les observations aux groupes. La fonction **partition()** identifie les individus appartenant au groupe « group » à partir des valeurs de Y.

```
#group : le numéro du groupe à traiter
#y : la variable cible délimitant les groupes
#sortie : un vecteur de numéros indiquant les individus appartenant à group
partition <- function(group, y){
  res <- which(y==group)
  return(res)
}
```

Nous l'appliquons à l'ensemble des groupes à l'aide de la fonction **lapply()** de R.

```
#pour chaque valeur (niveau) de Y, nous appliquons la fonction partition
#sortie : liste de « vecteurs de numéros »
ind.per.group <- lapply(levels(Y),partition,y=Y)
```

**ind.per.group** est une liste où chaque item est un vecteur d'index associant les observations aux groupes. Il y a donc K vecteurs dans la liste.

**[2]** La seconde étape consiste à calculer $W_k$ à partir des individus associés au groupe k et des descripteurs X[5]

[5] Bien sûr, la fonction COV() de R aurait très bien fait l'affaire. Il s'agit pour nous de décrire de manière très scolaire les calculs afin que tout un chacun puisse les reconstituer.

```
#instances : vecteur des numéros des individus appartenant au groupe k
#descriptors : matrice des variables X : (X1, X2, …, Xp)
#sortie : la matrice de variance covatriance Wk de dimension (p x p)
n.my.cov <- function(instances,descriptors){
  p <- ncol(descriptors)
  m <- colMeans(descriptors[instances,])
  the.cov <- matrix(0,p,p)
  for (i in 1:p){
    for (j in 1:p){
      the.cov[i,j] <- (sum((descriptors[instances,i]-m[i])*(descriptors[instances,j]-m[j])))
    }
  }
  return(the.cov)
}
```

Nous l'appliquons à l'ensemble des groupes avec la fonction **lapply()**

```
#pour chaque liste d'individus
#appliquer la fonction n.my.cov()
#sortie : liste des matrices Wk (K matrices en tout)
list.cov.mat <- lapply(ind.per.group,n.my.cov,descriptors=X)
```

Le nœud de l'affaire est ici : lapply() applique n.my.cov() à chaque vecteur de numéros de manière indépendante, nous pouvons très bien lancer ces tâches en parallèle et procéder à la consolidation à l'issue de la construction des matrices $W_k$. Nous exploiterons cette particularité dans les implémentations parallèles que nous présenterons dans la suite de ce document.

**[3]** Dernière étape enfin, nous construisons la matrice S en sommant les valeurs contenues dans les matrices $W_k$ et en appliquant la correction par les degrés de liberté.

```
#additionner les valeurs dans les matrices contenues dans la liste
#appliquer la correction par les degrés de liberté
#sortie : la matrice S
pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
```

## 2.2 Fonction de calcul de la matrice S

Ces différentes étapes ont été regroupées dans la fonction pooled.sequential() :

```
#X matrice des descripteurs
#Y variable indiquant l'appartenance aux groupes
#sortie : matrice S variance covariance intra-groupes
pooled.sequential <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  list.cov.mat <- lapply(ind.per.group,n.my.cov,descriptors=X)
  pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
  return(pooled.cov.mat)
}
```

Dans la suite de ce document, notre philosophie consistera à mettre en place différentes approches parallèles pour calculer list.cov.mat. C'est la démarche que nous avions suivie dans notre

implémentation multithread de l'analyse discriminante de Sipina 3.10. Avec les inconvénients que nous avions identifiés, à savoir : K cœurs sont sollicités, indépendamment des caractéristiques de la machine qui peut être sous-exploitée ; les charges sont mal réparties lorsque les classes sont déséquilibrées c.-à-d. lorsque les vecteurs d'index sont de taille très dissemblables[6].

## 2.3 Exemple de calcul sur un fichier de données

Nous appliquons les calculs sur la base WAVE500K décrite dans notre précédent tutoriel. Il comporte n = 500.000 observations, réparties en K = 3 groupes équilibrés, décrits par p = 21 variables. Nous utilisons le programme R suivant.

```
#chargement du fichier
wave <- read.table(file="wave500k.txt",header=T,sep="\t",dec=".")
print(summary(wave))

#définition de la cible Y et des descripteurs X
Y <- wave$ONDE
X <- wave[2:22]

partition <- function(group,y){
  res <- which(y==group)
  return(res)
}

n.my.cov <- function(instances,descriptors){
  p <- ncol(descriptors)
  m <- colMeans(descriptors[instances,])
  the.cov <- matrix(0,p,p)
  for (i in 1:p){
    for (j in 1:p){
     the.cov[i,j] <- (sum((descriptors[instances,i]-m[i])*(descriptors[instances,j]-m[j])))
    }
  }
  return(the.cov)
}

pooled.sequential <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  list.cov.mat <- lapply(ind.per.group,n.my.cov,descriptors=X)
  pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
  return(pooled.cov.mat)
}

#appel de la fonction de calcul de S et affichage
system.time(cov.seq <- pooled.sequential(X,Y))
print(cov.seq)
```

**system.time()** est utilisé pour mesurer la durée des traitements.

[6] http://tutoriels-data-mining.blogspot.fr/2013/05/multithreading-pour-lanalyse.html

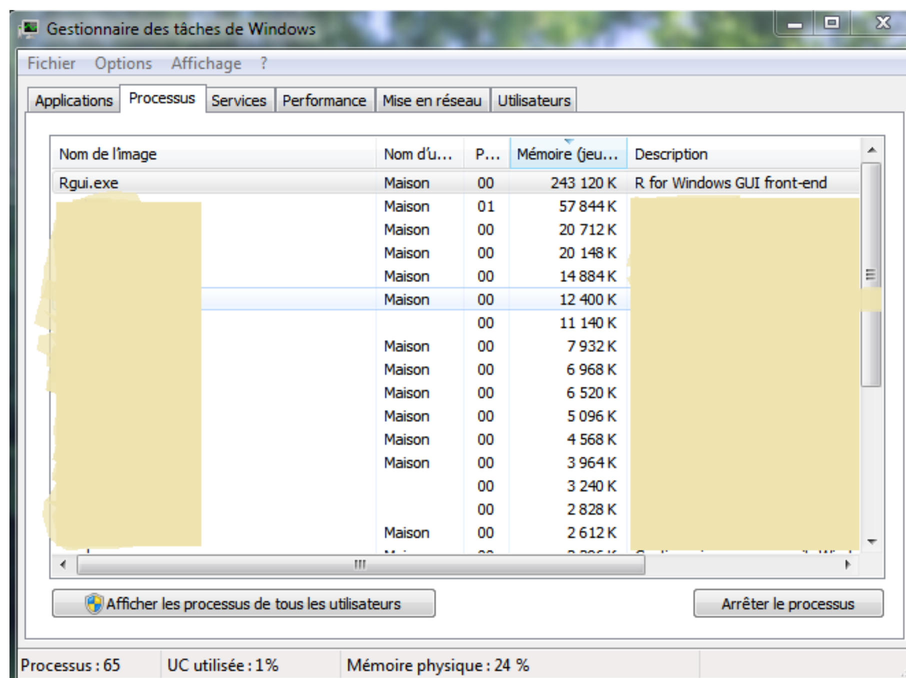
R 3.0.0 sous Windows 7 (64 bits) tournant sur un Quad-Core (Q9400 – 4 cœurs)…

```
R version 3.0.0 (2013-04-03) -- "Masked Marvel"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
```

…produit les informations suivantes.

```
> system.time(cov.seq <- pooled.sequential(X,Y))
utilisateur     système      écoulé
      10.71        4.26       15.08
```

L'occupation mémoire visible dans le gestionnaire de tâches est ≈ 237 Mo.




Nous nous intéressons principalement au « temps écoulé » (**16.80 sec.**) qui correspond à notre durée d'attente de l'achèvement des calculs. C'est ce que nous avions mesuré avec un chronomètre par ailleurs. Pour un calcul séquentiel, il est approximativement égal à l'addition du « temps utilisateur » (la durée de l'exécution directe des instructions) et du « temps système » (le temps consacré par le système pour la bonne exécution de nos instructions). Voyons s'il est possible de diminuer ce temps écoulé en répartissant les tâches sur les différents cœurs du système.

# 3 Le package « parallel »

## 3.1 Mécanisme de parallélisation des calculs

Le package « parallel »[7] est élaboré à partir des librairies « snow » et « multicore », ce dernier n'étant plus disponible pour R 3.0.0. L'article de présentation (datée du 8 avril 2013) résume parfaitement les idées que nous essayons d'appliquer dans ce tutoriel. On peut la traduire comme suit.

[7] http://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf. Je conseille vraiment la lecture de ce document, il décrit admirablement bien la problématique de la parallélisation des calculs.

*Ce package permet d'exécuter des tranches de calculs en parallèle. Un exemple typique serait d'évaluer la même fonction sur des jeux de données différents… Le point important est que ces tranches doivent être indépendantes et n'ont pas besoin de communiquer entres elles. Souvent, ces tranches ont des durées d'exécution approximativement identiques. Le modèle de fonctionnement standard est le suivant :*

*(a) Démarrer M « worker » (traduit librement par « moteurs de calcul »), effectuer les initialisations pour chacun d'entre eux.*
*(b) Envoyer les données nécessaires à chaque moteur.*
*(c) Découper le processus global en M tâches de volume similaire, et les envoyer sur les moteurs.*
*(d) Attendre que tous les moteurs aient achevé les tâches et récupérer les résultats.*
*(e) Répéter les étapes (b) à (d) pour les éventuelles tâches restantes (si lé découpage a été réalisé en Q tâches avec Q > M)*
*(f) Stopper les moteurs de calcul.*

Nous souhaitons exploiter ce schéma pour la construction de la matrice S, avec M = K = nombre de groupes décrits par la variable cible Y. Chaque individu n'appartenant qu'à un seul groupe, les calculs des $W_k$ peuvent être réalisés de manière indépendante.

Concrètement, chaque « moteur » correspond à un processus « rscript » visible dans le gestionnaire de tâche sous Windows. Dans notre cas, puisque M = K = 3, nous verrons 3 exemplaires de rscript, avec la même occupation mémoire parce que la totalité des données leur est transférée. C'est la principale faiblesse de la solution. En effet, en dupliquant les données, on augmente d'autant l'occupation mémoire. Et leur transfert vers les processus lors de l'initialisation des rscript est aussi consommatrice de temps. De fait, cette étape va fortement grever les performances du système comme nous le verrons par la suite.

## 3.2 Détection du nombre de cœurs

Le package « parallel » propose la procédure **detectCores()** pour déceler le nombre de cœurs disponibles sur la machine. Pour notre machine Quad core, elle en affiche 4 évidemment.

```
#charger la librairie
library(parallel)
#afficher le nombre de coeurs dispos
print(detectCores())
```

```
> print(detectCores())
[1] 4
```

## 3.3 La procédure mclapply()

La procédure **mclapply()**, issue de la librairie « multicore », est une version parallélisée de **lapply(),** nous pouvons spécifier le nombre de cœurs à utiliser sur la machine. De fait, une seule ligne est à modifier dans notre fonction de calcul de la matrice S :

```
pooled.multicore <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  list.cov.mat <- mclapply(ind.per.group,n.my.cov,descriptors=X,mc.cores=1)
  pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
  return(pooled.cov.mat)
}
```

L'option « mc.cores » permet de définir le nombre de cœurs à utiliser. Nous fixons mc.cores = 1 dans un premier temps. Lorsque nous lançons la procédure, la durée d'exécution est très proche de la méthode séquentielle. Ce qui est tout à fait logique puisqu'un seul cœur est utilisé. Les calculs des $W_k$ sont lancés les uns après les autres.

```
> system.time(cov.mc <- pooled.multicore(X,Y))
utilisateur     système      écoulé
      10.58        3.93       14.58
```

Hélas, lorsque nous avons voulu augmenter le nombre de cœurs à exploiter (mc.cores = 3), la procédure a échoué.

```
> system.time(cov.mc <- pooled.multicore(X,Y))
Erreur dans mclapply(ind.per.group, n.my.cov, descriptors = X, mc.cores = 3) :
  'mc.cores' > 1 is not supported on Windows
Timing stopped at: 0.17 0 0.18
```

La procédure s'appuie sur une technologie qui n'est pas opérationnelle sous Windows[8]. Dommage parce que la méthode était très facile à mettre en œuvre. Une seule ligne de code était à modifier, avec des transformations très mineures. Il semble en revanche que le dispositif soit pleinement opérationnel sous Linux.

## 3.4 La procédure parLapply()

La procédure **parLapply()** est aussi une variante de **lapply()**. Il nécessite deux étapes supplémentaires : nous devons initialiser explicitement les moteurs à utiliser au préalable avec **makeCluster()**[9], nous devons les détruire à la fin des calculs avec **stopCluster()**. De fait, la fonction de calcul de S devient maintenant :

```
pooled.parLapply <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  #création de K = 3 rscript que l'on voit apparaître
  #dans le gestionnaire de tâches de windows
  cl <- makeCluster(spec=nlevels(Y))
  #appel de parLapply
  list.cov.mat <- parLapply(cl,ind.per.group,n.my.cov,descriptors=X)
  pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
  #destruction des rscript
  stopCluster(cl)
  return(pooled.cov.mat)
}
```

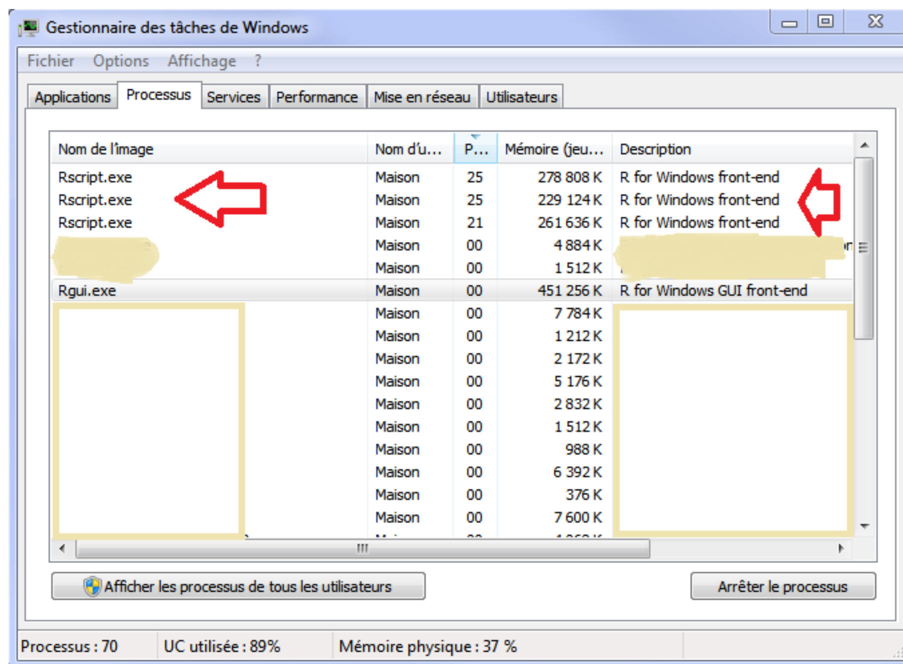
Nous obtenons l'affichage suivant :

```
> system.time(cov.par <- pooled.parLapply(X,Y))
utilisateur     système      écoulé
       1.75        1.17       15.12
```

[8] http://stat.ethz.ch/R-manual/R-devel/library/parallel/html/mclapply.html

[9] Nous créons les moteurs sur le même système ici pour exploiter les cœurs existants. Mais makeCluster() peut aller plus loin en accédant aux ressources d'une machine distante pourvu que R y soit installée.

Tout ça pour ça serait-on tenté de dire. Le gain par rapport à la version séquentielle est nul avec un temps écoulé de 15.12 sec. Le temps utilisateur (1.75 sec.) et le temps système (1.17 sec.) correspondent à l'activité du thread principal, non affecté aux calculs des $W_k$.




En surveillant le gestionnaire de tâches de Windows, nous constatons que les calculs sont bien répartis sur 3 processus rscript différents. Nous constatons également que les données sont complètement dupliquées sur chaque processus au regard de l'occupation mémoire. Pourquoi n'y a-t-il pas d'amélioration globale du temps de traitement ?

Nous avons la réponse en produisant une meilleure décomposition des durées d'exécution. Nous utilisons la procédure **snow.time()**[10] de la librairie « snow ».

```
library(snow)
#meilleure décomposition du temps de traitement
snow.time(cov.par <- pooled.parLapply(X,Y))
```

Les idées sont plus claires maintenant.

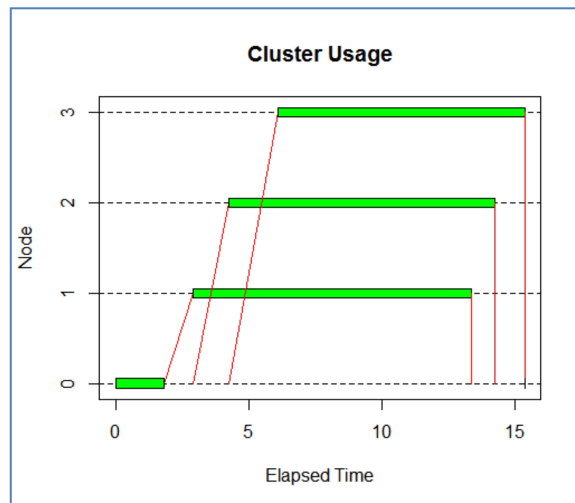
```
> snow.time(cov.par <- pooled.parLapply(X,Y))
elapsed     send receive   node 1   node 2   node 3
  14.92     4.33    0.02     9.53     9.62     8.78
```

La durée de traitement sur chaque moteur est de 9 secondes approximativement. Il y a effectivement un gain à dispatcher les tâches sur plusieurs rscript. Mais il est totalement annihilé par le transfert des données (send = 4.33 sec.), la récupération des matrices étant peu coûteuse (receive = 0.02 sec.). Le package propose une présentation des temps de traitement sous forme de diagramme de Gantt.

```
> a <- snow.time(cov.par <- pooled.parLapply(X,Y))
> plot(a)
```

[10] http://cran.r-project.org/web/packages/snow/snow.pdf

La durée d'exécution globale est tributaire du temps de transfert des données (Node 0) et de l'achèvement des calculs sur le dernier moteur lancé (Node 3).



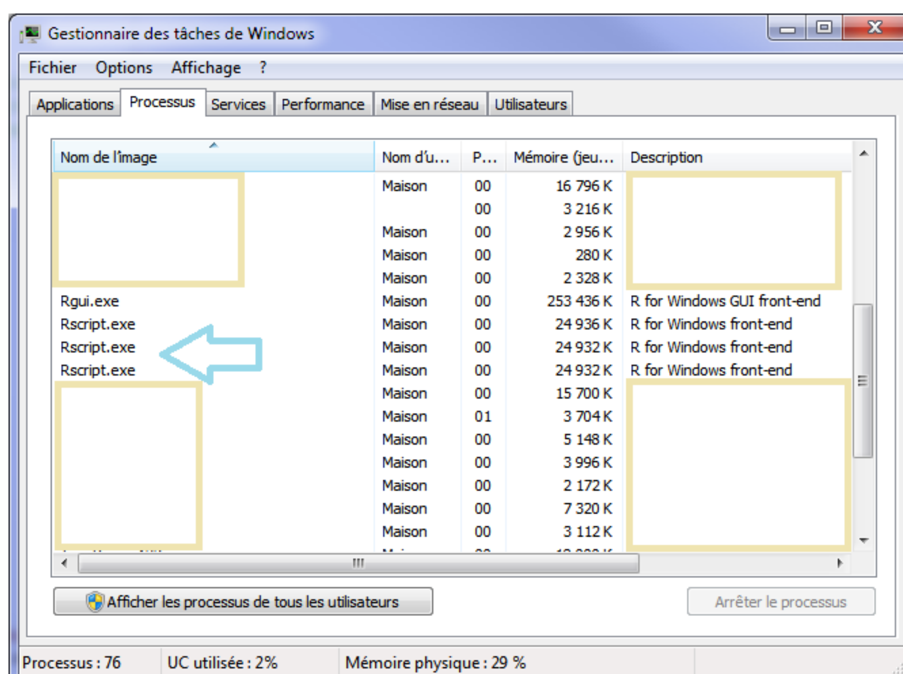

Manifestement, la parallélisation n'est avantageuse que si la durée de traitement des tâches est nettement plus élevée par rapport au temps de transfert des informations.

## 3.5 Utilisation de la boucle « foreach » … « %dopar% »

La librairie « doParallel », via le package « foreach », propose un mécanisme de boucle permettant de lancer des tâches en parallèle. La démarche est toujours la même (voir section 3.1). Nous devons tout d'abord démarrer les moteurs avec **registerDoParallel()**.

```
library(doParallel)
#création de 3 rscript en mémoire sous Windows
registerDoParallel(3)
```

Les 'rscript' apparaissent dans le gestionnaire de tâches. Ils sont en attente : l'utilisation du CPU est 0 (Processeur) et les données n'ont pas été transmises encore (Mémoire ≈ 24 Mo).

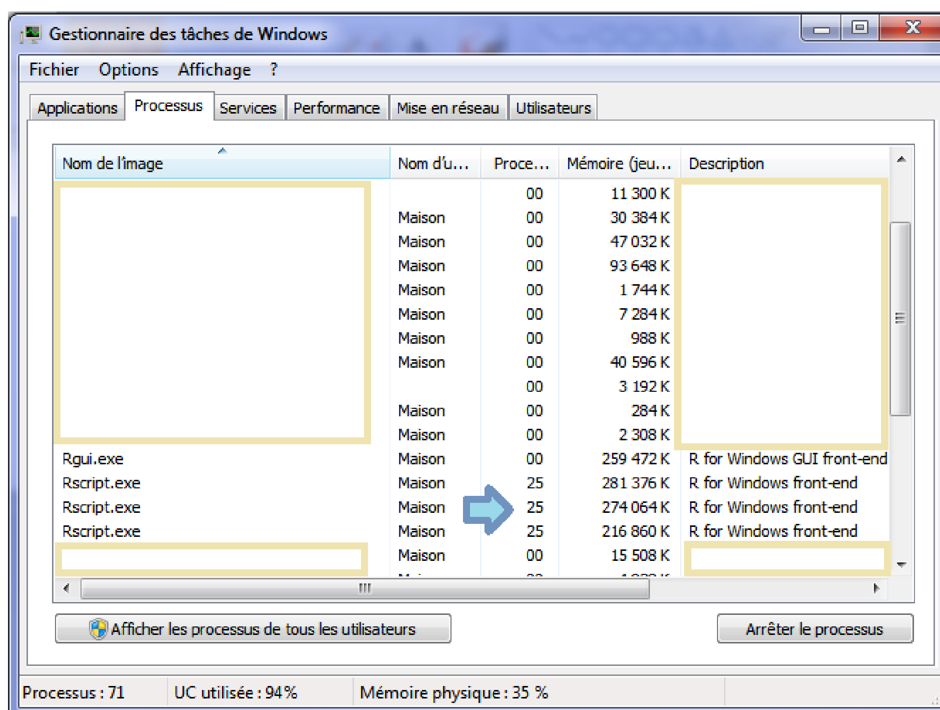


Les calculs sont lancés via une boucle « foreach…%dopar% » qui peut se lire : pour chaque élément « instances » de la liste « ind.per.group » passé en paramètre, exécuter en parallèle les instructions comprises entre ‘{‘ et ‘}’.

```
pooled.foreach <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  #boucle foreach ... %dopar%
  list.cov.mat <- foreach(instances = ind.per.group, .export = c("n.my.cov"), .inorder = FALSE) %dopar%
            {
               n.my.cov(instances,descriptors=X)
            }
  pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
  return(pooled.cov.mat)
}
```

L’option « .export » permet d’exporter explicitement la fonction non-standard « n.my.cov » dans l’environnement de la boucle ; « .inorder = FALSE » indique qu’il n’est pas nécessaire de respecter l’ordre d’envoi des calculs lors de la récupération des résultats.

A l’exécution, nous voyons les rscript s’activer dans le gestionnaire de tâches.




Dixit la colonne « Mémoire », nous constatons que les données X sont aussi dupliquées dans les rscript dans ce dispositif.

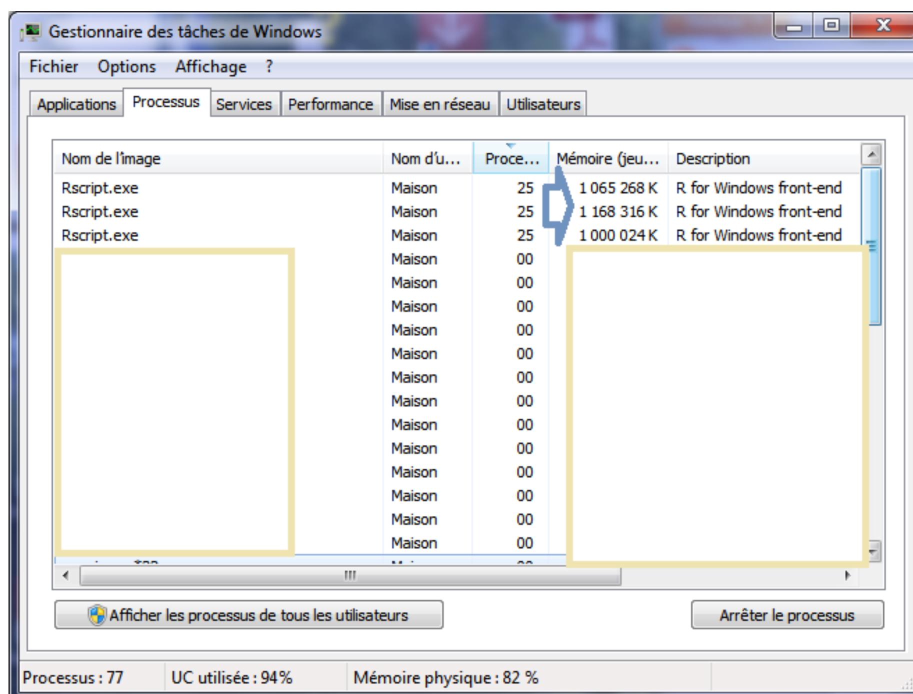
Avec « .verbose = T », nous avons le détail du déroulement des opérations : les blocs d’instructions (« task ») sont lancés en parallèle ; les résultats – en l’occurrence les matrices $W_k$ – sont ensuite combinés dans une liste. Tout comme pour parLapply(), la durée globale d’exécution est proche de la version séquentielle (14.18 sec.). Nous observons de nouveau la part importante dévolue au transfert des données (3.10 sec.).

```
> snow.time(cov.foreach <- pooled.foreach(X,Y))
numValues: 3, numResults: 0, stopped: TRUE
automatically exporting the following variables from the local environment:
  X
explicitly exporting variables(s): n.my.cov
got results for task 1
numValues: 3, numResults: 1, stopped: TRUE
returning status FALSE
got results for task 2
numValues: 3, numResults: 2, stopped: TRUE
returning status FALSE
got results for task 3
numValues: 3, numResults: 3, stopped: TRUE
calling combine function
evaluating call object to combine results:
  fun(accum, result.1, result.2, result.3)
returning status TRUE
elapsed    send receive  node 1  node 2  node 3
  14.18    3.10   -0.02   10.69   10.77   10.85
```

Il n'y a pas de gain véritable par rapport aux variantes parallèles de lapply. Je dirais surtout que la boucle « foreach…%dopar% » en est une alternative en matière de parallélisation des tâches.

# 4 Conclusion

Dans ce tutoriel, nous avons présenté différents solutions du package « parallel » sous R pour paralléliser des procédures. L'objectif est de rentabiliser les capacités accrues des processeurs multi-cœurs. Le dispositif fonctionne, c'est déjà un résultat très important. Cependant, les gains sont décevants sur le jeu de données que nous avons étudié, en grande partie à cause de la nécessité de dupliquer et de transférer les données aux moteurs de calcul. Les bases peuvent être volumineuses dans le contexte du Data Mining. Par exemple, pour WAVE2M avec n = 2.000.000 d'observations et p = 21 variables, la durée d'exécution est de 59 secondes en séquentiel, il passe à 56 en parallèle : avec 40 secondes de traitements sur chaque rscript et 15 secondes pour le transfert des données. Plus problématique encore, chaque rscript occupe près de 1Go en mémoire.



Cette démultiplication de l'occupation mémoire constitue certainement le principal frein à l'utilisation de ce dispositif dans le contexte de l'exploitation des machines à processeurs multi-cœurs pour le data mining.

Néanmoins, les outils du package « parallel » permettent d'aller plus loin et dépasser le cadre restreint que nous avions défini. Il est possible de dispatcher des calculs sur des machines distantes reliées en réseau. Cela ouvre d'autres perspectives. L'occupation mémoire devient moins contraignante. Il y a là matière à faire des choses intéressantes dans un de nos prochains tutoriels…