1 Topic

Parallel programming in R. Using the « parallel » and « doParallel » packages.

Personal computers become more and more efficient. They are mostly equipped with multi-core processors. At the same time, most of the data mining tools, free or not, are often based on single-threaded calculations. Only one core is used during calculations, while others remain inactive.

Previously, we have introduced two multithreaded variants of linear discriminant analysis in Sipina 3.10[1] and 3.11[2]. We focused on the construction of the within-class covariance matrix which appeared to be the main bottleneck in the process. Two types of subdivision of calculations have been proposed. We use threads programming in Delphi (the development language of Sipina) to make use of these solutions. We found that the improvement of performance compared with the sequential version is dramatic when all of the available cores on the computer are fully used.

During the analysis that allowed me to develop the solutions introduced in Sipina, I had much studied parallelization mechanisms available in other Data Mining Tools. They are rather scarce. I noted that highly sophisticated strategies are proposed for the R software. These are often environments that enable to develop programs for multi-core processors machines, multiprocessor machines, and even for computer cluster. I studied in particular the "parallel" package which is itself derived from 'snow' and 'multicore' packages. Let us be quite clear. The library cannot miraculously accelerate an existing procedure. It gives us the opportunity to effectively use the machines resources by rearranging properly the calculations. Basically, the idea is to break down the process into tasks that can be run in parallel. When these tasks are completed, we perform the consolidation.

In this tutorial, we detail the parallelization of the calculation of the within-class covariance matrix under R 3.0.0. In a first step, we describe single-threaded approach, but easily convertible i.e. the basic tasks are easily identifiable. As a second step, we use the tools of “parallel” and “doParallel” packages to run elementary tasks on the available cores. We will then compare the processing time. We note that, unlike the toy examples available on the web, the results are mixed. The bottleneck is the managing of the data when we handle a large dataset.

2 Calculation of the within-class covariance matrix

We dispose of a learning sample of size *n*. They instances are described by a set of *p* quantitative measurements (X_1, X_2, ..., X_p), they are assigned to predefined groups described by a categorical variable Y. There are K groups {1, 2, …, K}. Let ω an instance, $y(\omega)$ correspond to the class value of this instance. The absolute frequency of the class *k* is n_k. To obtain the within-class covariance matrix, we must compute the K conditional covariance matrices $\mathbf{W_k}$ (ignoring a multiplication factor).

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

[1] http://data-mining-tutorials.blogspot.fr/2013/05/multithreading-for-linear-discriminant.html

[2] http://data-mining-tutorials.blogspot.fr/2013/09/load-balanced-multithreading-for-lda.html

Where $\bar{x}_{i,k}$ corresponds to the mean of X_i for the individuals belonging to the group (Y = k).

The within-class covariance matrix (or pooled covariance matrix) S is computed as follows

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

2.1 Details of the calculations

We have chosen to organize calculations in three steps. The goal is to split up the construction of the $\mathbf{W_k}$ into independent tasks that we can be run in parallel.

[1] First, we create an index which enables to assign each instance to its group membership.

```
#group: the group number to handle
#y: the class-attribute
#output: a number vector which associates the instances to the groups
partition <- function(group, y){
  res <- which(y==group)
  return(res)
}
```

We apply this function to each group with the lapply() R function.

```
#to each level of Y, we launch partition
#output : a list of indexes
ind.per.group <- lapply(levels(Y),partition,y=Y)
```

ind.per.group is a list structure where each item correspond to an index for a group. There are K vectors into the list.

[2] We calculate W_k in the second step, from the index associated to the group and the descriptors X[3]

```
#instances: the index vector associating the instances to the group k
#descriptors: descriptors X : (X1, X2, …, Xp)
#output: conditional covariance matrix Wk (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)
}
```

We apply this function to all the groups with the **lapply()** function.

[3] Of course, we can use the **cov()** R function for these calculations. The aim here is to describe deeply the calculations in the learning process.

```
#for each index vector
#we apply the function n.my.cov()
#output: a list of Wk matrices (K matrices)
list.cov.mat <- lapply(ind.per.group,n.my.cov,descriptors=X)
```

The key of the calculations is here: lapply() launches n.my.cov() to each index independently. We can very well run these tasks in parallel and perform the consolidation at the end of the construction of matrices $\mathbf{W_k}$. We will use this characteristic in the parallel implementations that will be presented later in this document.

[3] In the last step, we build the S matrix by adding the values in the W_k matrices. Then, we divide the resulting values by the degree of freedom.

```
#adding the values from each Wk
#applying the degree of freedom correction
#output: S, the within class covariance matrix
pooled.cov.mat <- (1.0/(length(Y)-nlevels(Y)))*Reduce('+',list.cov.mat)
```

2.2 Function for the calculation of the S matrix

These steps are implemented into the pooled.sequential() function:

```
#X descriptors matrix
#Y class attribute
#sortie: S matrix
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)
}
```

In the remainder of this document, our philosophy will be to implement parallel approaches to calculate list.cov.mat. The strategies differ mainly for the calculation of the source code in red in pooled.sequential() function above. This is the approach that we have followed in the multithreaded implementation of the discriminant analysis in Sipina 3.10. The drawbacks were identified. K threads are asked for the processing, whatever the characteristics of the machine that we used for the calculations, some cores can remain inactive. In addition, the loads are not well distributed when we deal with a dataset with unbalanced classes i.e. when index vectors have very dissimilar size. Some cores can complete their treatments before the others.

2.3 Example of calculations on a dataset

We apply these functions on the WAVE500K data file that we described in the previous tutorial. There are n = 500,000 instances, with K = 3 balanced groups, and p = 21 descriptors. We use the following program in R.

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

```
#Y class attribute, X descriptors
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)
}

#launching the calculation function - computation time measurement
system.time(cov.seq <- pooled.sequential(X,Y))
print(cov.seq)
```

system.time() is used for the calculation of the processing time.

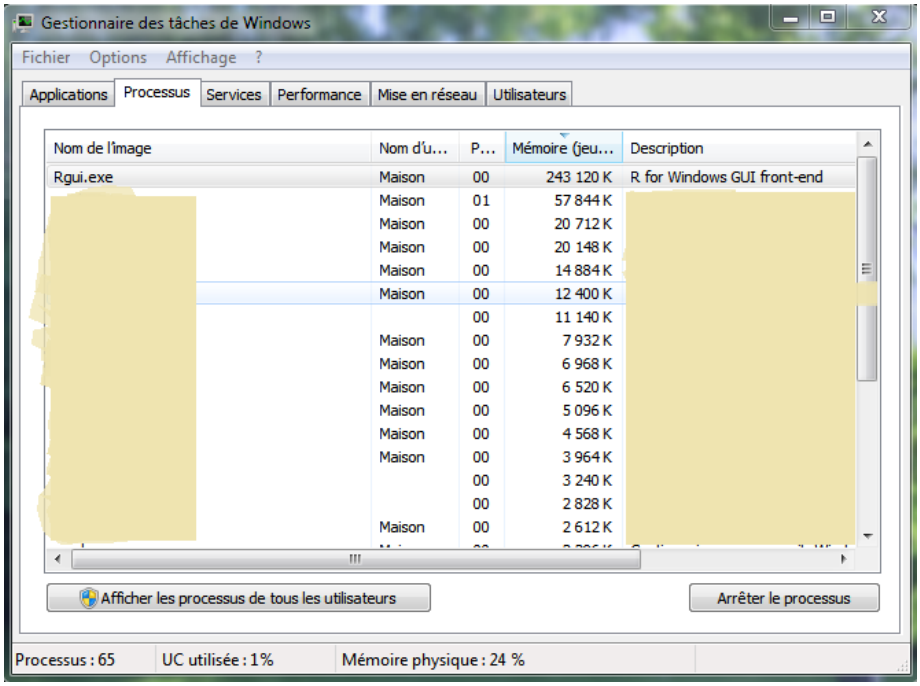
We use R 3.0.0 under Windows 7 (64 bits) on a Quad-Core (Q9400 – 4 cores) processor…

```
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)
```

…we obtain the following values.

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

The memory occupation displayed into the Windows Task Manager is ≈ 237 Mo.



The « temps écoulé » (**16.80 sec.**) corresponds to the waiting time to the achievement of the calculations (“real time” in the R terminology). This is what we had measured with a chronometer moreover. In a single-threaded processing, this value is very close to the user time ("utilisateur")[4]. In the following, we check if the multithreaded strategy can reduce the “real time”.

3 The « parallel » package

3.1 Parallel computing under R

The « parallel »[5] package is build from the « snow » and « multicore » packages. This last one is no longer available under R 3.0.0. The overview paper describes well the ideas that we want to implement in this tutorial. ”This package handles running much larger chunks of computations in parallel. A typical example is to evaluate the same R function on many different sets of data: often simulated data as in bootstrap computations (or with `data' being the random-number stream). The crucial point is that these chunks of computation are unrelated and do not need to communicate in any way. It is often the case that the chunks take approximately the same length of time.

The basic computational model is

(a) Start up M `worker' processes, and do any initialization needed on the workers.
(b) Send any data required for each task to the workers.
(c) **Split the task into M roughly equally-sized chunks**, and send the chunks (including the R code needed) to the workers.
(d) Wait for all the workers to complete their tasks, and ask them for their results.
(e) Repeat steps (b - d) for any further tasks.
(f) Shut down the worker processes.”

[4] http://stat.ethz.ch/R-manual/R-devel/library/base/html/system.time.html

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

We want to operate this framework for the construction of S, with M = K = number of groups described by the target variable Y. Each individual belonging only to a single group, the calculations of W_k can be performed independently.

Each 'worker' is visible as 'rscript' process into Windows Task Manager. In our case, since M = K = 3, we will see 3 copies of rscript, with the same memory occupation because all the data is transferred to them. This is the main drawback of the solution. Indeed, by duplicating data, it increases the memory occupation. Their transfer to the rscript during the initialization processes takes also time. In fact, this step will degrade the whole performance of the system as we will see later.

3.2 Detecting the number of CPU cores

The **detectCores()** procedure detects the number of CPU cores available. For our computer, which is a Quad-Core, we obtain 4 cores of course.

```
#loading the package
library(parallel)
#printing the number of available cores
print(detectCores())
```

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

3.3 The mclapply() procedure

The **mclapply()** is a parallel version of **lapply().** We can specify the number of cores to use. Thus, only one line is modified in our function for the calculation of the S matrix.

```
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)
}
```

The « mc.cores » option enables to set the number of cores to use. We set mc.cores = 1 in a first time. When we launch the procedure, the execution time is very close to the single-threaded method. The calculations of the Wk are launched sequentially.

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

Alas, when we wanted to increase the number of cores to use (mc.cores = 3), the procedure has failed. The command relies on a technology that is not operational on Windows[6]. This is a pity because that method was very easy to implement. It seems however that the function is fully operational under Linux.

```
> 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
```

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

3.4 The parLapply() procedure

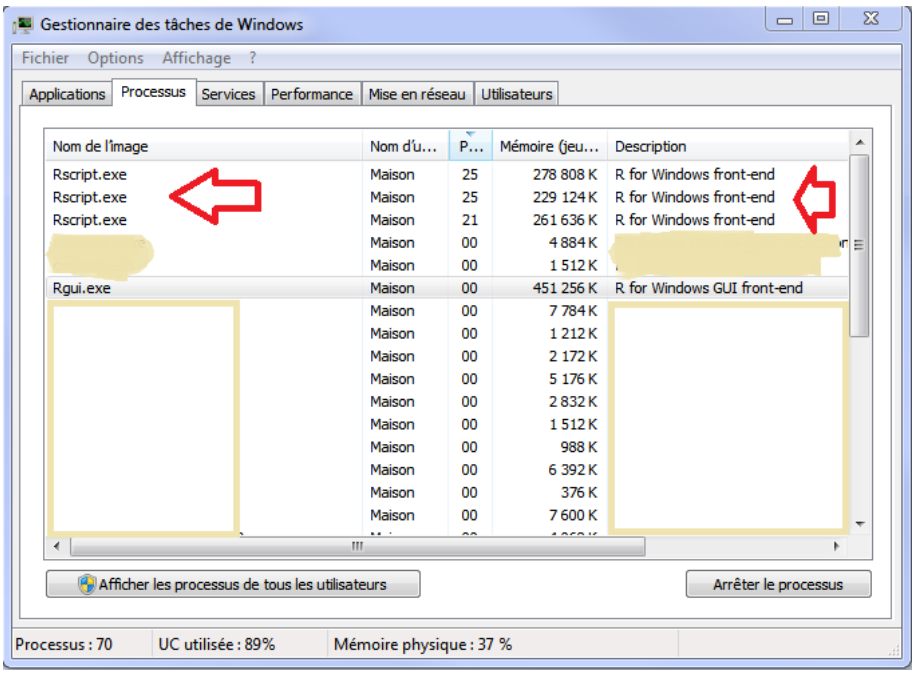
The **parLapply()** procedure is also a variant of **lapply()**. It needs to additional steps: we must initialize explicitly the workers to use with the **makeCluster()**[7] command; we must destroy them at the end of the calculations with the **stopCluster()** command. The function for the calculation of S becomes:

```
pooled.parLapply <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  #creation of K = 3 rscript
  #which appear in the Windows task Manager
  cl <- makeCluster(spec=nlevels(Y))
  #calling 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 of rscript
  stopCluster(cl)
  return(pooled.cov.mat)
}
```

We obtain the following output:

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

The improvement of the calculation time is disappointing. The real time is 15.12 sec. It is worse than the singe-threaded version. The user time (1.75 sec.) and the system time (1.17 sec.) correspond to the activity of the main thread. They are not assigned to the calculations of the W_k matrices.



[7] We create the 'workers' on the same system here to use the existing cores. But makeCluster() can go further in accessing resources from a remote machine. R must be installed on these machines.

By observing the Windows Task Manager, we note that the calculations are well distributed on 3 different “rscript” processes. We also find that the data are completely duplicated on each process by considering the memory occupation. Despite the splitting of the calculations on the cores, we do not obtain an overall improvement in the processing time. Why?

We have an idea on the response by using a more sophisticated tool for the processing time measurement. We use the **snow.time()**[8] from the "snow" package.

```
library(snow)
#a better decomposition of the processing time
snow.time(cov.par <- pooled.parLapply(X,Y))
```

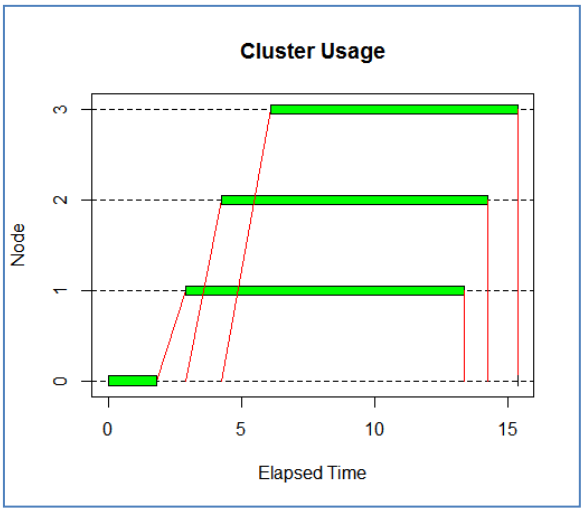
We understand better the behavior of the system now.

```
> 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
```

The processing time on each worker (node) is 9 seconds approximately. We can improve the whole process by dispatching the calculations on cores. But this improvement is completely annihilated by the duration time of data transfer (send = 4.33 sec.). The recovery of the matrices is inexpensive (receive = 0.02 sec.). The tool offers a comprehensive presentation of the processes in the form of Gantt chart.

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

The data transfer from the main thread to the nodes is made sequentially. Thus, the last node (3rd node) is started with a big lag. We must await the end of its calculations before retrieving the results.



Clearly, the parallelization is advantageous only if the duration of node processing is significantly higher compared to the duration time of the dataset transfer to the nodes.

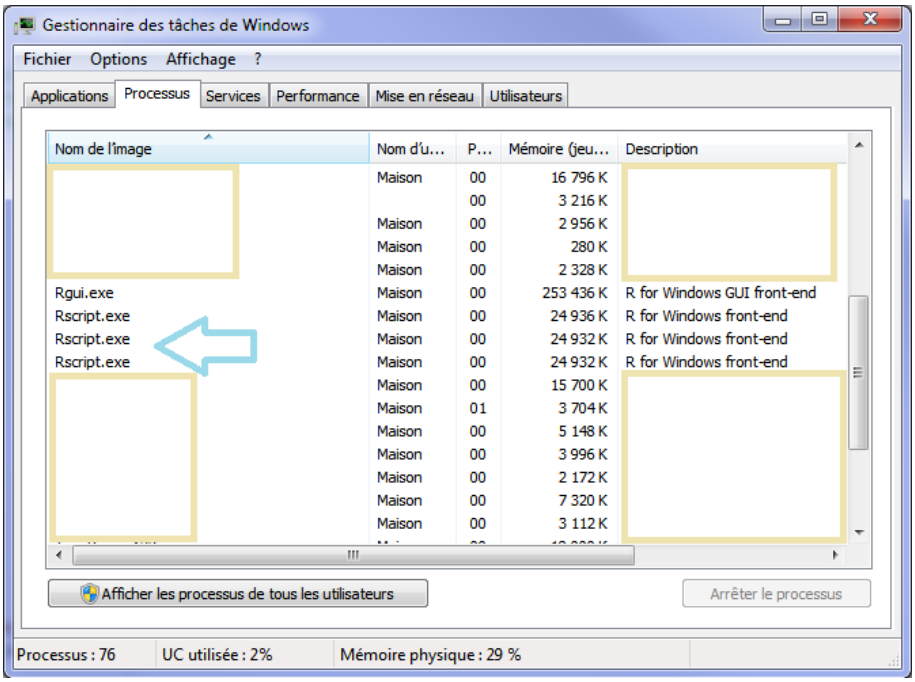
[8] http://cran.r-project.org/web/packages/snow/snow.pdf

3.5 Using the « foreach » … « %dopar% » loop

The "doParallel" supplies a foreach procedure for loop mechanism. It enables to launch tasks in parallel. We must before start the workers with the **registerDoParallel()** command.

```
library(doParallel)
#creation of 3 rscript
registerDoParallel(3)
```

The workers (rscript) are available in Windows Task Manager. They are waiting: they are not started (Processeur [CPU] = 0%), and the dataset is not transferred (Memoire [Memory] ≈ 24 MB).

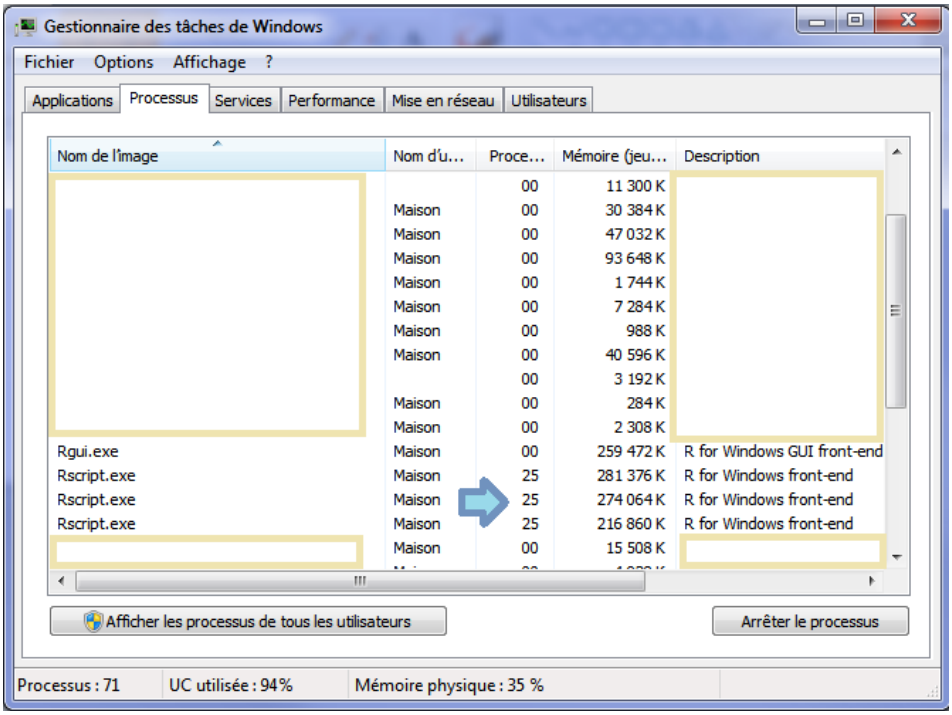


The calculations are launched by a “foreach...%dopar%” mechanism i.e. for each instances from the list ind.per.group, we process the instructions comprised between ‘**{**‘ and ‘**}**’.

```
pooled.foreach <- function(X,Y){
  ind.per.group <- lapply(levels(Y),partition,y=Y)
  #loop 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)
}
```

The option « .export » enables to use the non-standard function “n.my.cov” into the environment; “.inorder=FALSE” stands that we can retrieve the results (the W_k matrices) in any order.

When we start the process, we see that the rscript workers are activated in the Windows Task Manager. We observe that, according the memory occupation reported here, the whole dataset is duplicated like for the parLapply approach.



By using the ".verbose=T" option, we obtain the details of the process. The "tasks" are launched in parallel. The results (the W_k matrices) are combined in a list. Here also, the processing time (14.18 sec.) is equivalent to the single-threaded approach. The data transfer takes an important part in the process (send = 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
```

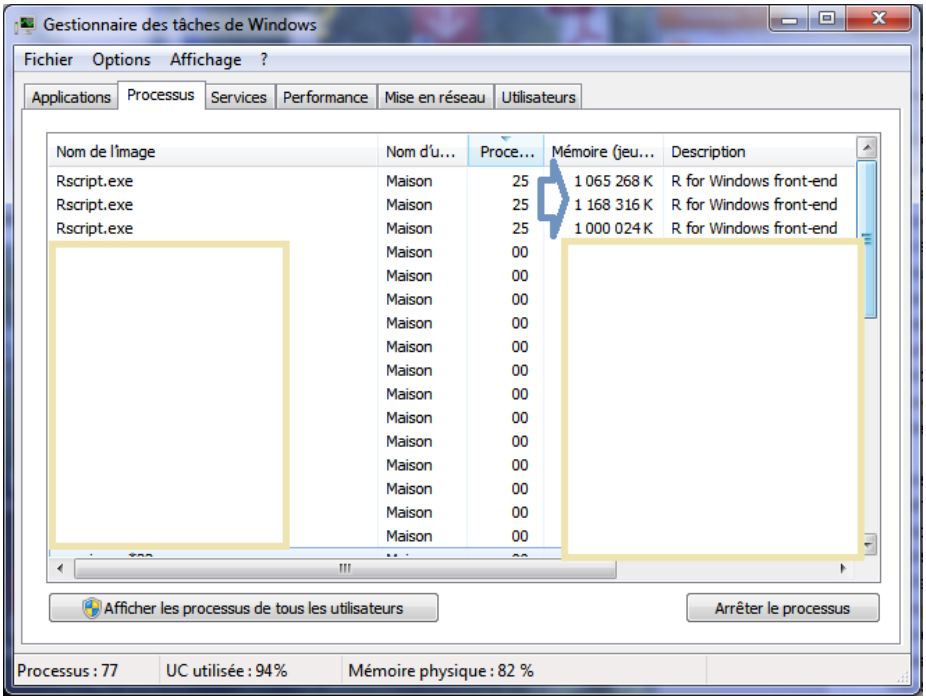
There is no real improvement compared to the parLapply approach. I would say that the “foreach...%dopar%” loop is simply an alternative to parallelization of tasks.

4 Conclusion

In this tutorial, we have presented different solutions from the "parallel" package under R to parallelize calculations. The goal is to maximize the utilization of the multicore processors capabilities. The system works well. This is a very important result. However, the processing time is

disappointing on the dataset that we studied, mainly because the need to duplicate and transfer the data to the calculation engines. The memory occupation becomes also a problem in this context.

For example, for WAVE2M dataset with n = 2,000,000 observations and p = 21 variables, the execution time is 59 seconds in sequential mode. It is 56 sec. in parallel mode: with 40 seconds for the processing on each rscript and 15 seconds for the data transfer. Each rscript occupies almost 1 GB in memory. This is really a bottleneck to the use of this solution for the mining large dataset.



Nevertheless, the "parallel" package tools allow to go further than the utilization of the multicore processors capabilities. It is possible to distribute the calculations on remote machines. This opens up other perspectives. The memory occupation is no longer a problem in this context because the dataset is duplicated on the remote machines. There is material to do interesting things in one of our upcoming tutorials...